Applications of Depth First Search

Thomas Schwarz, SJ

Topological Sort

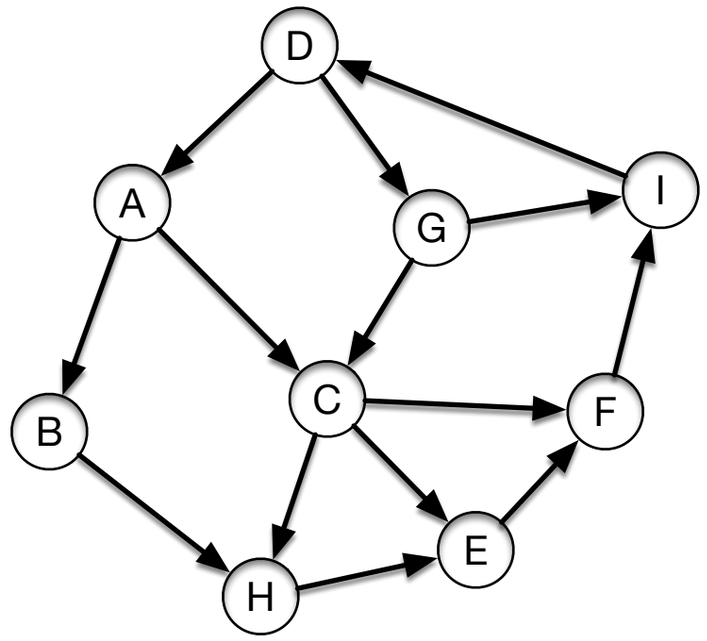
- Recall topological sort
  - We are given a directed graph
  - Want to order all vertices such that no edge goes from a higher-numbered vertex to a lower-numbered vertex
  - If this is impossible, then we have a cycle
  - So, our algorithm also detects whether there is a cycle in a directed graph
  - We use DFS for an even better algorithm

Topological Sort

- Run DFS on all nodes
  - Order nodes according to finish time in descending order

Topological Sort

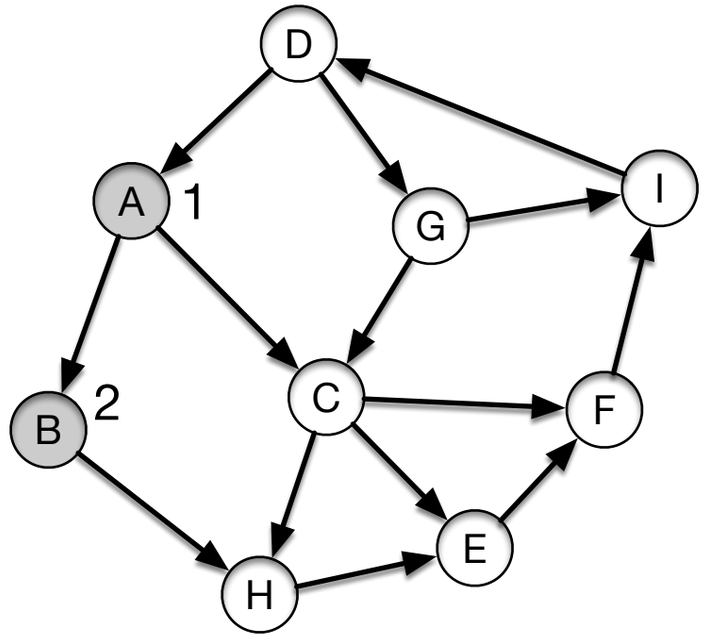
- Start in A
  - Order adjacency lists alphabetically




```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

Topological Sort

- Visit B

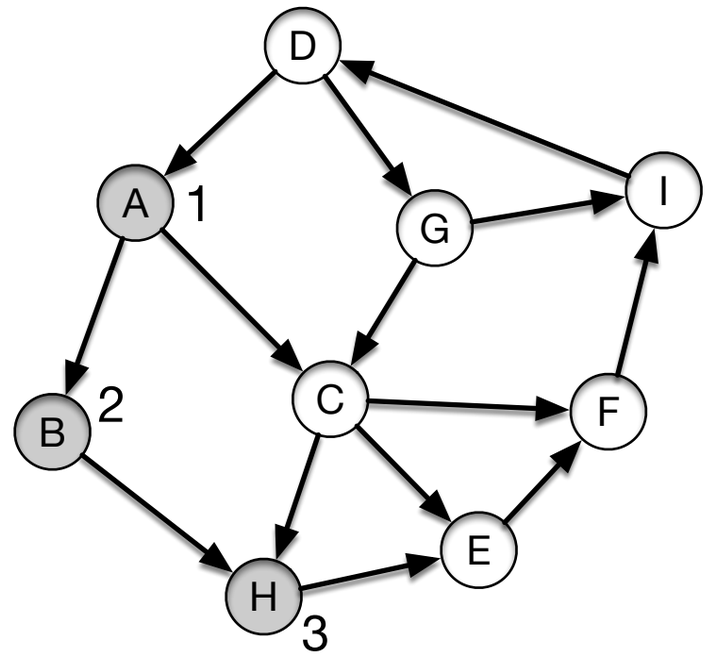



```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

```
visit(B)
visit(A)
```

Topological Sort

- Visit H

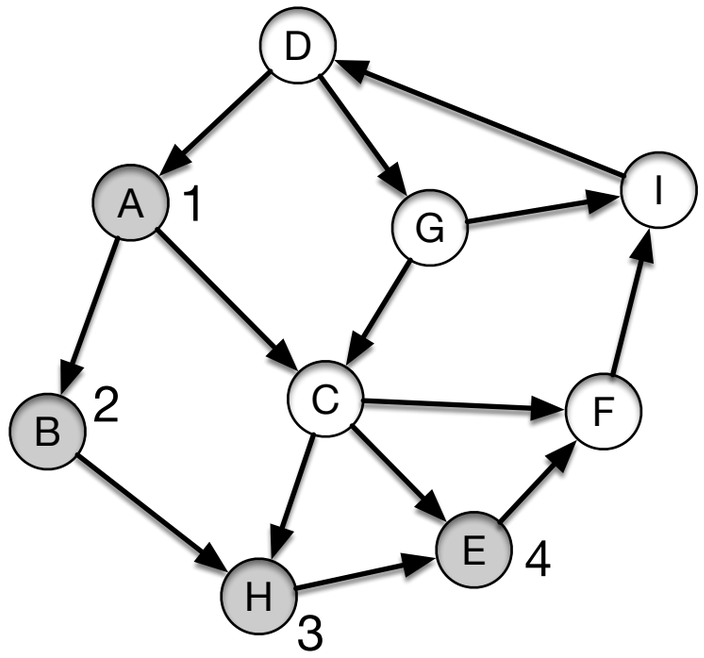



```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

```
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Visit E

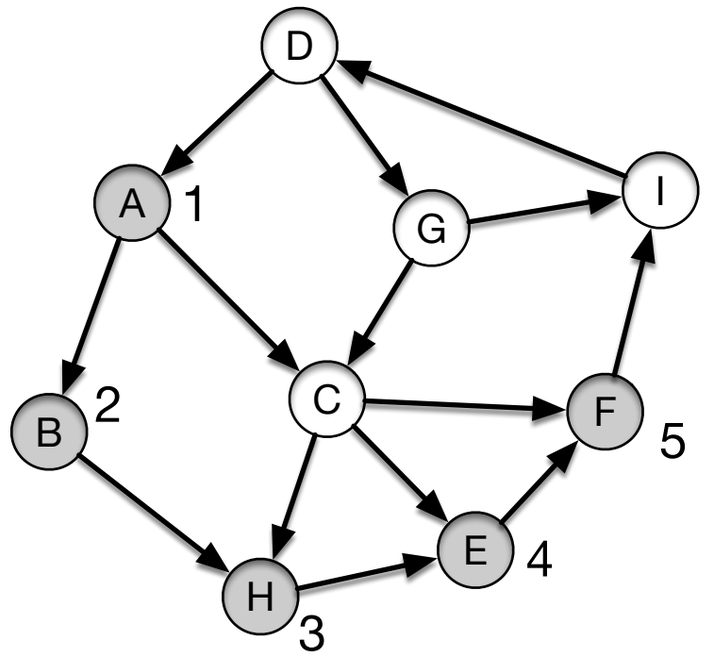



```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

```
visit(E)
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Visit F

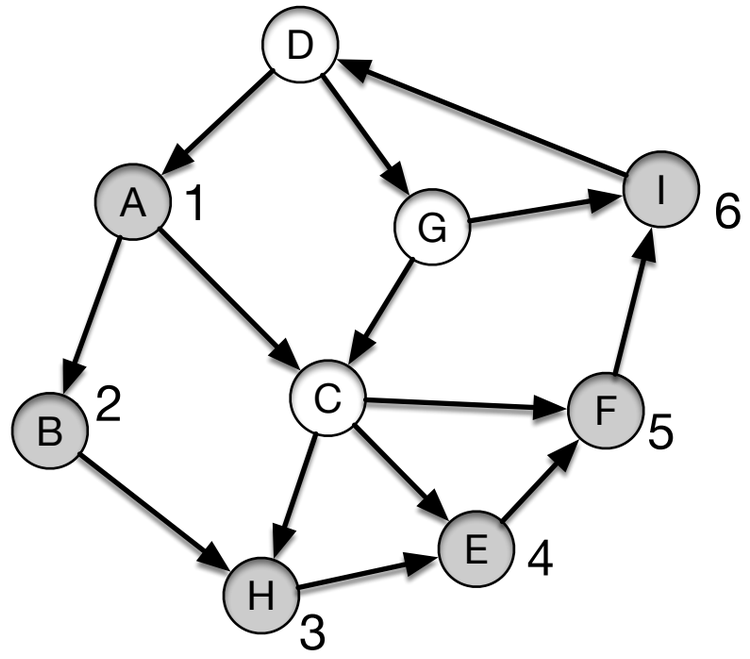



```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

```
visit(F)
visit(E)
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Visit I

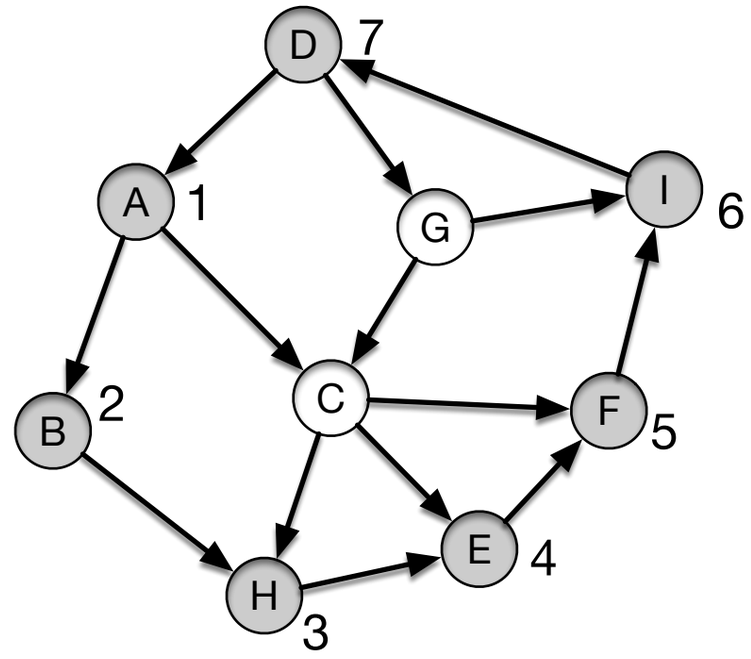



```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

```
visit(I)
visit(F)
visit(E)
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Visit D

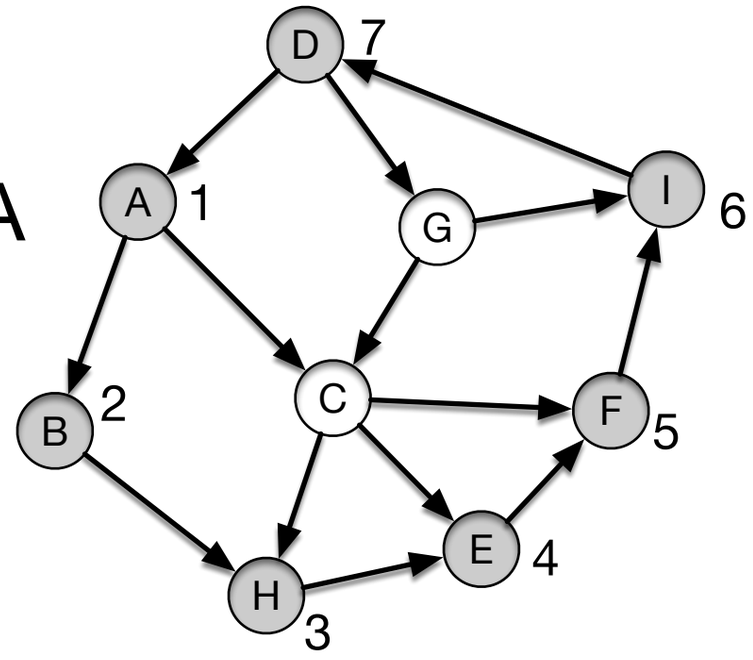



```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

```
visit(D)
visit(I)
visit(F)
visit(E)
visit(H)
visit(B)
visit(A)
```

Topological Sort

- At this point:
  - The adjacency list of D starts with A
  - A is gray
  - This edge becomes a back edge!
  - And shows that there is a cycle

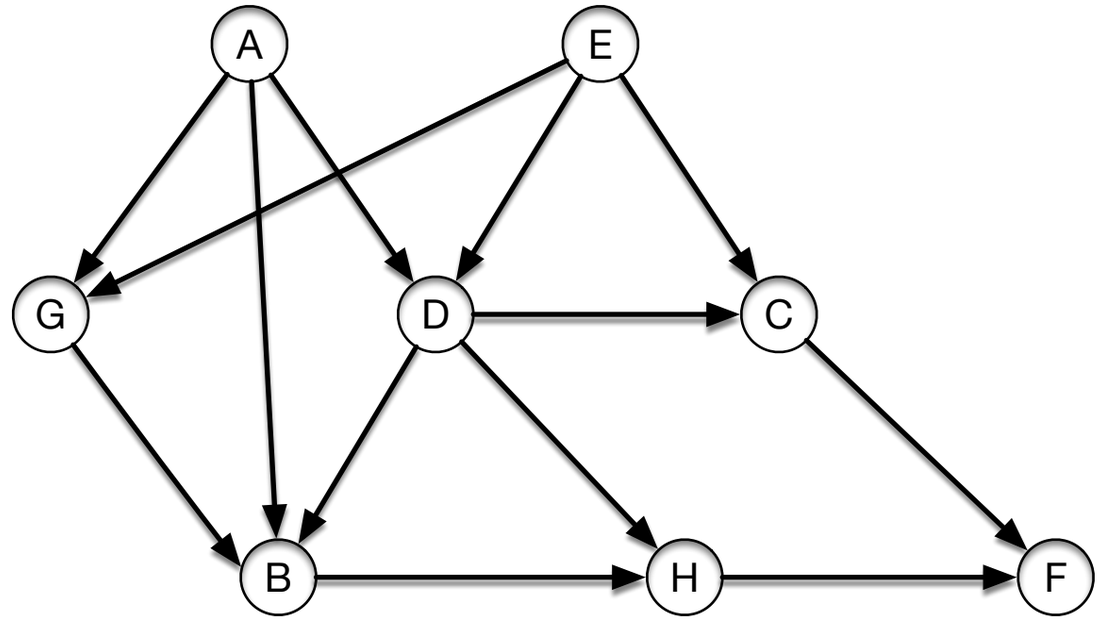



```
A: B,C
B: H
C: E,F,H
D: A,G
E: F
F: I
G: C
H:
I: D
```

```
visit(D)
visit(I)
visit(F)
visit(E)
visit(H)
visit(B)
visit(A)
```

Topological Sort

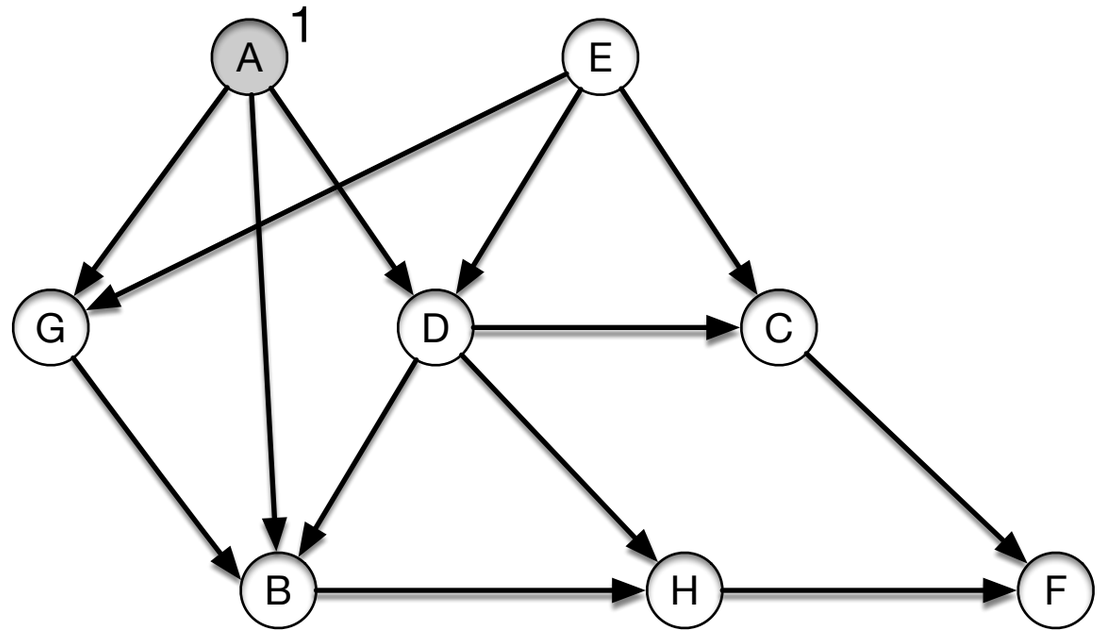
- A different example
  - Start in A




```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

Topological Sort

- A different example
  - Start in A

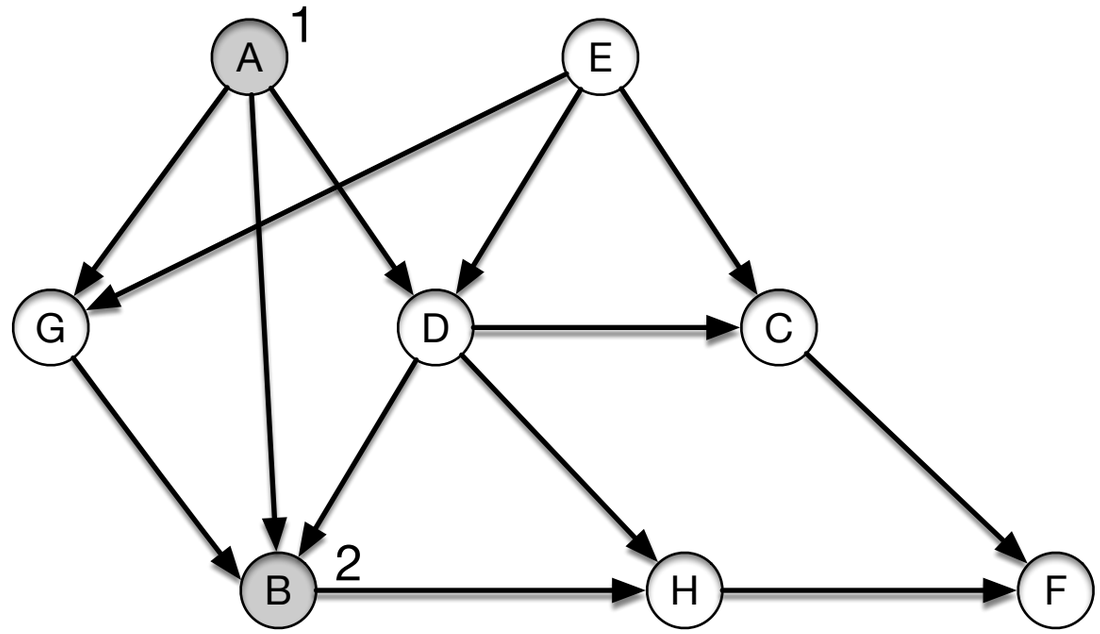



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(A)
```

Topological Sort

- Visit B

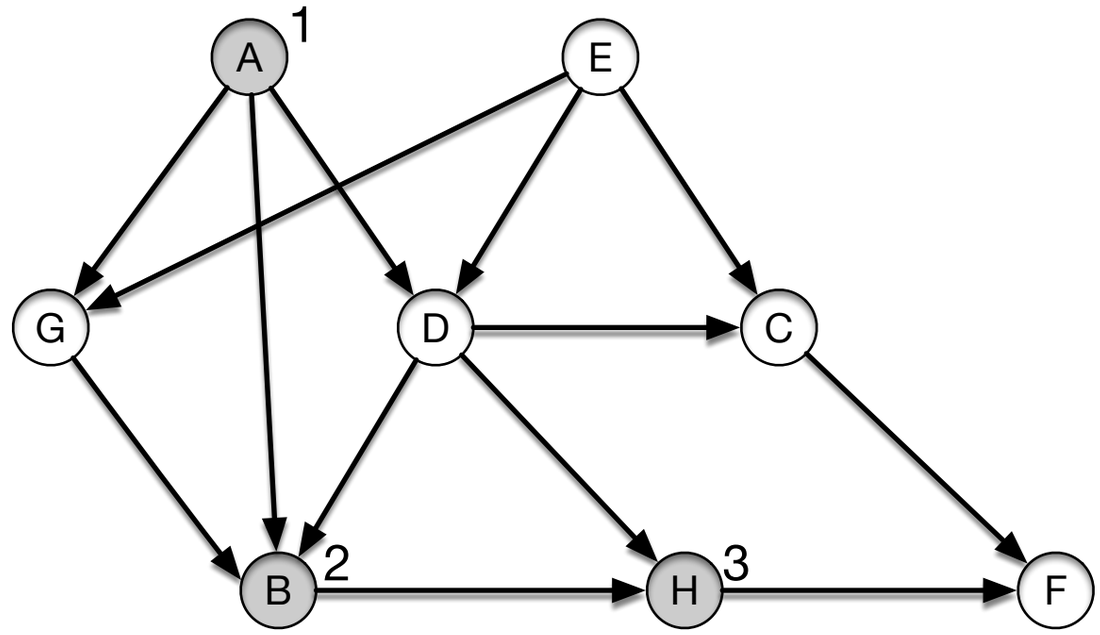



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(B)
visit(A)
```

Topological Sort

- Visit H

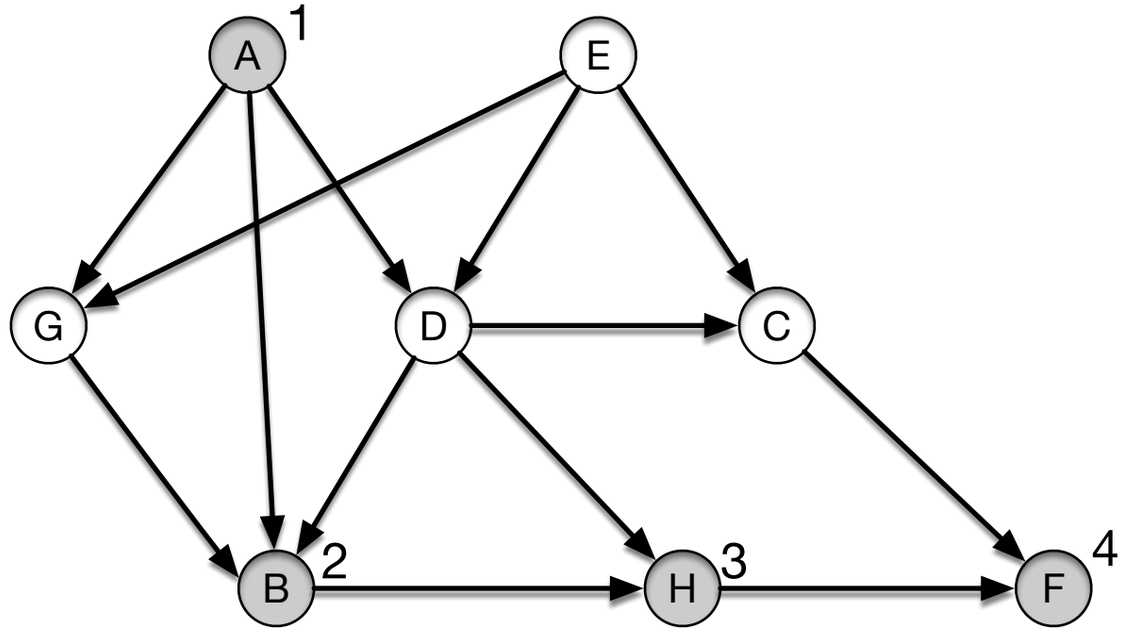



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Visit F

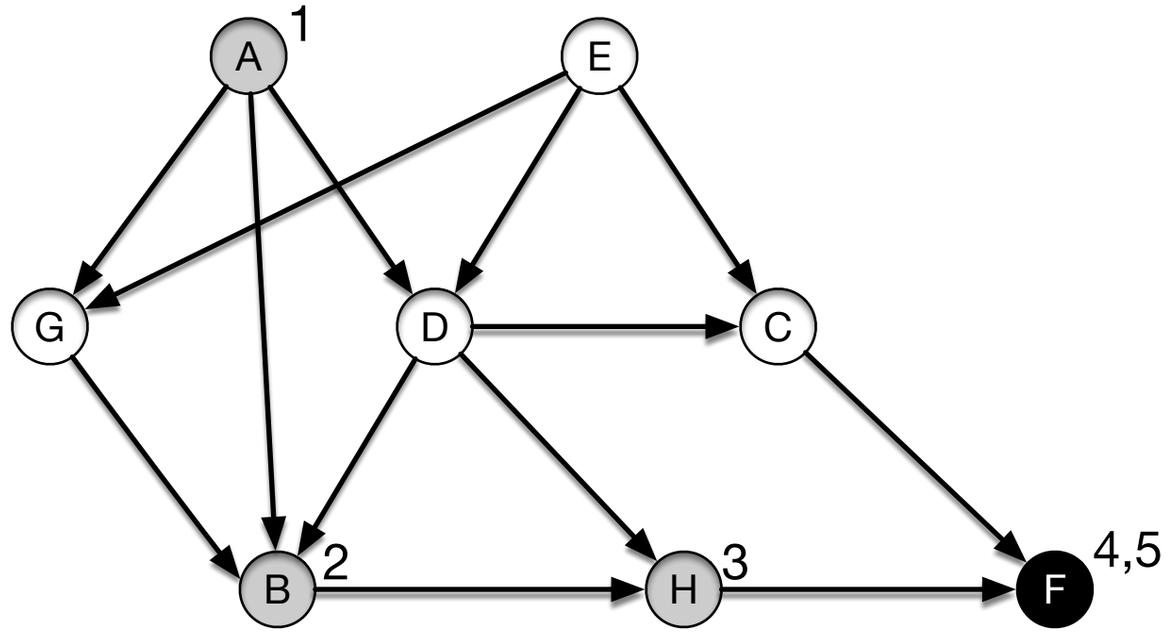



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(F)
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Finish F
- Push F at front: [F]

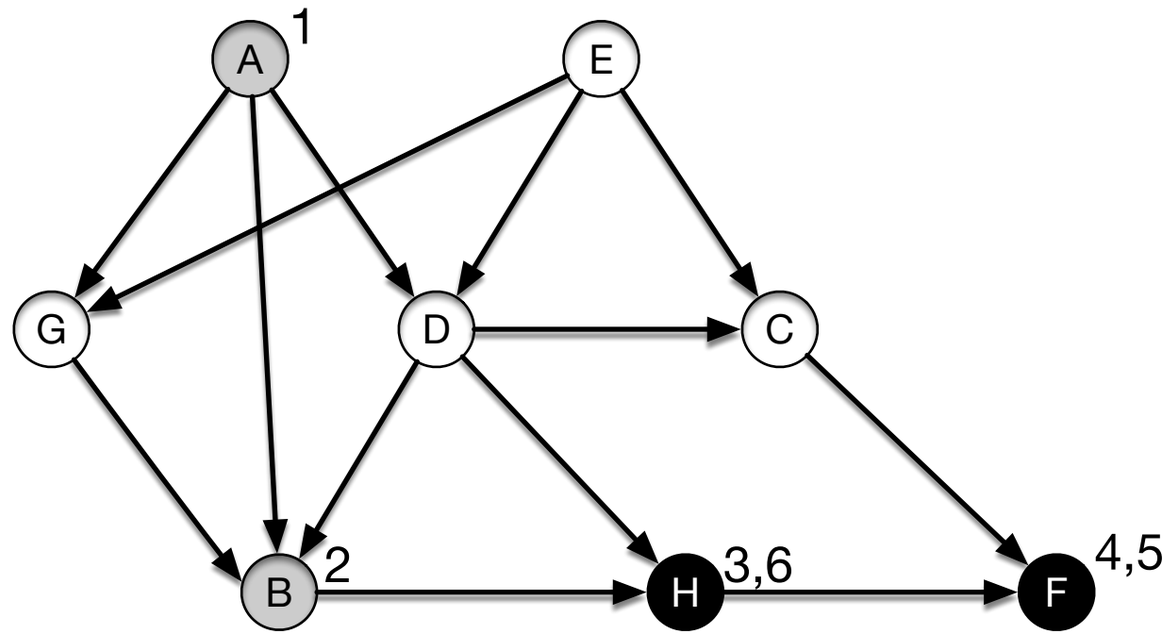



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(F)
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Finish H
- Push H at front: [H, F]

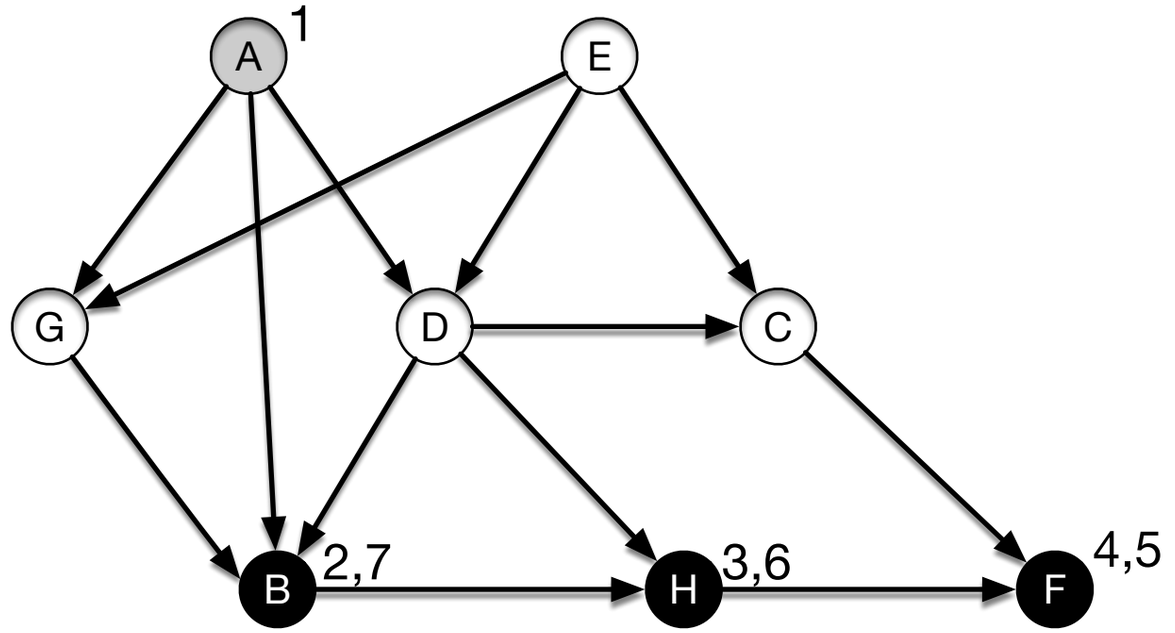



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(H)
visit(B)
visit(A)
```

Topological Sort

- Finish B
- Push B at front: [B,H, F]

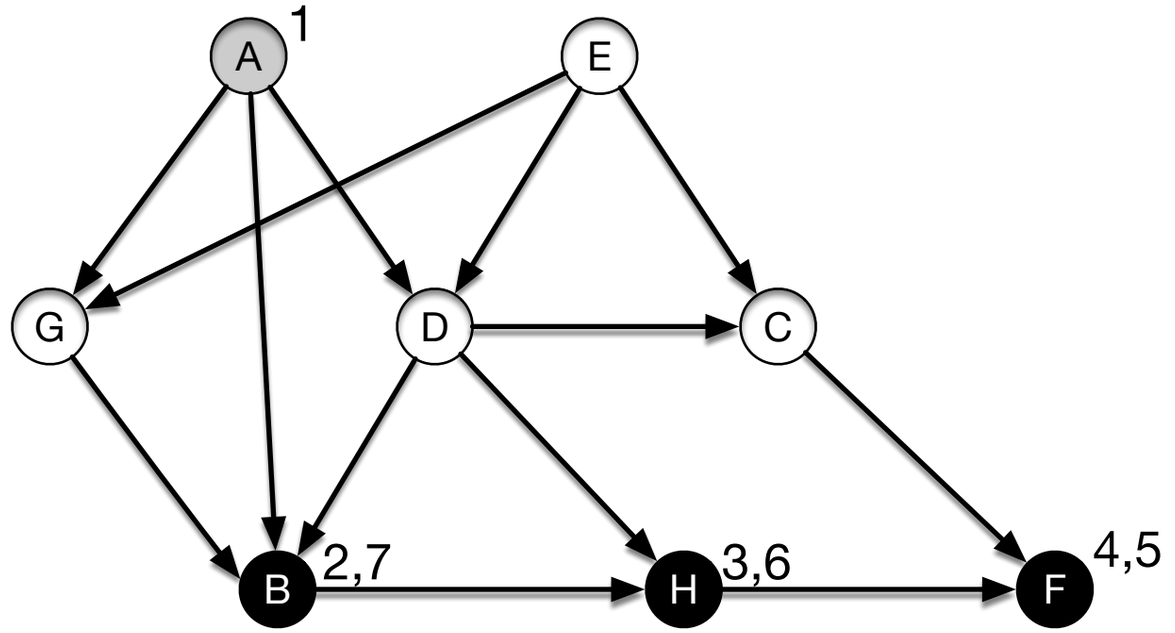



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(B)
visit(A)
```

Topological Sort

- Go back to visit A
- [B,H, F]

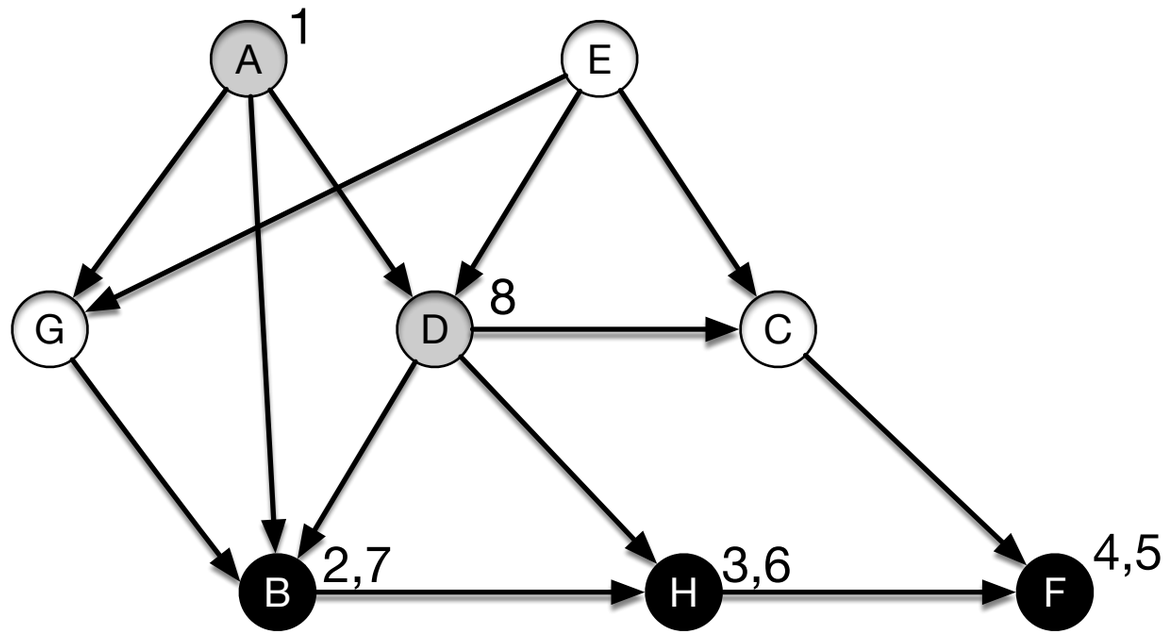


visit(A)

```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

Topological Sort

- Visit D
- [B,H, F]

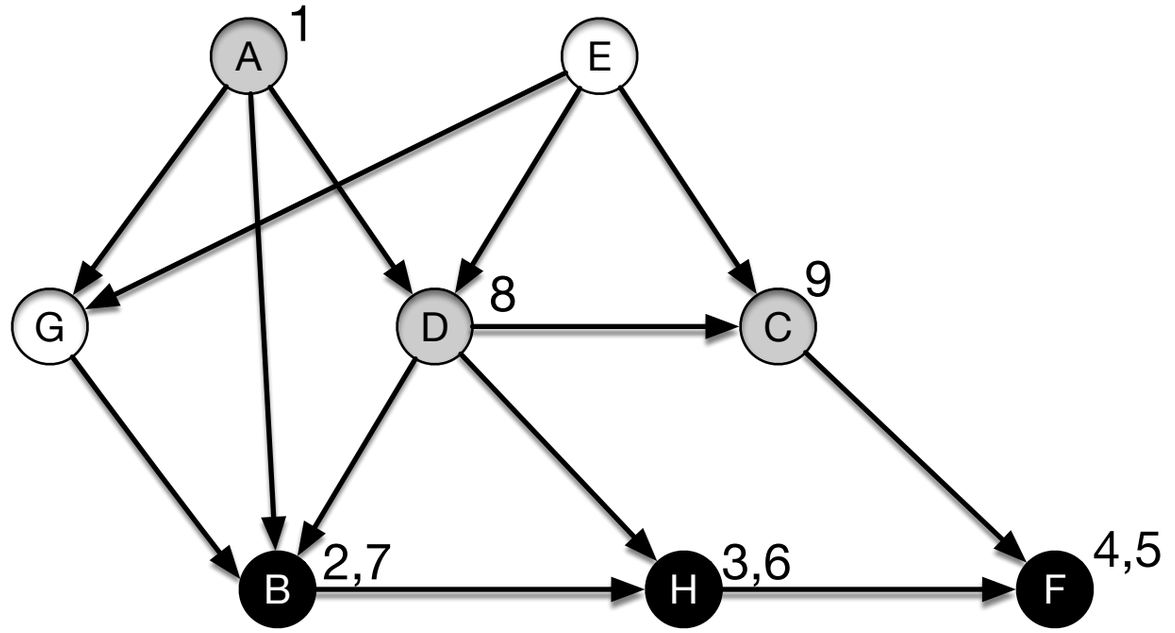



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(D)
visit(A)
```

Topological Sort

- Visit C
- [B,H, F]

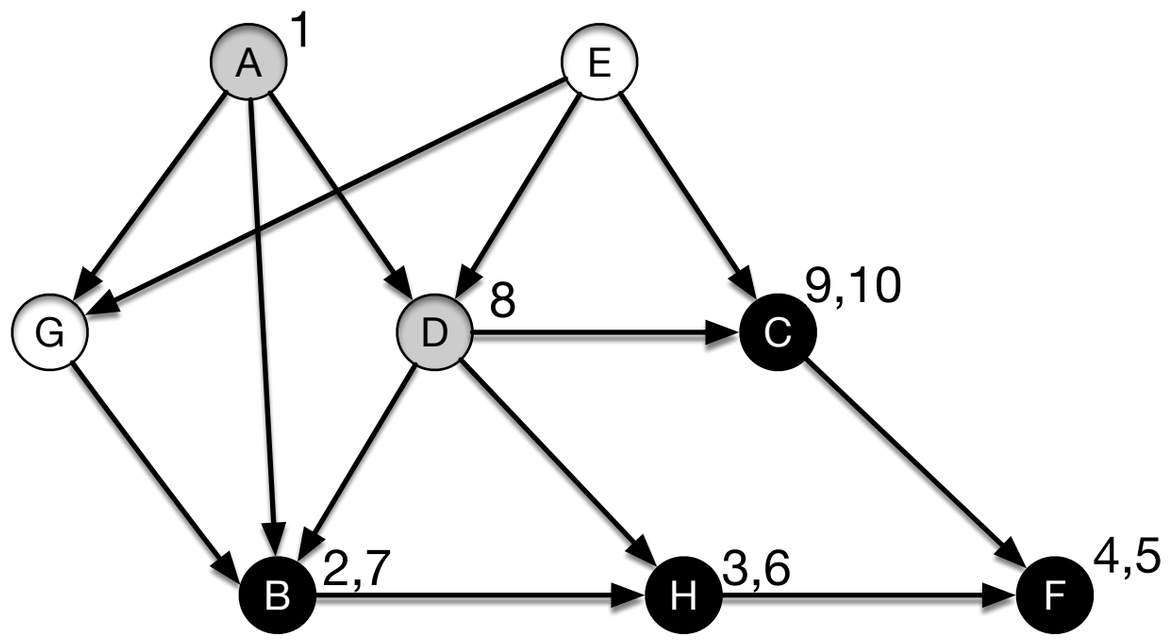



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(C)
visit(D)
visit(A)
```

Topological Sort

- Finish C
- [C, B,H, F]

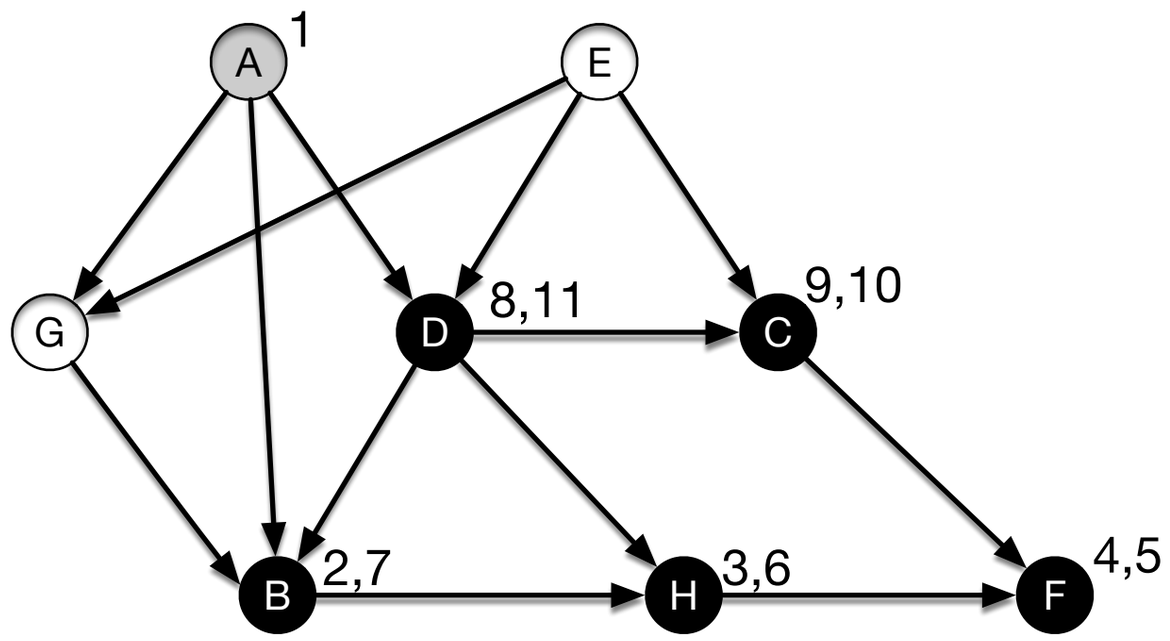



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(C)
visit(D)
visit(A)
```

Topological Sort

- Go back and finish D
- [D, C, B, H, F]

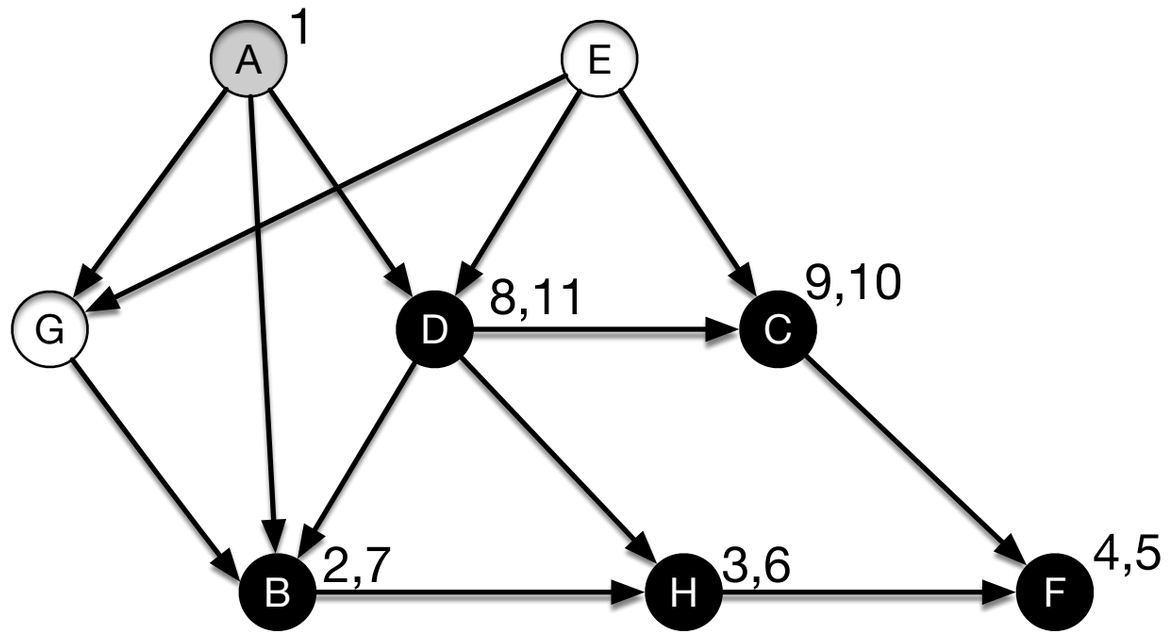



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(D)
visit(A)
```

Topological Sort

- We are back to visit A
- Next node is G
- [D, C, B, H, F]

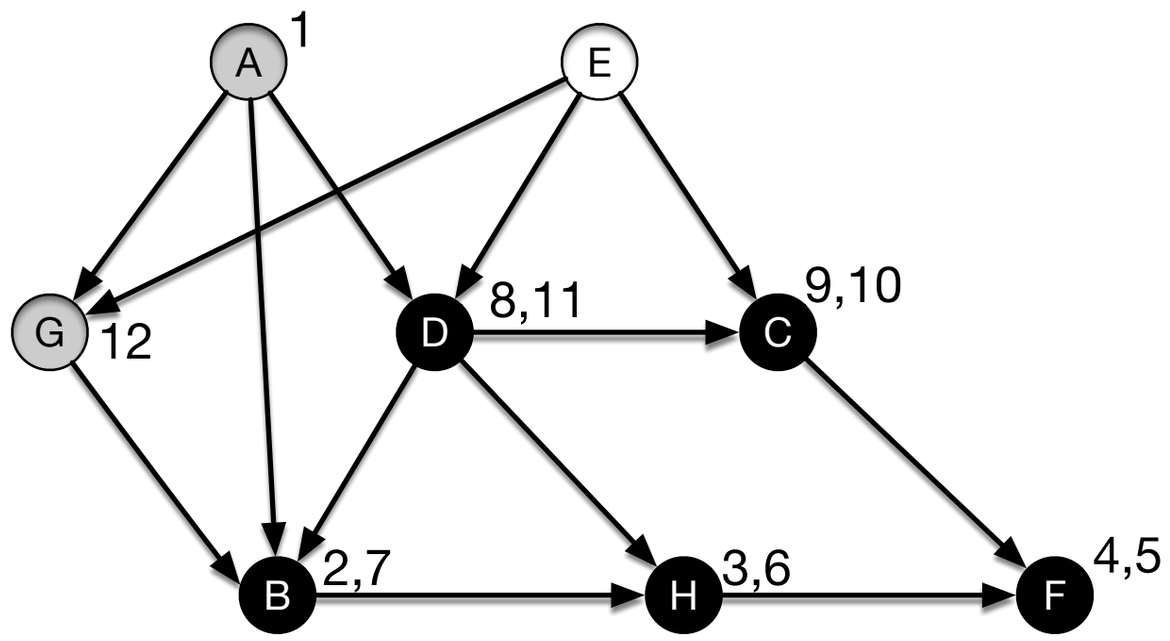



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

`visit(A)`

Topological Sort

- Visit G
- [D, C, B, H, F]

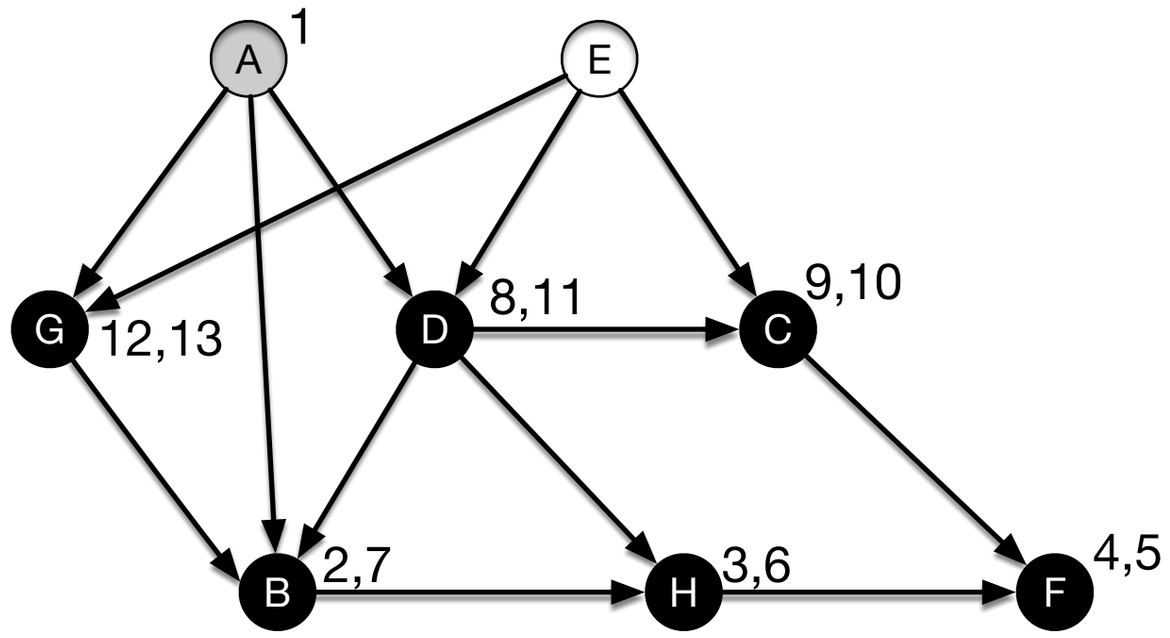



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(G)
visit(A)
```

Topological Sort

- Finish G
- [G, D, C, B, H, F]

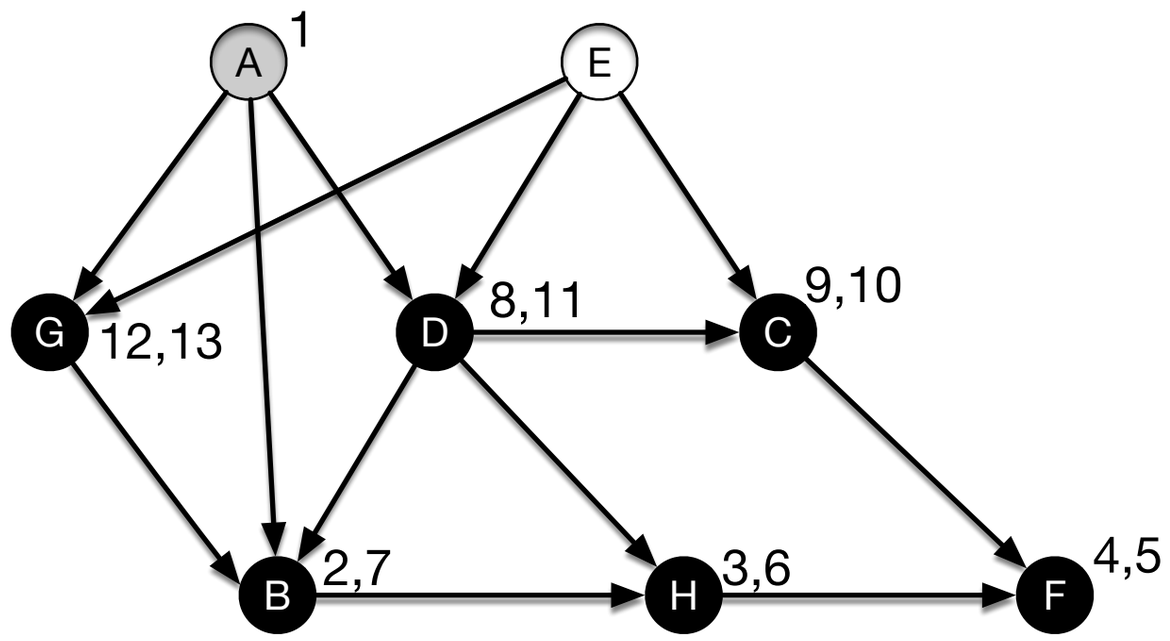



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

```
visit(G)
visit(A)
```

Topological Sort

- Go back to A
- [G, D, C, B, H, F]

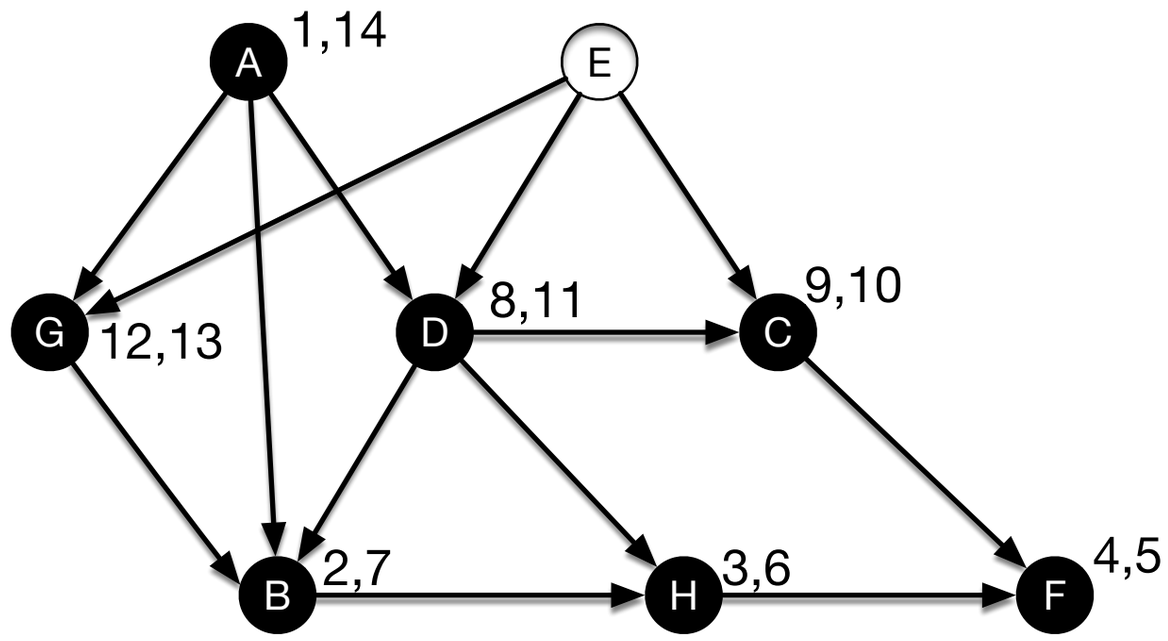



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

visit(A)

Topological Sort

- Finish A
- [A, G, D, C, B, H, F]

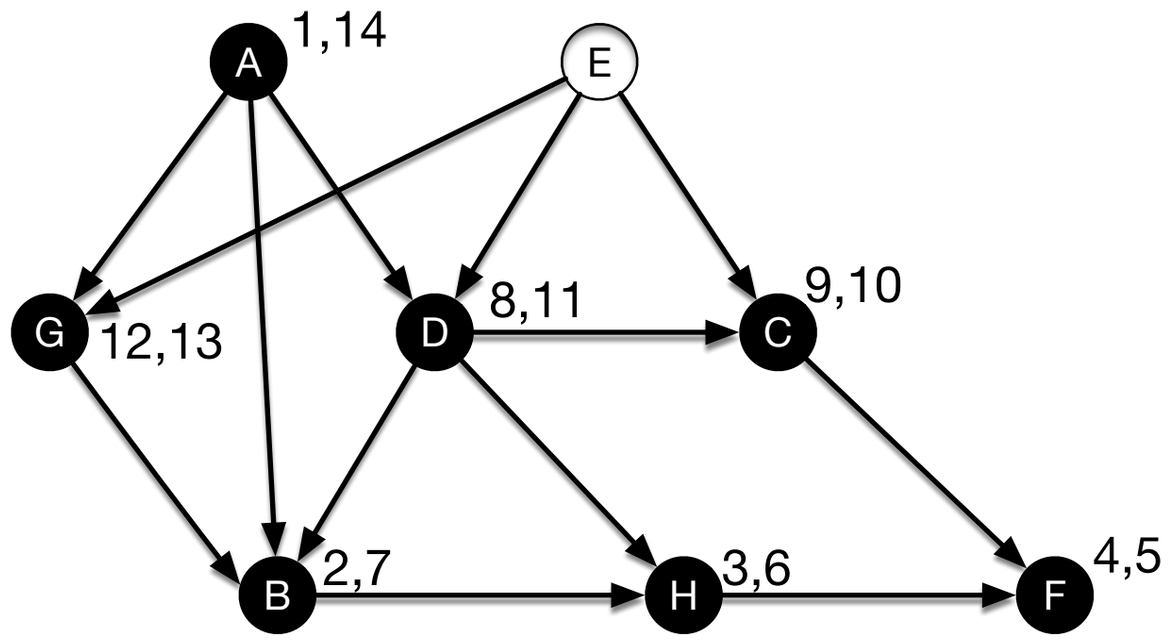



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

∅

Topological Sort

- Done with visit(A)
- [A, G, D, C, B, H, F]

