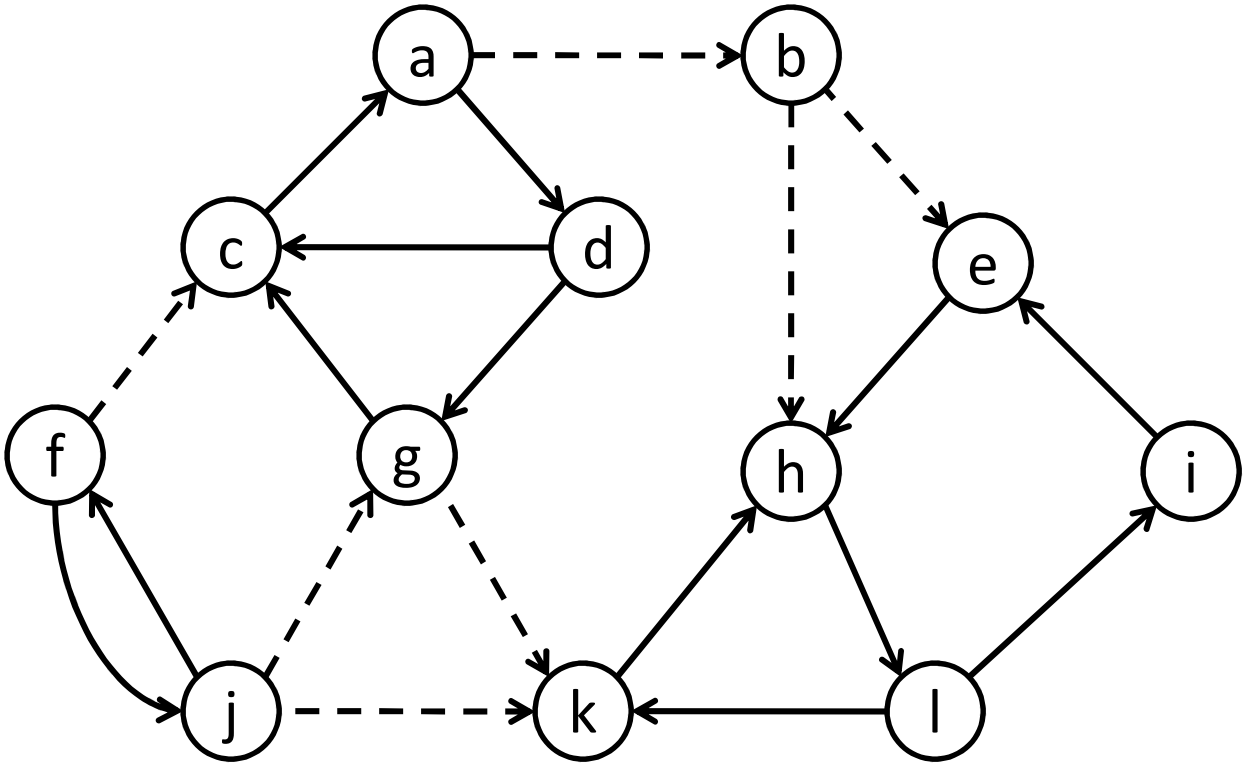
COS 423 Lecture 15
Strong components and blocks

Strong components



One-way algorithm (variant of Tarjan 1982, Gabow 2000)

Starting idea (Purdom 1968): Do a depth-first exploration. When traversing a cycle arc, contract the set of vertices on the corresponding cycle (back arc + tree path) into a single vertex. When postvisiting a vertex, list the original vertices contracted into it as a component.

Correctness: Contraction preserves strong components

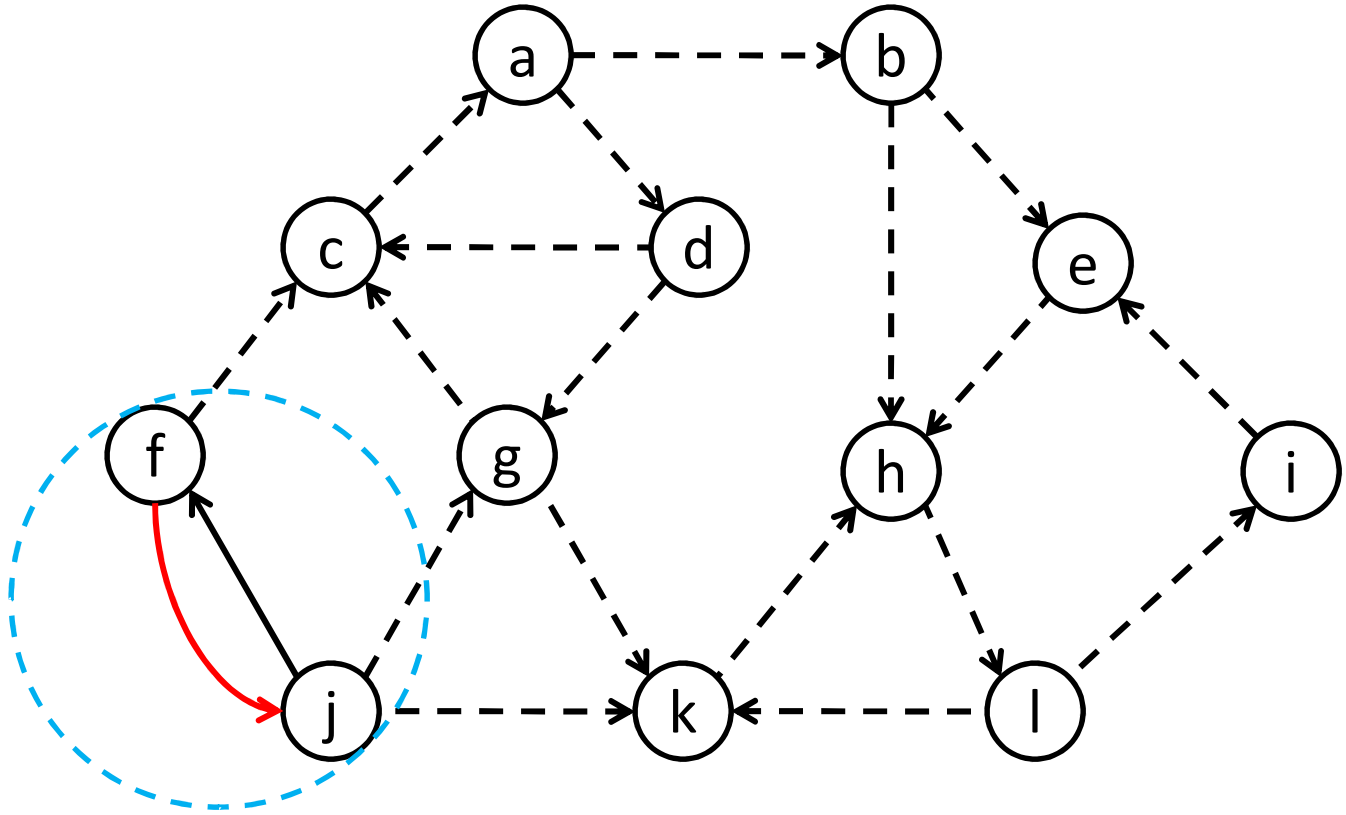
Straightforward to implement using disjoint set data structure, running time = $O(n + m\alpha(n, d))$

Components are listed in reverse topological order

Search from j

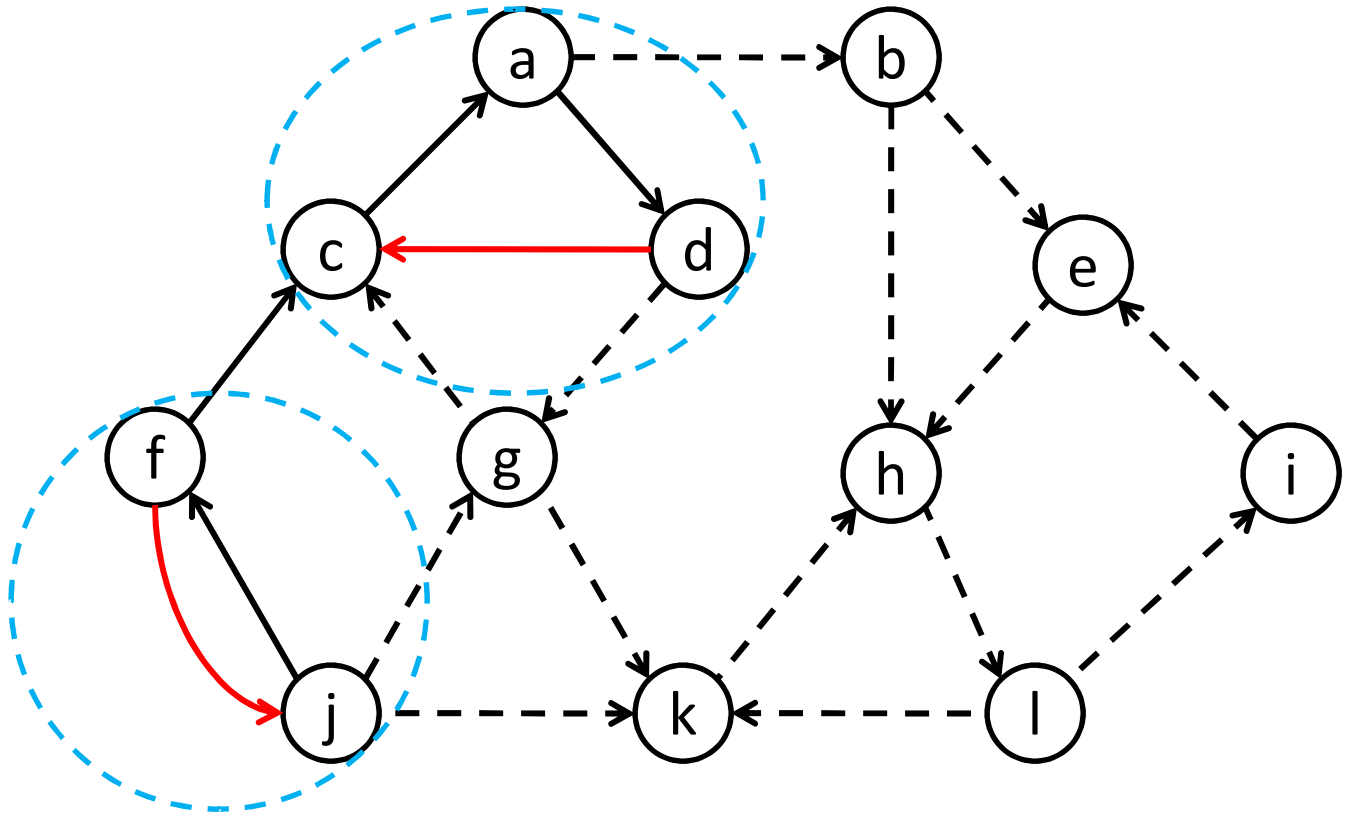
Current path j, f

Traversal of (f, j) contracts f, j into j’



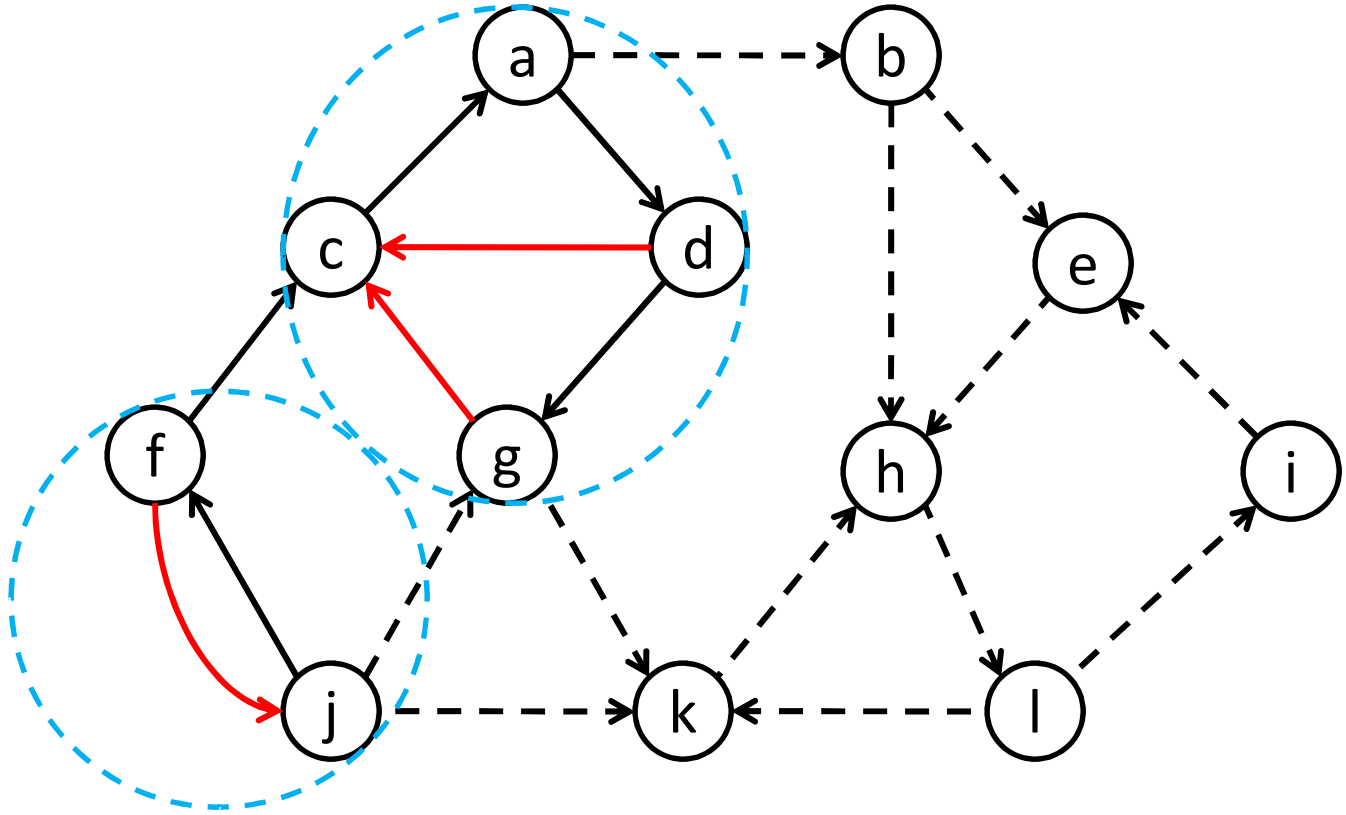
Current path j’, c, a, d

Traversal of (d, c) contracts d, a, c into c’



