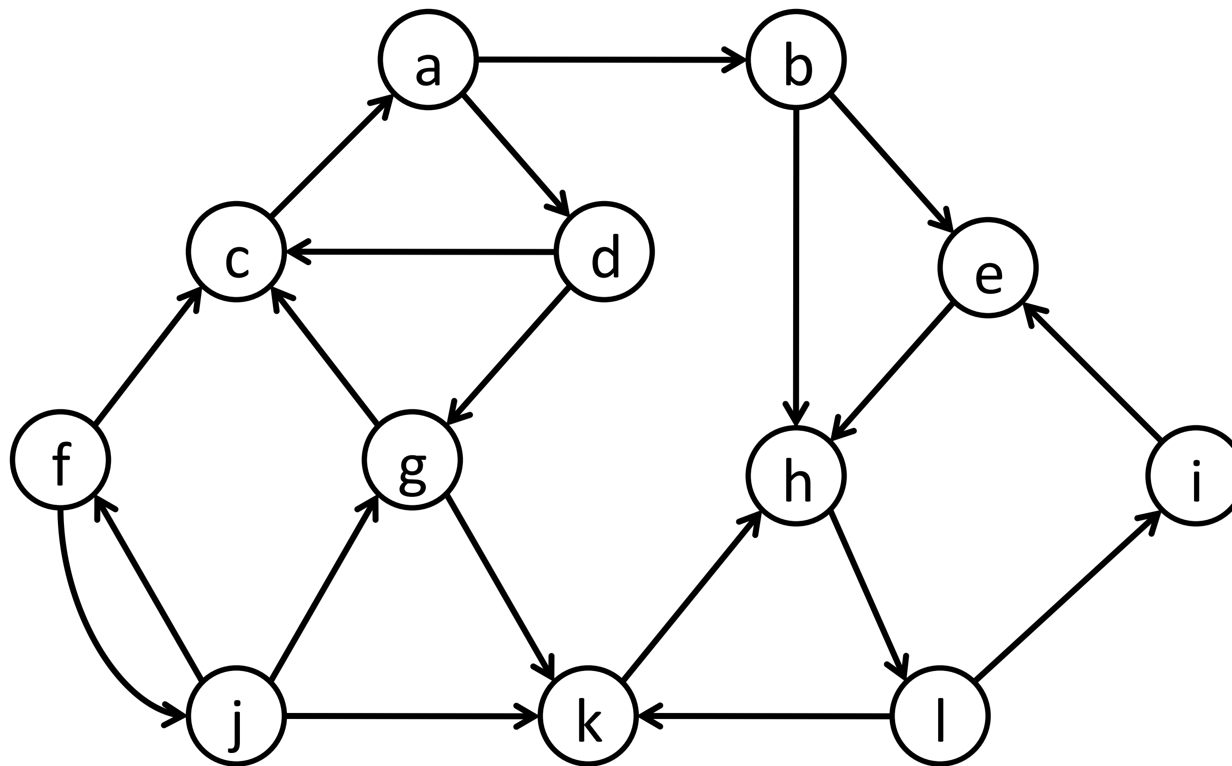
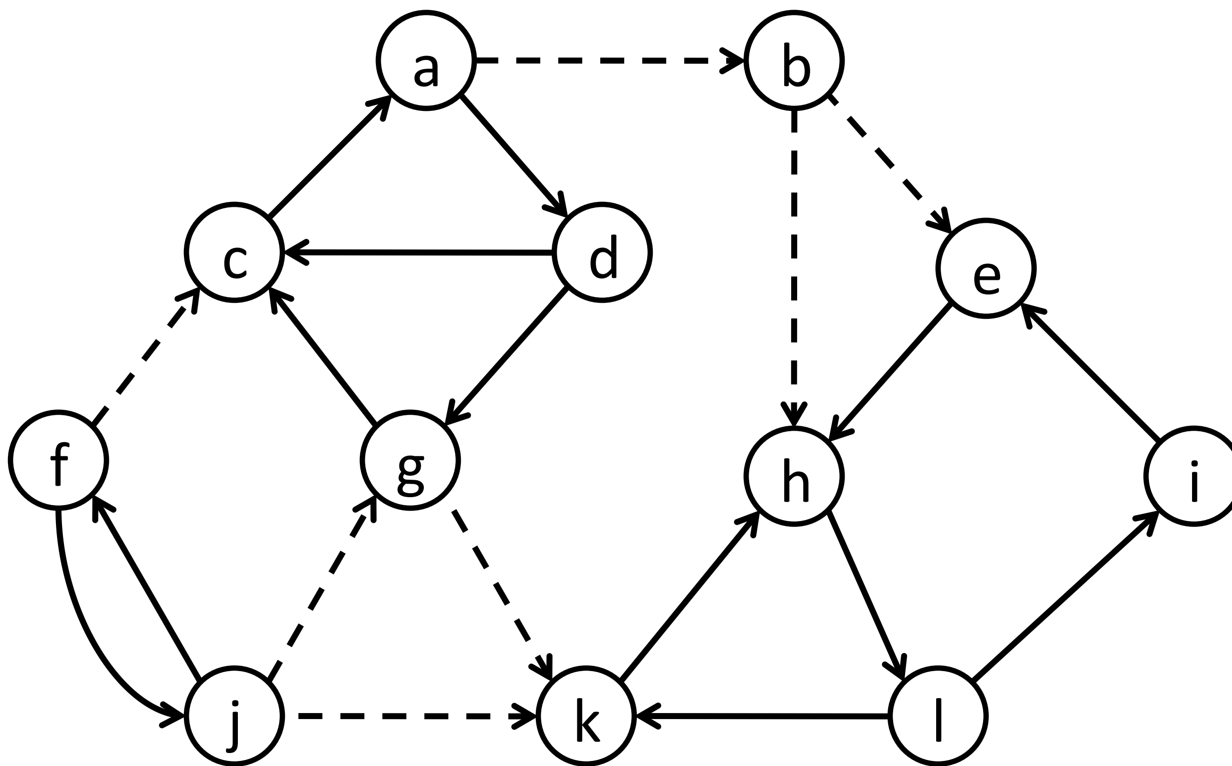
COS 528
Strong Components

© Robert E. Tarjan 2013

Two vertices v and w are *strongly connected* if there is a path from v to w and a path from w to v. Strong connectivity is an equivalence relation. A *strong component* is a subgraph induced by a maximal set of strongly connected vertices. The strong components can be *topologically ordered*: numbered so that each arc is either within a component or leads from a smaller component to a larger.

Strong components




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

One-way algorithm

Gabow 2000 via Tarjan 1971

Starting idea (Purdom 1968): Do a depth-first exploration. When traversing a back arc (to a vertex on the current path) contract the set of vertices on the corresponding cycle (back arc + tree path) into a single vertex. When postvisiting a vertex not contracted into an ancestor, list the original vertices contracted into it as a component.

Correctness: Contraction preserves strong components

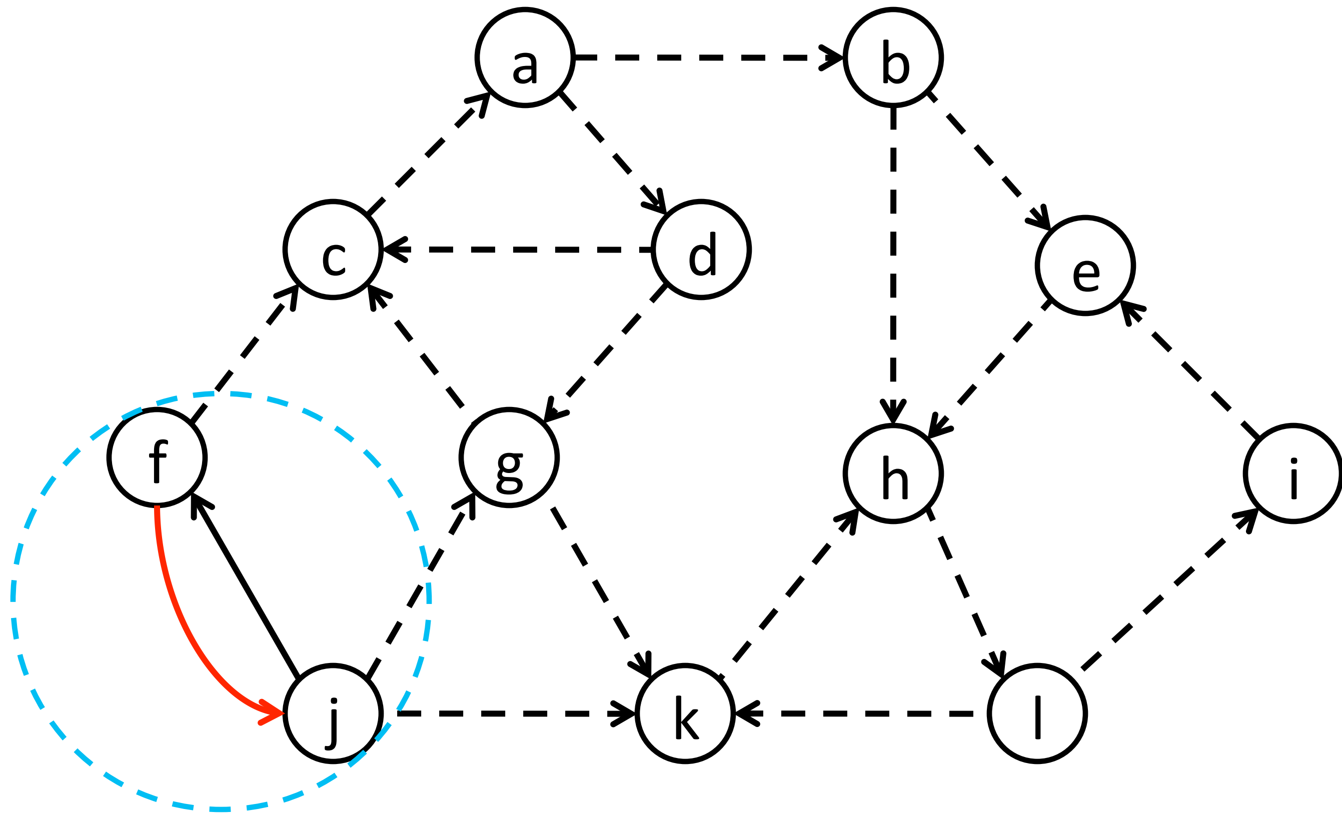
Straightforward to implement using disjoint set data structure, running time = $O(n + m\alpha(n, d))$