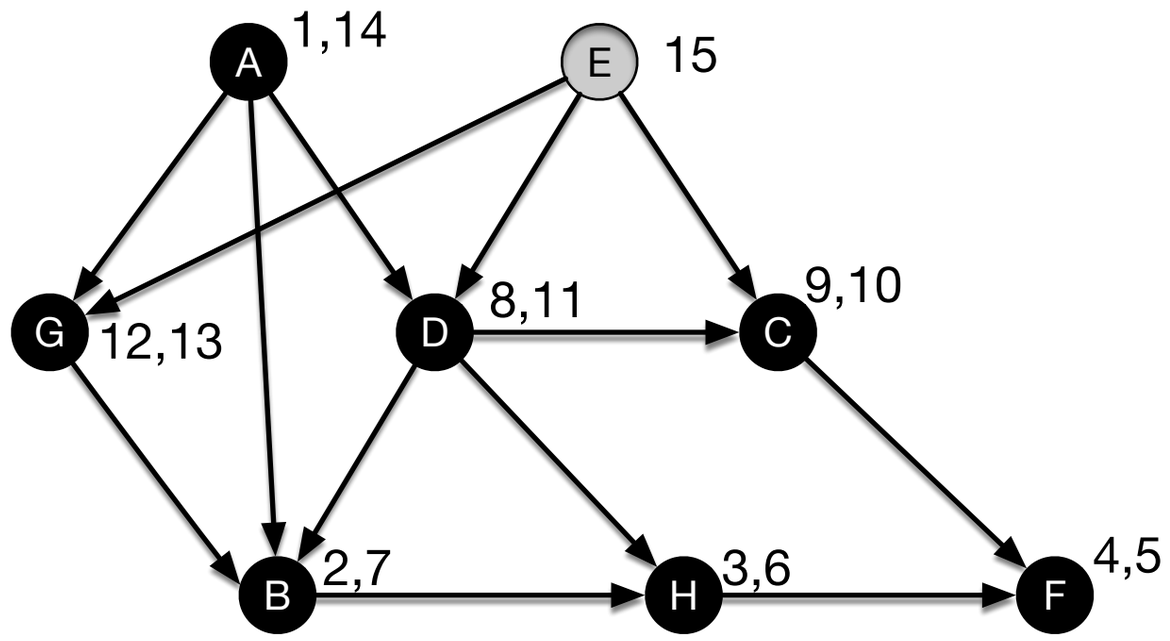



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

∅

Topological Sort

- One white node left: E
- Visit E
- [A, G, D, C, B, H, F]

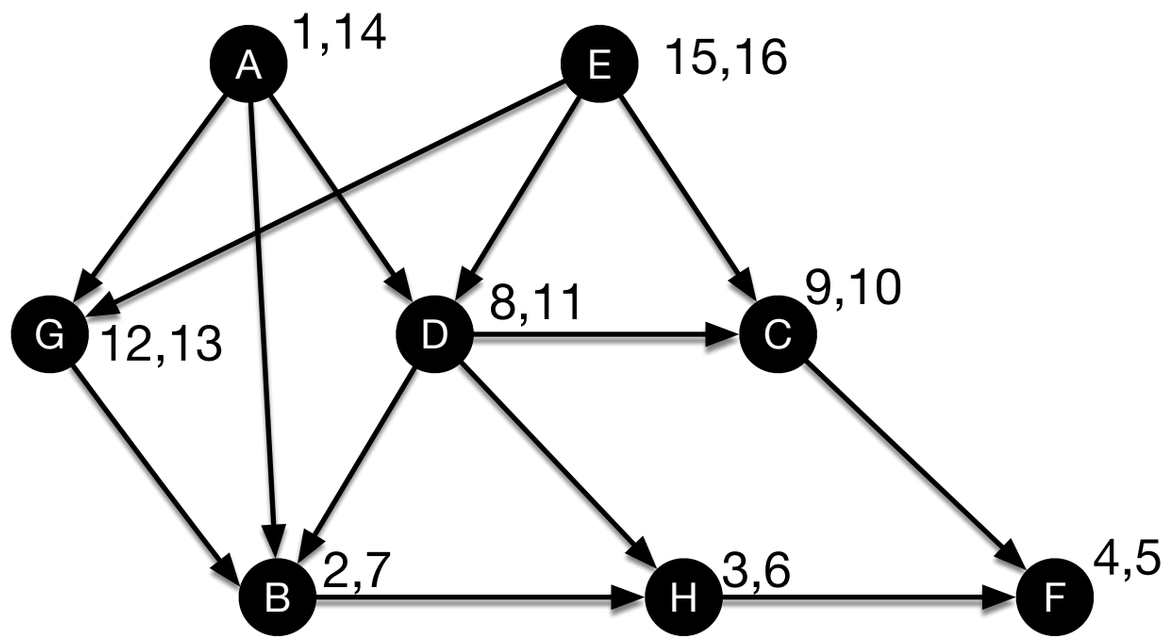



```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

visit(E)

Topological Sort

- Finish E
- [E, A, G, D, C, B, H, F]




