Karlsruhe Institute of Technology

Algorithmen II

Peter Sanders

Exercise:

Daniel Seemaier, Tobias Heuer

Institute of Theoretical Informatics

Web:

`http://algo2.iti.kit.edu/AlgorithmenII_WS20.php`

Karlsruhe Institute of Technology

4 Applications of DFS

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

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s                                  // make s a root and grow
        root(s)                                 // a new DFS-tree rooted at it.
        DFS(s, s)
Procedure DFS(u, v : NodeId)                    // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v)      // return from v along the incoming edge
```

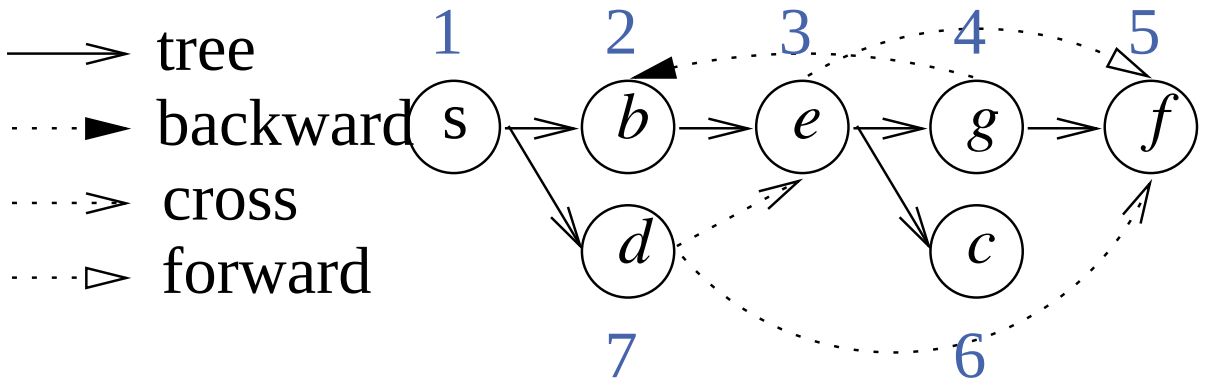
DFS Ordering

init:	dfsPos$=1$: $1..n$
root(s):	dfsNum$[s]$:= dfsPos$++$
traverseTreeEdge(v,w):	dfsNum$[w]$:= dfsPos$++$

$$u \prec v \Leftrightarrow \text{dfsNum}[u] < \text{dfsNum}[v] \ .$$

Observation:

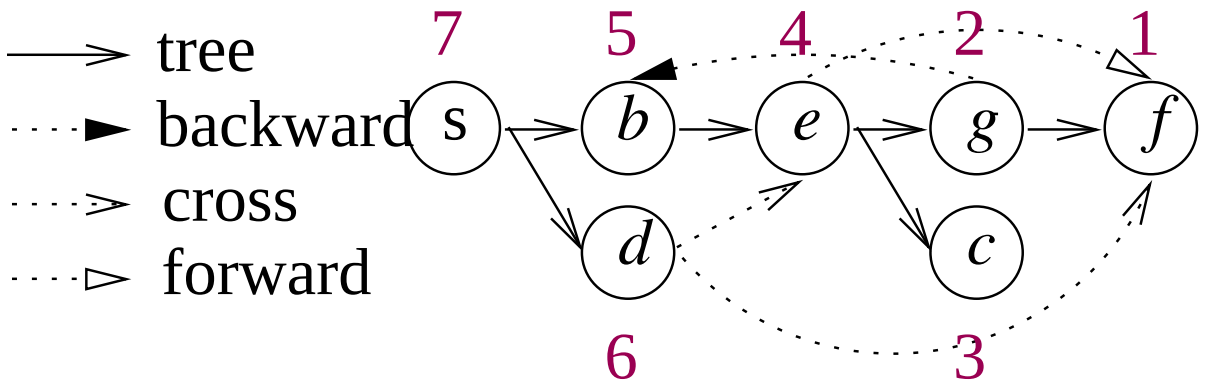
Nodes on the recursion stack are sorted w.r.t. $\prec$



Finishing Time

init: finishingTime$=1$: $1..n$

backtrack(u,v): finishTime$[v]$:= finishingTime$++$



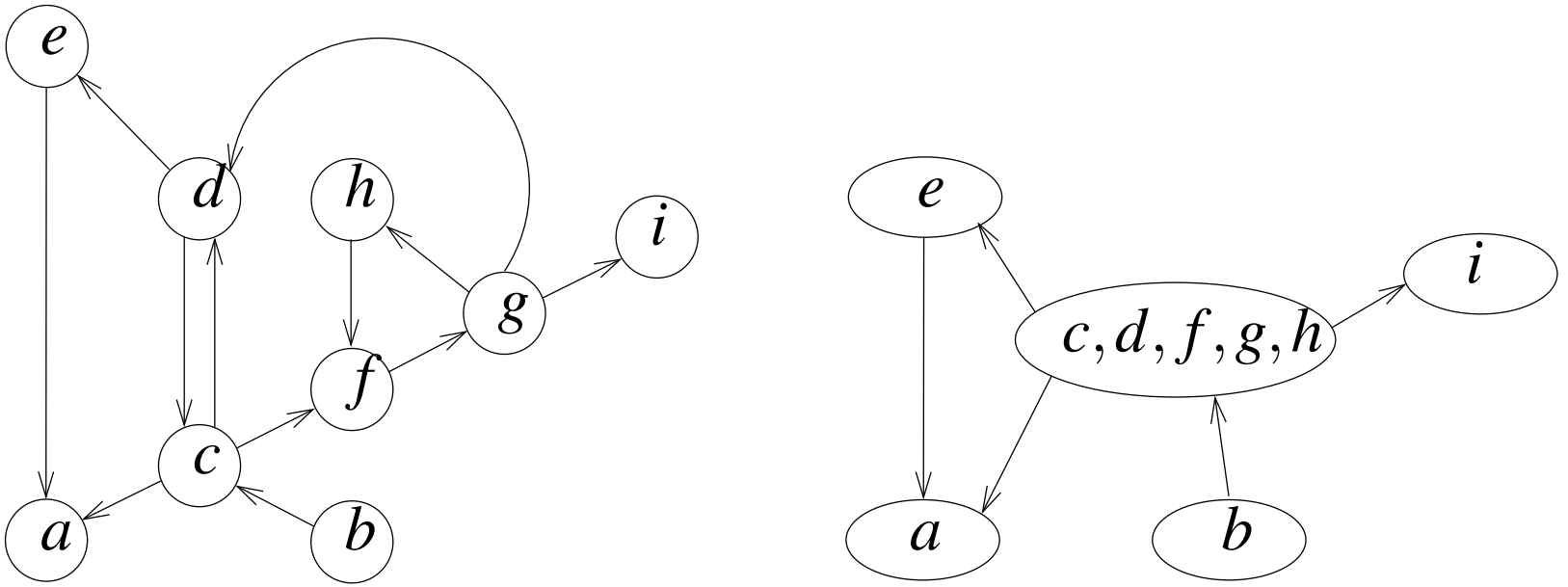
Strongly Connected Components

Consider the relation $\overset{*}{\leftrightarrow}$ where

$u \overset{*}{\leftrightarrow} v$ if $\exists$ path $\langle u, \ldots, v \rangle$ and $\exists$ path $\langle v, \ldots, u \rangle$.

Observation: $\overset{*}{\leftrightarrow}$ is an equivalence relation Exercise

The equivalence classes of $\overset{*}{\leftrightarrow}$ are called strongly connected components.



