Algorithmen II

Peter Sanders, Timo Bingmann

Übungen:

Sebastian Lamm, Demian Hespe

Institut für Theoretische Informatik

Web:

`http://algo2.iti.kit.edu/AlgorithmenII_WS18.php`

4 Anwendungen von DFS

Tiefensuchschema für $G = (V, E)$

```
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 Nummerierung

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

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

Beobachtung:

Knoten auf dem Rekursionsstapel sind bzgl., $\prec$ sortiert

Fertigstellungszeit

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

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

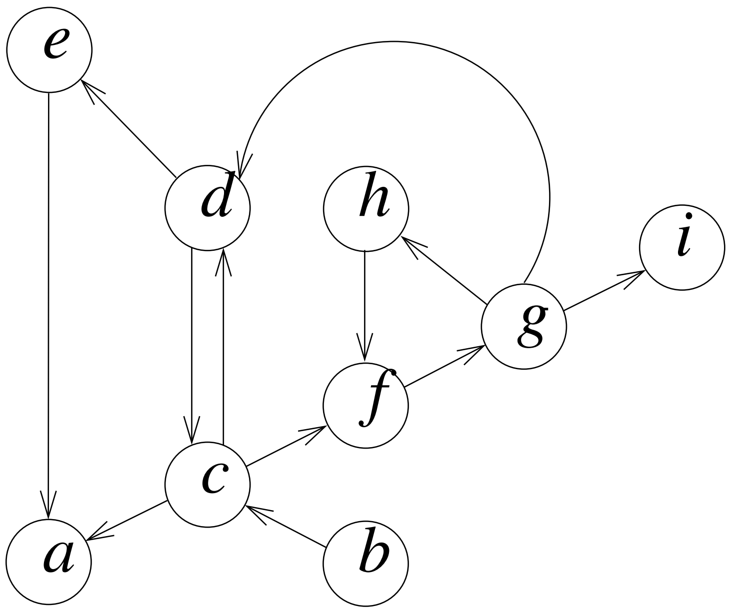
Starke Zusammenhangskomponenten

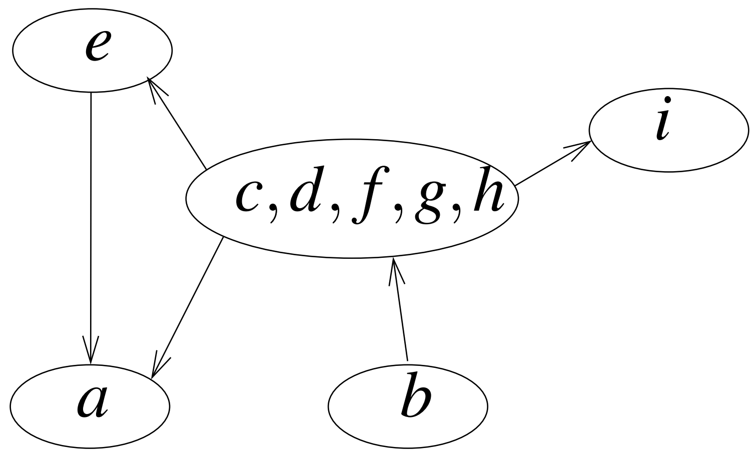
Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit

$u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle u, \ldots, v \rangle$ und $\exists$ Pfad $\langle v, \ldots, u \rangle$.

Beobachtung: $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation Übung

Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.





Starke Zusammenhangskomponenten – Abstrakter Algorithmus

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

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

  invariant SCCs of G_c are known

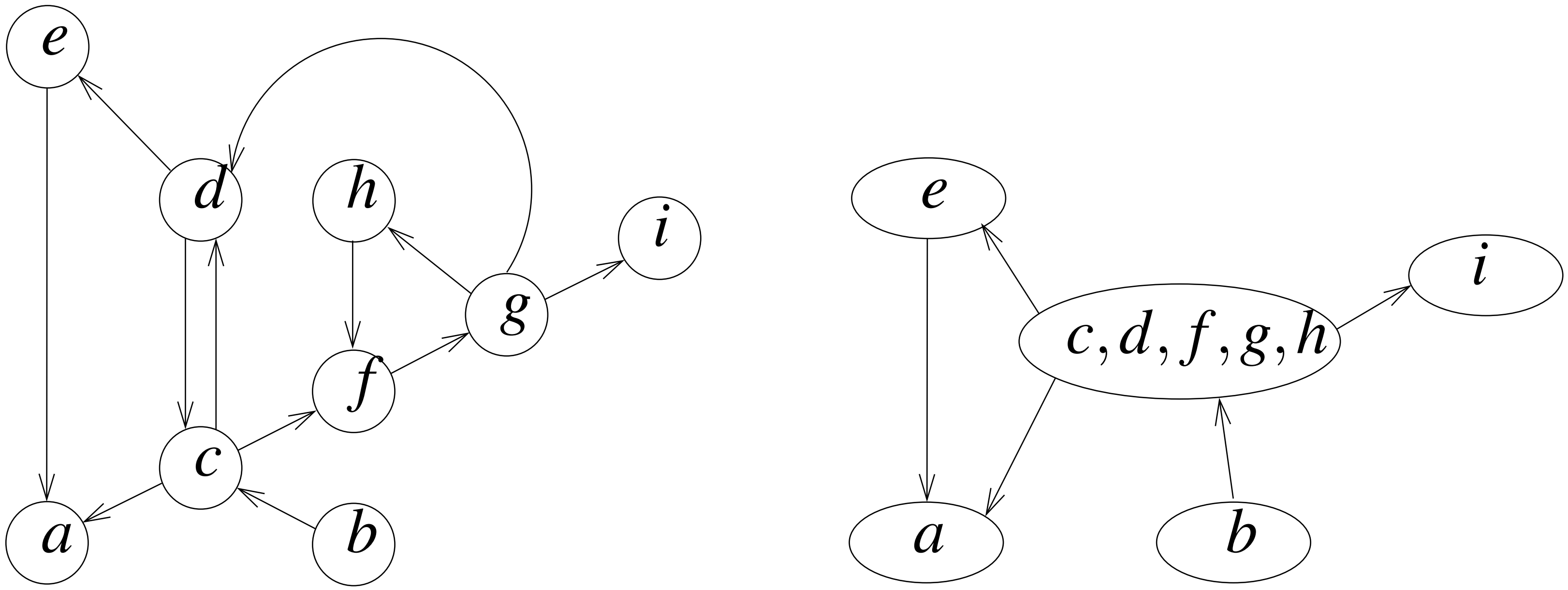
  $E_c := E_c \cup \{e\}$

Schrumpfgraph

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

Knoten: SCCs von G_c.

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



Beobachtung: Der Schrumpfgraph ist azyklisch

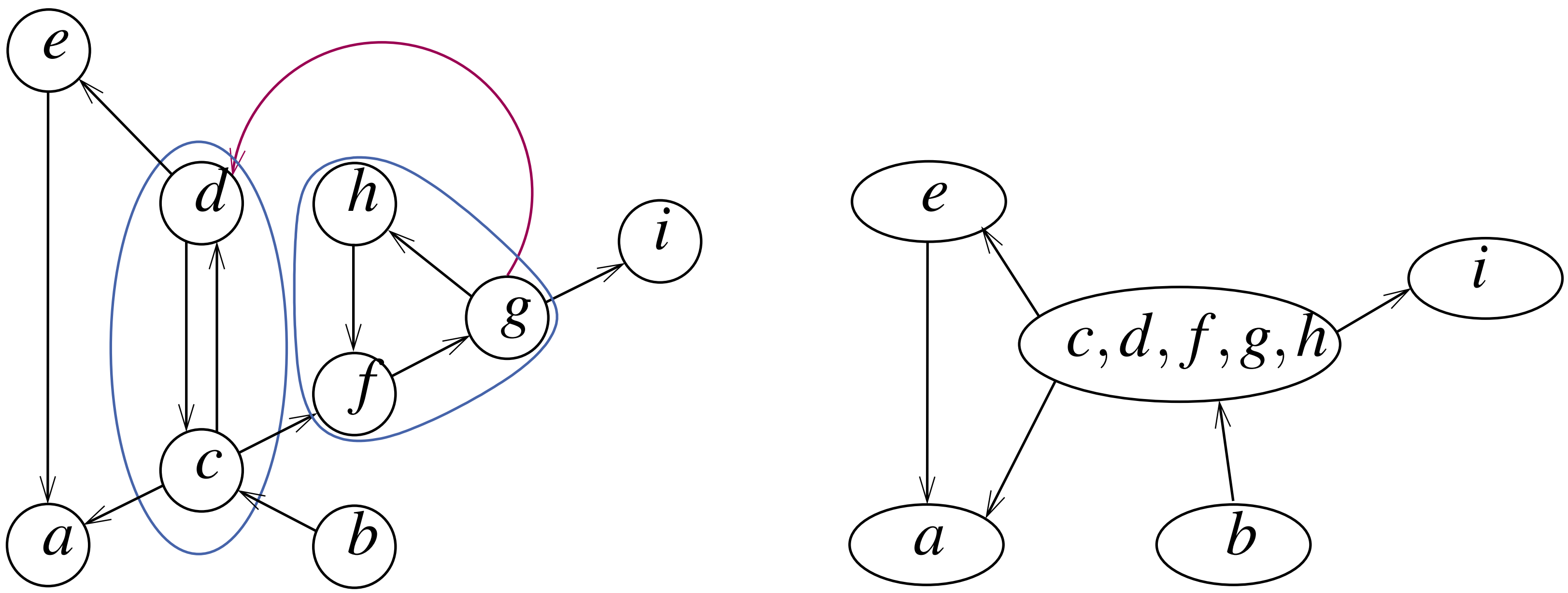
Auswirkungen einer neuen Kante e auf G_c, G_c^s

SCC-intern: Nichts ändert sich

zwischen zwei SCCs:

Kein Kreis: Neue Kante in G_c^s

Kreisschluss: SCCs auf Kreis kollabieren.



Konkreter: SCCs mittels DFS

[Cheriyan/Mehlhorn 96, Gabow 2000]

$V_c =$ markierte Knoten

$E_c =$ bisher explorierte Kanten

Aktive Knoten: markiert aber nicht finished.