```
A: B,D,G
B: H
C: F
D: B,H
E: C,D,G
F:
G: B
H: F
```

Topological Sort

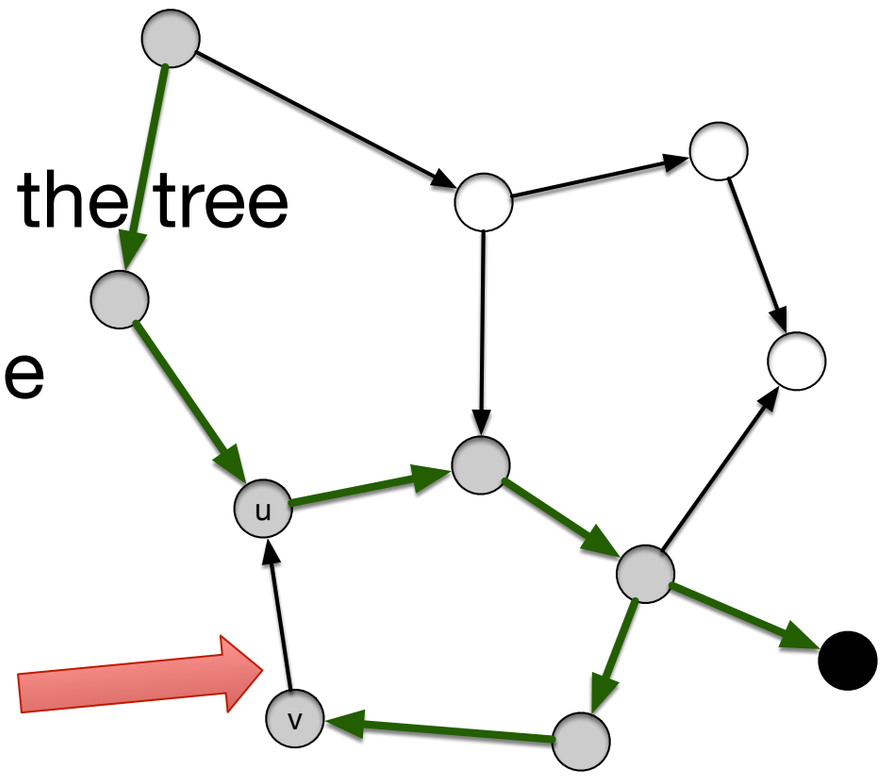
- Key observation from the examples:
  - We have a cycle if we ever try to visit a gray node

Topological Sort

- Lemma: A directed graph $G = (V, E)$ is acyclic if and only if a DFS of G yields no back edges

Topological Sort

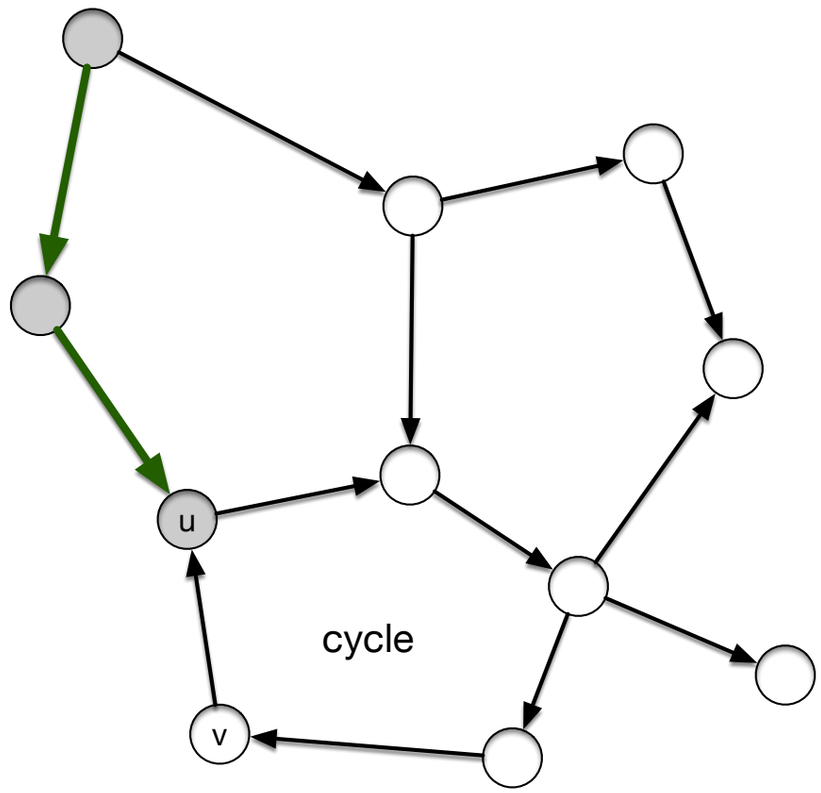
- Proof: "⇒"
  - If DFS produces a back-edge (v, u) then u is an ancestor of v
    - There is a path from u to v in the tree
    - The edge (v, u) closes a cycle
      - from u to v back to u



visiting v and discovering a gray node

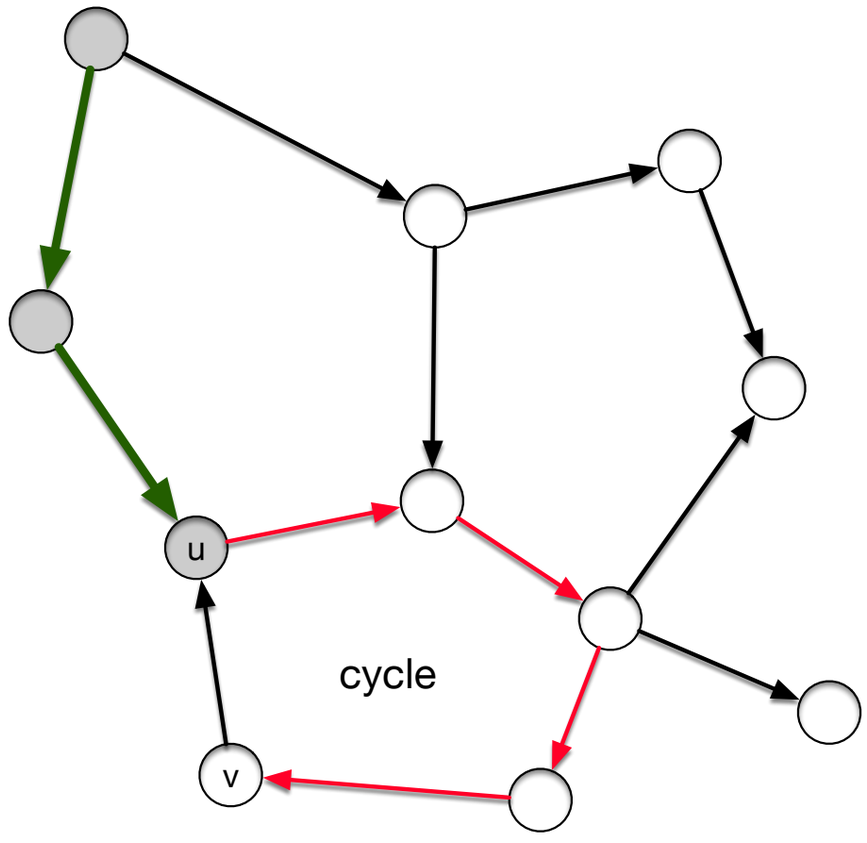
Topological Sort

- Proof: "$\Leftarrow$"
  - Suppose G has a cycle
  - Let u be the first vertex in the cycle to be discovered



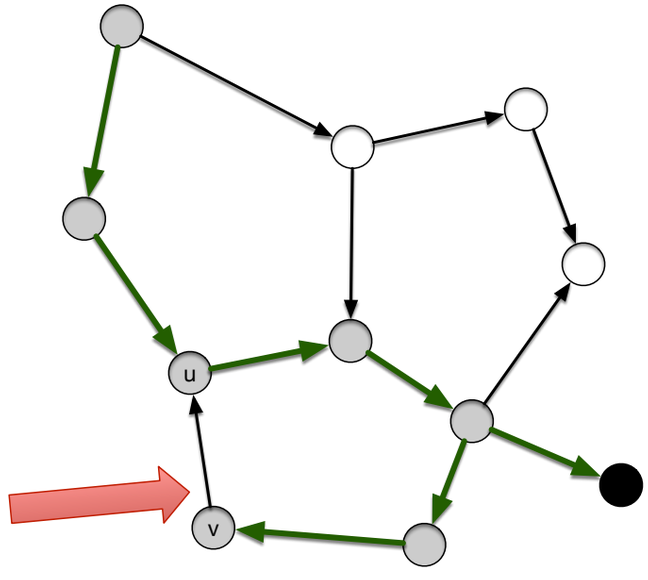
Topological Sort

- All other vertices in the cycle are white and there is a white-path to the node v just in front of u



Topological Sort

- By the white-path theorem:
  - We will discover v from u
    - (Though not necessarily through the cycle since there might be more cycles)
- Thus, (v, u) is a back edge



visiting v and discovering a gray node

Topological Sort

- Theorem: DFS gives a topological sort or discovers a cycle
- Proof:
  - Need to show:
    - If DFS does not discover a cycle, then for each edge (u, v), we have $u.f > v.f$

Topological Sort

- Proof:
  - At the time that we are first looking at (u, v):
    - v cannot be gray, because then we would have a back-edge

Topological Sort

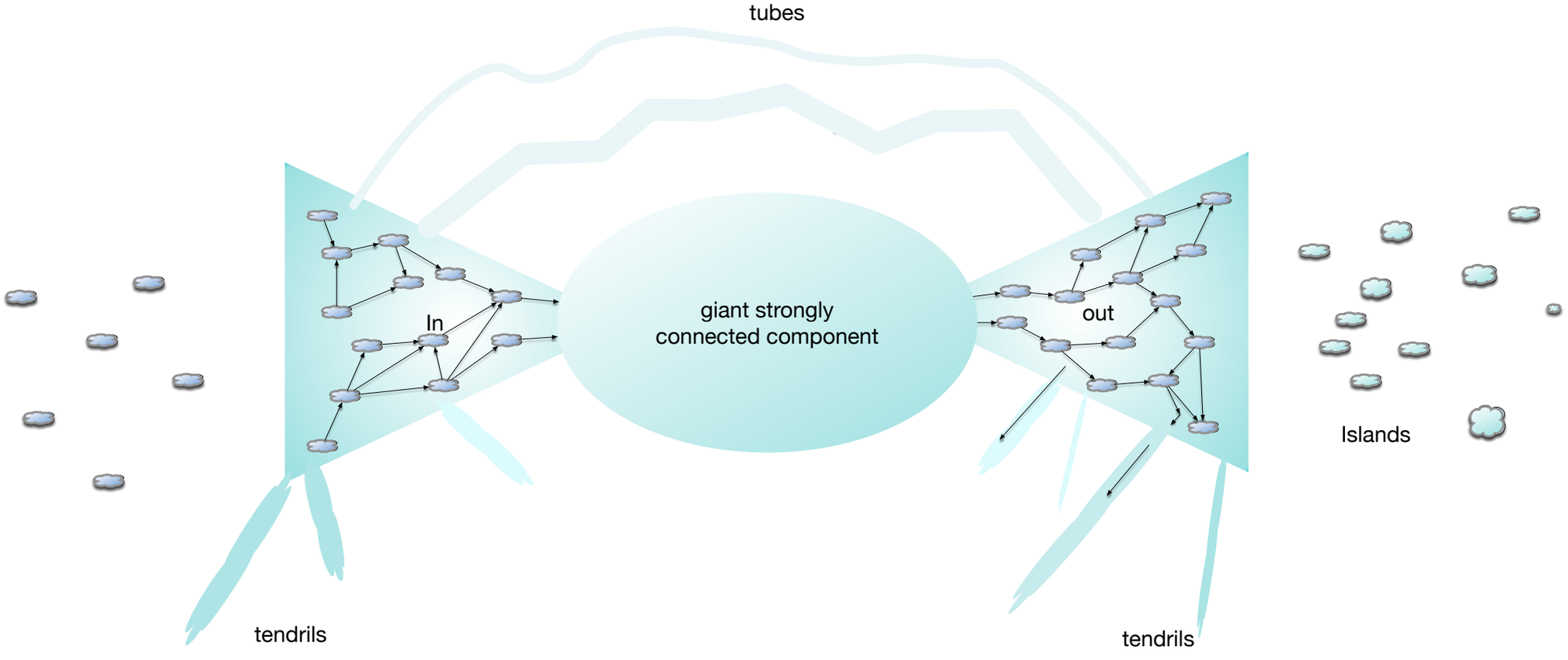
- At the time that we are first looking at (u, v):
  - If v is white:
    - Then by the white path theorem, u becomes an ancestor of v
  - By the parenthesis theorem $v.f < u.f$

Topological Sort

- Proof:
  - At the time that we are first looking at (u, v):
    - If v is black, then u is still be visited, so
      - u is not yet black
      - so, $u.f > v.f$
    - qed

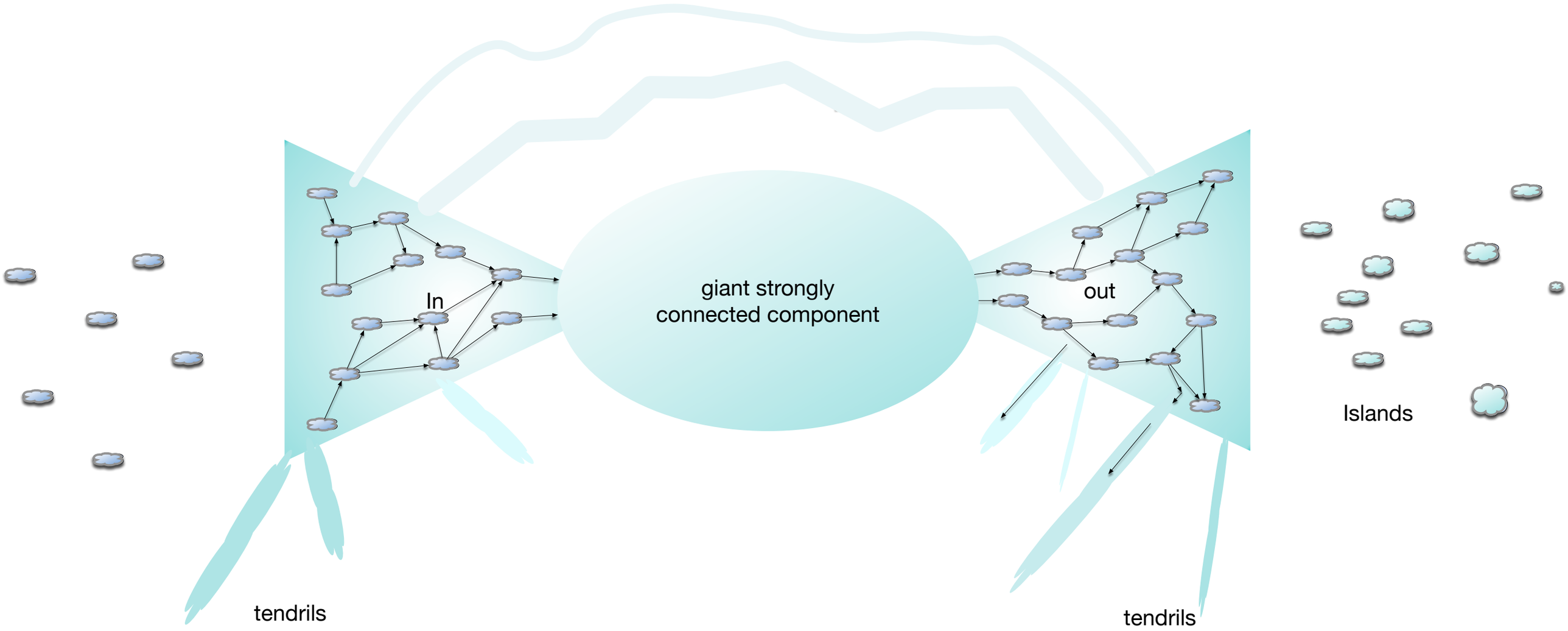
Strongly Connected Components

- WWW graph:
  - Nodes: pages
  - Edges: links from one page to another page
- Broder et al. study (2000): 200 million pages and 1.5 billion links



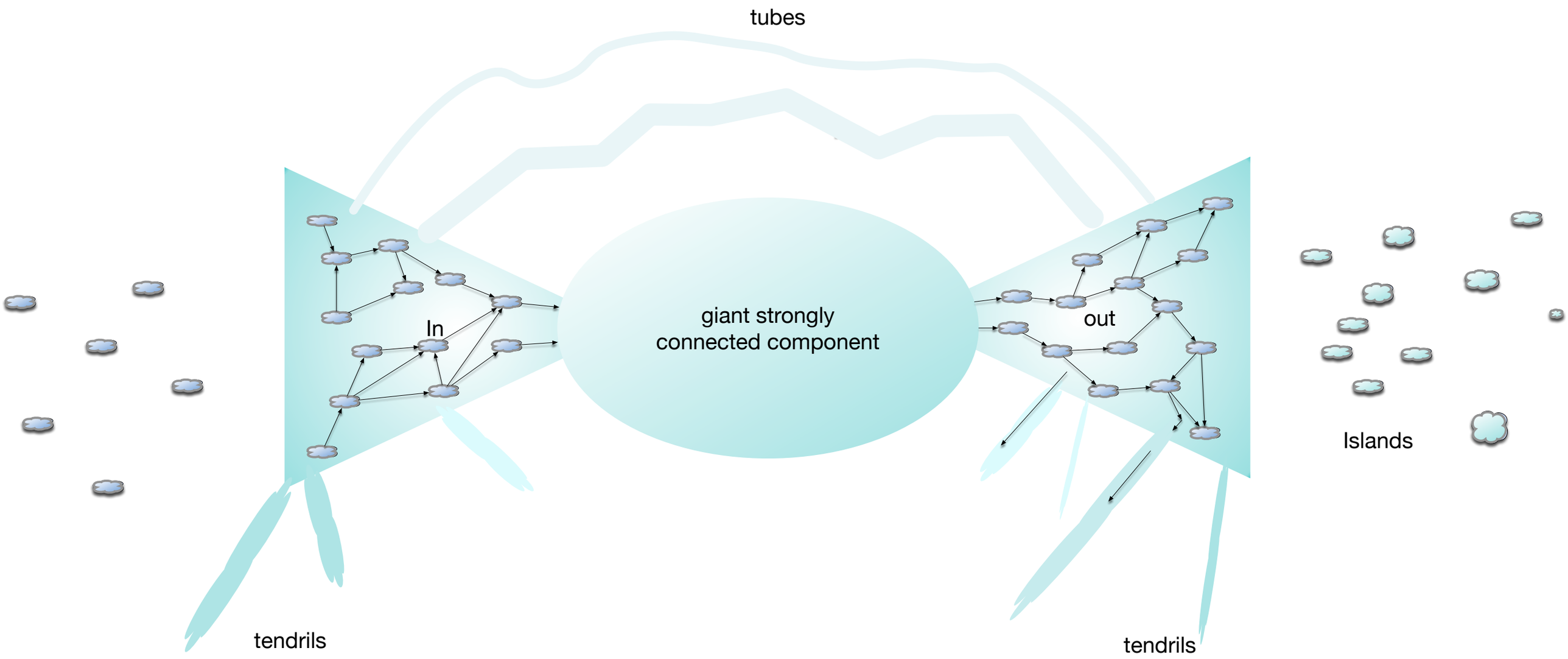
Strongly Connected Components

- Bowtie:
- Strongly connected component at the center of the WWW (28%) of all nodes



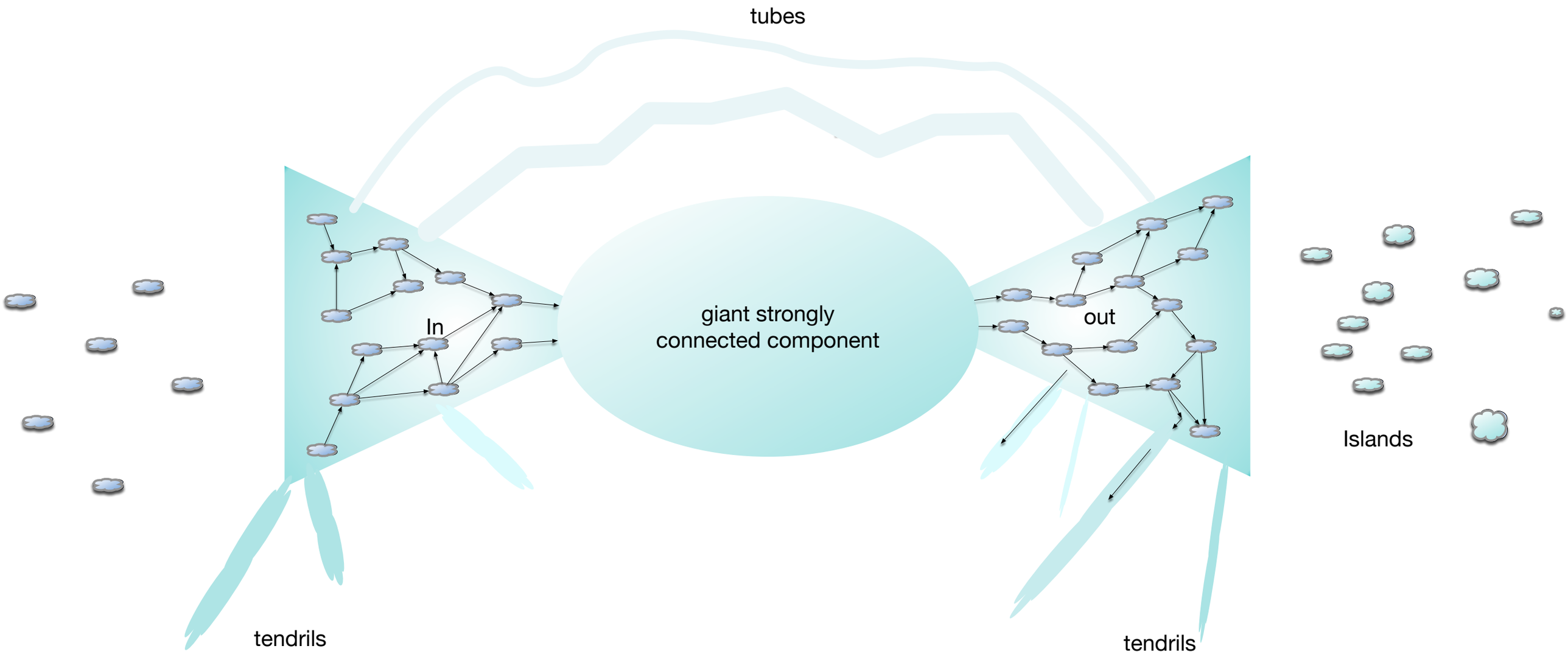
Strongly Connected Components

- Islands: Isolated areas of the web



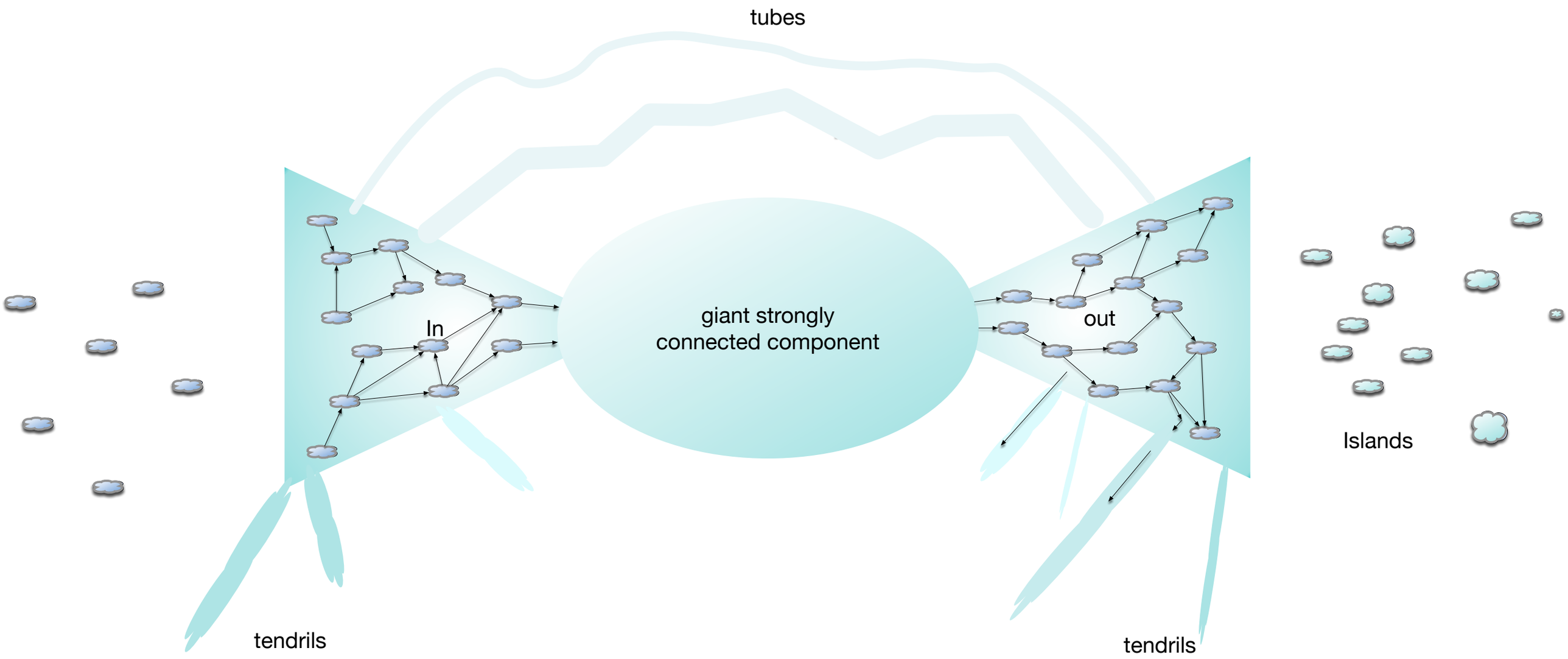
Strongly Connected Components

- In: Possible to reach the giant
- Out: Reachable from the giant



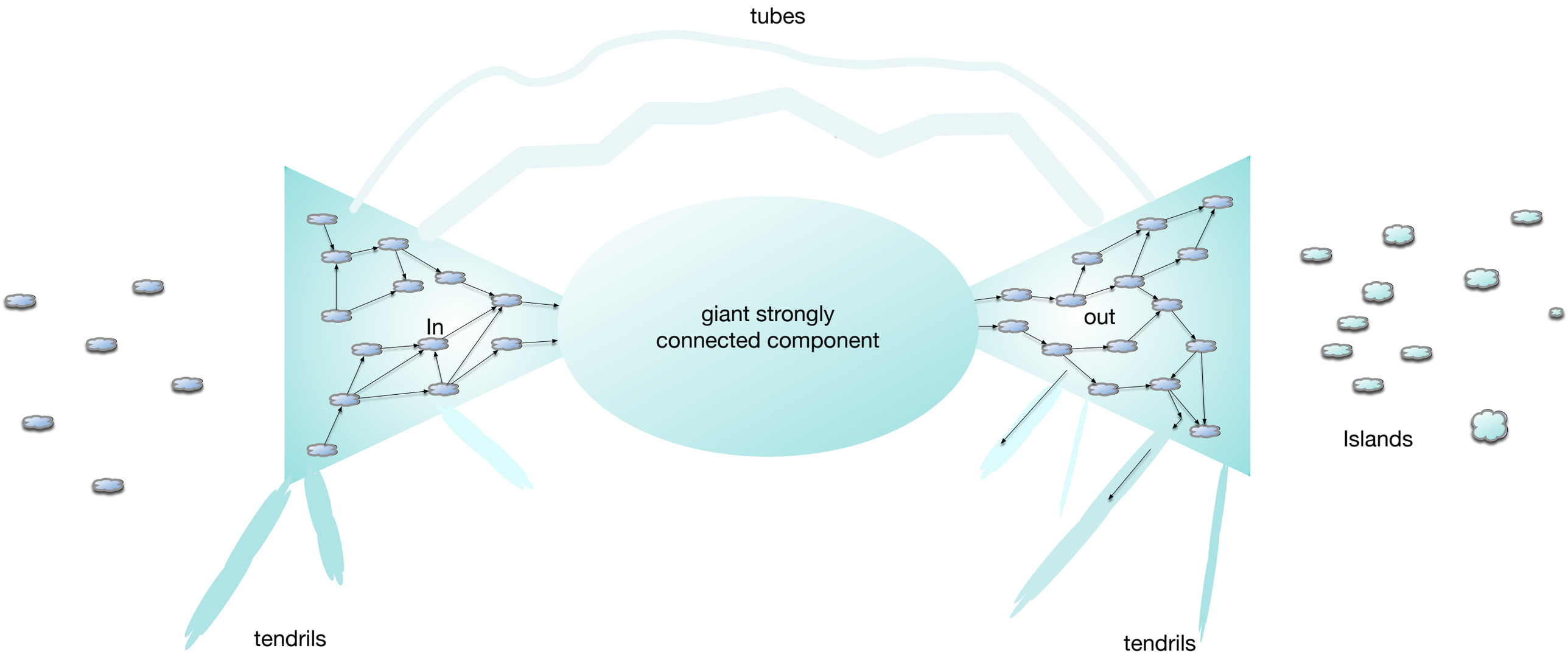
Strongly Connected Components

- Weird stuff: Tubes that move from In to Out bypassing the giant



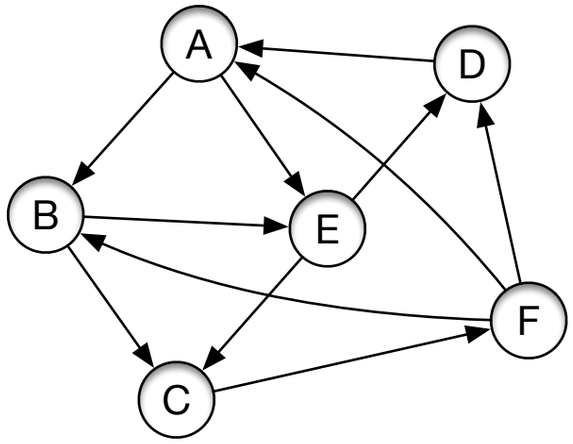
Strongly Connected Components

- Weird stuff: Tendrils to In and tendrils to Out



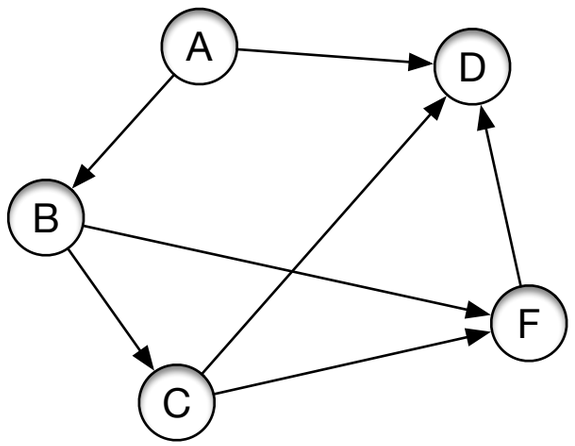
Strongly Connected Components

- Strongly connected component:
  - Can reach any vertex from any other vertex



Strongly Connected Components

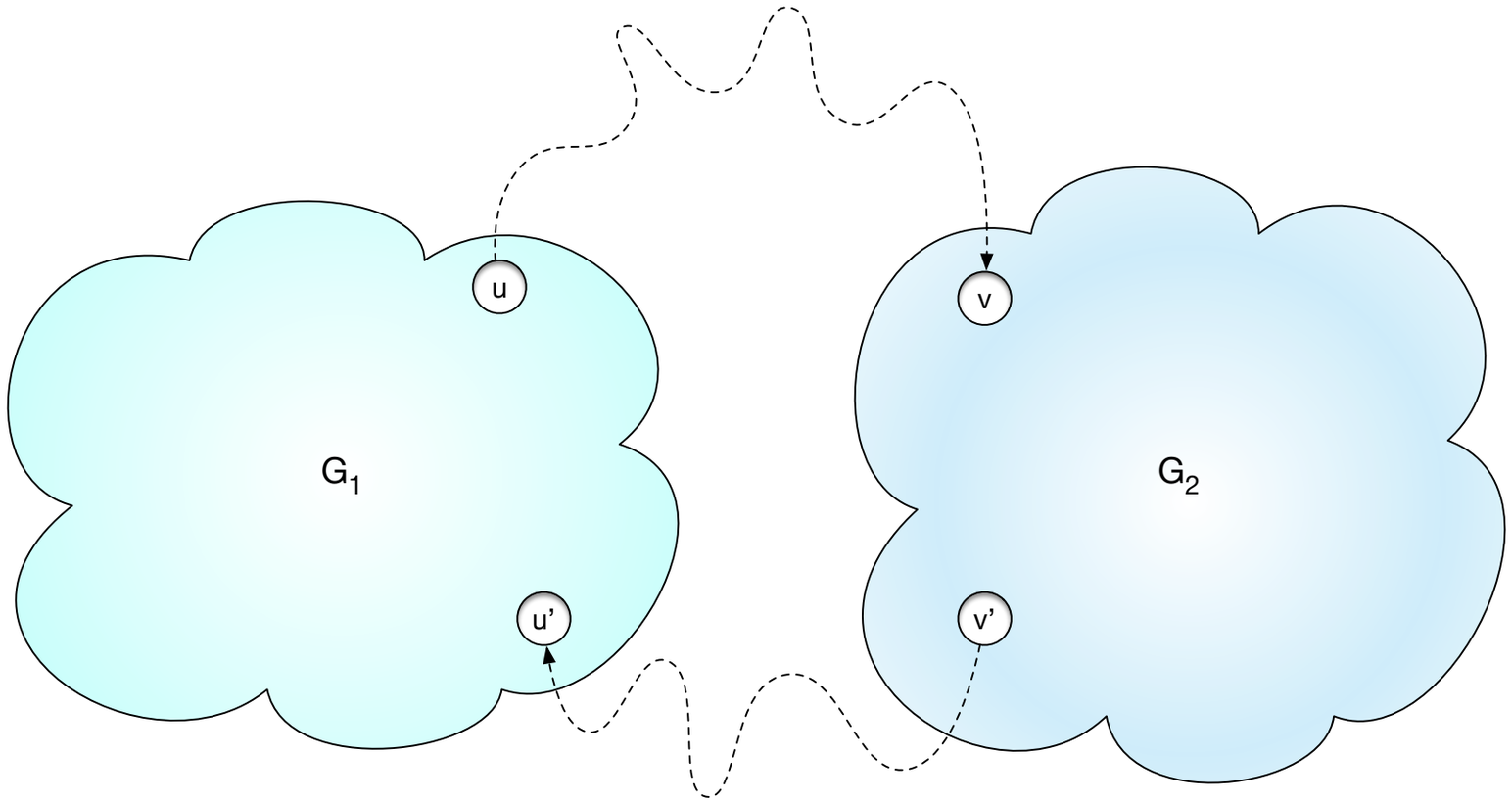
- Strongly connected component
  - This is **NOT** strongly connected



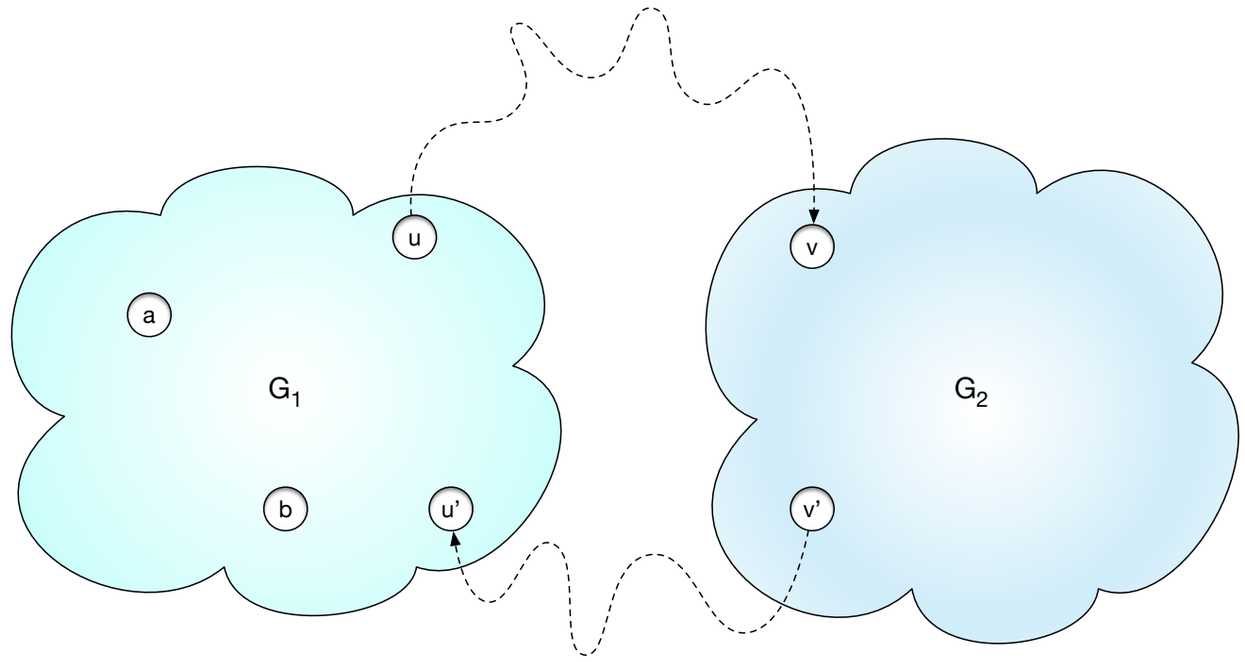
- There is no way to get from D to A

Strongly Connected Components

- Lemma: Let G_1 and G_2 be two strongly connected subgraphs of a graph G and assume that there is a path from a vertex in G_1 to a vertex in G_2 and also a path from a vertex of G_2 to G_1, then $G_1 \cup G_2$ is strongly connected

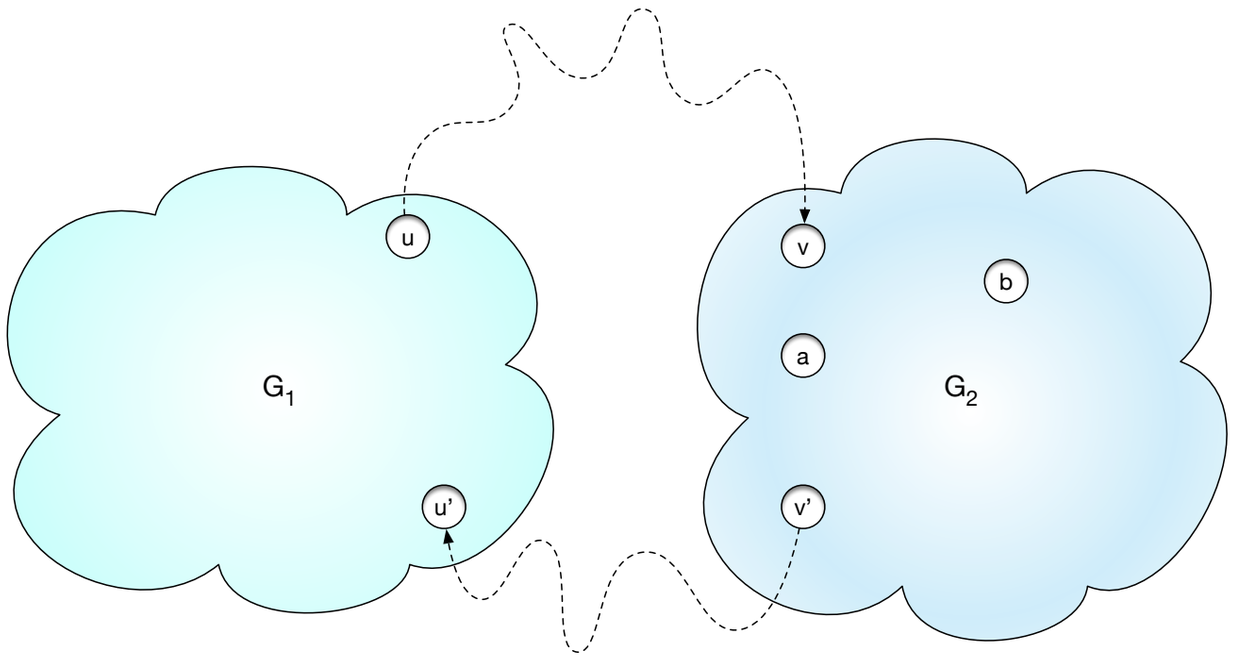


Strongly Connected Components



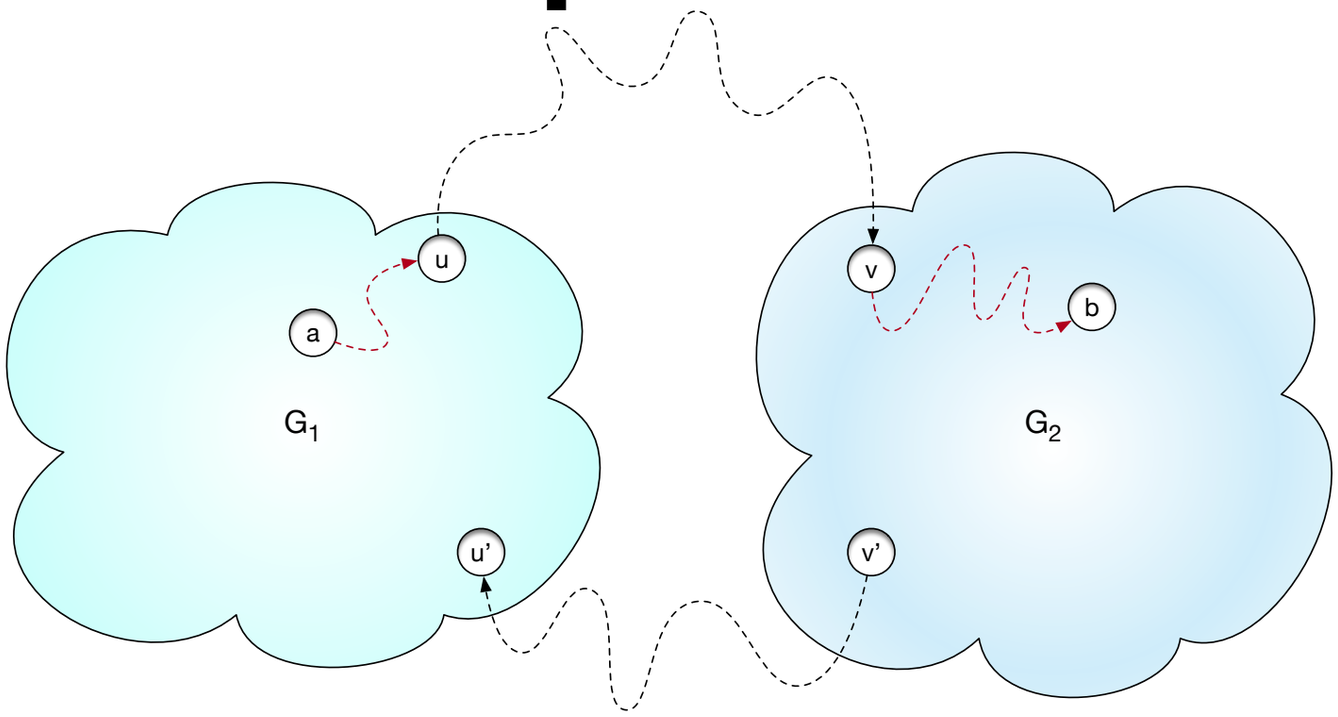
- Proof: Take two nodes a and b in $G_1 \cup G_2$.
  - If both are in G_1 then there is a path between a and b because they are in G_1

Strongly Connected Components



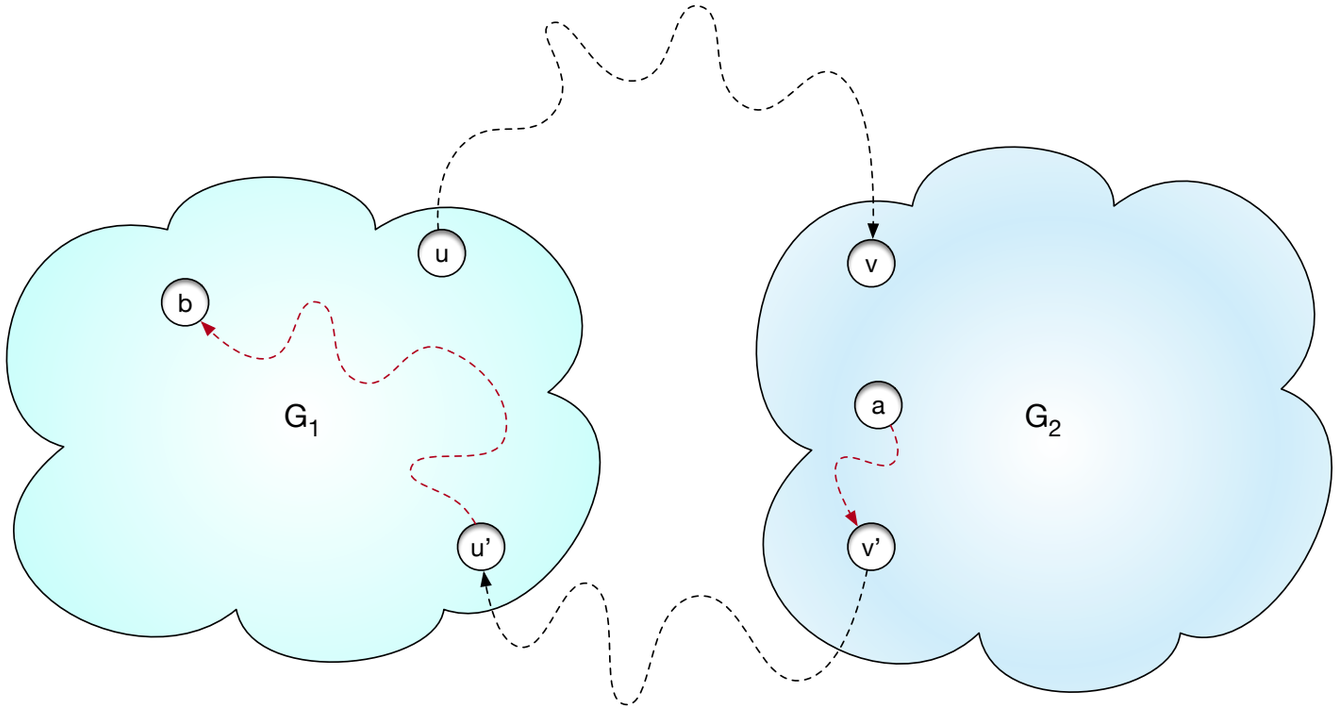
- Proof: Take two nodes a and b in $G_1 \cup G_2$.
  - If both are in G_2 then there is a path between a and b because they are in G_2

Strongly Connected Components



- If $a \in V(G_1)$ and $b \in V(G_2)$, then we can move from a to u and from u to v and then from v to b.
- After removing cycles, this is now a path from a to b

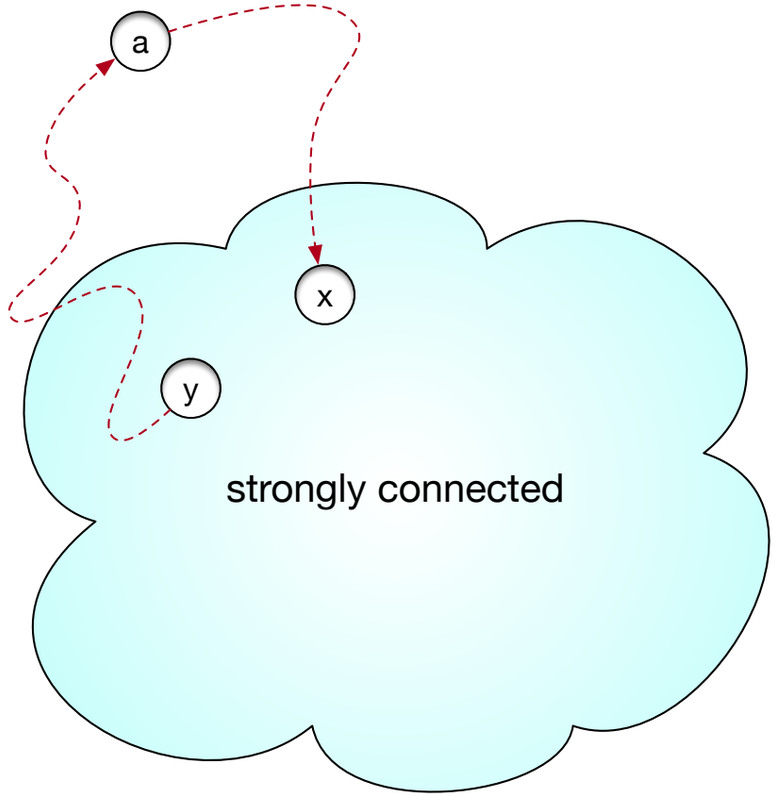
Strongly Connected Components



- Similarly, if $a \in V(G_2)$ and $b \in V(G_1)$, then we can move from a to v' and from v' to u' and then from u' to b.
- After removing cycles, this is now a path from a to b

Strongly Connected Components

- A single node is a strongly connected subgraph
- For each strongly connected subgraph, we can try to grow by adding other nodes
- If a node a has a path to and from a strongly connected subgraph, then by the lemma, we can add the node and get a bigger strongly connected subgraph

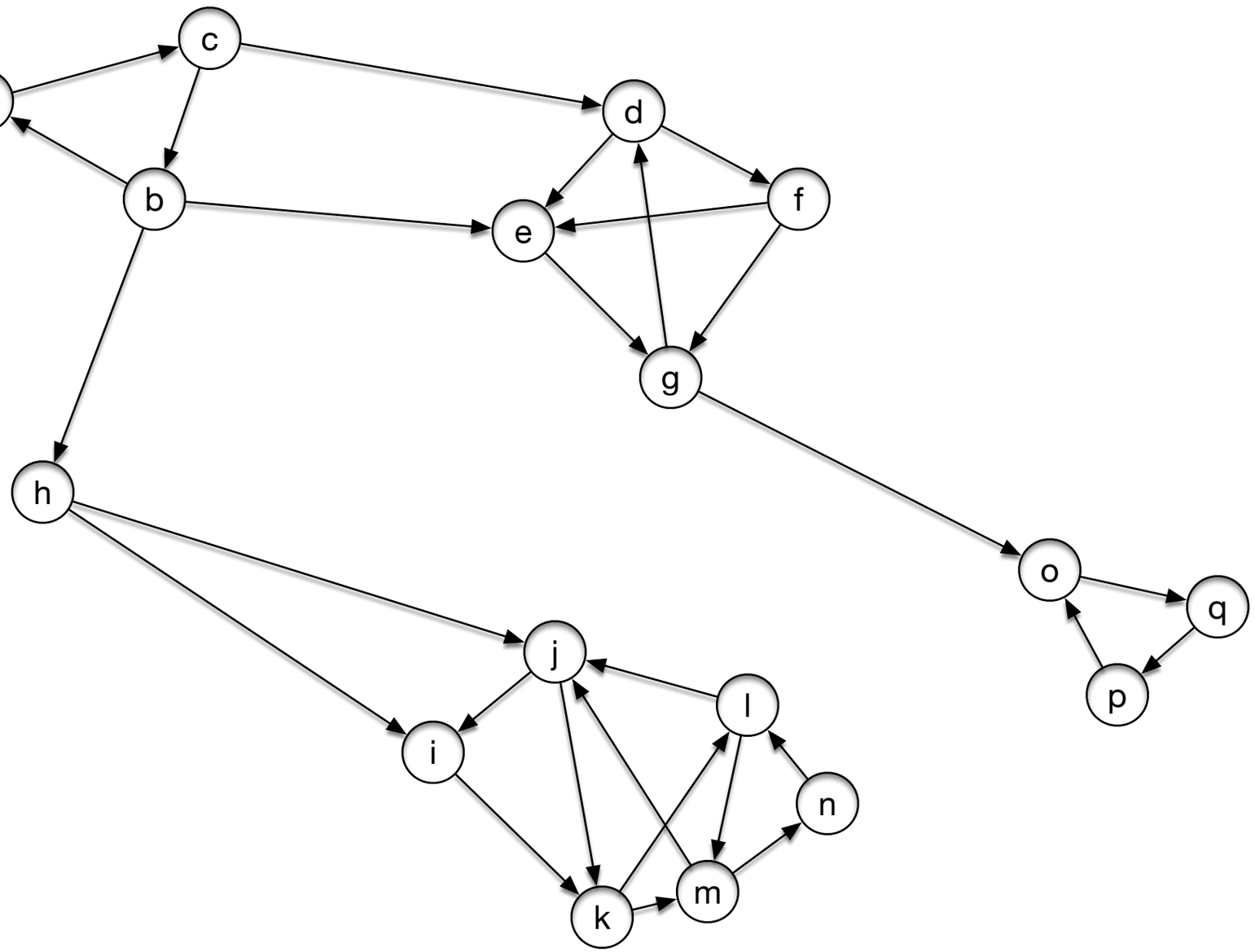


Strongly Connected Components

- Strongly connected component : A maximal strongly connected subgraph
- The nodes of any directed graph can be divided into strongly connected components

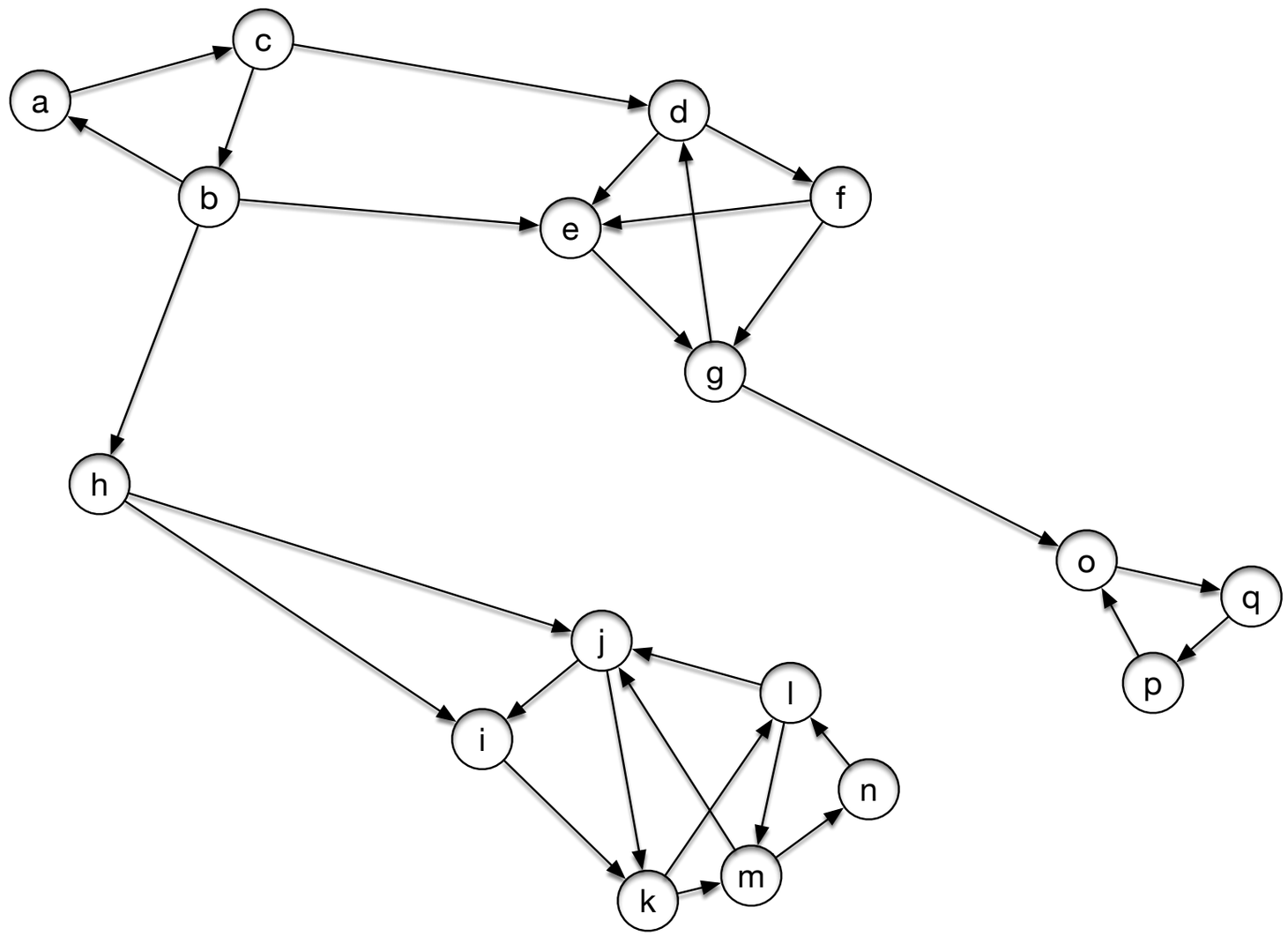
Strongly Connected Components

- Example:



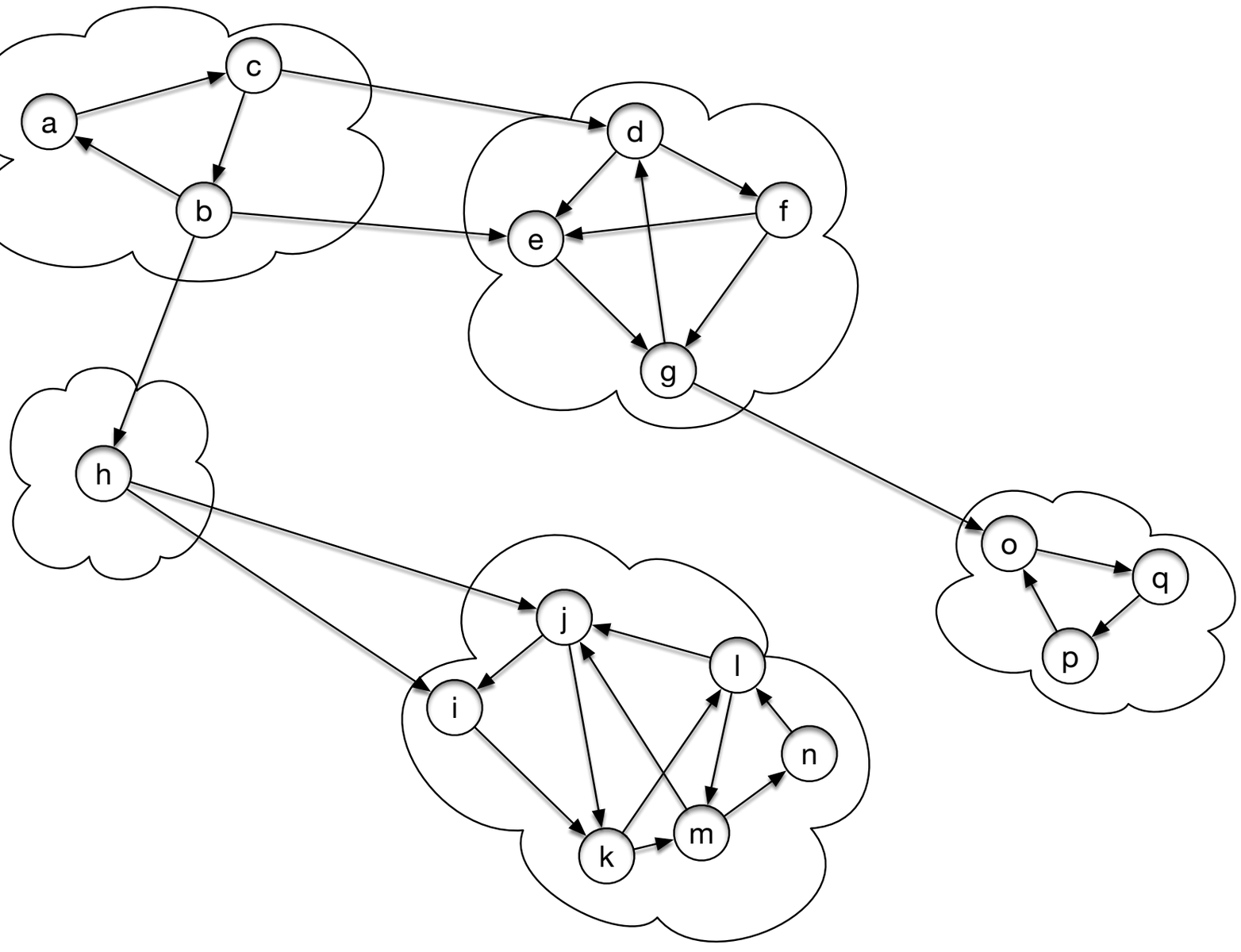
Strongly Connected Components

- Try it out by growing from individual nodes



Strongly Connected Components

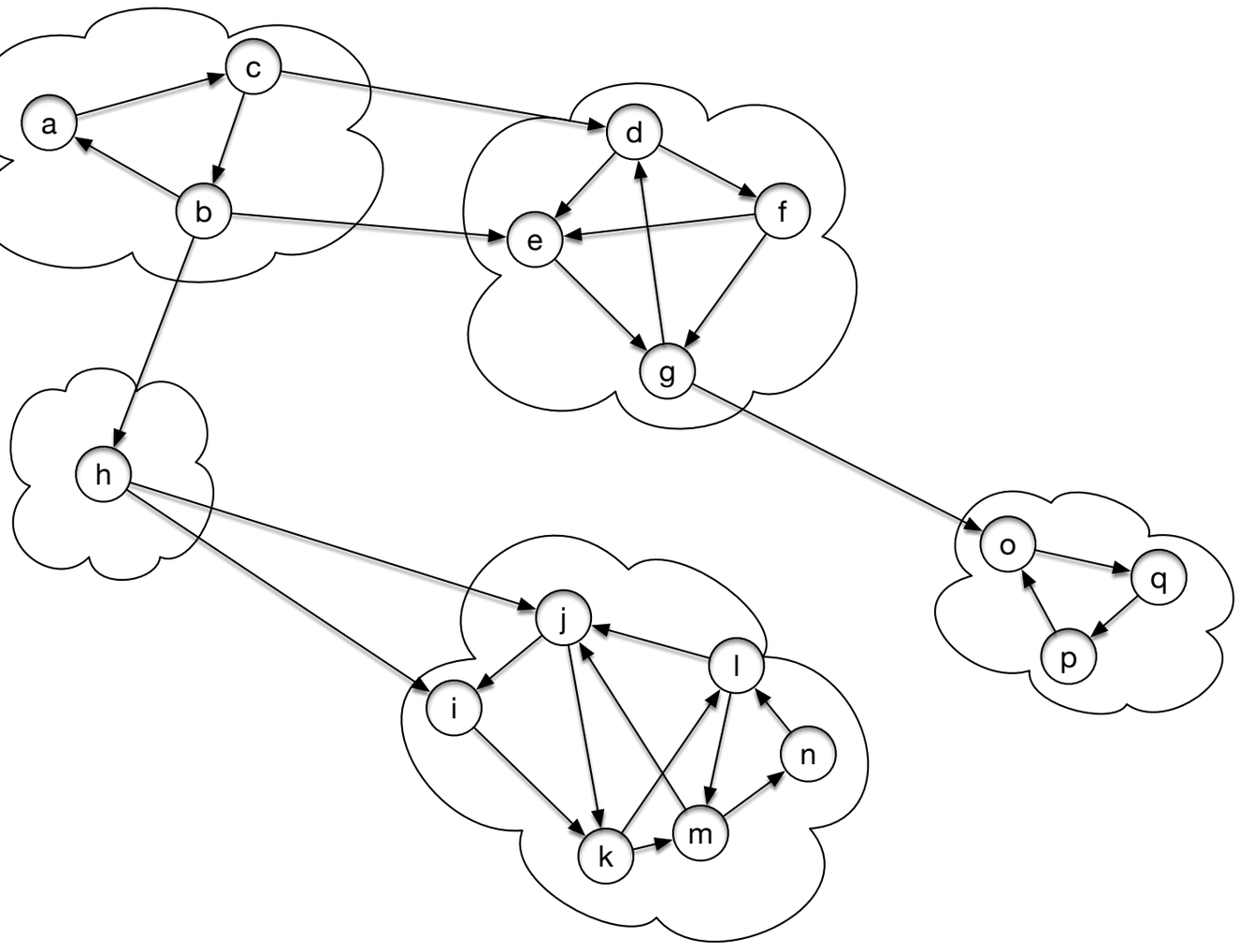
- Result:

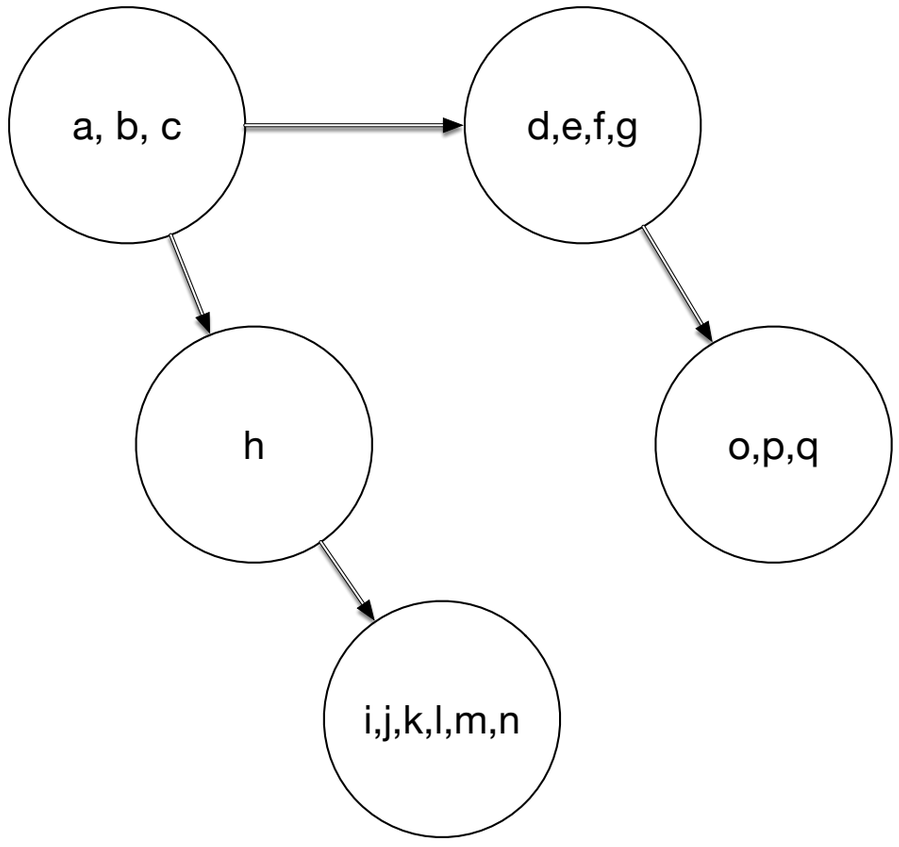


Strongly Connected Components

- If we only look at the connected components we get the SCC metagraph
  - Nodes are the strongly connected components
  - Edges represent the existence of an edge from one component to the next

Strongly Connected Components



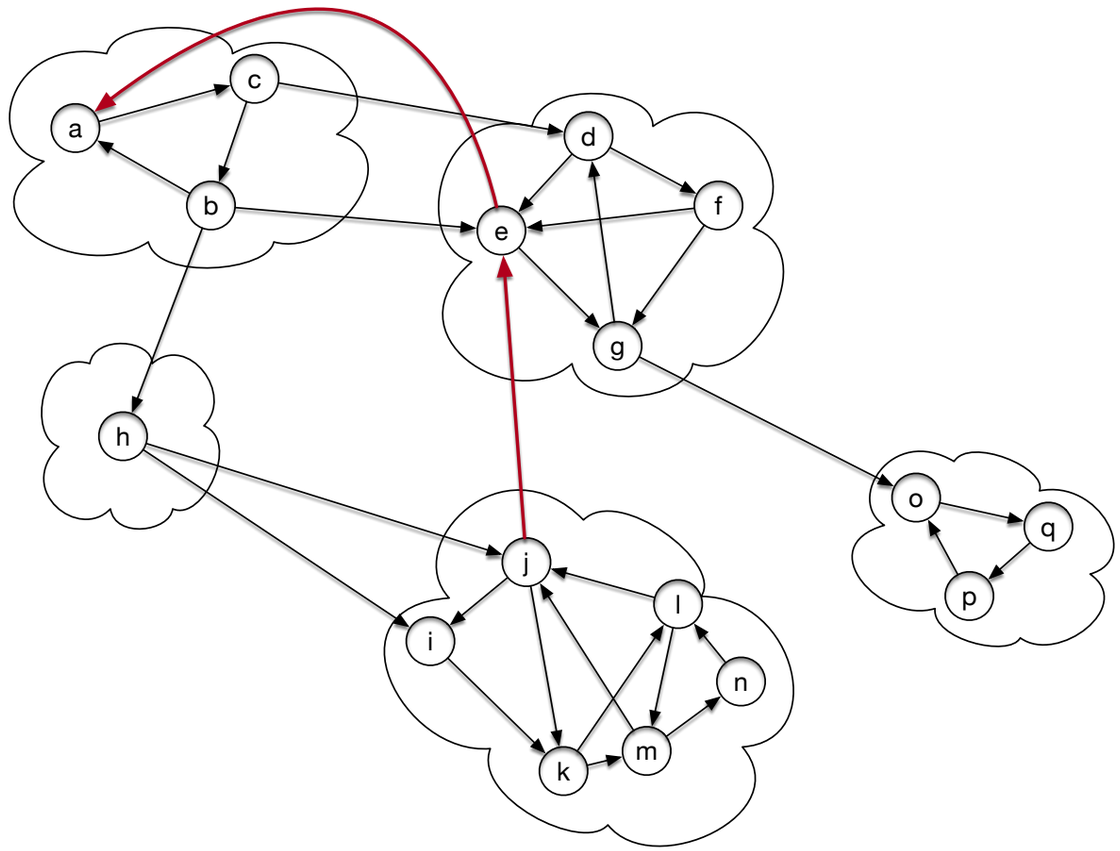


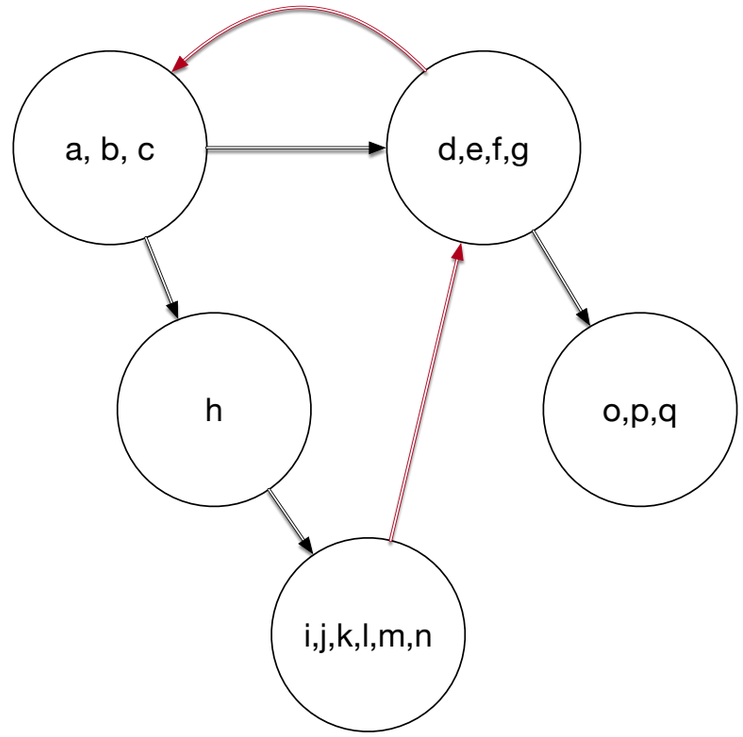
Strongly Connected Components

- The resulting metagraph has to be acyclic
  - If there is a cycle in the metagraph, then by the lemma, the metanodes can be merged into bigger strongly connected subgraphs

Strongly Connected Components

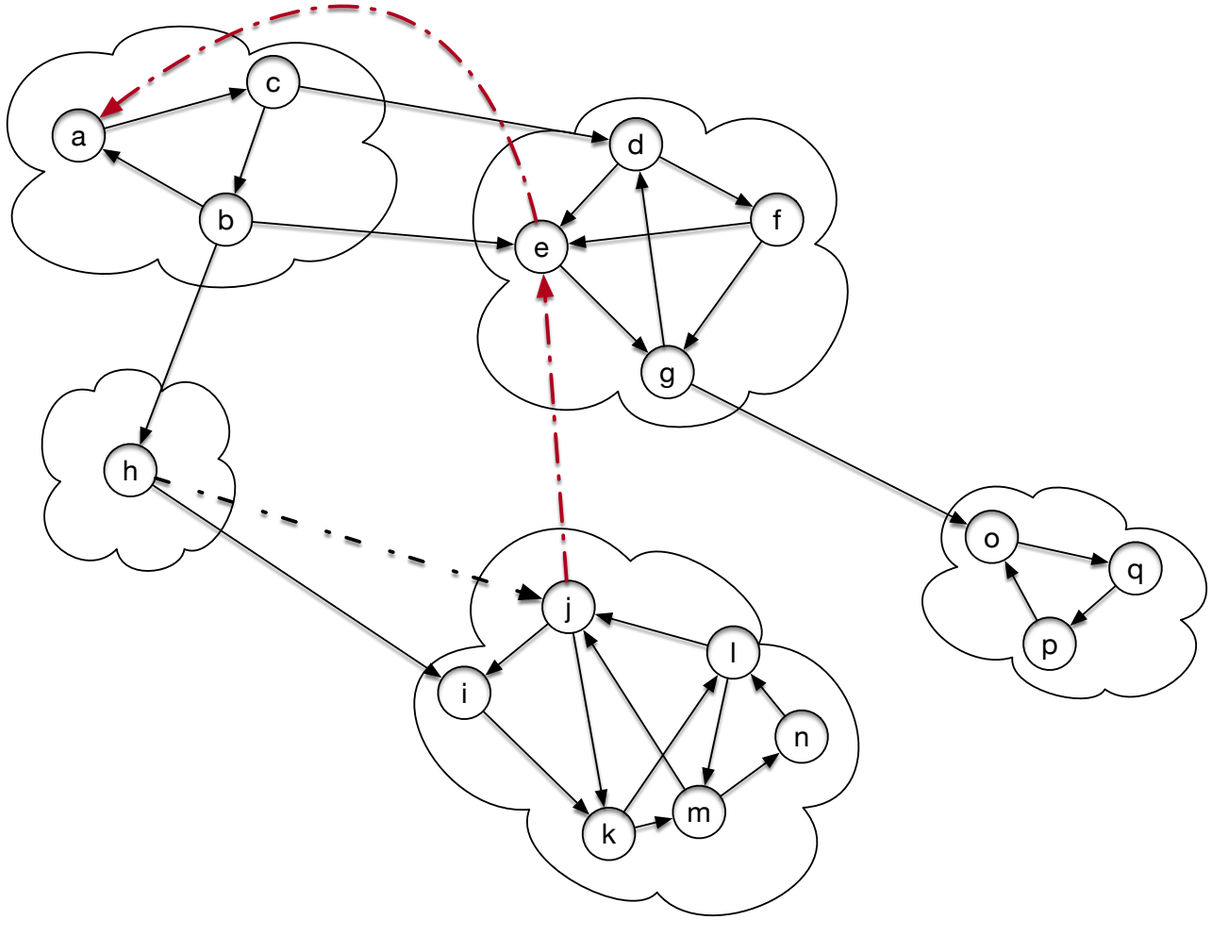
- Example: Add two edges

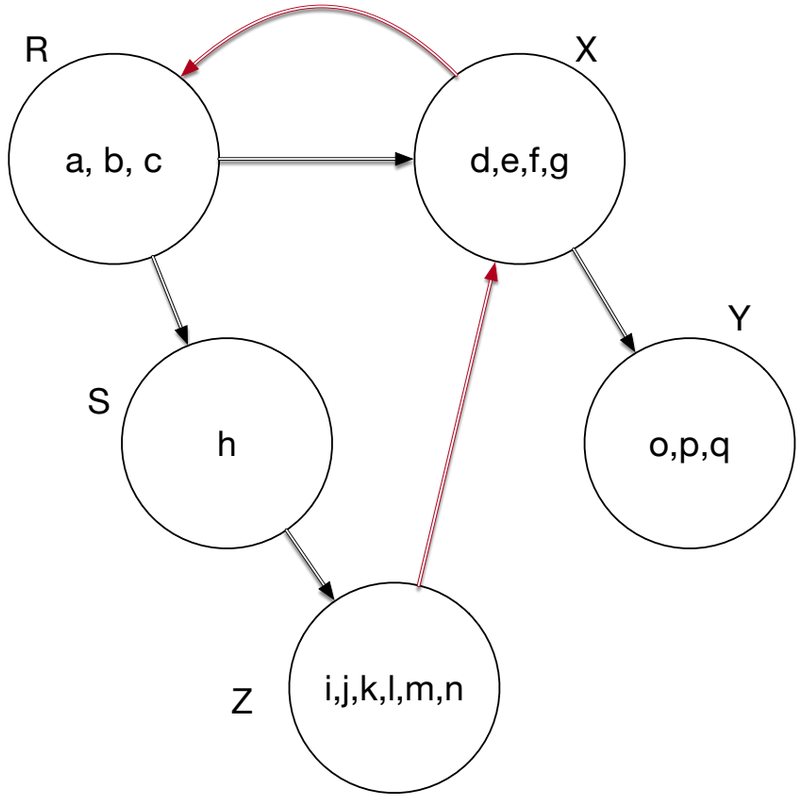




Strongly Connected Components

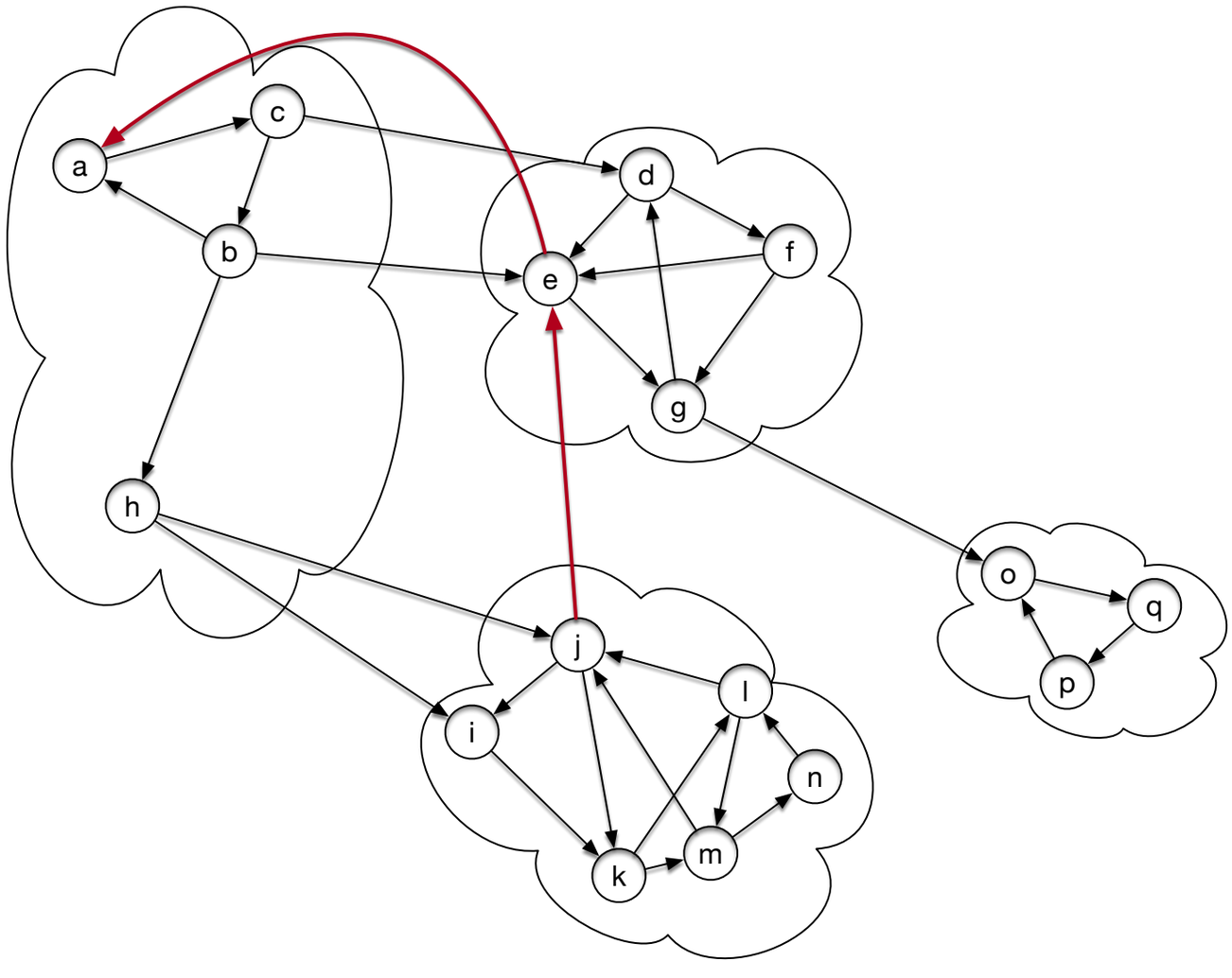
- Now we can start merging via the Lemma
- There is a path from components S to R and vice versa

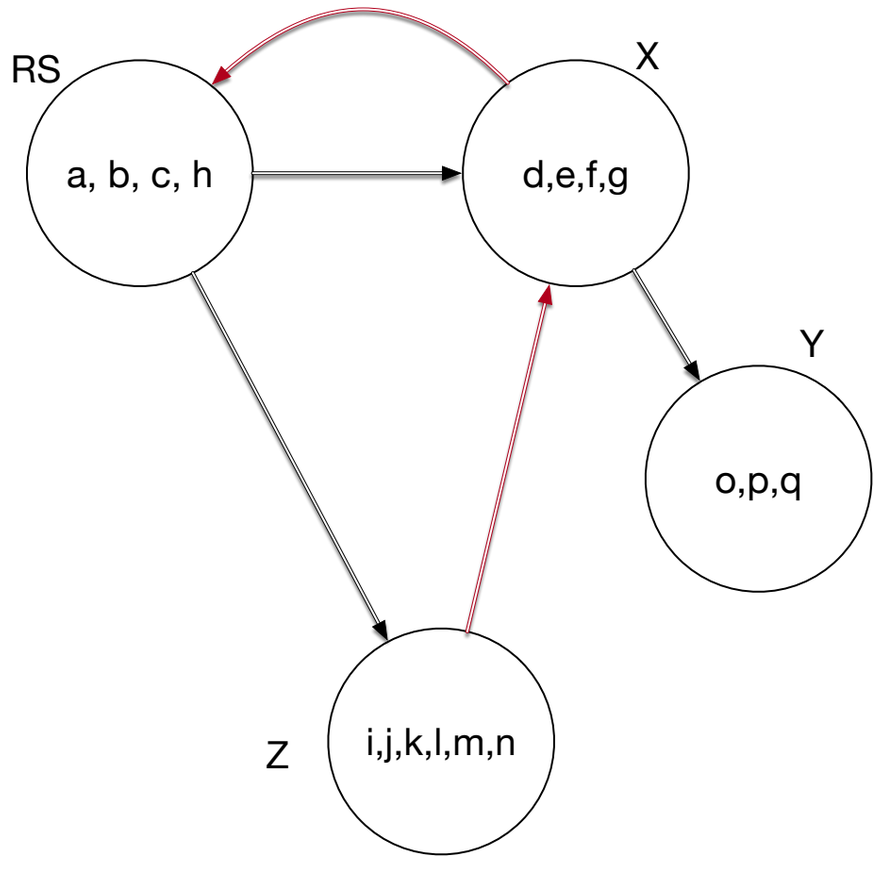




Strongly Connected Components

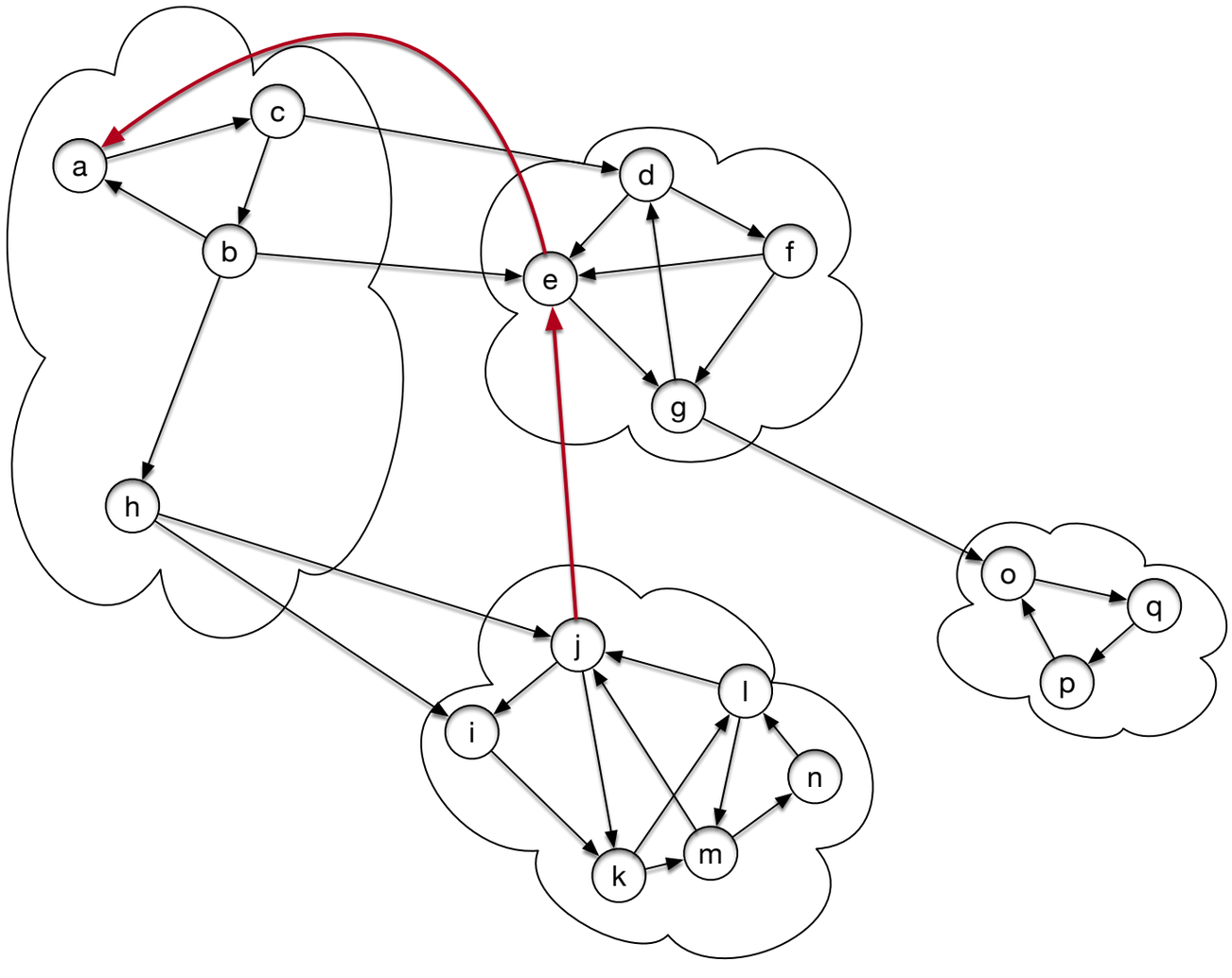
- So we merge

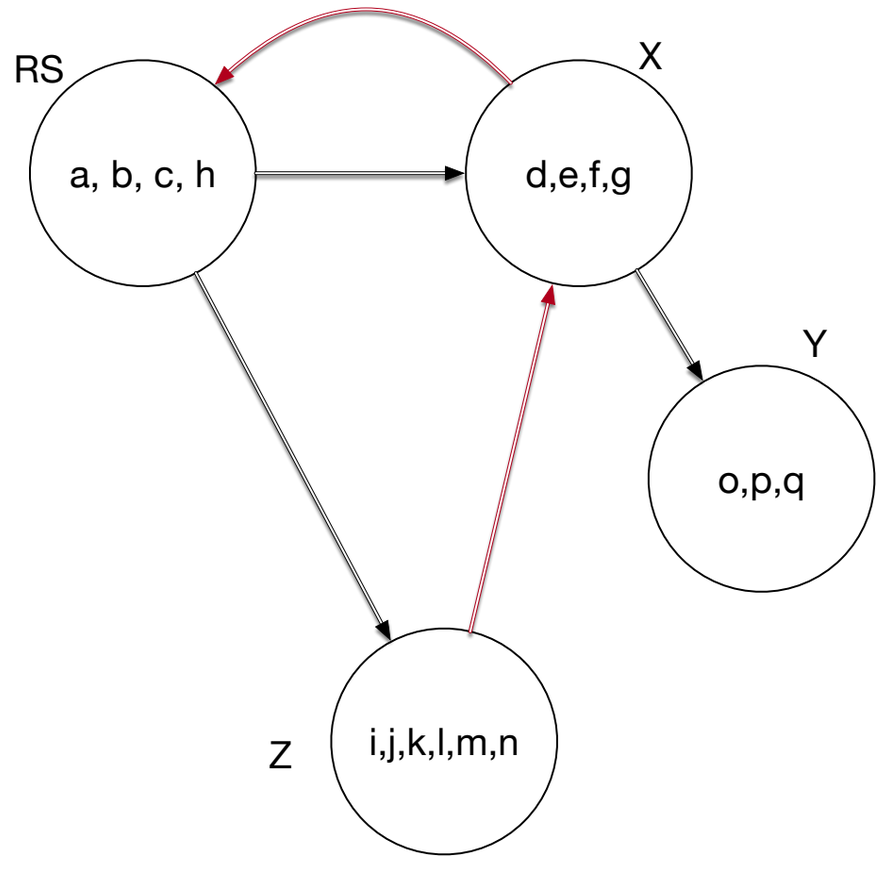




Strongly Connected Components

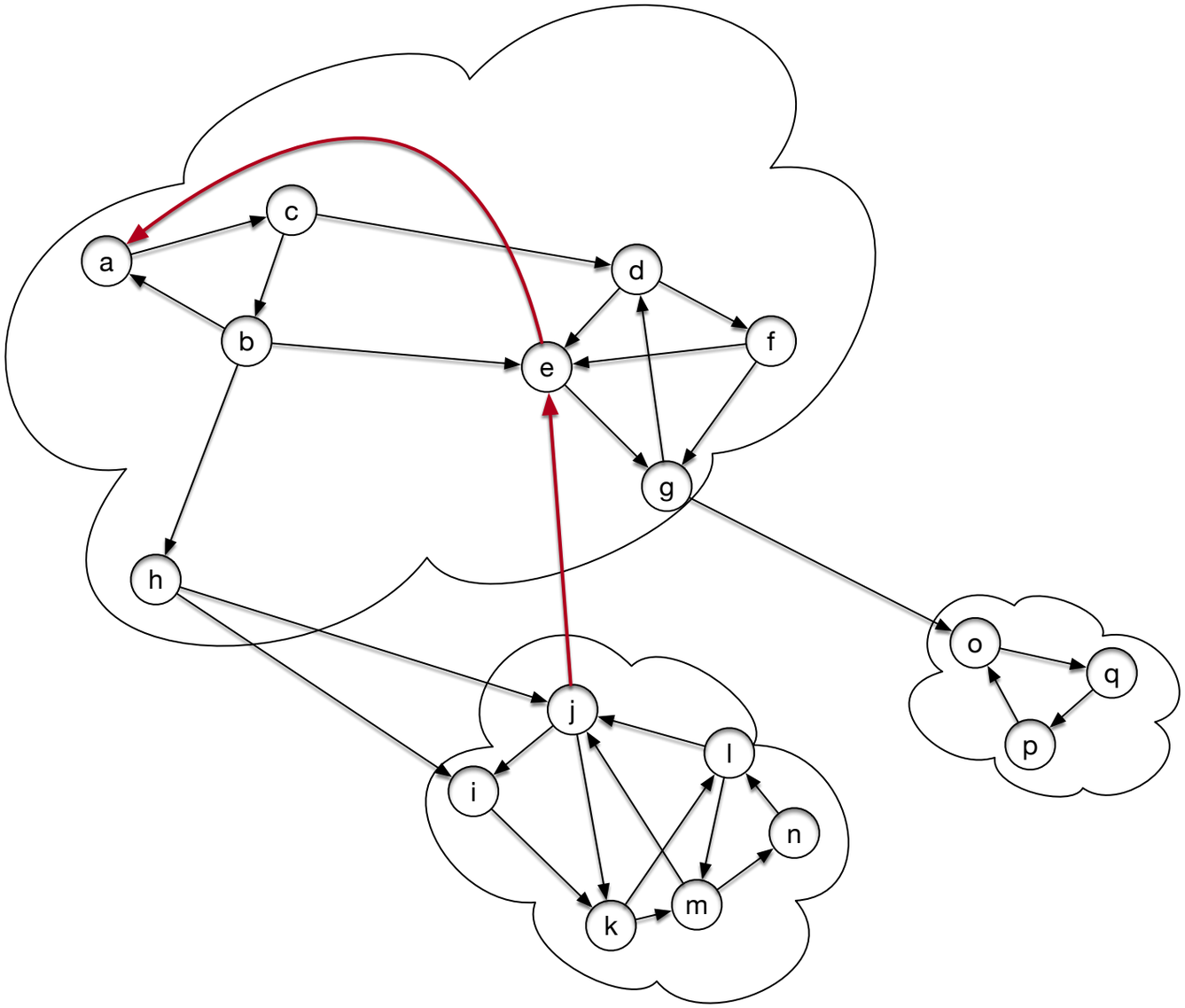
- There is a path from RS to X and vice versa:

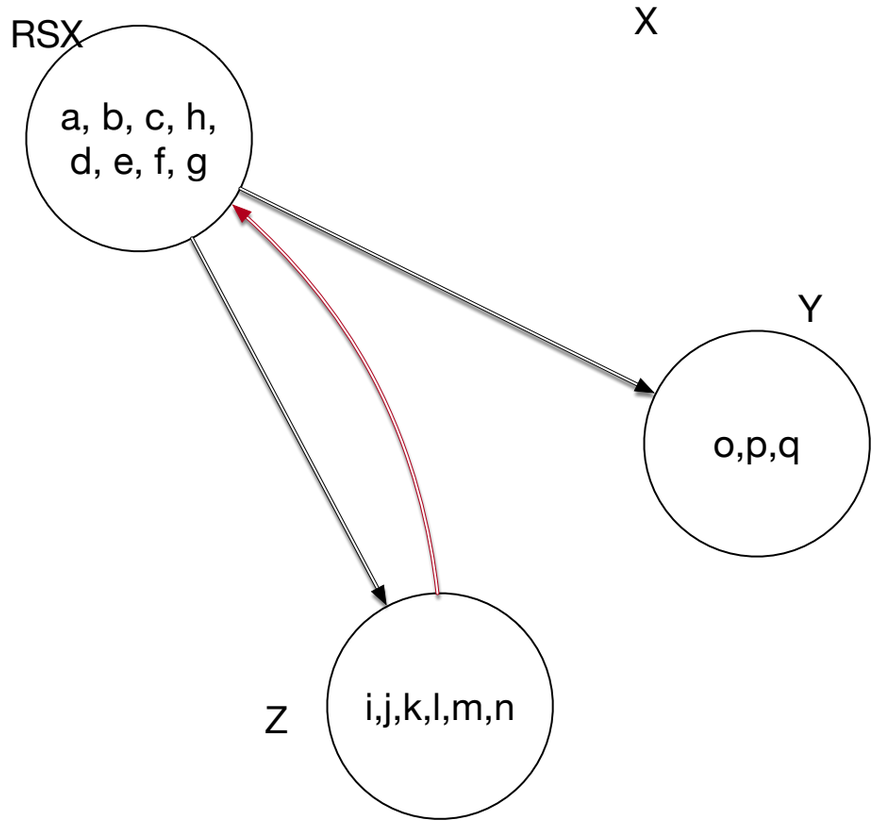




Strongly Connected Components

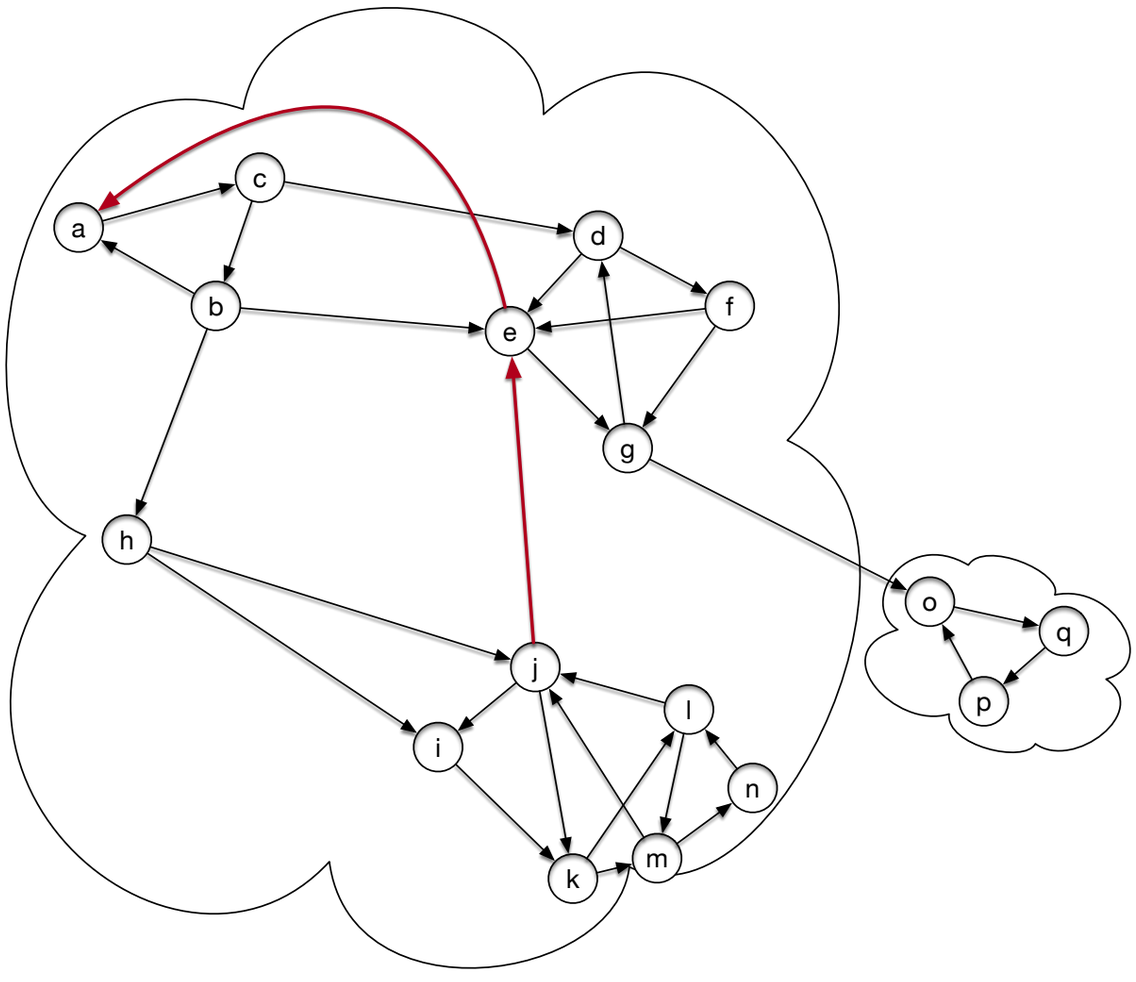
- We can merge

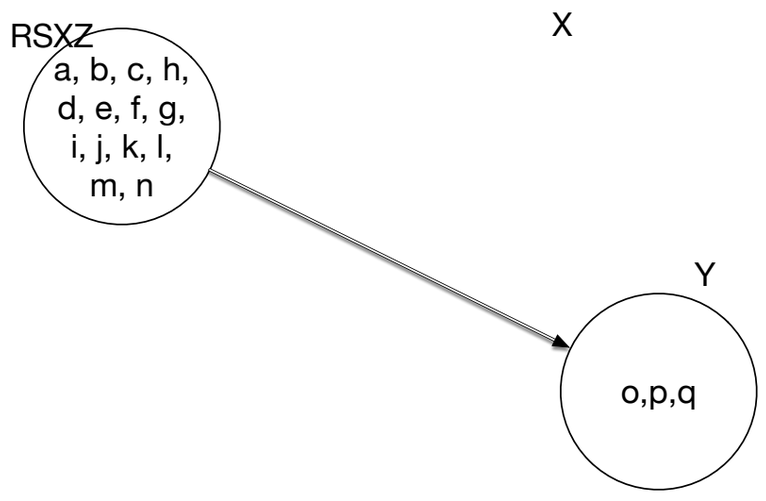




Strongly Connected Components

- Finally, we can merge Z with the new supernode





Strongly Connected Components

- This can be generalized:
  - Theorem: The metagraph is acyclic

Strongly Connected Components

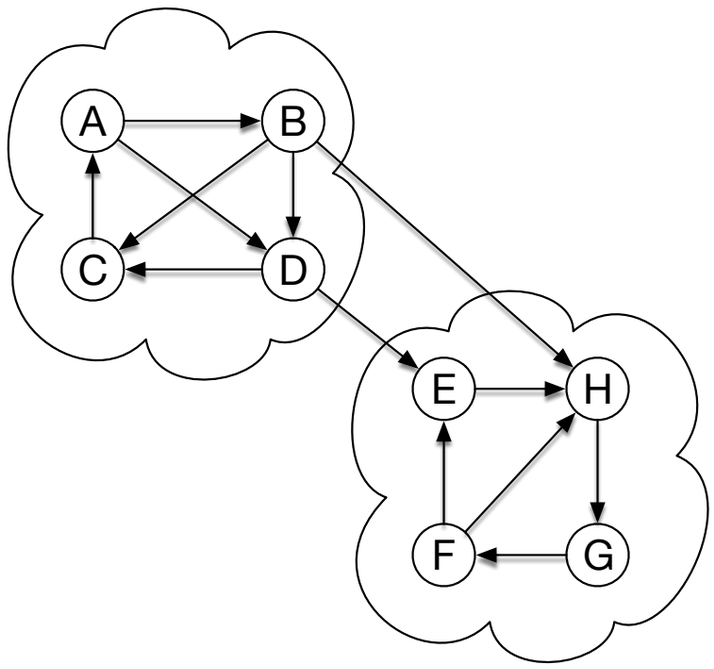
- How can we apply DFS to the problem of determining connected components?
  - The WWW graph in 2000 would have been to big for anything but linear time algorithms

Strongly Connected Components

- Answer:
  - Use DFS several times
  - Including indirectly on the metagraph

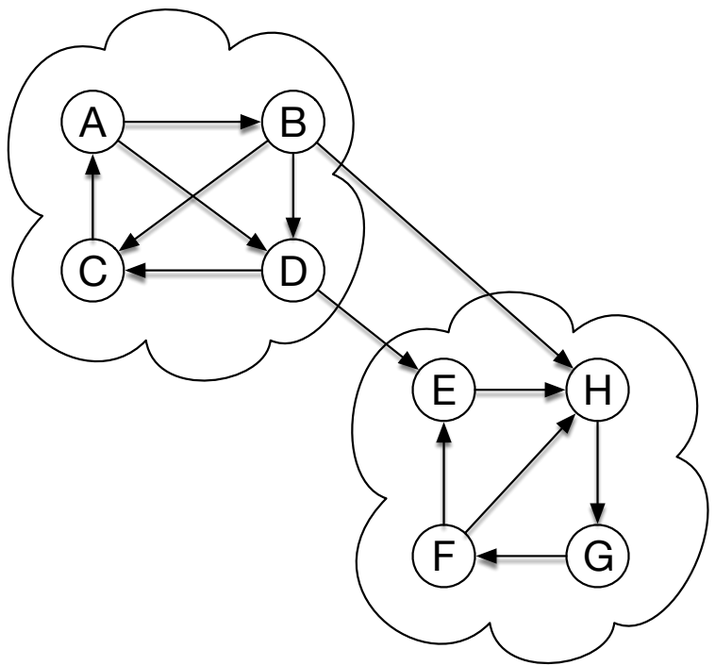
Strongly Connected Components

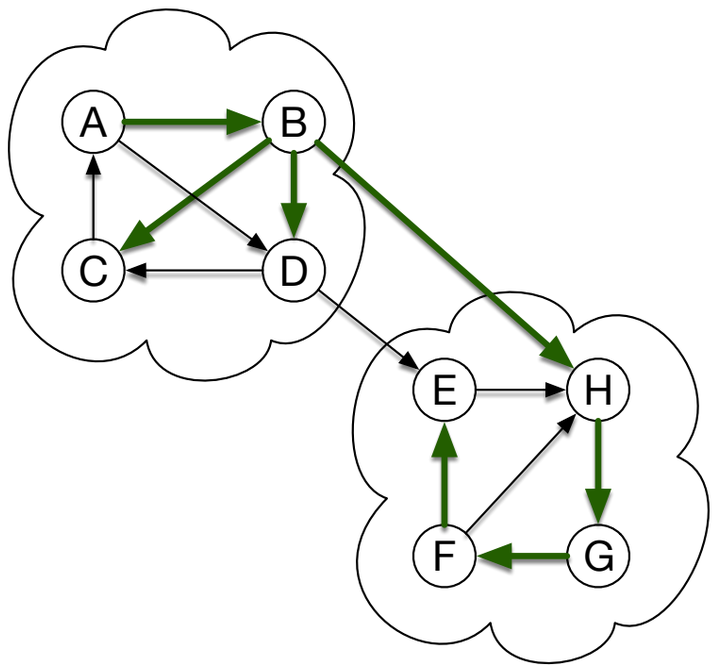
- If we start DFS at a node in a strongly connected component
  - DFS will visit **all** nodes in the strongly connected component by the white path theorem



Strongly Connected Components

- If we start DFS at A
- DFS will visit **all** nodes in the strongly connected component by the white path theorem
- But also possibly other strongly connected components



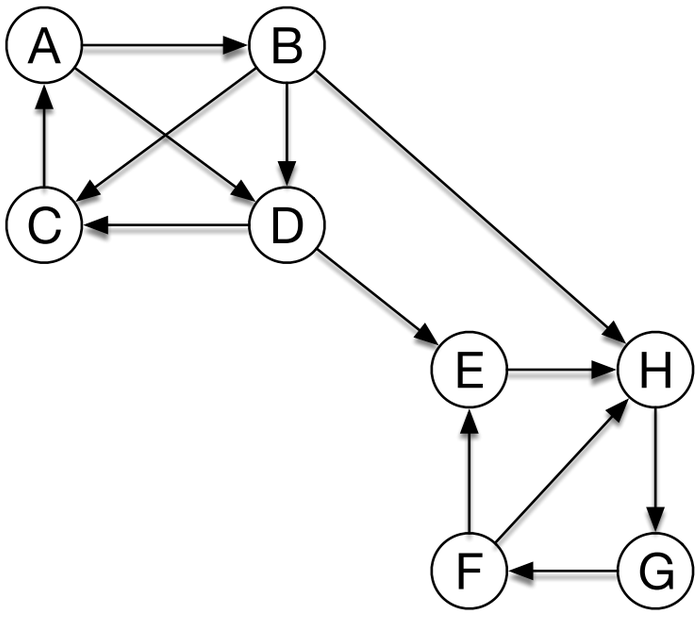


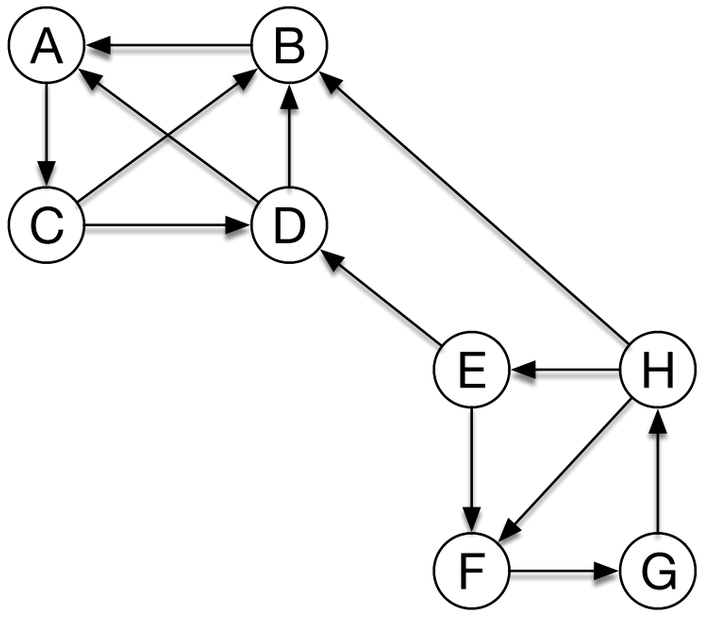
Strongly Connected Components

- Want to make sure that we start at a "sink" of the SCC metagraph
  - This we can do by running topological sort on the SCC metagraph
    - If we only had it!
  - But we can do something similar by running DFS on the graph itself
- After we make sure that we are in sinks of the SCC graph, we run DFS again to identify the strongly connected components

Strongly Connected Components

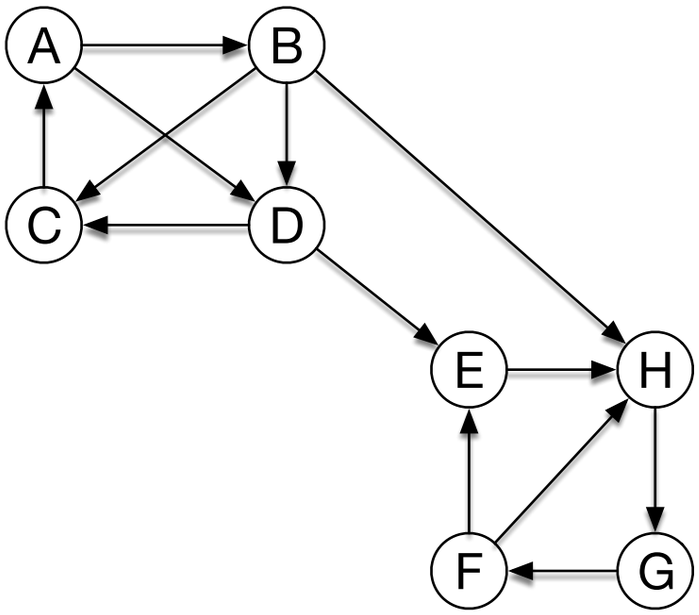
- To do this, need to run a DFS backward
  - Reverse graph:
    - Same nodes, but all edges are now reversed



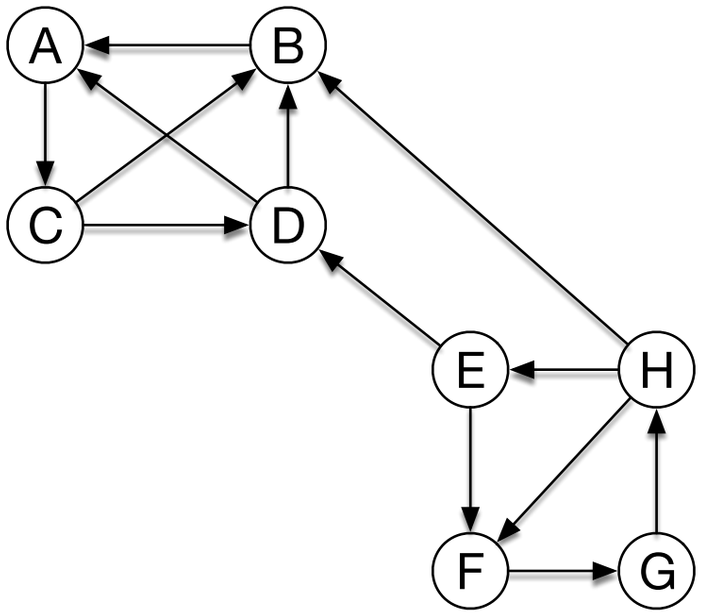


Strongly Connected Components

- How to build the reverse graph from adjacency lists?
  - If Y is in the adjacency list of X:
    - Then X is in the adjacency list of Y

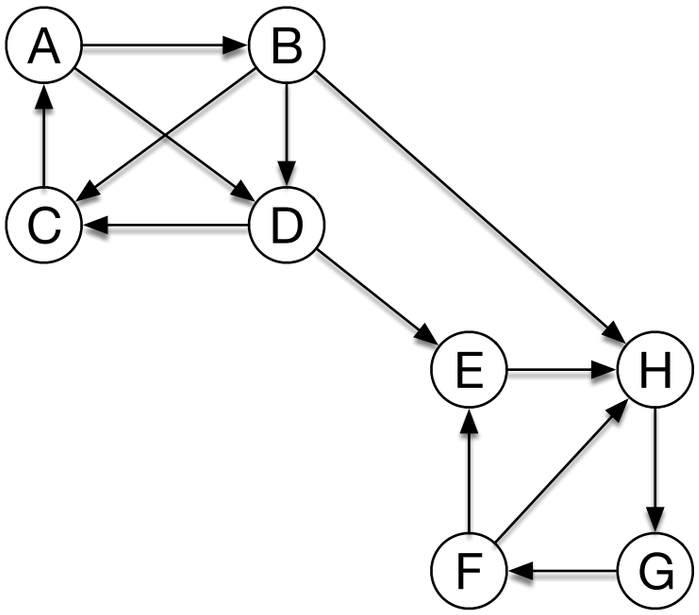