Components are listed in reverse topological order

Search from j

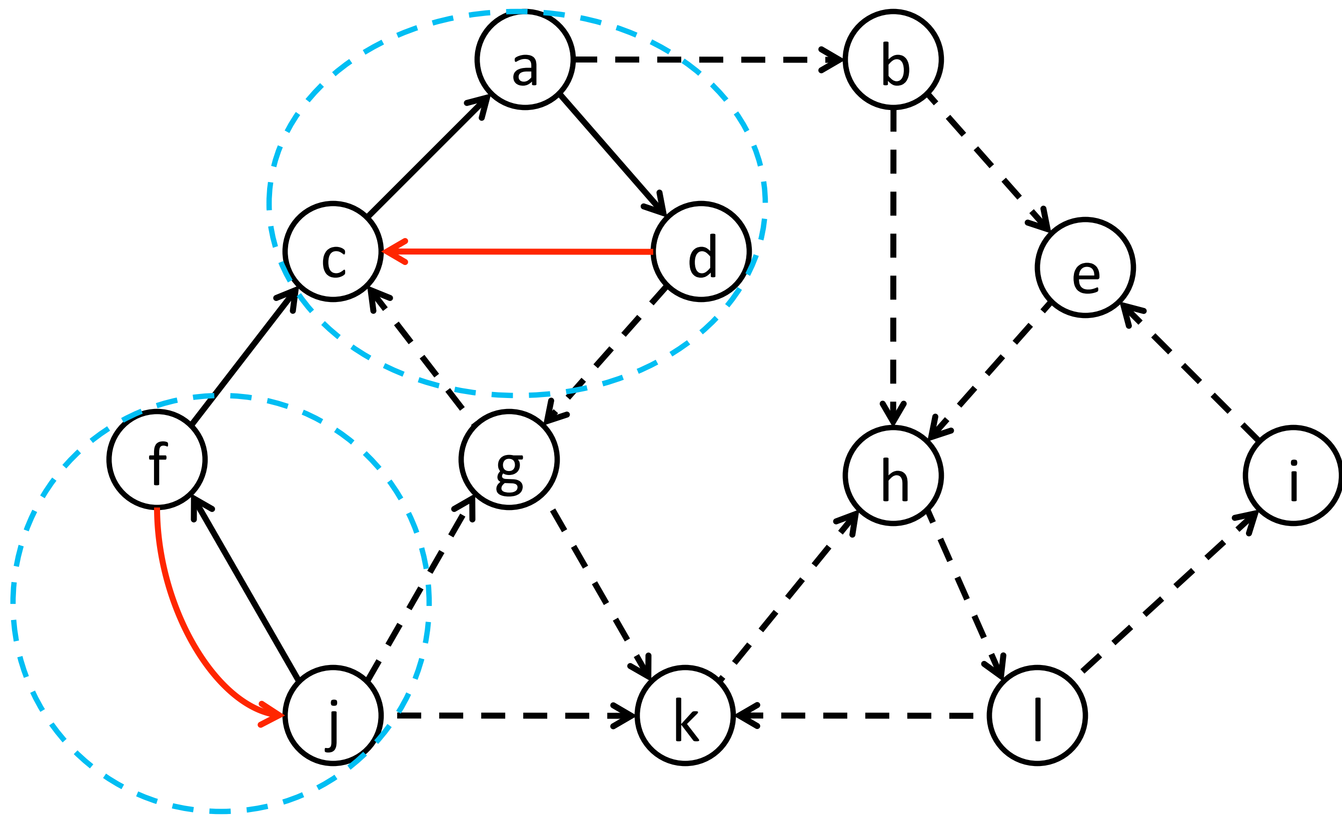
Current path j, f

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



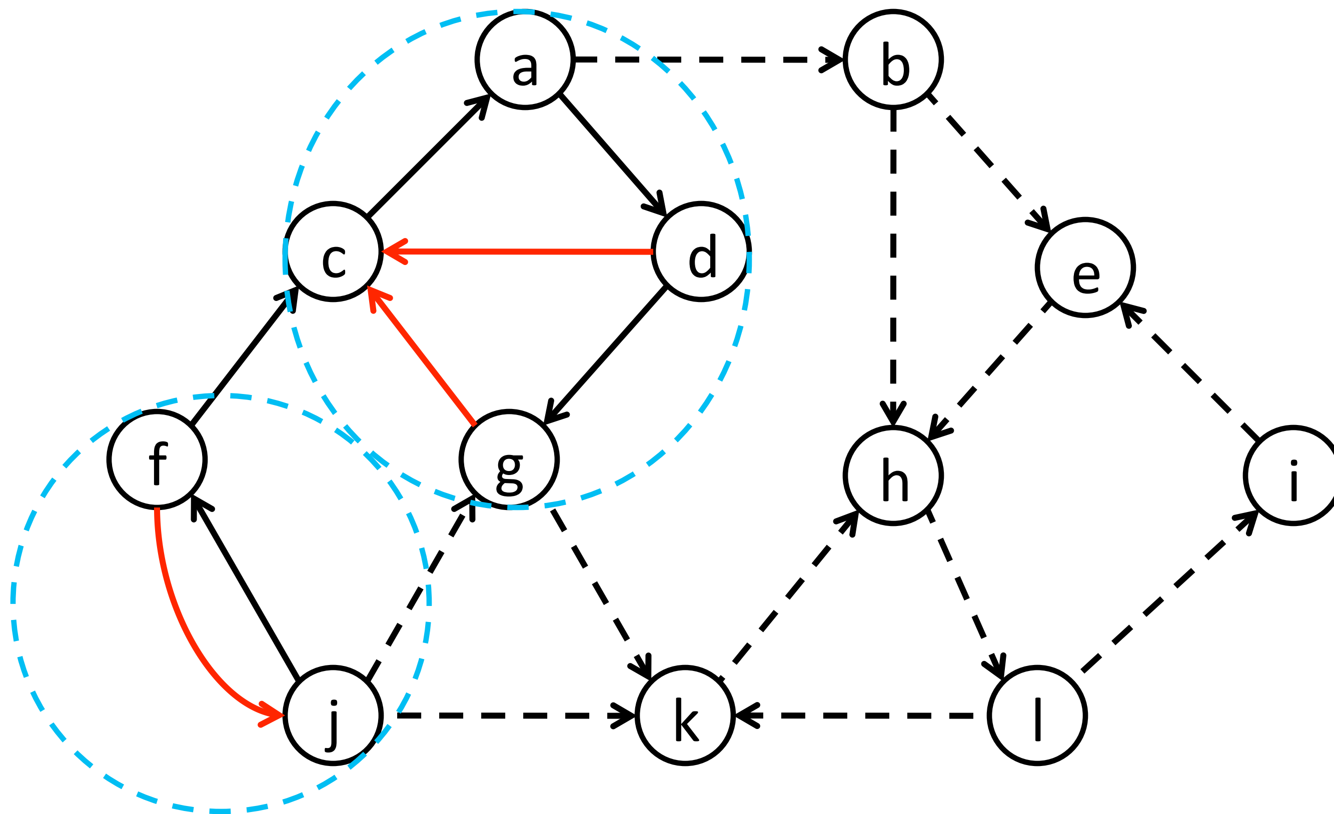
Current path j’, c, a, d

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



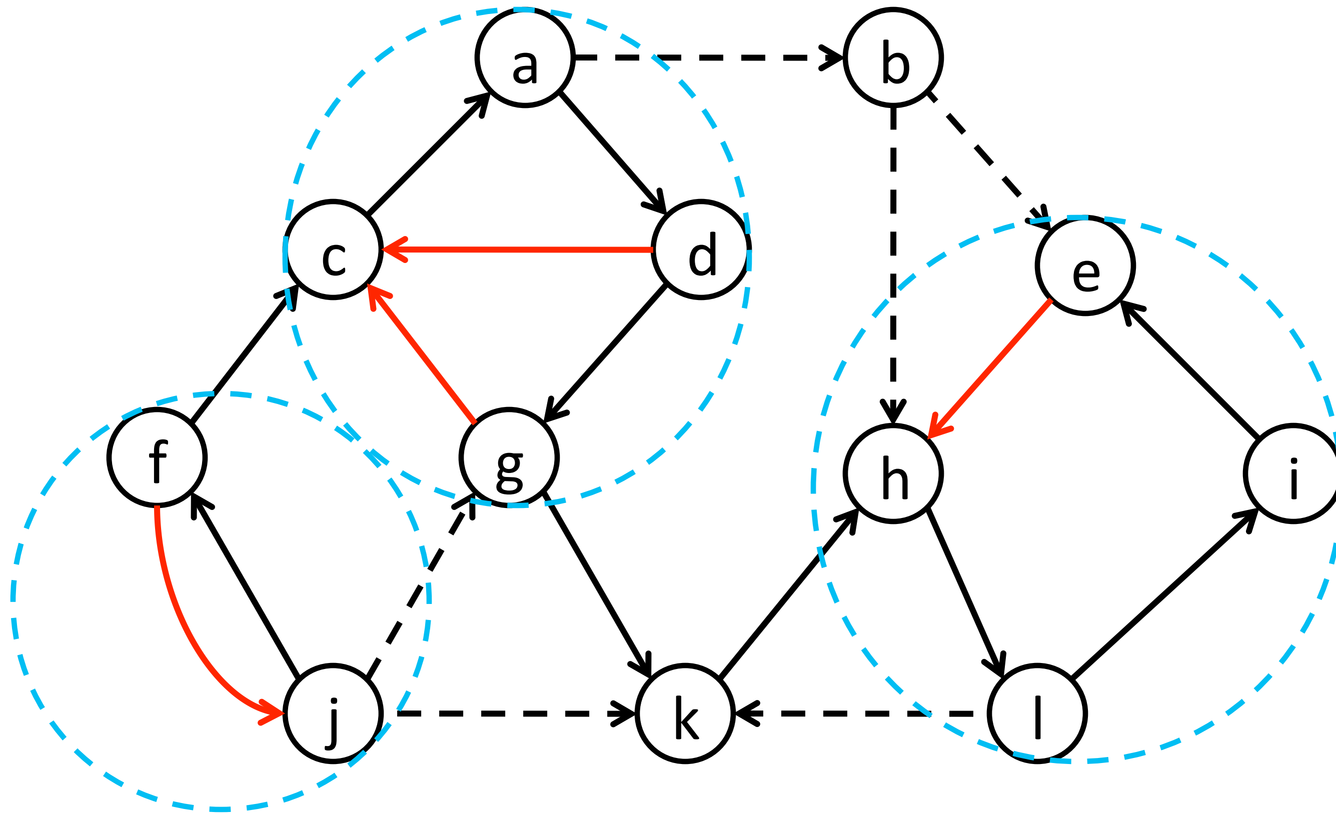
Current path j', c', g

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