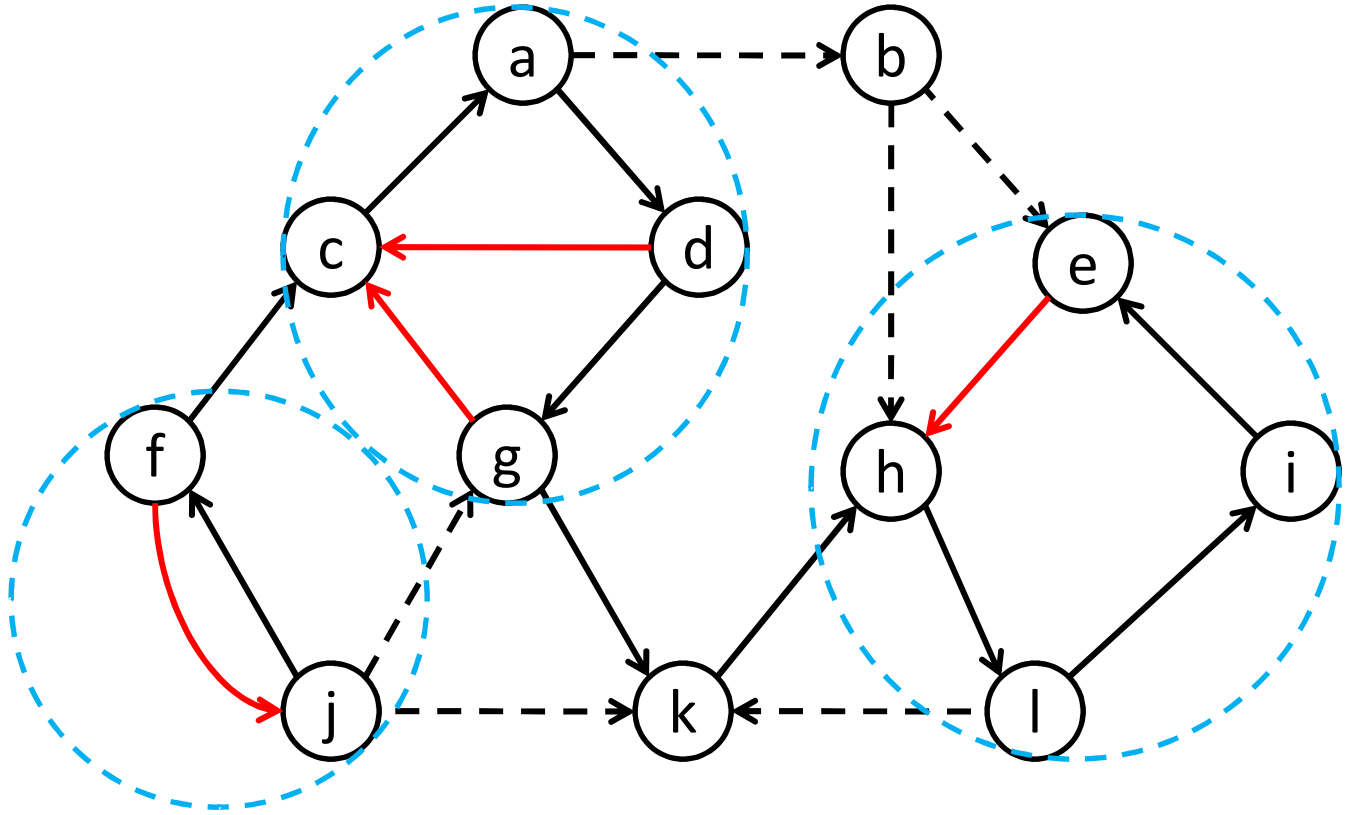
Current path j’, c’, g

Traversal of (g, c) contracts g, c’ into c’’



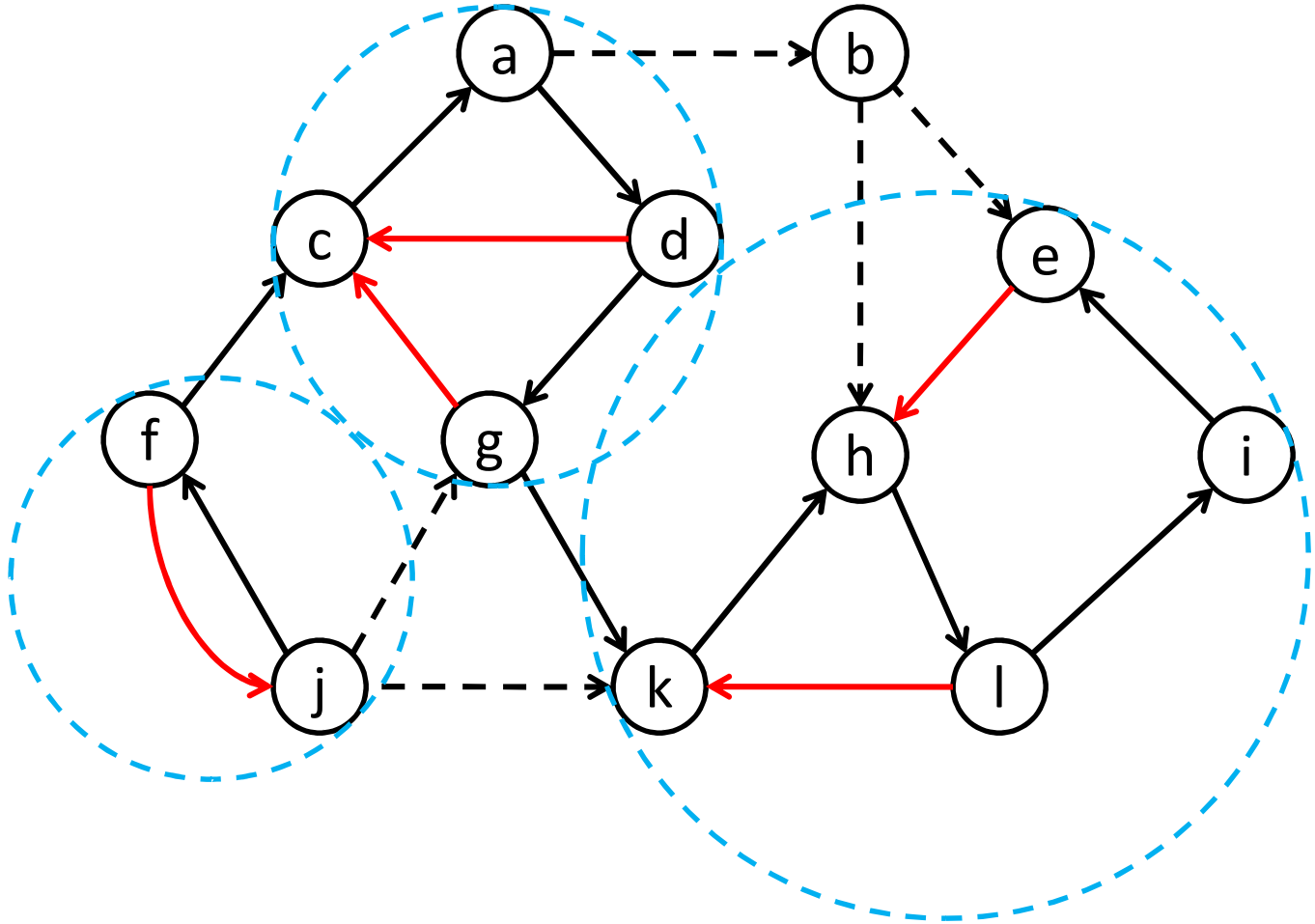
Current path j’, c’’, k, h, l, i, e

Traversal of (e, h) contracts e, i, l, h into h’



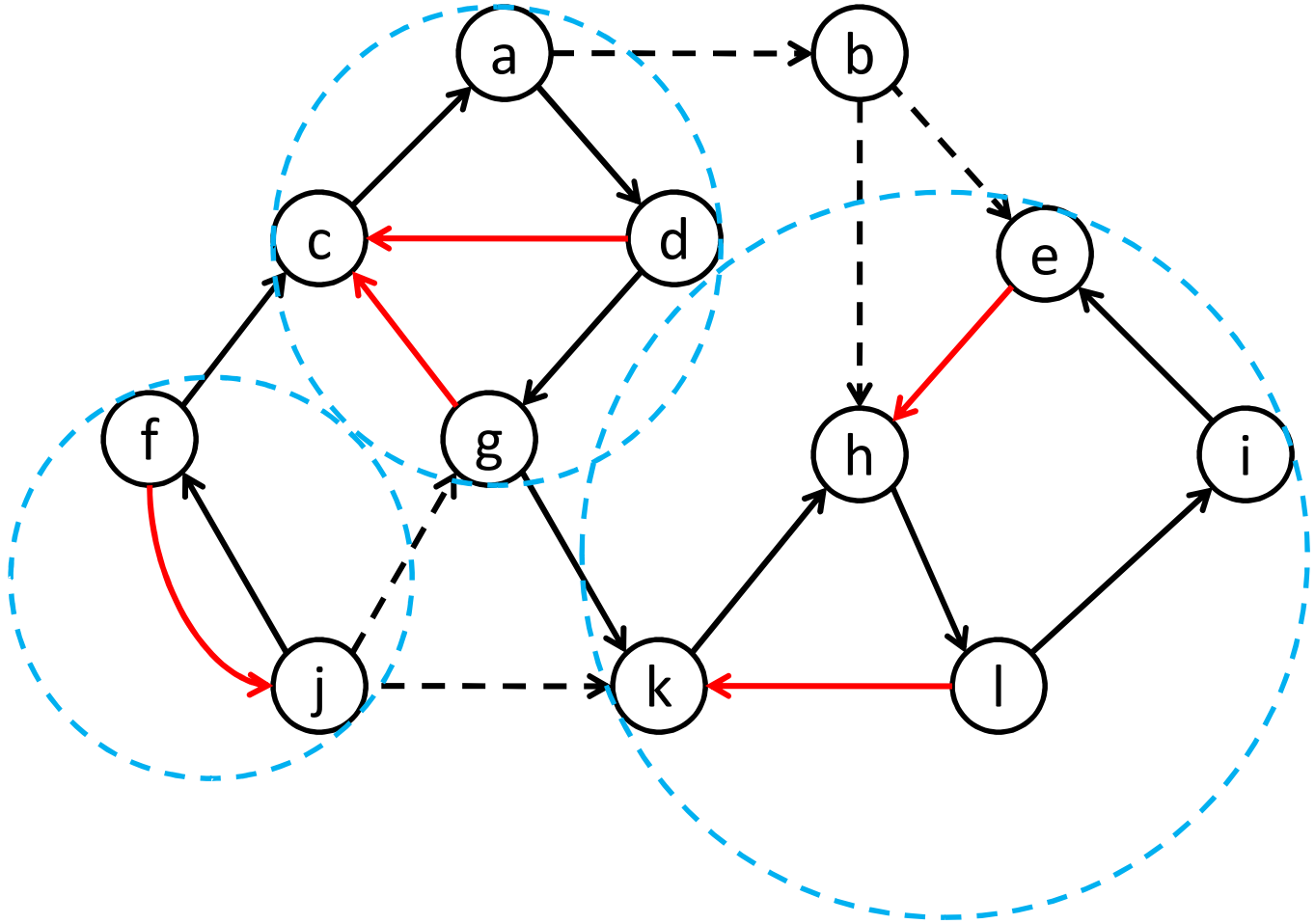
Current path j’, c’’, k, h’

Traversal of (I, k) contracts h’, k into k’



Current path j’, c’’, k’

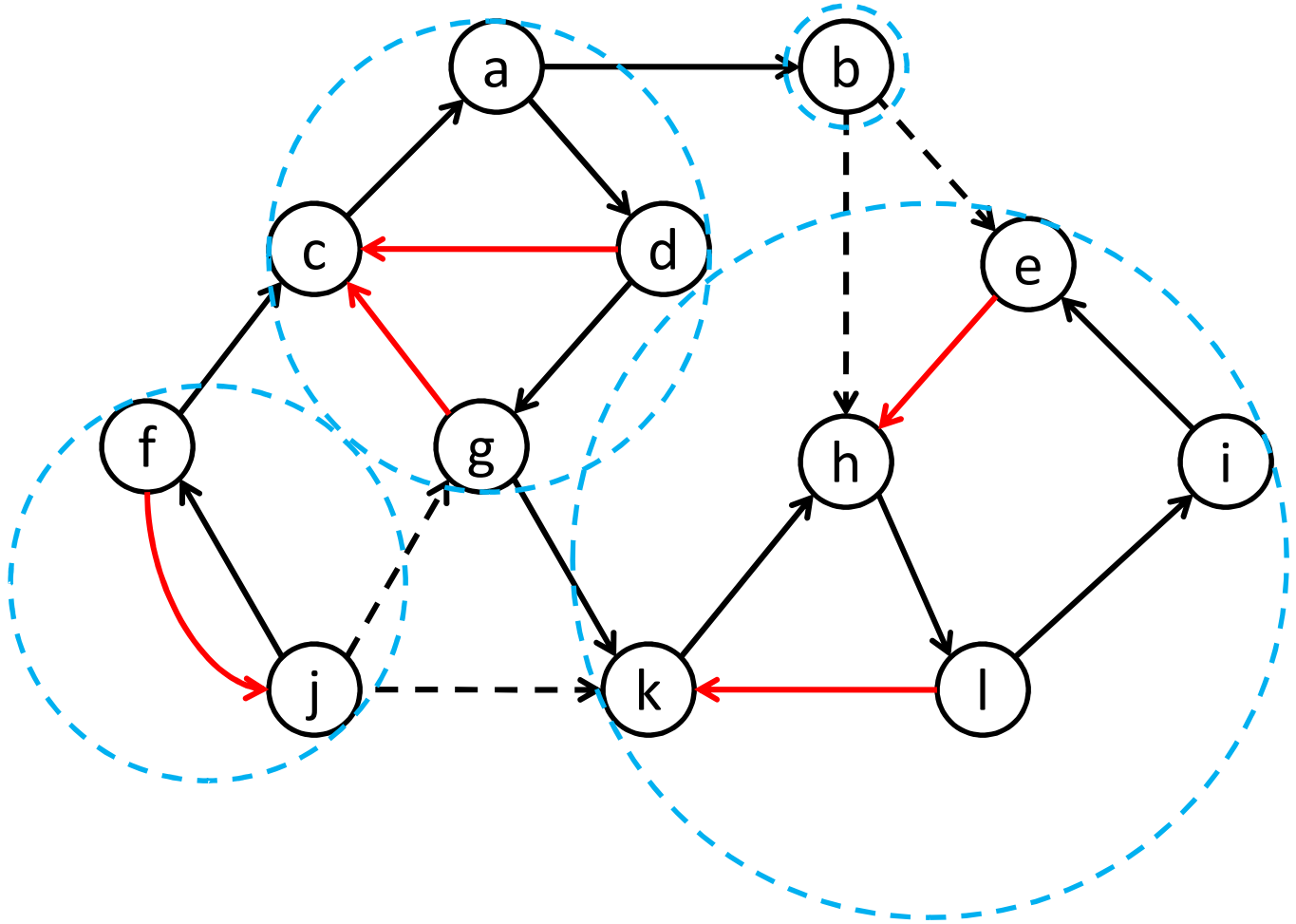
Postvisit of k’ gives component {e, i, l, h, k}