```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```

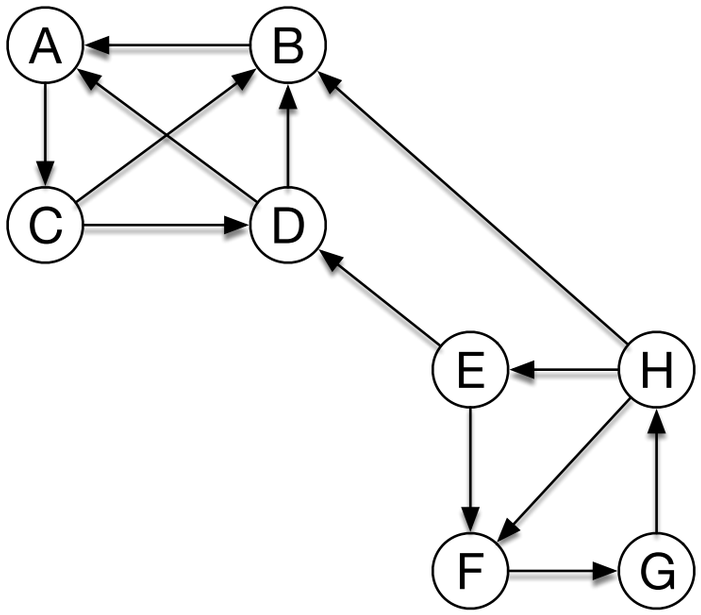


Strongly Connected Components

- Create empty adjacency list for the reverse graph



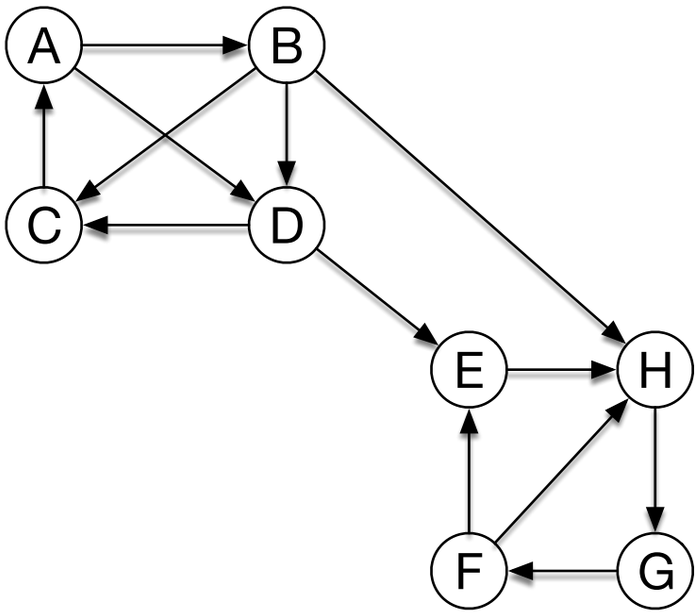

```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```



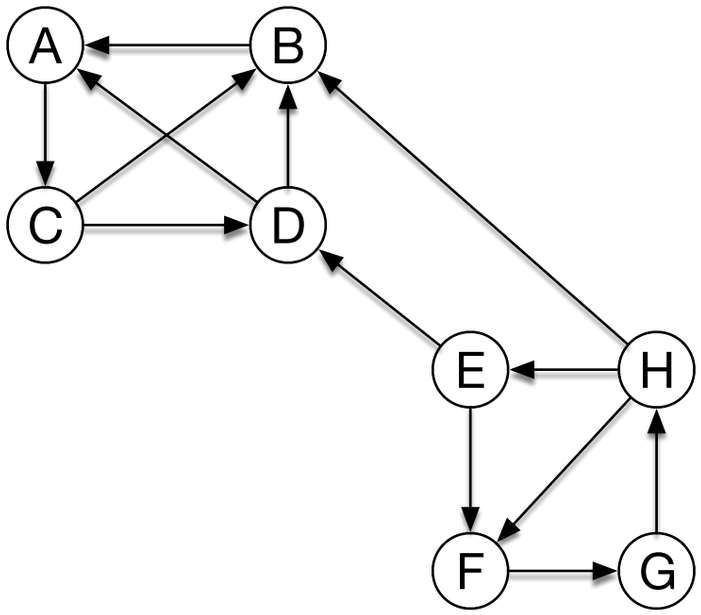

```
A:
B:
C:
D:
E:
F:
G:
H:
```

Strongly Connected Components

- Go through the adjacency list of all vertices
  - Start out with A
  - Find B
  - Add A to the list for B



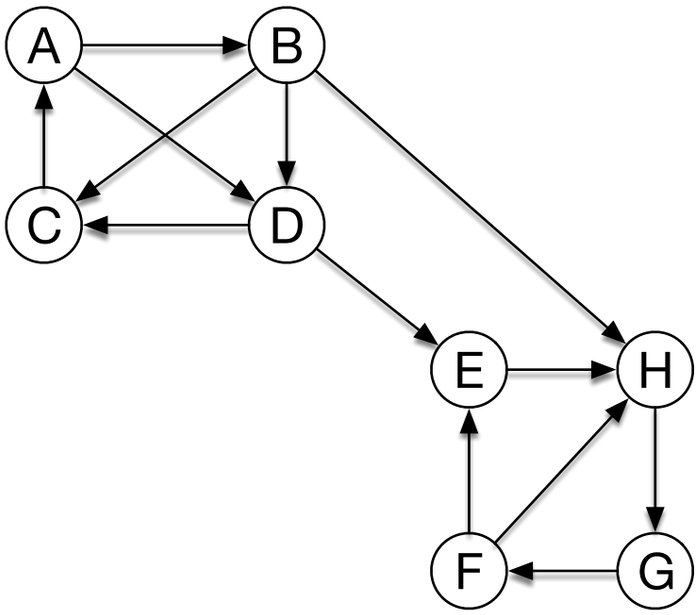

```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```



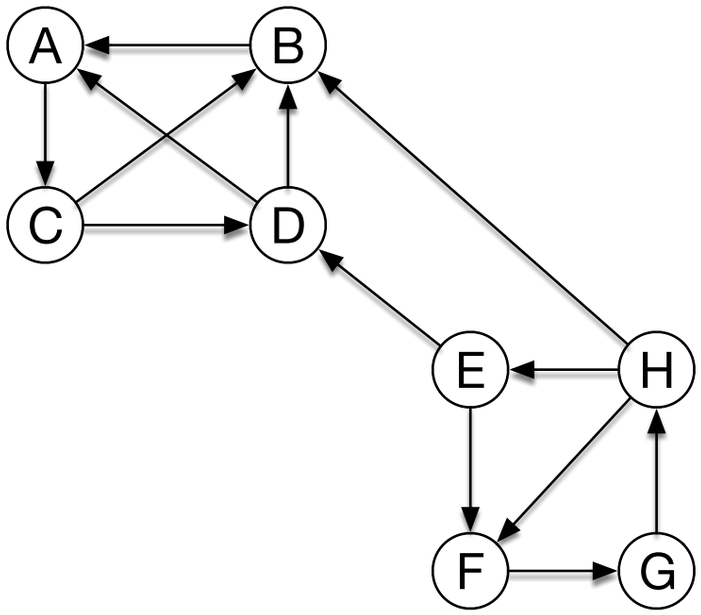

```
A:
B: A
C:
D:
E:
F:
G:
H:
```

Strongly Connected Components

- Go through the adjacency list of all vertices
  - Start out with A
  - Find D
  - Add A to the list for D



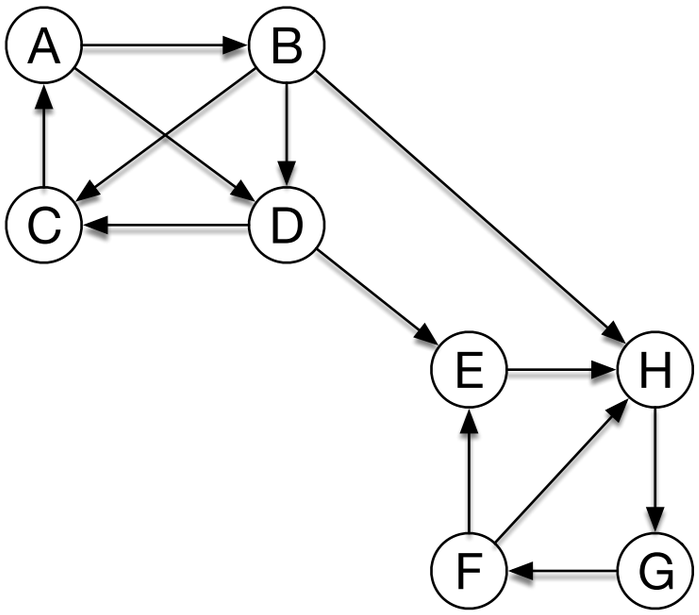

```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```



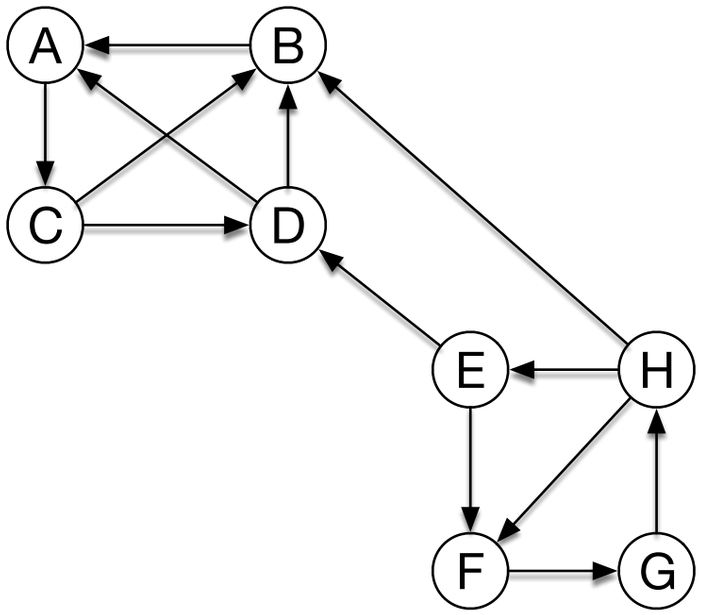

```
A:
B: A
C:
D: A
E:
F:
G:
H:
```

Strongly Connected Components

- Go through the adjacency list of all vertices
  - Continue with B
  - Find C
  - Add B to the list for C



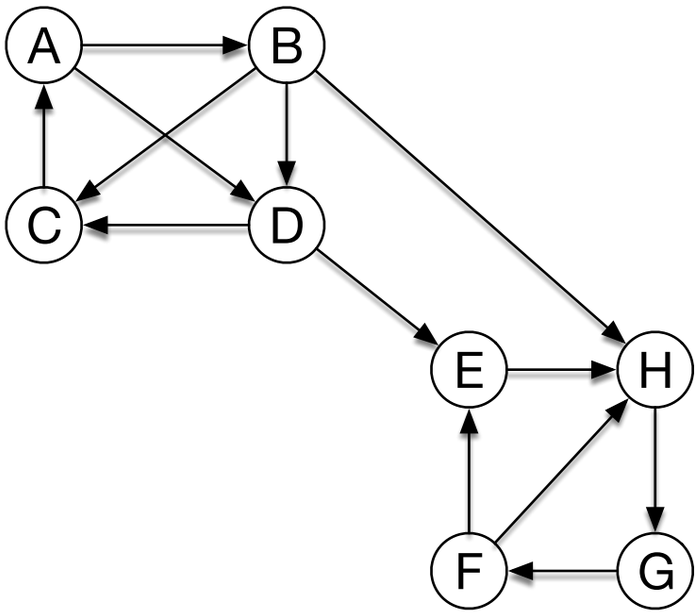

```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```



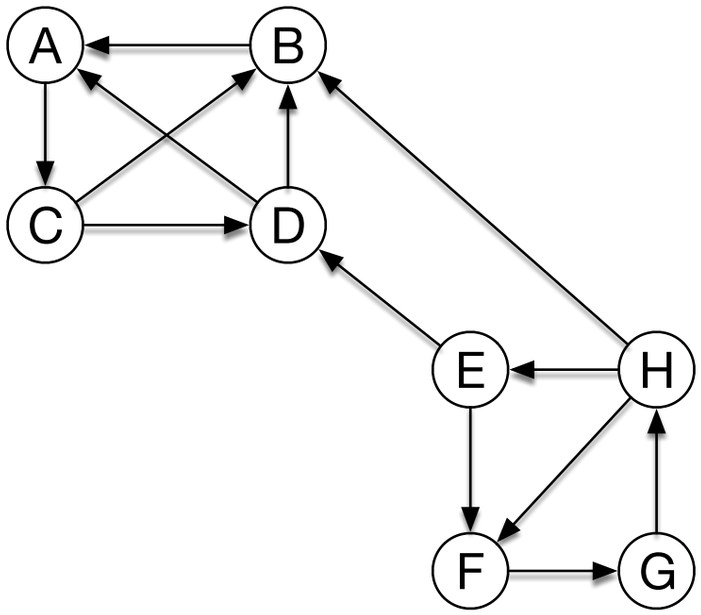

```
A:
B: A
C: B
D: A
E:
F:
G:
H:
```

Strongly Connected Components

- Go through the adjacency list of all vertices
  - Continue with B
  - Find D
  - Add B to the list for D



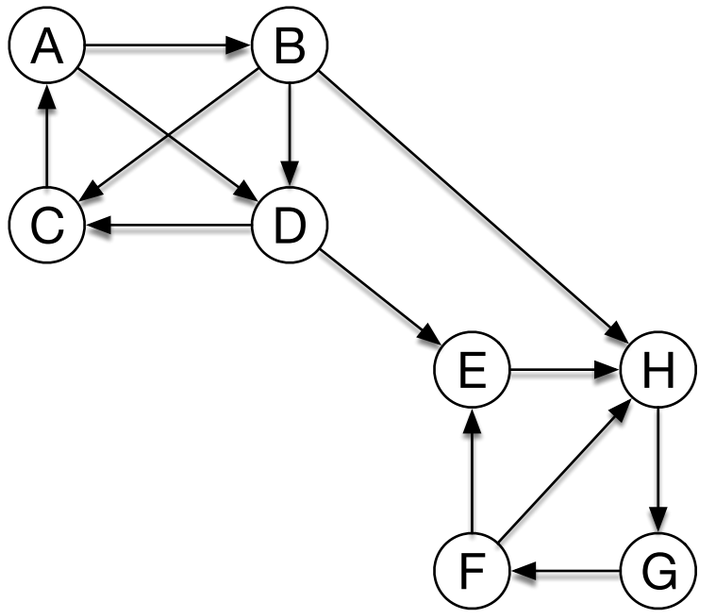

```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```



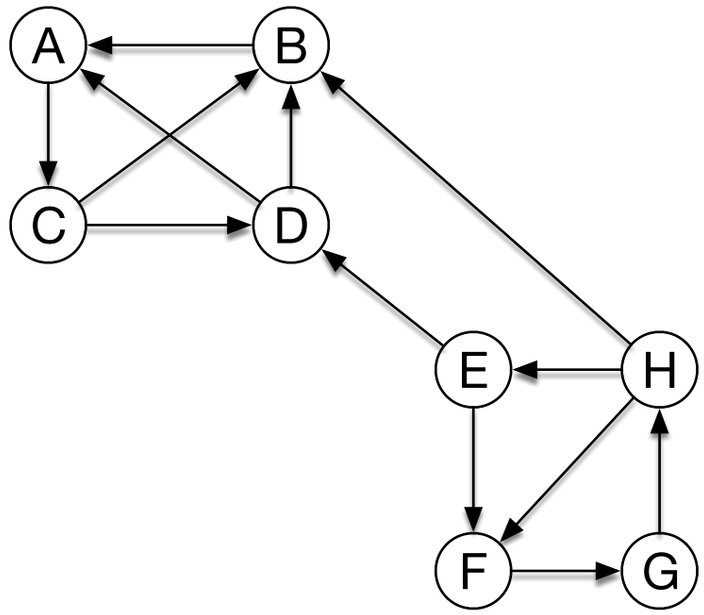

```
A:
B: A
C: B,
D: A,B
E:
F:
G:
H:
```

Strongly Connected Components

- Go through the adjacency list of all vertices
  - Continue with B
  - Find H
  - Add B to the list for H



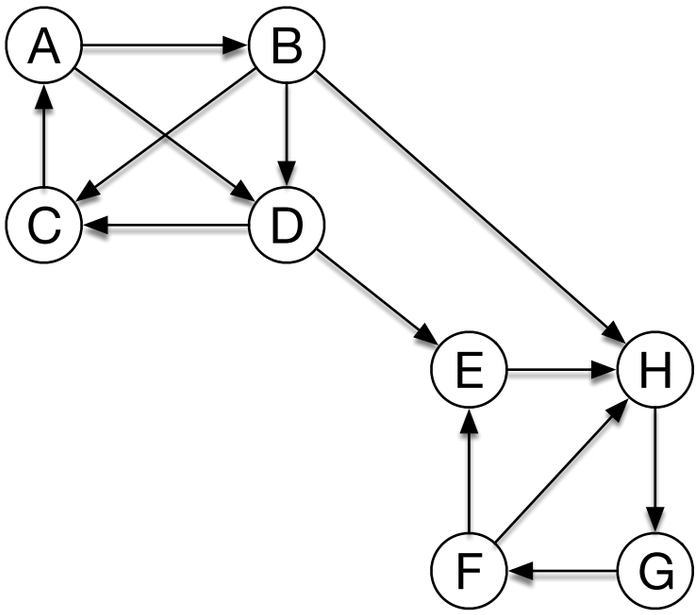

```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```



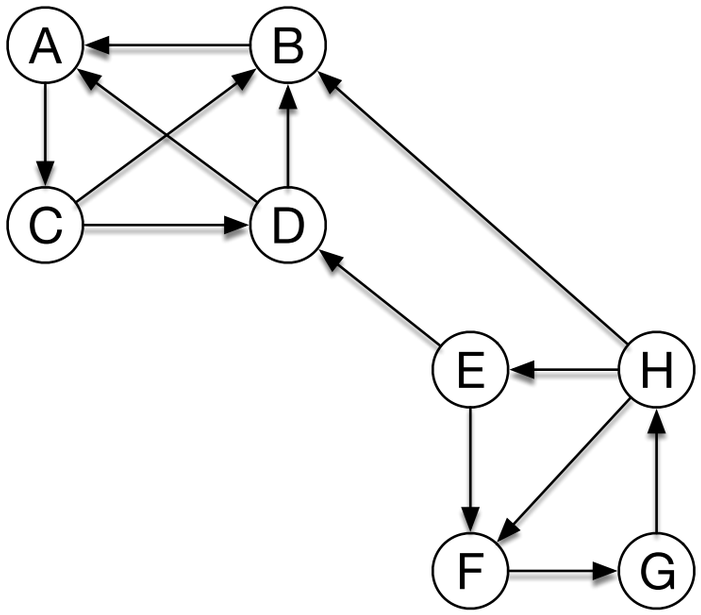

```
A:
B: A
C: B
D: A,B
E:
F:
G:
H: B
```

Strongly Connected Components

- Now continue




```
A: B,D
B: C,D,H
C: A
D: C,E
E: H
F: E,H
G: F
H: G
```



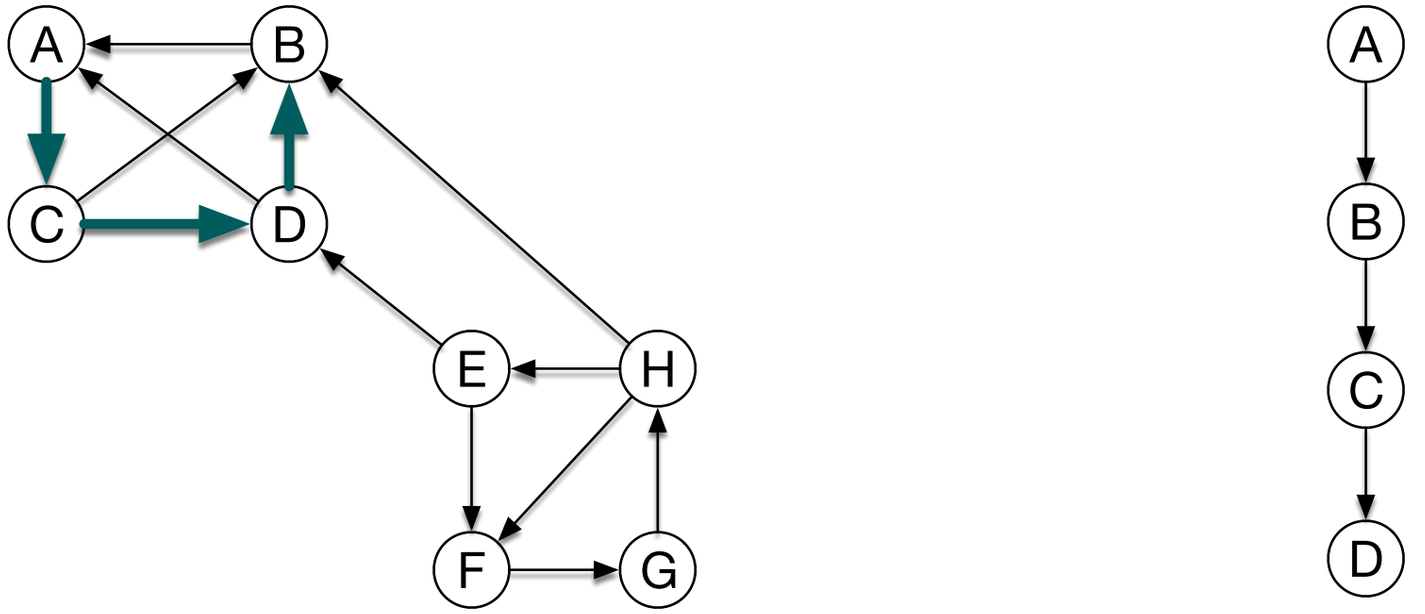

```
A: C
B: A
C: B,D
D: A,B
E: D
F: G
G: H
H: B,E
```

Strongly Connected Components

- Calculating the reverse graph costs
  - creating empty adjacency lists for the reverse graph
    - $\Theta(|V|)$
  - going through all elements in the adjacency list and add an element to one adjacency list for the reverse graph
    - $\Theta(|E|)$
- Total: $\Theta(|V| + |E|)$, i.e. linear in the size of the graph

Strongly Connected Components

- If we start out with A in the reverse graph, then DFS yields



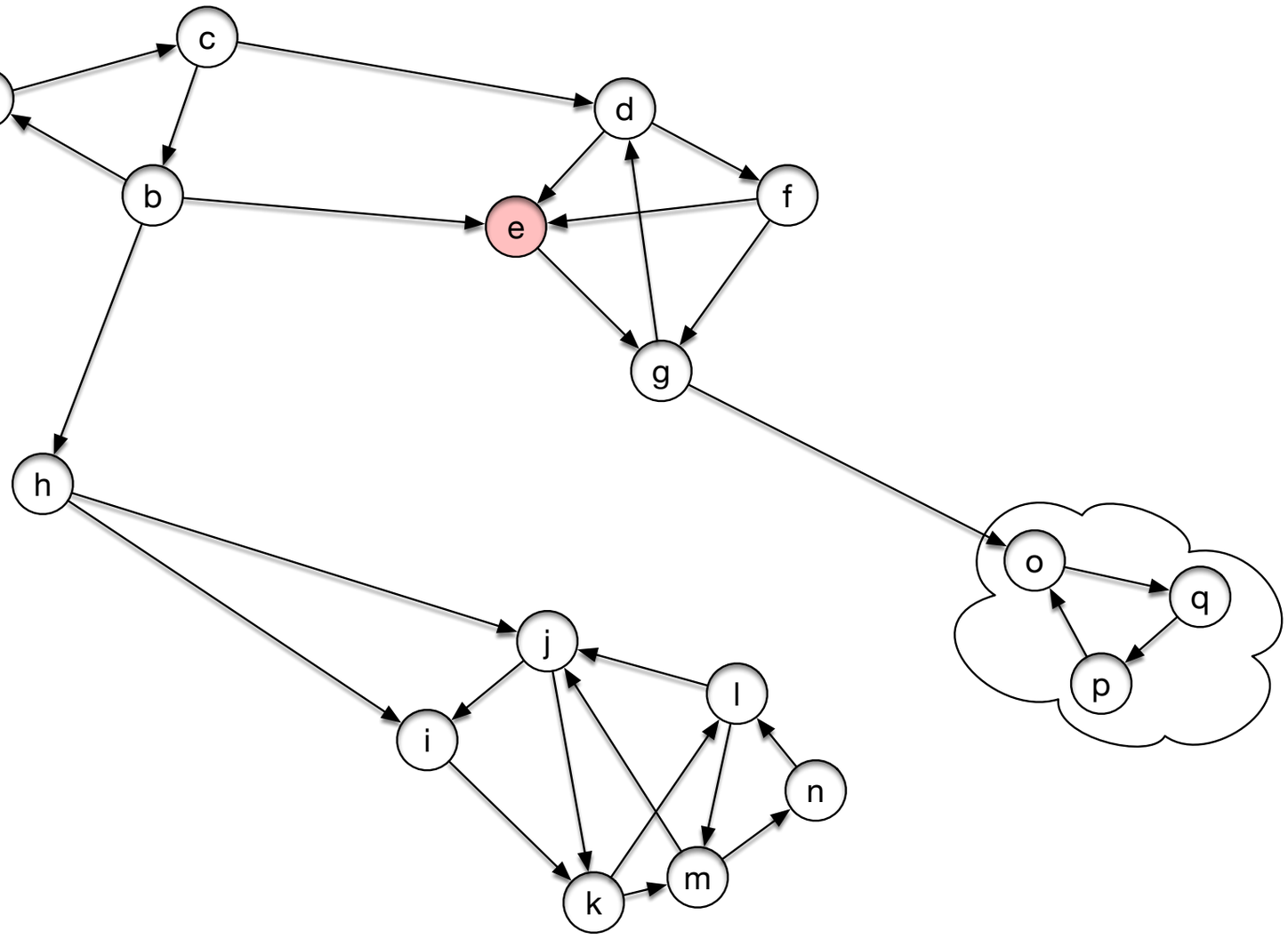
- The DFS forest has one component, with nodes in the strongly connected component of A
- Same thing if we start out with any **other** node in the strongly connected component

Strongly Connected Components

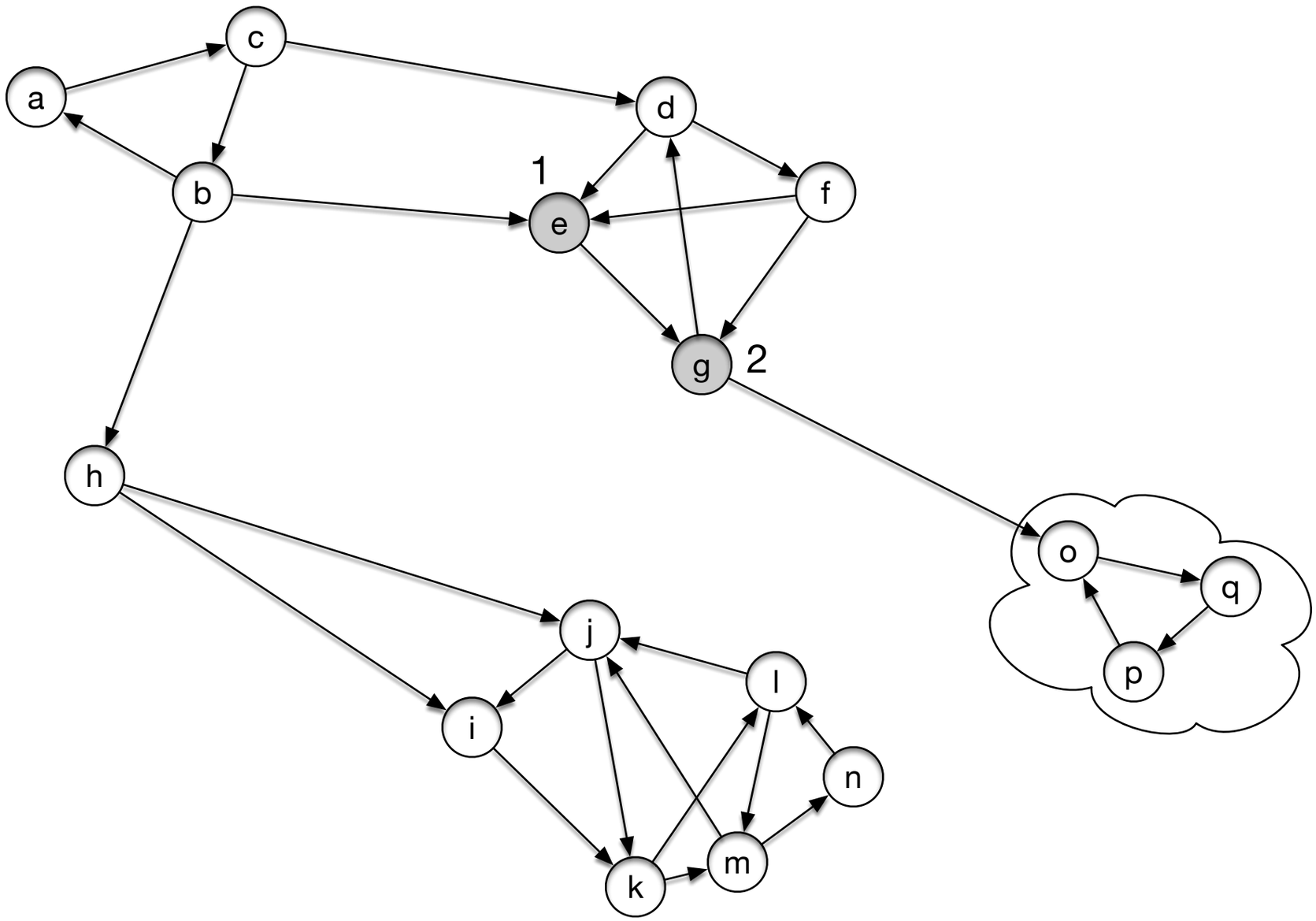
- Now we have the basic idea of Kosaraju's algorithm
  - Run DFS on the graph
  - Run DFS again, but on the reversed graph, select the nodes to visit in order of the finishing times of the first run
  - The trees in the DFS forest of the second run are the strongly connected components of the graph