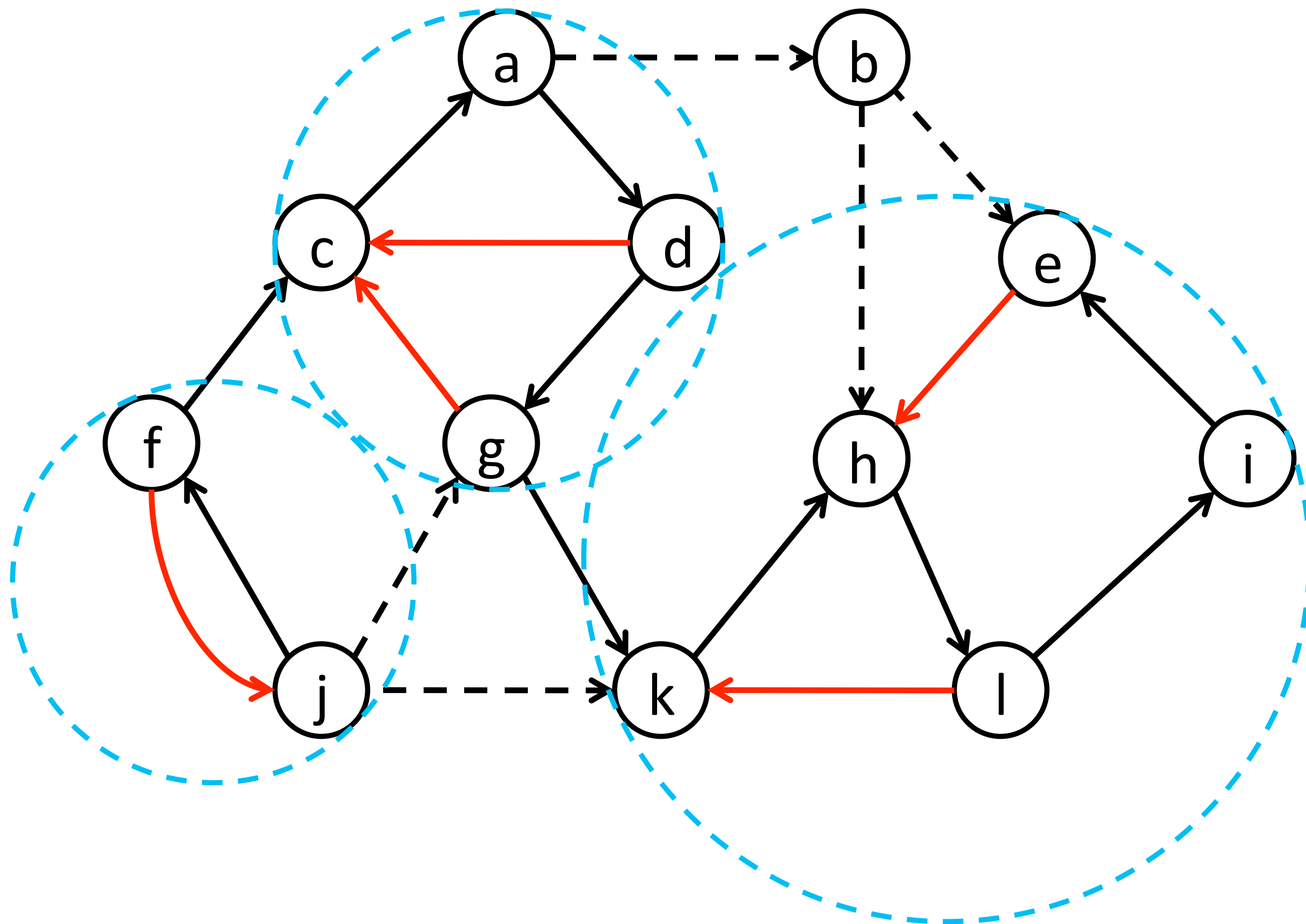
Current path j’, c’’, k, h, l, i, e

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



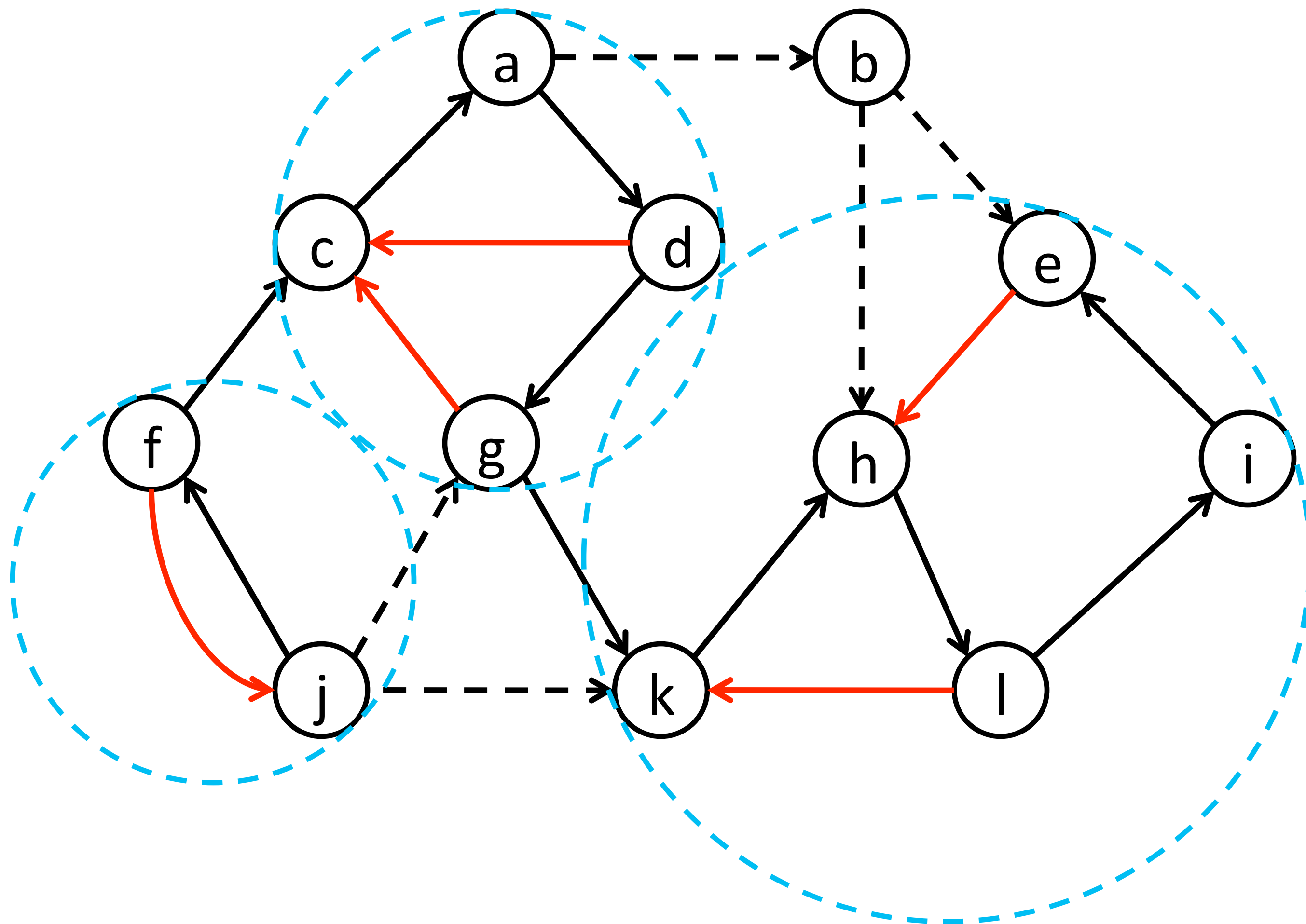
Current path j’, c’’, k, h’

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



Current path j’, c’’, k’

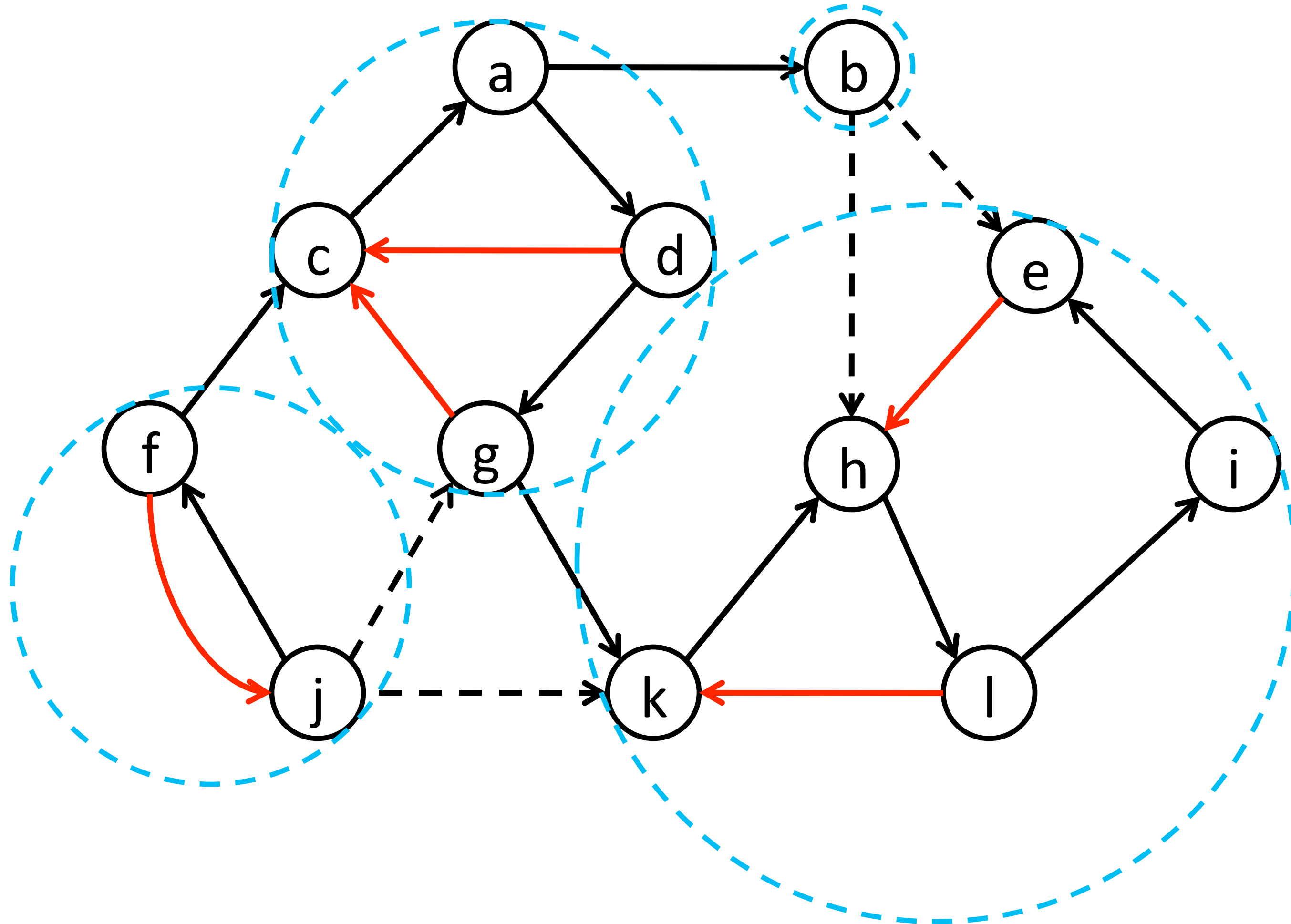
Postvisit of k’ gives component {e, i, l, h, k}