Strongly Connected Components – Abstract Algorithm

$G_c := (V, \emptyset = E_c)$

foreach edge $e \in E$ **do**

invariant SCCs of G_c are known

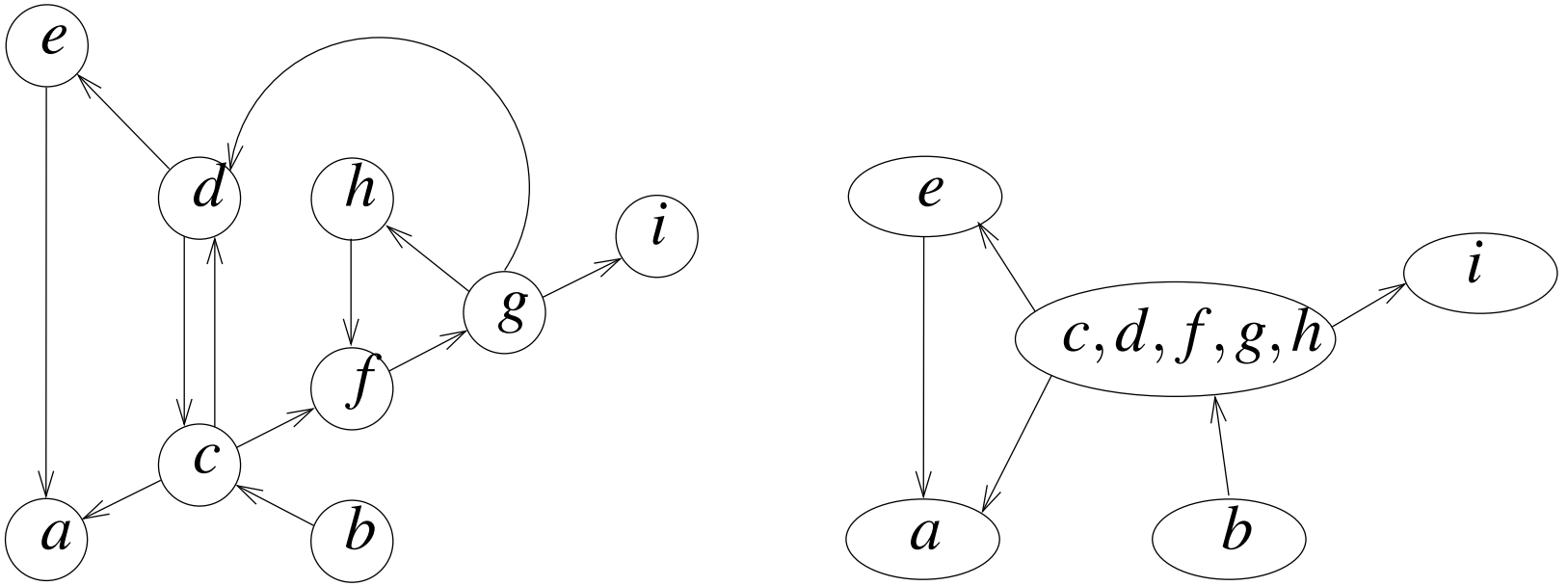
$E_c := E_c \cup \{e\}$

Shrunken Graph

$G_c^s = (V^s, E_c^s)$

Nodes: SCCs of G_c.

Edges: $(C,D) \in E_c^s \Leftrightarrow \exists (c,d) \in E_c : c \in C \wedge d \in D$



Observation: the shrunken graph is acyclic

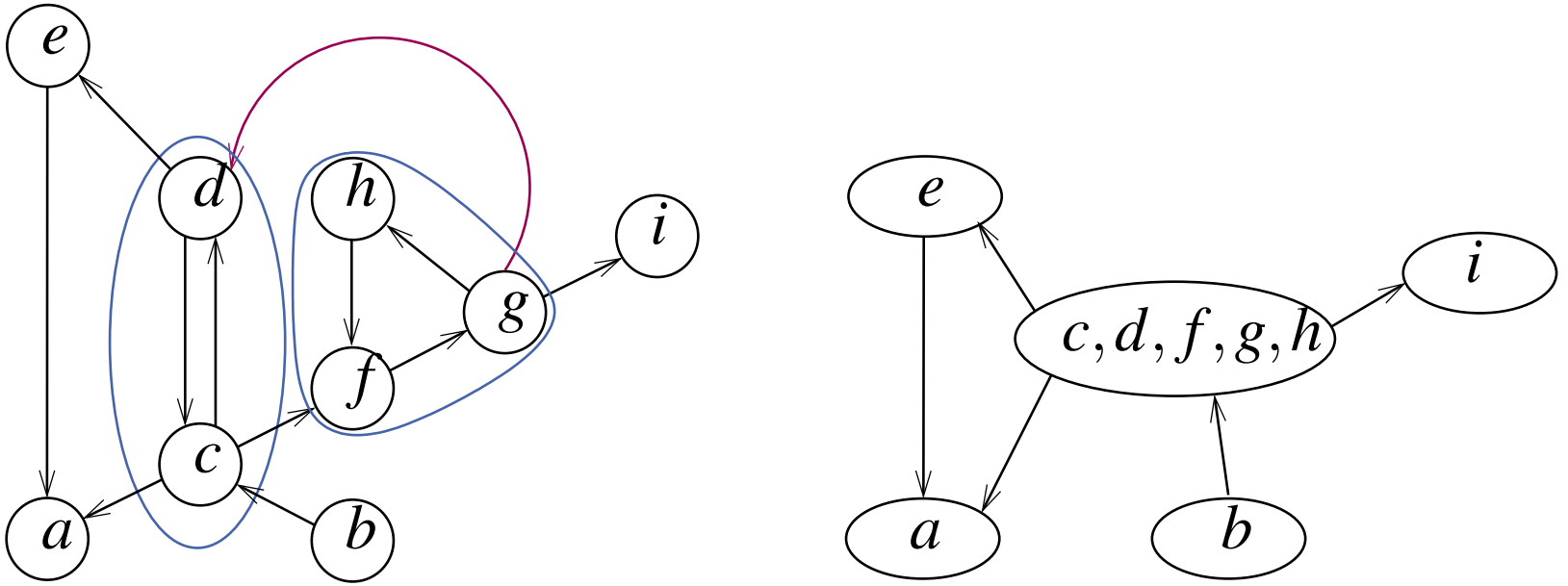
Effects of a New Edge e on G_c, G_c^s

internal to an SCC: Nothing changes

between two SCCs:

no cycle: new edge in G_c^s

closing a cycle: SCCs on the cycle collapse.



More Concretely: Finding SCCs with DFS

[Cheriyan/Mehlhorn 96, Gabow 2000]

$V_c =$ marked nodes

$E_c =$ edges explored so far

Active nodes: marked but not yet finished.

SCCs of G_c:

not reached: unmarked nodes

open: contains active nodes

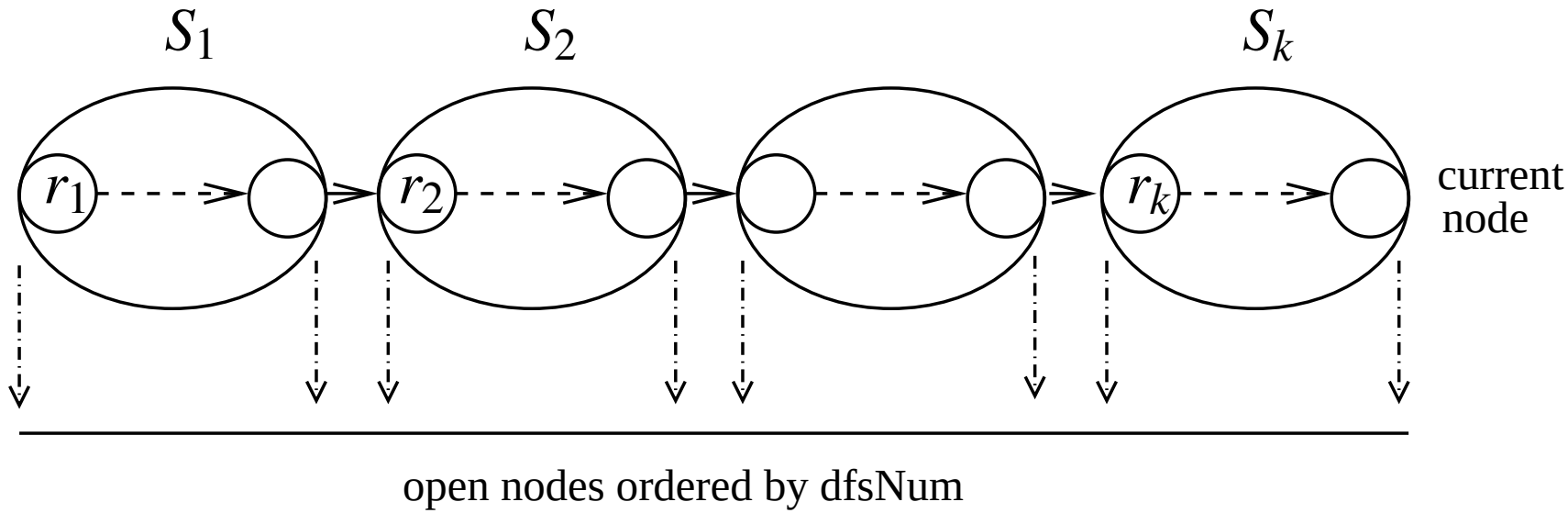
closed: all nodes finished

component[w] is the representative of an SCC.

Nodes of open (closed) components are called open (closed)

Invariants of G_c

1. Edges from closed nodes lead to closed nodes
2. Open components $S_1,\ldots,S_k$ form a path in G_c^s.
3. Representatives partition the open components w.r.t. their dfsNum.



Lemma: Finished SCCs of G_c are SCCs of G

Consider a closed node v

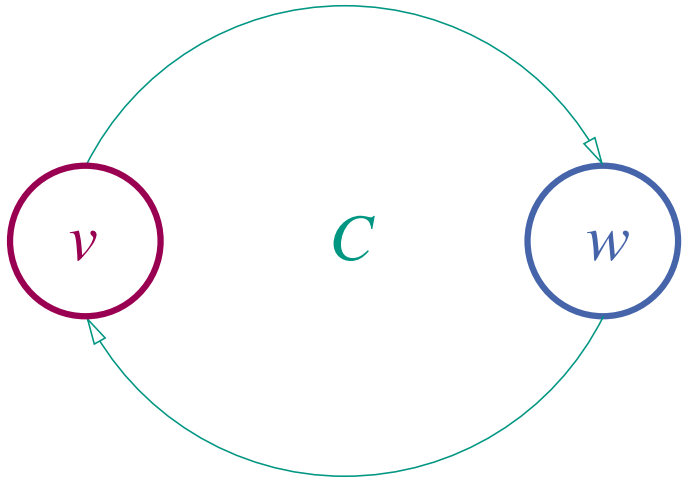
and an arbitrary node w

in the SCC of v w.r.t. G.

To prove: w is closed and

in the same SCC of G_c as v.

Consider cycle C containing v, w.



Inv. 1: nodes of C are closed.