Strongly Connected Components

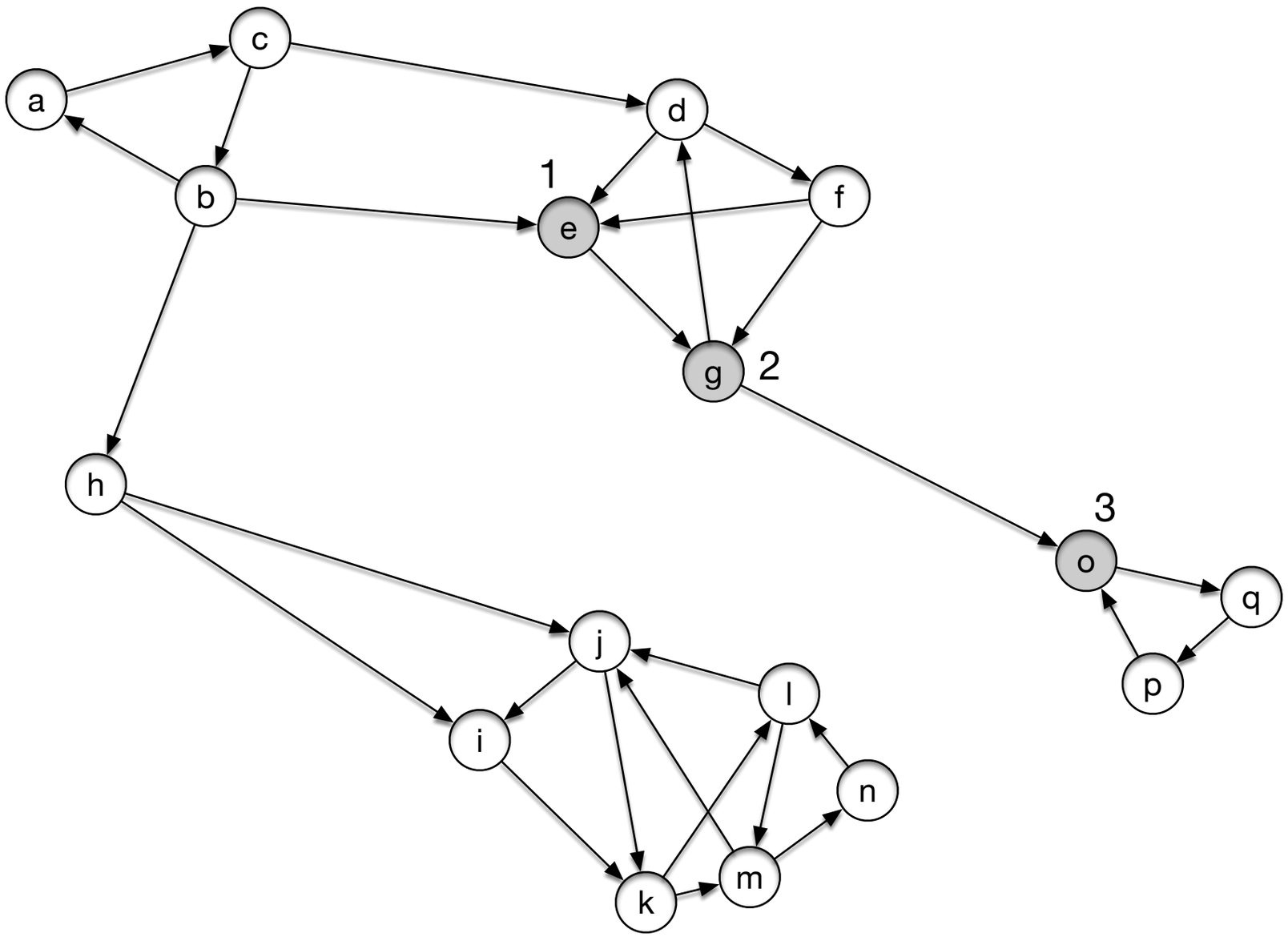
- Example: Start in E



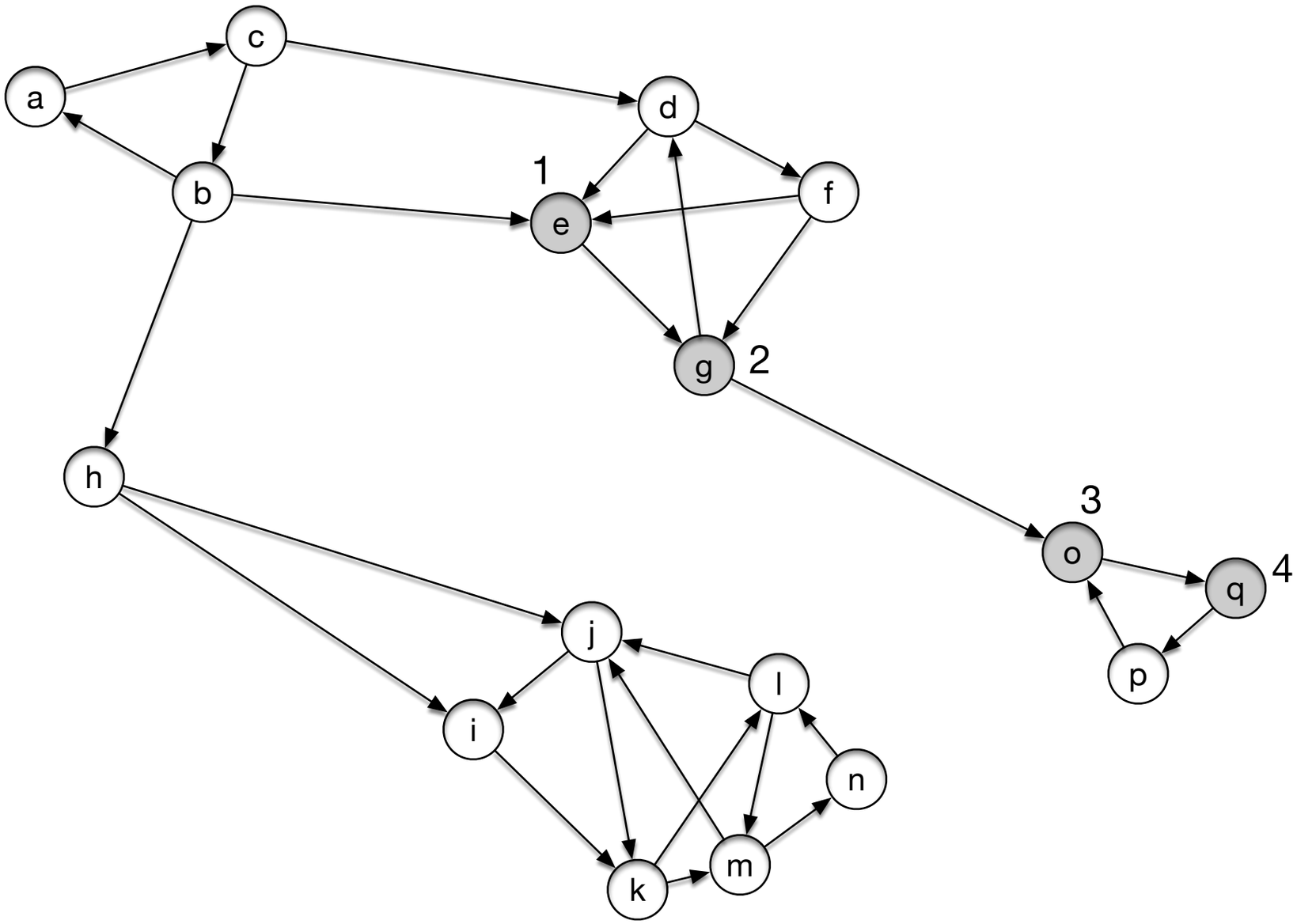
Strongly Connected Components



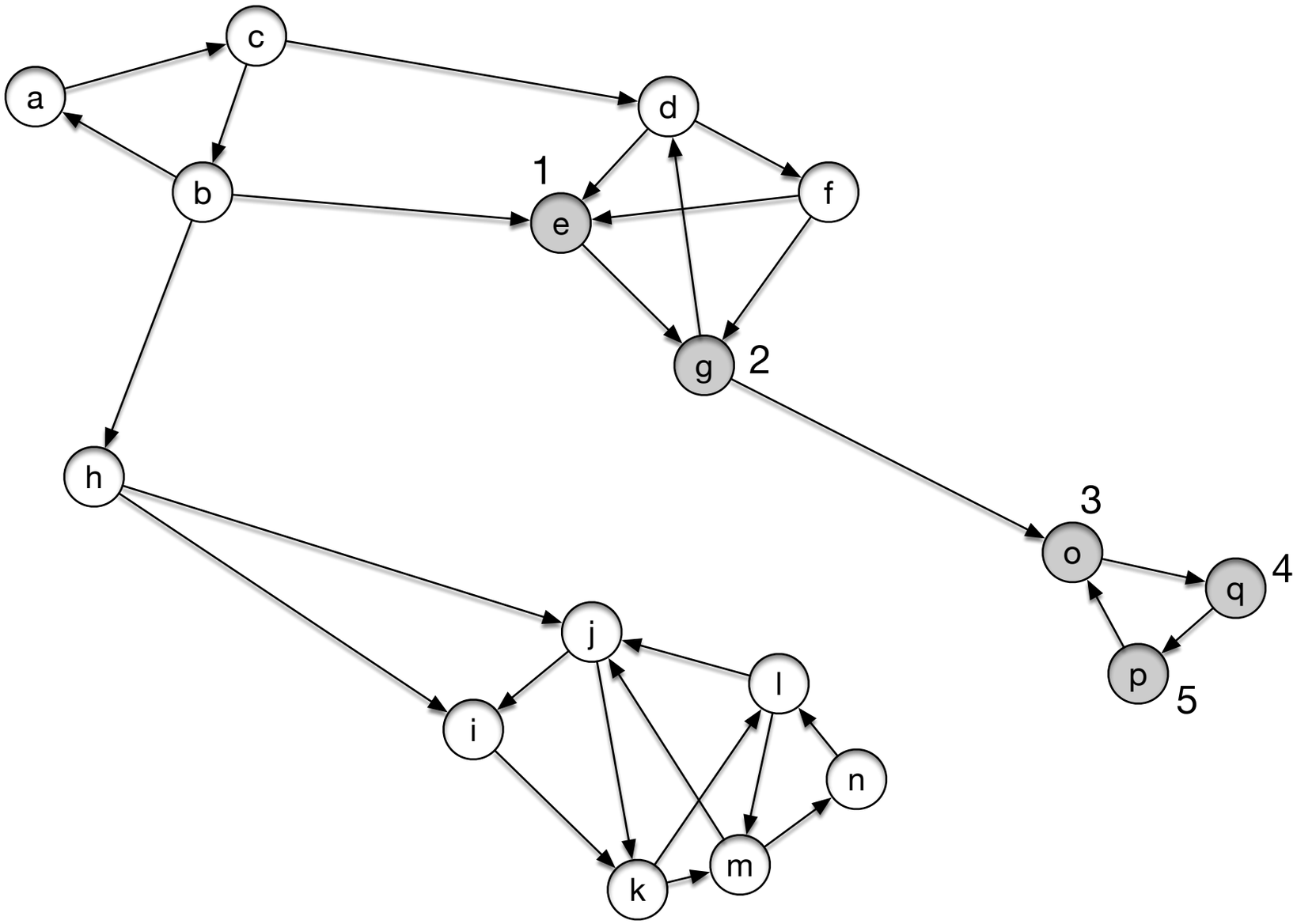
Strongly Connected Components



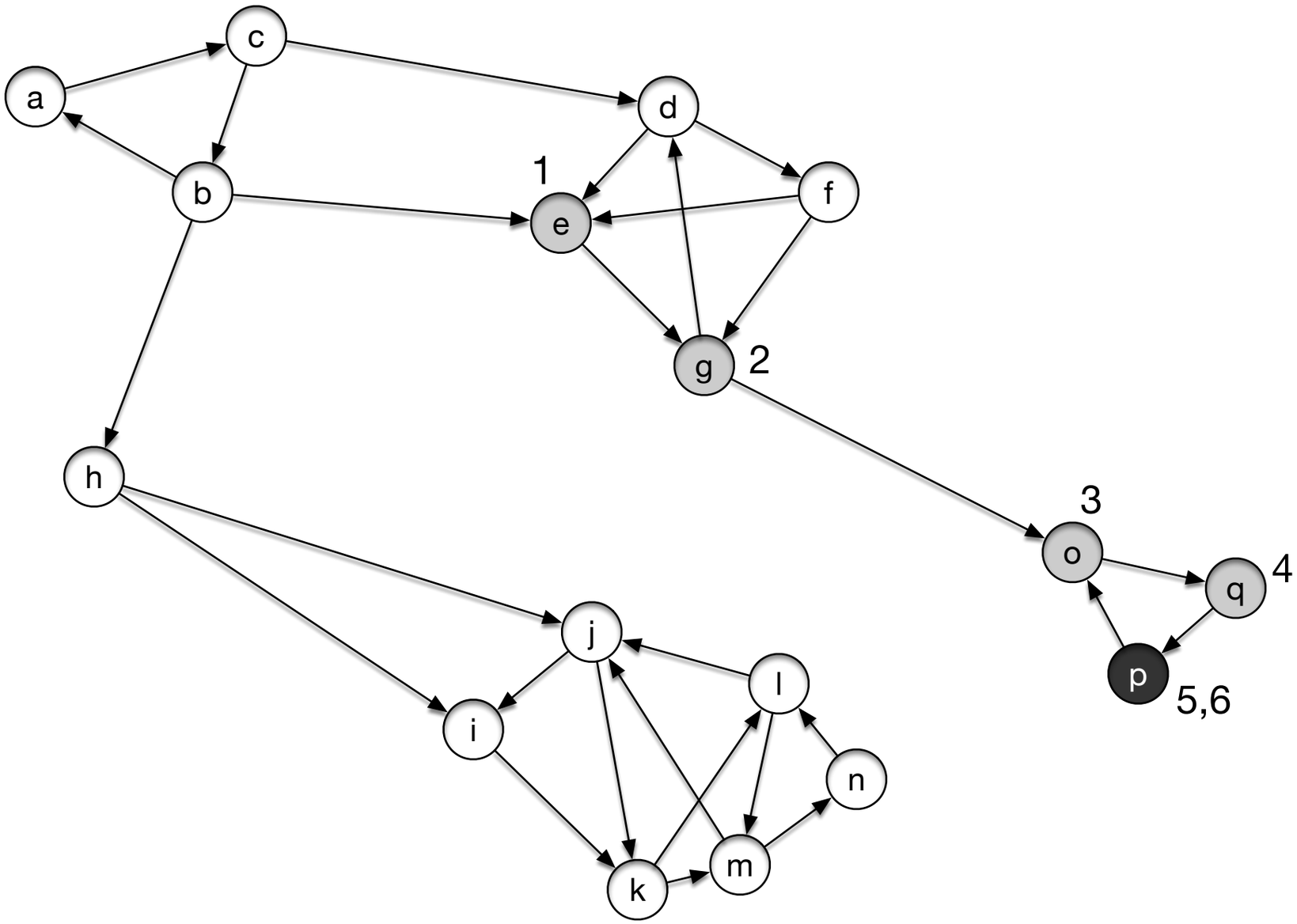
Strongly Connected Components



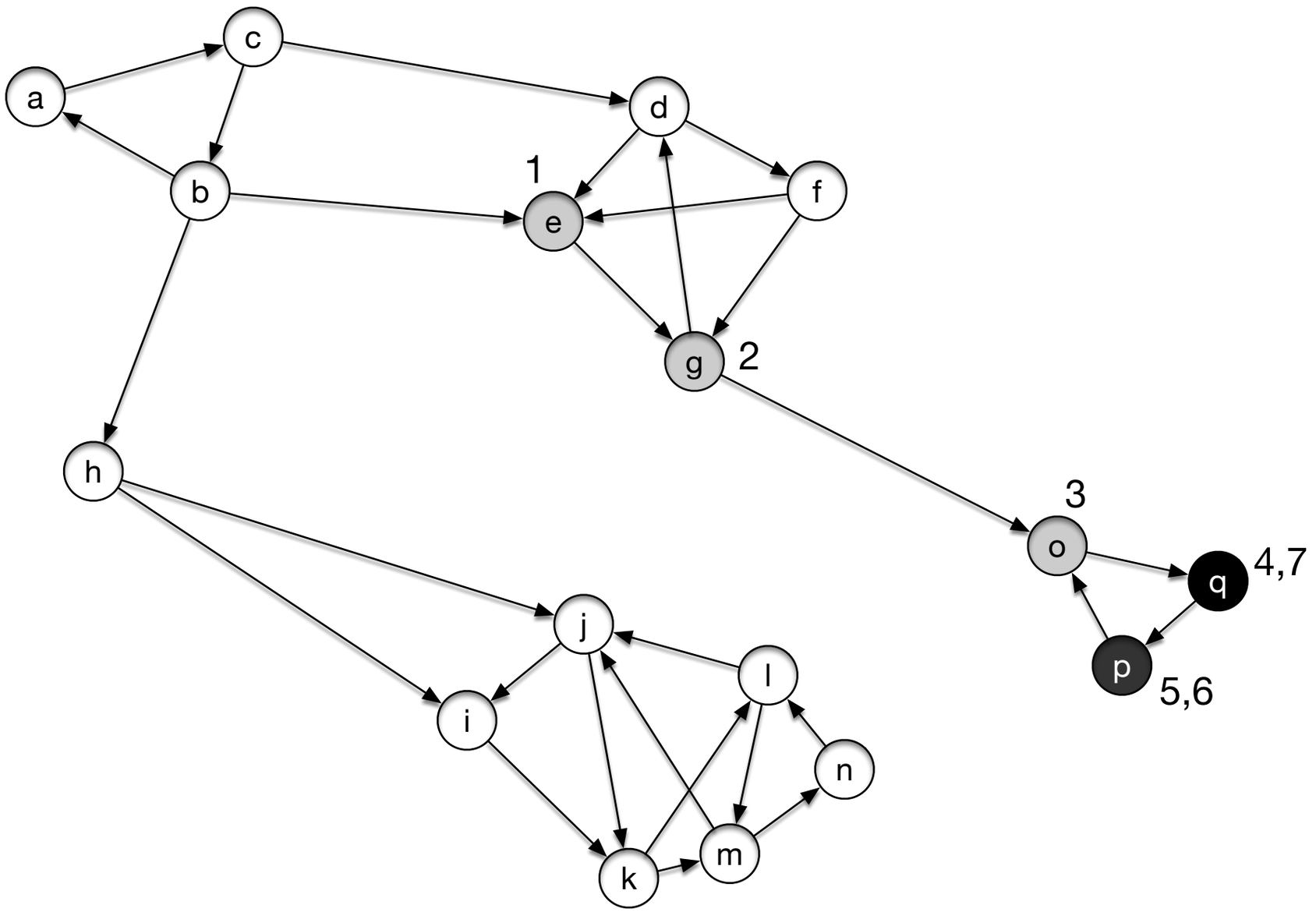
Strongly Connected Components



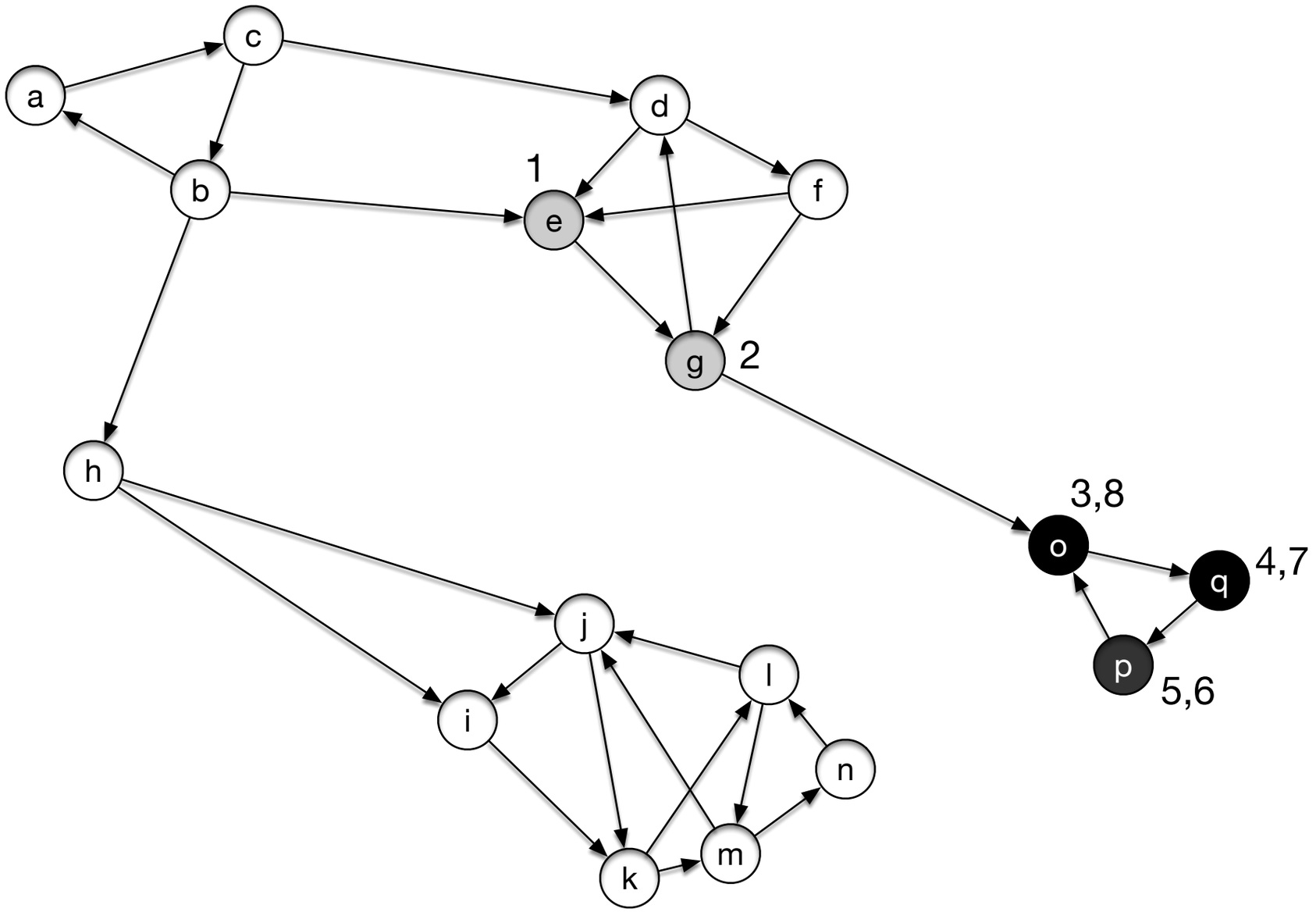
Strongly Connected Components



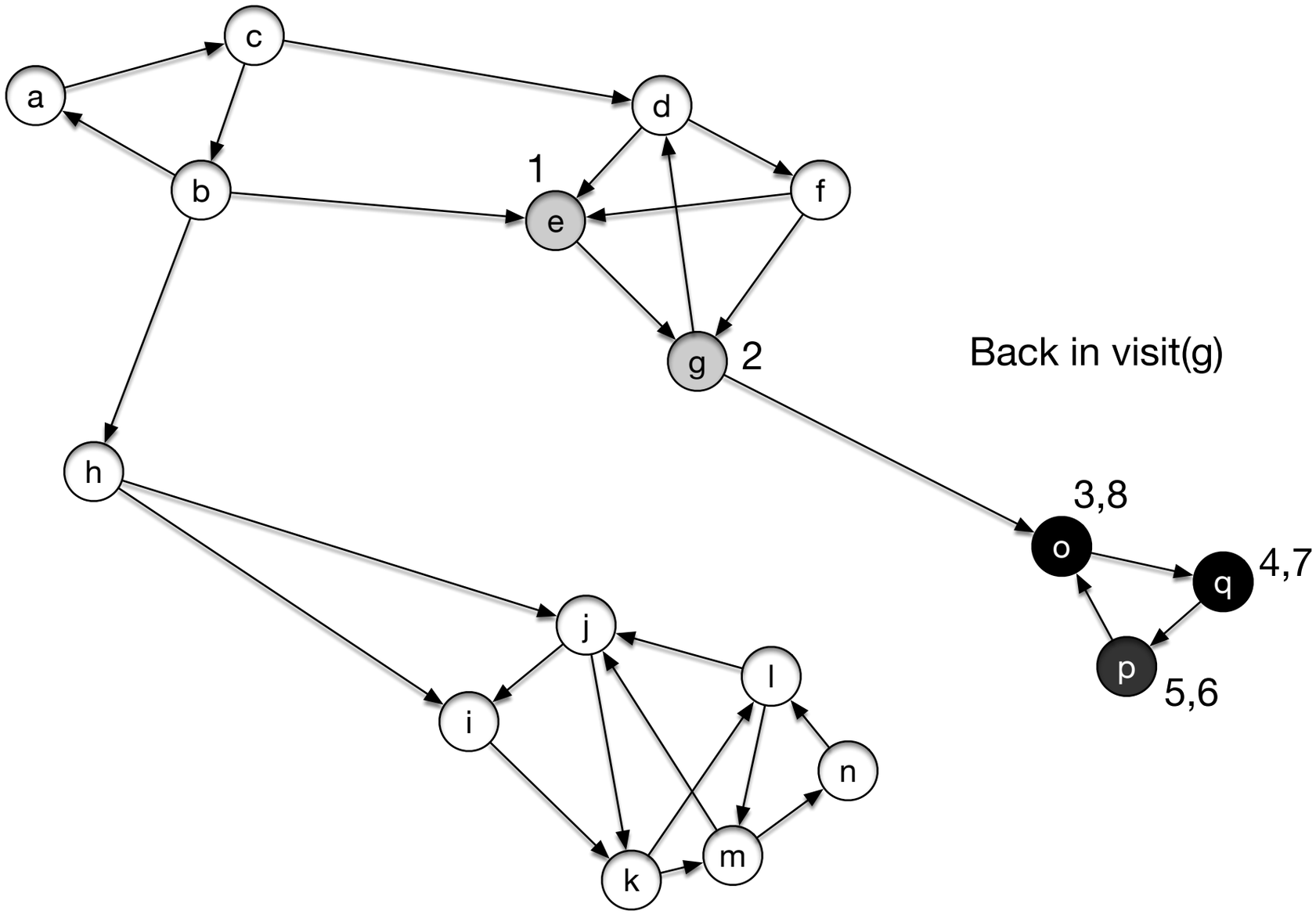
Strongly Connected Components



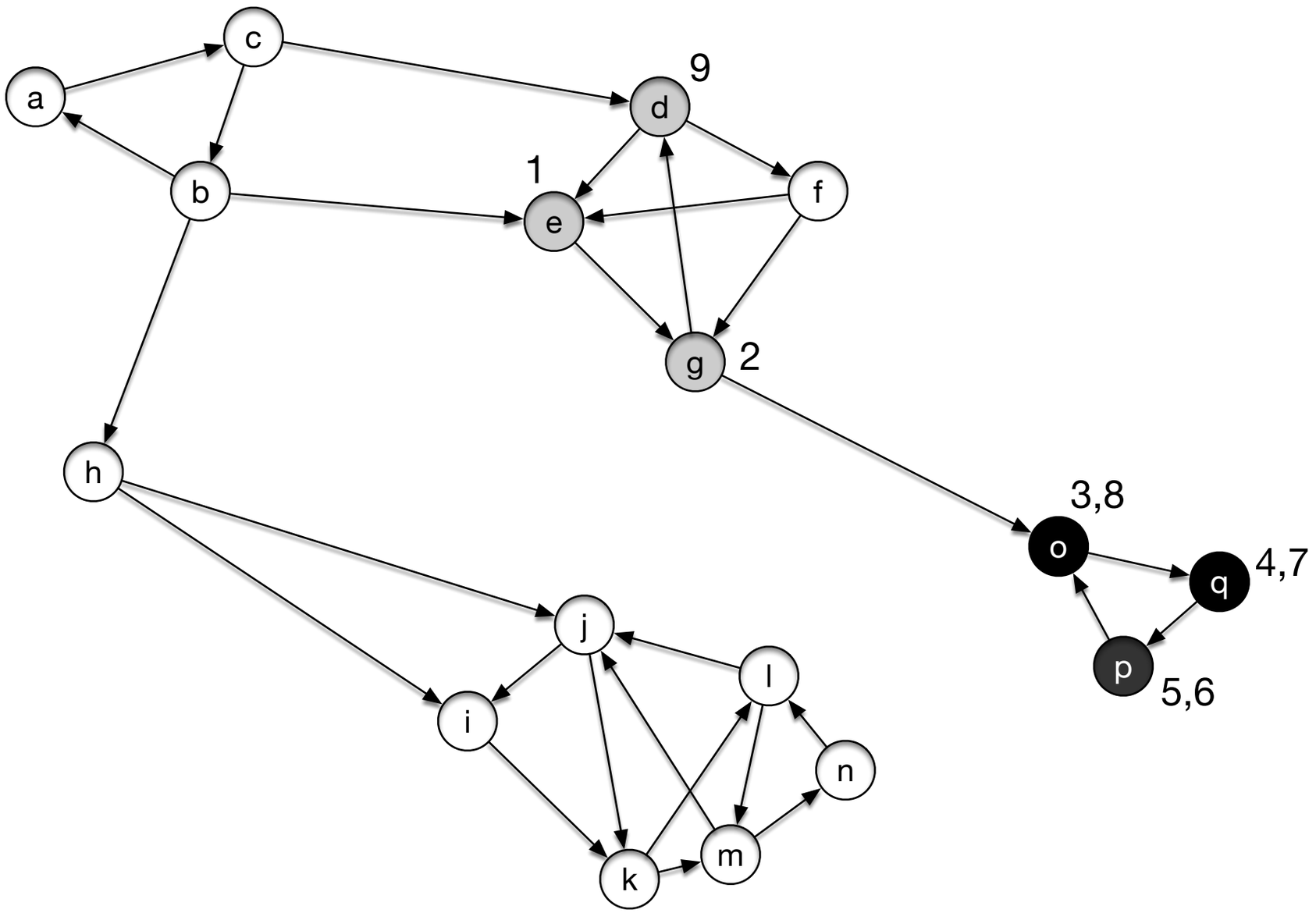
Strongly Connected Components



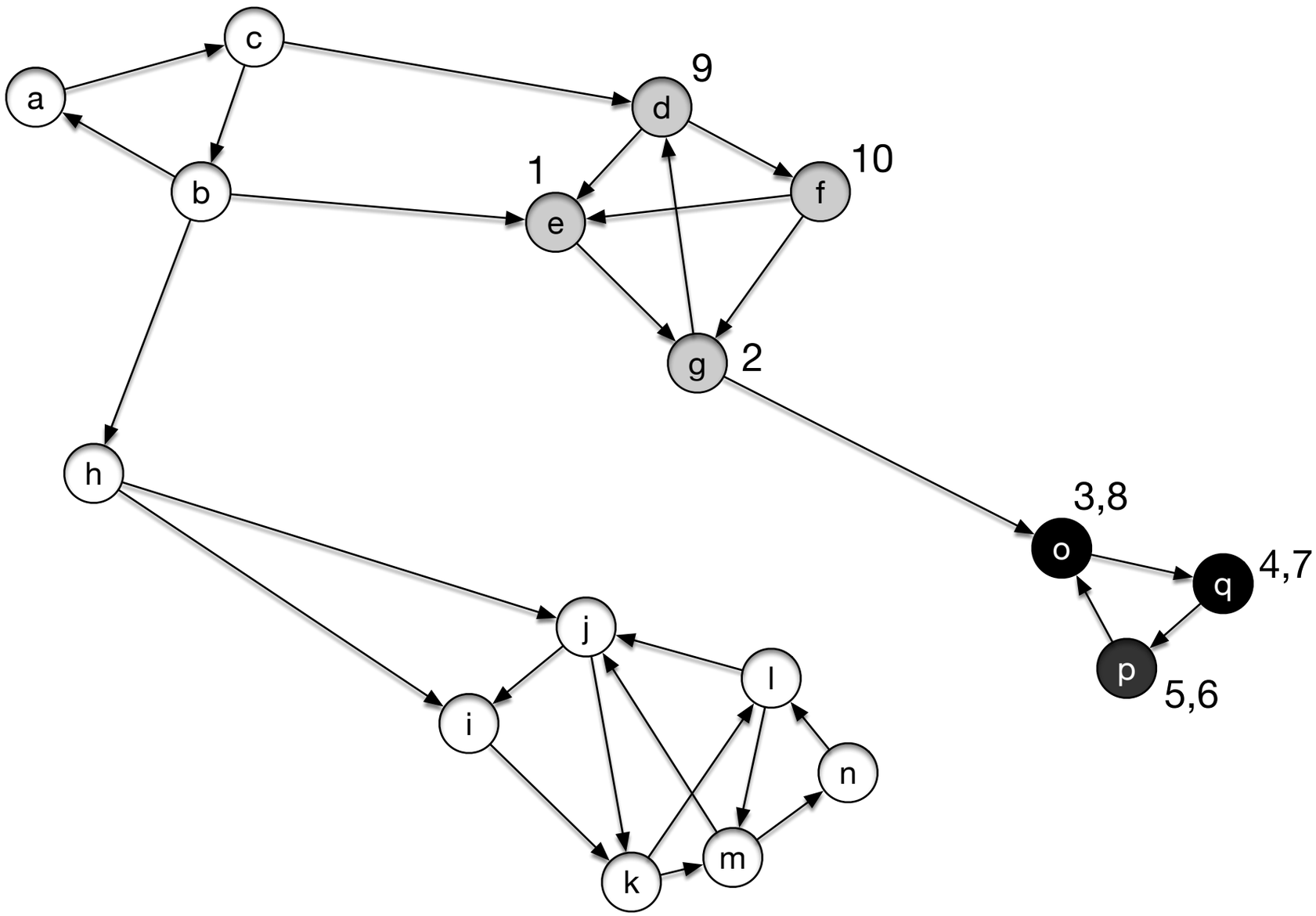
Strongly Connected Components



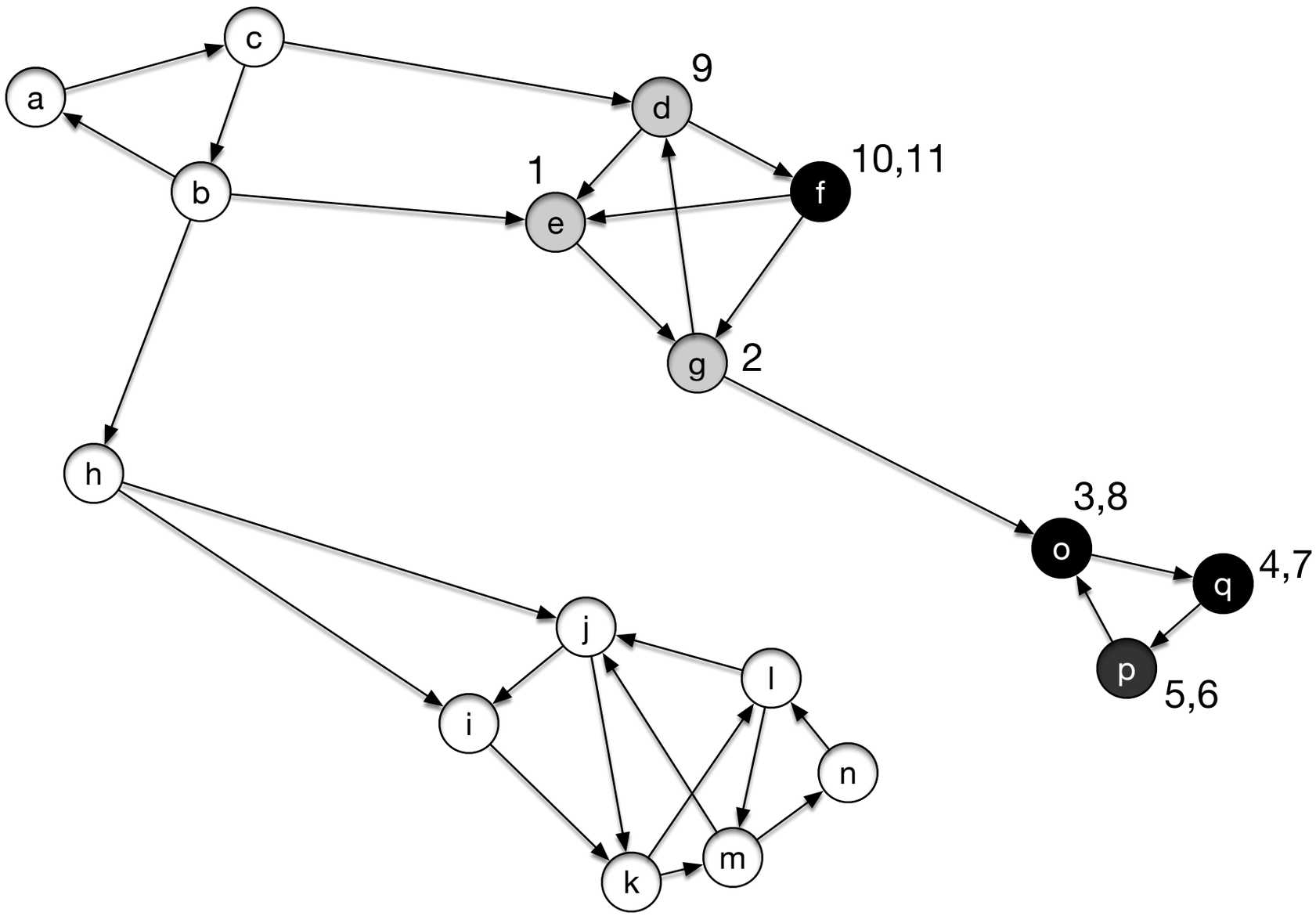
Strongly Connected Components



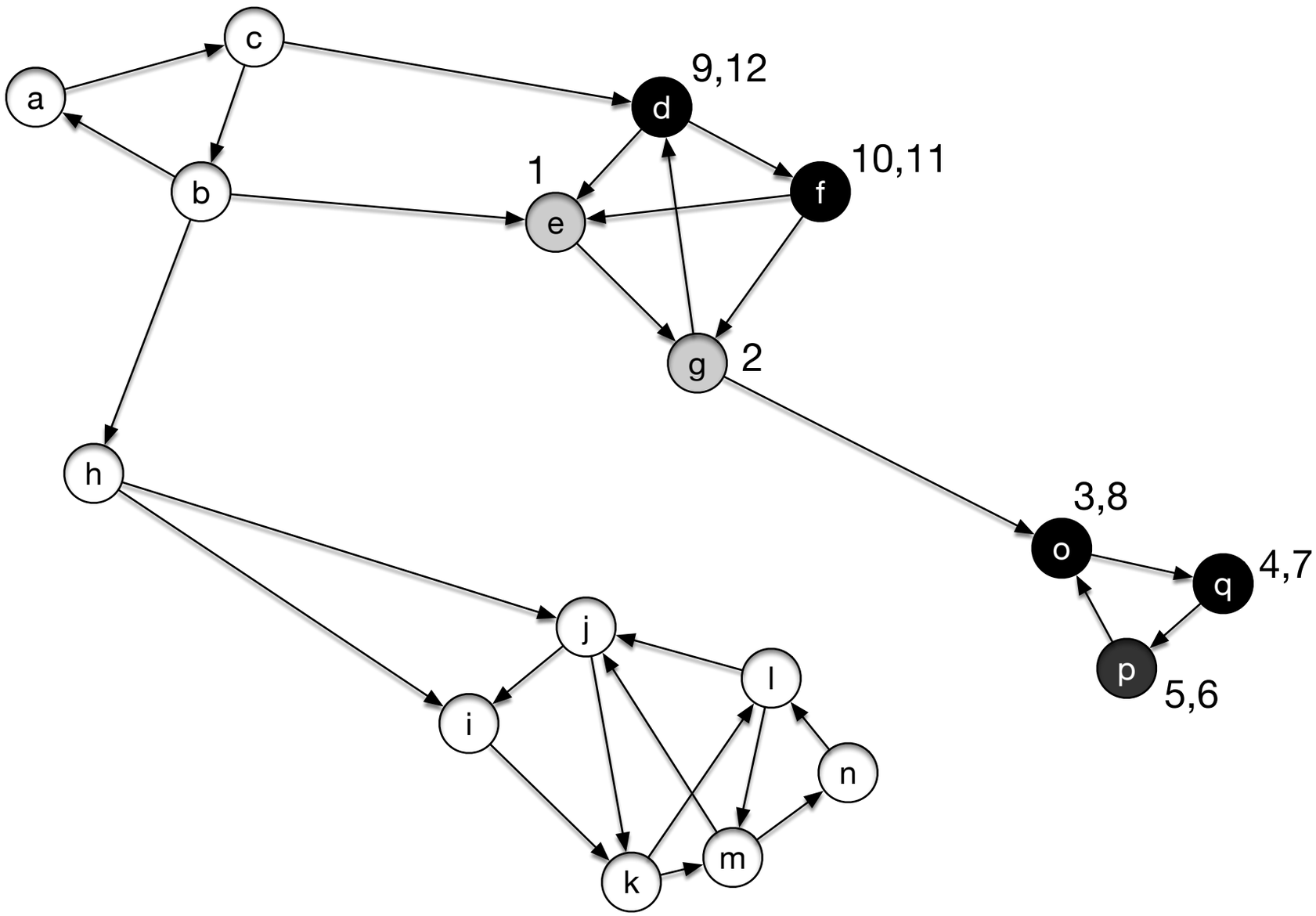
Strongly Connected Components



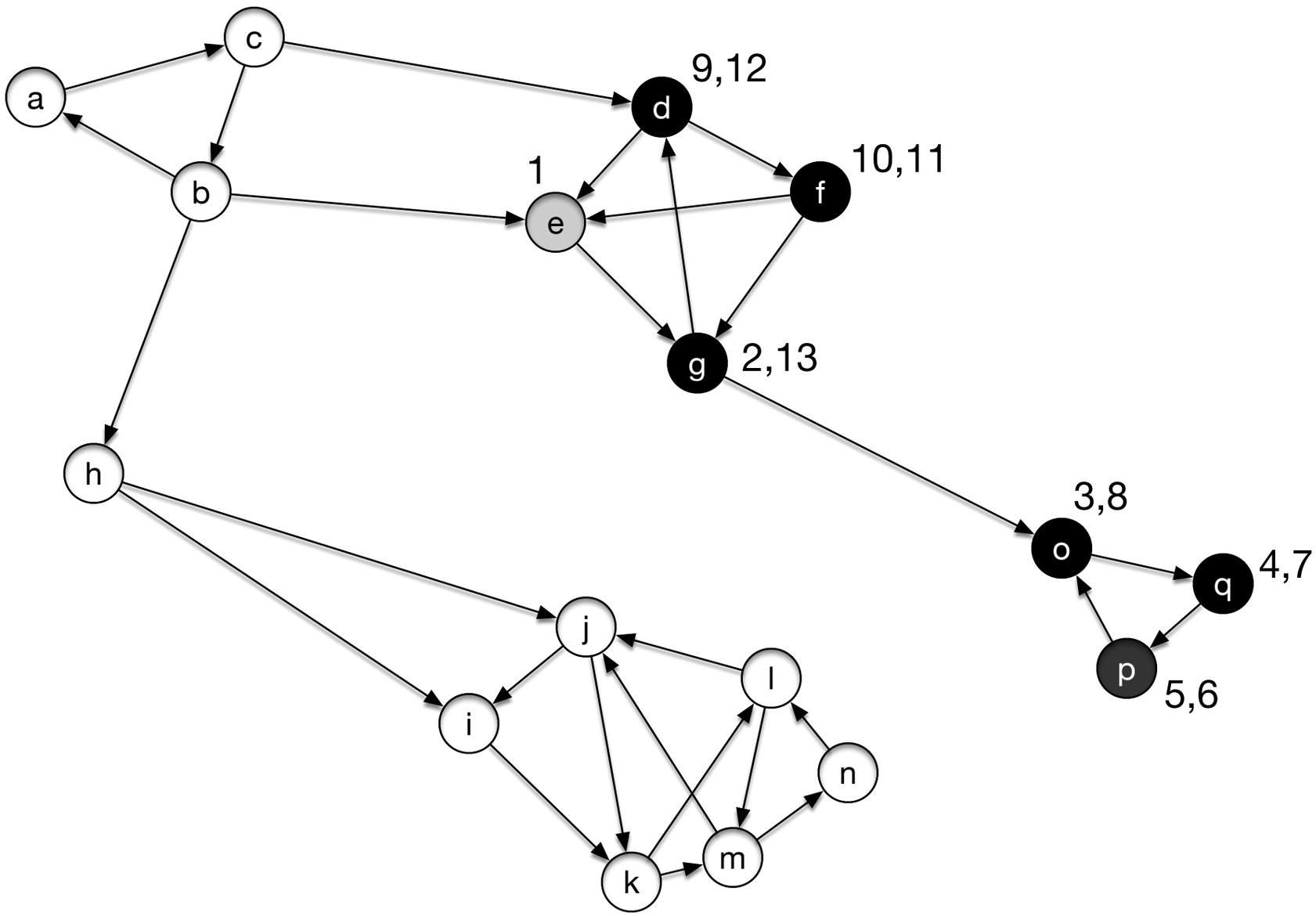
Strongly Connected Components



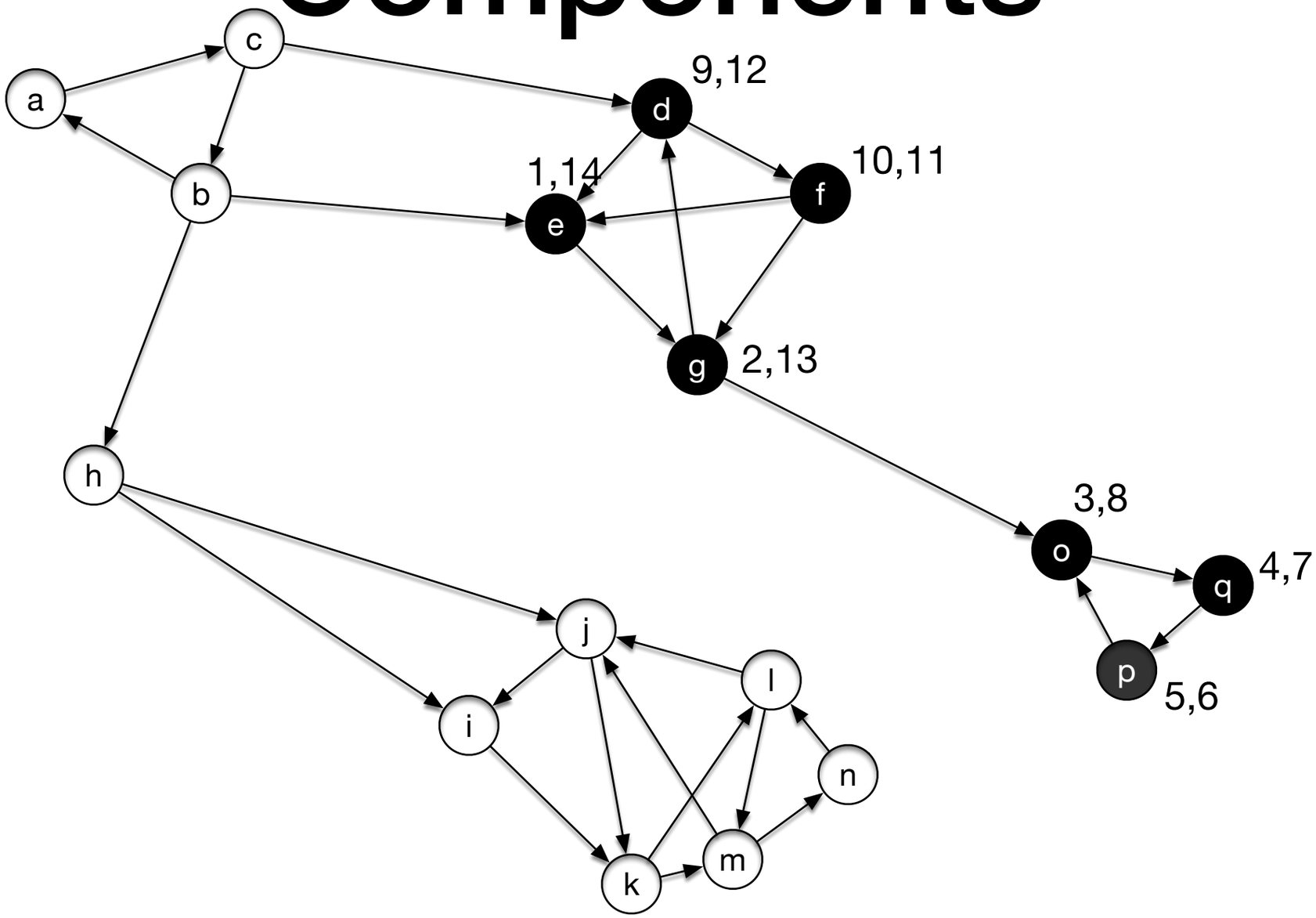
Strongly Connected Components



Strongly Connected Components



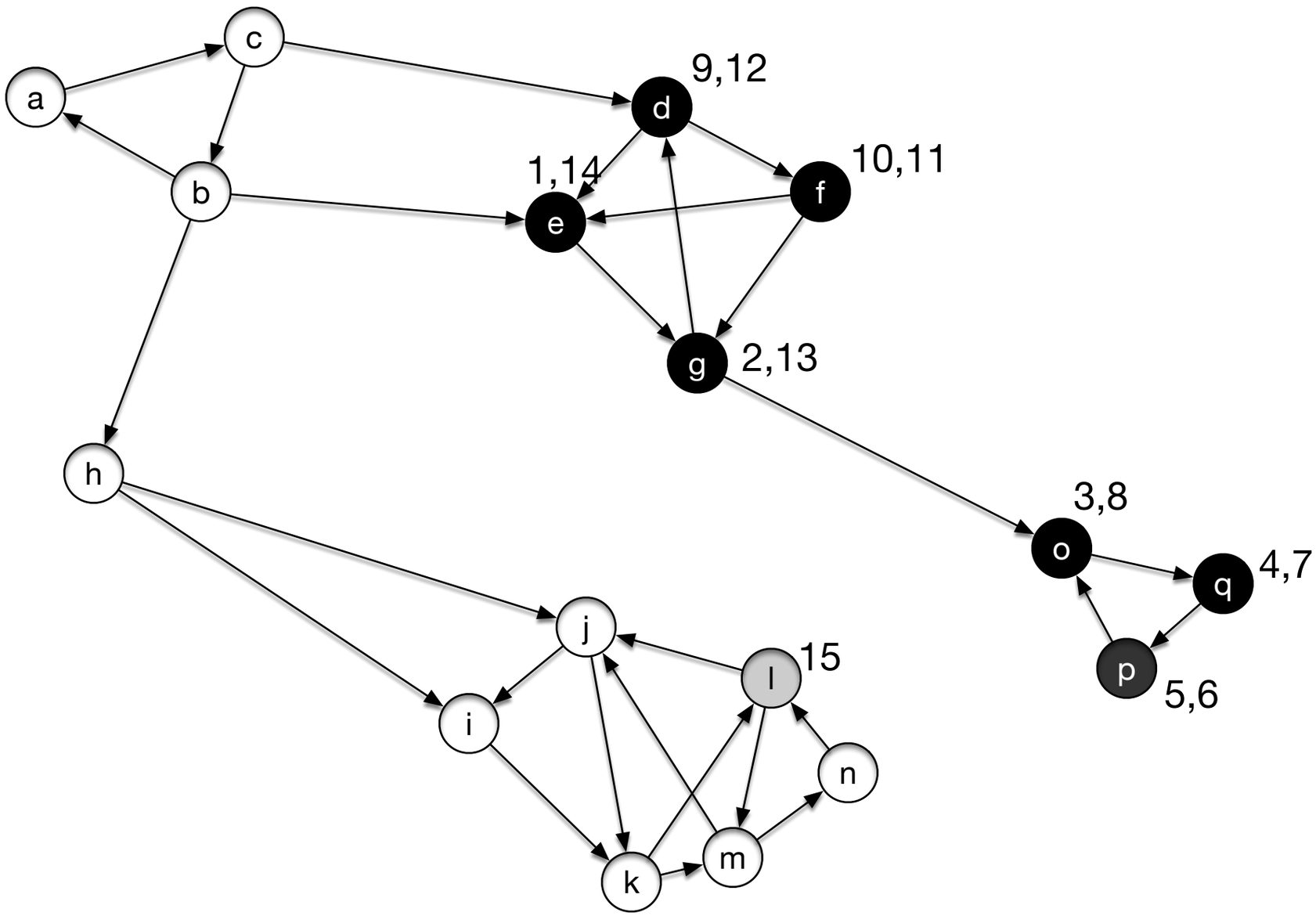
Strongly Connected Components



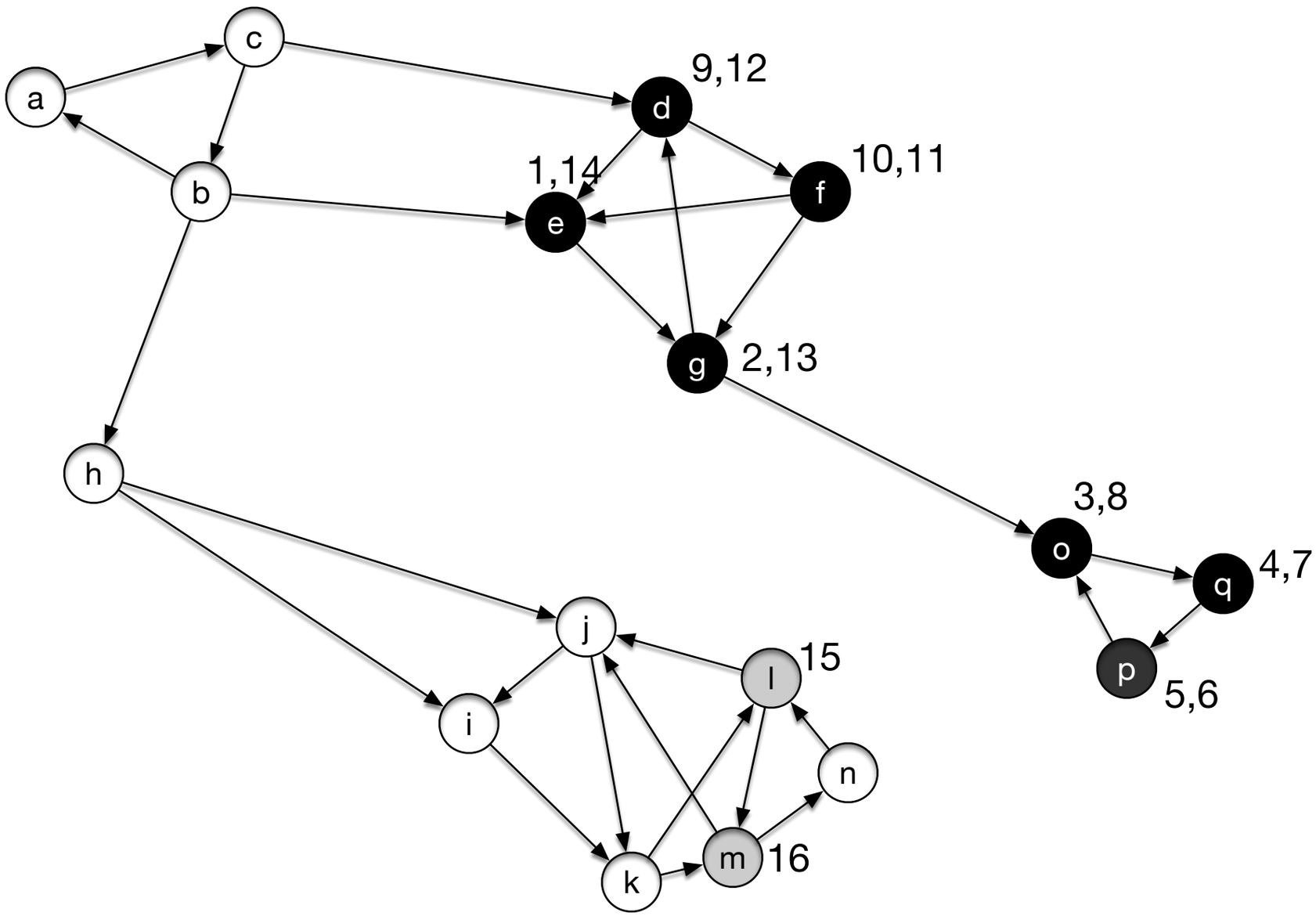
Strongly Connected Components

- The visit in E is done, we need to select a remaining white vertex. We pick I

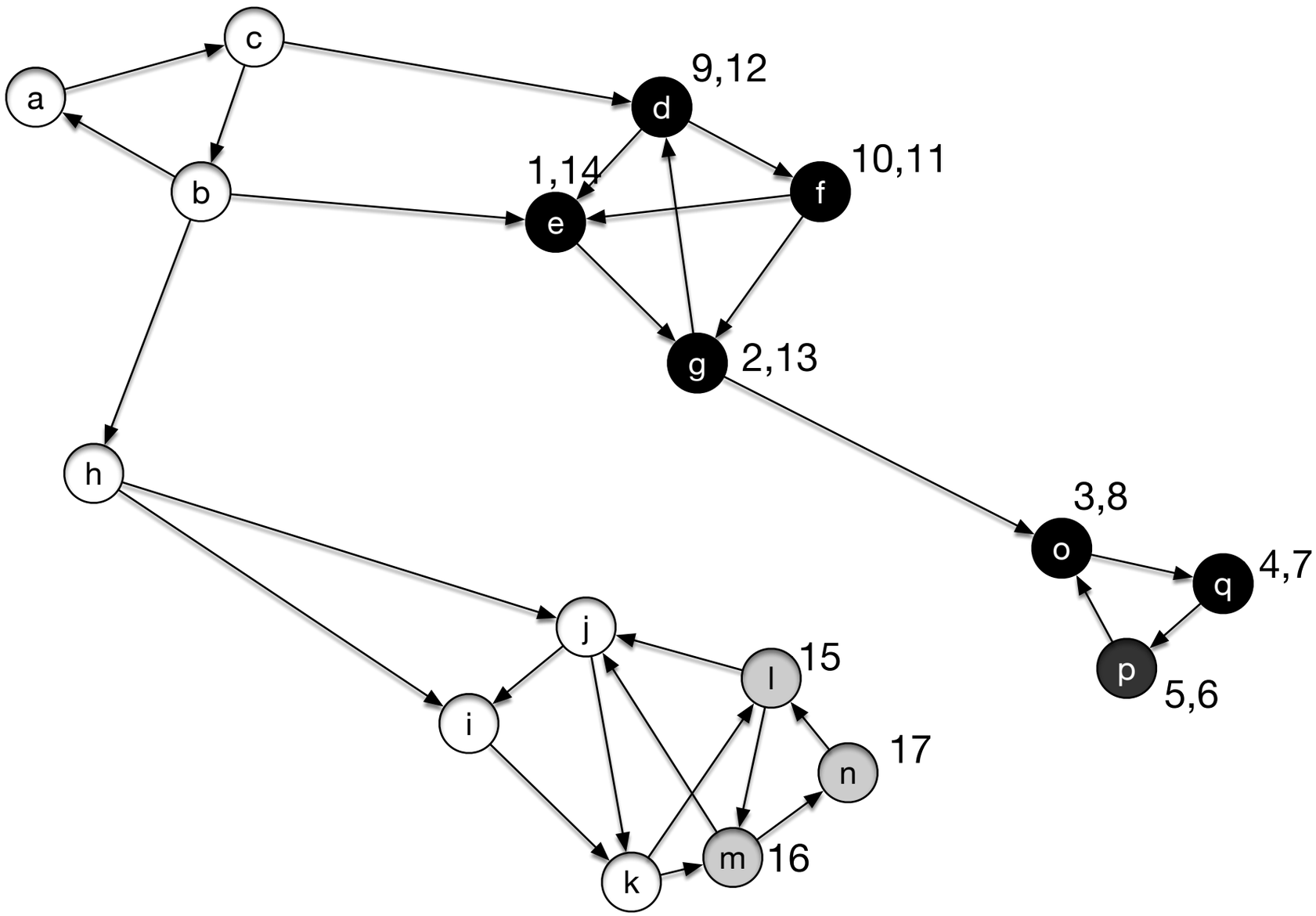
Strongly Connected Components



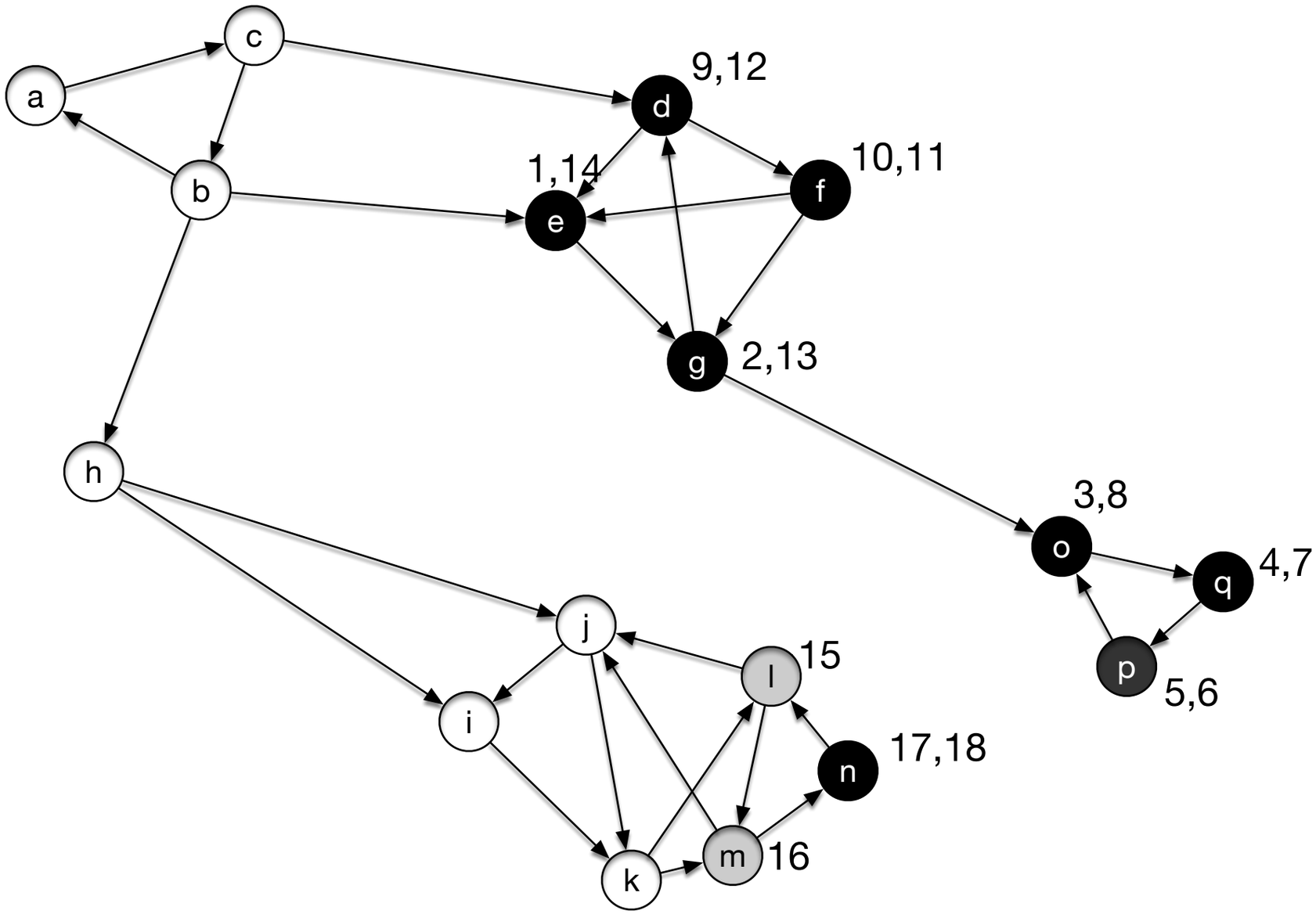
Strongly Connected Components



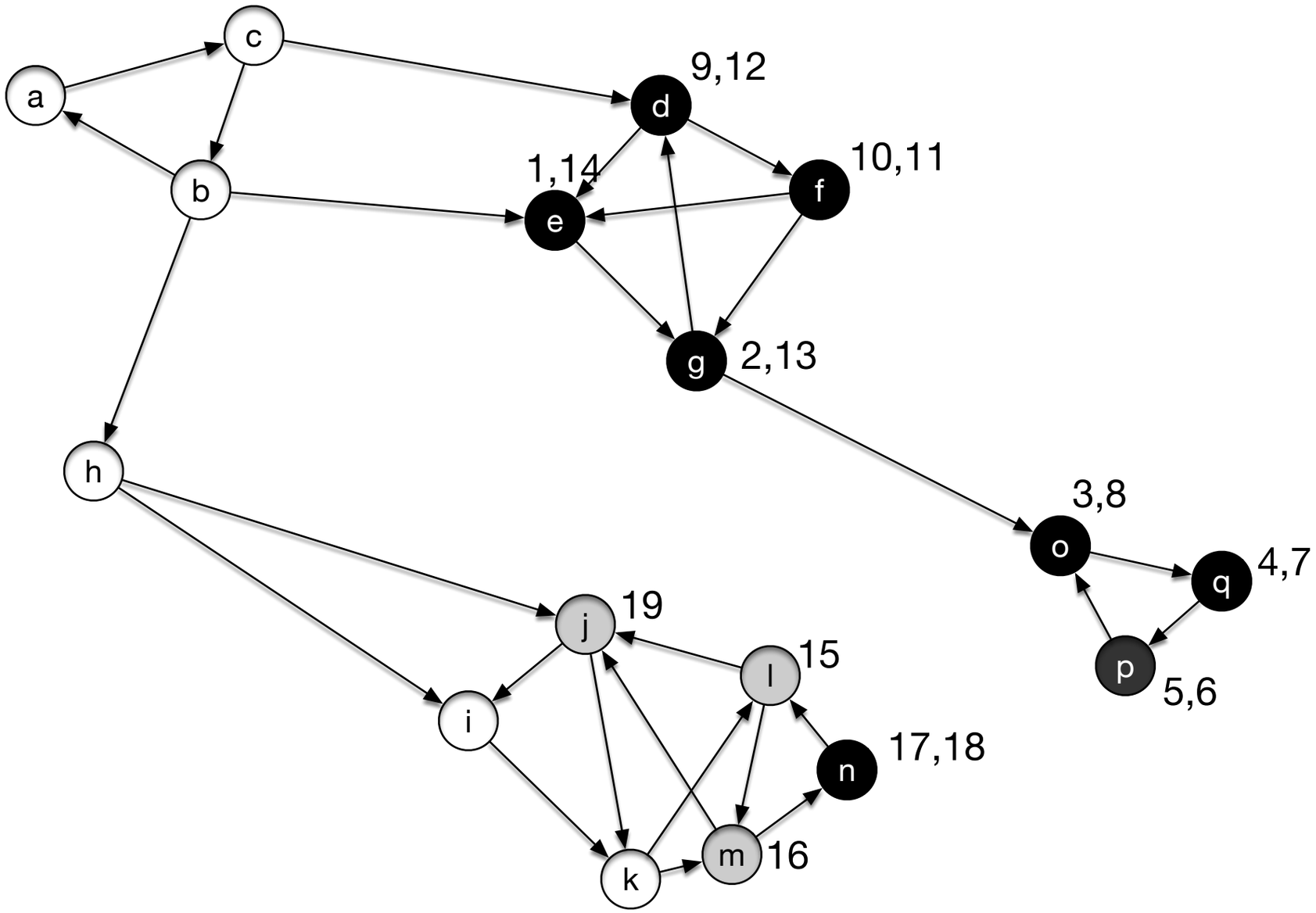
Strongly Connected Components



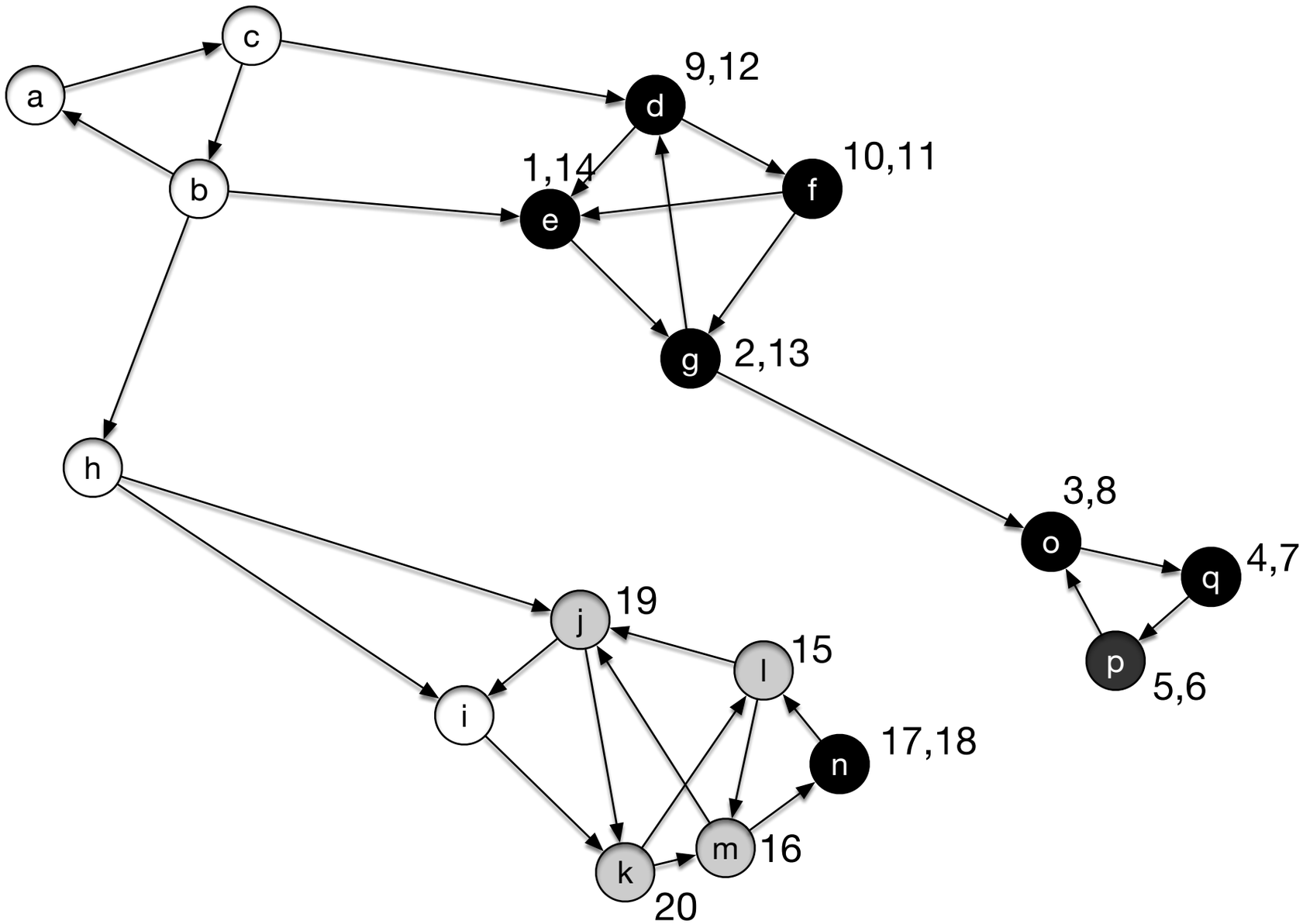
Strongly Connected Components



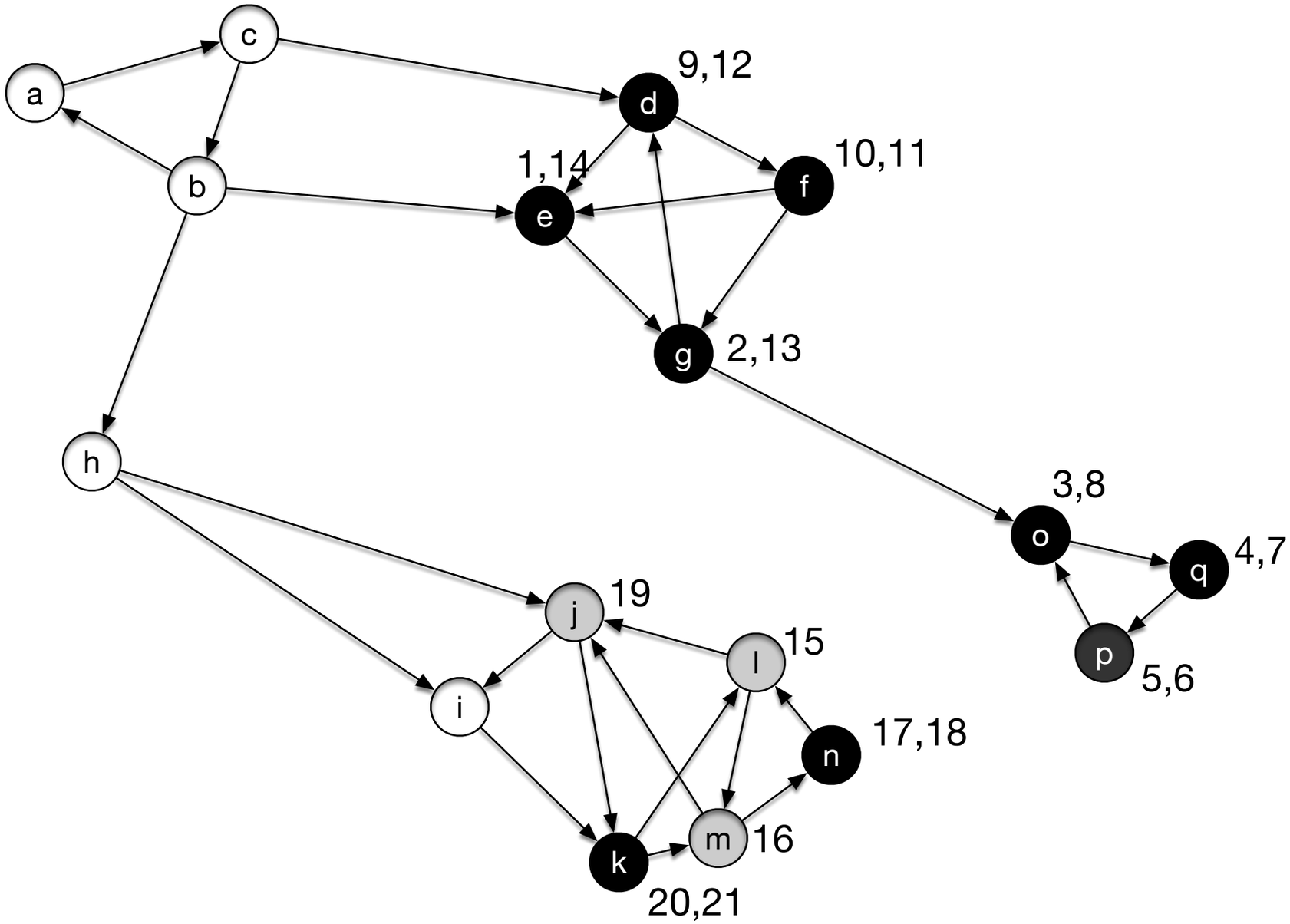
Strongly Connected Components



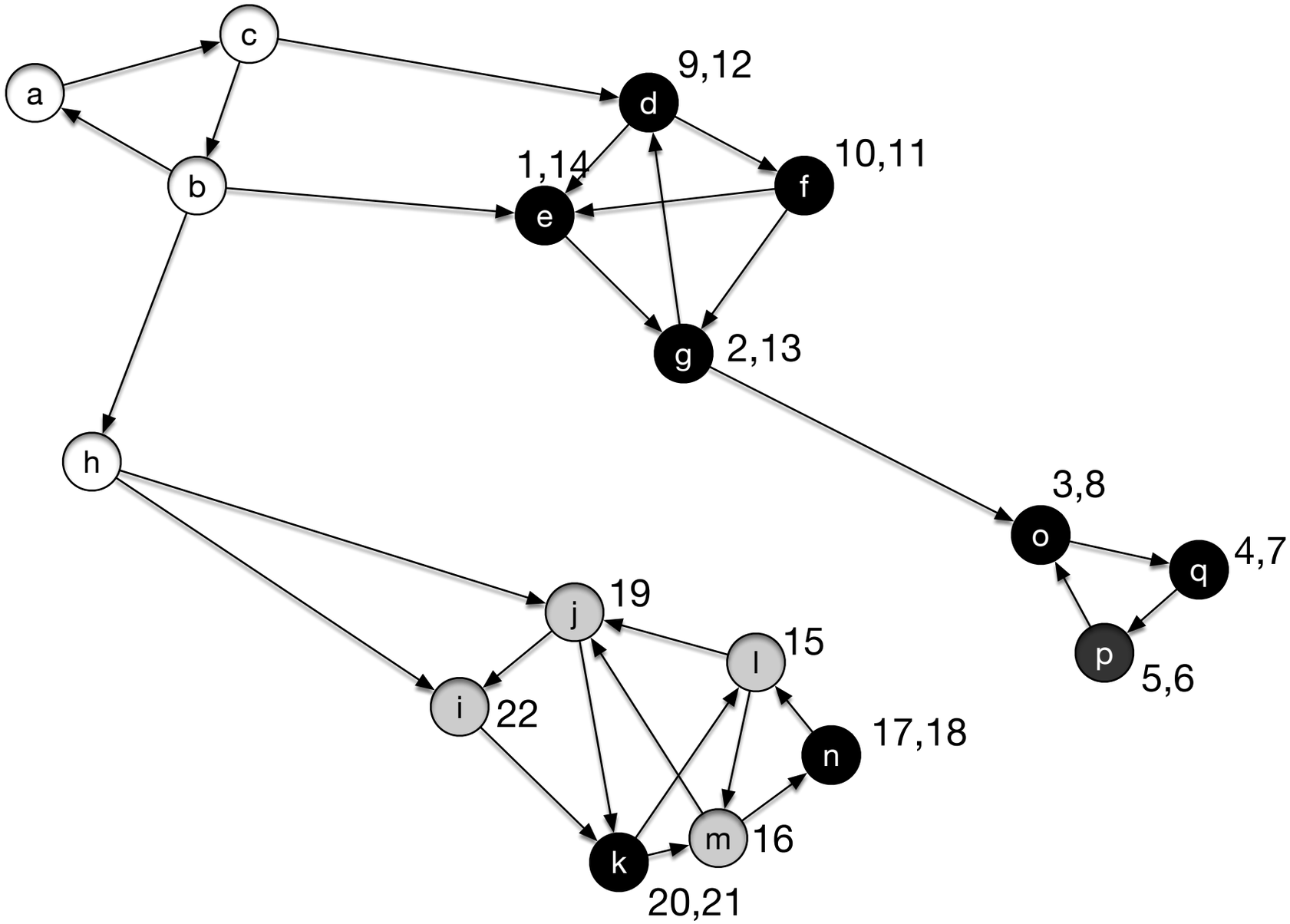