Current path j’, c’’, b

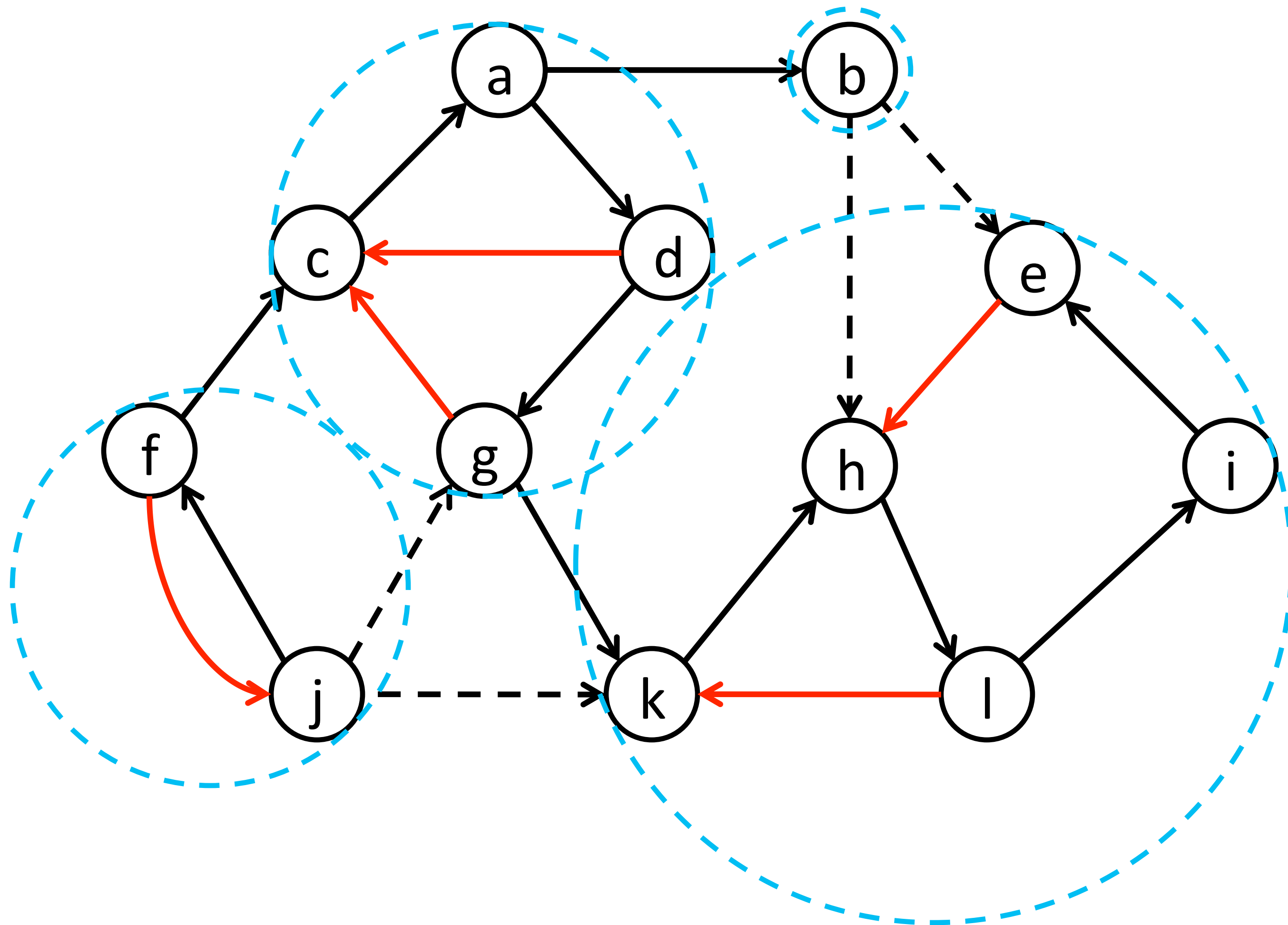
Traversal of (b, e), (b, k) leads to component k’

Postvisit of b gives component {b}



Current path j’, c’’

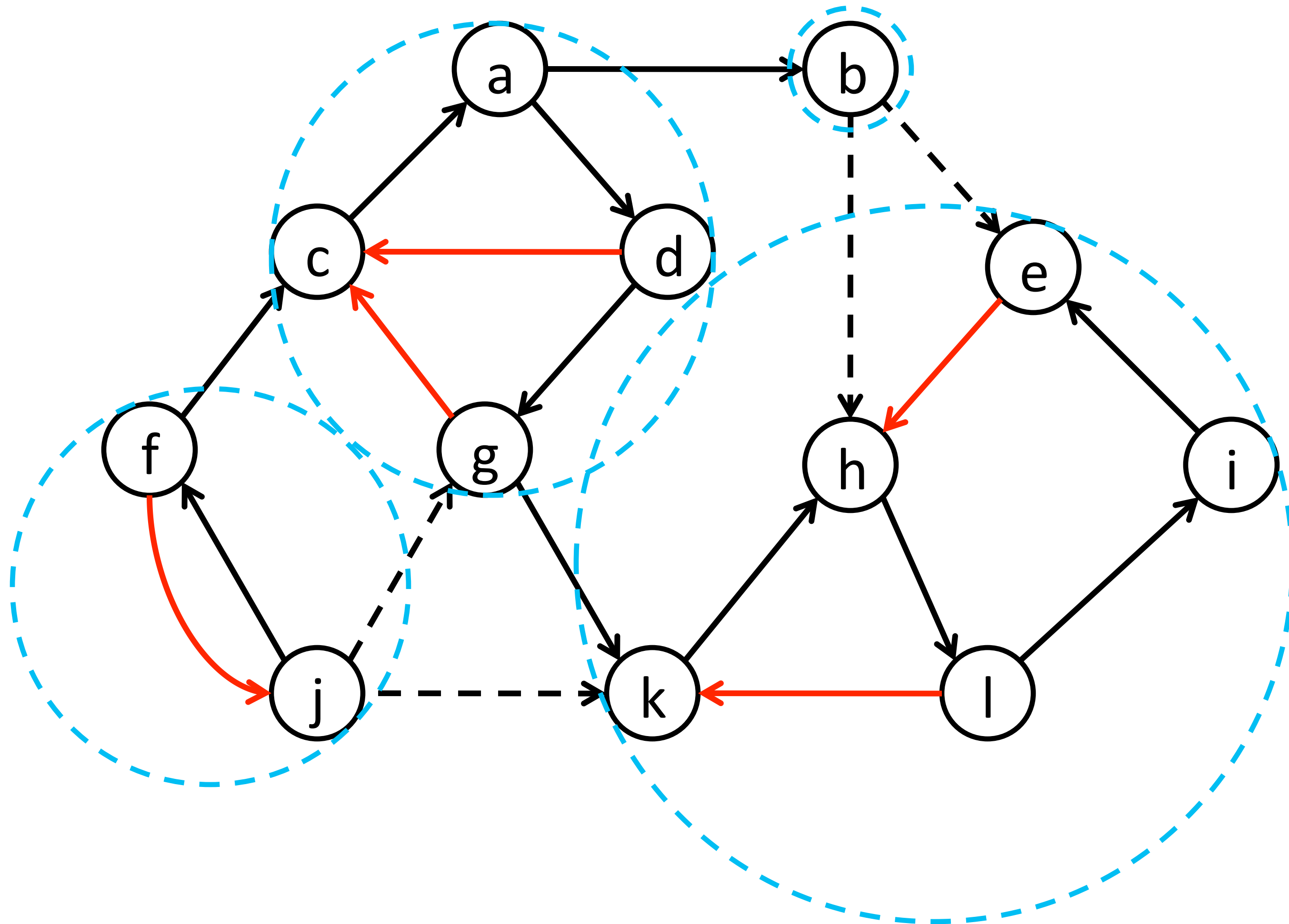
Postvisit of c’’ gives component {g, d, a, c}



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)

traverse(v, w): if w in set S on stack, combine all sets on stack down to and including S

If we maintain a stack of original vertices not yet in components, the elements of each set are contiguous on the stack

Let $s(v)$ = stack position of v (bottom = 1)

To represent sets, keep second stack of positions of bottommost vertices in sets

previsit(v): add v to vertex stack, position to second stack

traverse(v, w): if w on vertex stack, pop all positions bigger than $s(w)$ from second stack

postvisit(v): if $s(v)$ on top of second stack, form component

Implementation

explore(*V*, *E*):

{ $S \leftarrow$ [ ]; $P \leftarrow$ [ ]; $z \leftarrow 0$;

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

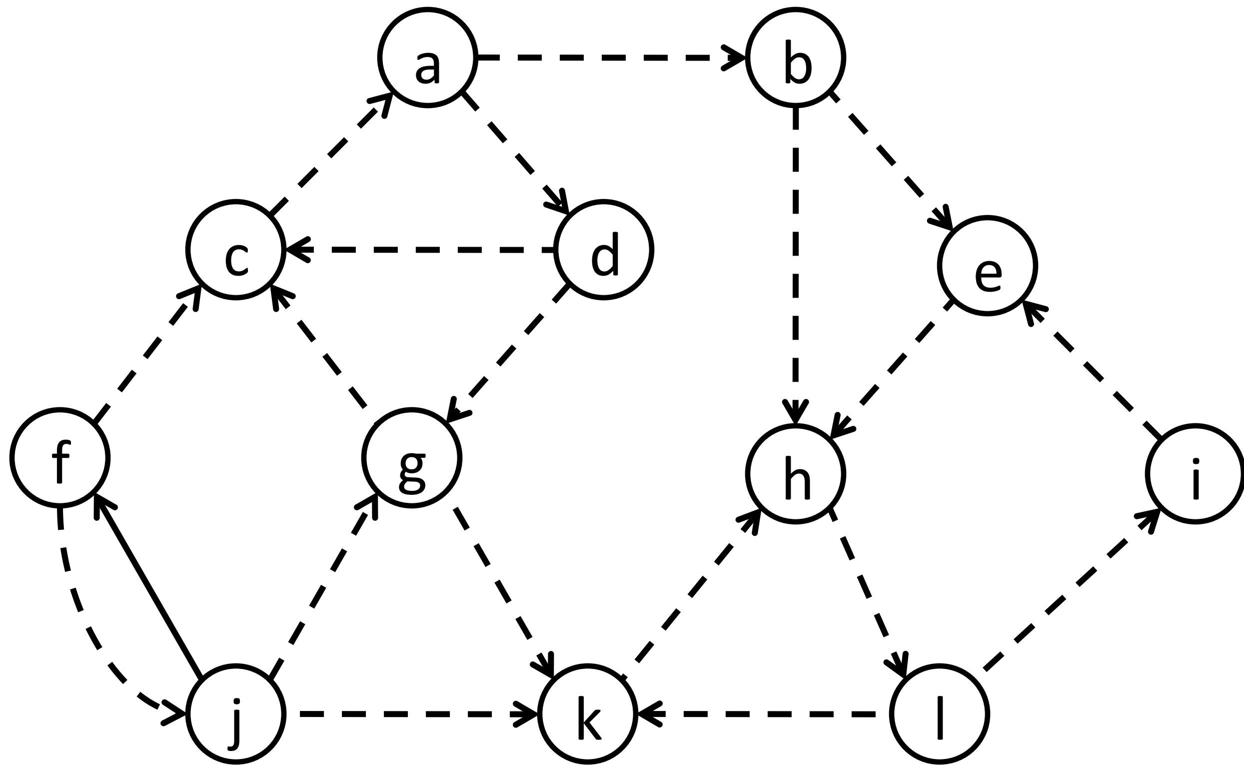
for $v \in V$ **do if** $s(v) = 0$ **then** *search*(v)}

```
search(v):
   {z ← z + 1; s(v) ← z; push(v, S); push(z, P);
    for (v, w) ∈ E do
        if s(w) = 0 then search(w)
        else while top(P) > s(w) do pop(P);
    if top(P) = s(v) then
        {pop(P); x ← null; C = { };
        while x ≠ v do
            {x ← pop(S); [z ← z – 1];
             C ← C ∪ {x}; s(v) ← ∞};
        output C}}
```