SCCs von G_c:

nicht erreicht: Unmarkierte Knoten

offen: enthält aktive Knoten

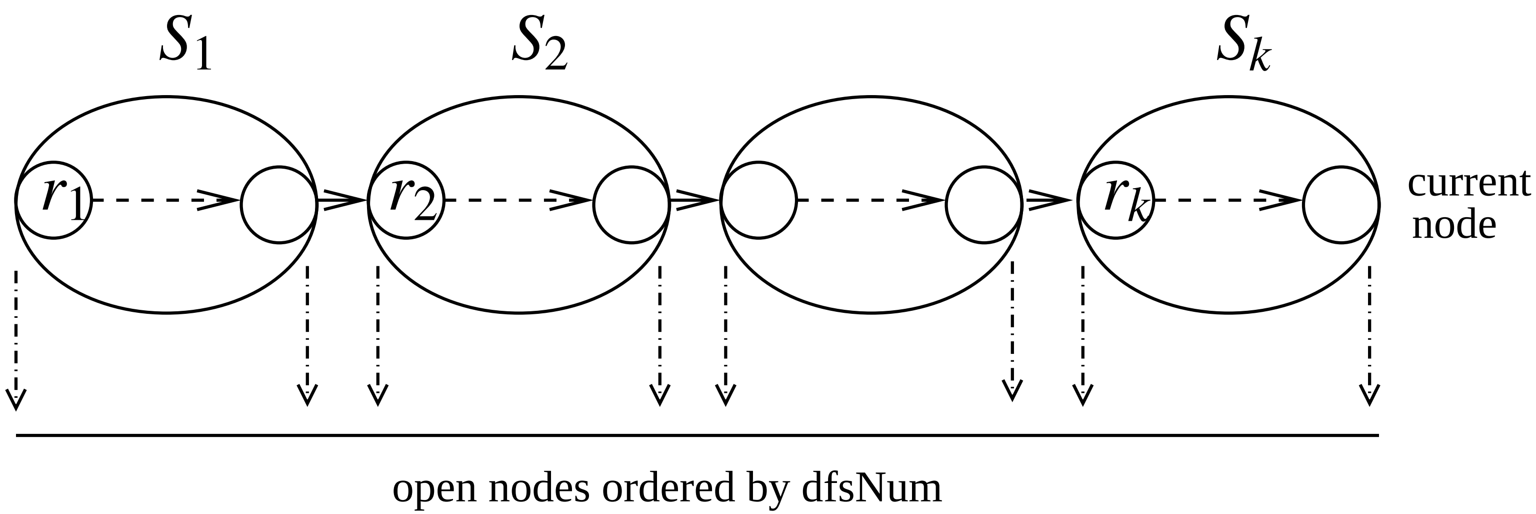
abgeschlossen: alle Knoten finished

component[w] gibt Repräsentanten einer SCC an.

Knoten von offenen (abgeschl.) Komponenten heißen offen (abgeschl.)

Invarianten von G_c

1. Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten
2. Offene Komponenten $S_1, \ldots, S_k$ bilden Pfad in G_c^s.
3. Repräsentanten partitionieren die offenen Komponenten bzgl. ihrer dfsNum.



Lemma: Abgeschlossene SCCs von G_c sind SCCs von G

Betrachte abgeschlossenen Knoten v

und beliebigen Knoten w

in der SCC von v bzgl. G.

z.Z.: w ist abgeschlossen und

in der gleichen SCC von G_c wie v.

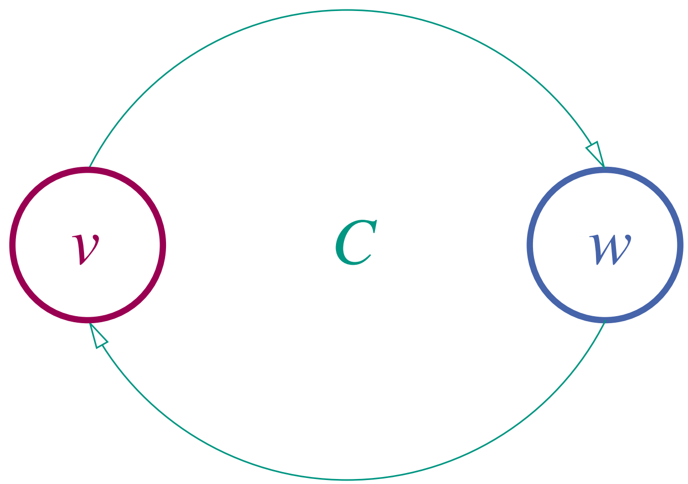
Betrachte Kreis C durch v, w.

Inv. 1: Knoten von C sind abgeschlossen.

Abgeschl. Knoten sind finished.

Kanten aus finished Knoten wurden exploriert.

Also sind alle Kanten von C in G_c. □

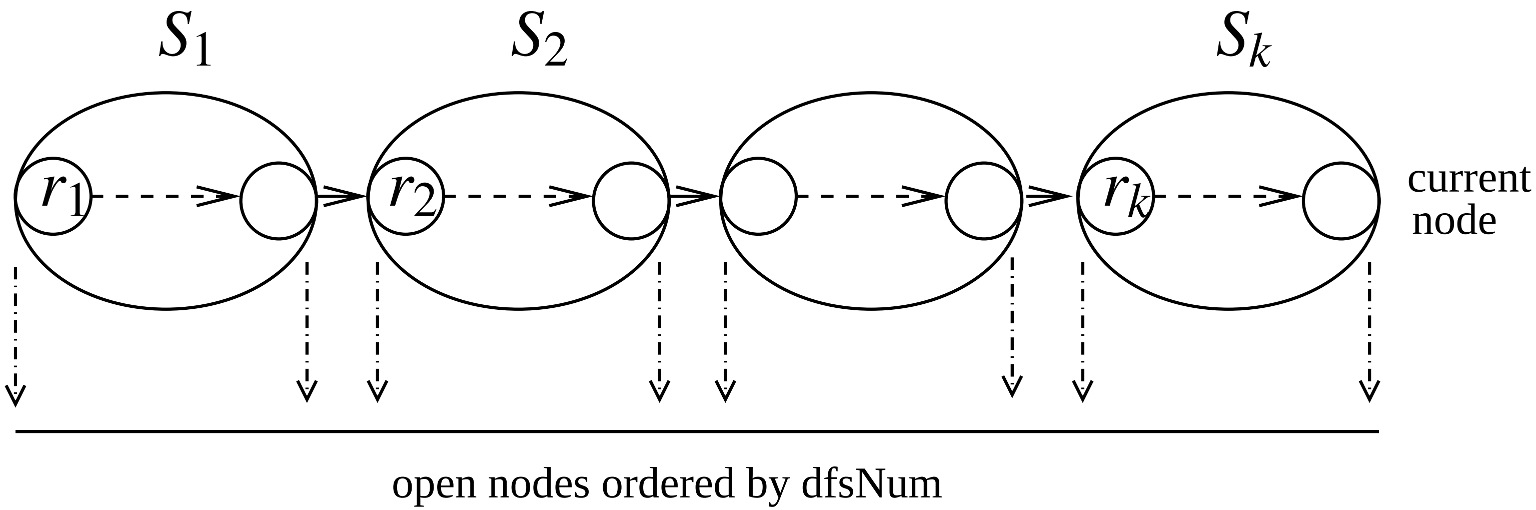


Repräsentation offener Komponenten

Zwei Stapel aufsteigend sortiert nach dfsNum

oReps: Repräsentanten offener Komponenten

oNodes: Alle offenen Knoten



init

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

oReps=$\langle\rangle$: Stack **of** NodeId // representatives of open SCCs

oNodes=$\langle\rangle$: Stack **of** NodeId // all nodes in open SCCs

Alle Invarianten erfüllt.

(Weder offene noch geschlossene Knoten)

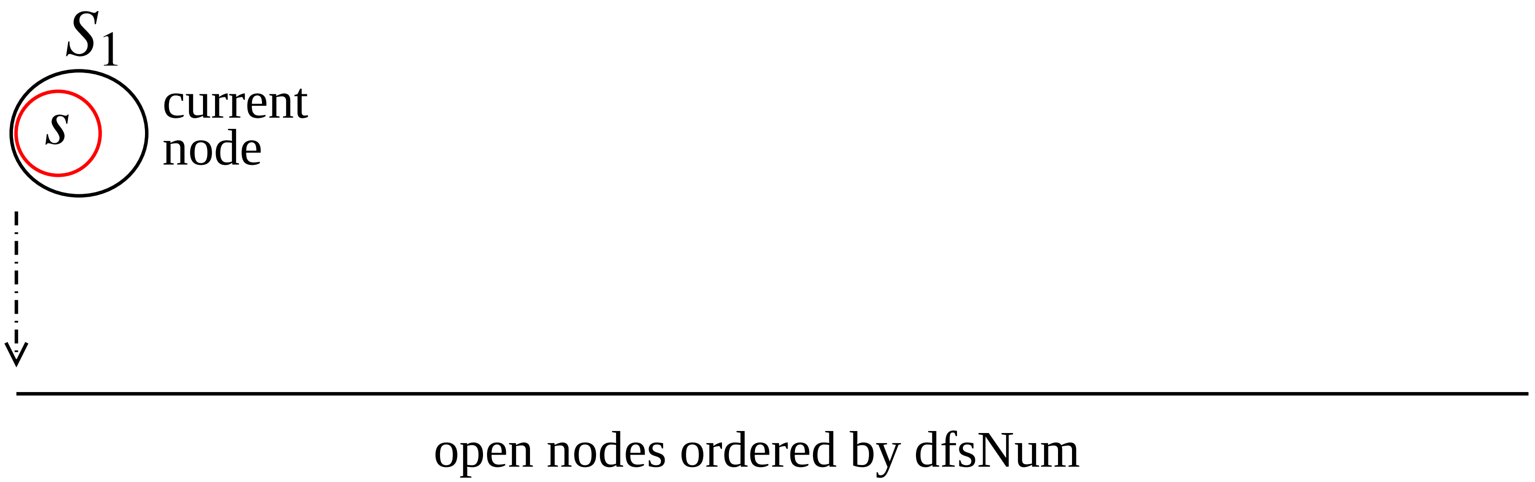
root(s)

oReps.push(s) // new open

oNodes.push(s) // component

$\{s\}$ ist die einzige offene Komponente.