Closed nodes are finished.

Edges out of finished nodes have been explored.

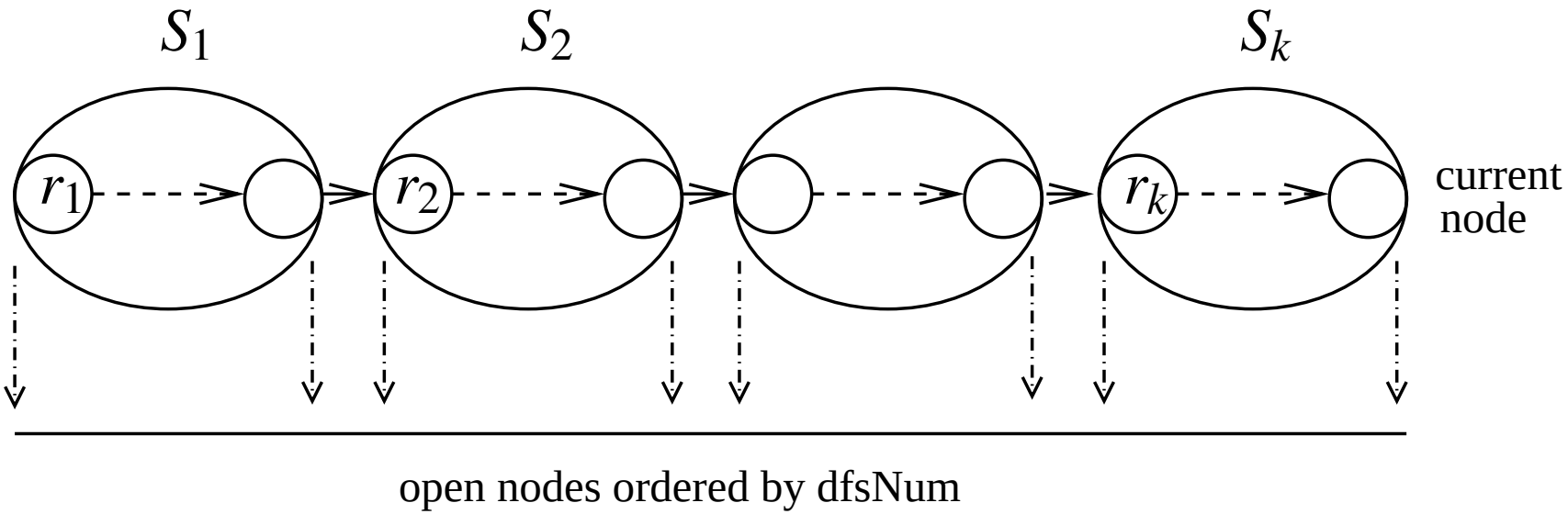
Hence, all edges of C are in G_c. □

Representation of Open Components

Two stacks ordered by dfsNum ascendingly

oReps: representatives of open components

oNodes: all open nodes



init

component : NodeArray **of** NodeId // SCC representatives

oReps=⟨⟩ : Stack **of** NodeId // representatives of open SCCs

oNodes=⟨⟩ : Stack **of** NodeId // all nodes in open SCCs

All invariants are satisified.

(Neither open nor closed nodes)

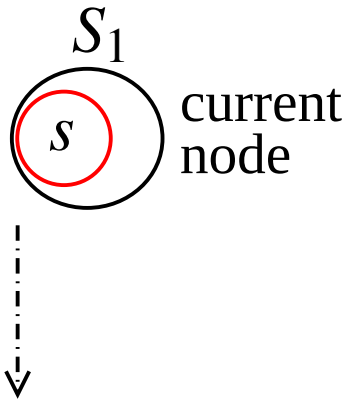
root(s)

oReps.push(s) // new open

oNodes.push(s) // component

$\{s\}$ is the only open component.

All invariants remain valid



open nodes ordered by dfsNum

traverseTreeEdge(v, w)

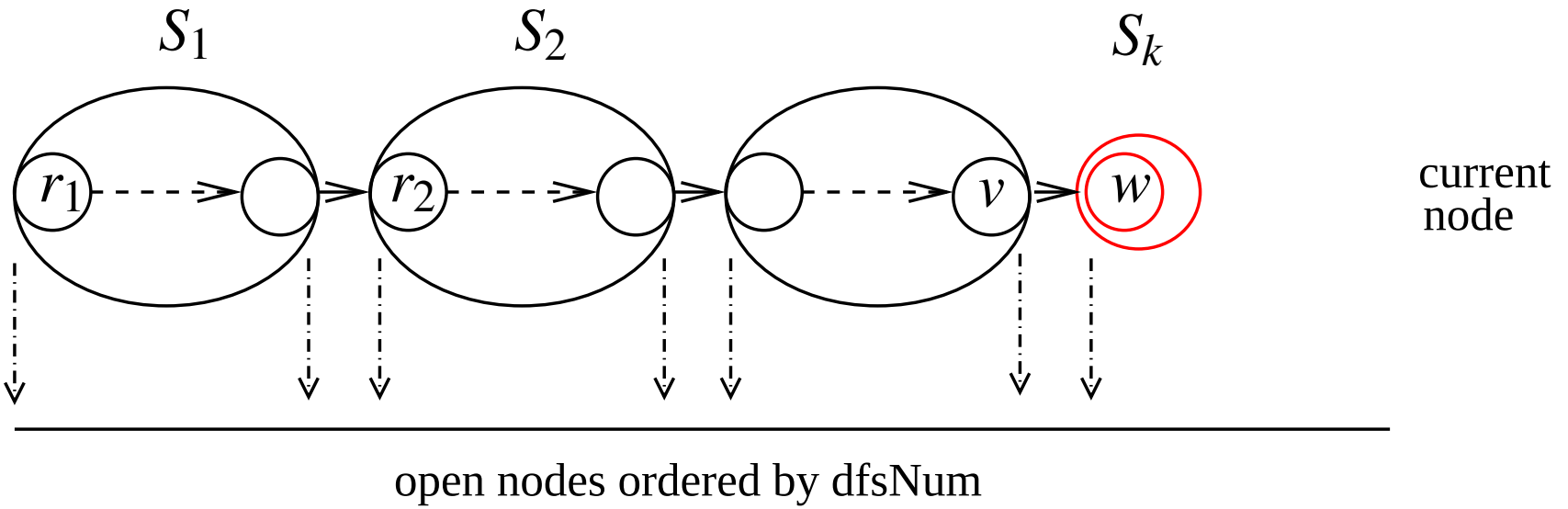
oReps.push(w) // new open

oNodes.push(w) // component

$\{w\}$ is a new open component.

dfsNum(w) $>$ all others.

$\rightsquigarrow$ All invariants remain valid



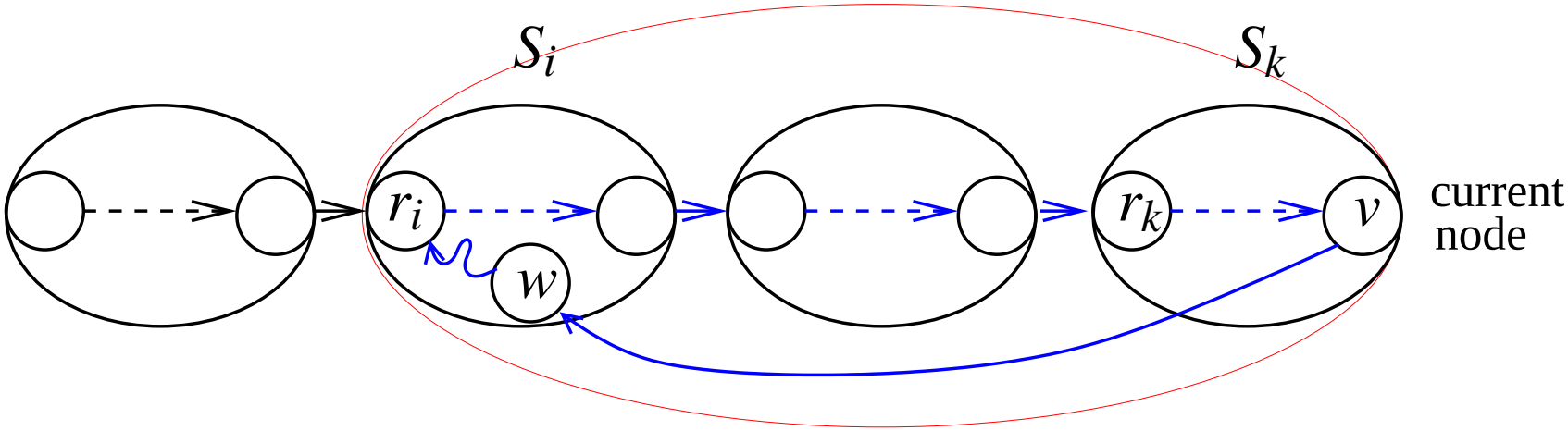
traverseNonTreeEdge(v, w)

if $w \in$ oNodes **then**

while $w \prec$ oReps.top **do** oReps.pop

$w \notin$ oNodes $\rightsquigarrow$ w is closed $\overset{Lemma(*)}{\rightsquigarrow}$ edge is not interesting

$w \in$ oNodes: collapse open SCCs on the cycle



backtrack(u, v)

if $v =$ oReps.top **then**

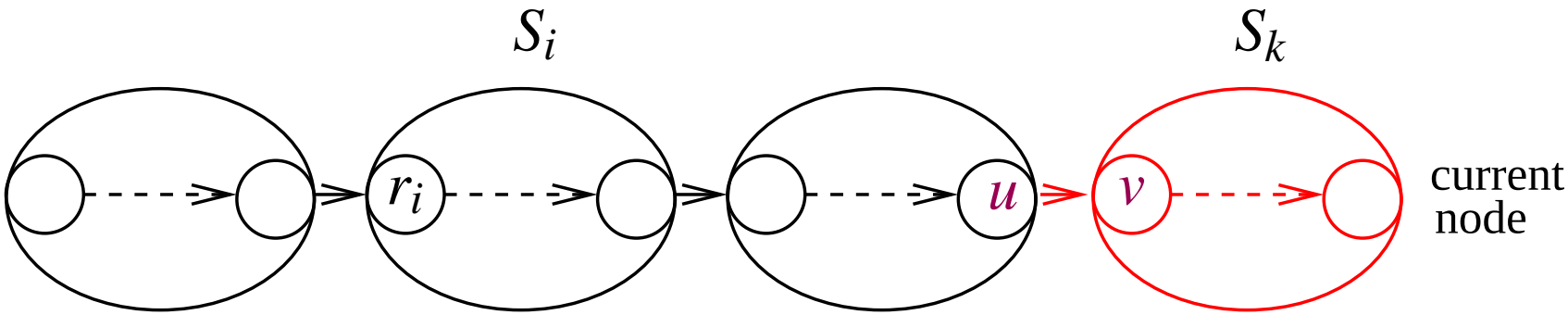
  oReps.pop // close

  repeat // component

    w:= oNodes.pop

    component$[w]$:= v

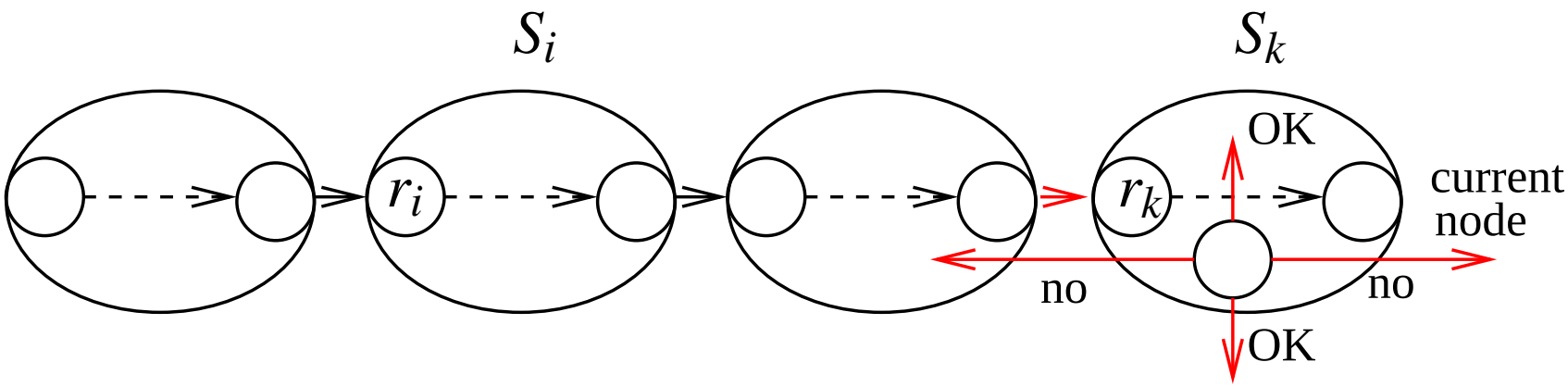
  until $w = v$



To prove: invariants remain valid...