Search from j

S: j, f

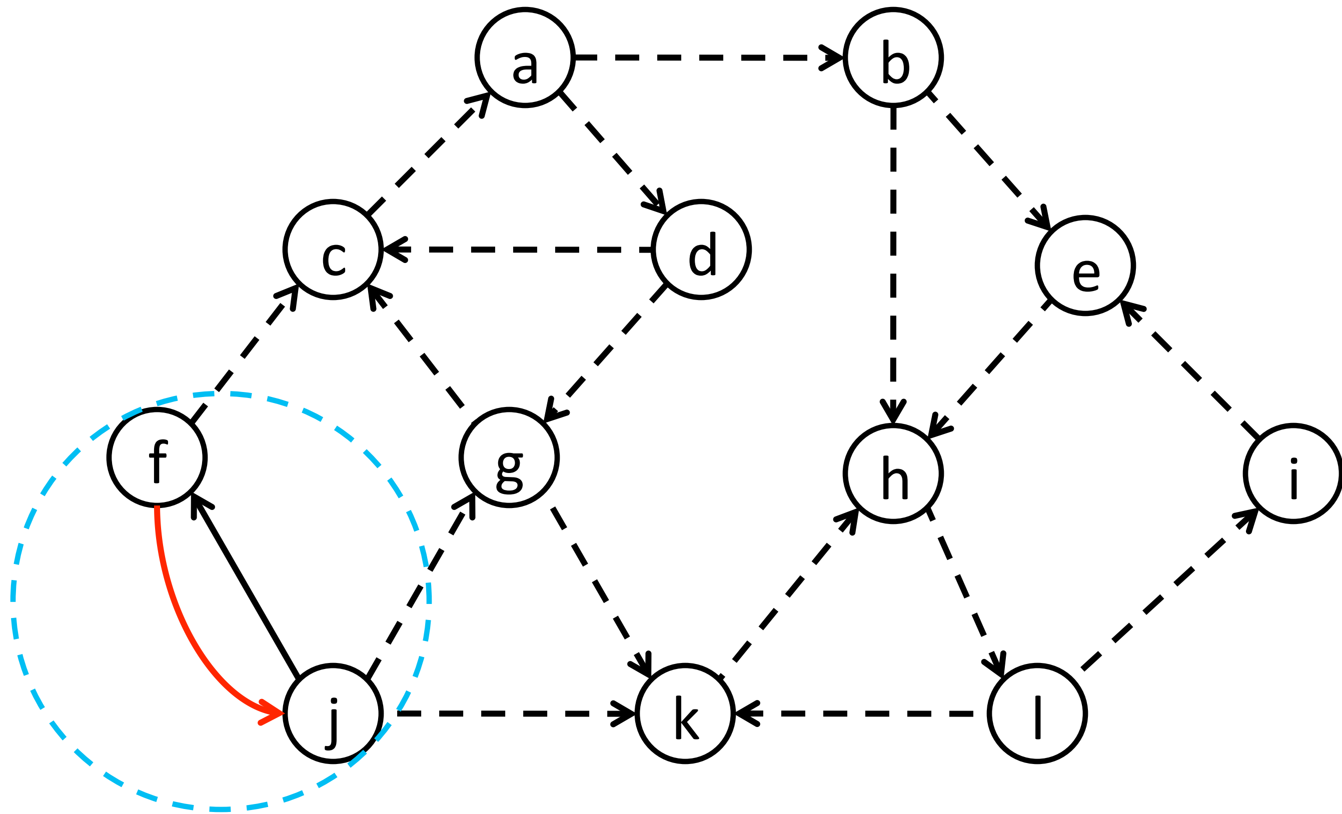
P: 1, 2



S: j, f

P: 1,2

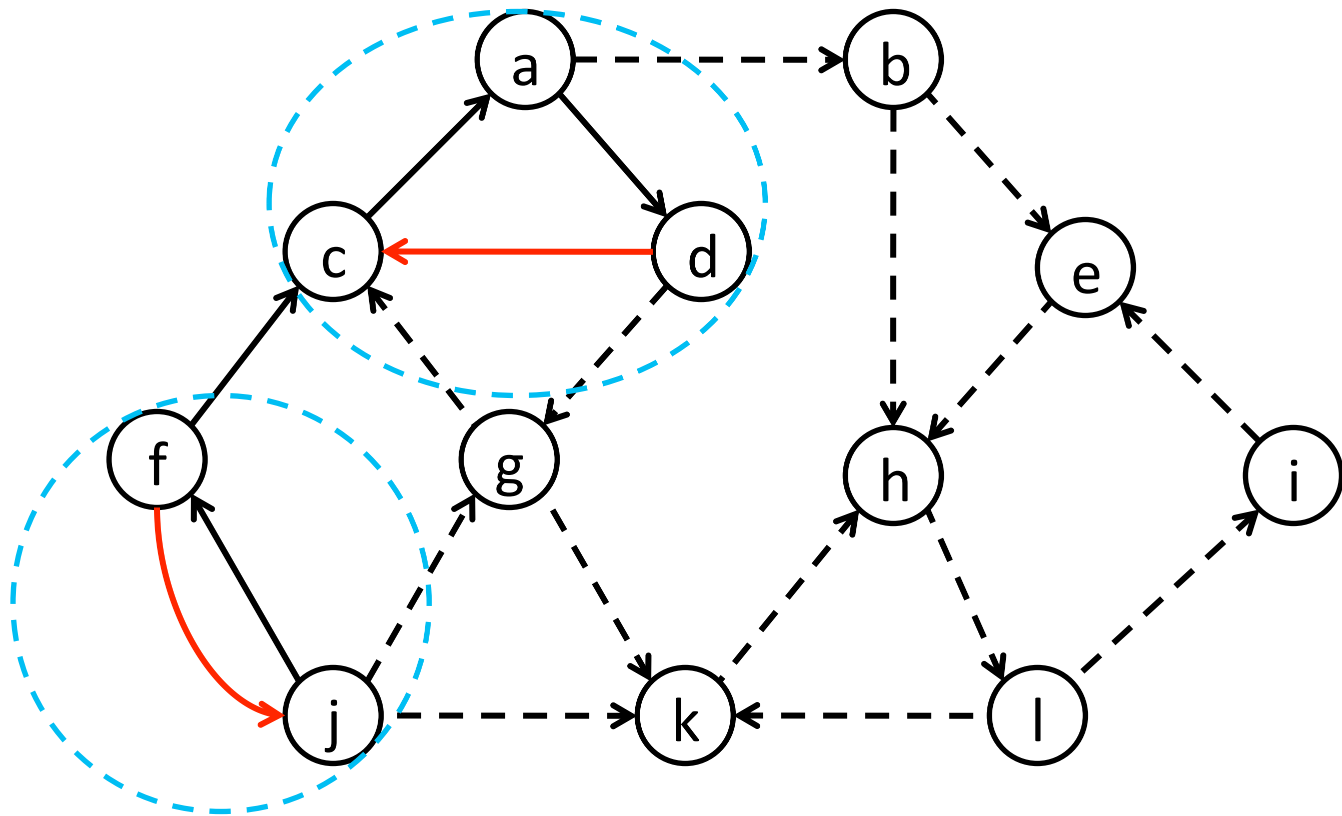
Traversal of (f, j) pops 2



S: j, f, c, a, d

P: 1, 3, 4, 5

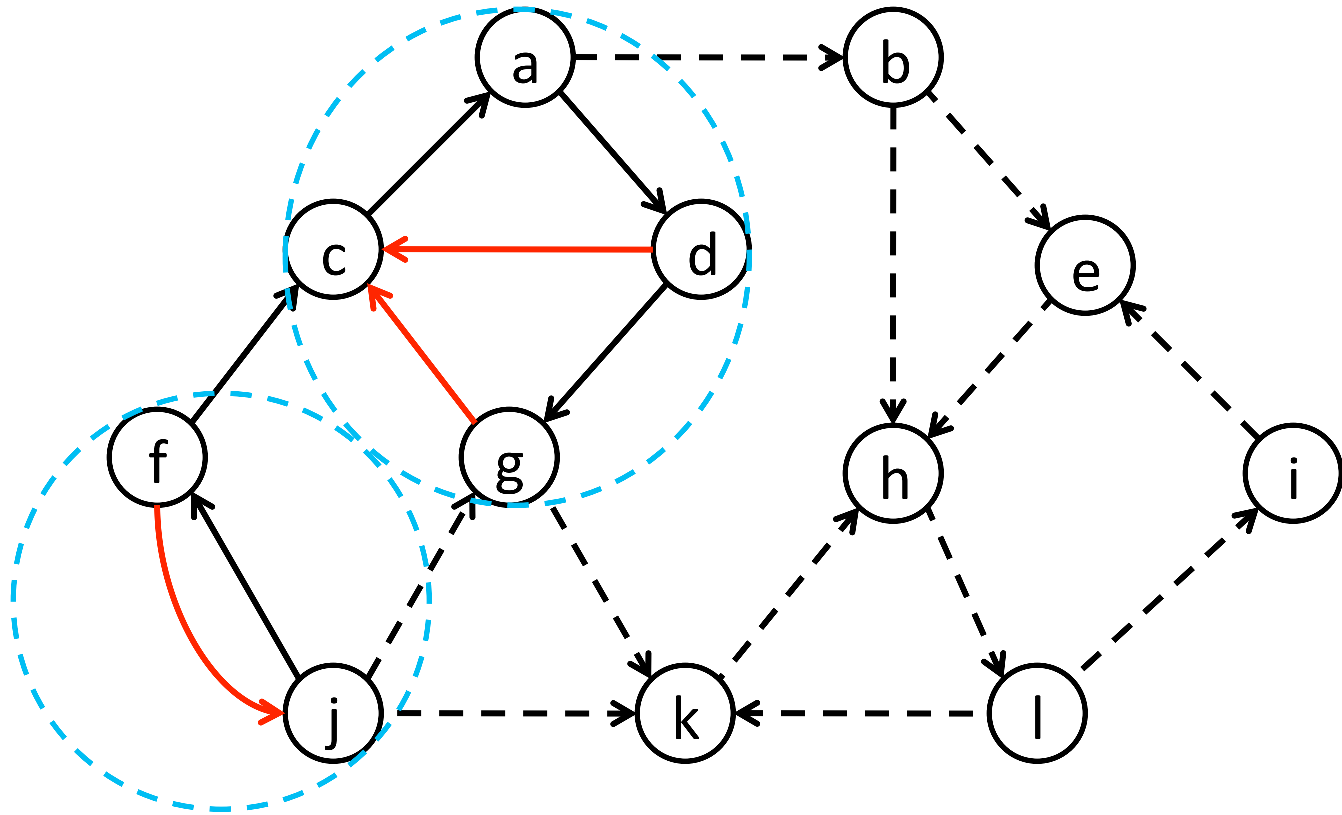
Traversal of (d, c) pops 5, 4



S: j, f, c, a, d, g

P: 1, 3, 6

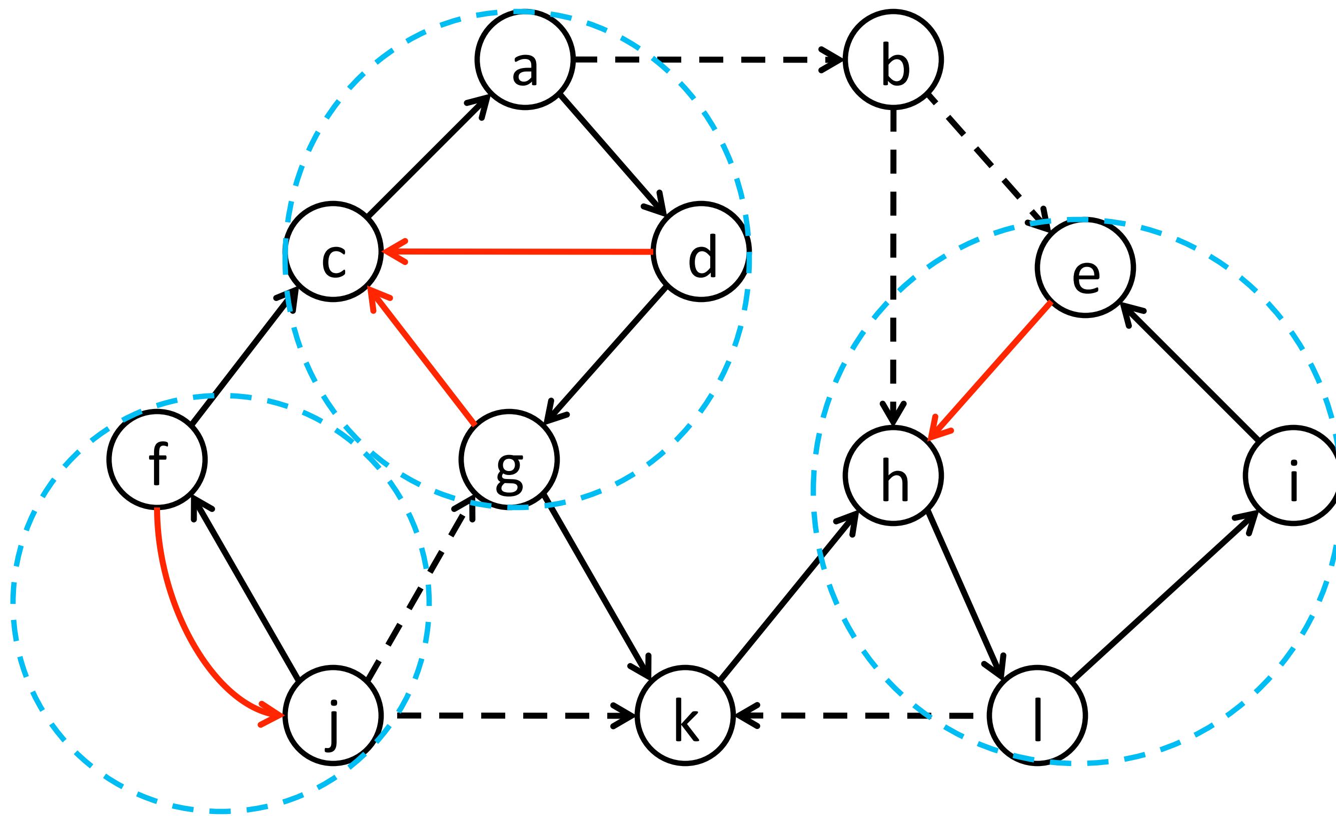
Traversal of (g, c) pops 6



S: j, f, c, a, d, g, k, h, l, i, e

P: 1, 3, 7, 8, 9, 10, 11