Alle Invarianten bleiben gültig



traverseTreeEdge(v,w)

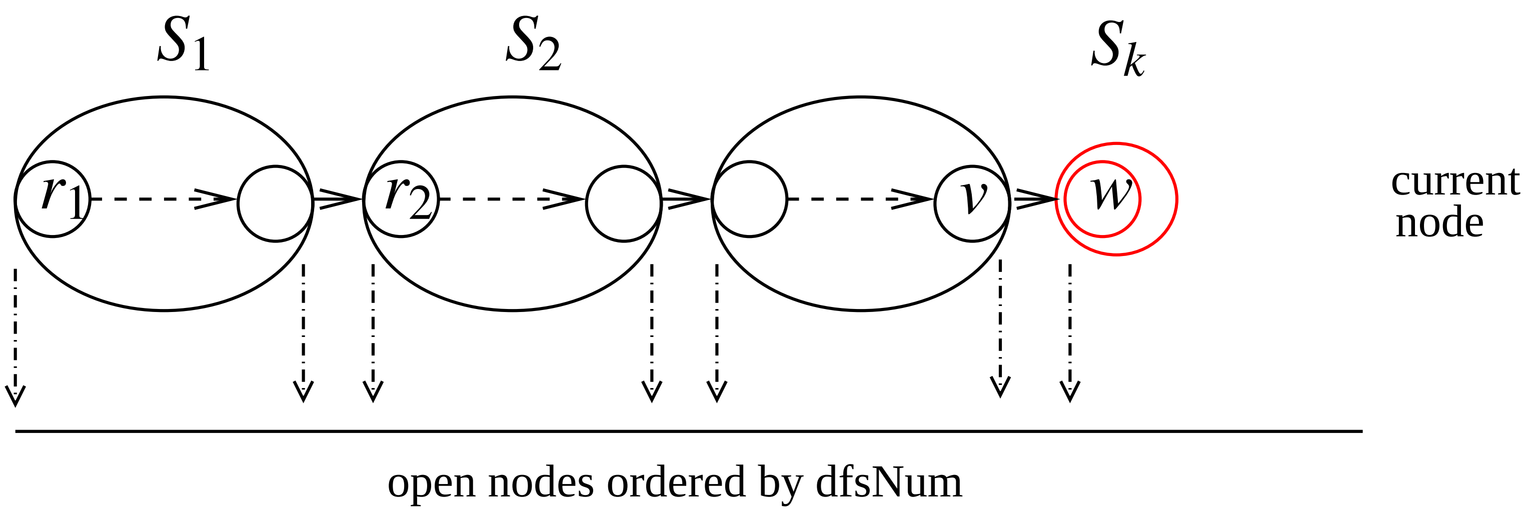
oReps.push(w) // new open

oNodes.push(w) // component

$\{w\}$ ist neue offene Komponente.

dfsNum$(w) >$ alle anderen.

$\rightsquigarrow$Alle Invarianten bleiben gültig



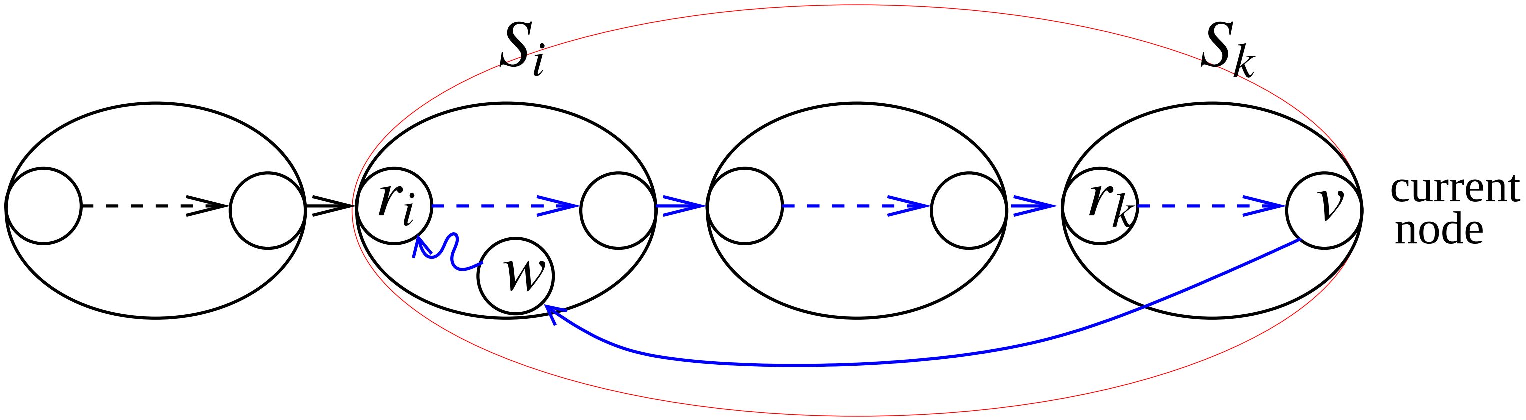
traverseNonTreeEdge(v, w)

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

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

$w \notin$ oNodes $\rightsquigarrow$ w is abgeschlossen $\overset{Lemma(*)}{\rightsquigarrow}$ Kante uninteressant

$w \in$ oNodes: kollabiere offene SCCs auf Kreis



backtrack(u,v)

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

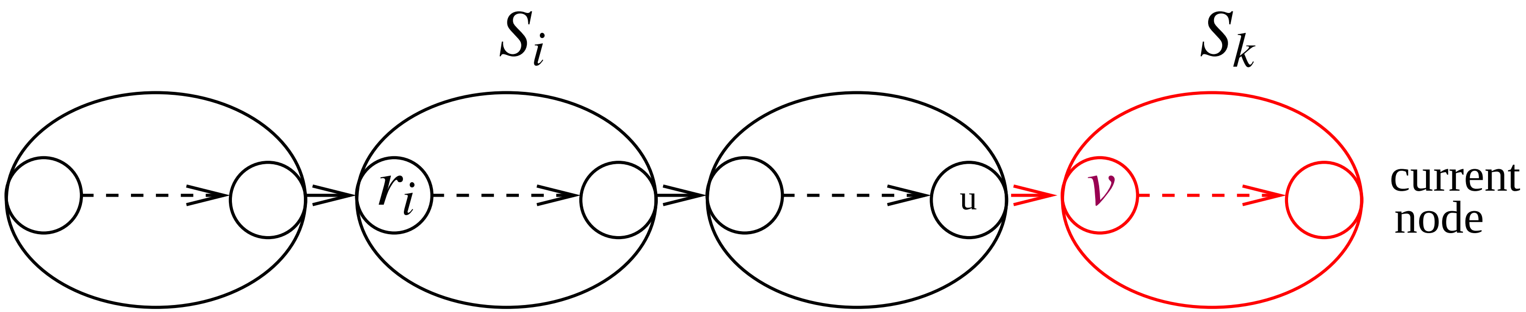
oReps.pop // close

repeat // component

w:= oNodes.pop

component$[w]$:= v

until $w=v$



z.Z. Invarianten bleiben erhalten. . .

backtrack(u,v)

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

    oReps.pop // close

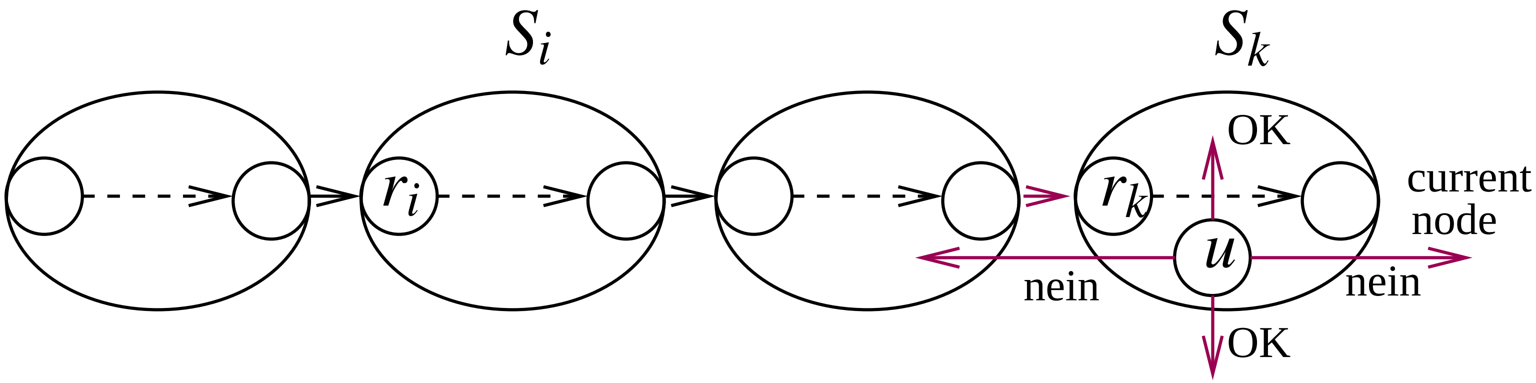
    repeat // component

      w:= oNodes.pop

      component$[w]$:= v

    until $w = v$

Inv. 1: Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten.