Current path j’, c’’, b

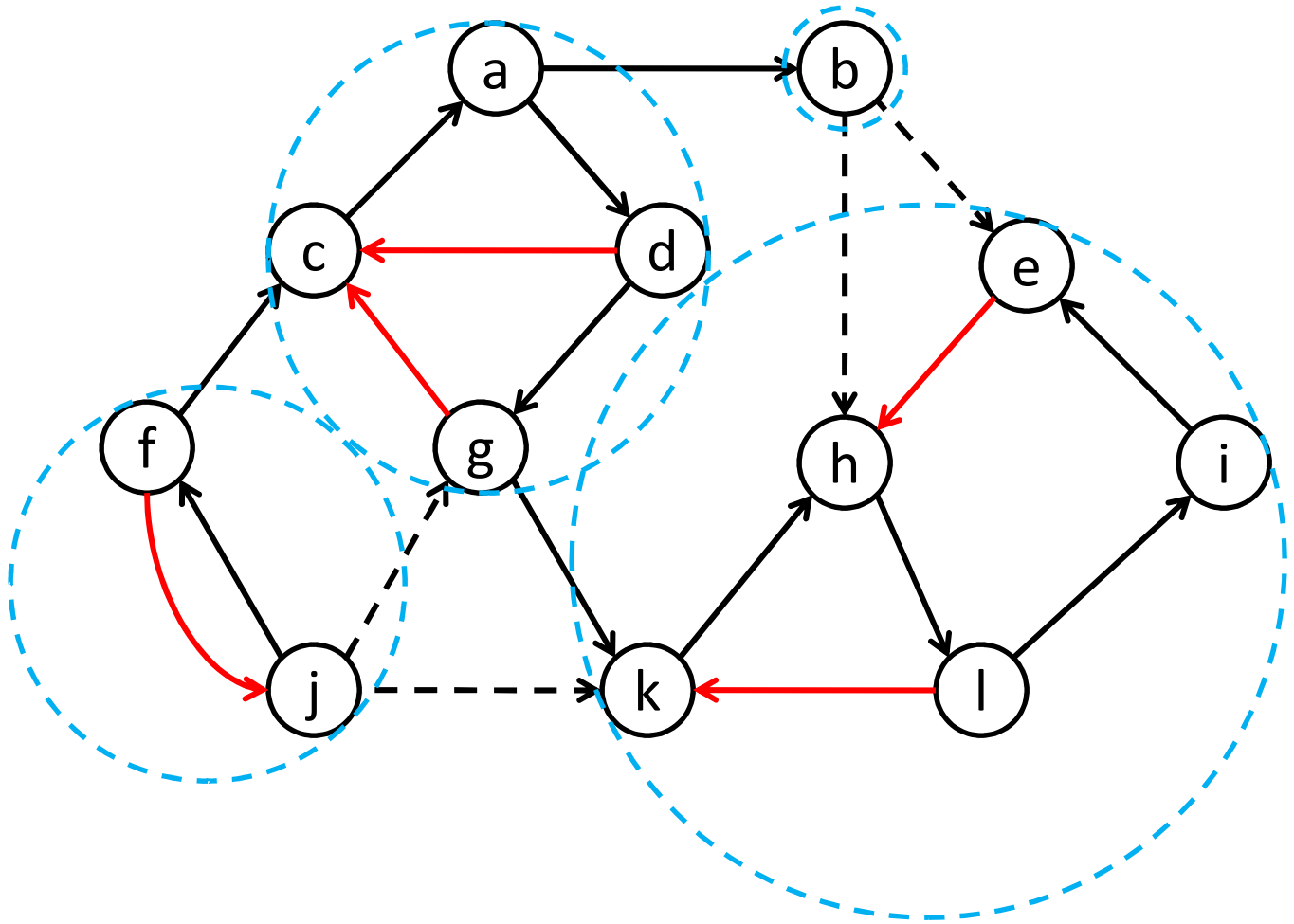
Traversal of (b, e), (b, k) leads to component k’

Postvisit of b gives component {b}



Current path j’, c’’

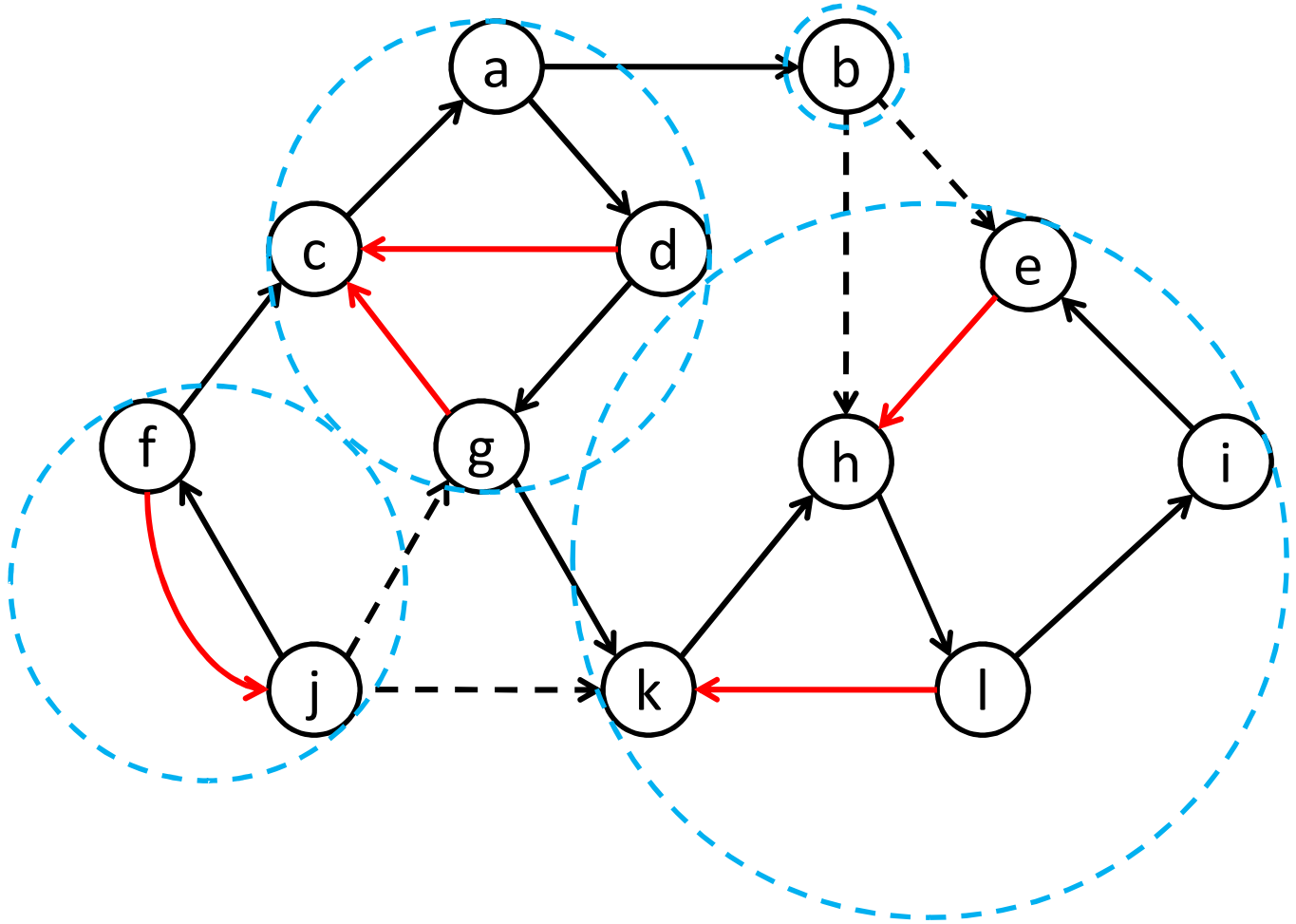
Postvisit of c’’ gives component {g, d, a, c}



Current path j’

Traversal of (j, g) leads to c’’, of (j, k) leads to k’

Postvisit of j’ gives component {f, j}



From almost-linear time to linear time

Special case of set union

Links and finds are *not* arbitrary: active sets form a stack (current path *P*)

Number vertices in preorder, order by number. In each set, choose smallest vertex as root

Store component roots on a stack *R*, in increasing order = order of components on *P*

w in a set → *find*(w) = max{$x \leq_{pre} w \mid x$ on R}: find top x on R with $x \leq_{pre} w$

To combine sets: pop from R all vertices above x

The pops pay for the find!

Can make the find implicit!

Need a way to test whether a vertex is in an active set, a way to form components

Number components consecutively from $n + 1$.
To form a component numbered c, set $pre(v) = c$ for all vertices v in component: vertices in components are larger than those in active sets

To keep track of non-roots in active sets, use a second stack S. When x is postvisited, push it on S. All postvisited vertices in a set are adjacent on S, order of sets on S is same as order on P. To form a component, when v is postvisited and on top of R, pop v from R and all x with $x >_{pre} v$ from S

Implementation

explore(*V*, *E*):

{ $S \leftarrow [\,]$; $P \leftarrow [\,]$; $k \leftarrow 0$; $c \leftarrow n$;

for $v \in V$ **do** *pre*(v) $\leftarrow 0$;

for $v \in V$ **do if** *pre*(v) = 0 **then** *search*(v)}

search(*v*):

{$k \leftarrow k + 1$; *pre*(*v*) $\leftarrow$ *k*; *push*(*v*, *R*);

for (*v*, *w*) $\in$ *E* **do**

if *pre*(*w*) = 0 **then** *search*(*w*)

else while *top*(*R*)) $>_{pre}$ *w* **do** *pop*(*R*)

if *top*(*R*) $\neq$ *v* **then** *push*(*v*, *S*) **else**

{$c \leftarrow c + 1$;

while *top*(*S*) $>_{pre}$ *v* **do**

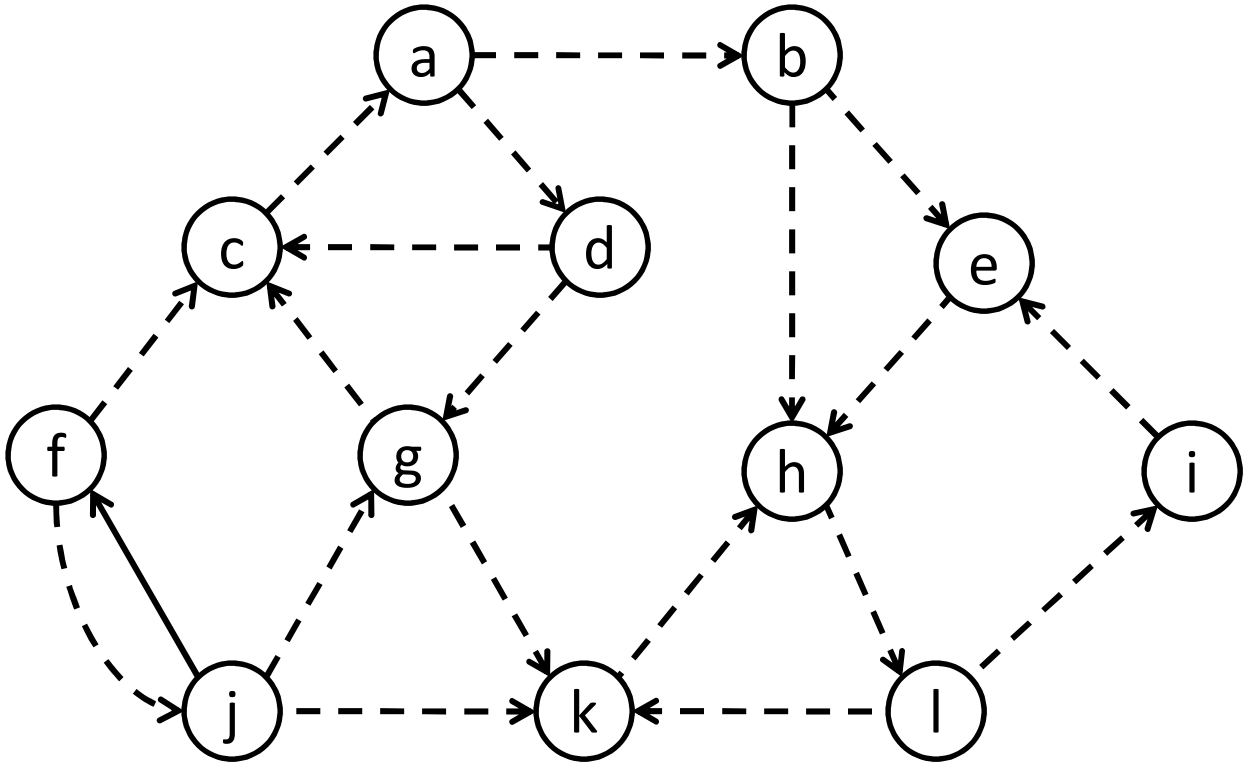
{$x \leftarrow$ *pop*(*S*); *pre*(*x*) $\leftarrow$ *c*};