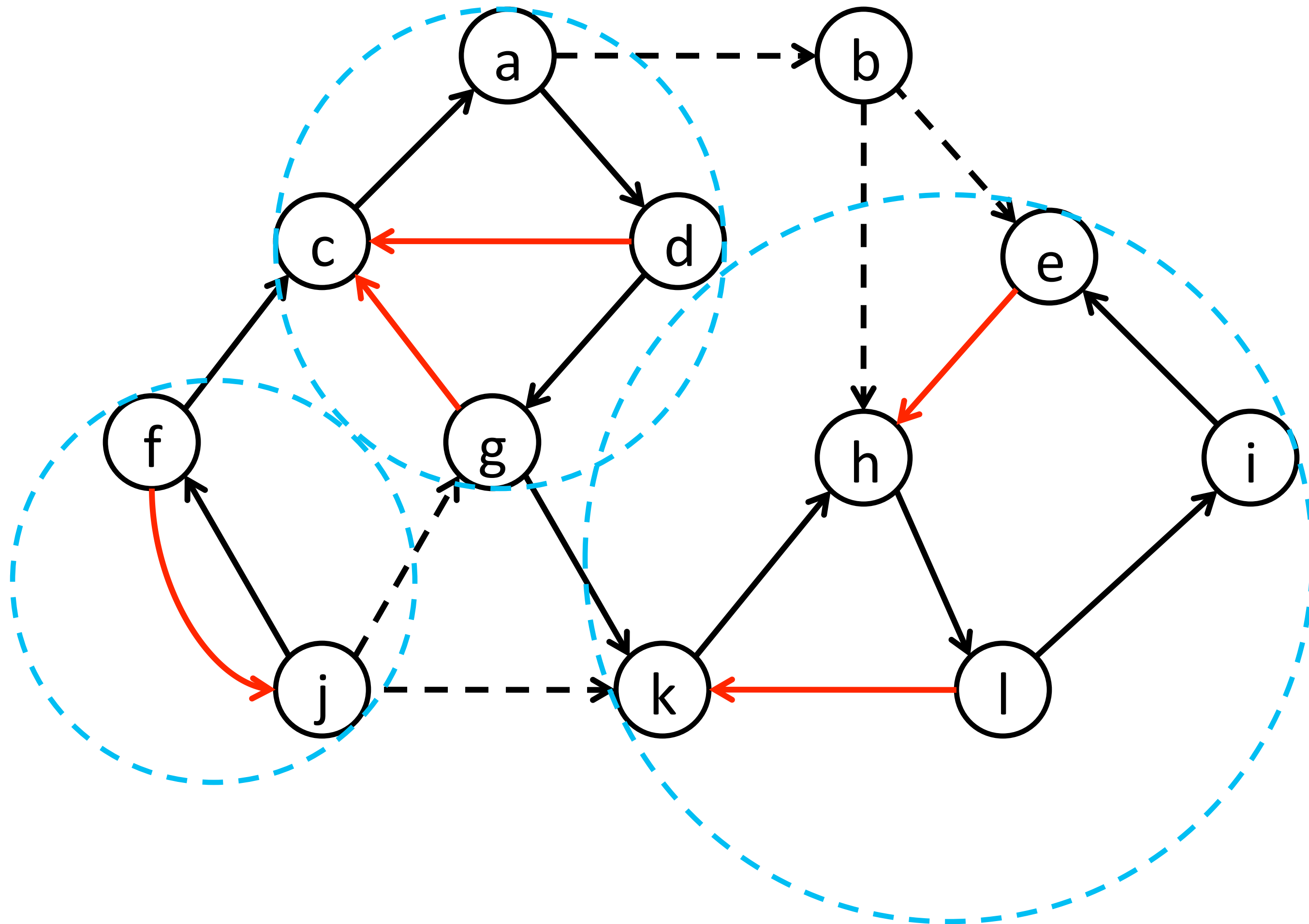
Traversal of (e, h) pops 11, 10, 9



S: j, f, c, a, d, g, k, h, l, i, e

P: 1, 3, 7, 8

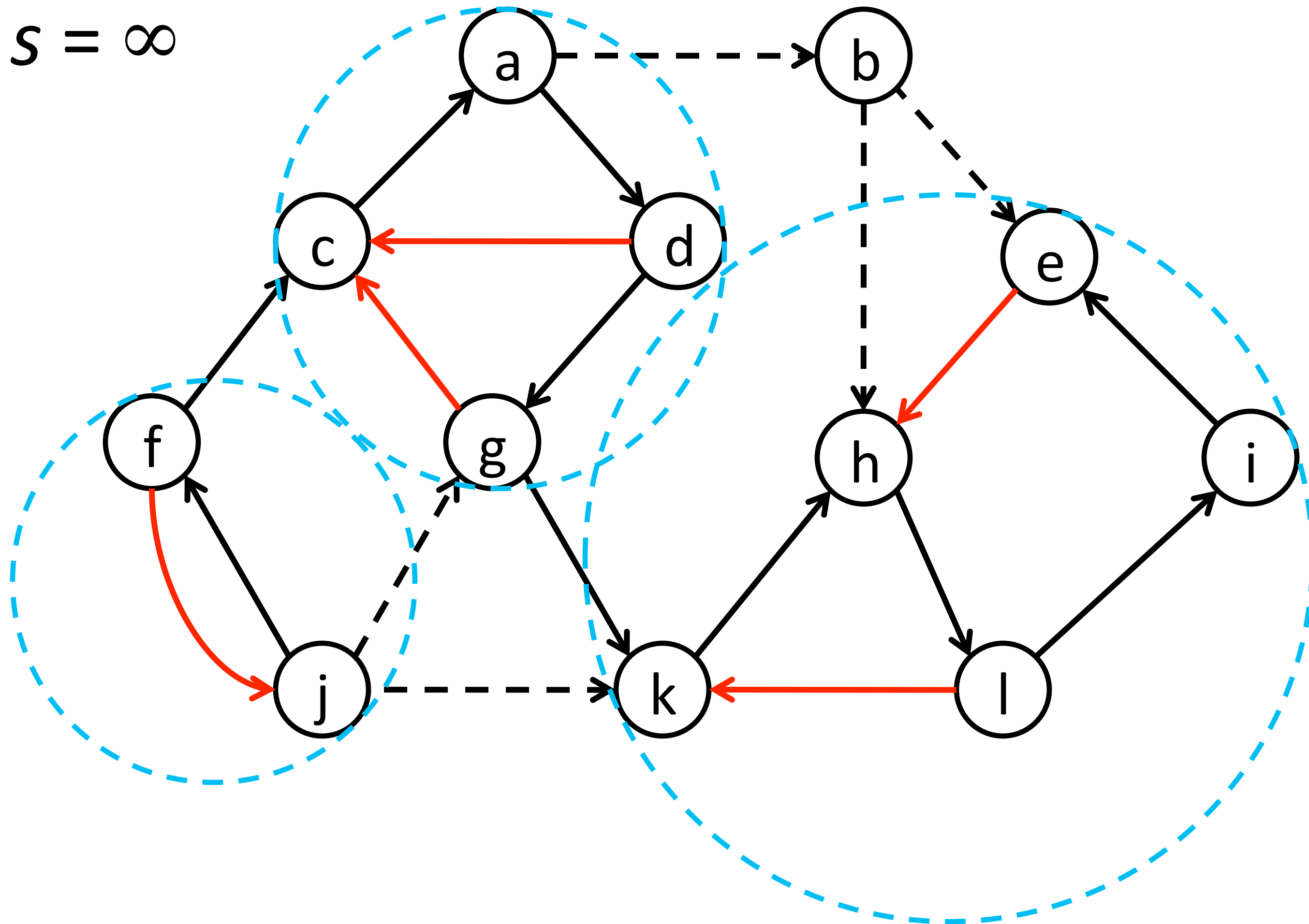
Traversal of (l, k) pops 8



S: j, f, c, a, d, g, k, h, l, i, e

P: 1, 3, 7, 8

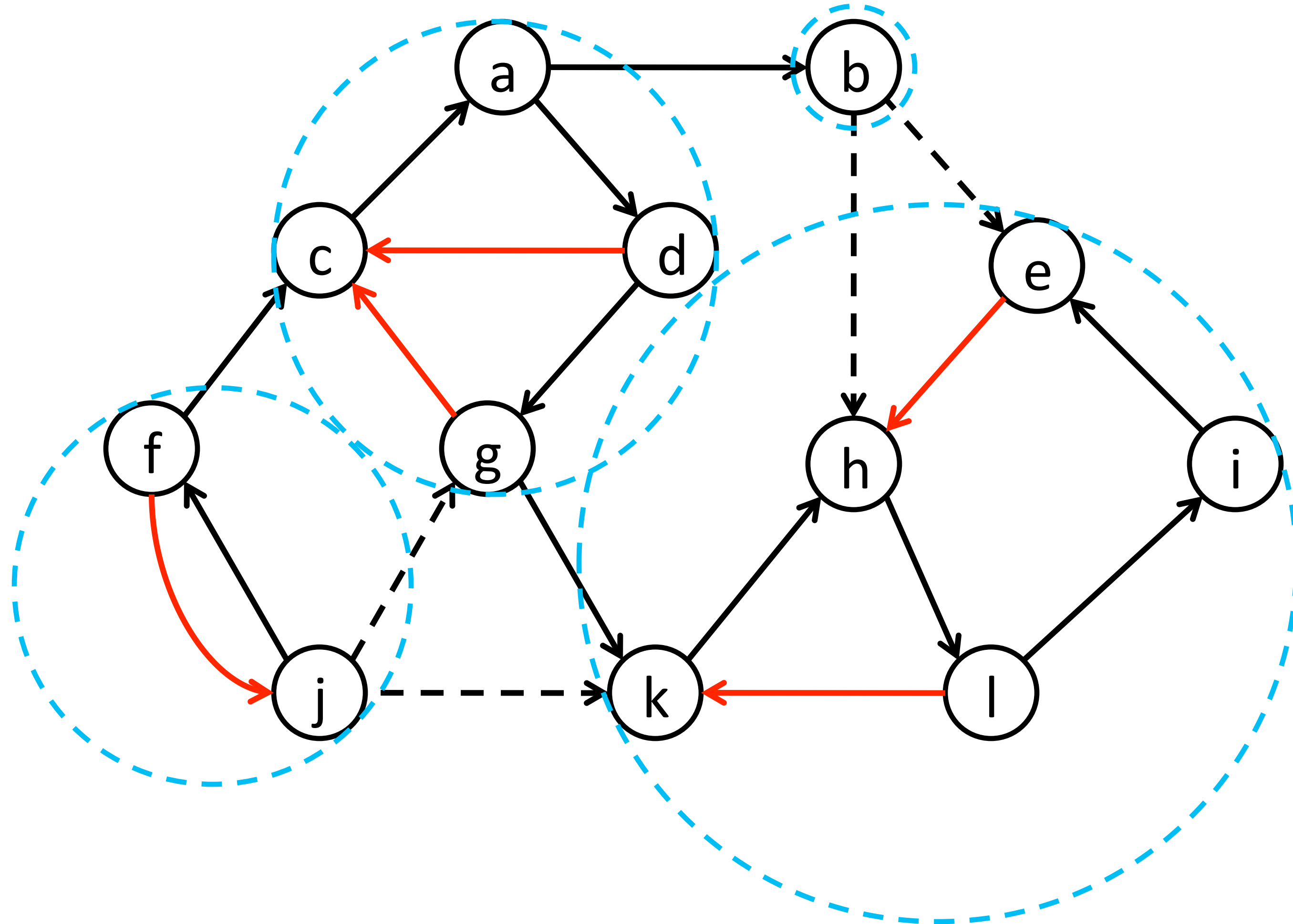
Postvisit of k gives component {e, i, l, h, k}; all get $s = \infty$



S: j, f, c, a, d, g, b

P: 1, 3, 7

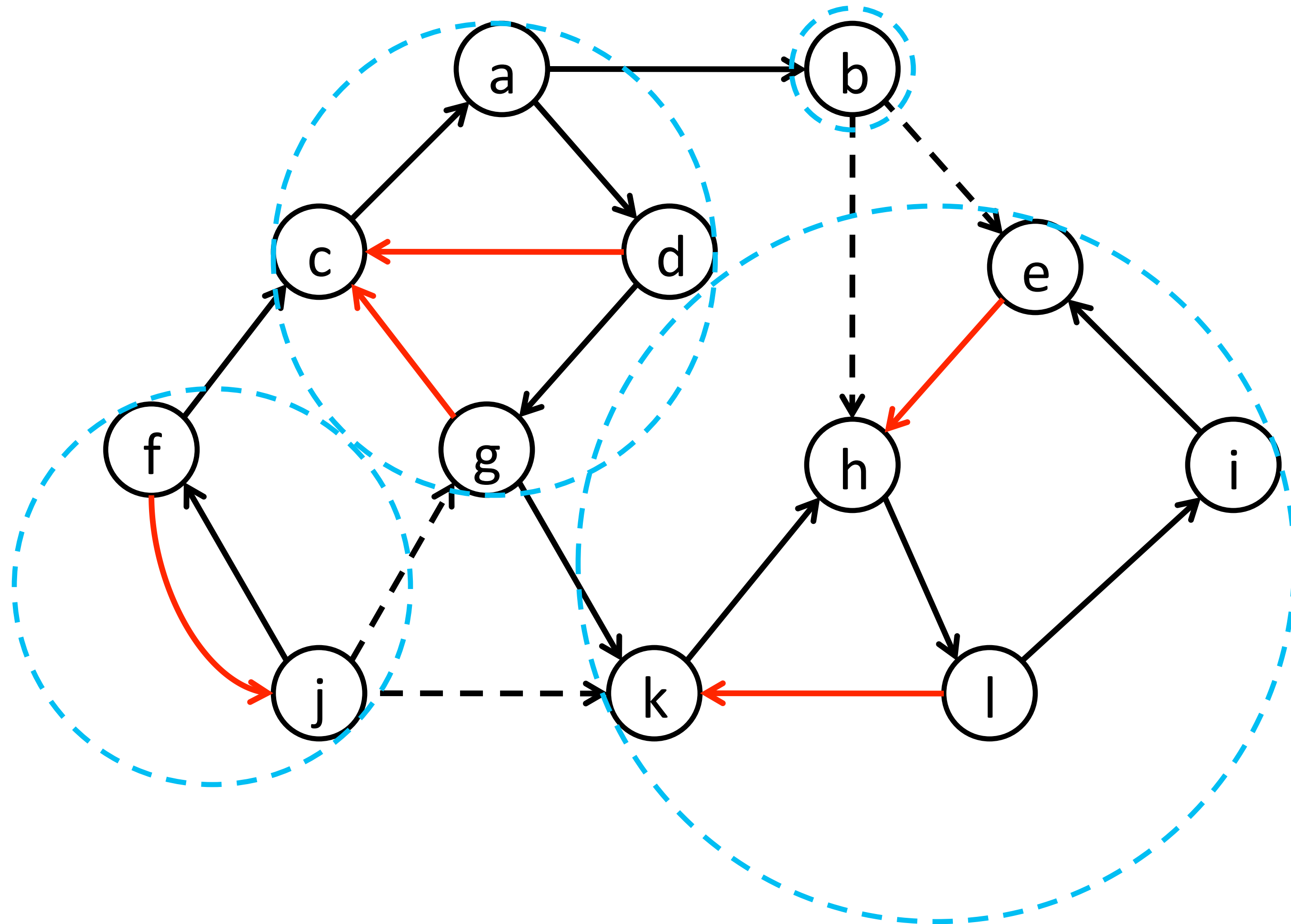