backtrack(u,v)

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

oReps.pop // close

repeat // component

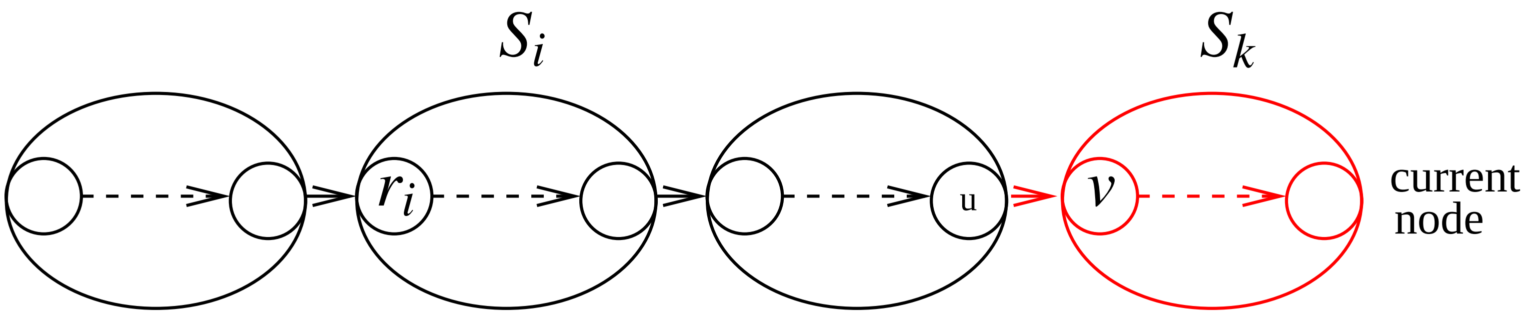
w:= oNodes.pop

component$[w]$:= v

until $w = v$

Inv. 2: Offene Komponenten $S_1,\ldots,S_k$ bilden Pfad in G_c^s

OK. (S_k wird ggf. entfernt)



backtrack(u,v)

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

    oReps.pop // close

    repeat // component

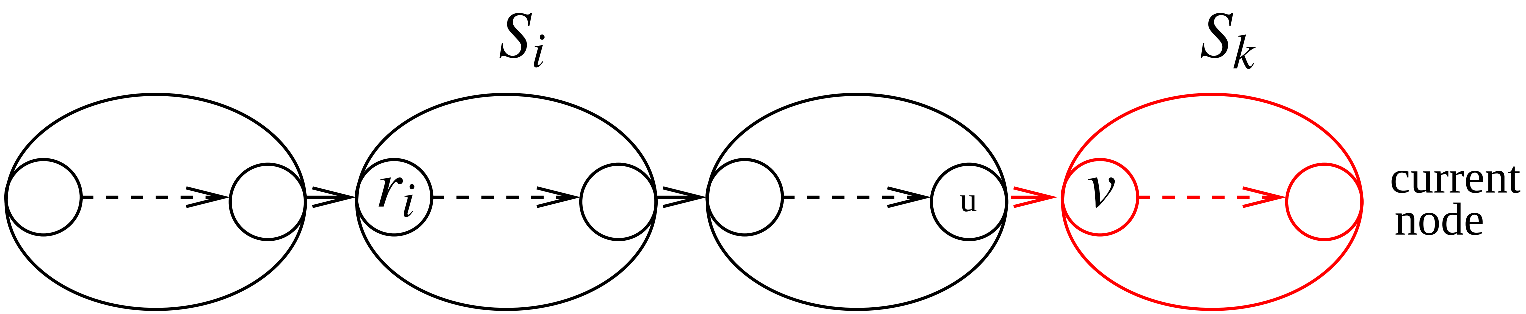
      w:= oNodes.pop

      component$[w]$:= v

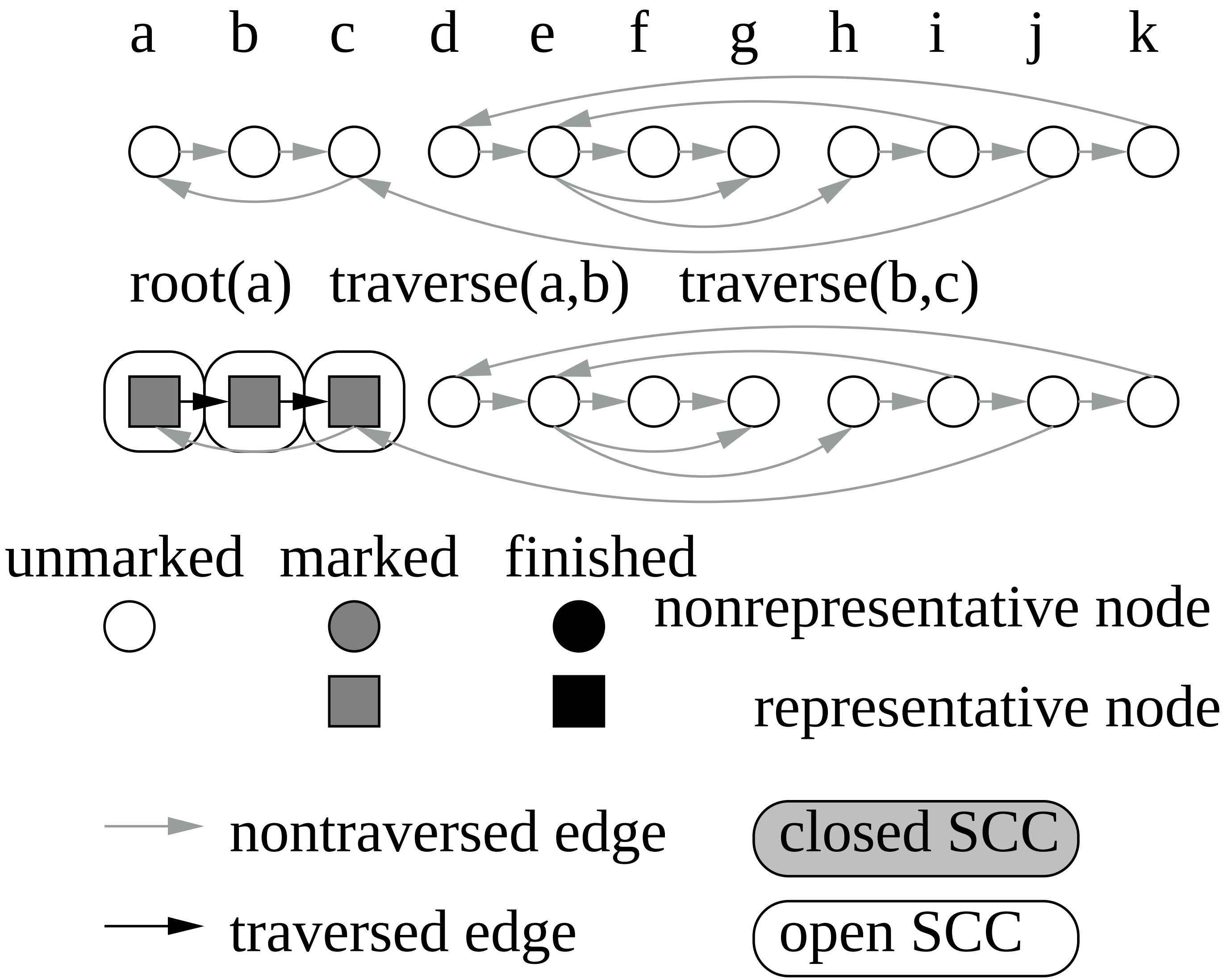
    until $w = v$

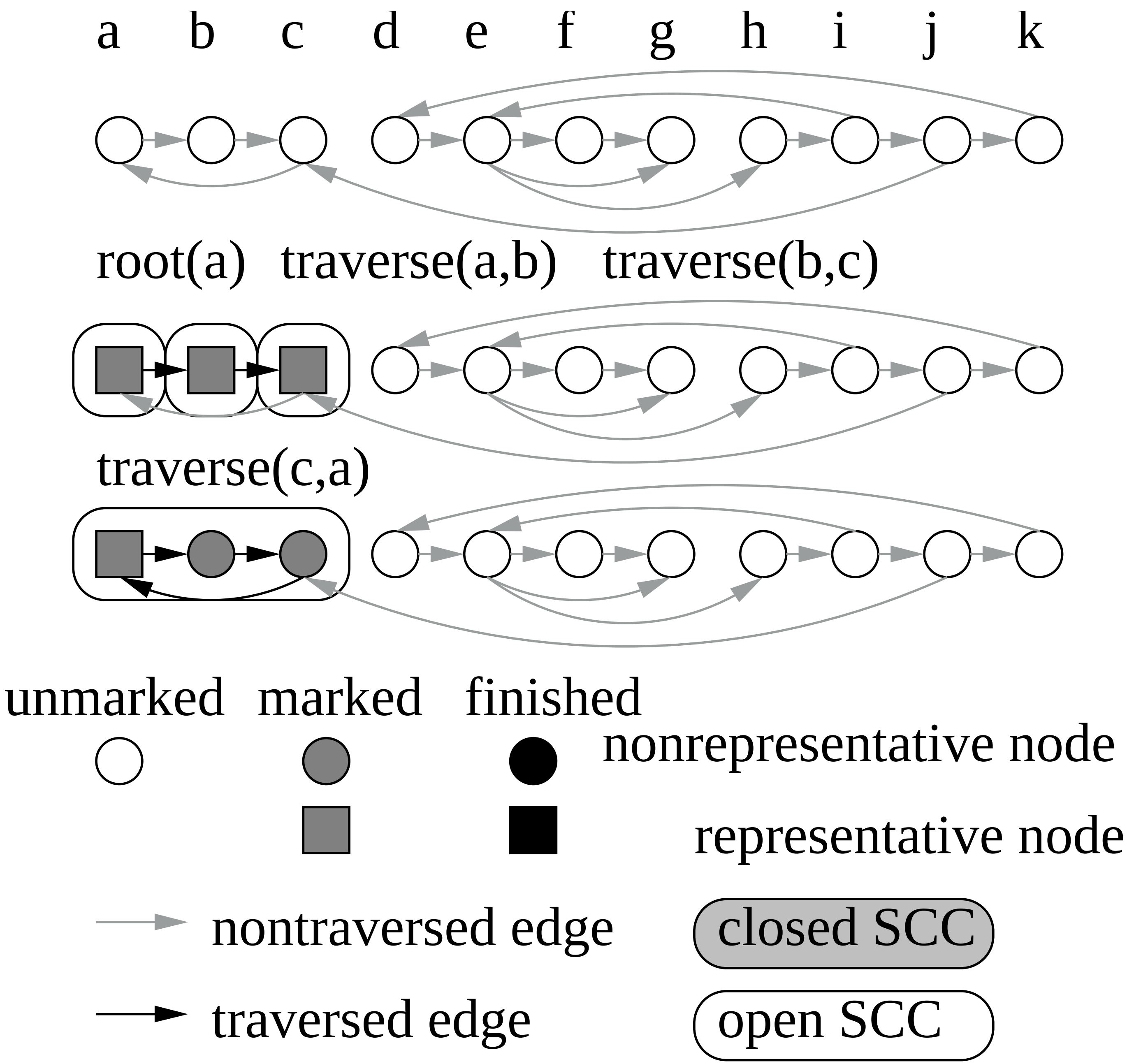
Inv. 3: Repräsentanten partitionieren die offenen Komponenten bzgl. ihrer dfsNum.