```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

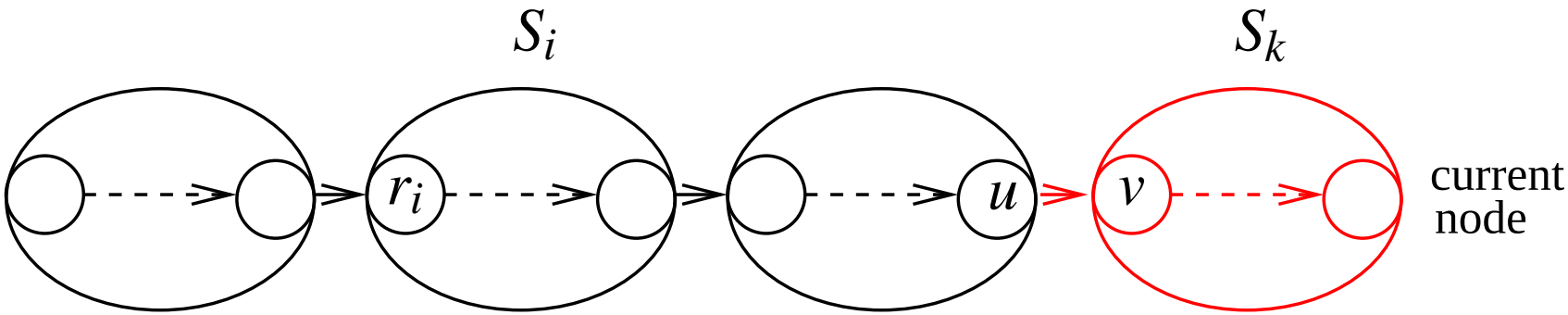
Inv. 1: edges from closed nodes lead to closed nodes.



```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

Inv. 2: open components $S_1,\ldots,S_k$ form a path in G_c^s

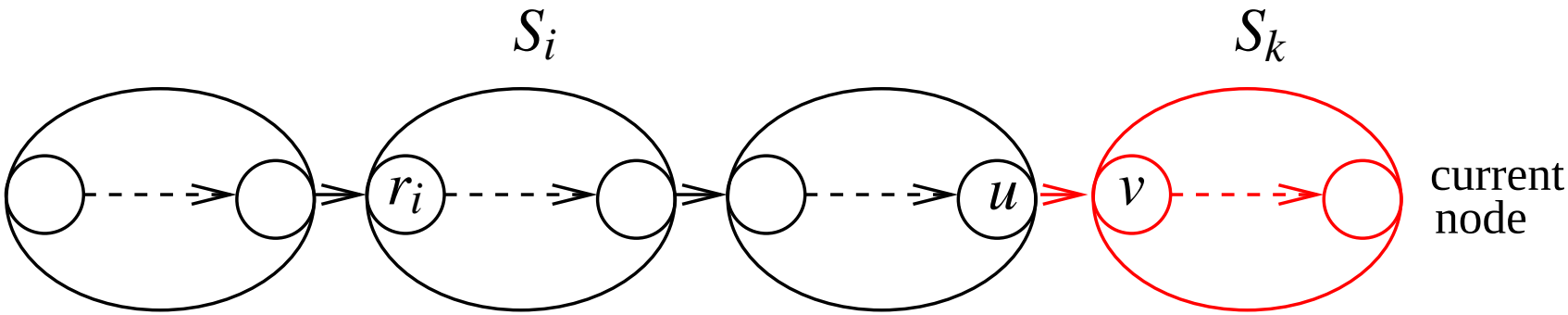
OK. (S_k may be removed)



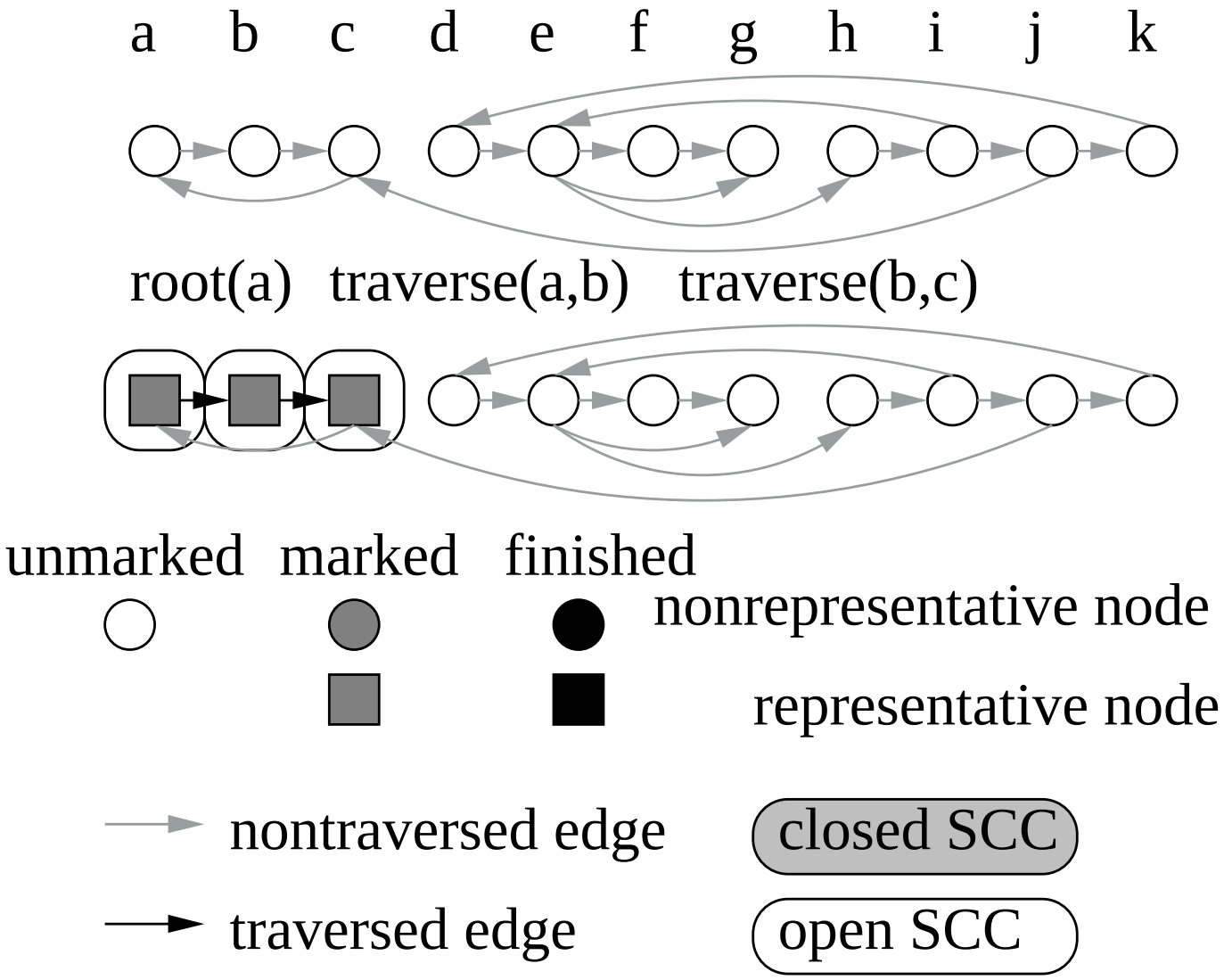
```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

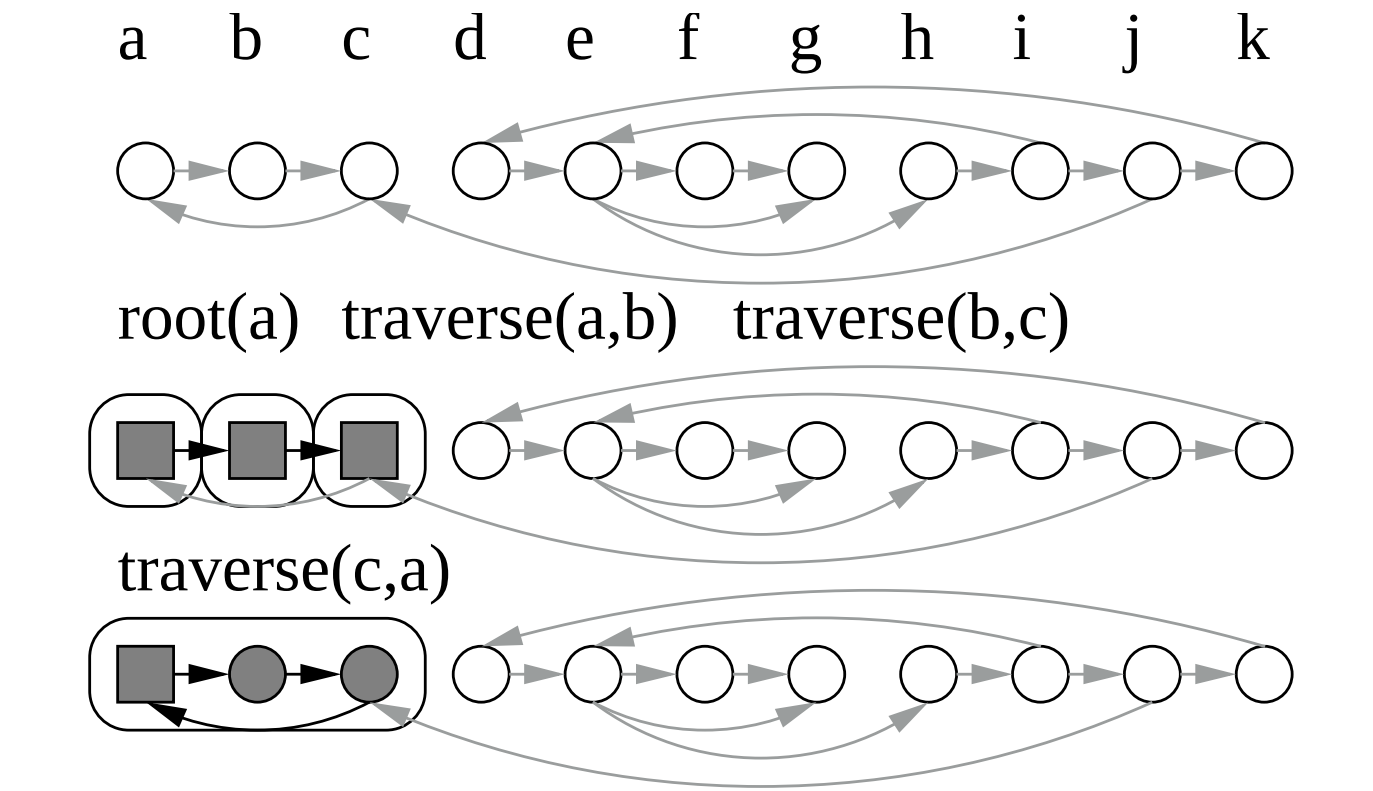
Inv. 3: representatives partition the open components w.r.t. their dfsNum.