pop(*R*); *pre*(*v*) $\leftarrow$ *c*}}

Search from j

R j:1, f:2

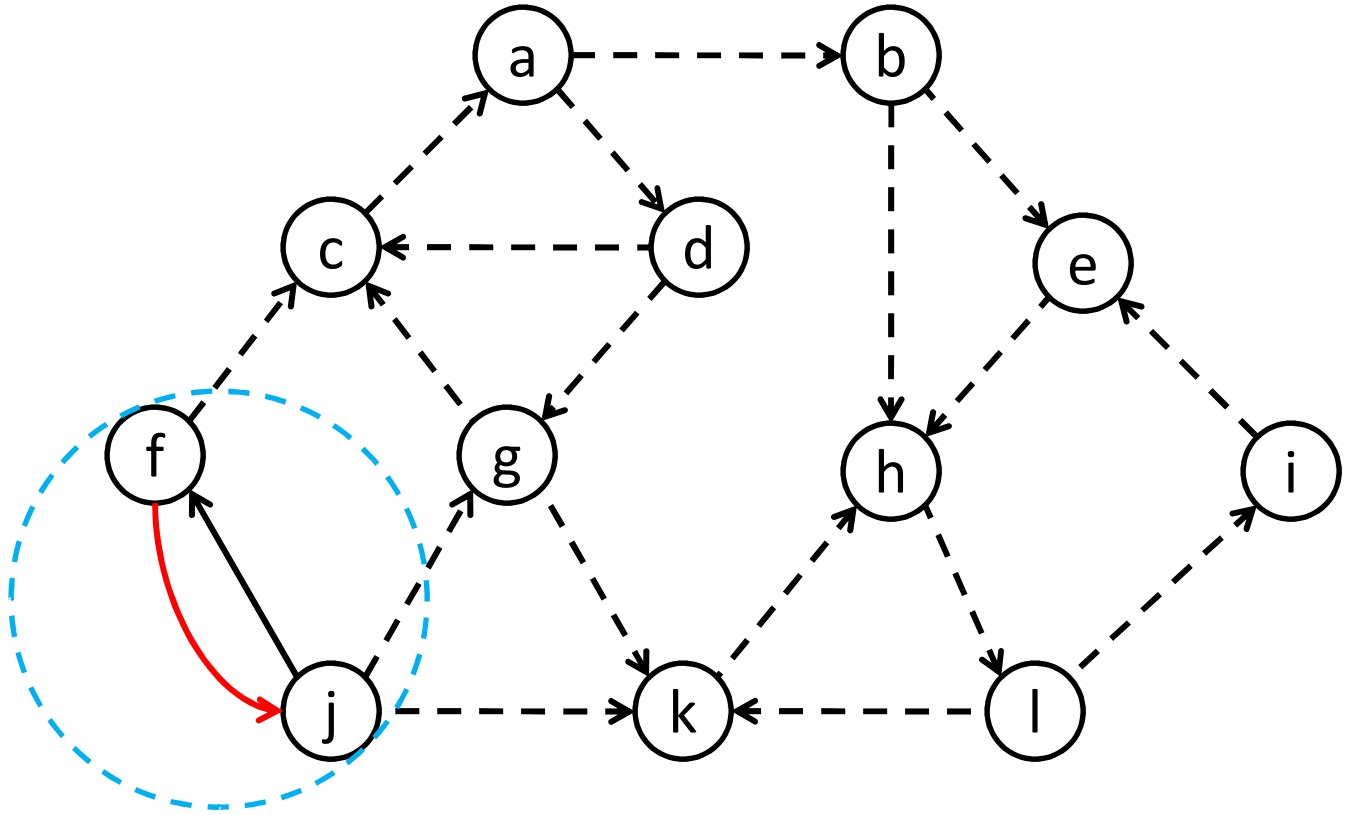
S



R j:1, f:2

S

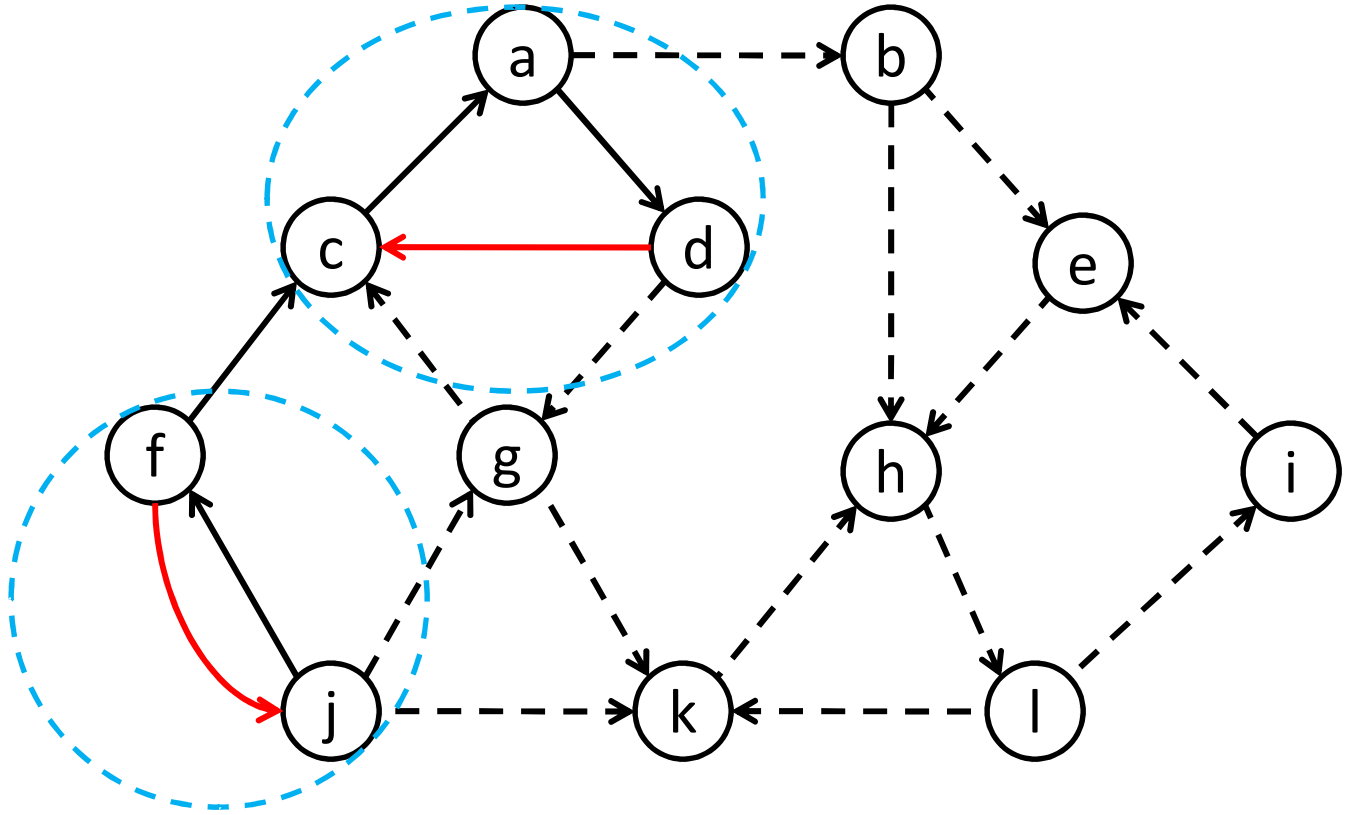
Traversal of (f, j) pops f



R j:1, c:3, a:4, d:5

S

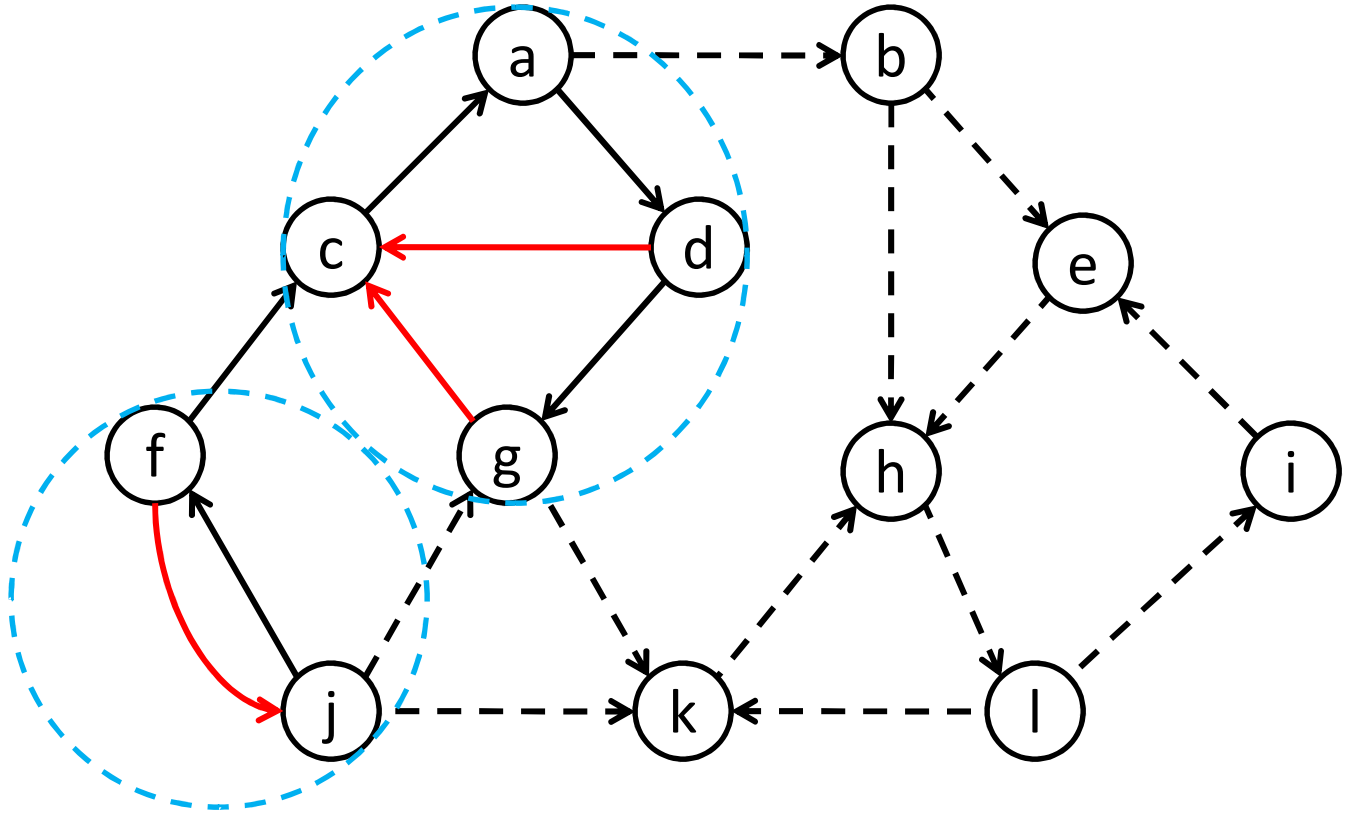
Traversal of (d, c) pops d, a



R j:1, c:3, g:6

S

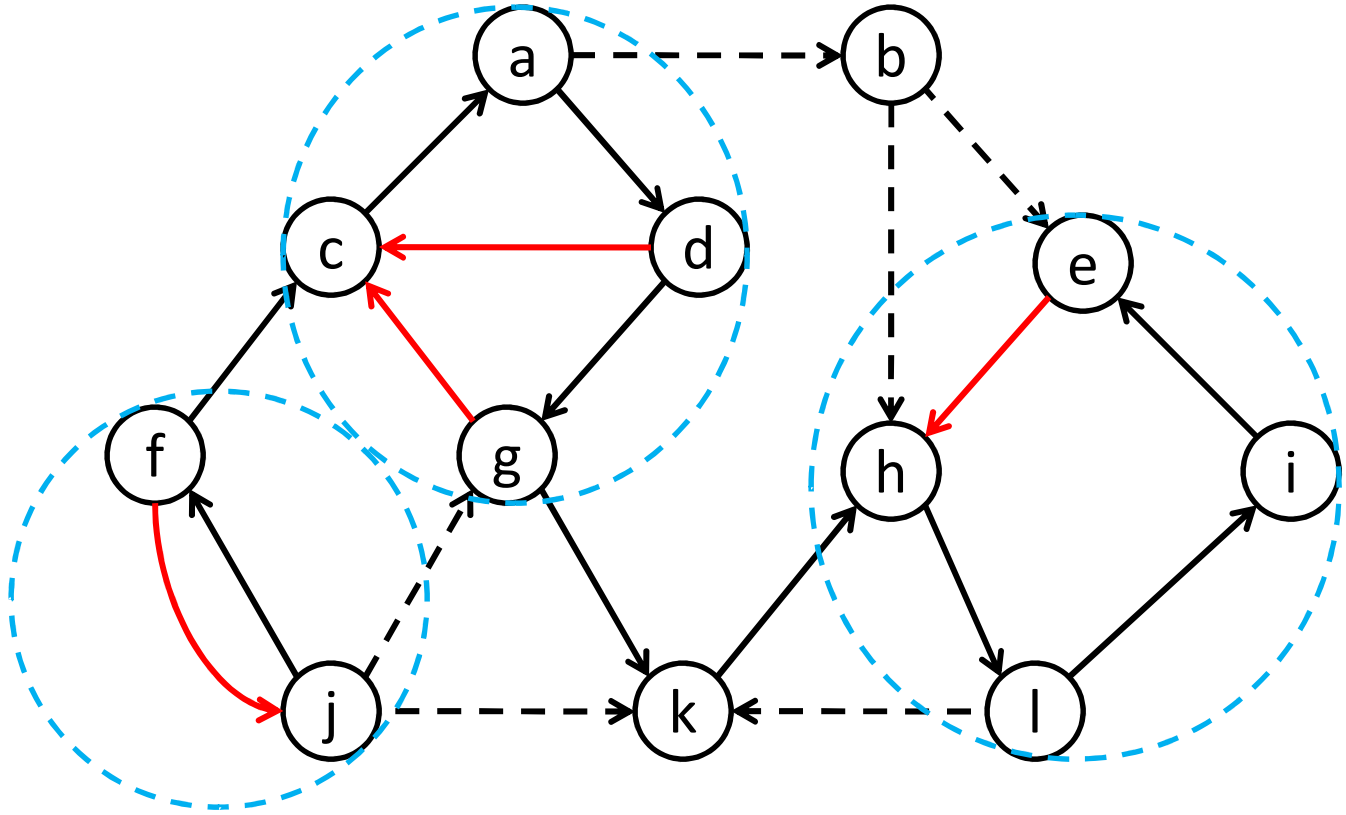
Traversal of (g, c) pops g



R j:1, c:3, k:7, h:8, l:9, i:10, e:11

S

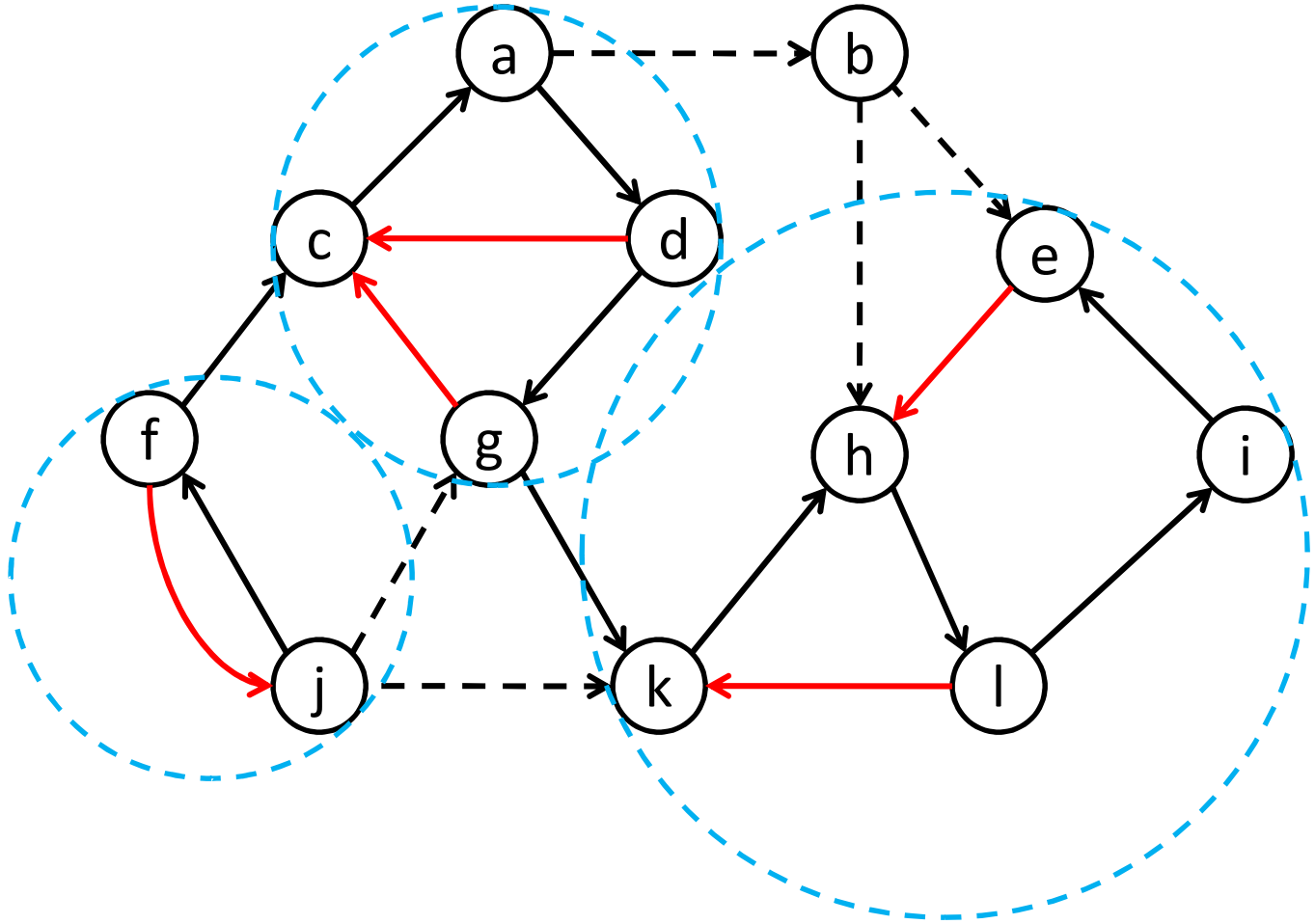
Traversal of (e, h) pops e, i, l



R j:1, c:3, k:7, h:8

S e:11, i:10

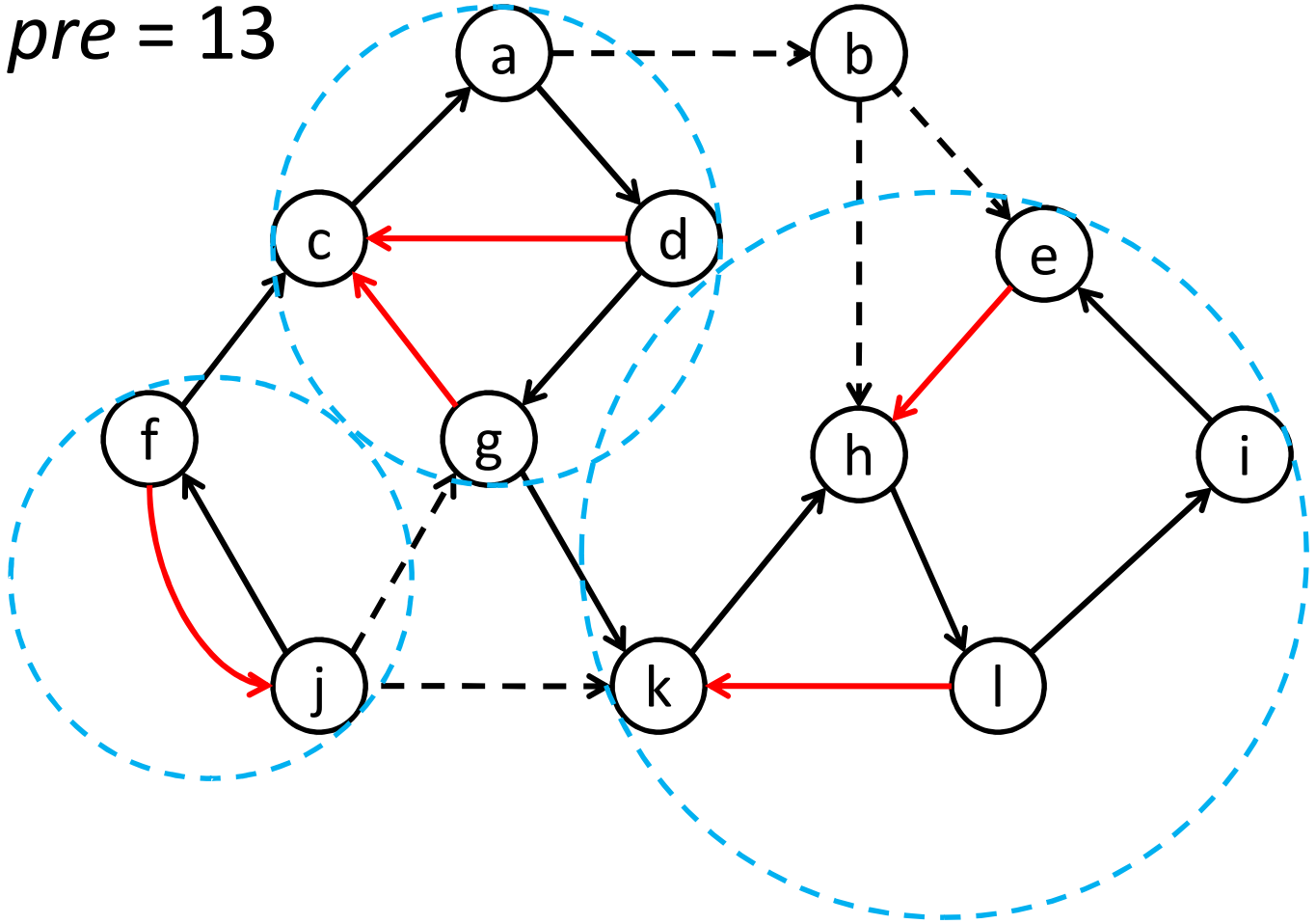
Traversal of (I, k) pops h



R j:1, c:3, k:7

S e:11, i:10, l:9, h:8

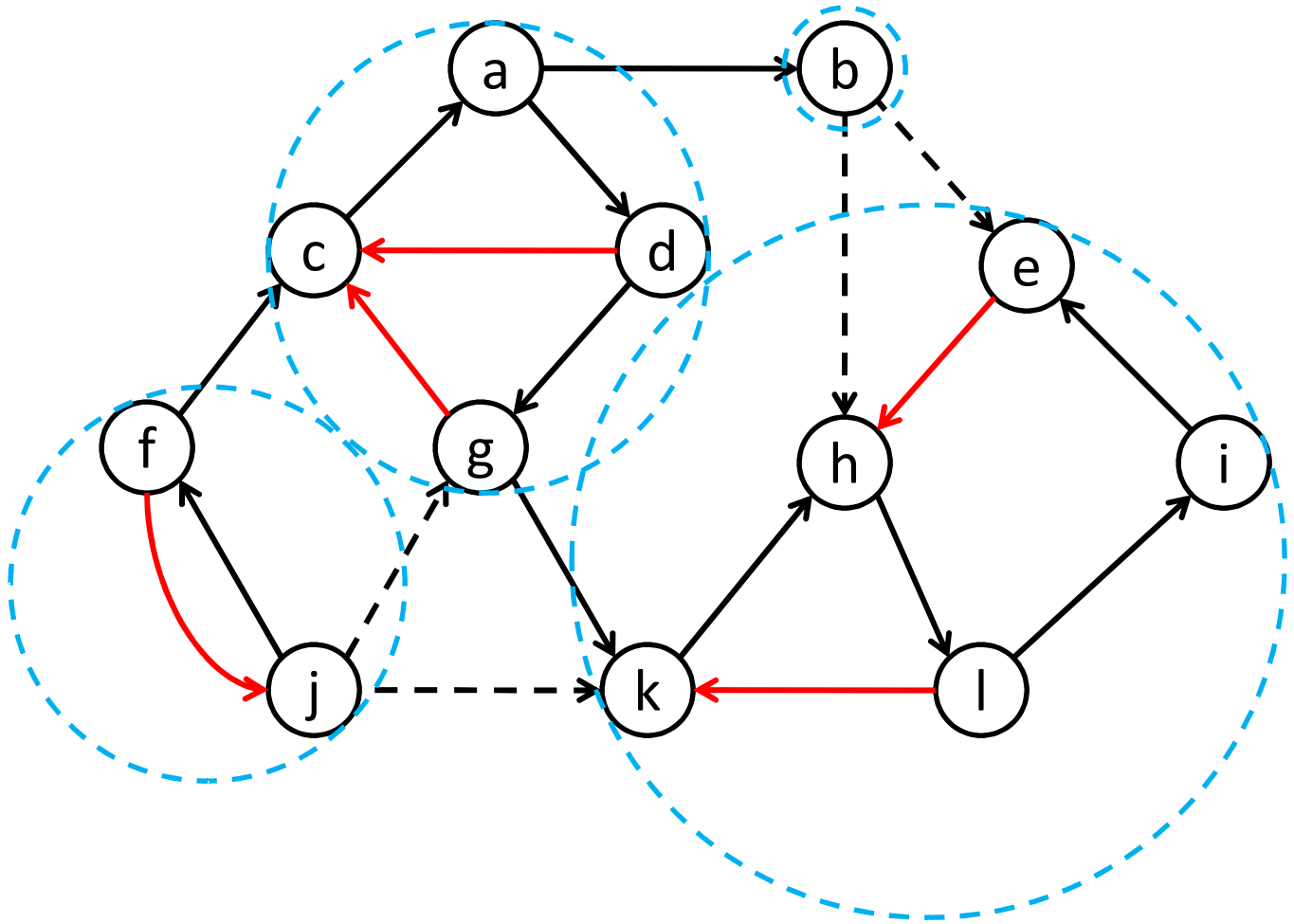
Postvisit of k gives component {h, l, i, e, k}; all get *pre* = 13



R j:1, c:3, b:12

S g:6, d:5

Postvisit of b gives component {b}, *pre* = 14

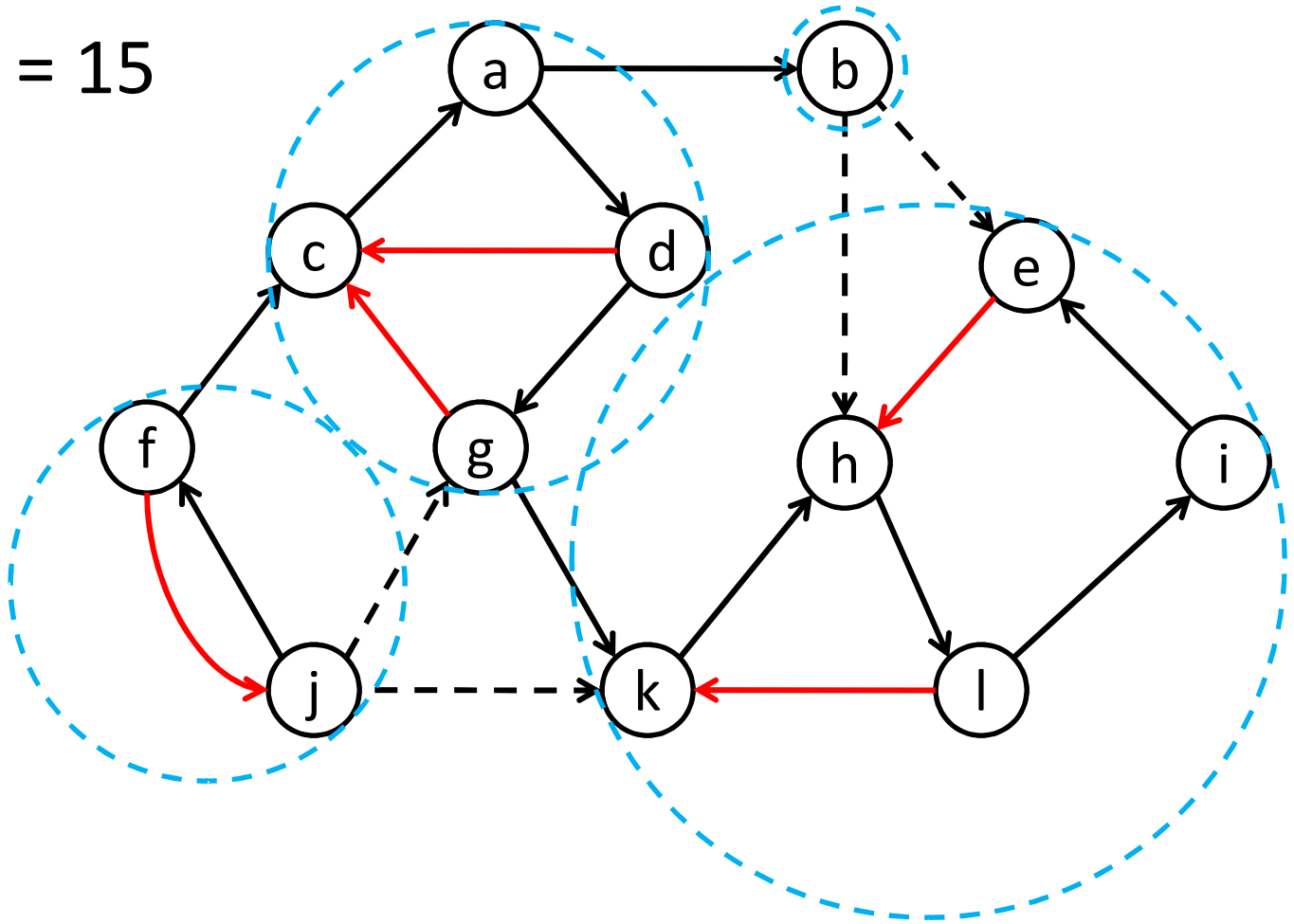