Strongly Connected Components



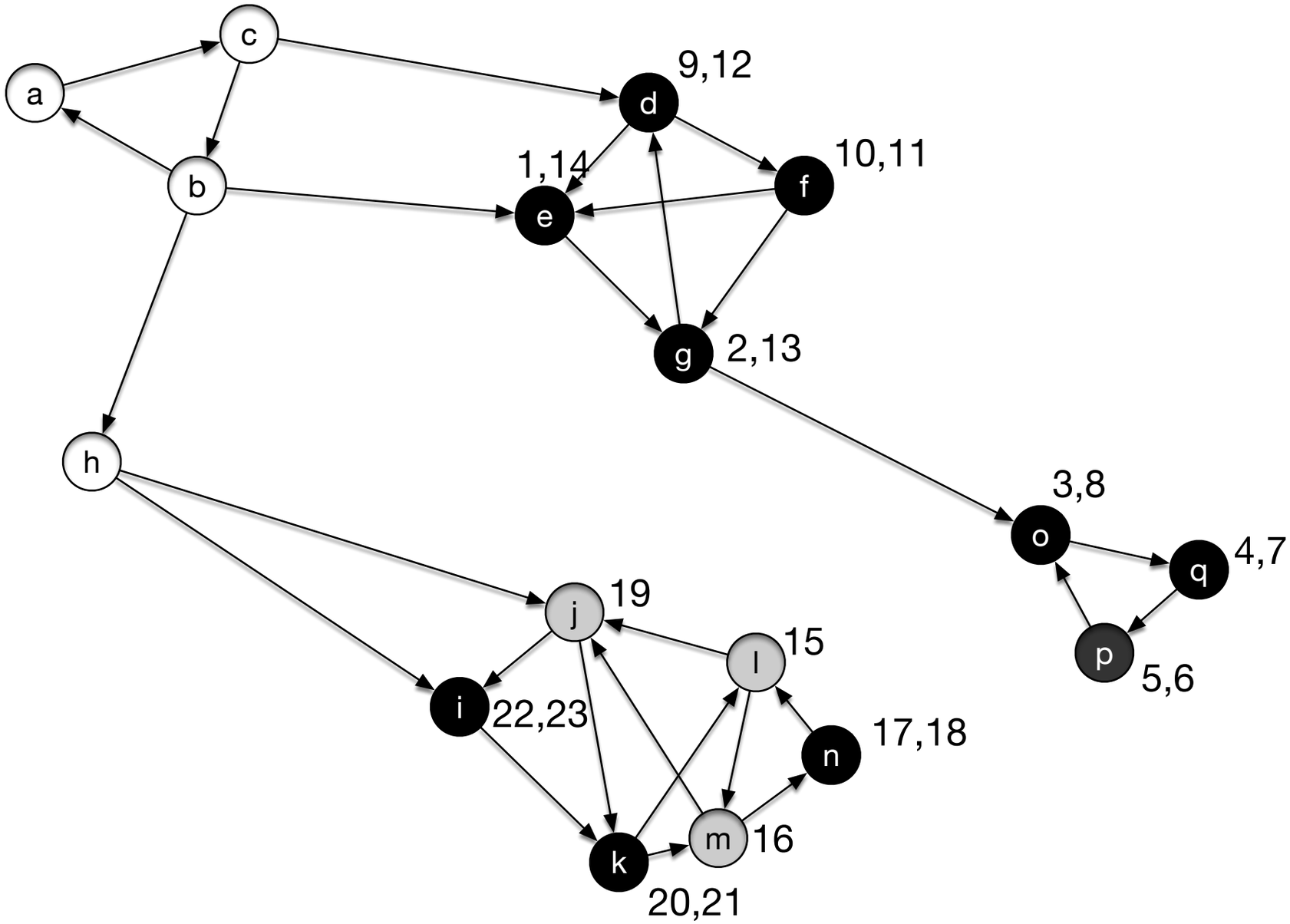
Strongly Connected Components



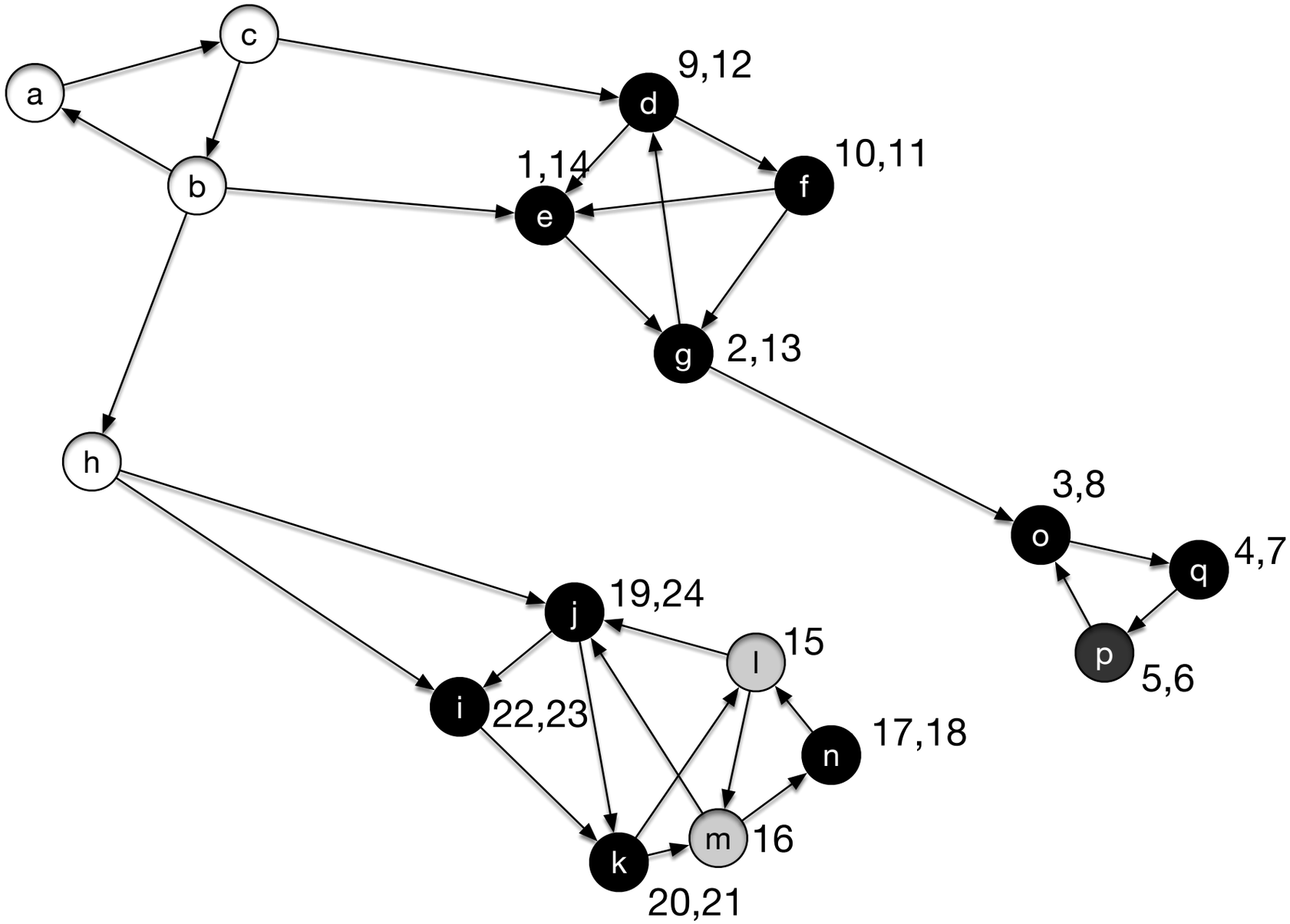
Strongly Connected Components



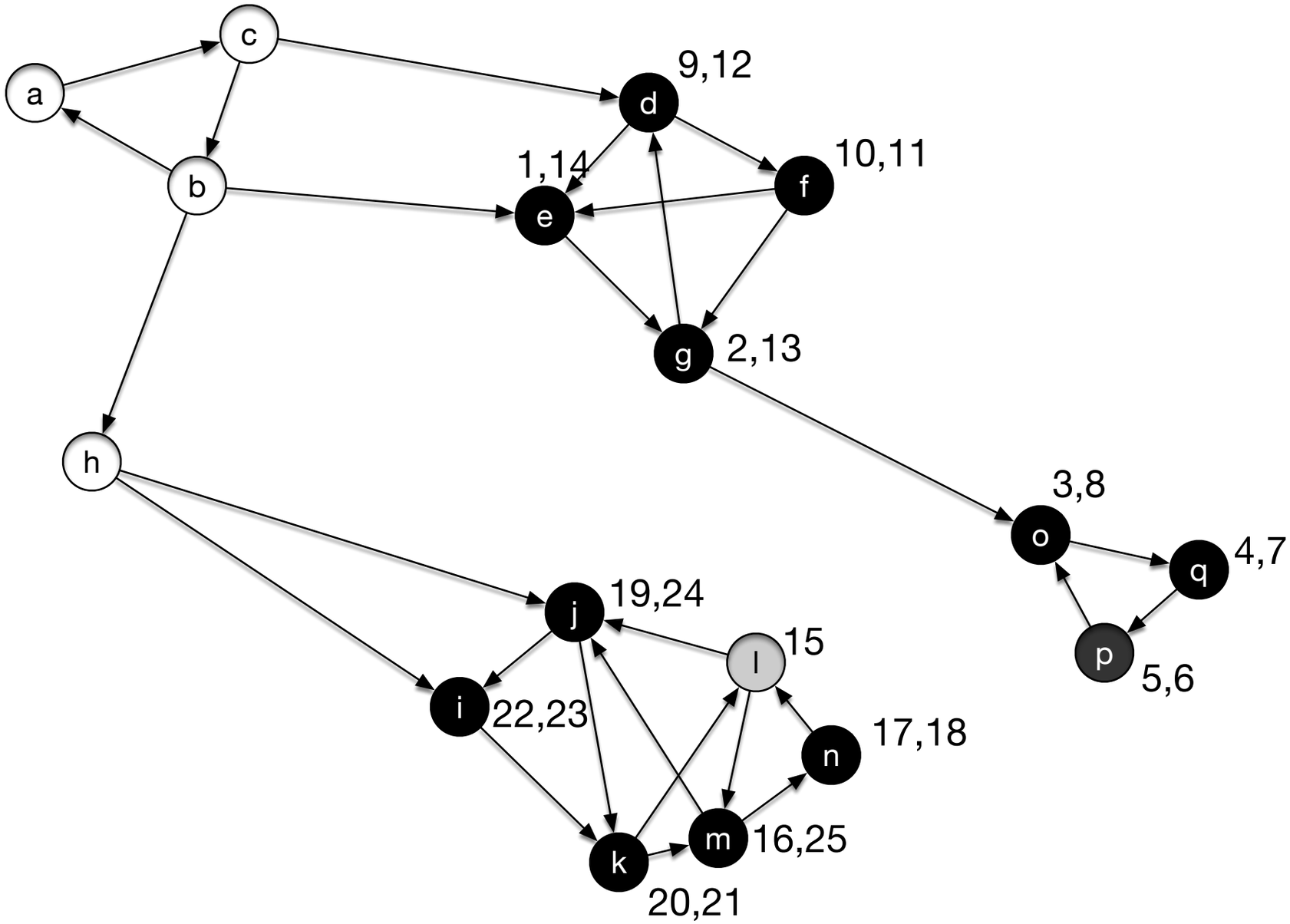
Strongly Connected Components



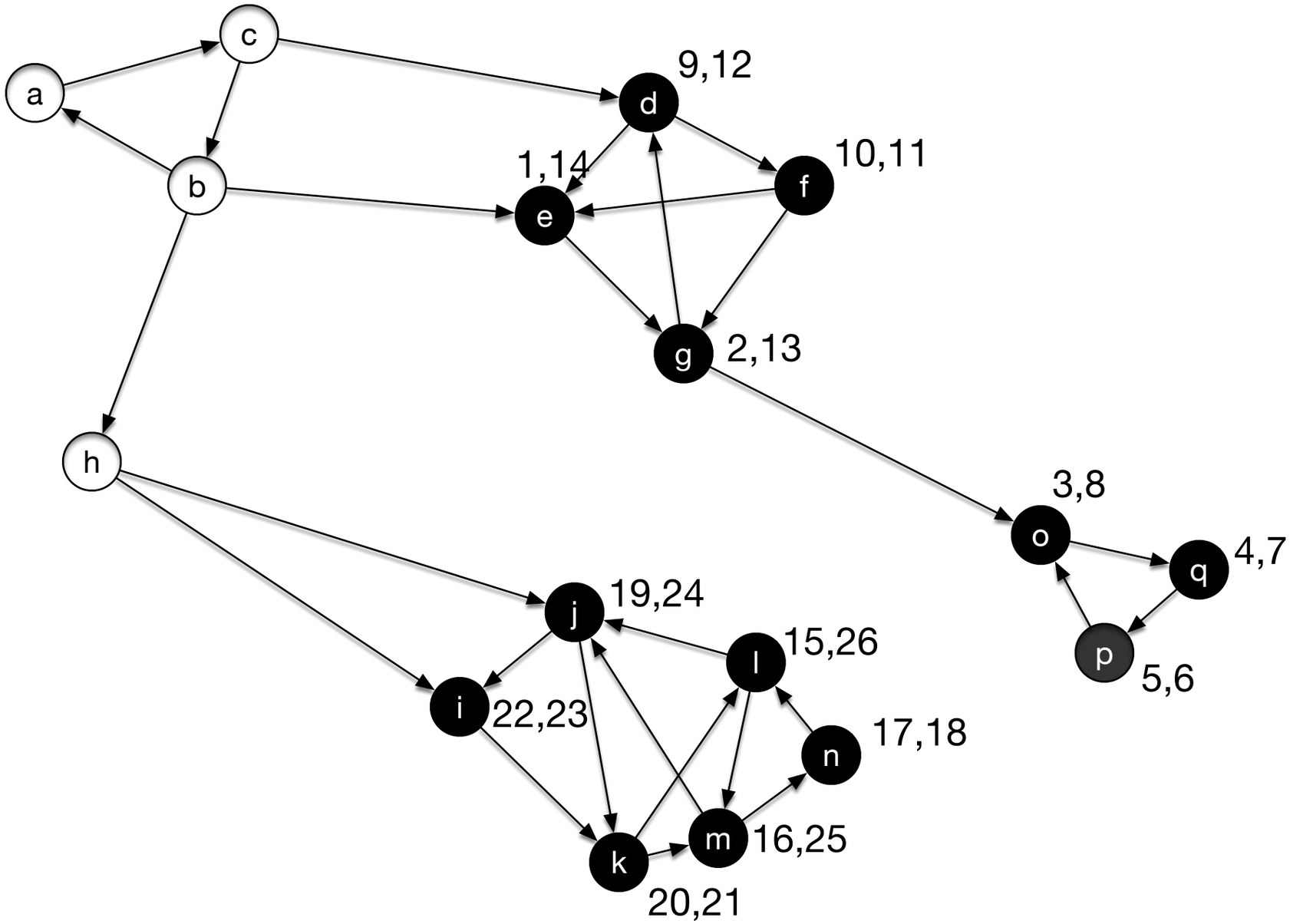
Strongly Connected Components



Strongly Connected Components



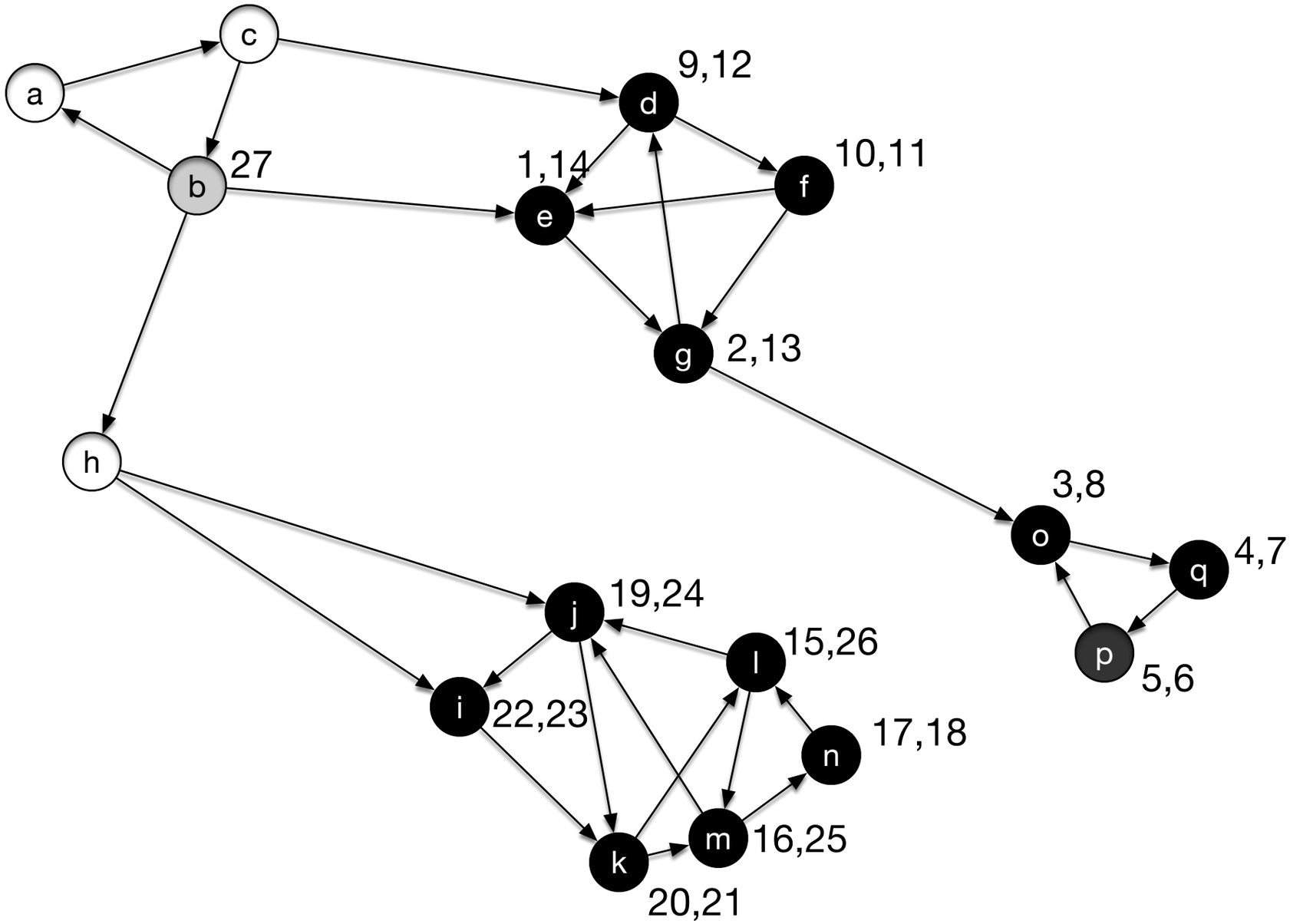
Strongly Connected Components



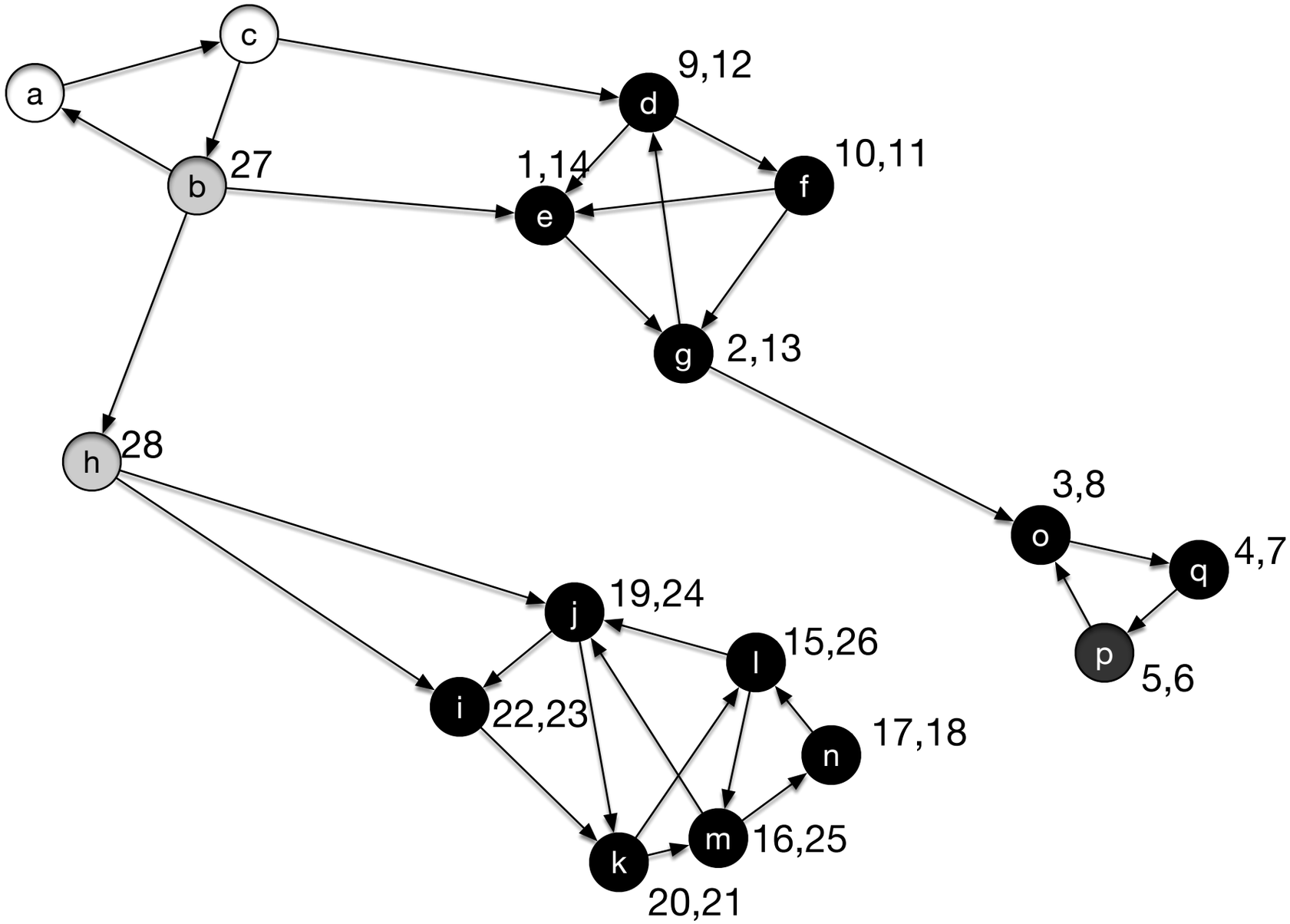
Strongly Connected Components

- Again, DFS will select a white node. Let it be b

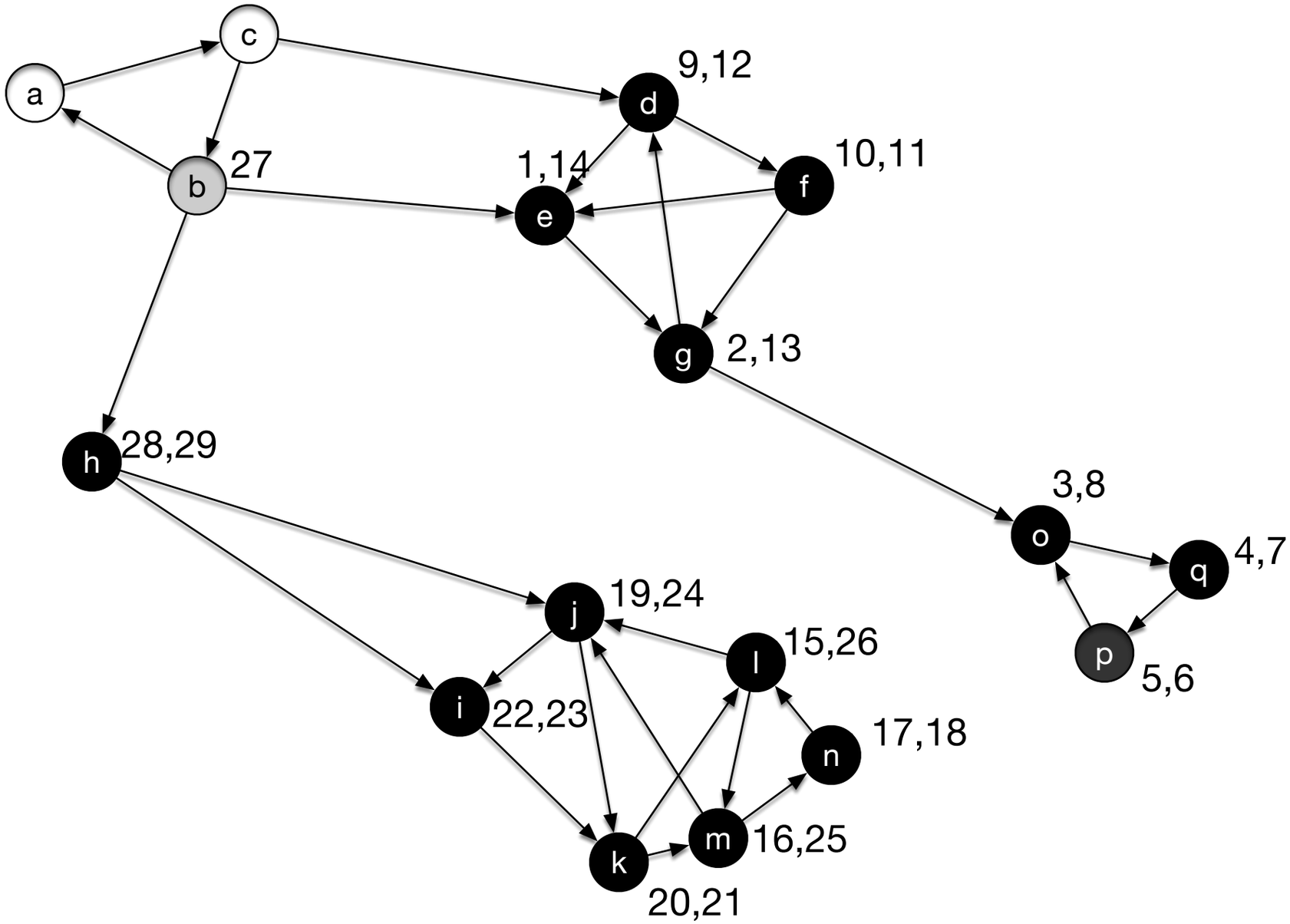
Strongly Connected Components



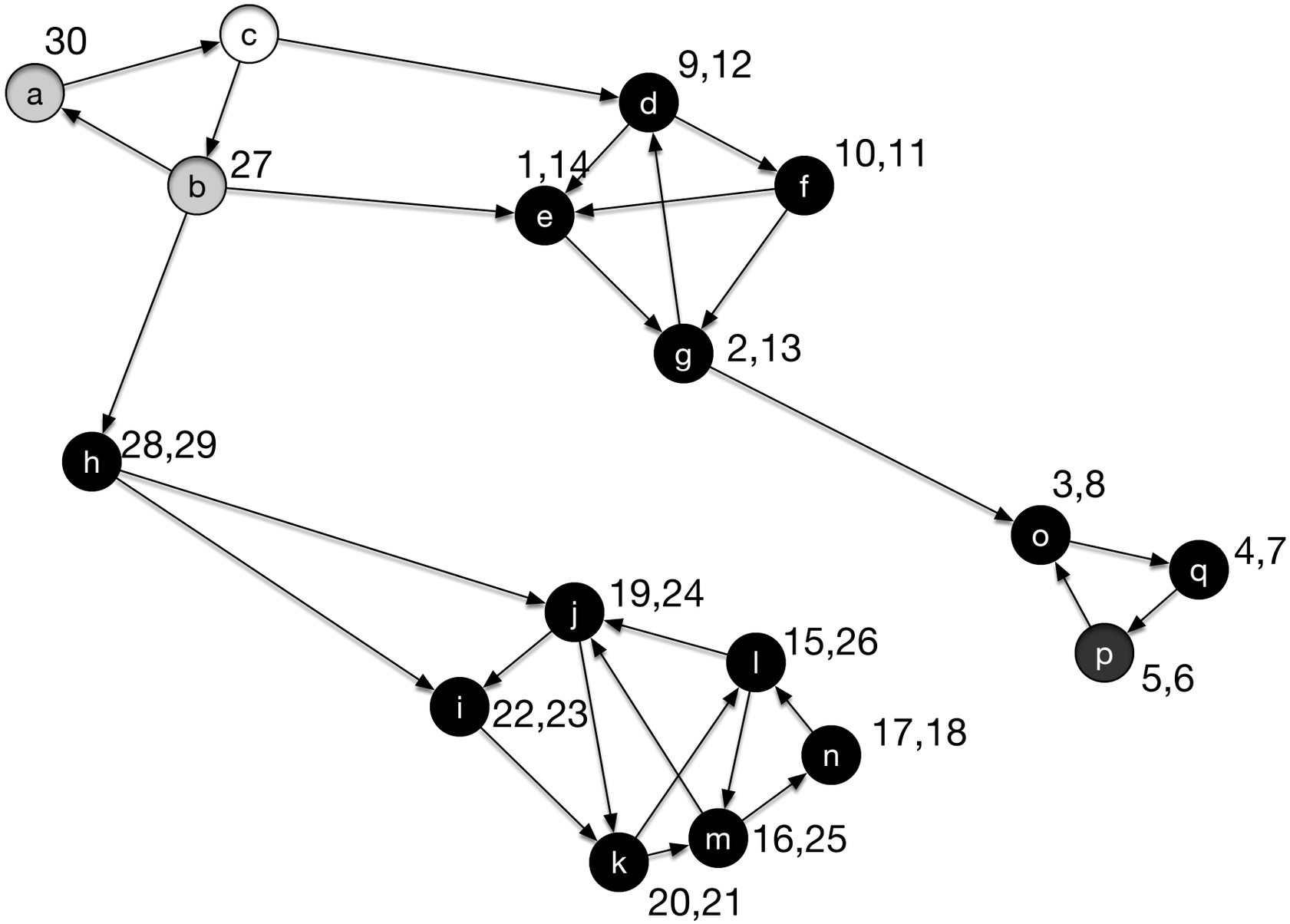
Strongly Connected Components



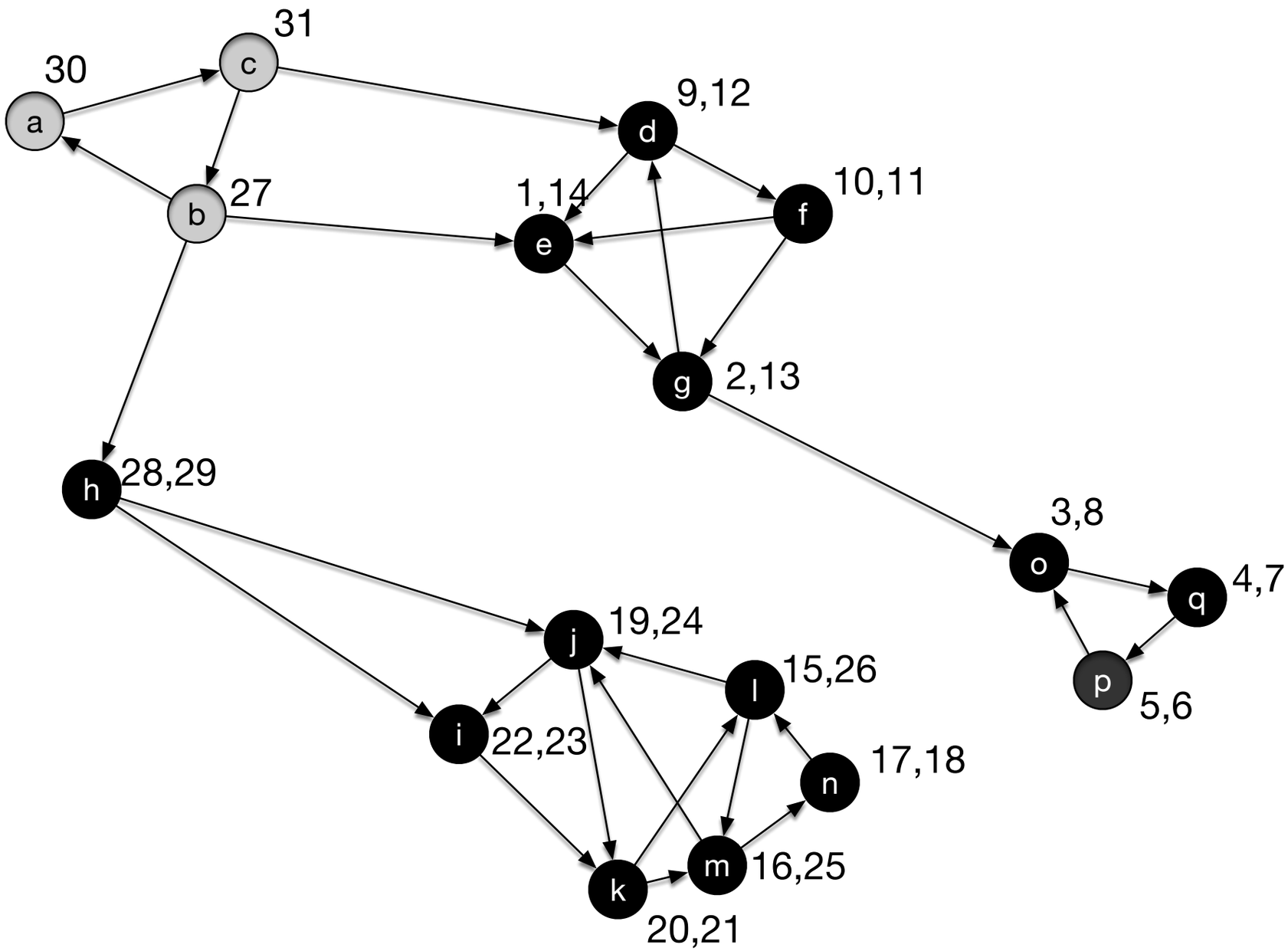
Strongly Connected Components



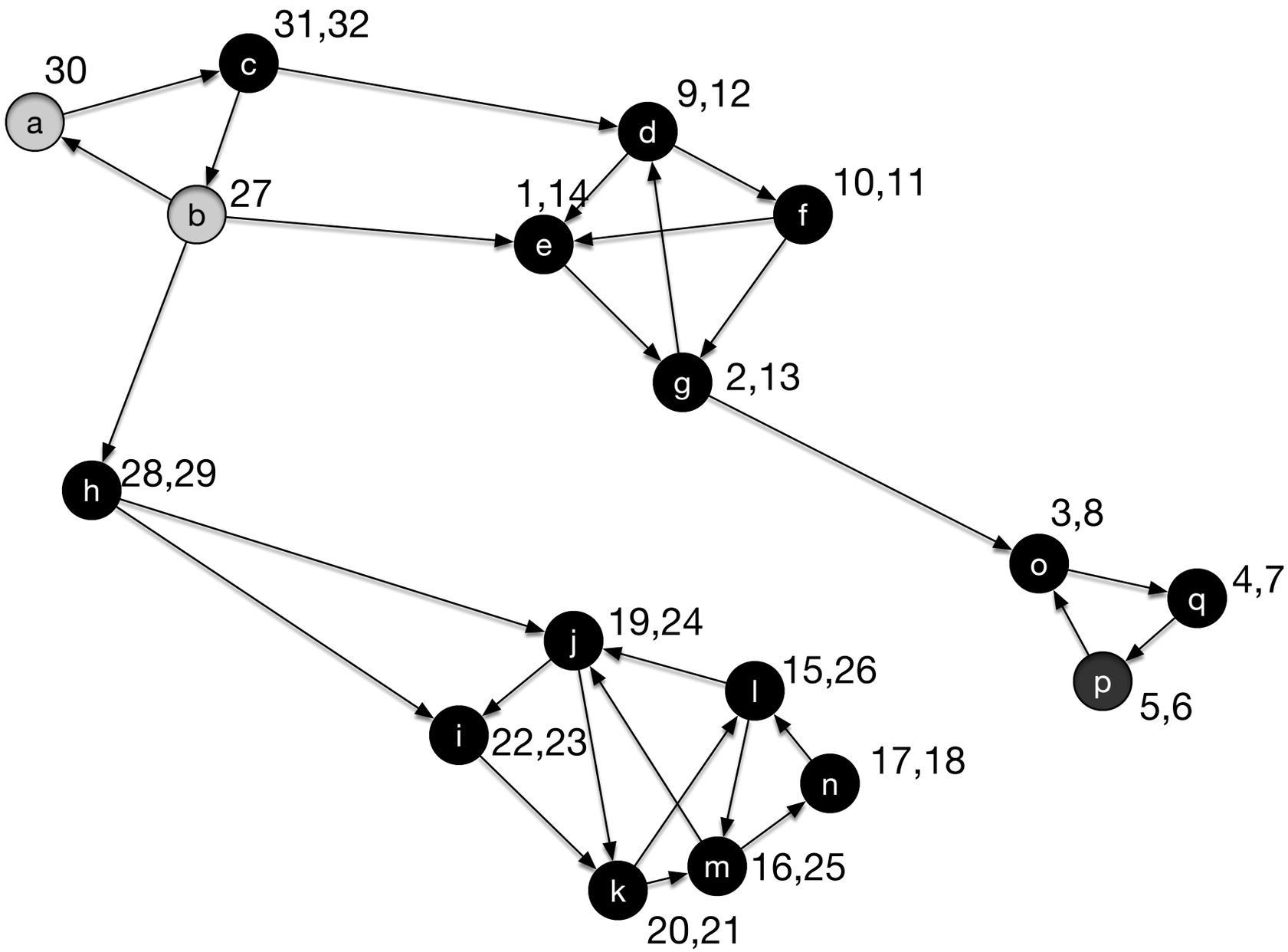
Strongly Connected Components



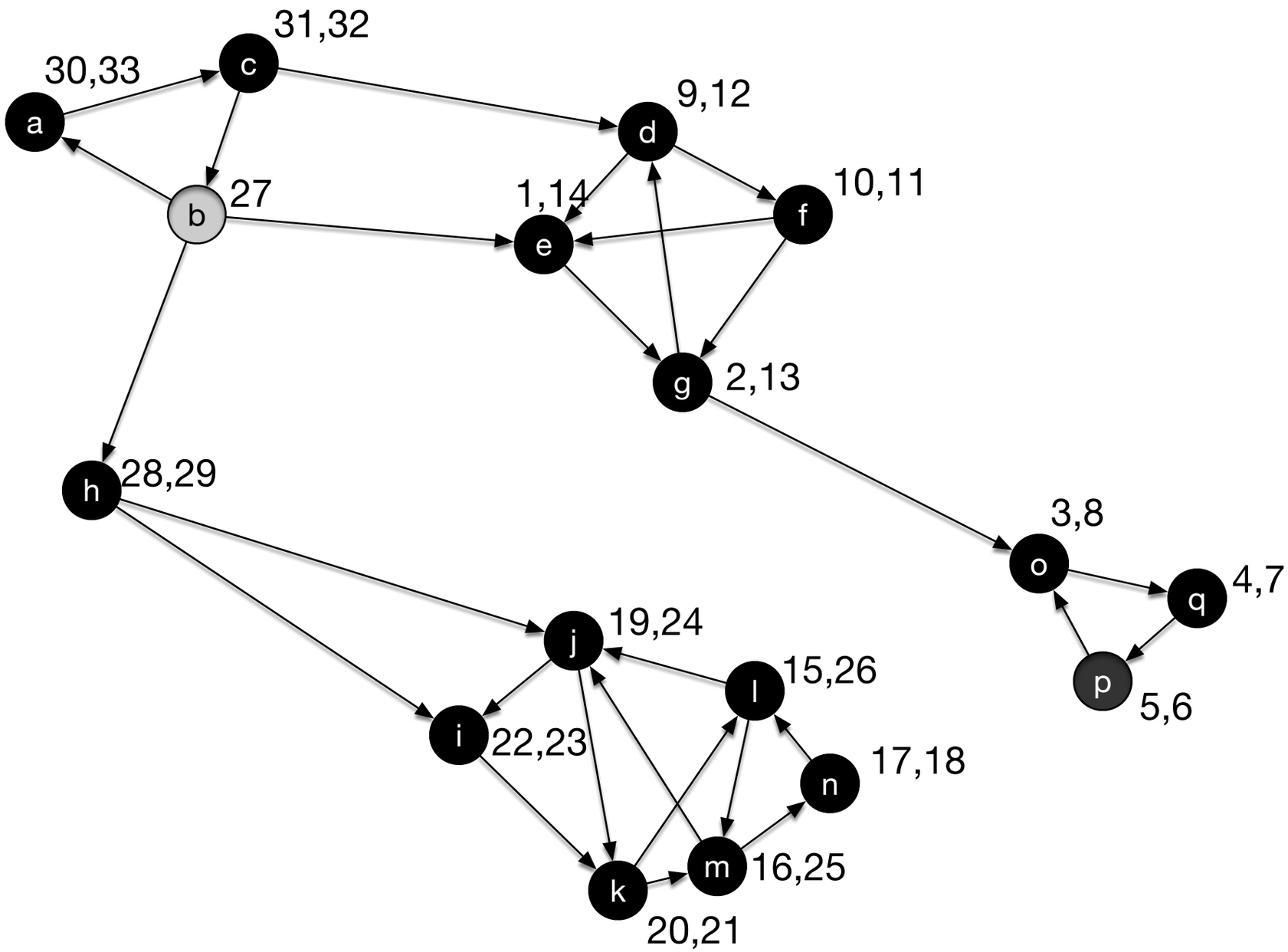
Strongly Connected Components



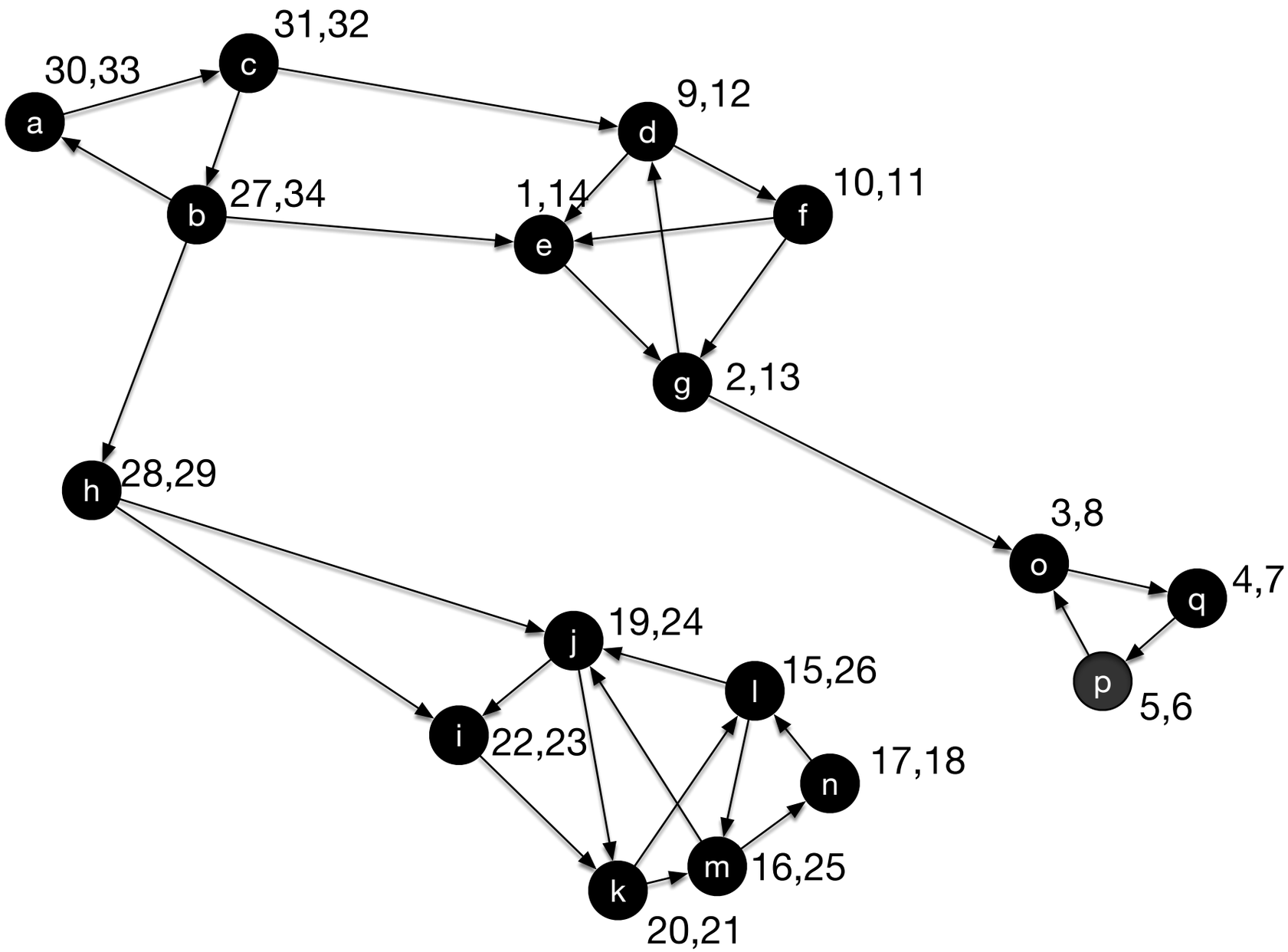
Strongly Connected Components



Strongly Connected Components



Strongly Connected Components

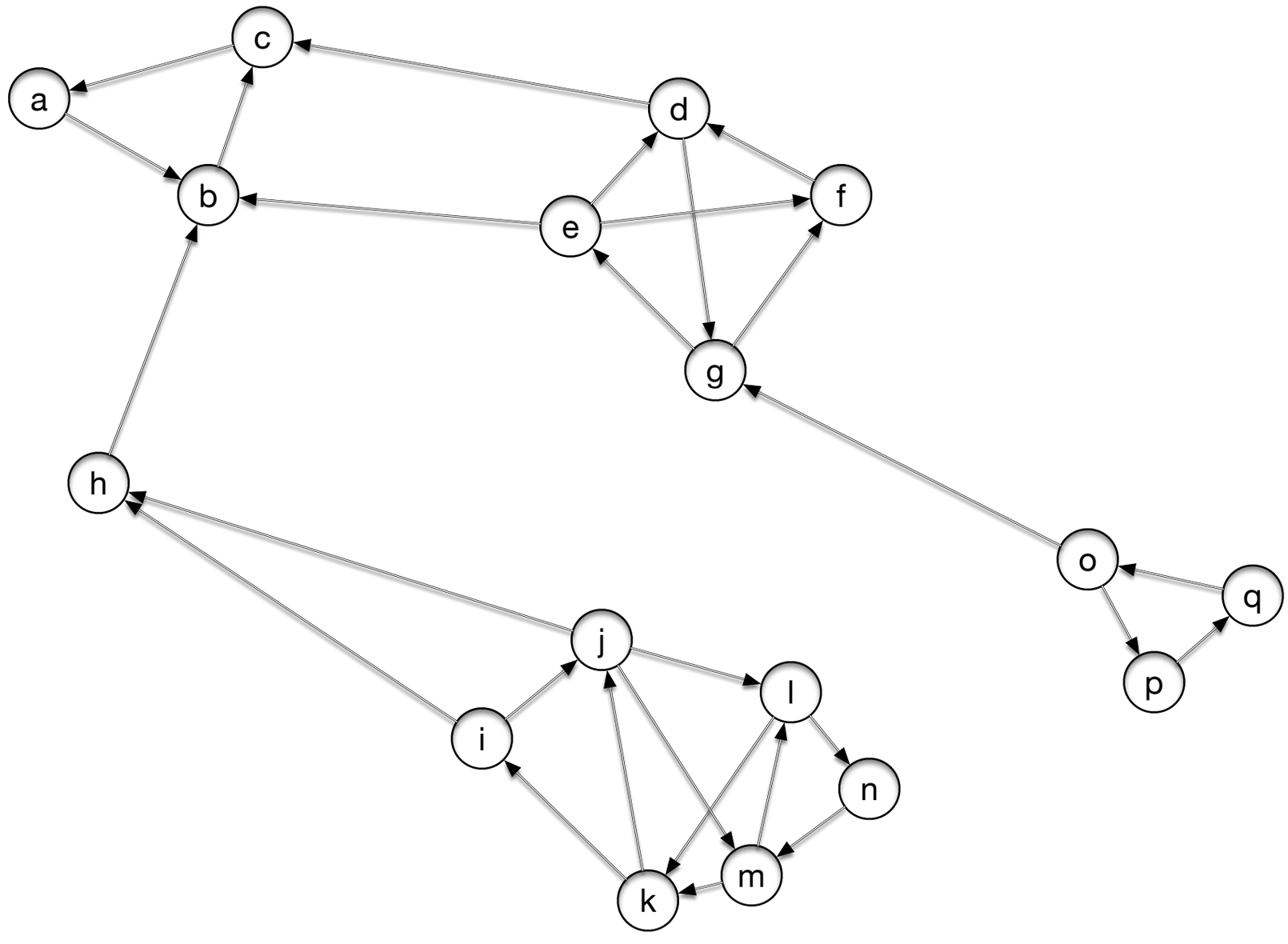


Strongly Connected Components

- Now:
  - Decorate each vertex with the finishing time obtained
  - Reverse all edges (in linear time)
  - Start DFS, but select starting vertices in decreasing order of the finishing time
  - This means we start with b

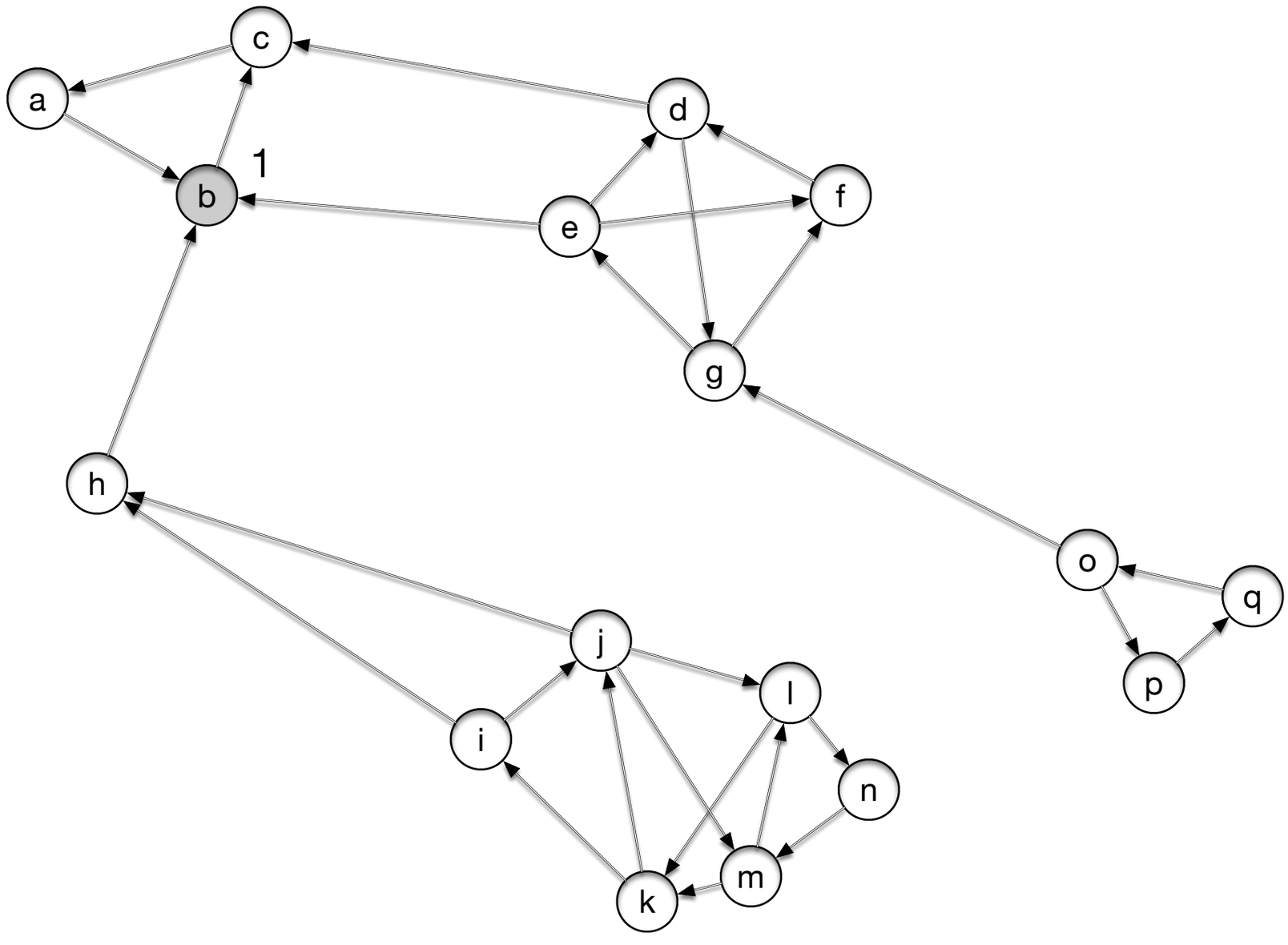
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



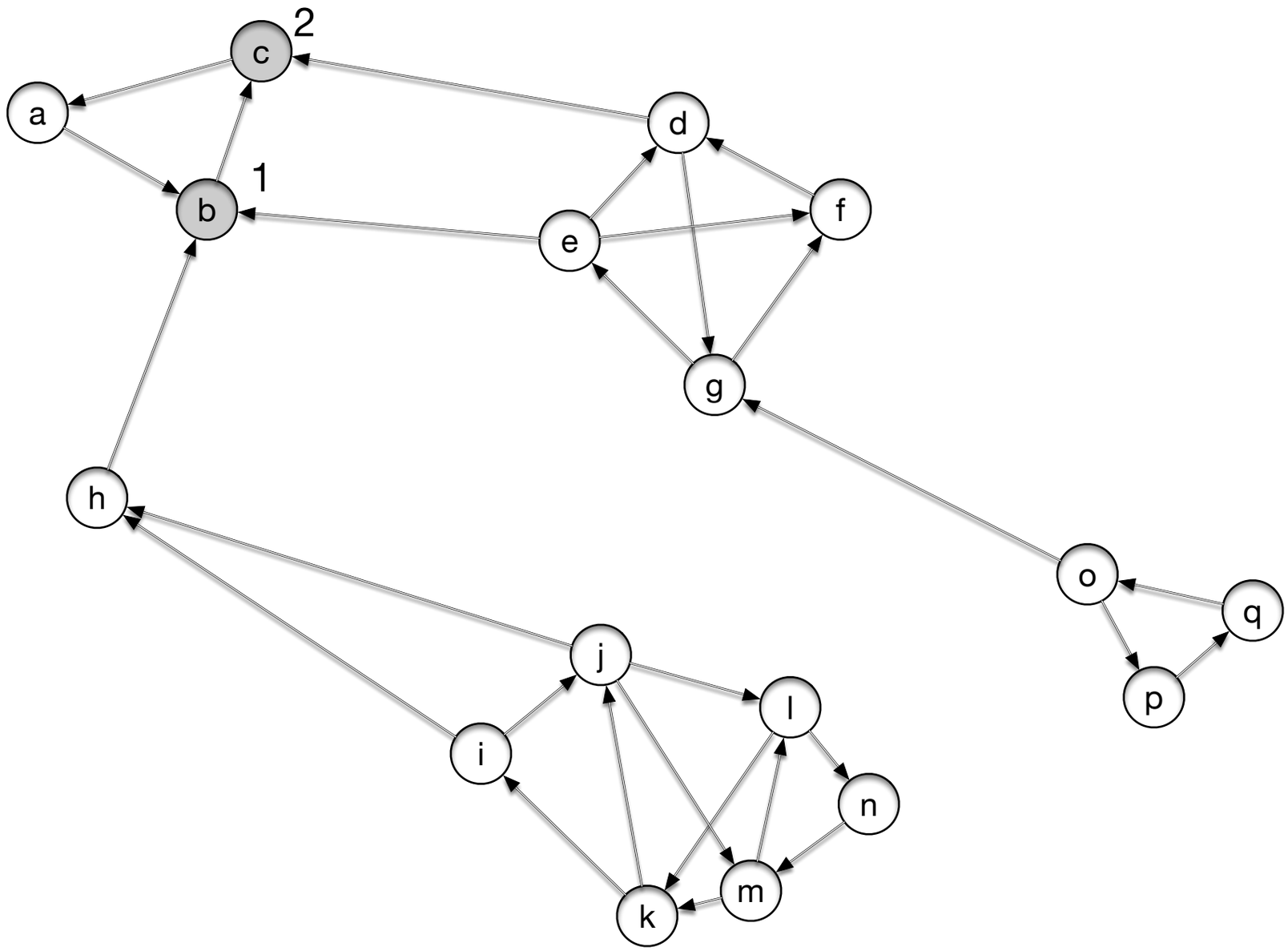
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



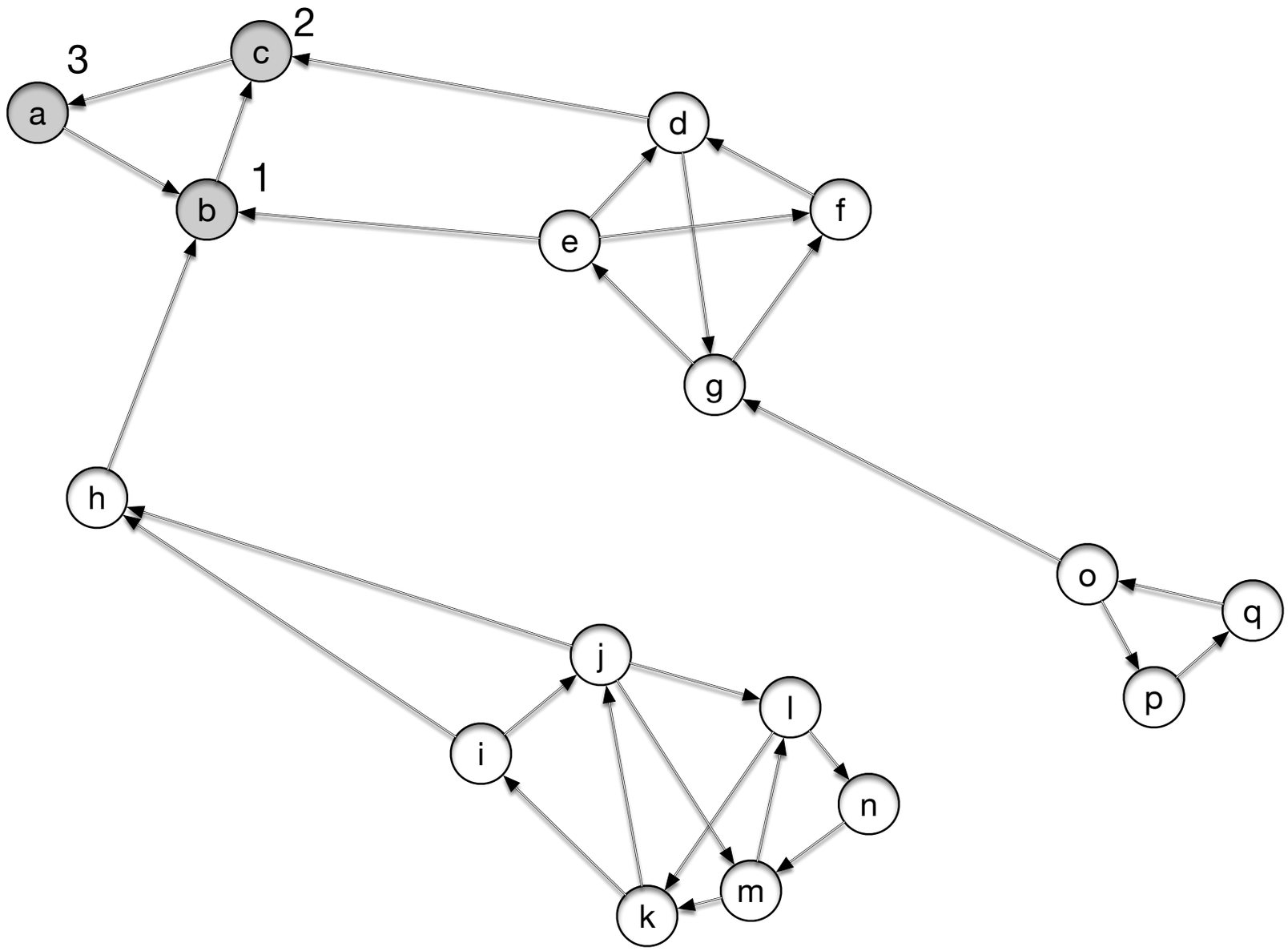
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