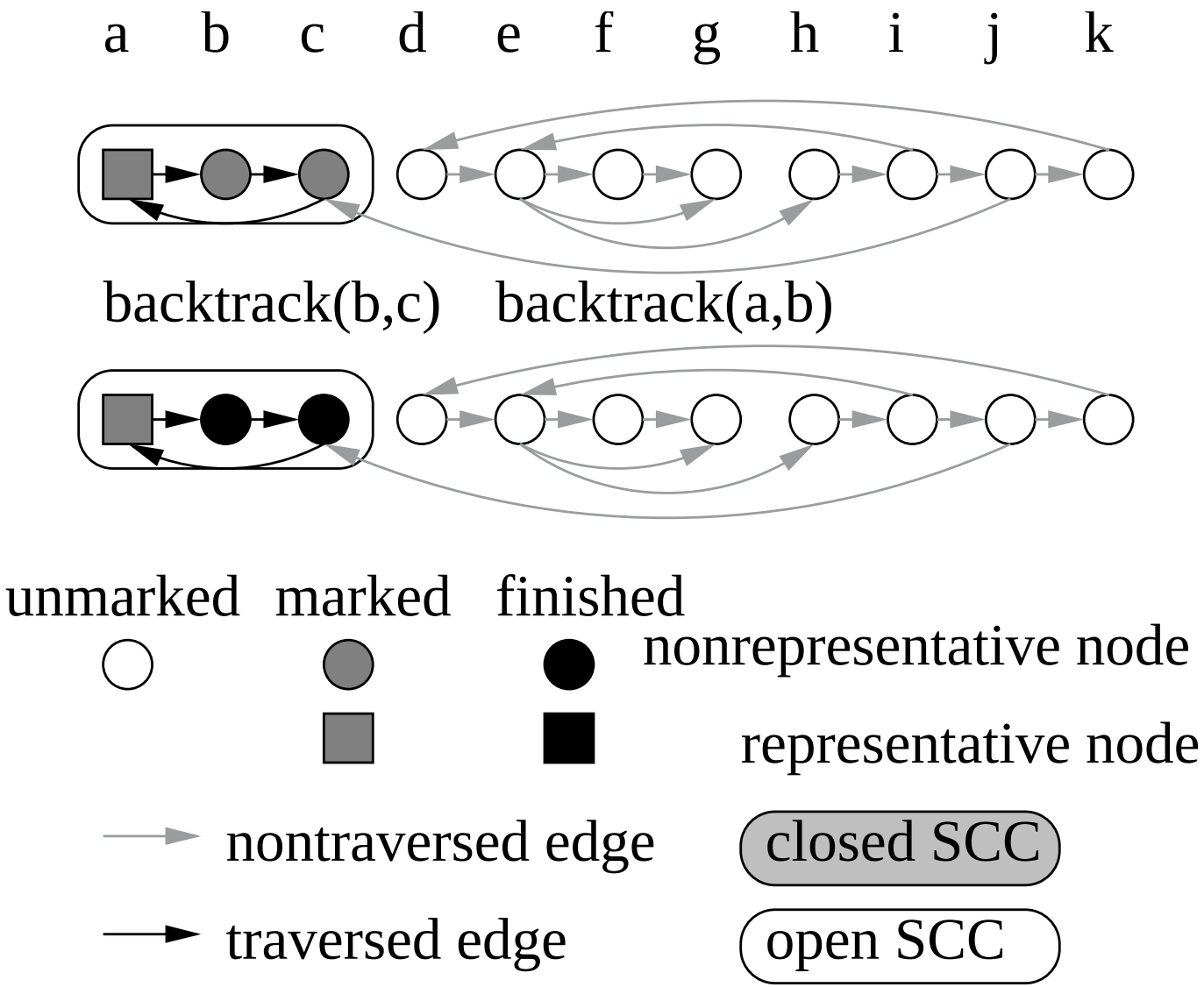
OK. (S_k may be removed)