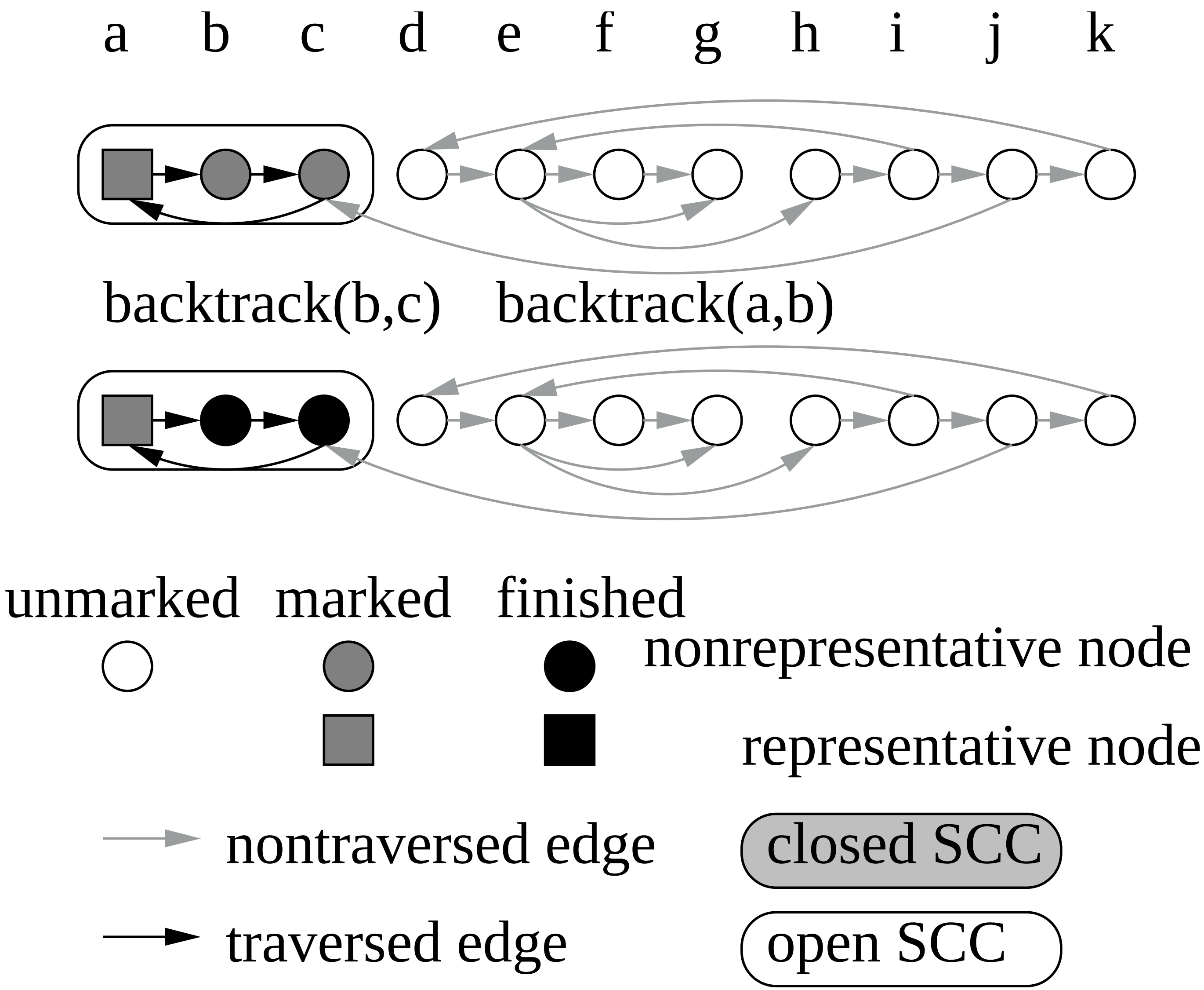
OK. (S_k wird ggf. entfernt)

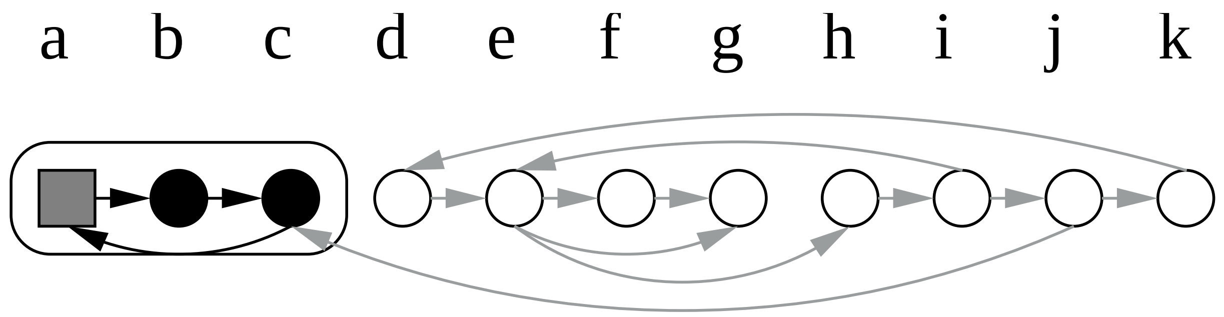


Beispiel

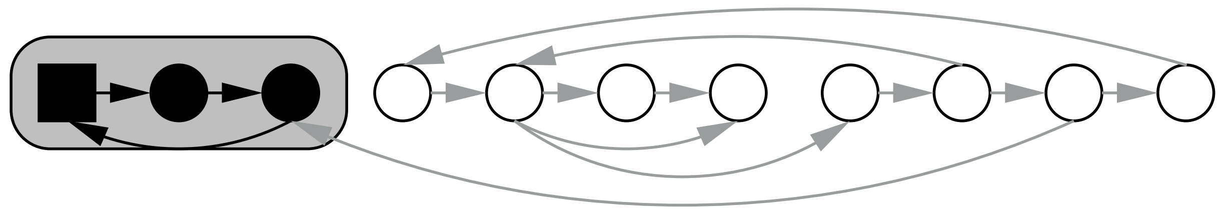

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

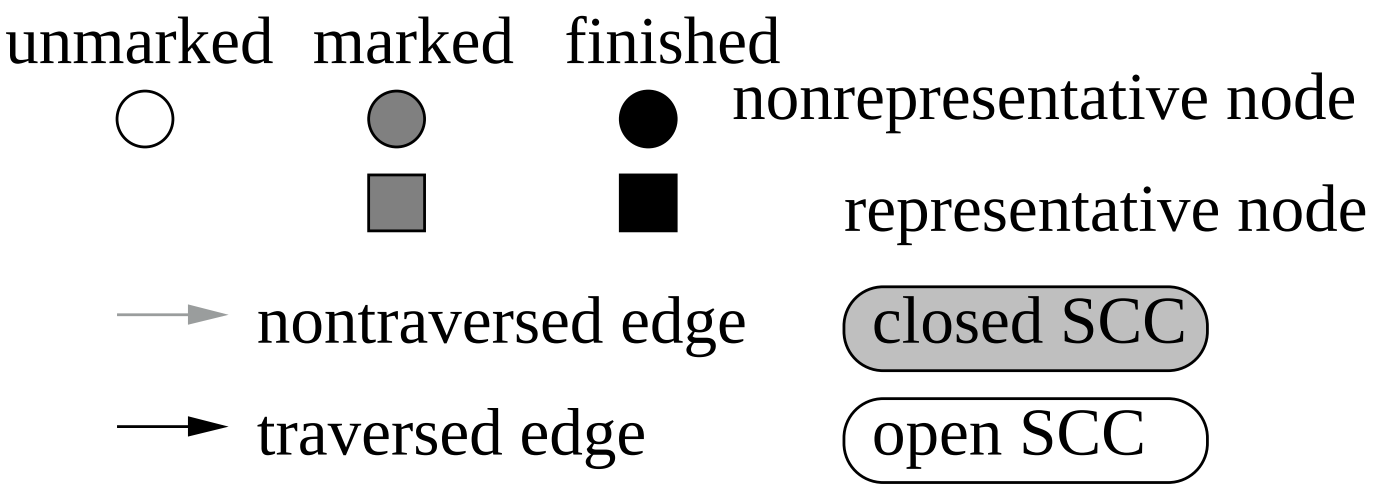

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
traverse(c,a)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

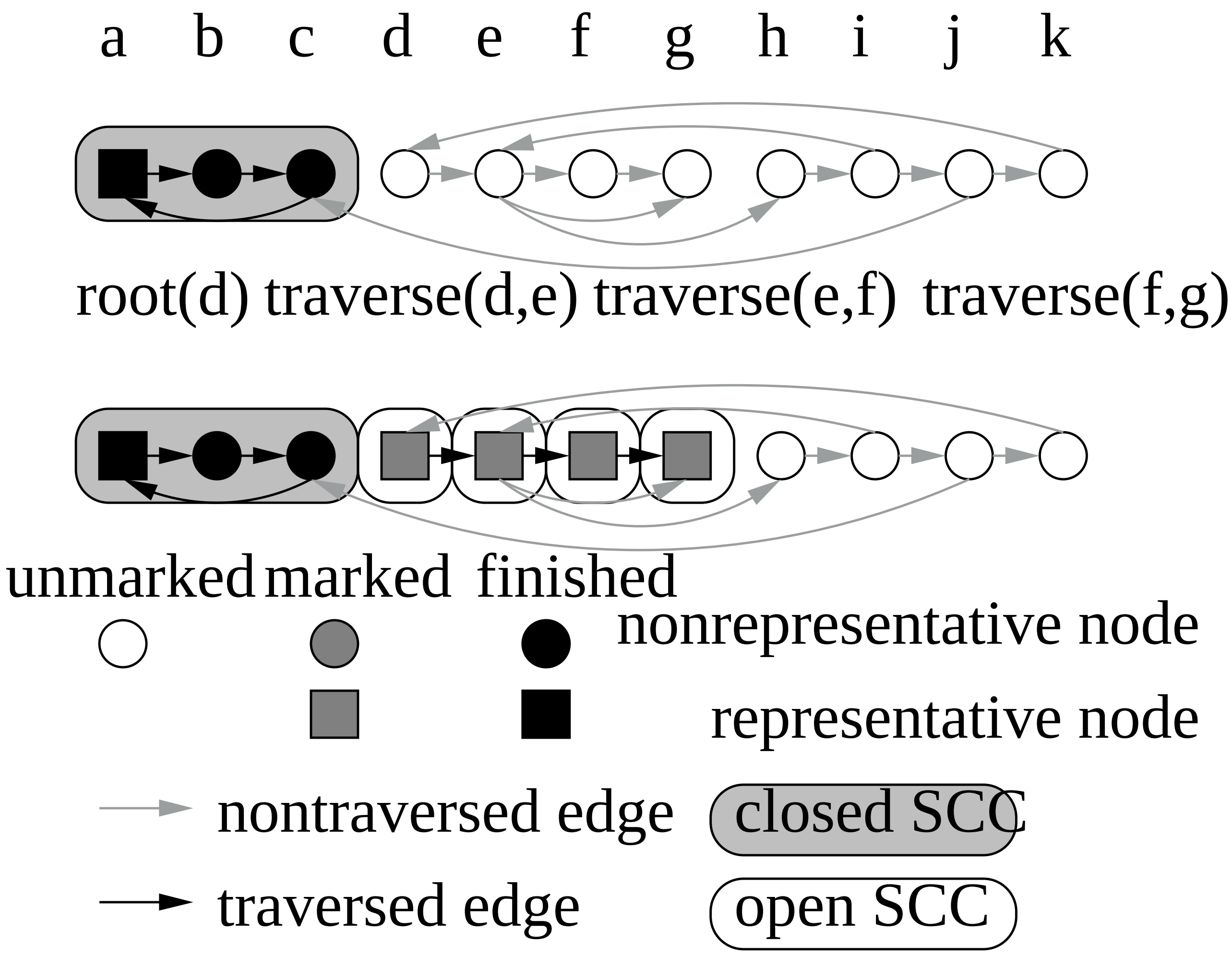

a b c d e f g h i j k
backtrack(b,c) backtrack(a,b)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

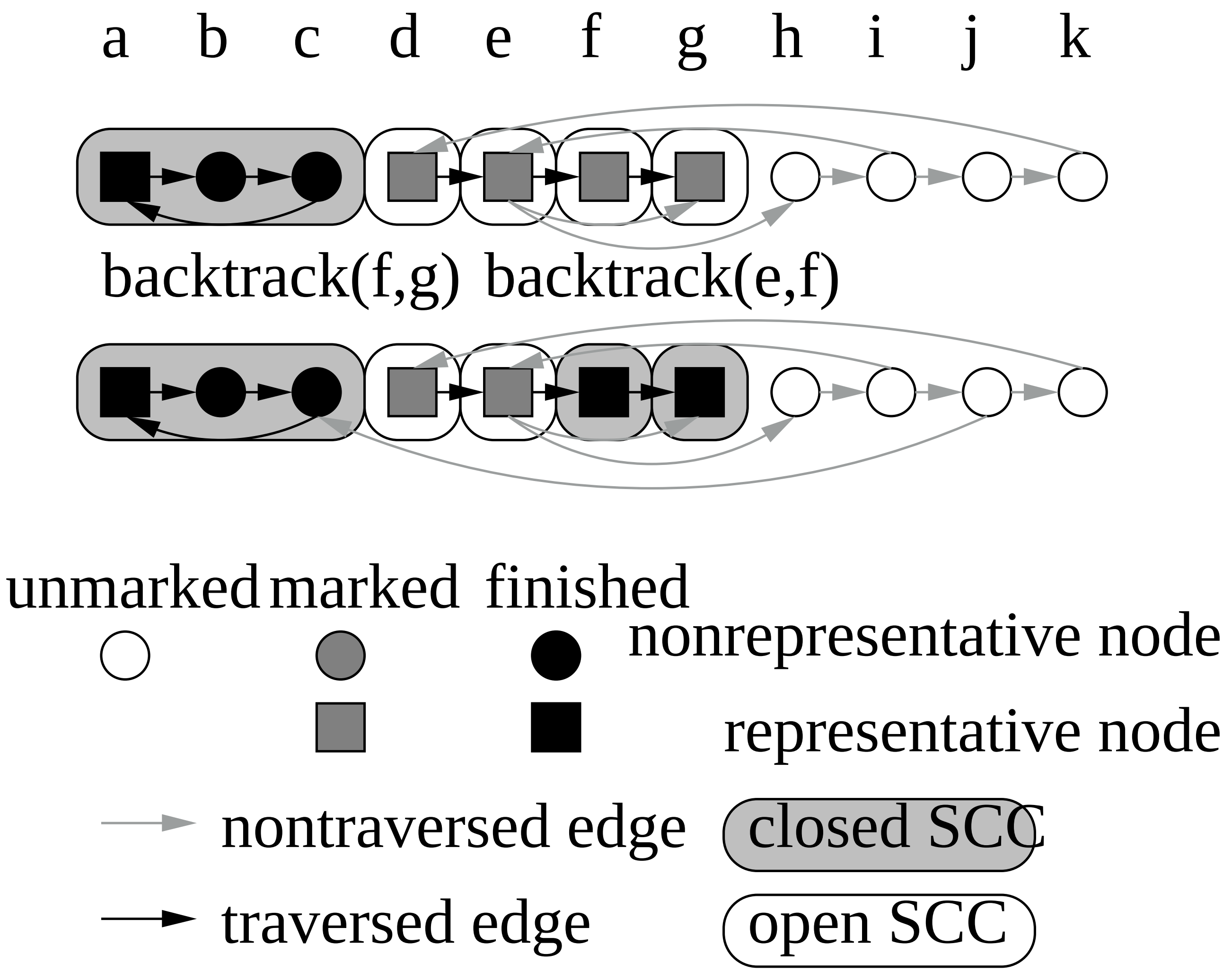


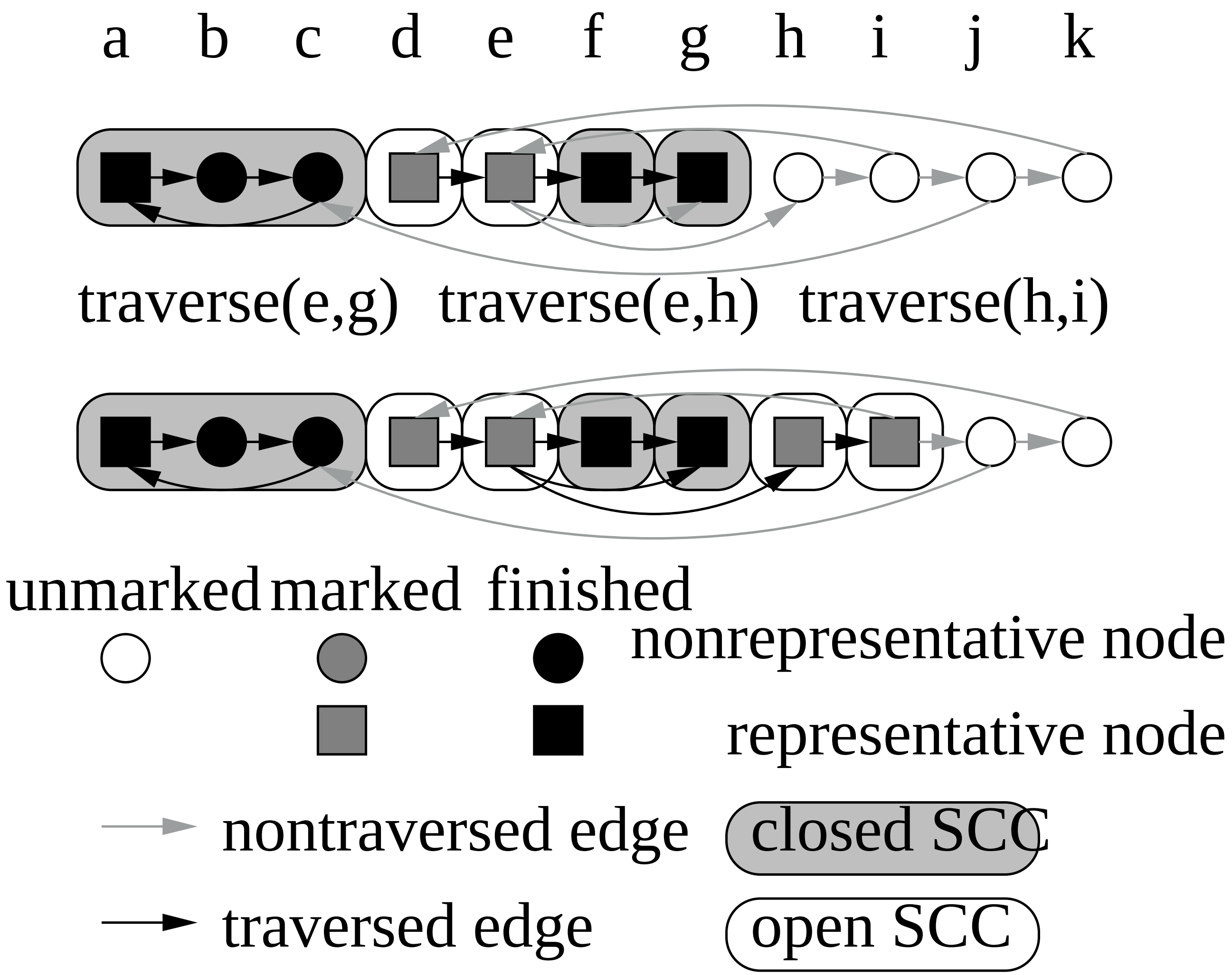