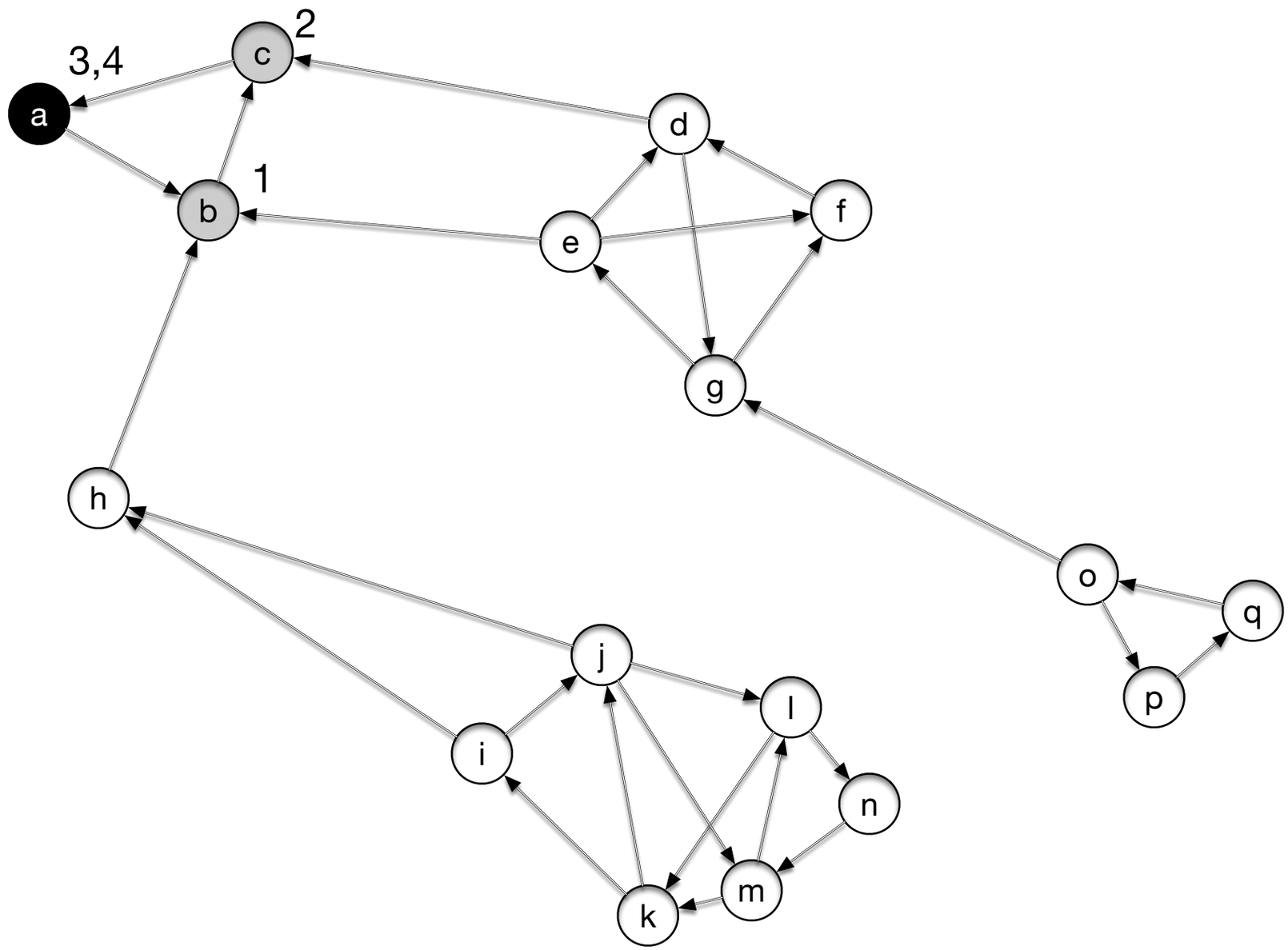
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



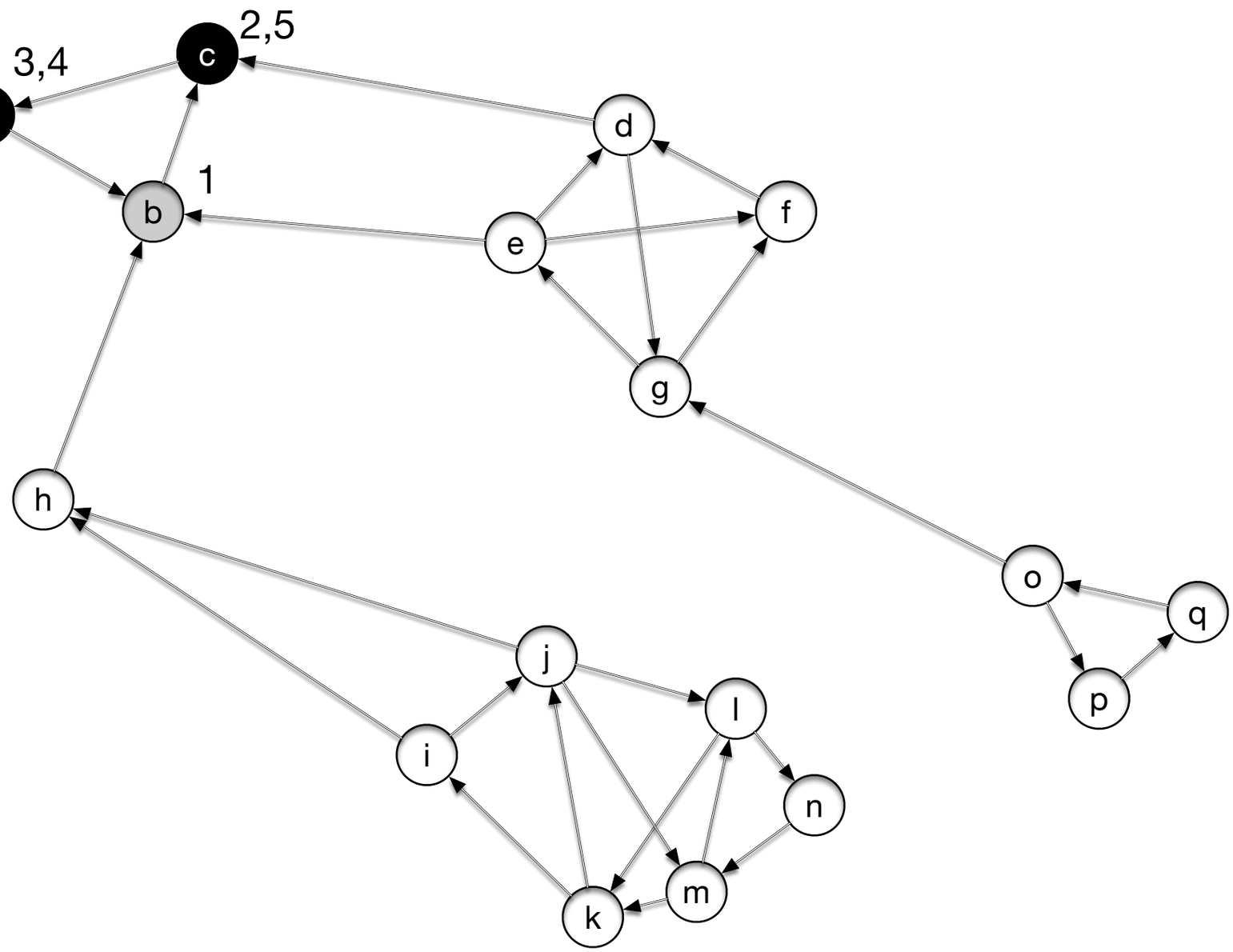
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



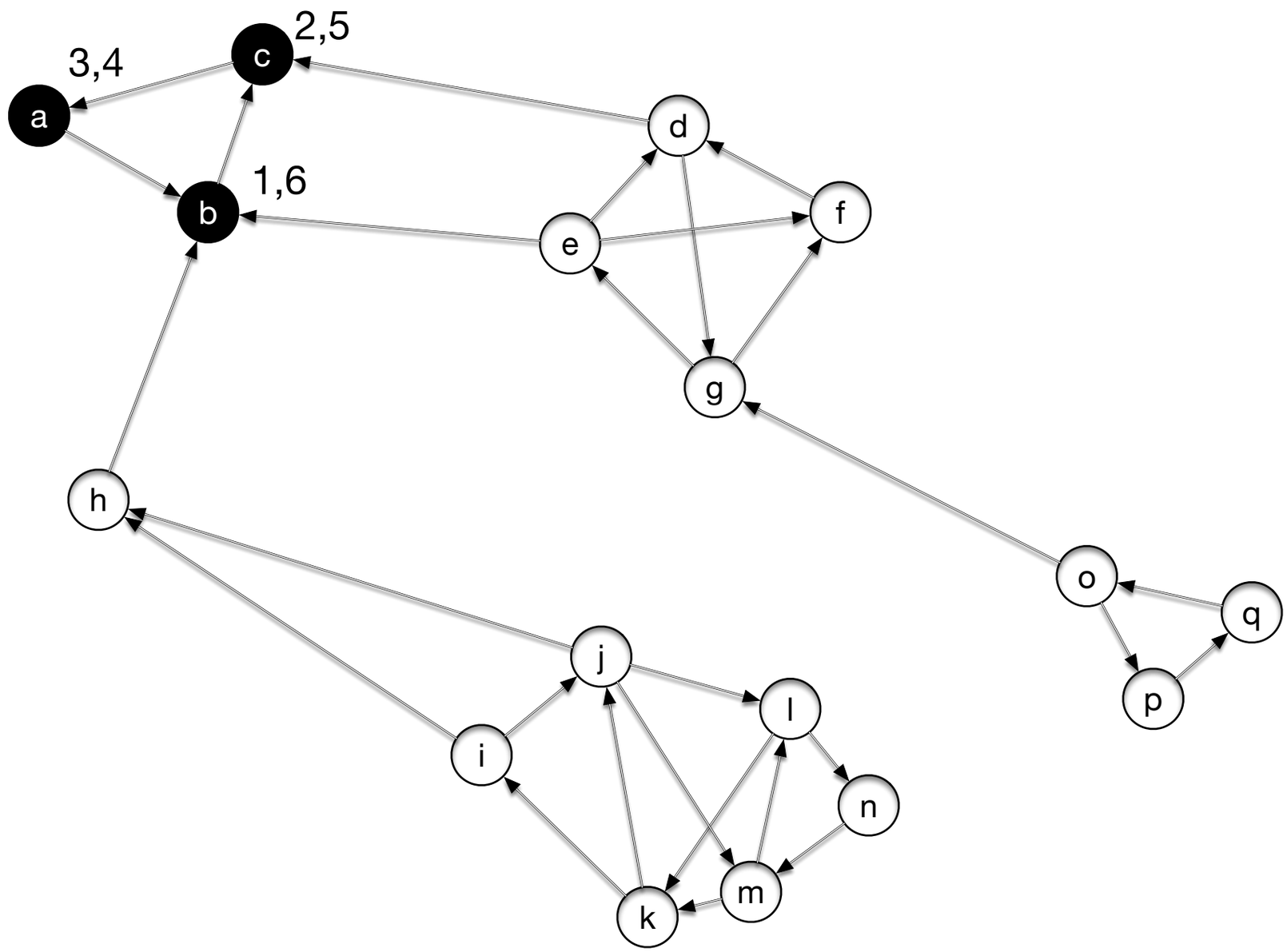
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7

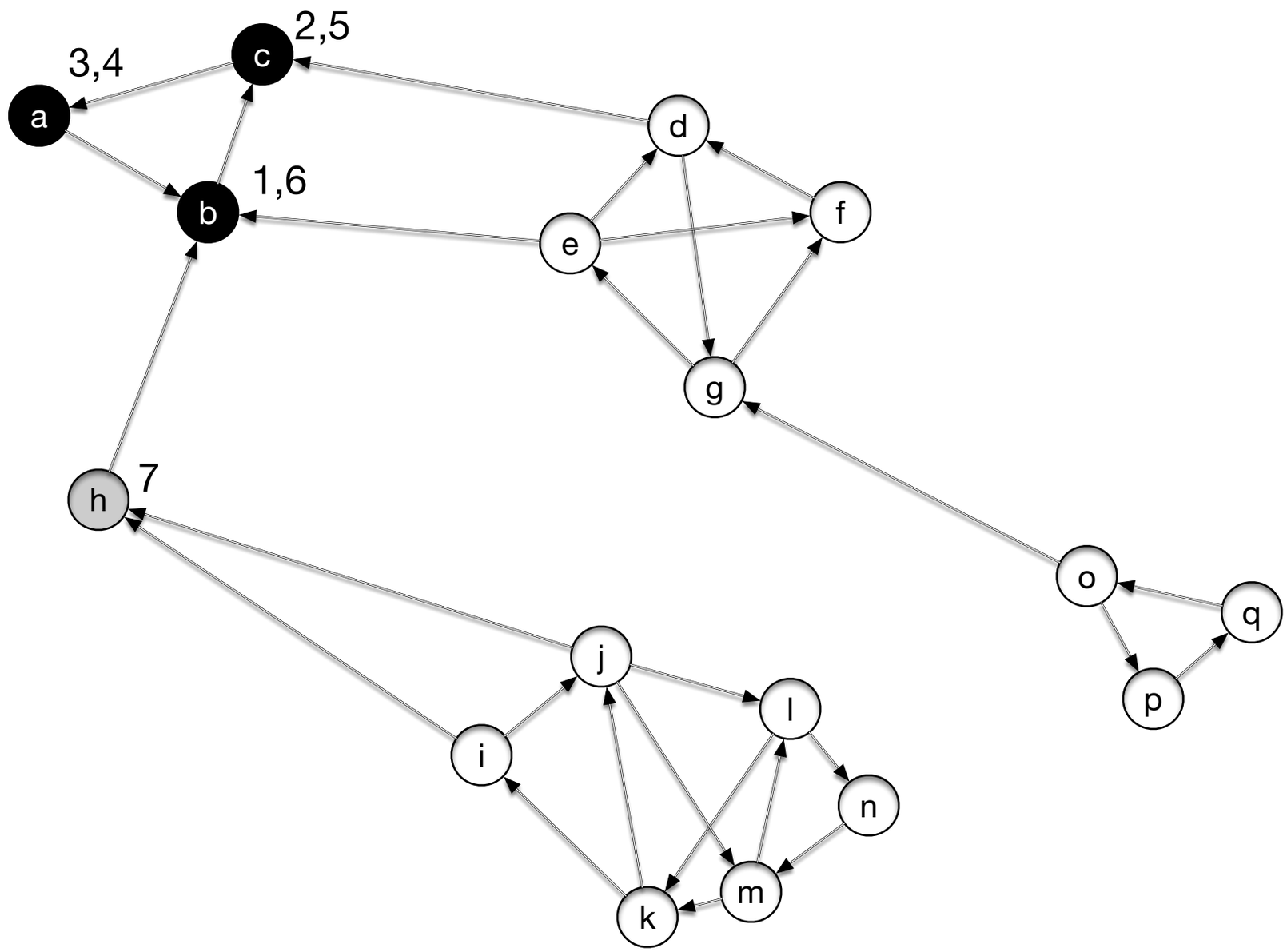


Strongly Connected Components

- We have finished visit in b
  - We can print out its descendants
    - $\{a, b, c\}$
    - This is the first connected component
    - We then select the white node with the highest finishing time from the previous run: h

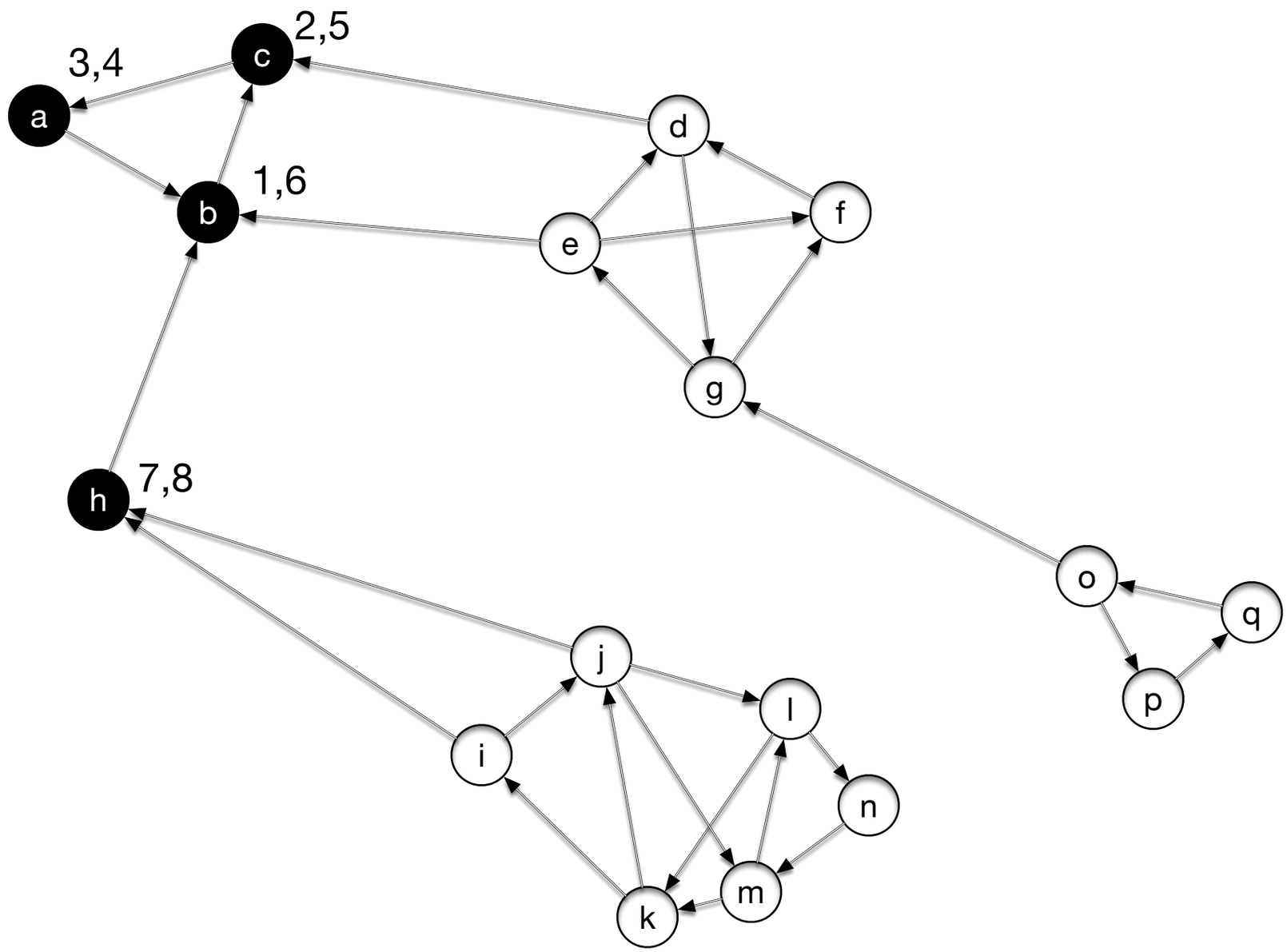
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7

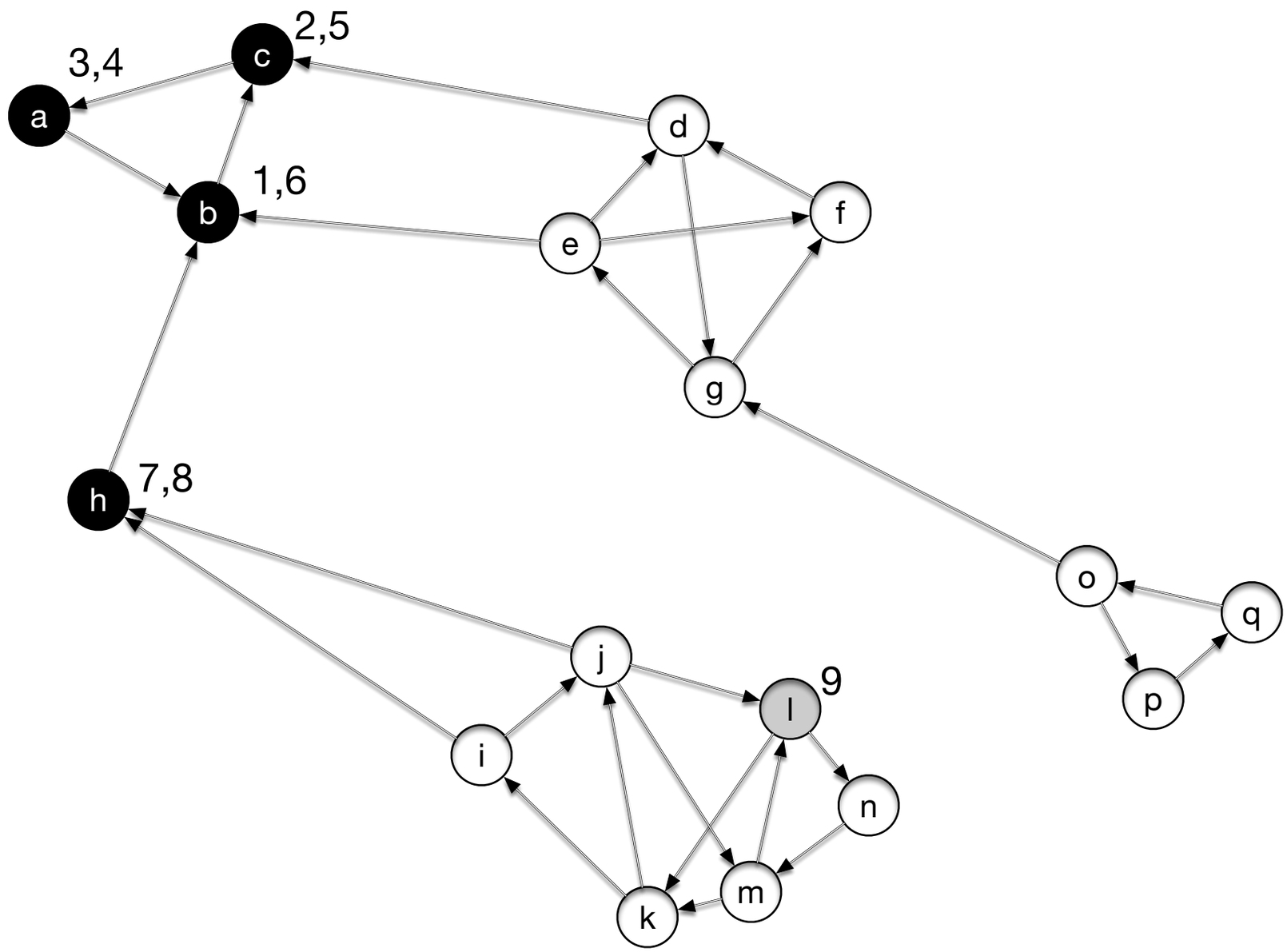


Strongly Connected Components

- Again, we print out the descendants of h and now have
  - $\{a, b, c\}, \{h\}$
- We then select the vertex with the next highest finishing time

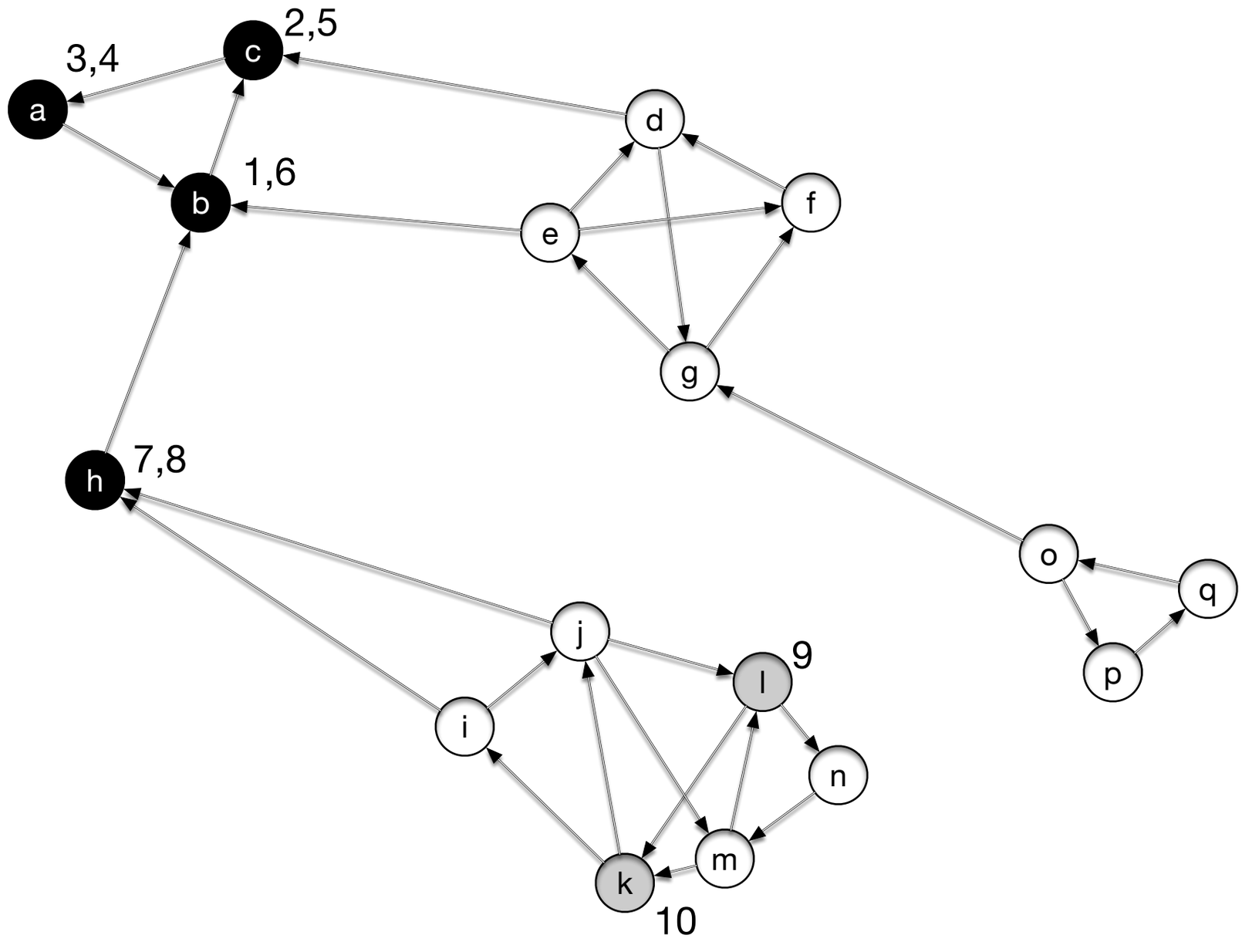
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



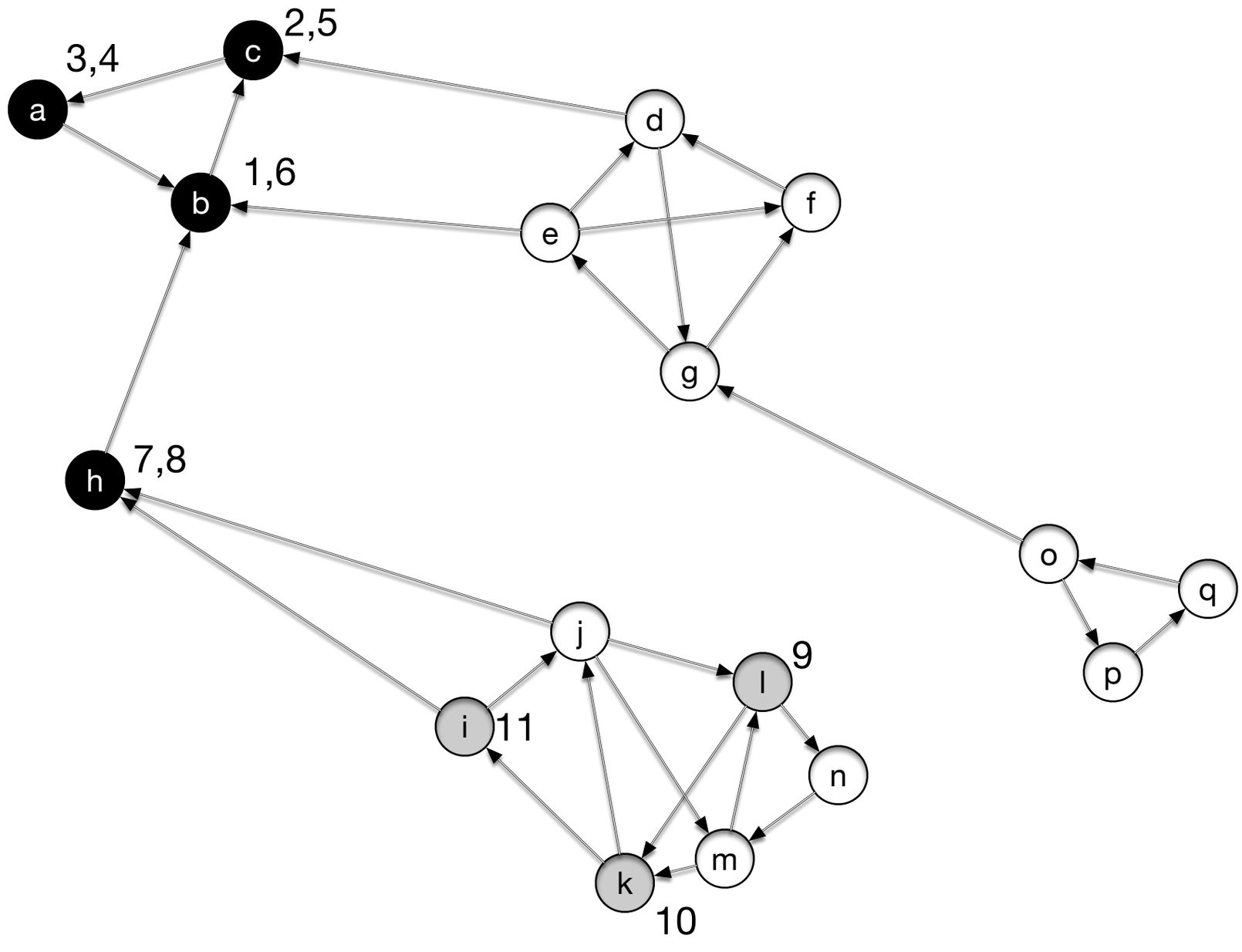
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



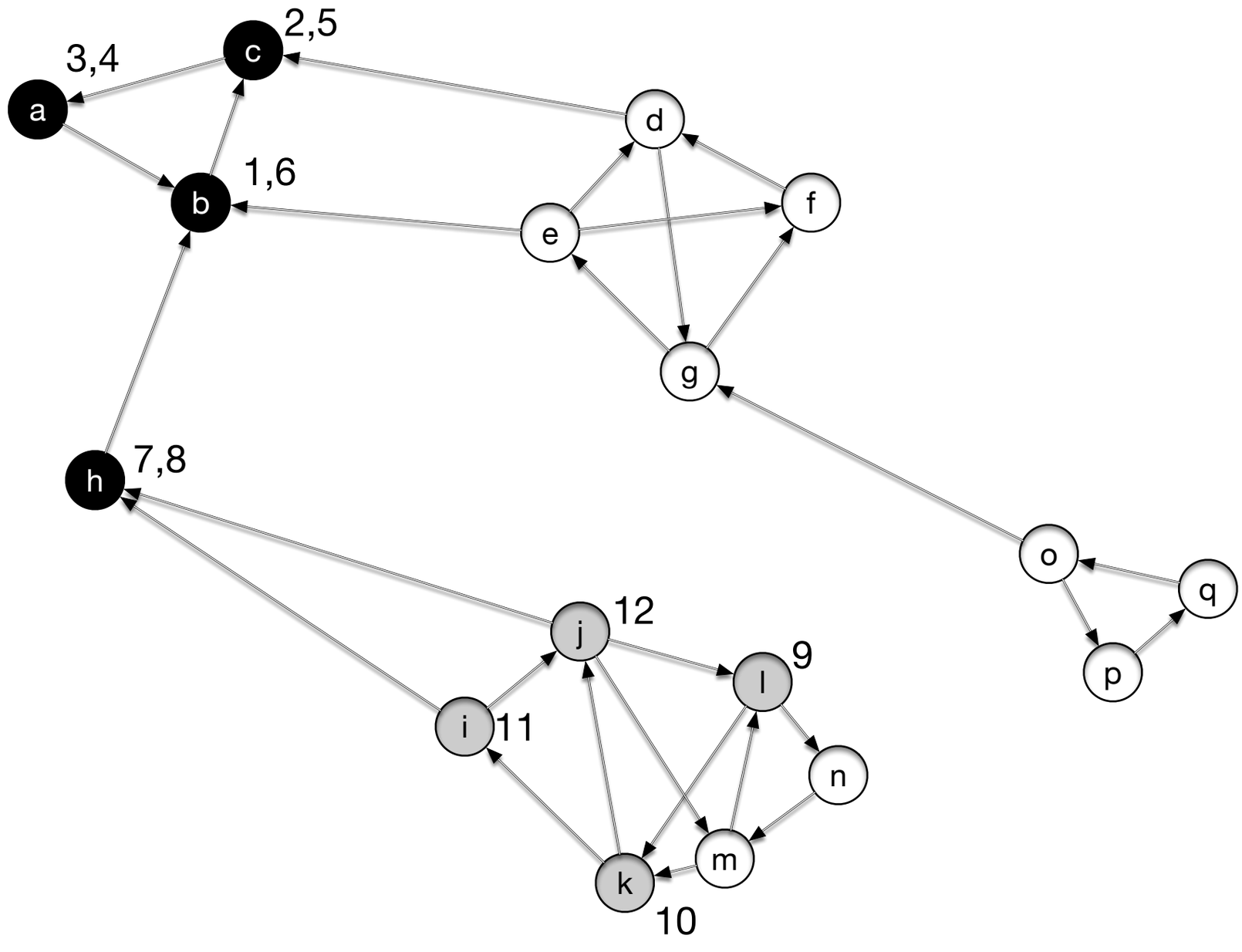
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



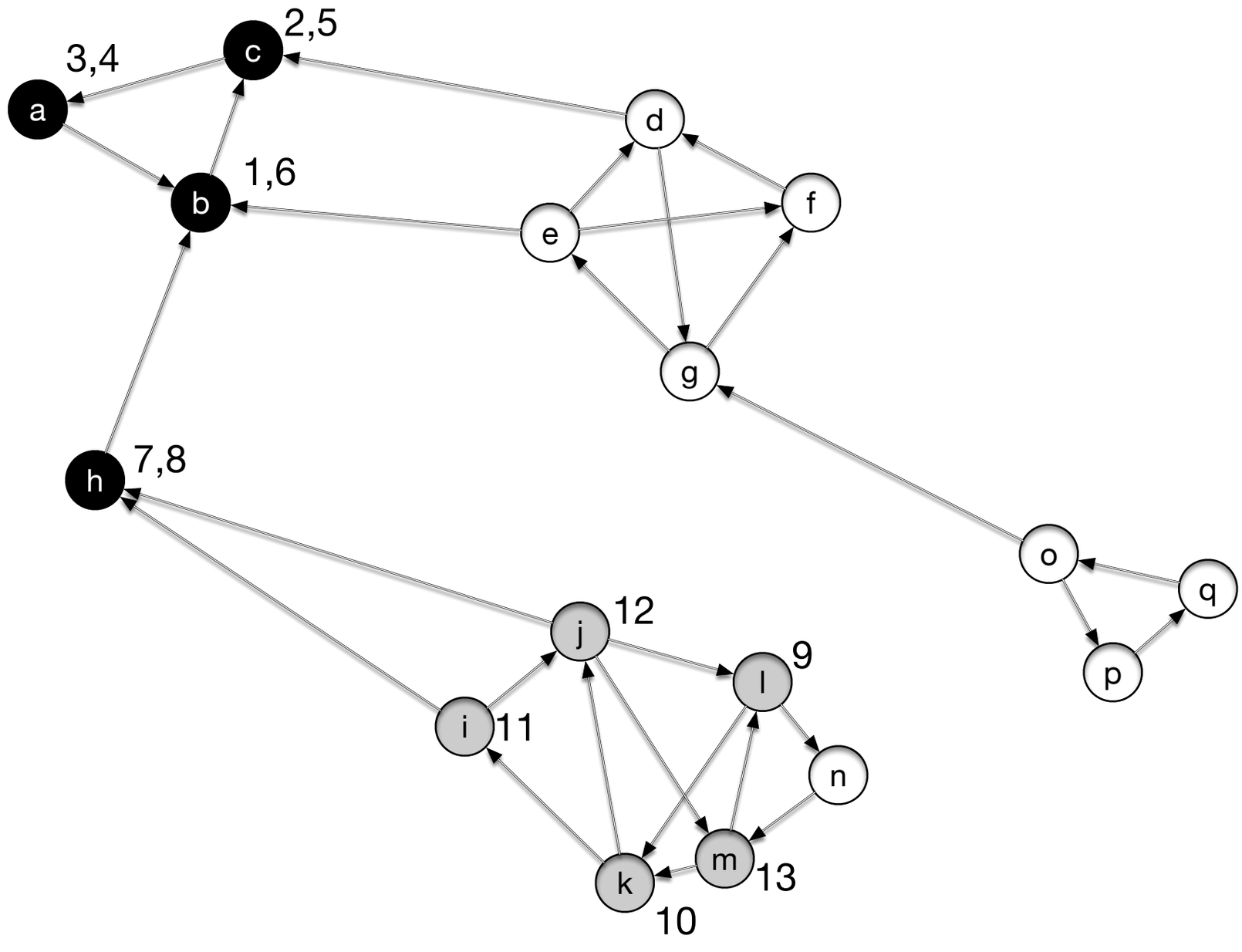
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



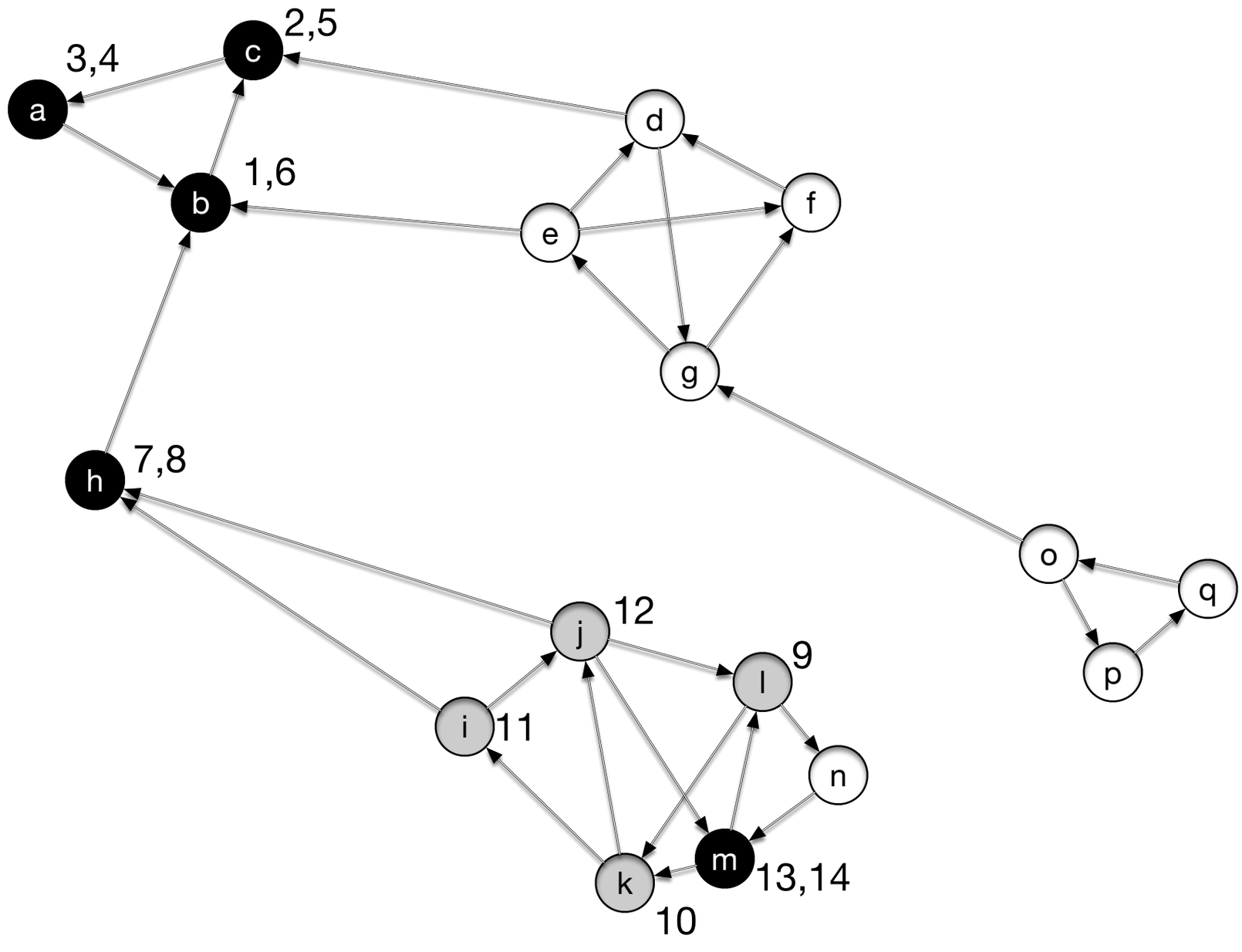
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



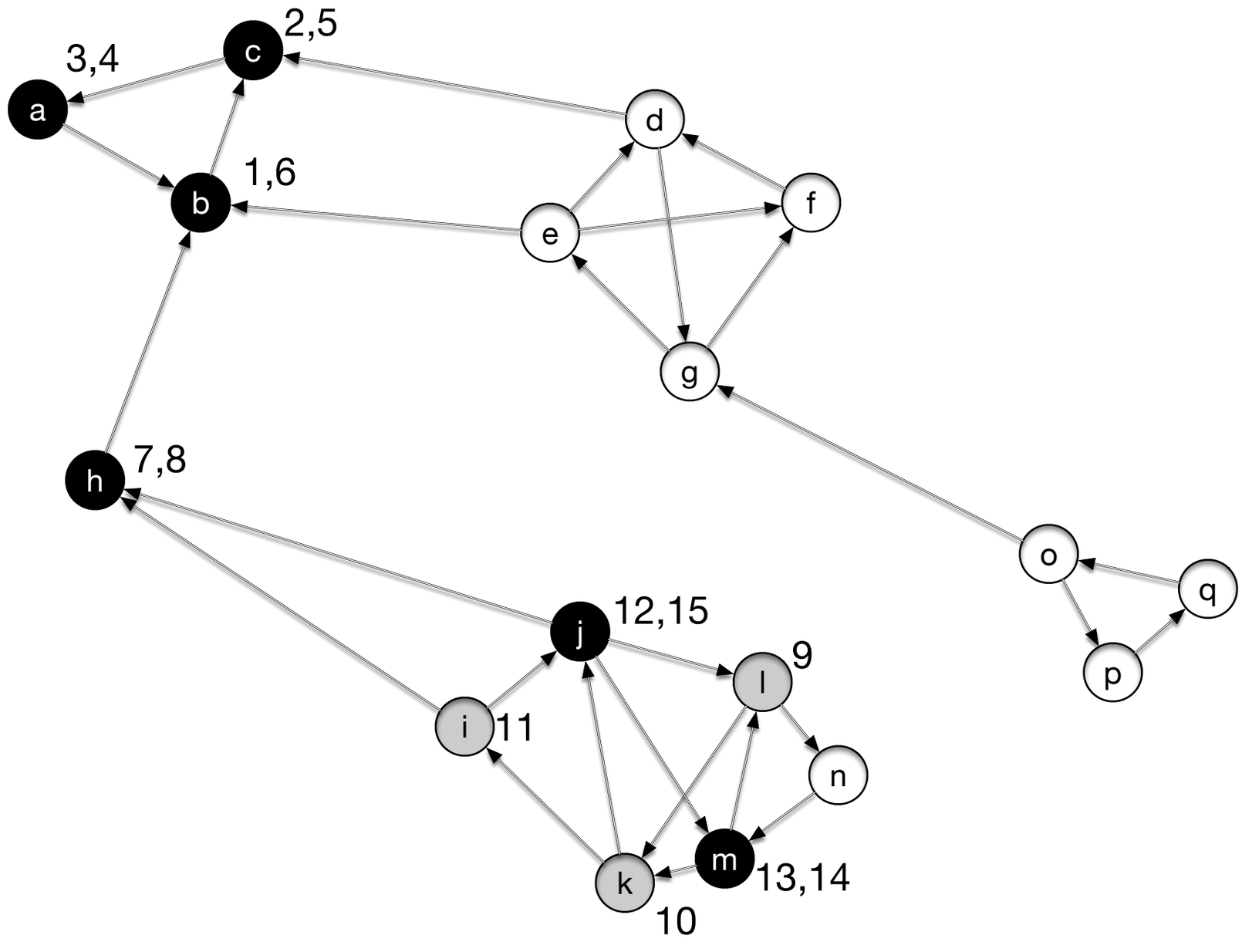
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



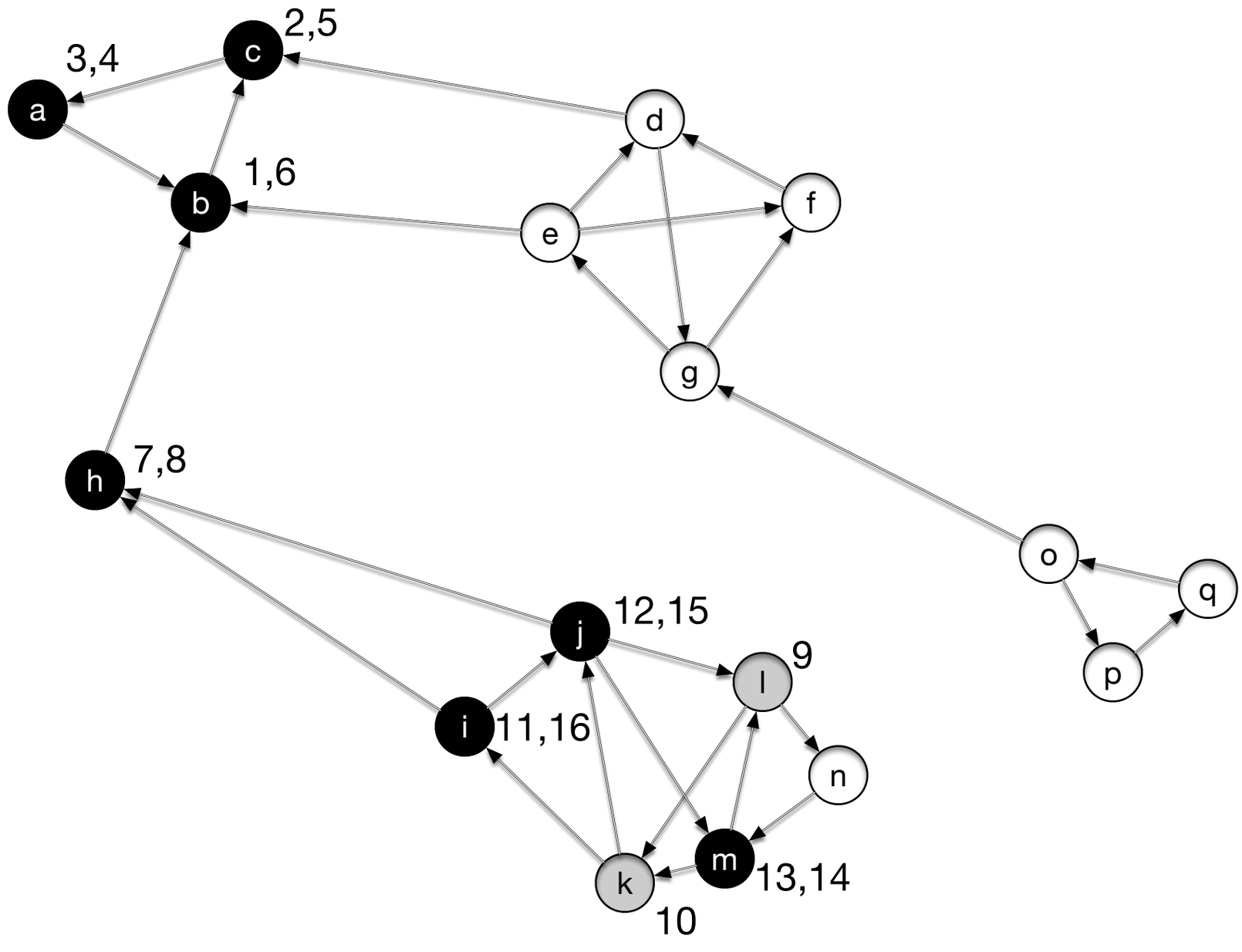
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



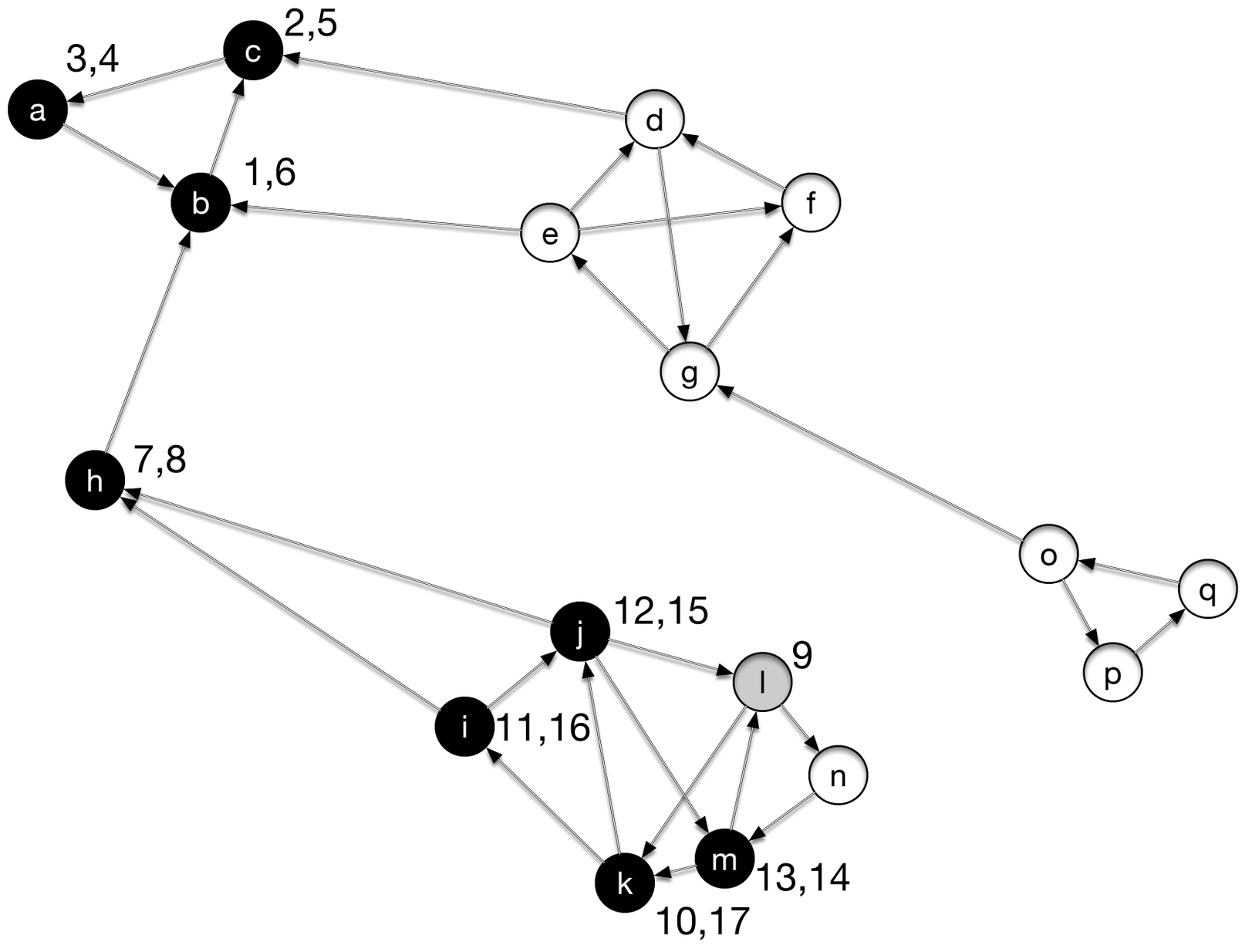
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



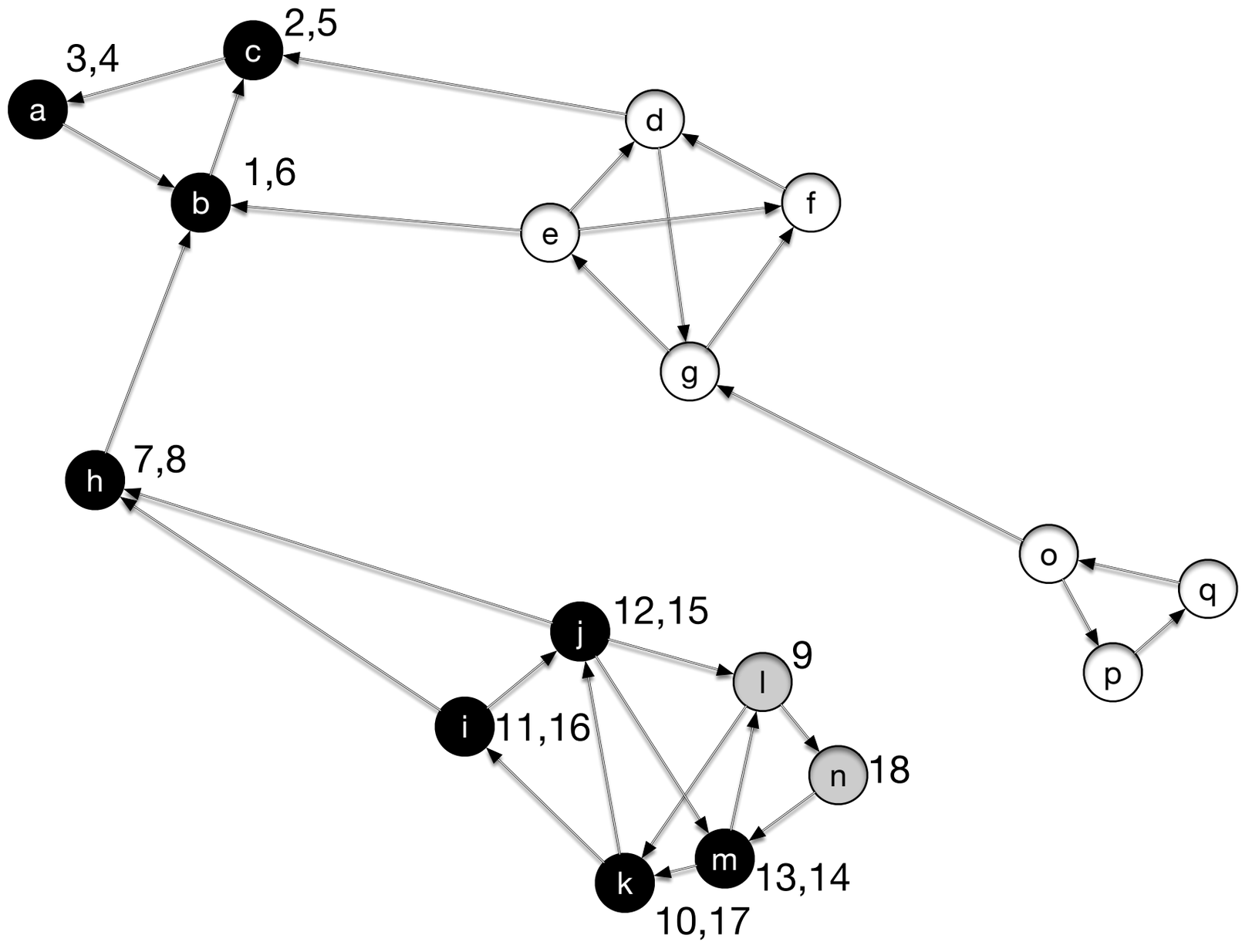
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



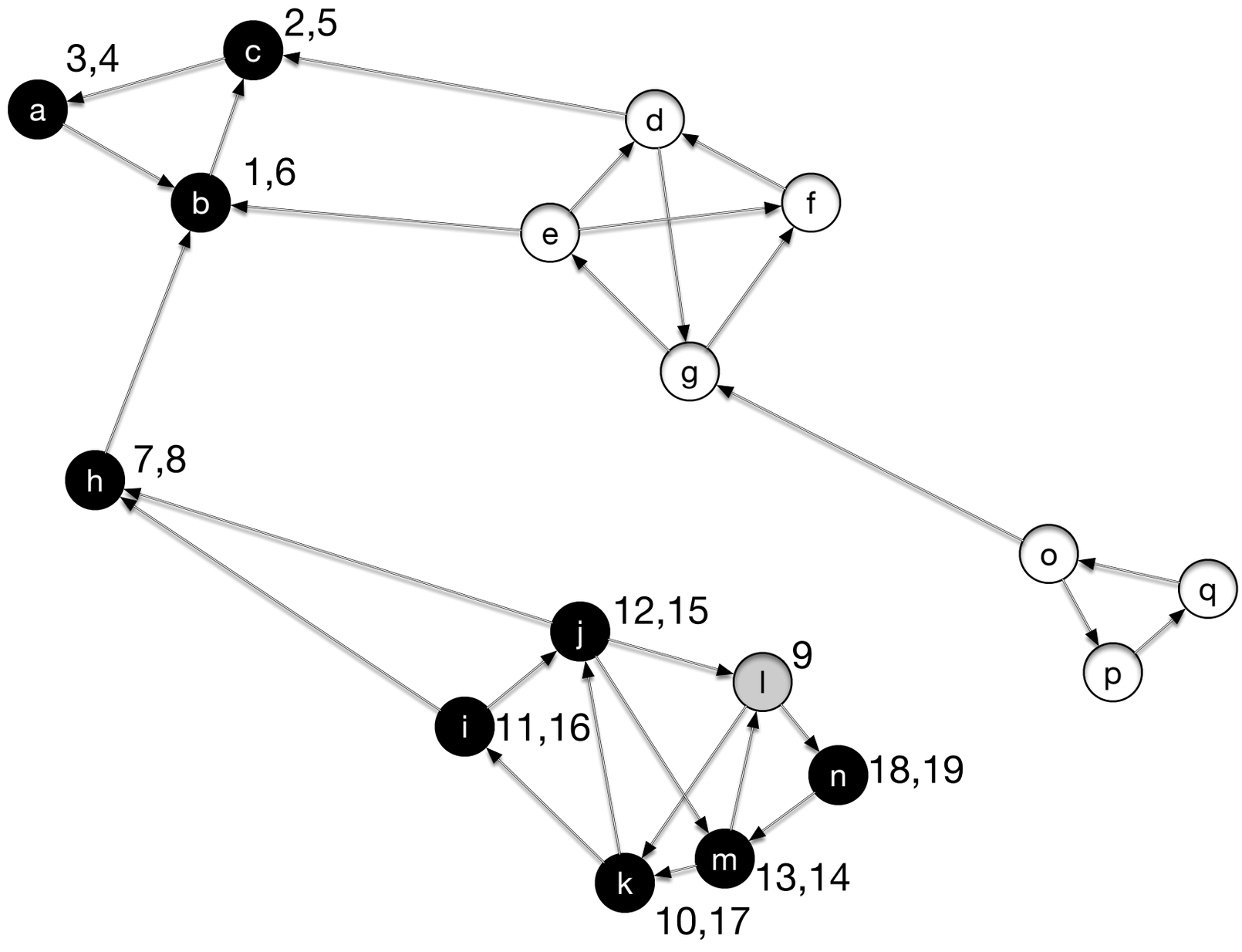
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