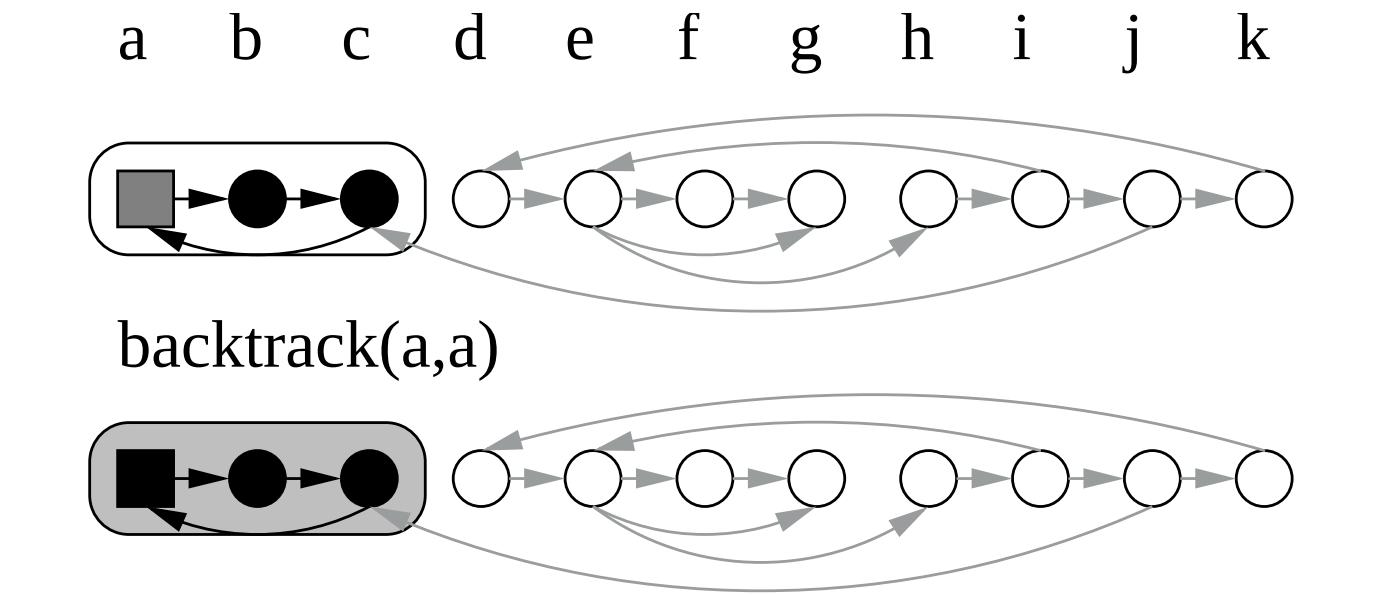


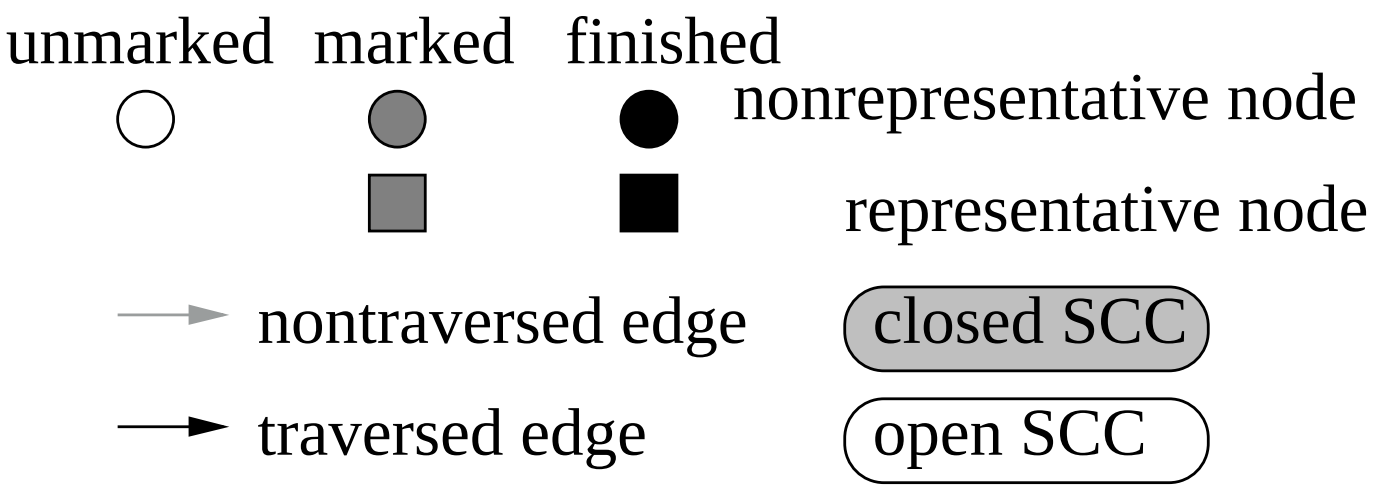
Example

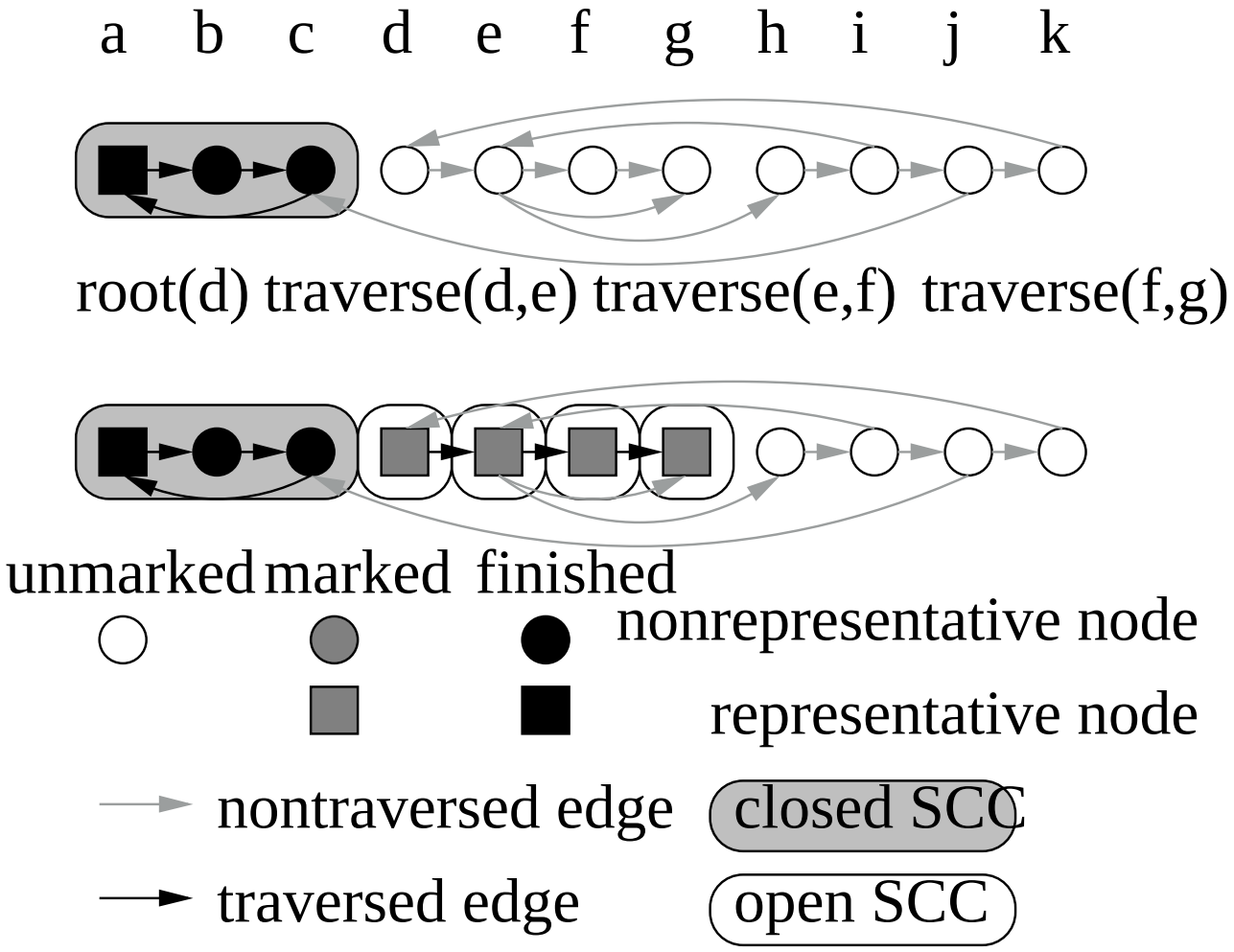

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

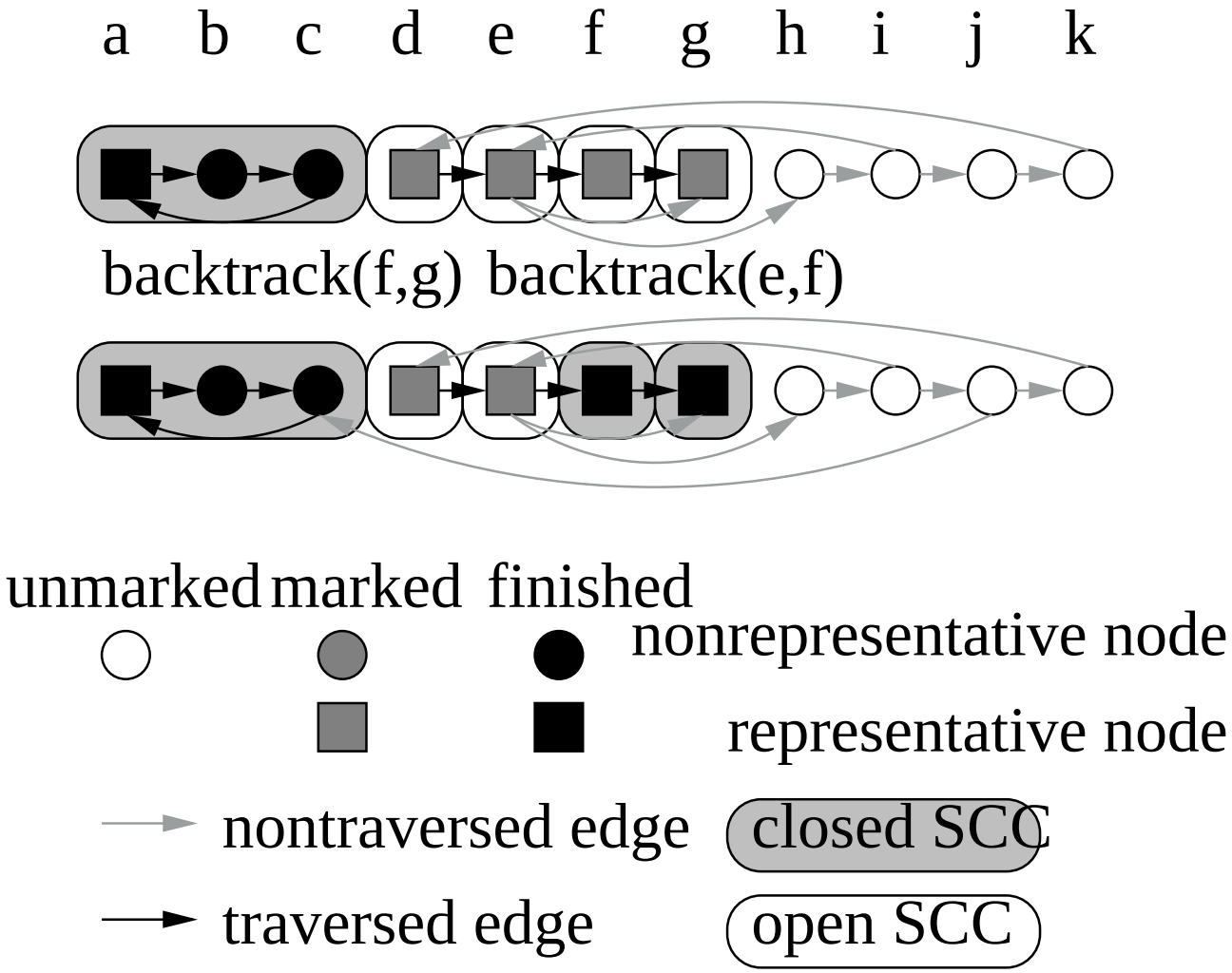

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
traverse(c,a)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

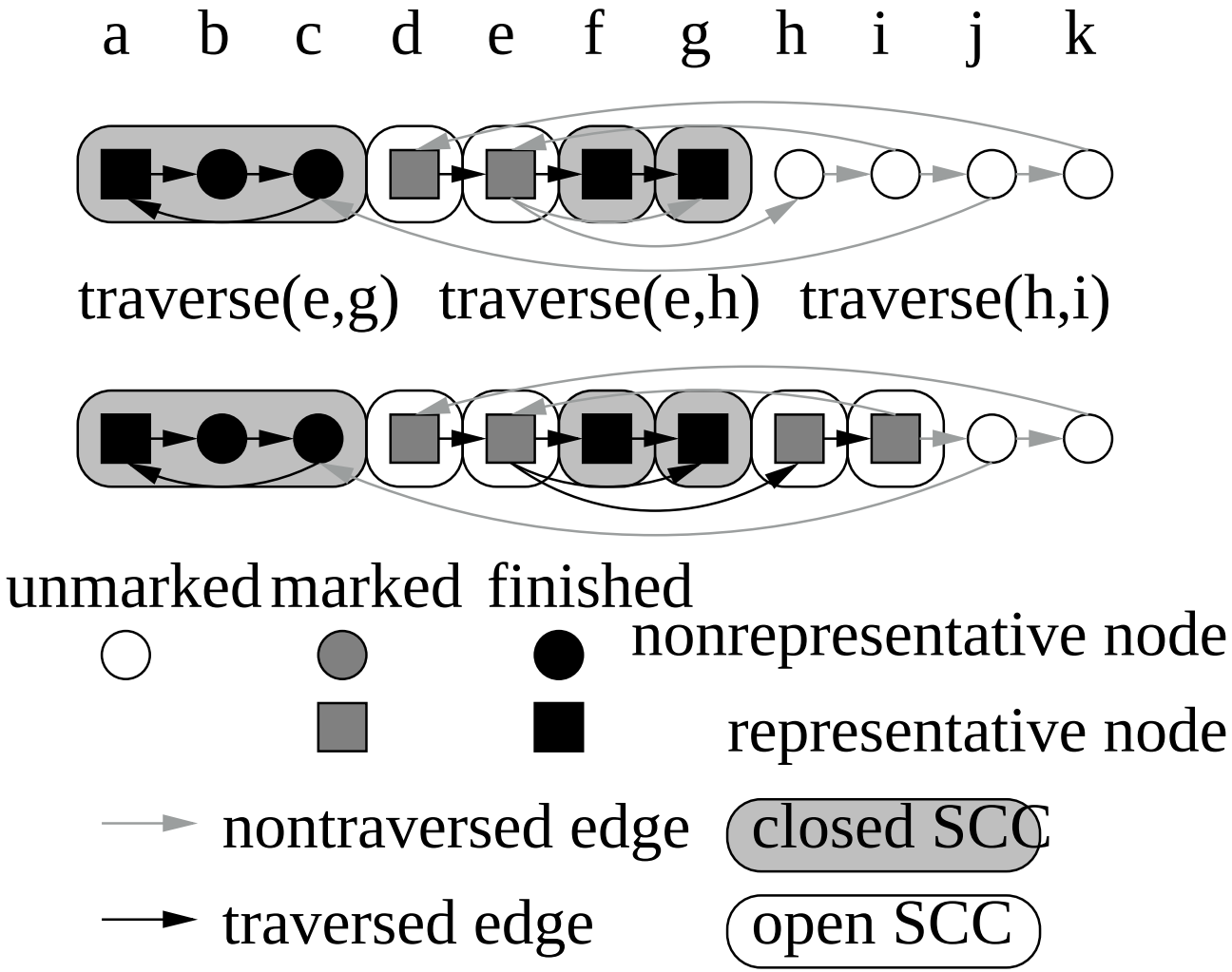

a b c d e f g h i j k
backtrack(b,c)
backtrack(a,b)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

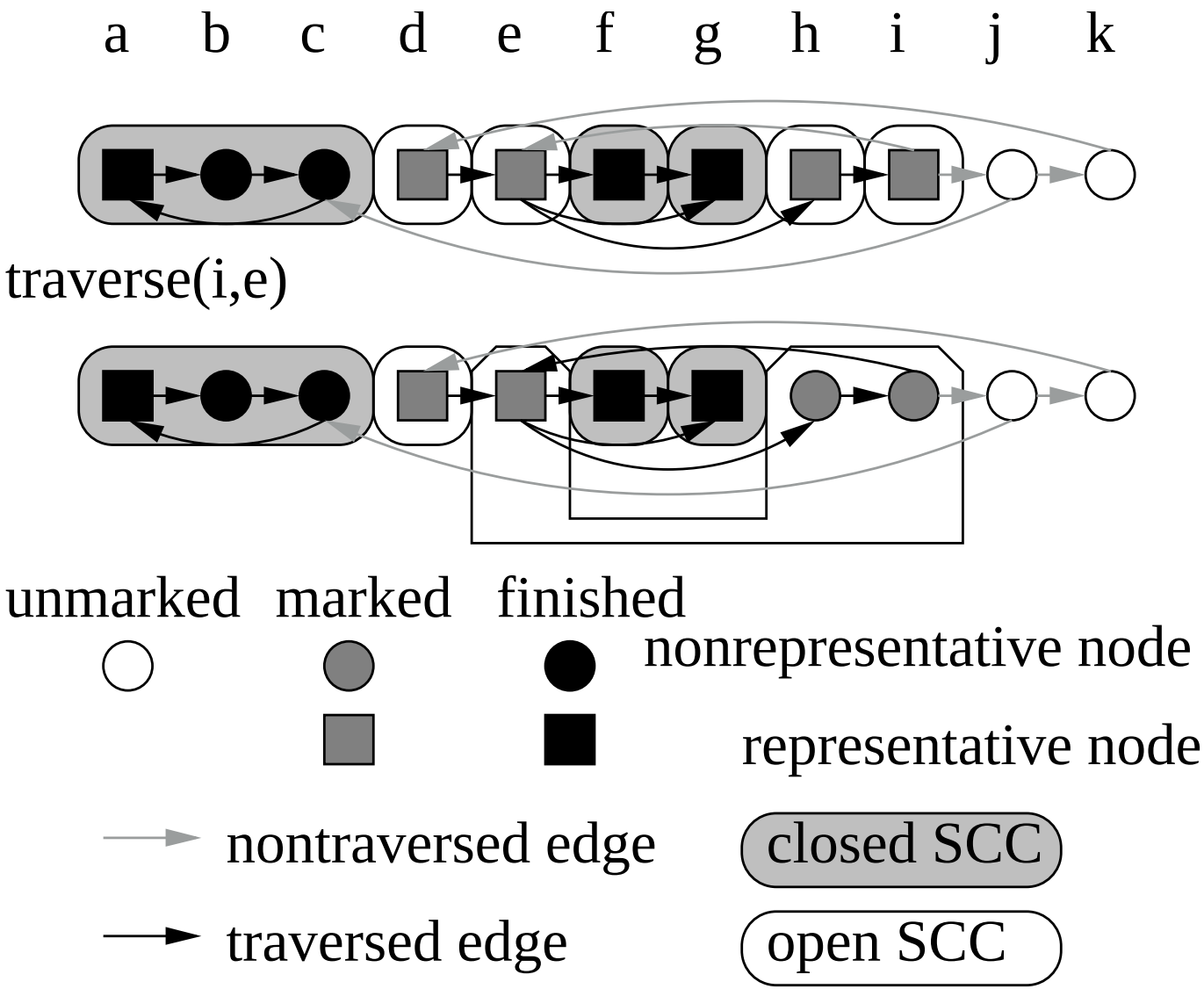

a b c d e f g h i j k
backtrack(a,a)