backtrack(a,a)

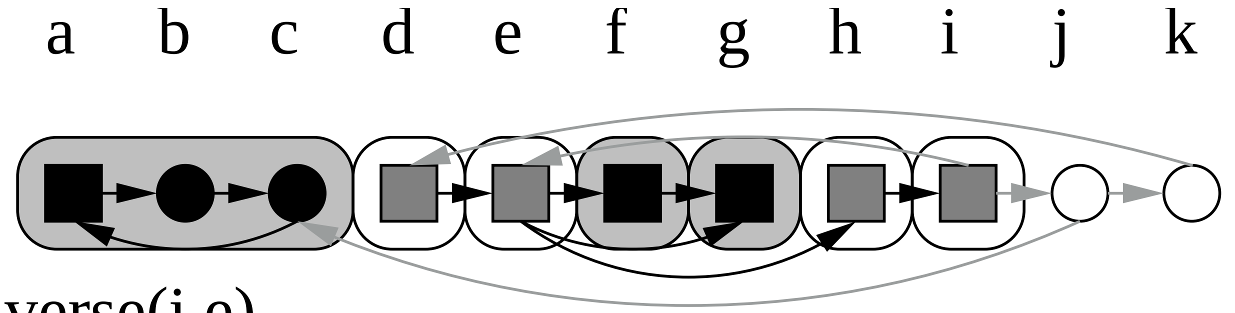




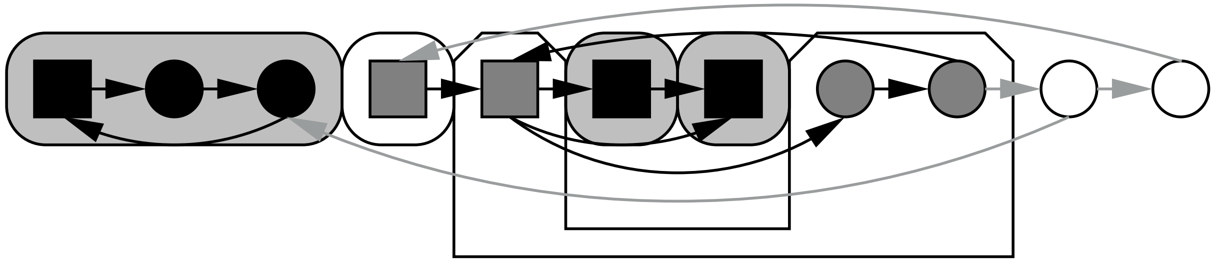

a b c d e f g h i j k
root(d) traverse(d,e) traverse(e,f) traverse(f,g)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


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



traverse(i,e)



unmarked marked finished

nonrepresentative node

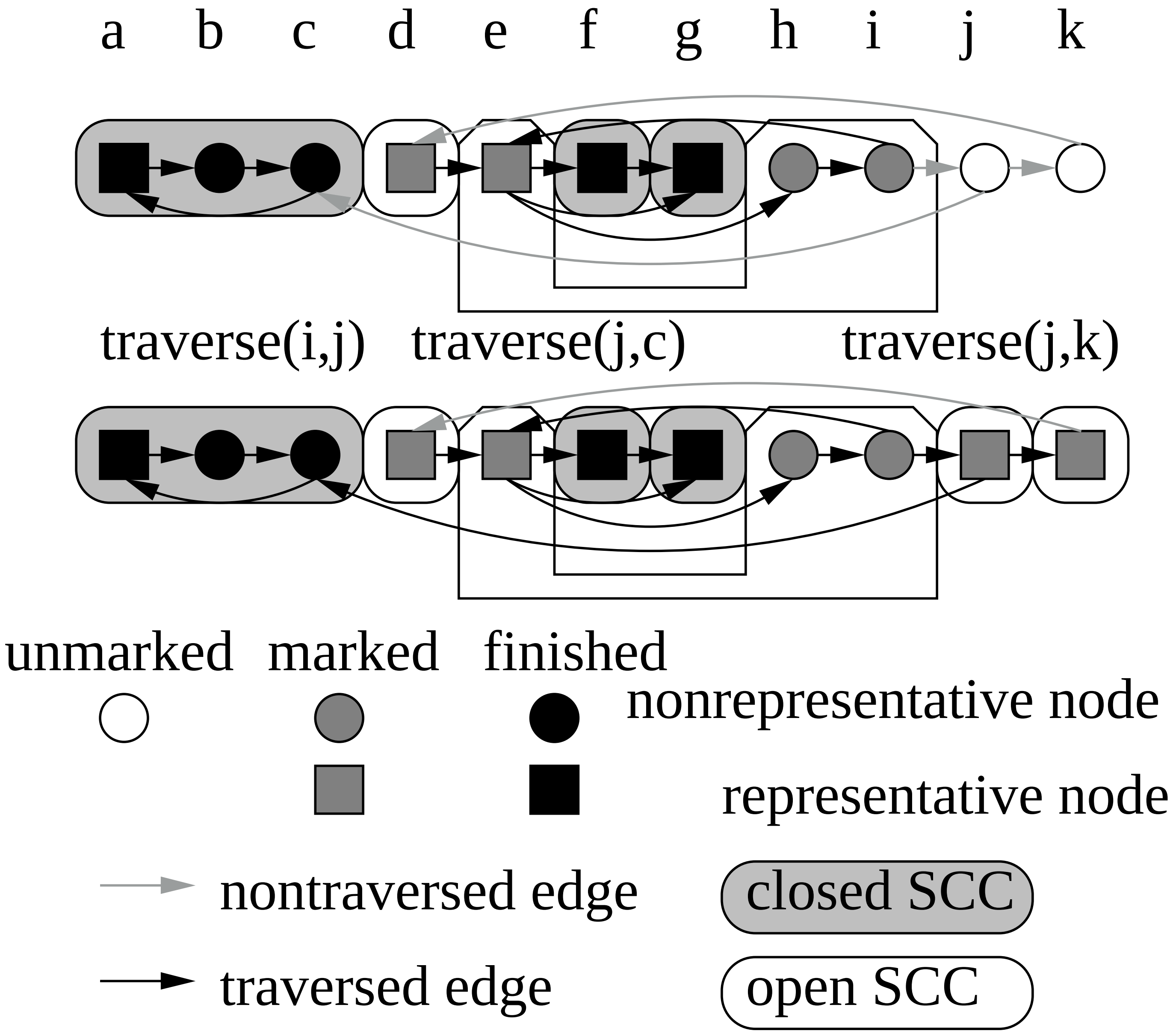
representative node

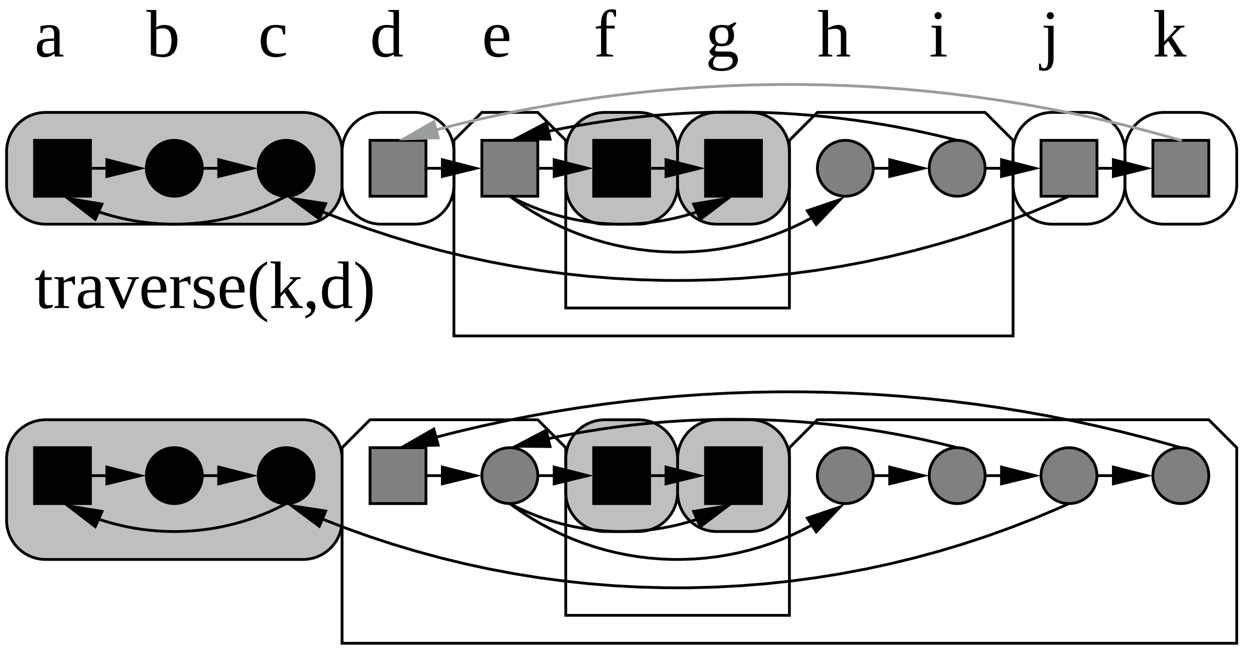
nontraversed edge

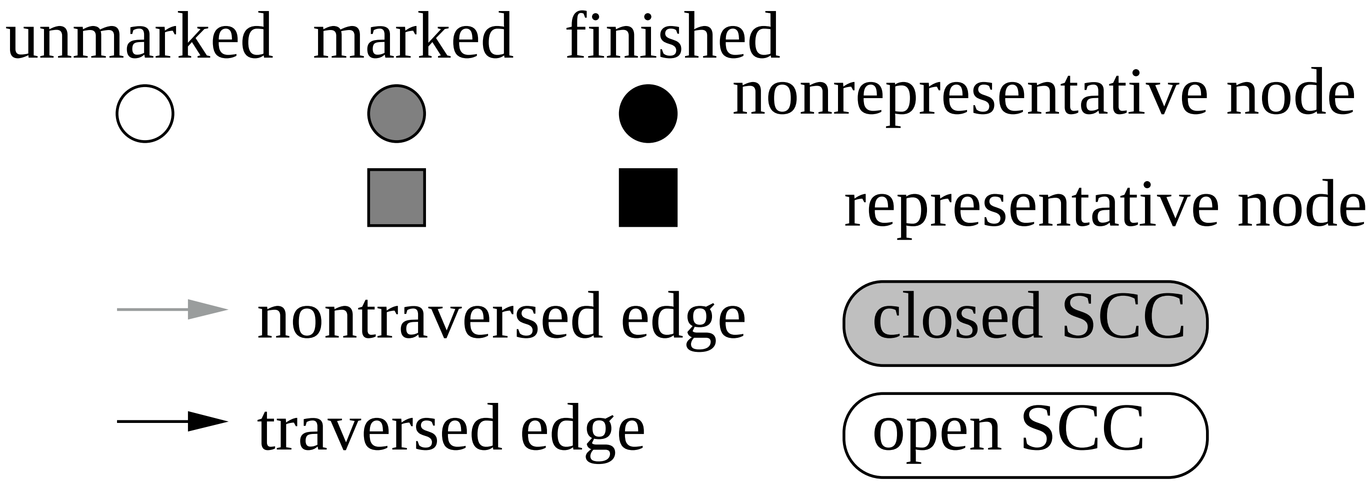
closed SCC

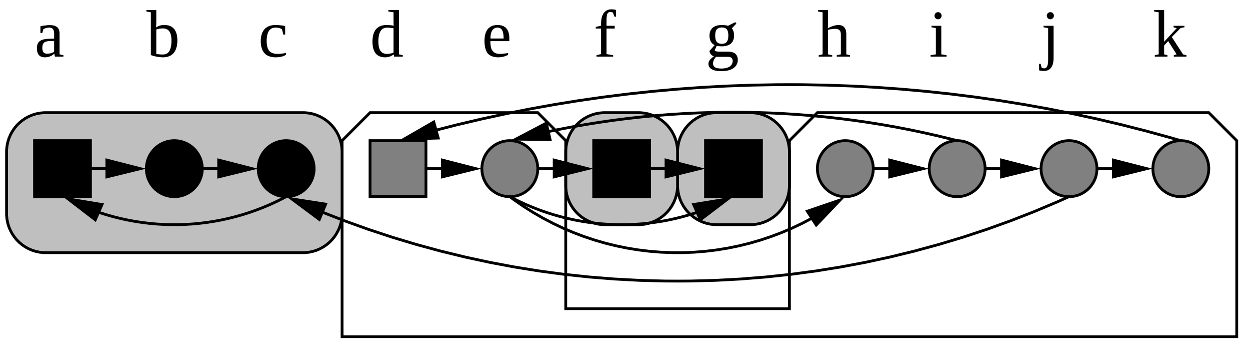
traversed edge

open SCC