Postvisit of b gives component {b}



S: j, f, c, a, d, g

P: 1, 3

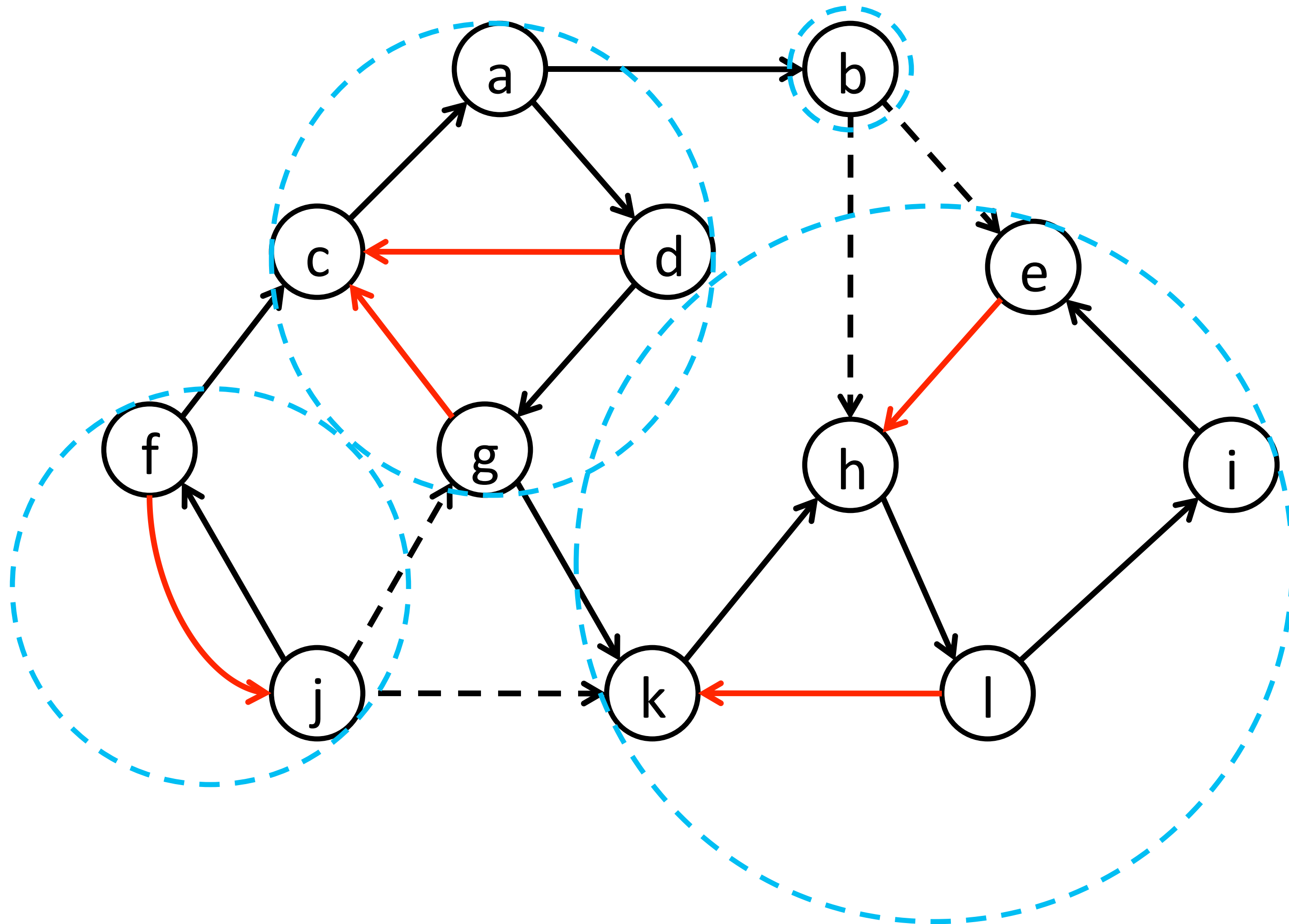
Postvisit of c gives component {g, d, a, c}



S: j, f

P: 1

Postvisit of j’ gives component {f, j}



Correctness proof: The algorithm maintains the following invariants:

Each set of vertices on the vertex stack between adjacent positions on the position stack (bottom inclusive, top exclusive) is strongly connected.