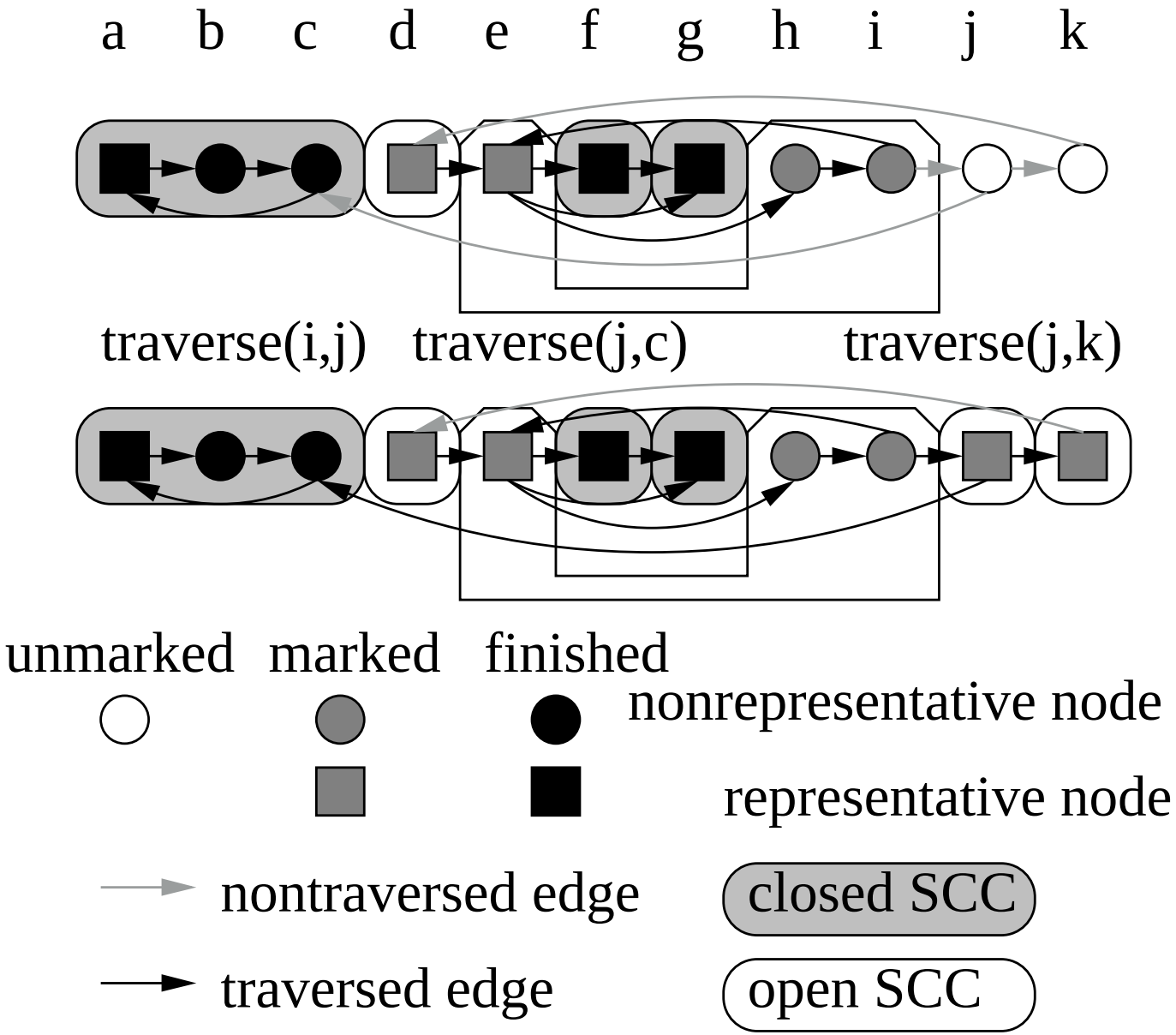

unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

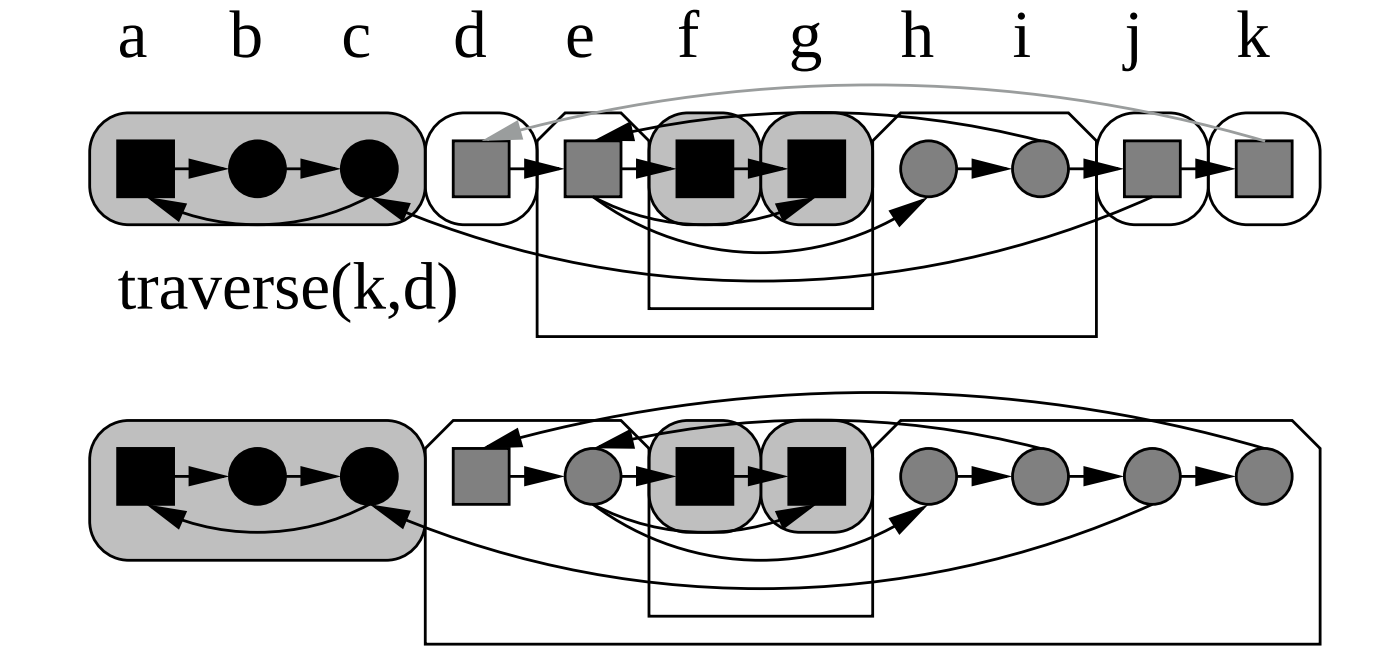

a b c d e f g h i j k
root(d) traverse(d,e) traverse(e,f) traverse(f,g)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

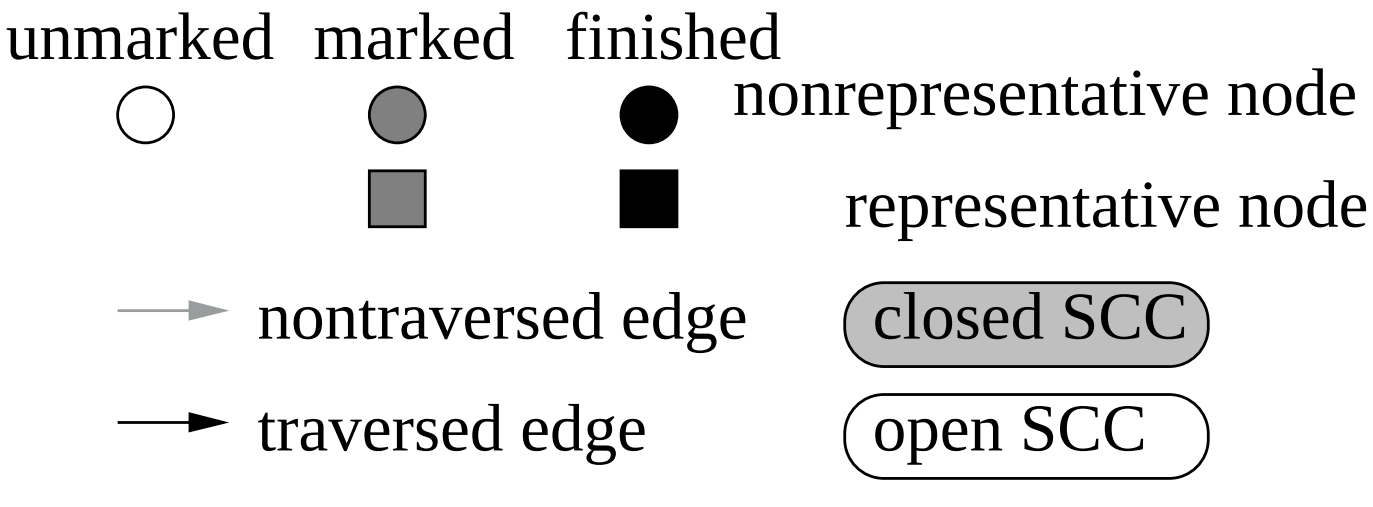

a b c d e f g h i j k
backtrack(f,g)
backtrack(e,f)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