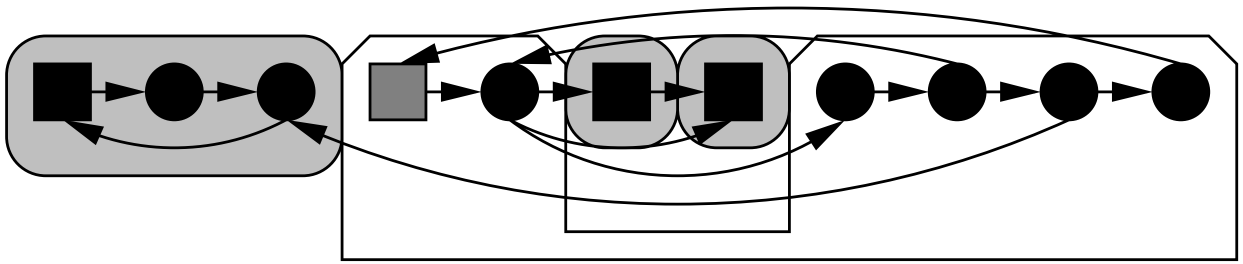

a b c d e f g h i j k
traverse(i,j) traverse(j,c) traverse(j,k)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

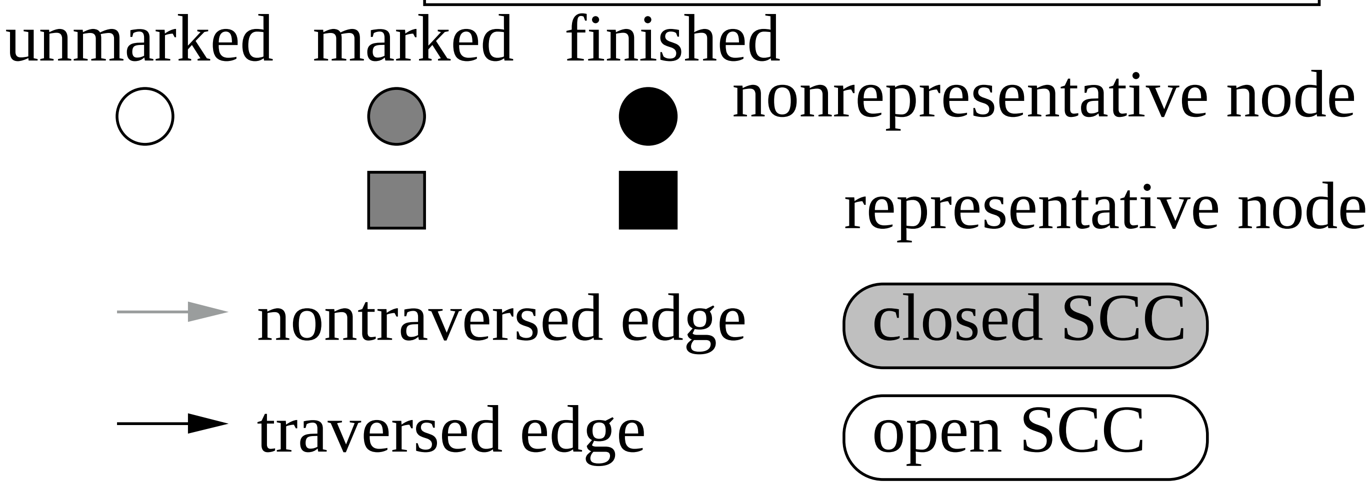

a b c d e f g h i j k
traverse(k,d)

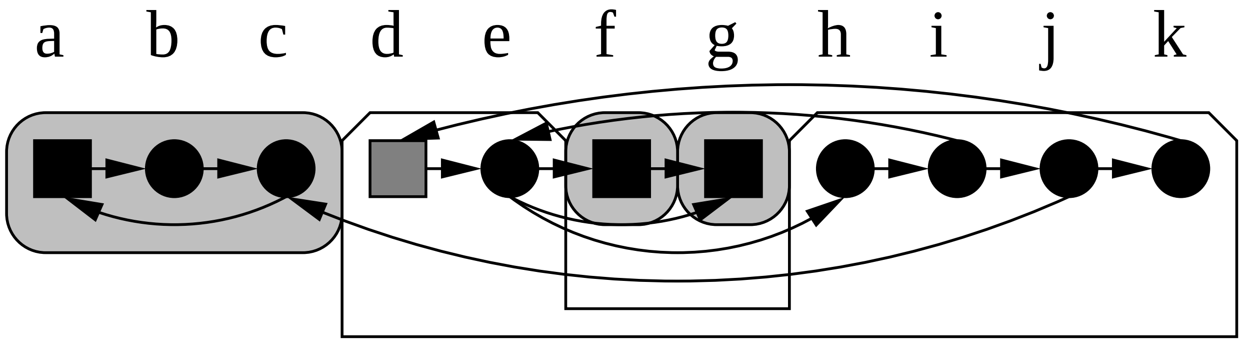

unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC



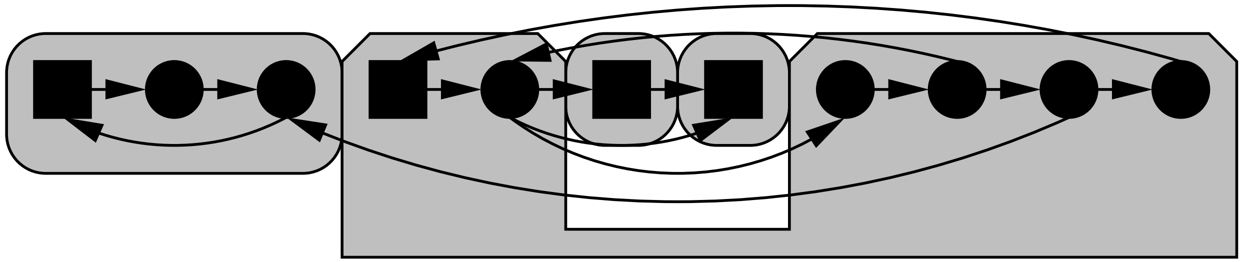
backtrack(j,k) backtrack(i,j) backtrack(h,i)
backtrack(e,h) backtrack(d,e)

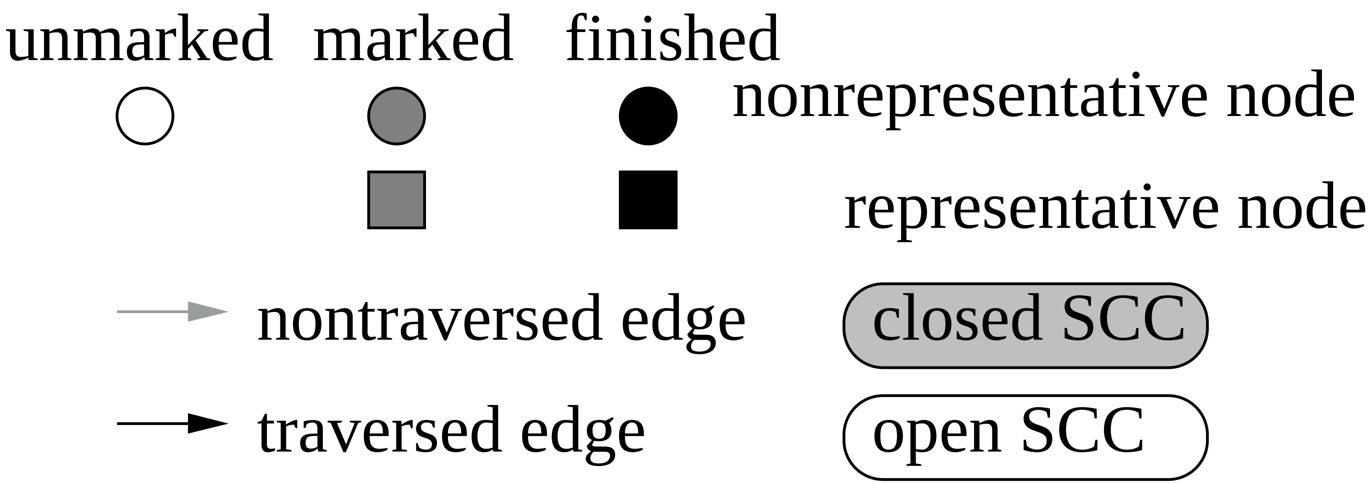






backtrack(d,d)





Zusammenfassung: SCC Berechnung

- Einfache Instantiierung des DFS-Musters
- Nichttrivialer Korrektheitsbeweis