R j:1, c:3

S g:6, d:5, a:4

Postvisit of c gives component {a, d, g, c},

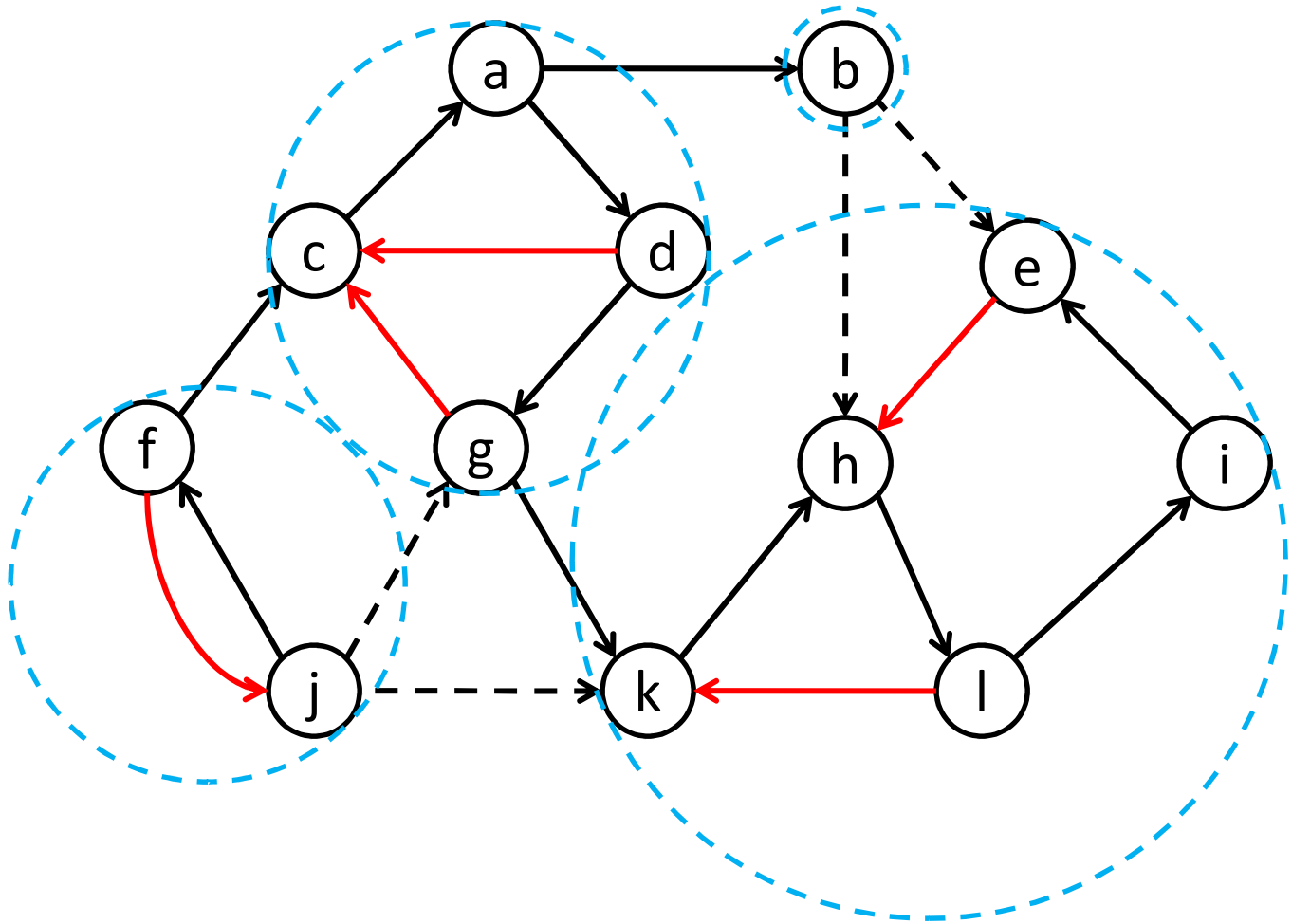
pre = 15



R j:1

S f:2

Postvisit of j gives component {f, j}, *pre* = 16



Correctness proof: The algorithm maintains the following invariants:

Each traversed arc either has both ends in the same active set, or leads from one set to the next-higher one, or leads to a vertex already in a component

Each set of vertices on the postvisit stack between adjacent roots on the root stack is strongly connected with the smaller bounding root

Each set of popped vertices forms a component

A variant and a question

Eliminate postvisit stack. Form a component by searching from its root, adding to the new component each visited vertex not yet in a component

Is there a simple way to choose search start vertices such that each top-level search will span a component?