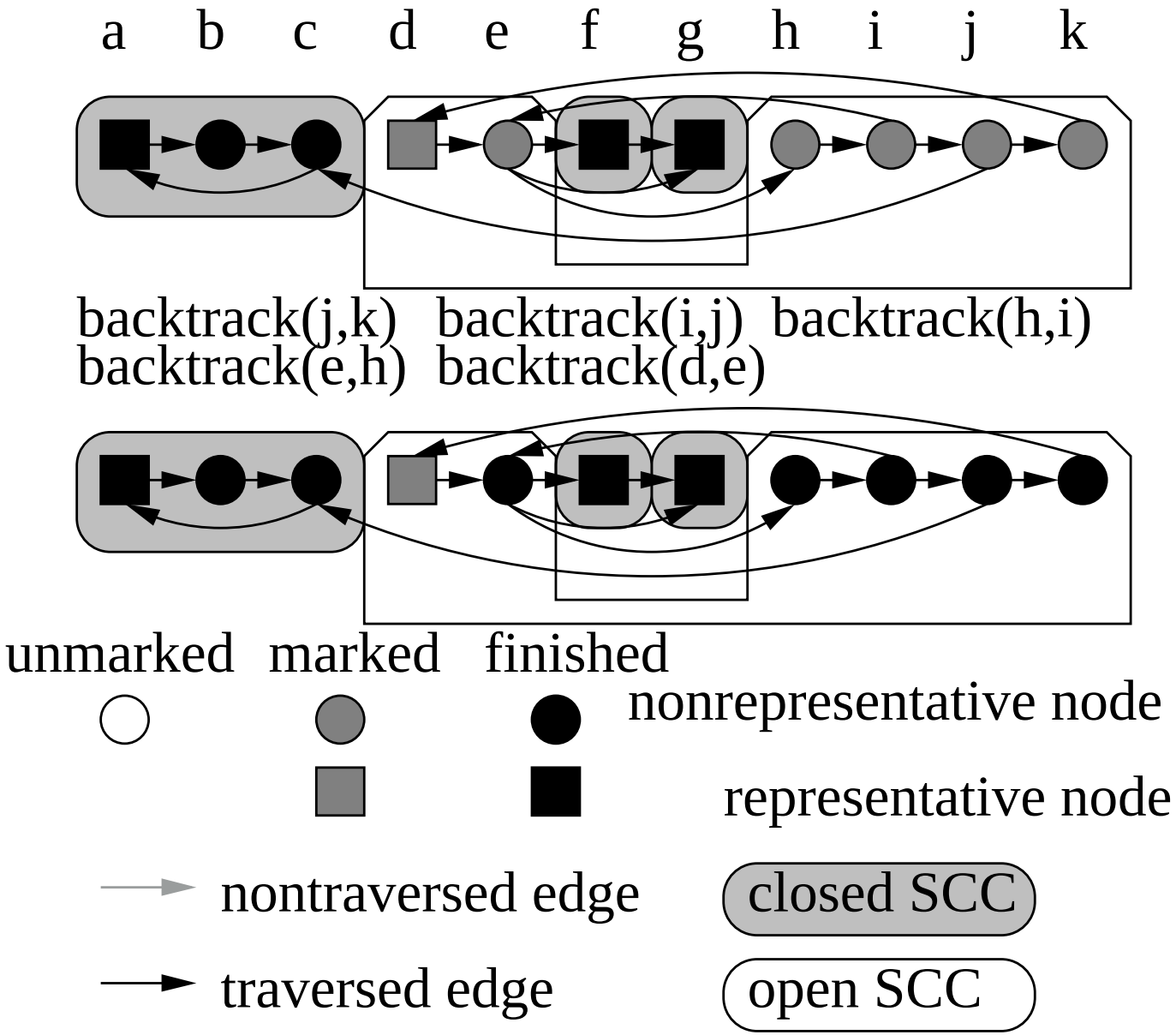

a b c d e f g h i j k
traverse(e,g) traverse(e,h) traverse(h,i)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

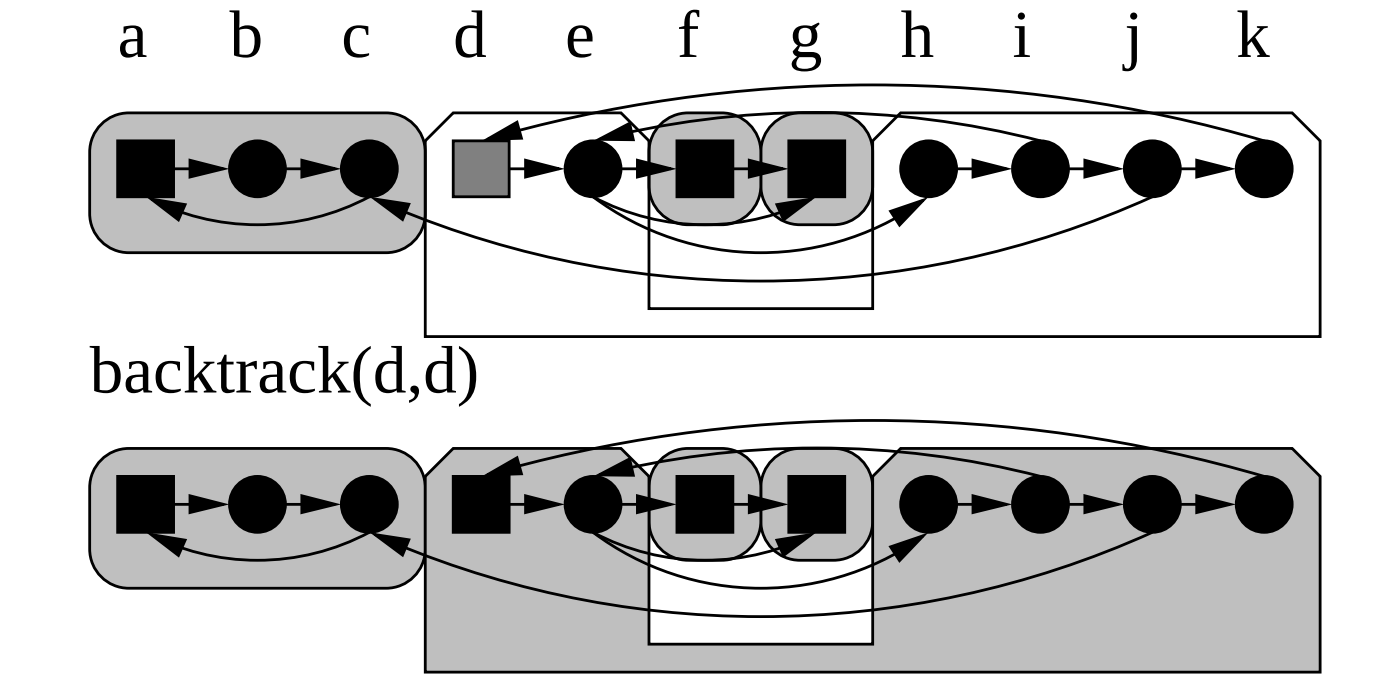

a b c d e f g h i j k
traverse(i,e)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

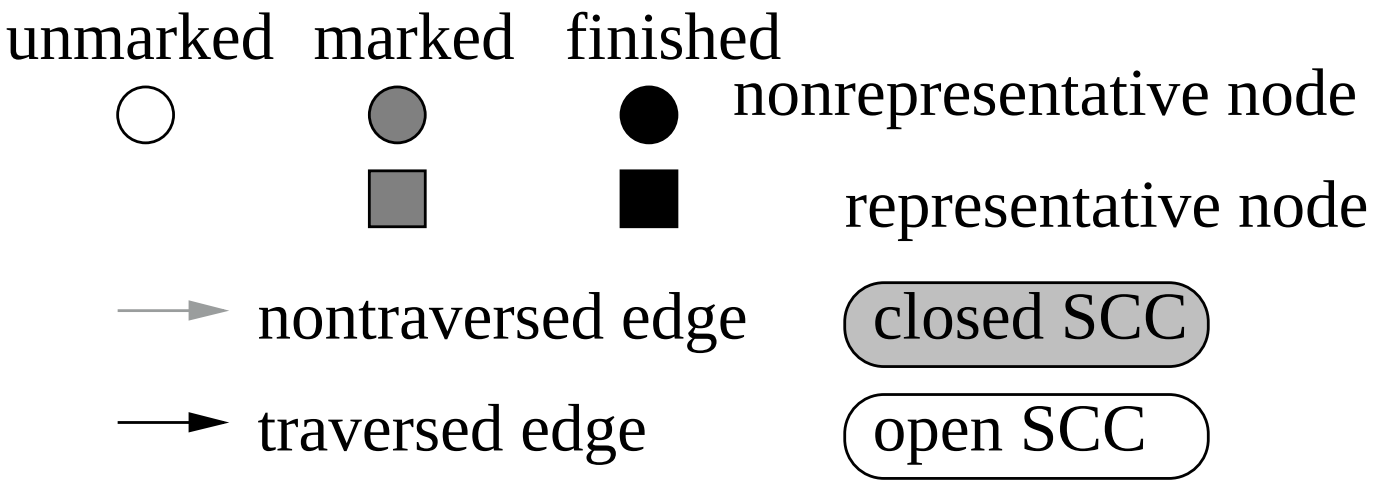

a b c d e f g h i j k
traverse(i,j)
traverse(j,c)
traverse(j,k)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
traverse(k,d)


unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
backtrack(j,k) backtrack(i,j) backtrack(h,i)
backtrack(e,h) backtrack(d,e)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
backtrack(d,d)


unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

Summary: Computing SCCs

- Simple instantiation of the DFS template
- Nontrivial correctness proof
- Running time $O(m+n)$: (at most n push/pop operations, resp.)
- A single iteration

Implementation details:

Mehlhorn, Näher, Sanders

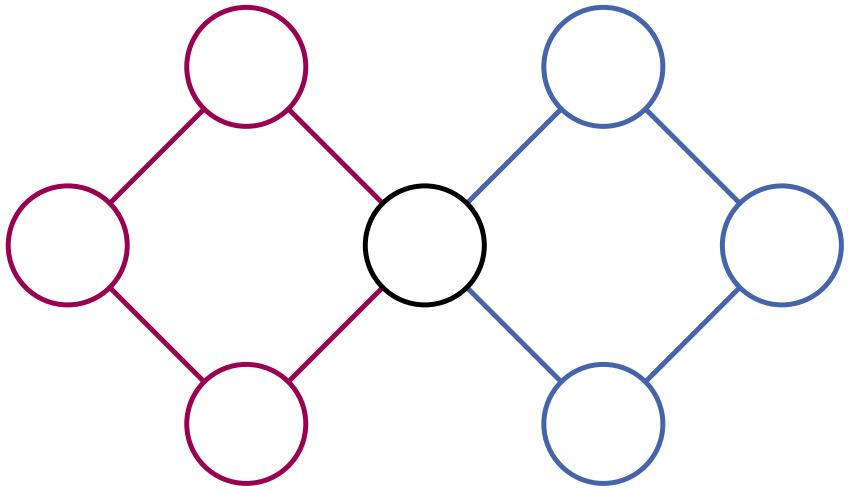
Engineering DFS-Based Graph Algorithms

arxiv.org/abs/1703.10023

2-Connected Components (Undirected)

Components remain connected when removing a single node.

(Partitioning of the edges)



Possible in $O(m+n)$ time with an algorithm similar to that for SCCs.

More DFS-based Linear Time Algorithms

- 3-connected components
- Planarity testing
- Embedding of planar graphs