- Laufzeit $\mathrm{O}(m+n)$: (Jeweils max. n push/pop Operationen)
- Ein einziger Durchlauf

Implementierungsdetails:

Mehlhorn, Näher, Sanders

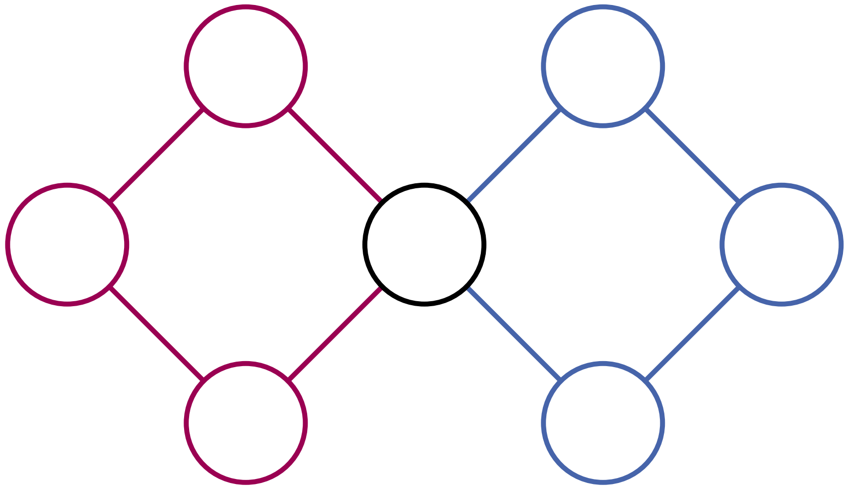
Engineering DFS-Based Graph Algorithms

arxiv.org/abs/1703.10023

2-zusammenhängende Komponenten (ungerichtet)

Bei entfernen eines Knotens bleibt die Komponente zusammenhängend.

(Partitionierung der Kanten)



Geht in Zeit $\mathrm{O}(m+n)$ mit Algorithmus ähnlich zu SCC-Algorithmus

Mehr DFS-basierte Linearzeitalgorithmen

- 3-zusammenhängende Komponenten
- Planaritätstest
- Einbettung planarer Graphen