Yes, if the component-spanning searches are *backward*

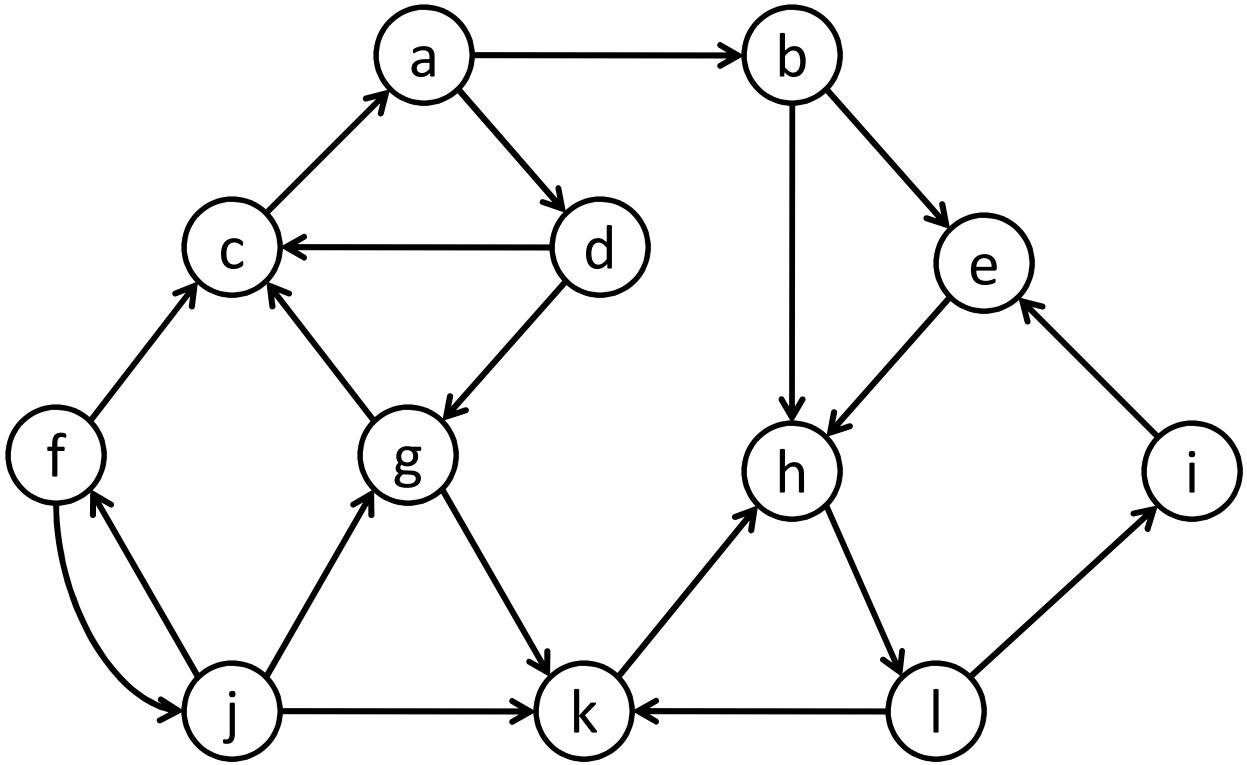
Two-way algorithm (Kosaraju 1978, Sharir 1981)

Do a forward depth-first exploration, ordering the vertices in reverse postorder.

Do a backward exploration, choosing start vertices in the order generated by the forward exploration.

(Or, equivalently, do the first exploration backward and the second one forward.)

Each search done by the second exploration spans a strong component.


a
b
c
d
e
f
g
h
i
j
k
l

preorder: a, b, e, h, l, i, k, d, c, g; f, j

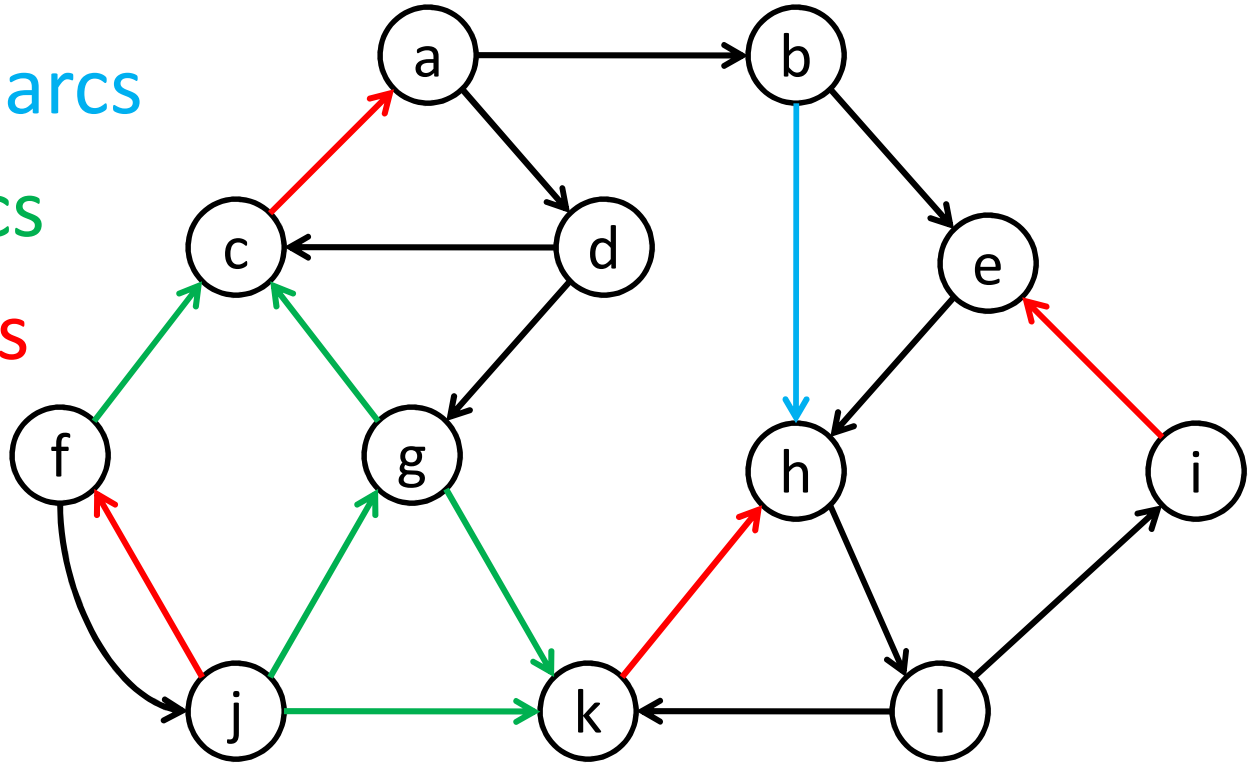
postorder: i, k, l, h, e, b, c, g, d, a; j, f

tree arcs

forward arcs

cross arcs

back arcs



postorder: i, k, l, h, e, b, c, g, d, a; j, f

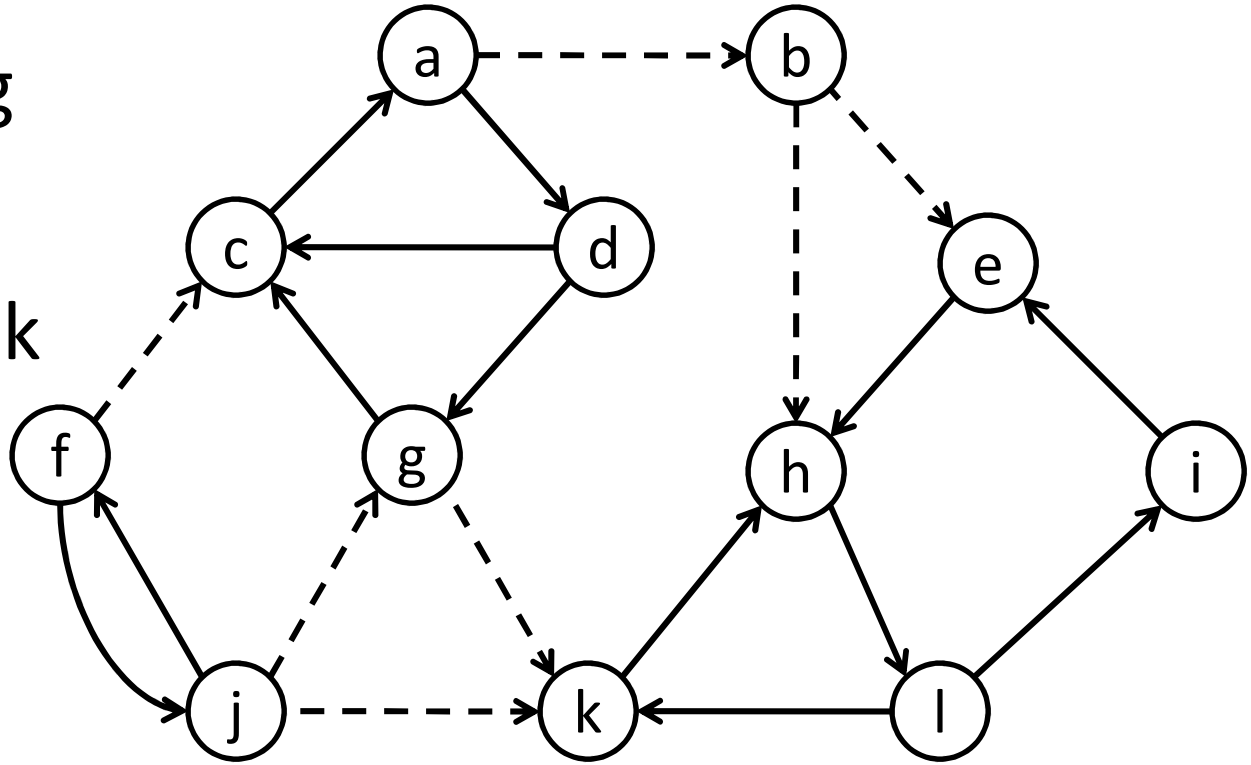
backward searches:

f, j

a, c, d, g

b

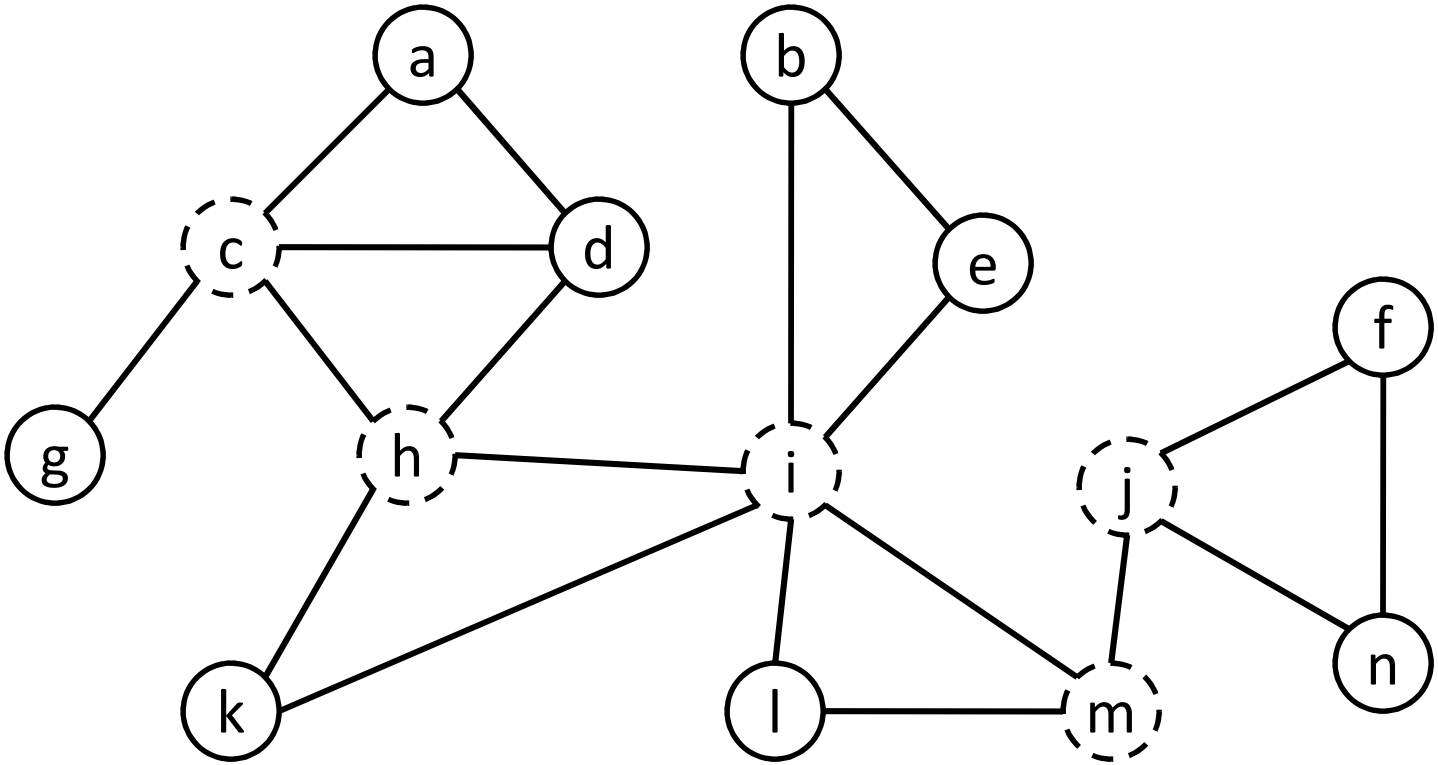
e, i, l, h, k



Correctness Proof: By induction on the components in order by their smallest vertices. Let *u* be the smallest vertex in a component *C*. Suppose the components with smallest vertices smaller than *u* have been correctly generated by previous searches. The next search will start from *u*, and when it starts, all vertices in *C* are unvisited. Thus the search will visit all vertices in *C*.