Each traversed arc either has both ends in the same set defined by the positions on the position stack, or leads from one set to the next-higher one, or leads to a vertex not on the vertex stack.

Each set of popped vertices forms a component.

Variants

Give vertices not in a component any numbering consistent with order on vertex stack, e.g. preorder number

Put vertices, instead of their numbers, on second stack.

Add each newly visited vertex only to second stack, push onto vertex stack when popped from second stack: one number per vertex and one array or set of pointers for both stacks, saves space

Number components consecutively from $n + 1$; when forming a component, give all its vertices the component number

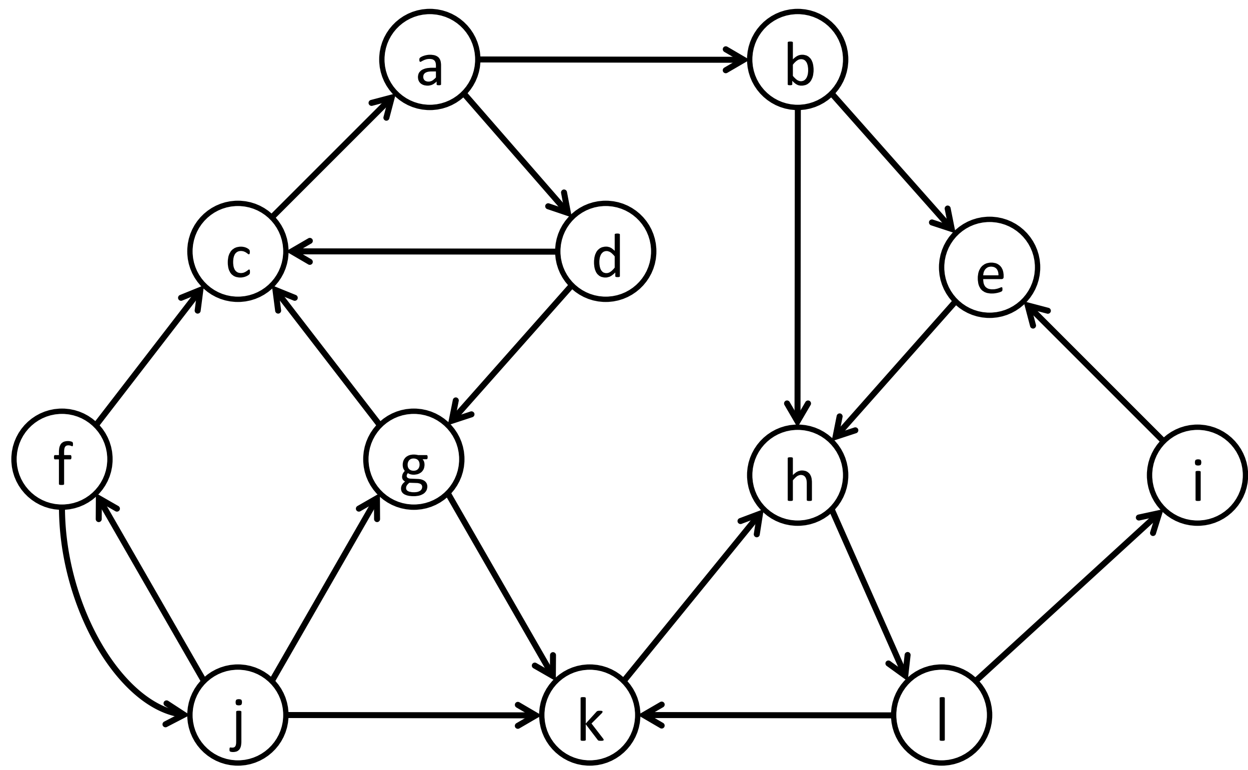
Two-way algorithm
Kosaraju 1978

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.)

The DFS trees generated by the second exploration span the strong components.


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

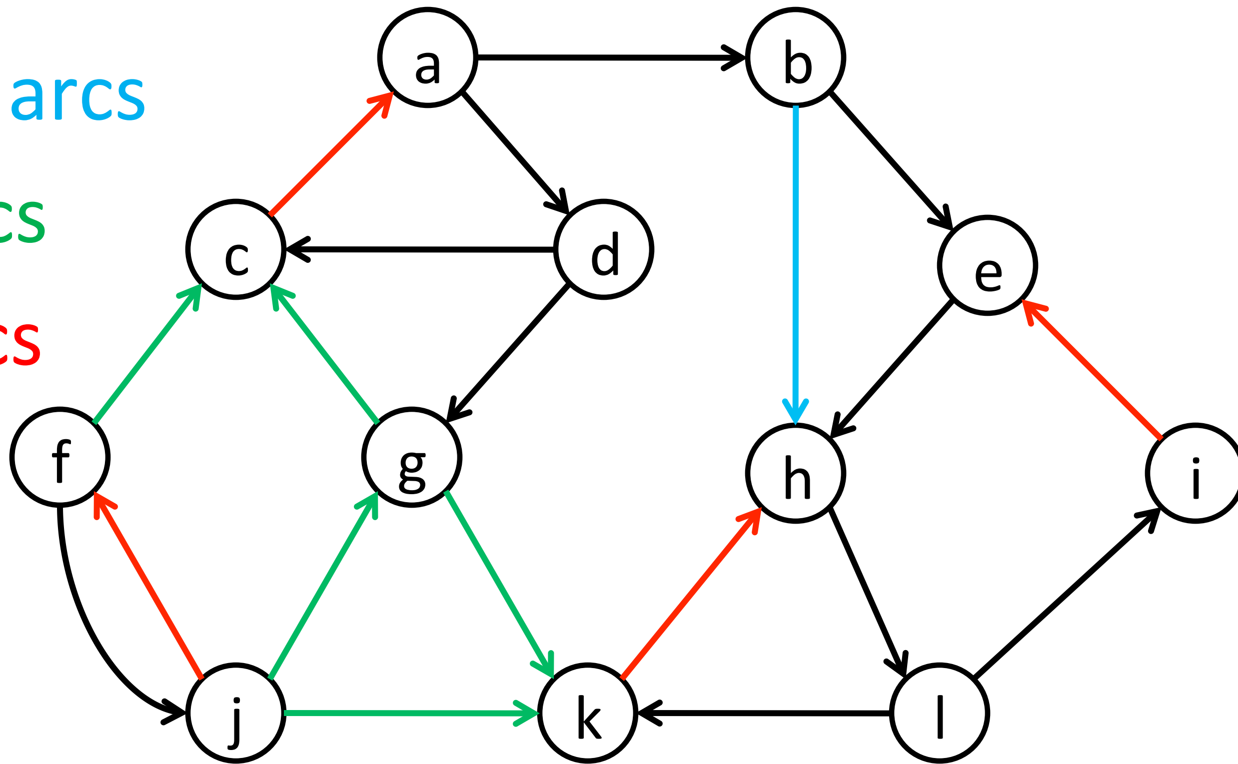
postorder: i, k, l, h, e, b, c, g, d, a; j, f

tree arcs

forward arcs

cross arcs

cycle arcs



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

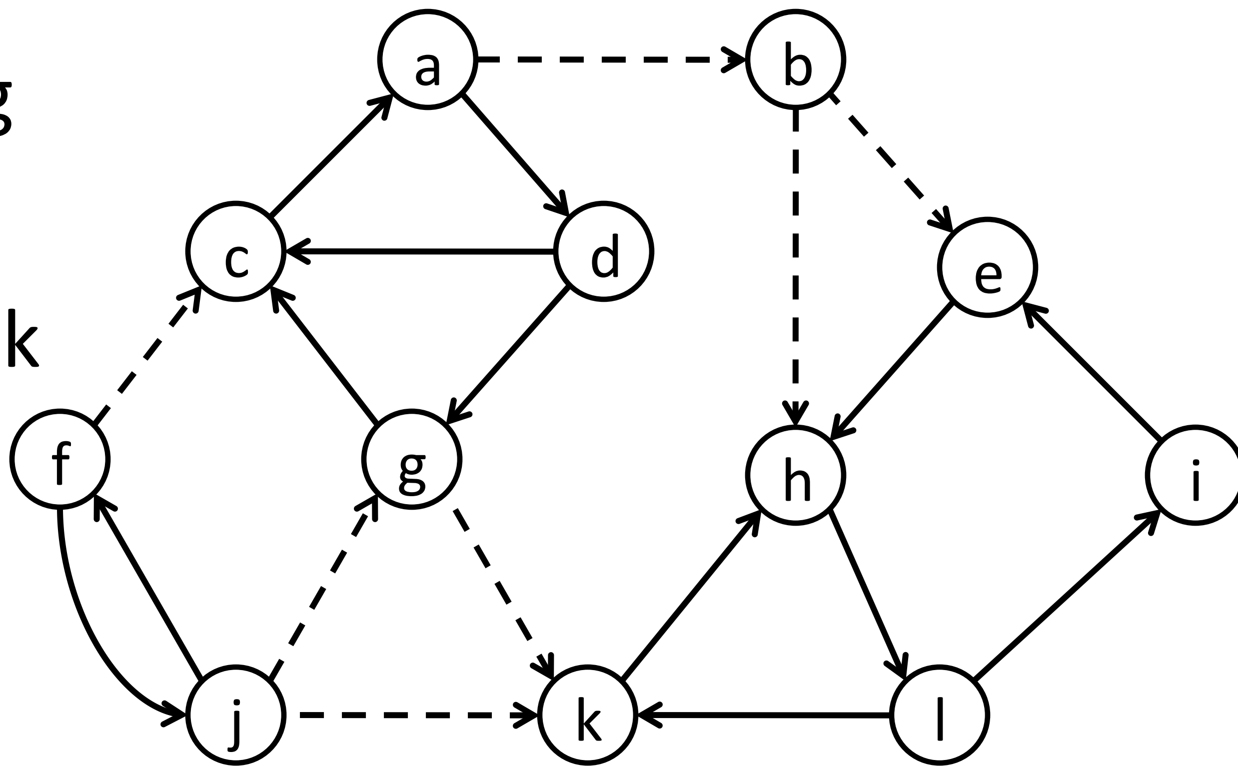
backward searches:

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.