Correctness Proof (cont.): Let $x \neq w$ be visited by the search. Then x is larger than w, since when the search started w was the smallest unvisited vertex. By the postorder lemma, x is a descendant of u, which implies that x is in C. Thus the search from u visits precisely the vertices in C.

Blocks



One-way algorithm (simplifies Gabow 2000)

Use depth-first exploration. Blocks are like strong components but partition edges, not vertices: think of current path as a list of tree arcs, not as a list of vertices. When traversing a cycle arc, condense all tree arcs on the corresponding cycle into a single "super-arc." When retreating along a tree arc or super-arc, form the corresponding component.

Details

Number vertices in preorder and order vertices by number. Maintain two stacks *R* and *S*, the former storing the first-traversed arc in each tentative block (the root of the corresponding set), the latter storing all tree arcs retreated along but not yet in blocks. On *R* represent each tree arc by its tail; on *S*, by its head.

Number each block formed, starting from $n + 1$. When a tree arc with head x is added to block c, set $pre(x) = c$. When a search started at v finishes, set $pre(v) = 2n$. When all done, each edge (v, w) is in block $\min\{pre(v), pre(w)\}$

Implementation

explore(*V*, *E*):

{ $R \leftarrow [\,]$; $S \leftarrow [\,]$; $k \leftarrow 0$; $c \leftarrow n$;

for $v \in V$ **do** *pre*(*v*) $\leftarrow 0$;

for $v \in V$ **do if** *pre*(*v*) = 0 **then**

{*search*(*v*); *pre*(*v*) $\leftarrow 2n$}}

search(*v*):

{$k \leftarrow k + 1$; $pre(v) \leftarrow k$;

for $(v, w) \in E$ **do**

if $pre(w) = 0$ **then**

{*push*(*v*, *R*); *search*(*w*); *push*(*w*, *S*);

if $top(R) = v$ **then**

{*pop*(*R*); $c \leftarrow c + 1$;

while $top(S) \geq_{pre} w$ **do**

{$x \leftarrow pop(S)$; $pre(x) \leftarrow c$}}}

else while $top(R)) >_{pre} w$ **do** *pop*(*R*)}

Correctness proof: *Exercise*

A root of a DFS tree is a cut vertex iff it has at least two children. A non-root is a cut vertex iff a retreat along an outgoing tree arc forms a block

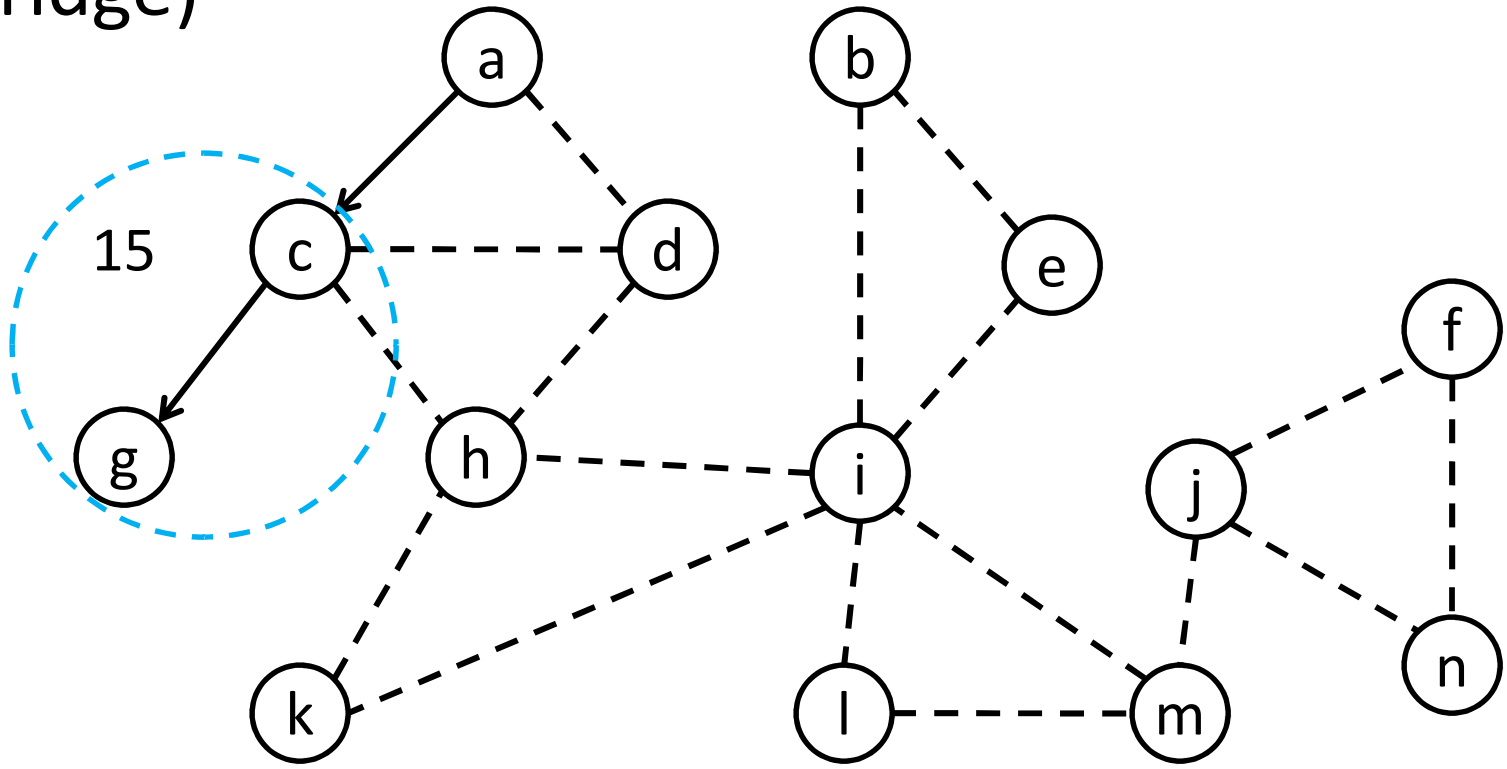
An edge is a bridge if it is in a block by itself

The bridge components are the connected components after the bridges are deleted (or can find by directing the edges during DFS and finding strong components)

Search from a

R a:1, c:2

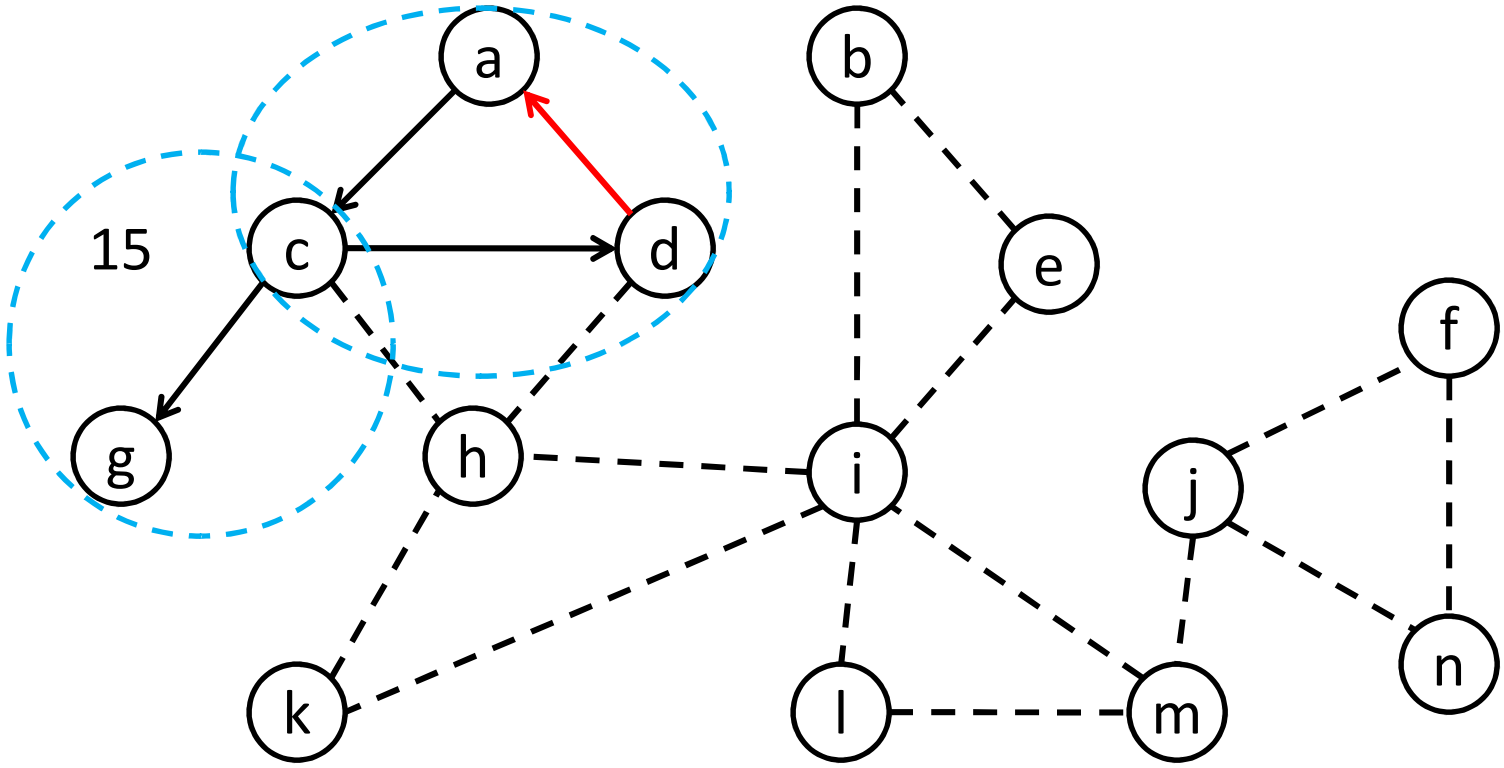
S g:3 Retreat on (c, g) forms component 15 (bridge)



R a:1, c:2

S

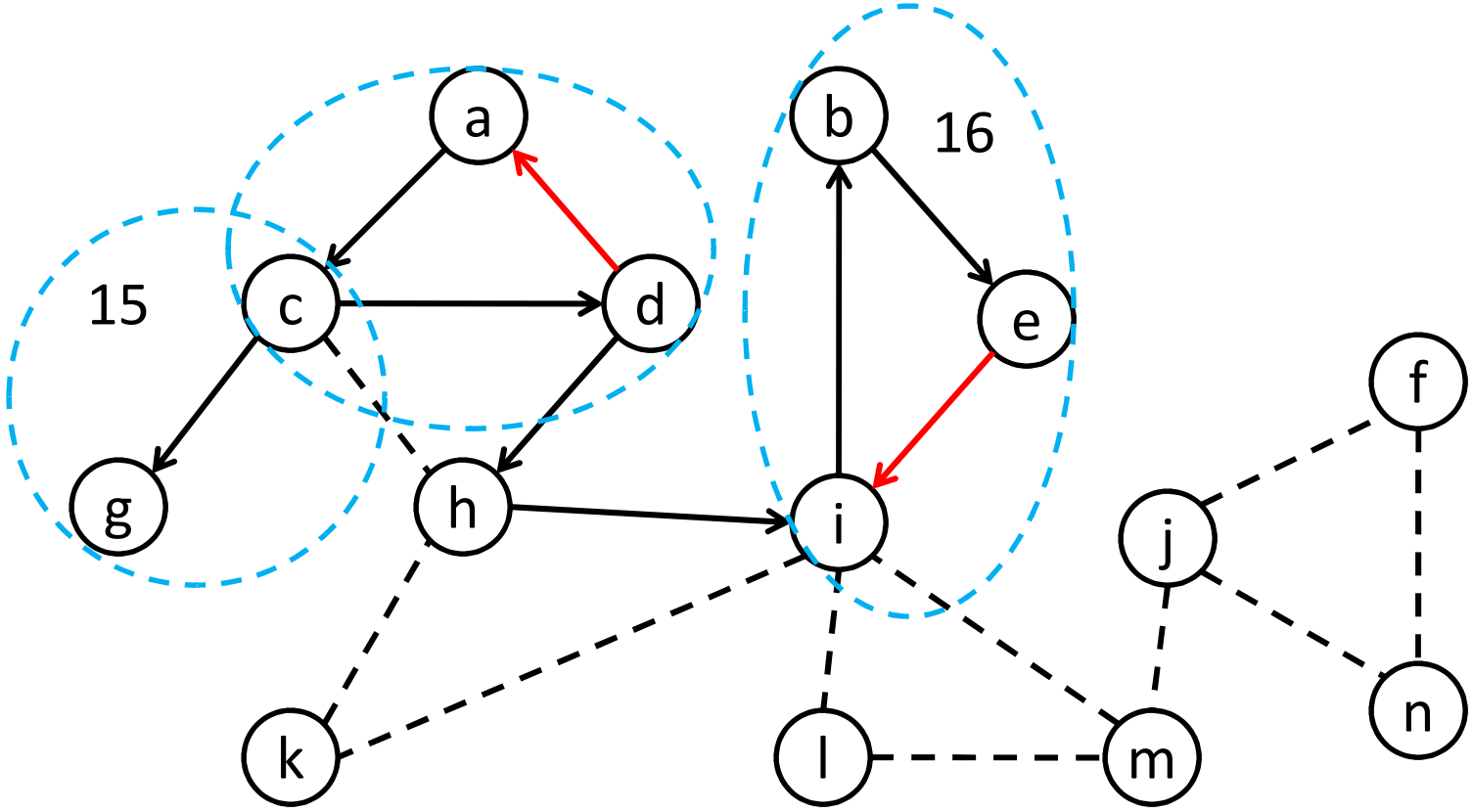
Advance on (d, a) pops c



R a:1, d:4, h:5, i:6, b:7

S

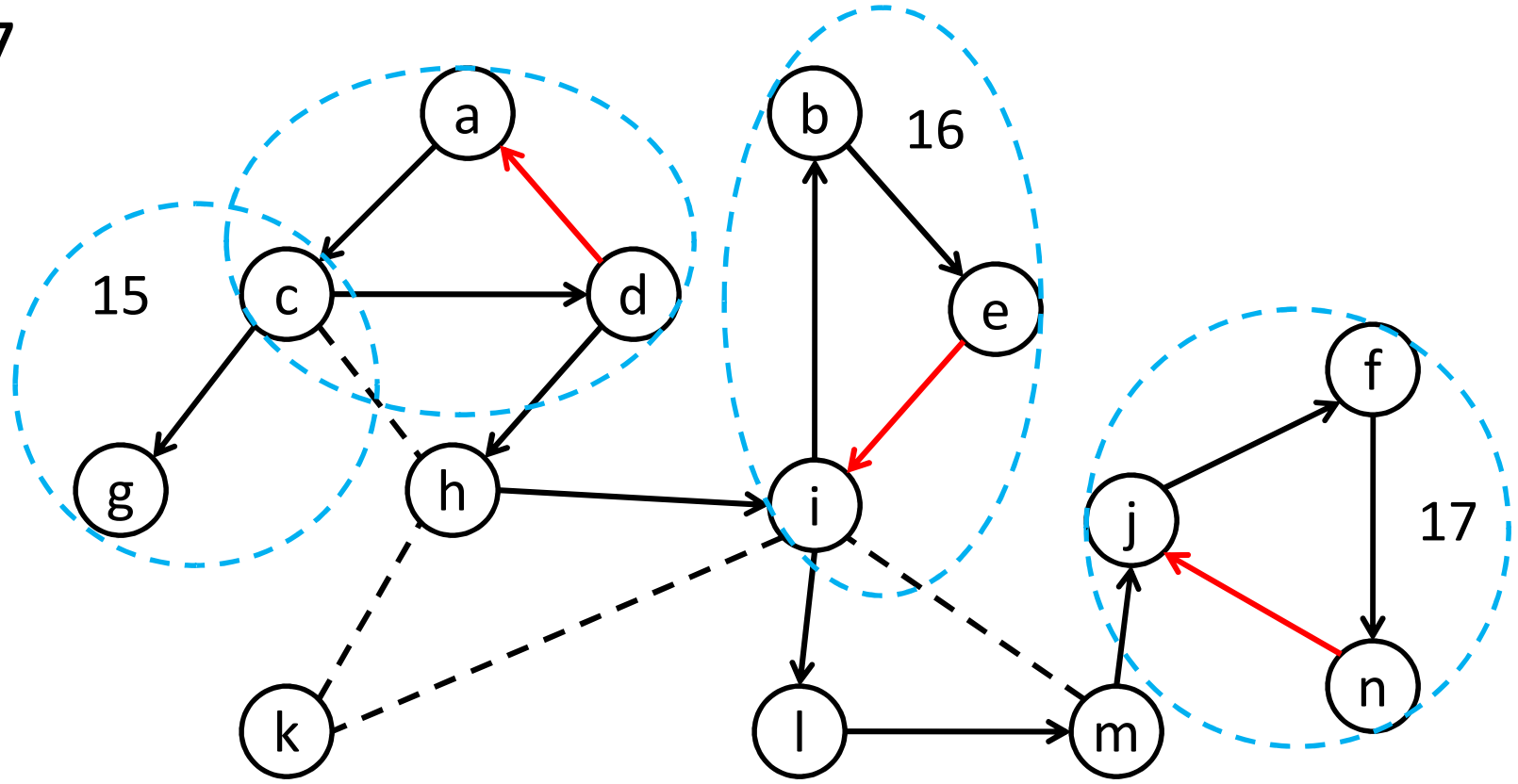
Advance on (e, i) pops b; retreat on (i, b) forms 16



R a:1, d:4, h:5, i:6, l:9, m:10, j:11, f:12

S

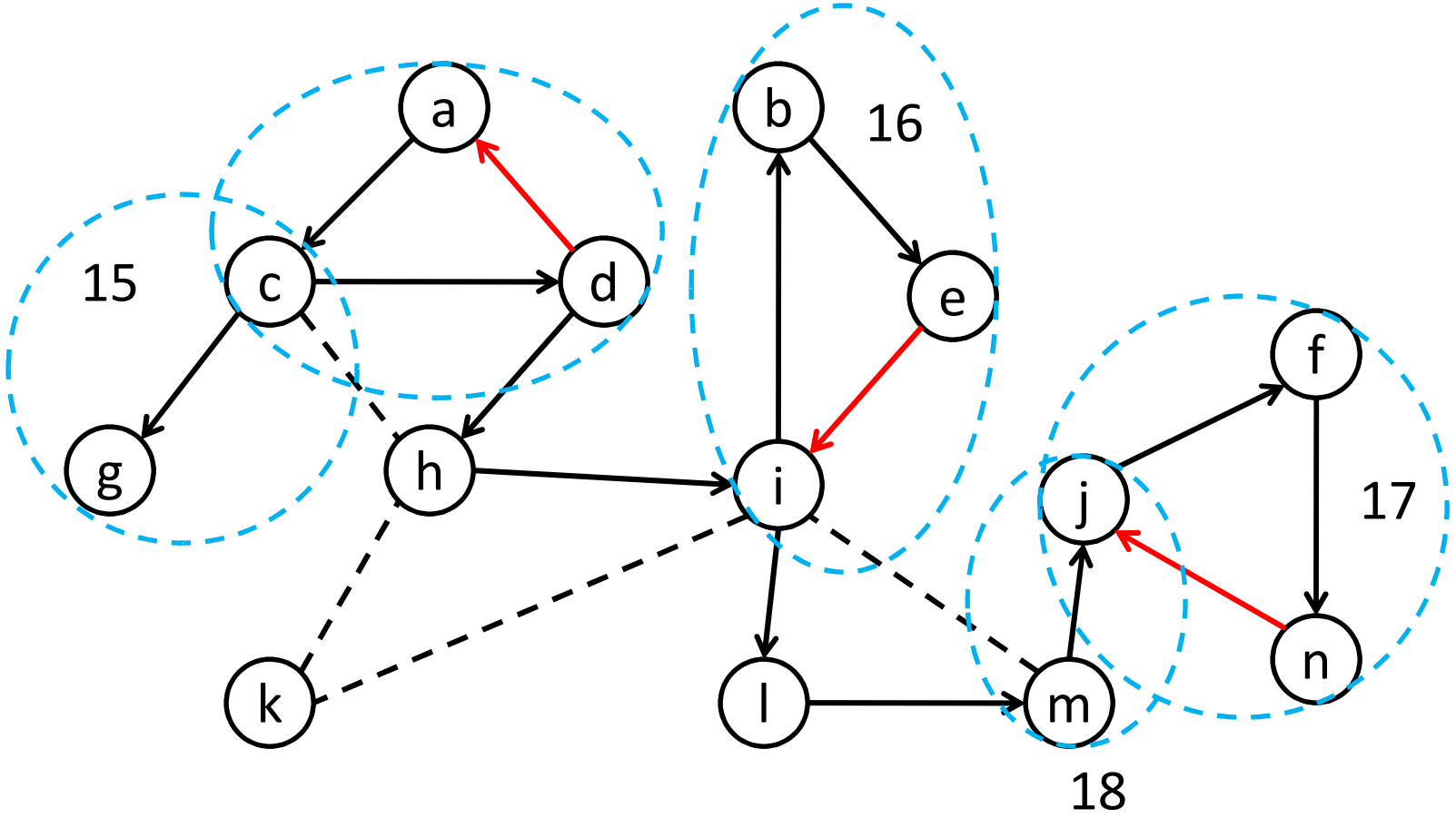
Advance on (n,j) pops f; retreat along (j, f) forms 17



R A:1, d:4, h:5, i:6, l:9, m:10

S

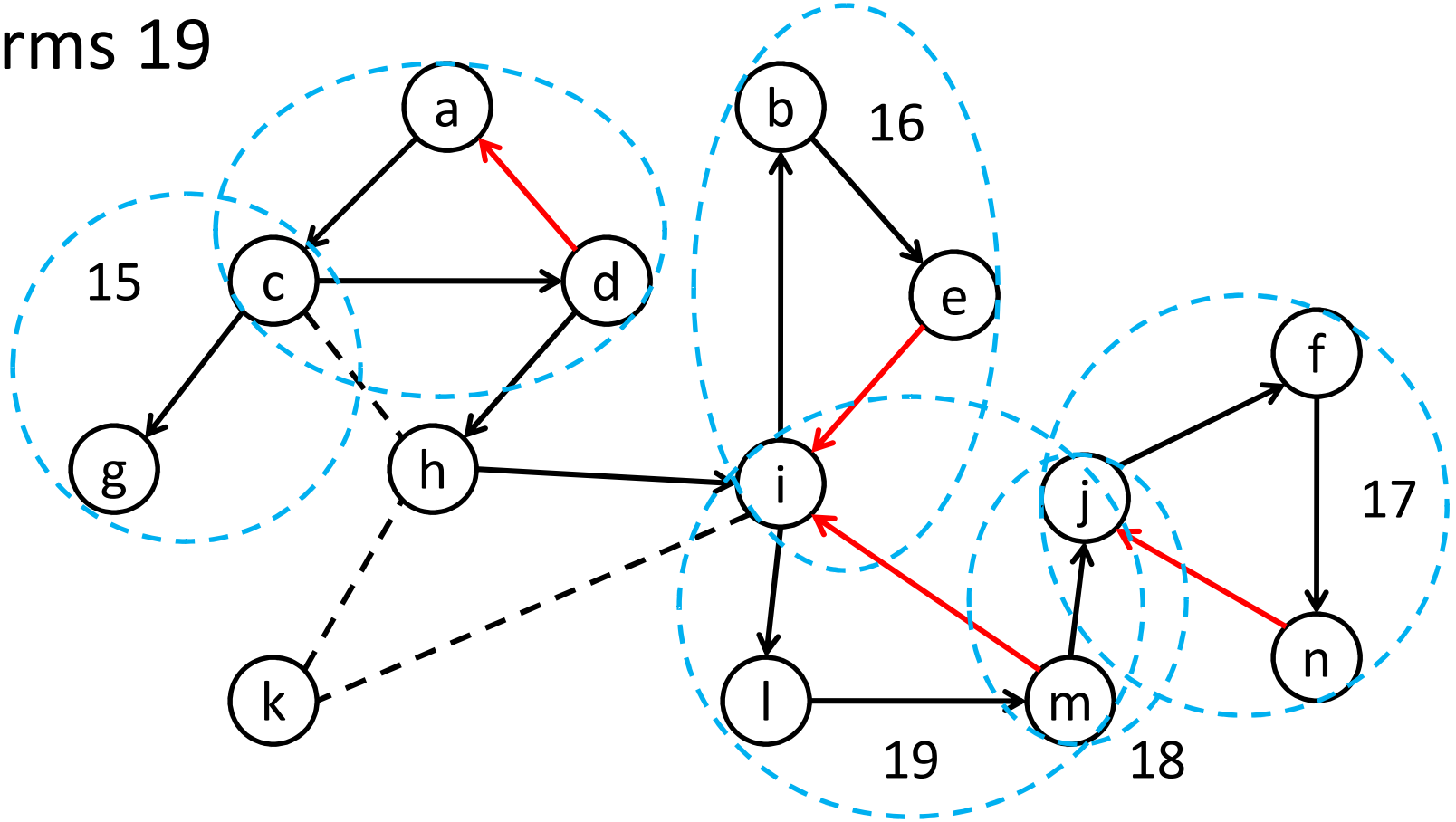
Retreat along (m, j) forms 18 (bridge)



R a:1, d:4, h:5, i:6, l:9

S

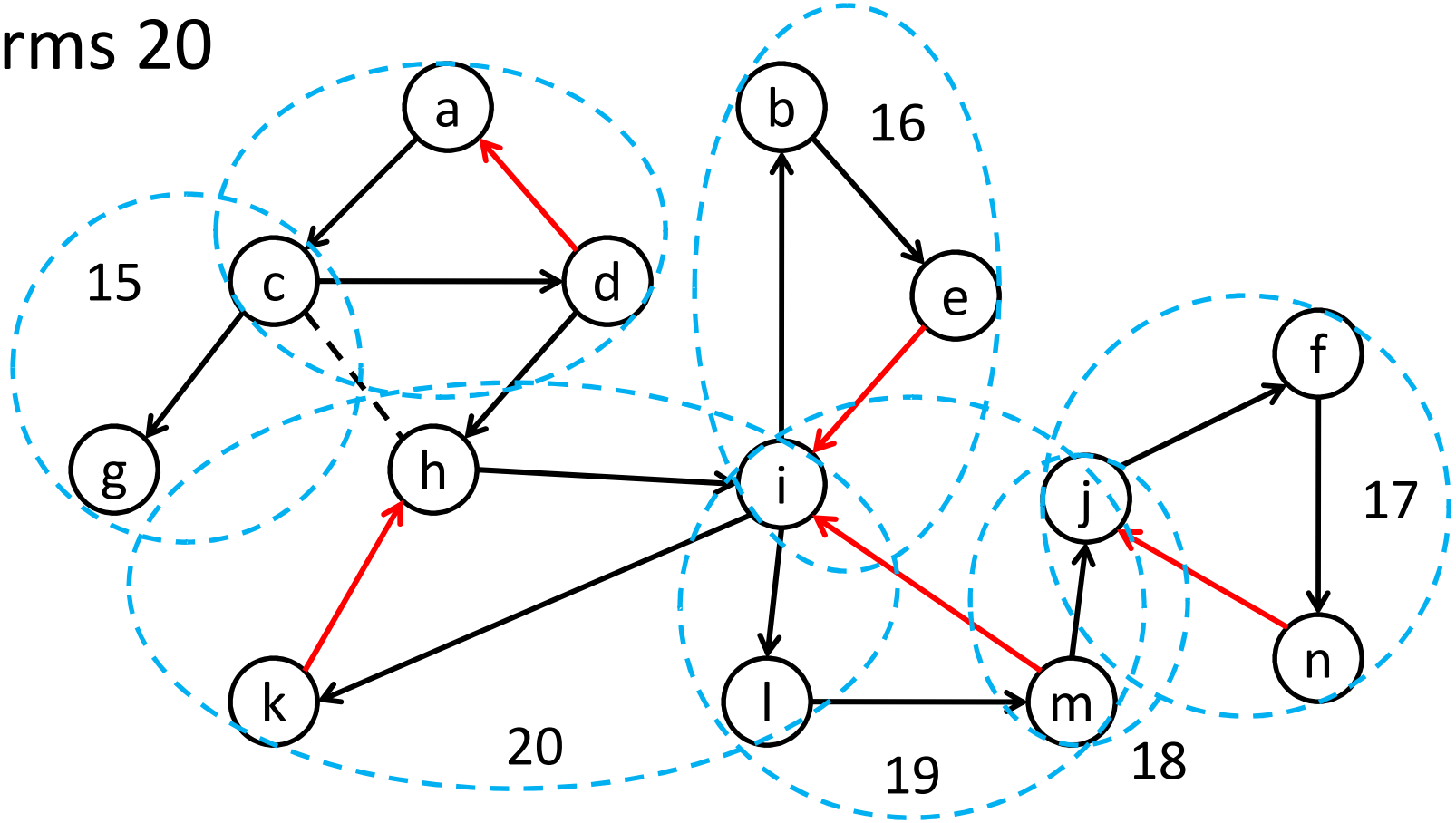
Advance along (m, i) pops l; retreat along (l, i) forms 19



R a:1, d:4, h:5, i:6

S

Advance along (k, h) pops i; retreat along (h, i) forms 20



R a:1, d:4

S

Advance along (h, c) pops d; retreat along (c, a) forms 21