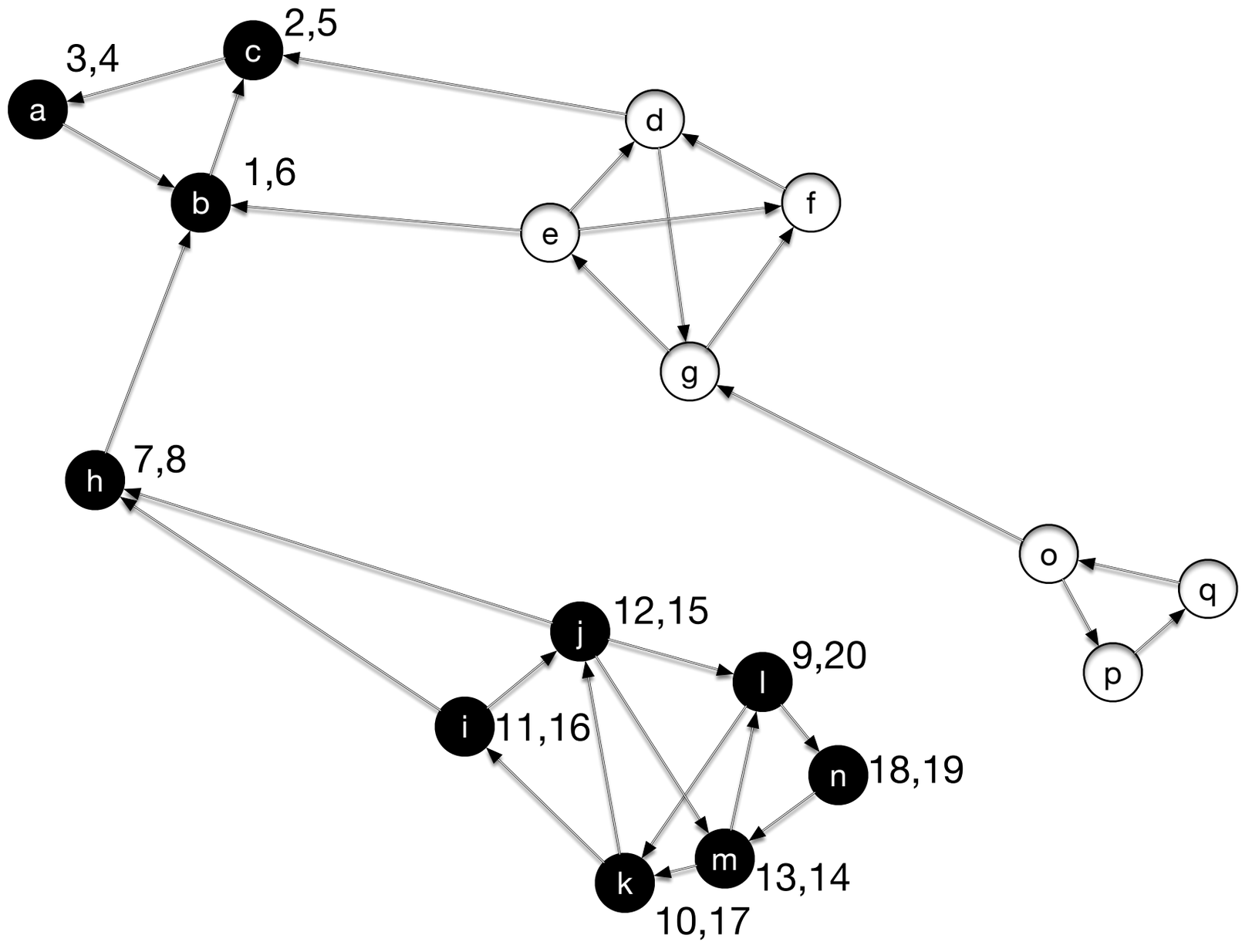
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7

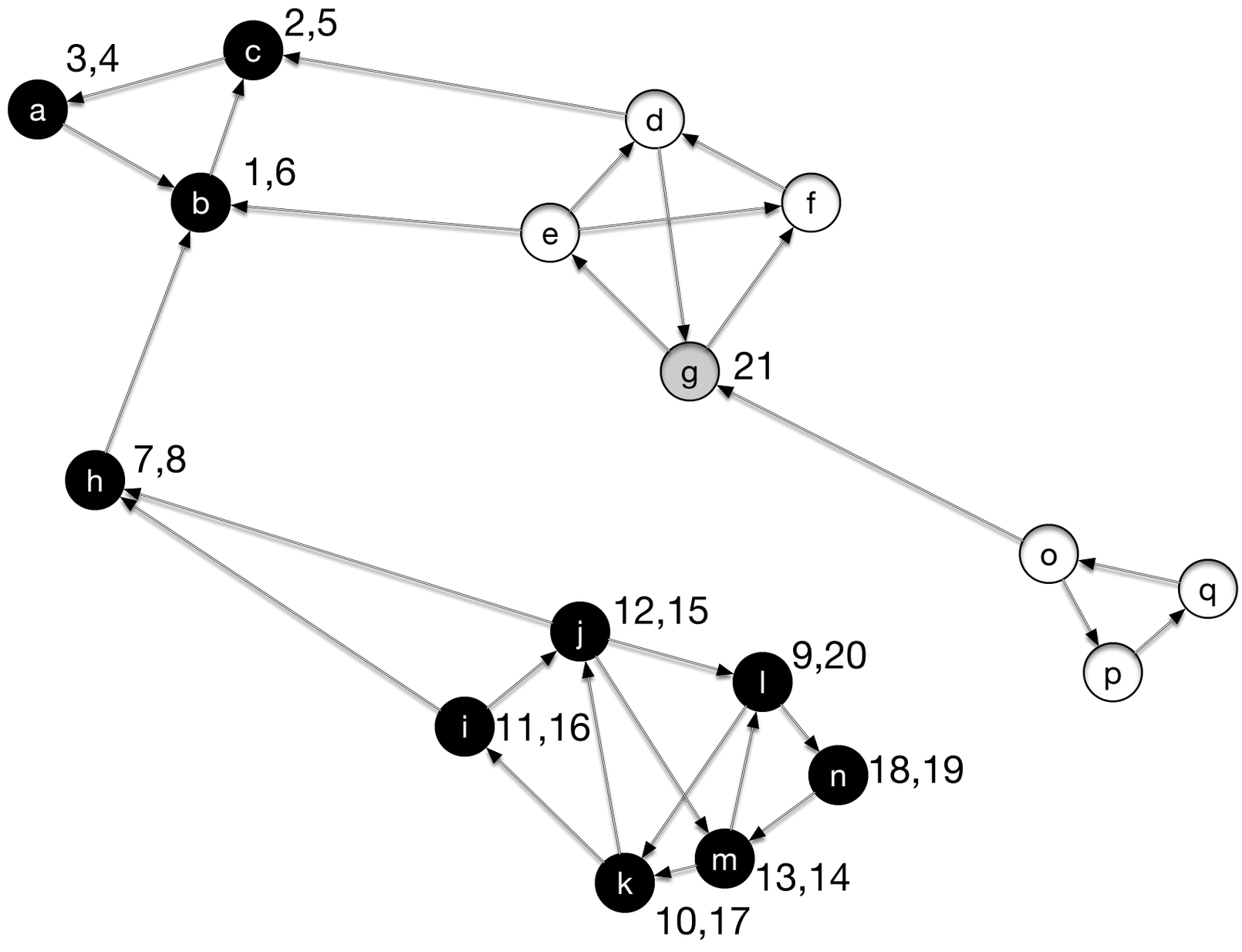


Strongly Connected Components

- We admit the new tree as a SCC
  - $\{a, b, c\}$, $\{h\}$, $\{l, i, k, j, m, n\}$
  - Now we go with g

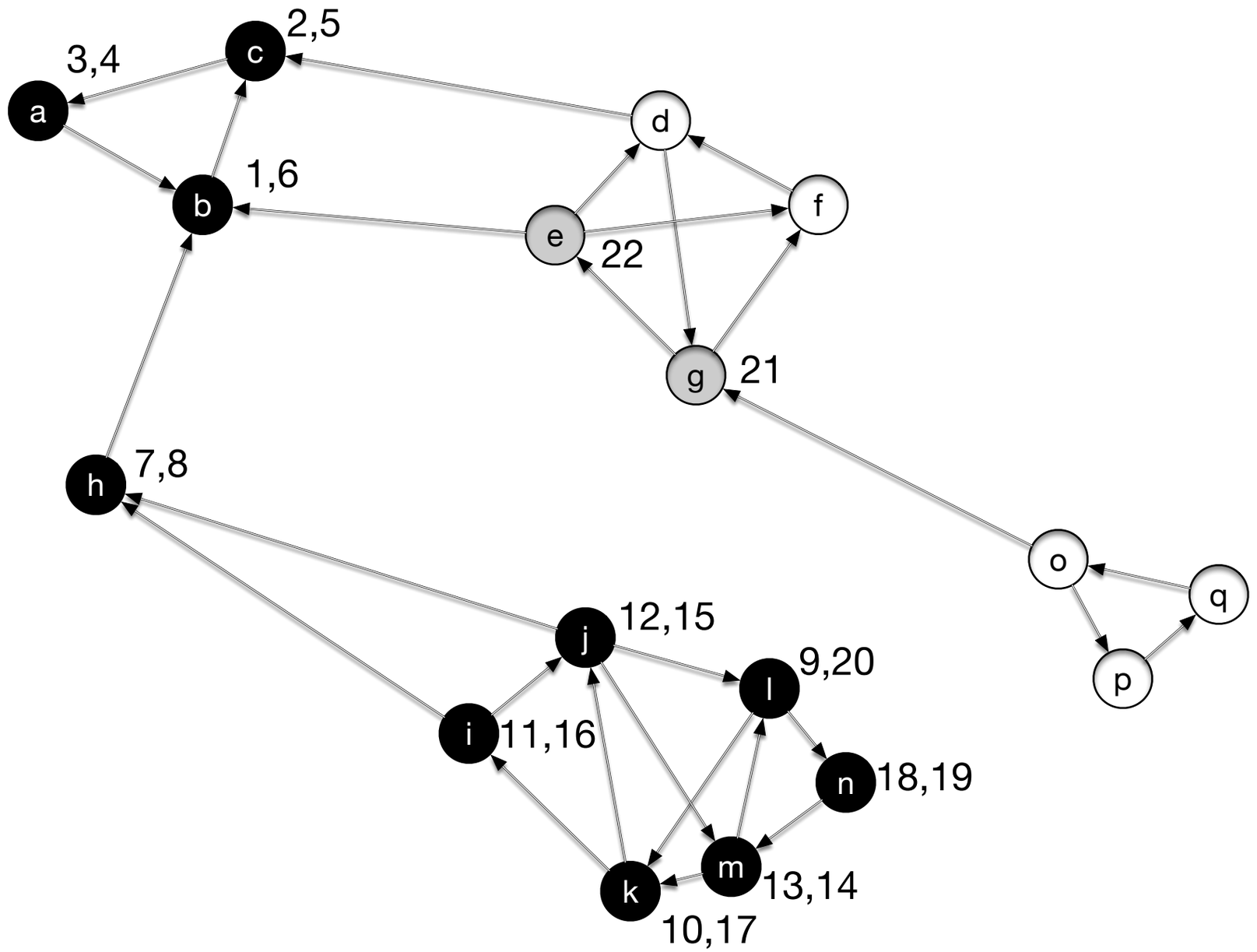
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



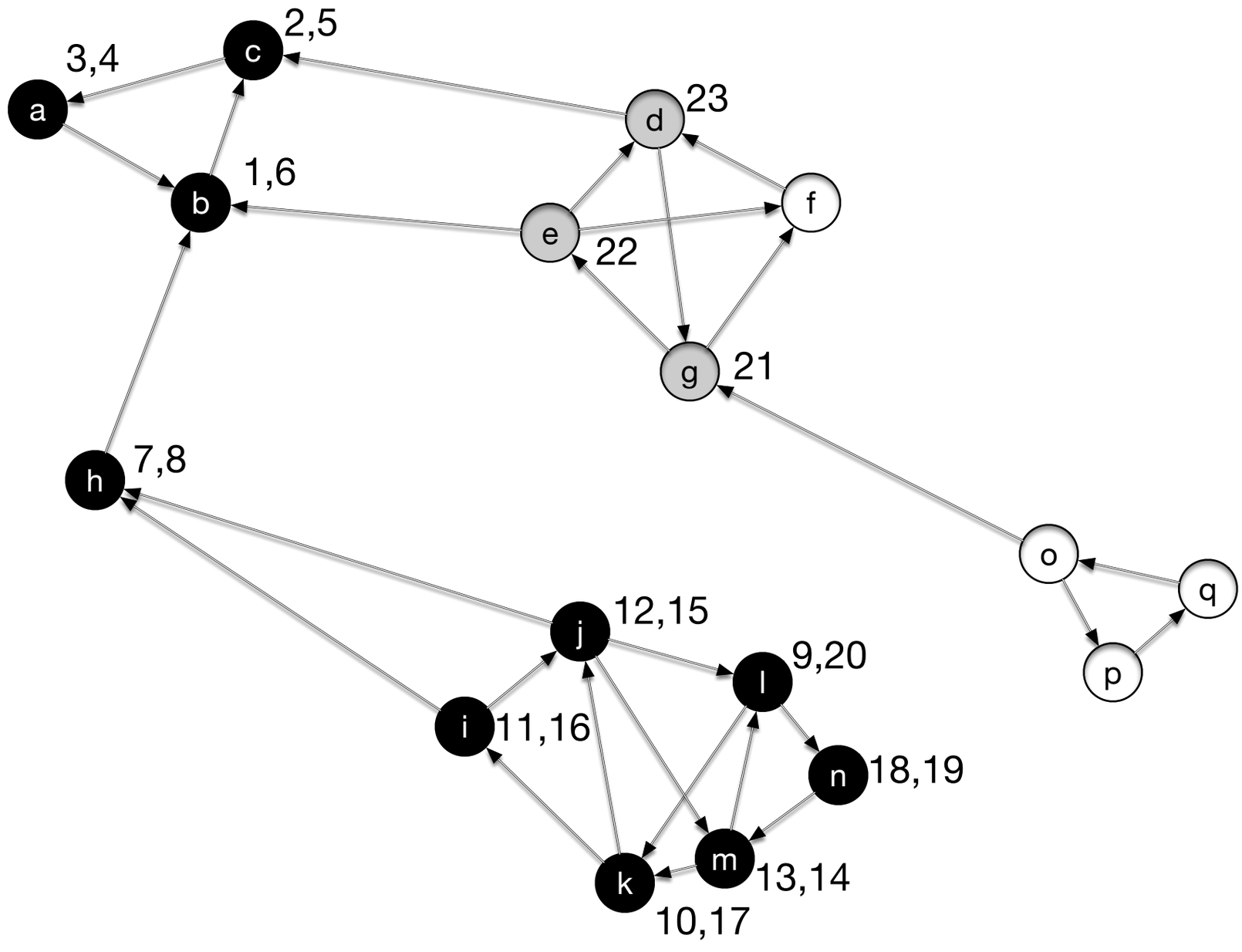
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



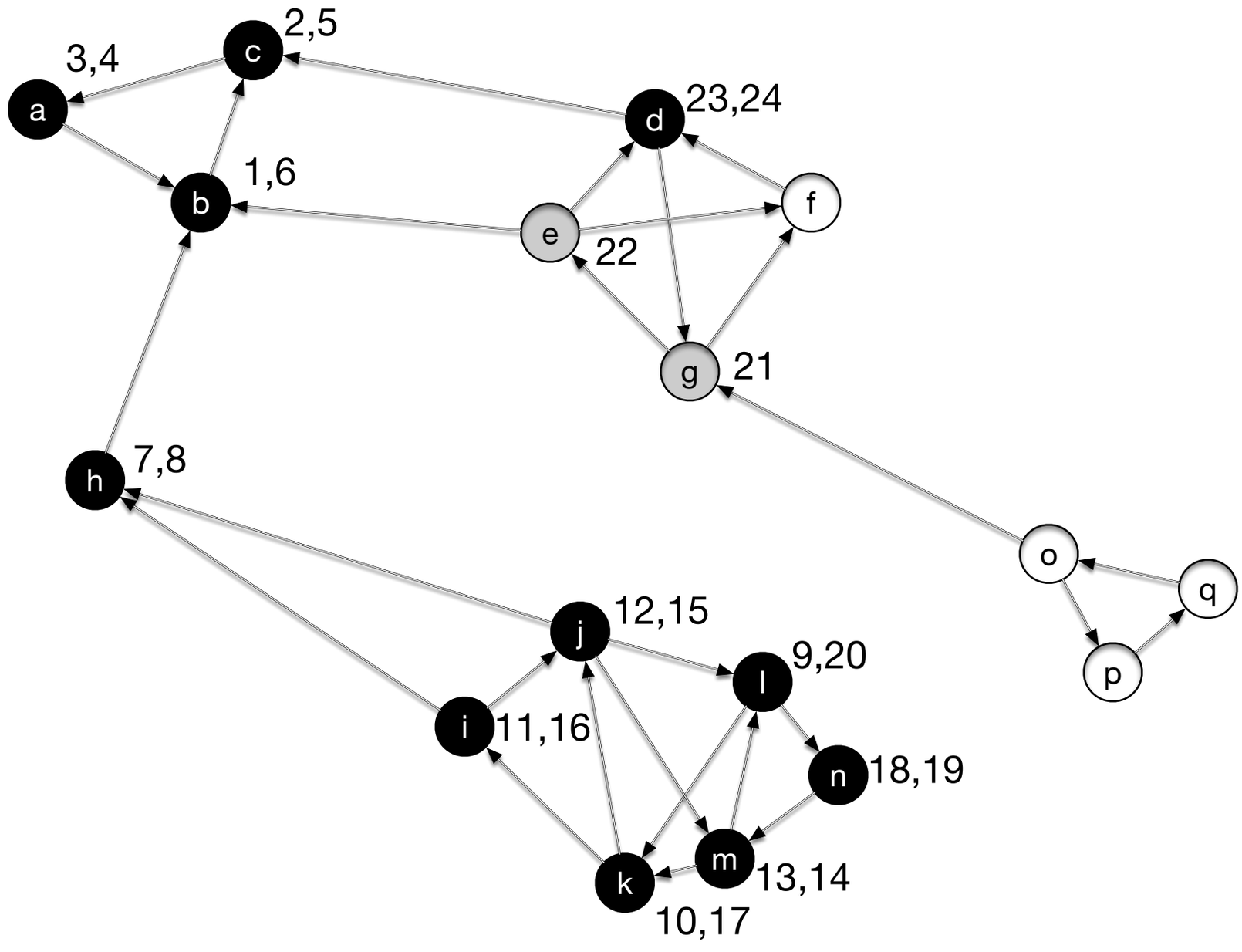
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



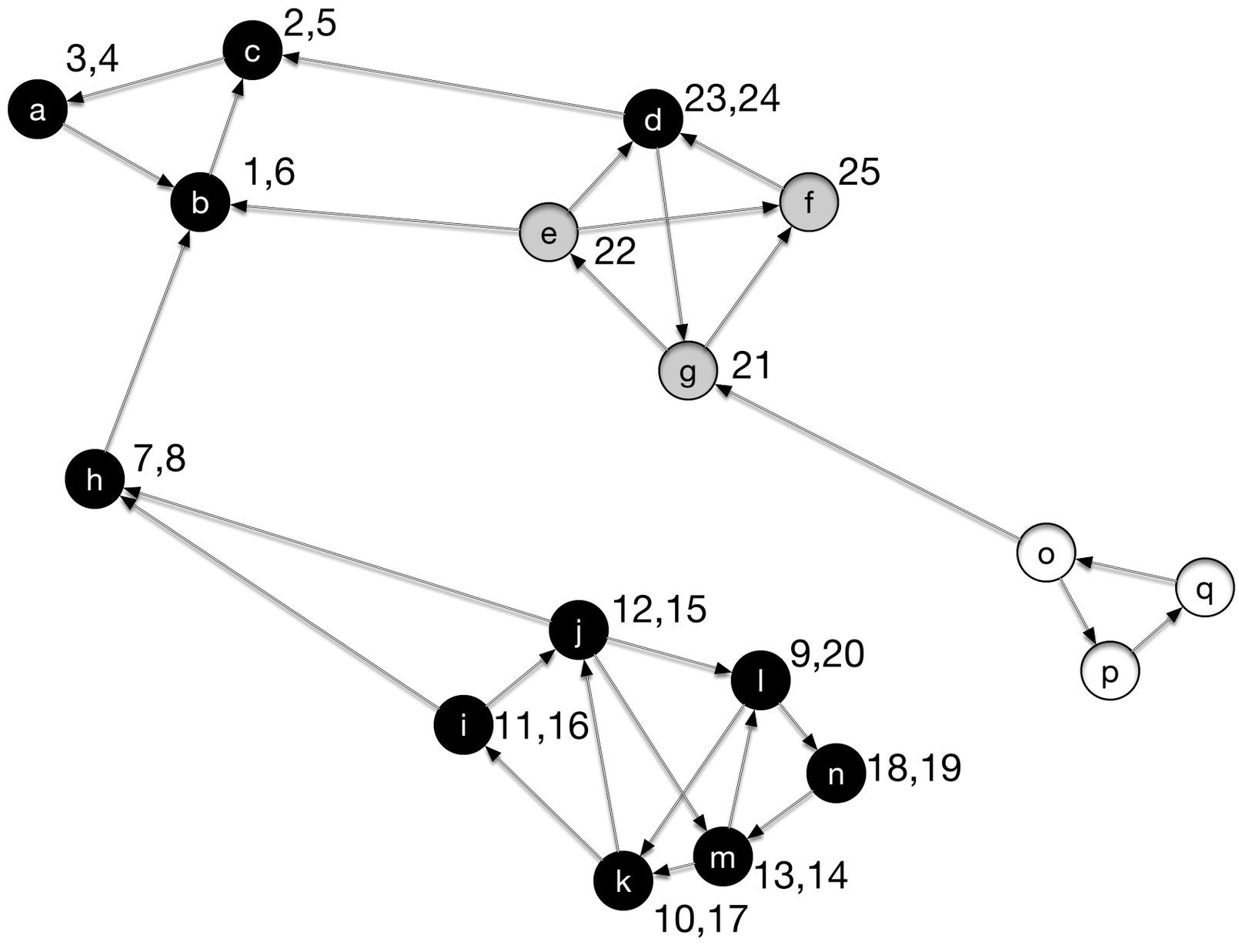
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



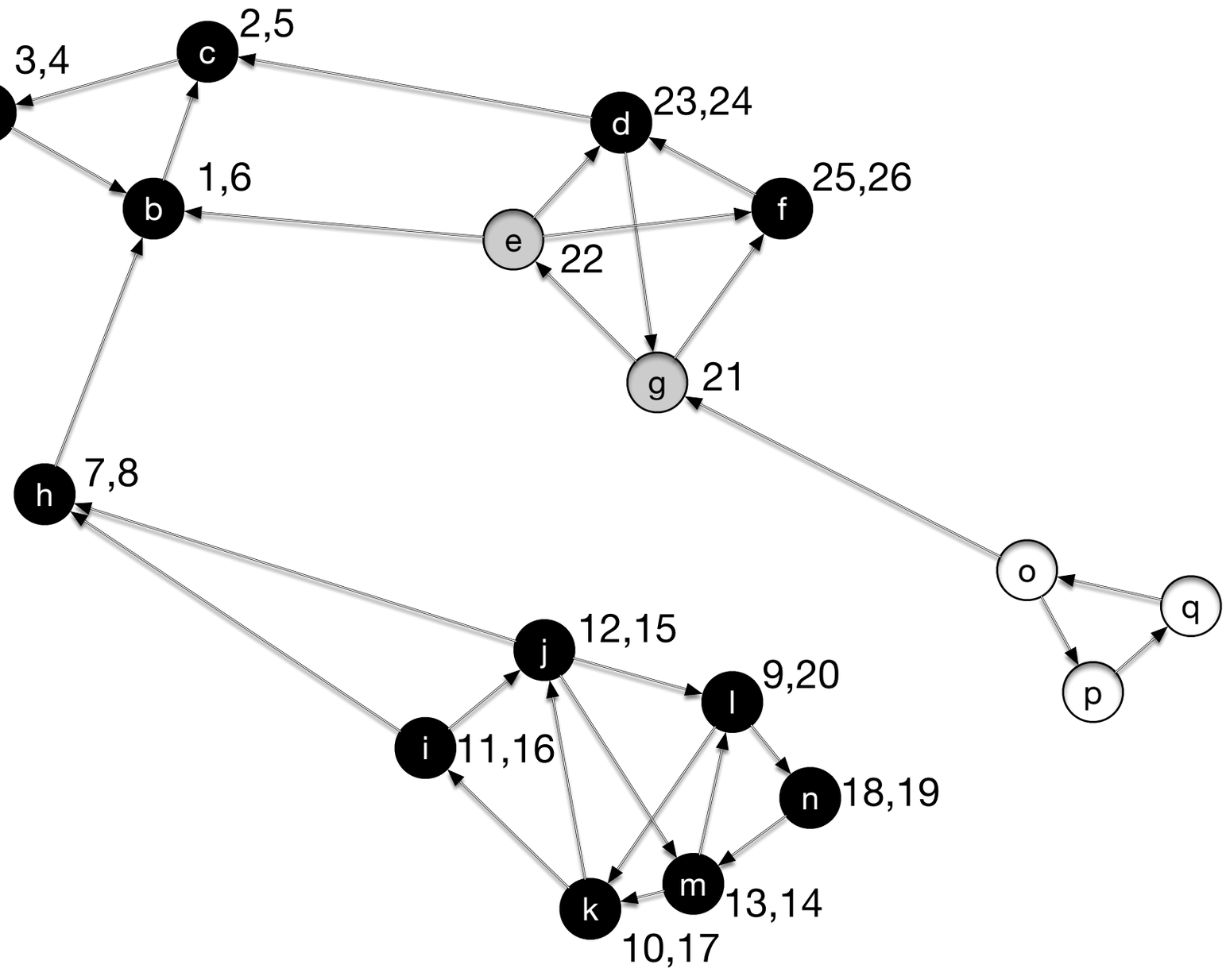
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



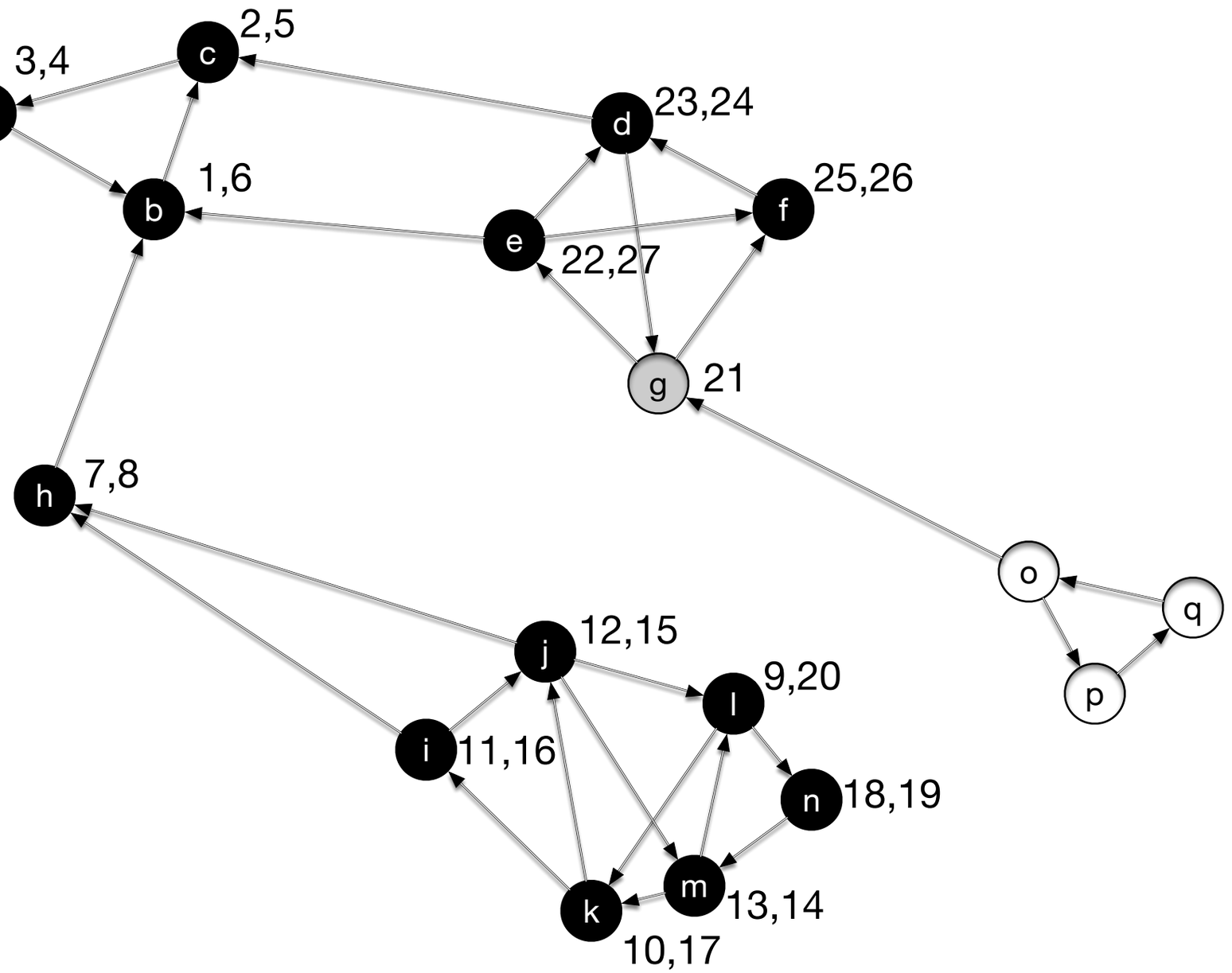
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



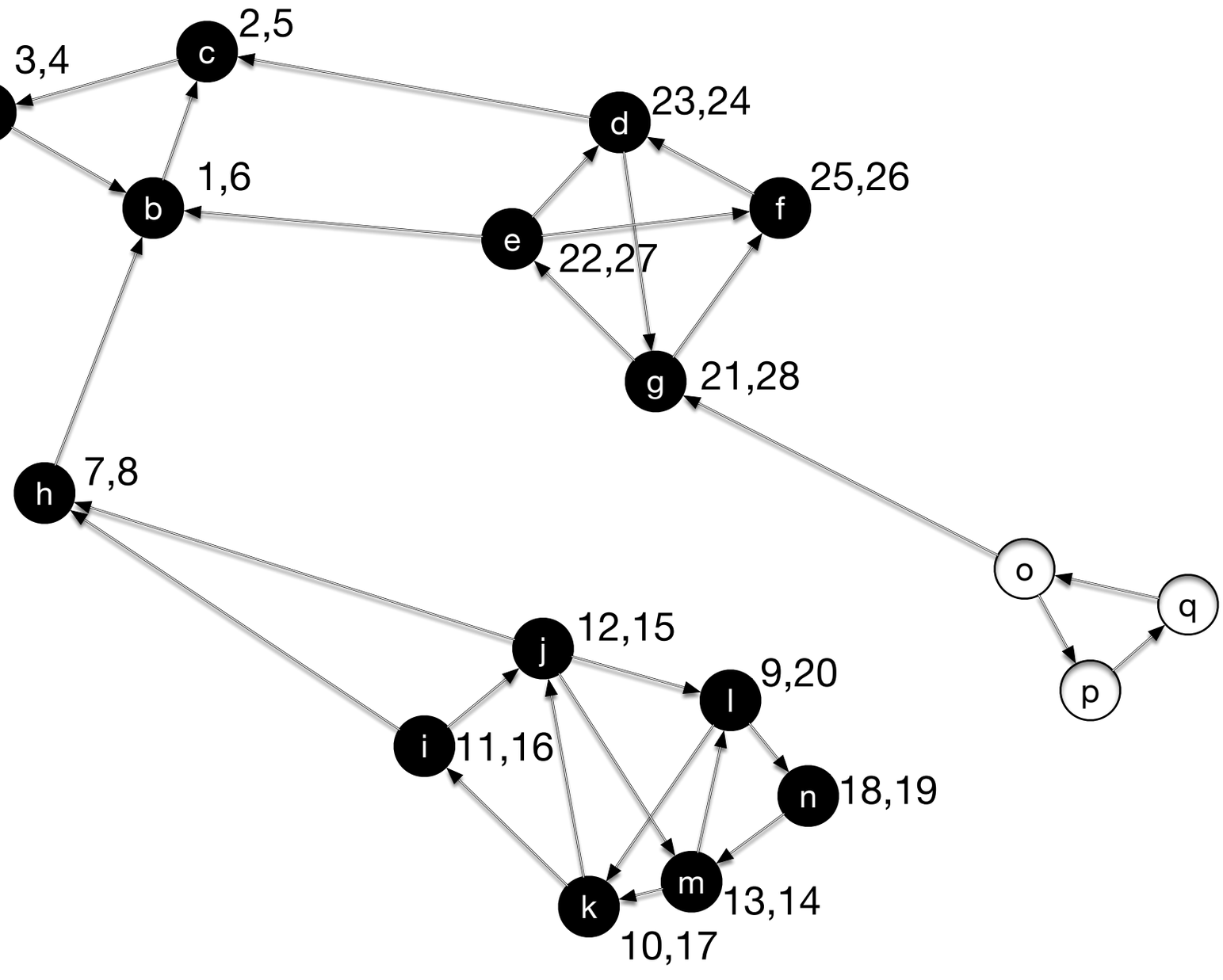
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7

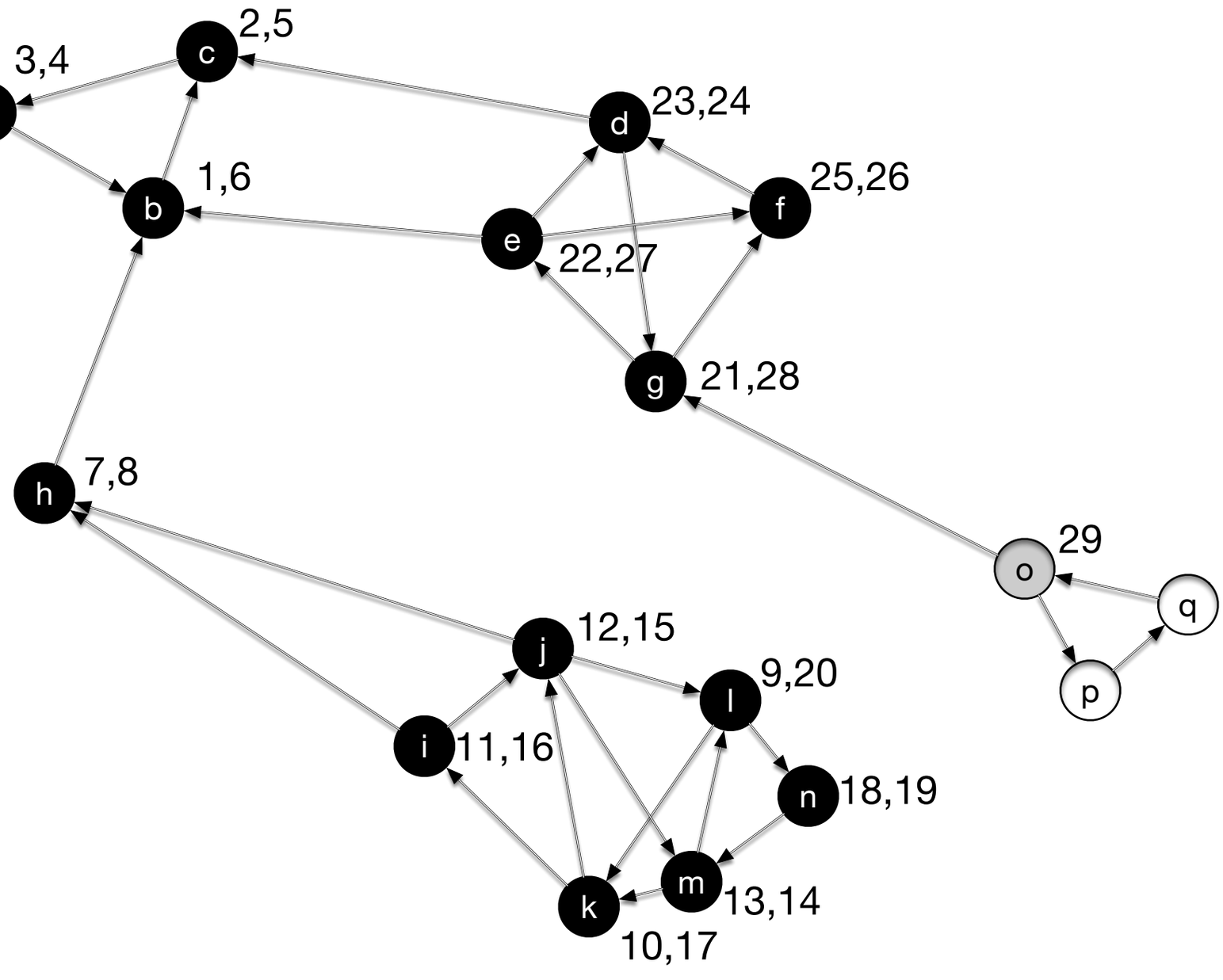


Strongly Connected Components

- We add $\{g, d, e, f\}$ to the list of SCC

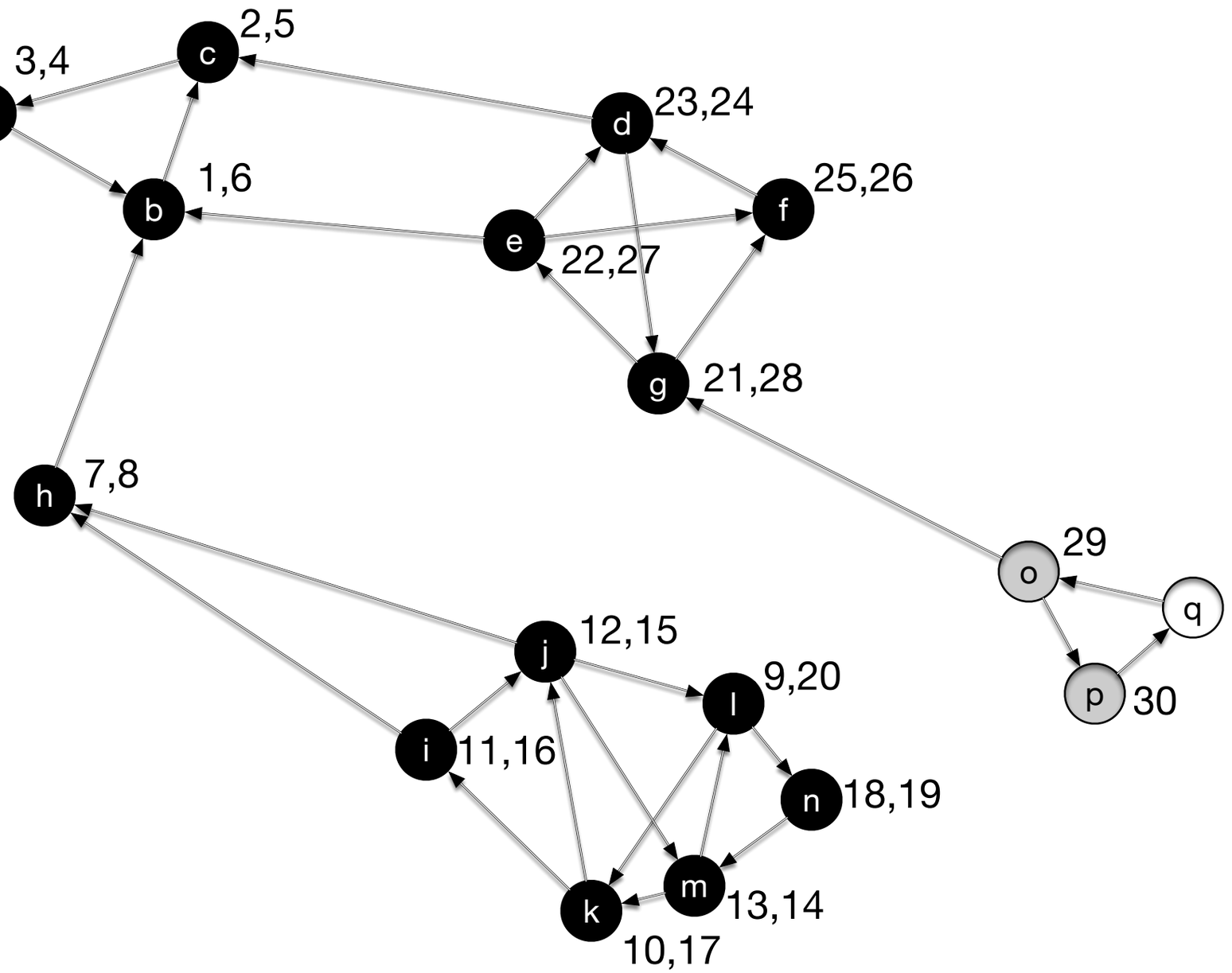
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



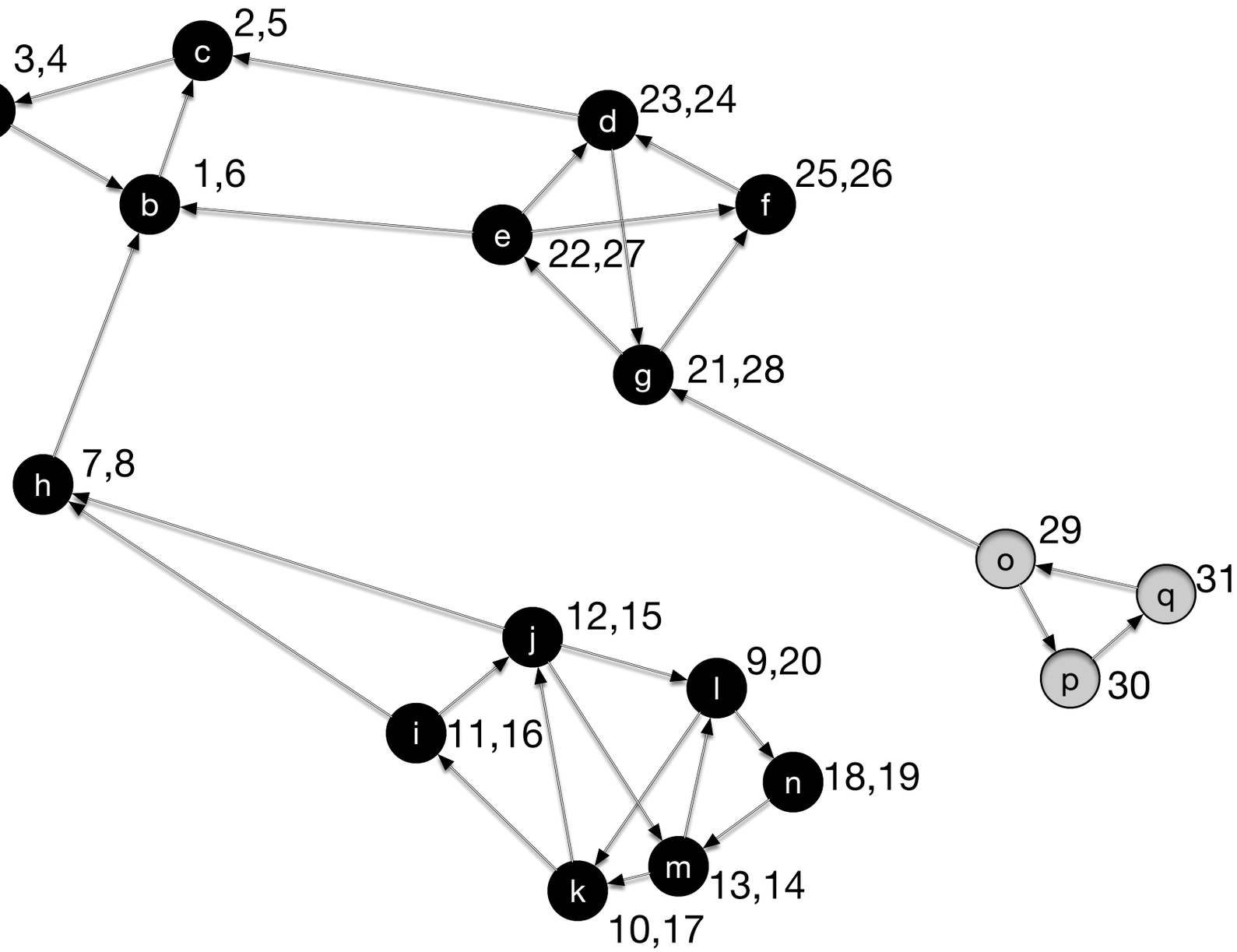
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



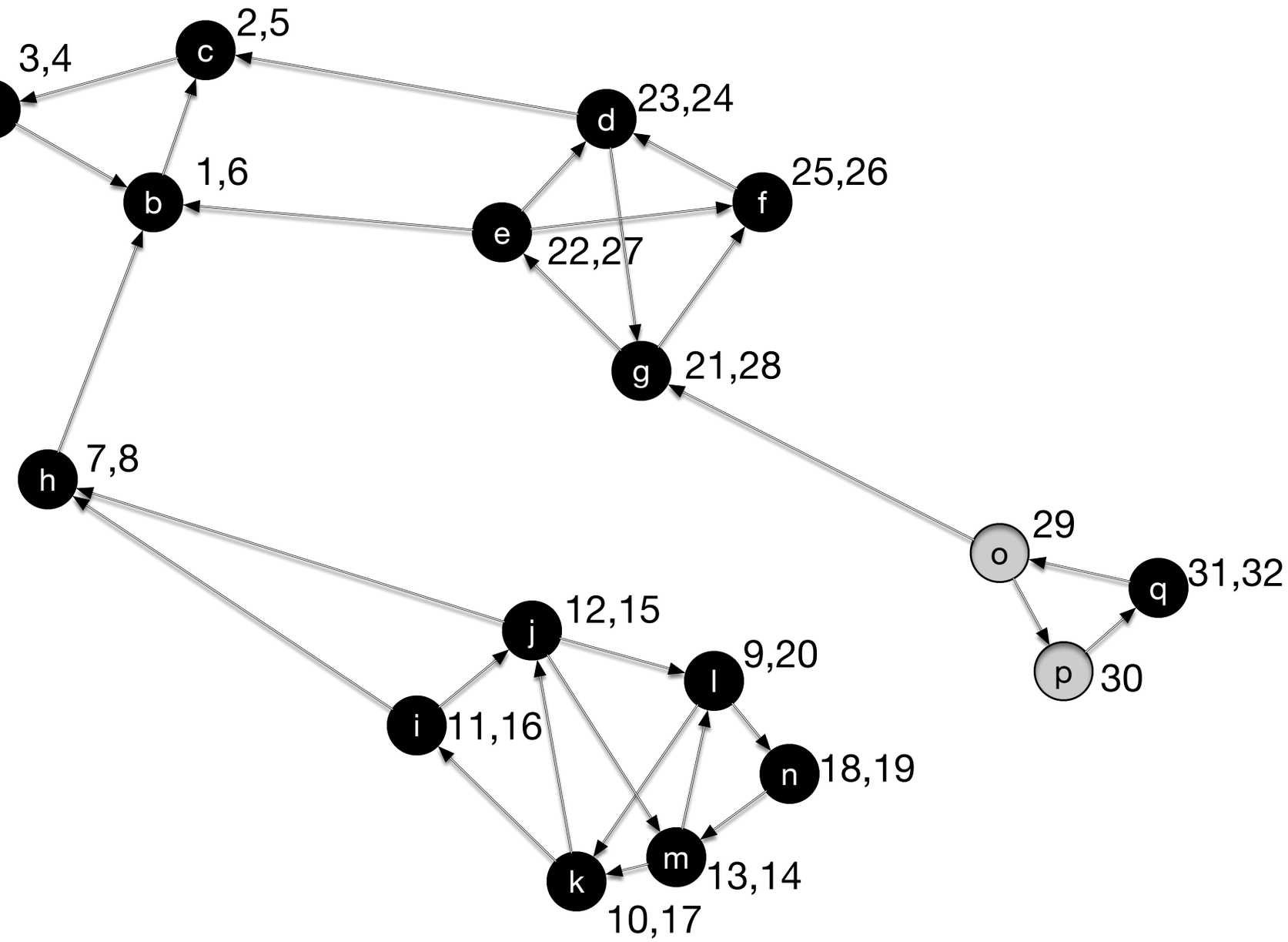
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



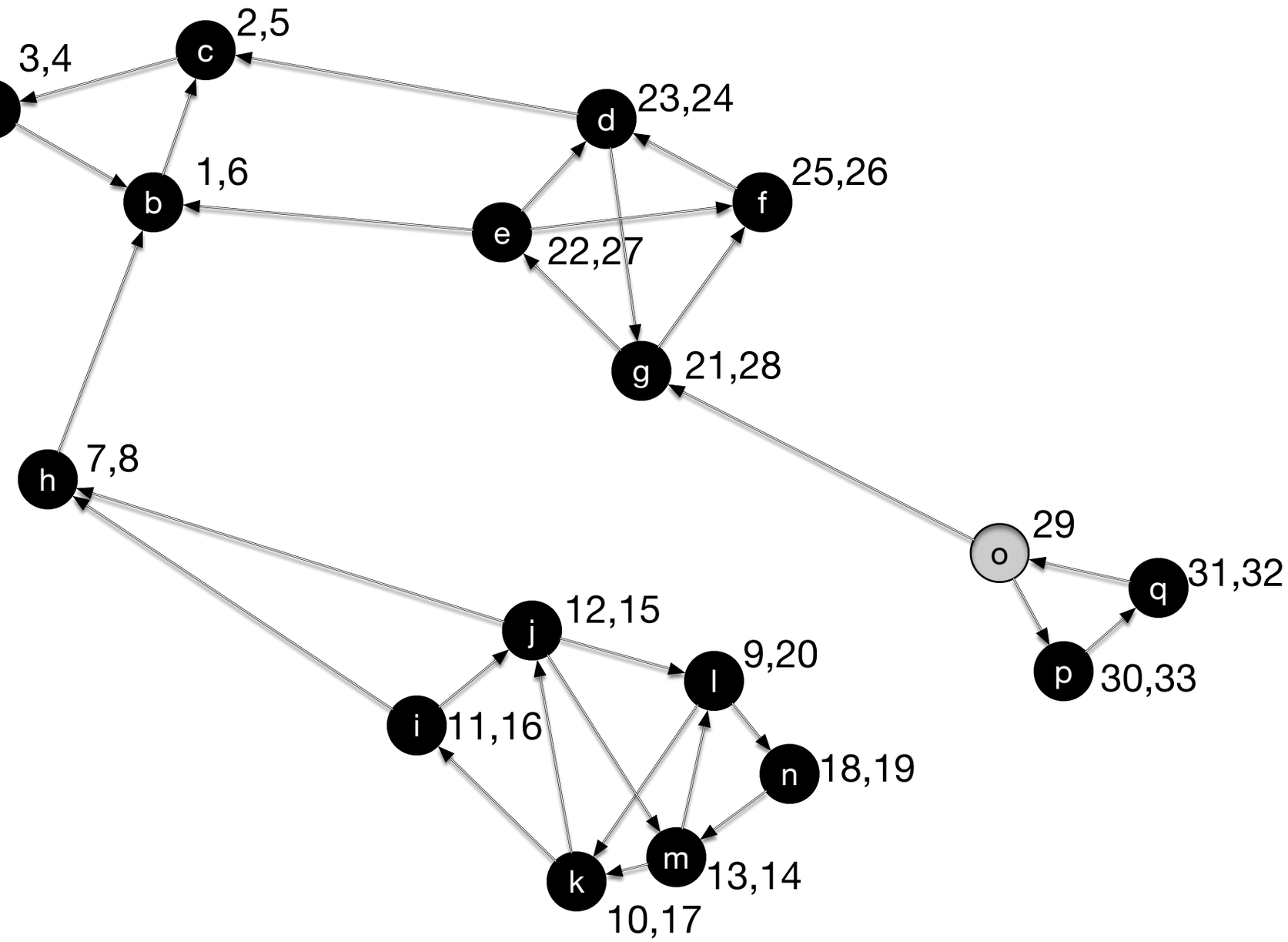
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



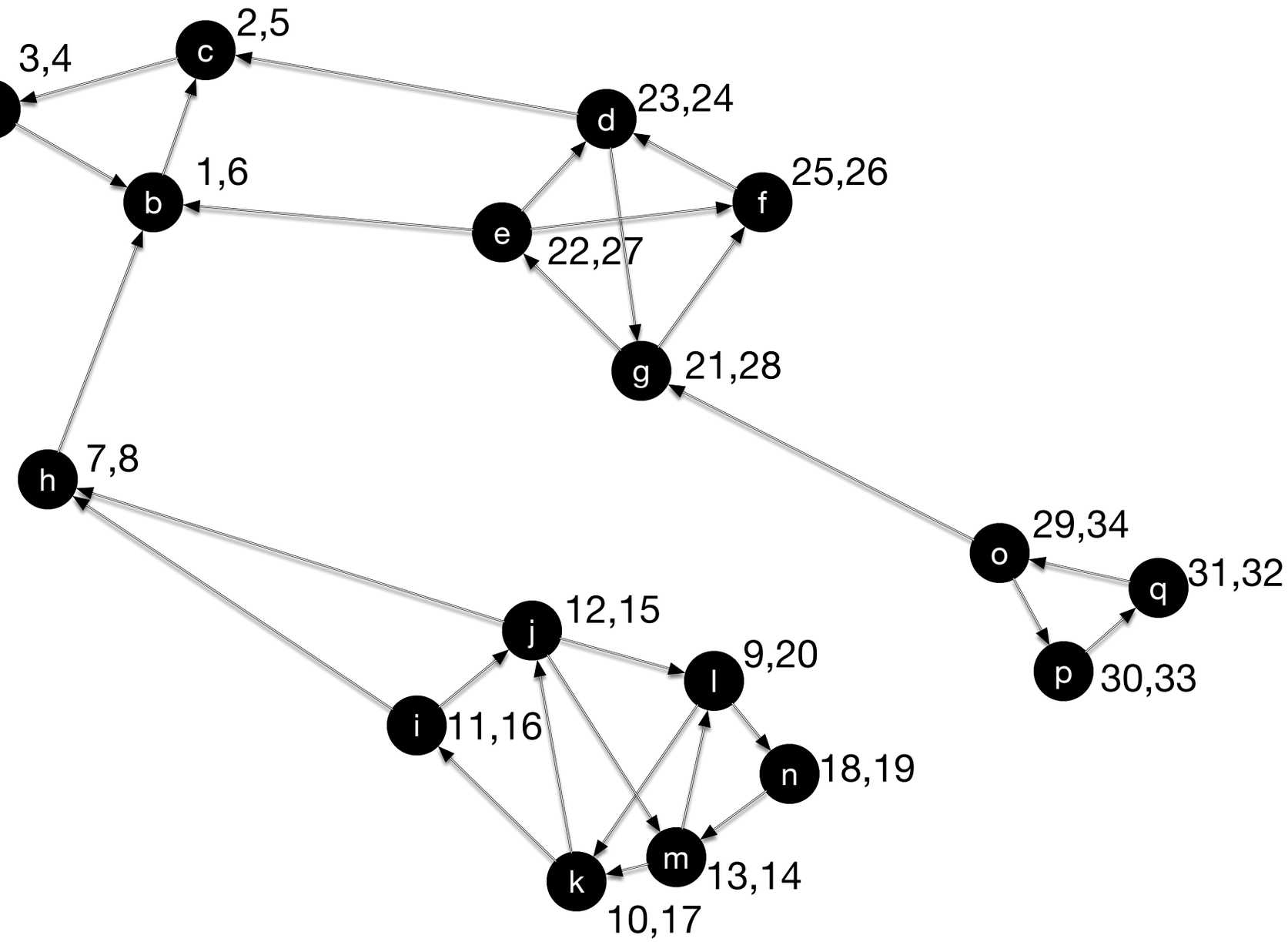
Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



Strongly Connected Components

a: 33
b: 34
c: 32
d: 12
e: 14
f: 11
g:13
h: 29
i: 23
j: 24
k: 21
l: 26
m: 25
n: 18
o: 8
p: 6
q: 7



Strongly Connected Components

- At this point, we emit our final SCC as $\{o, p, q\}$

Strongly Connected Components

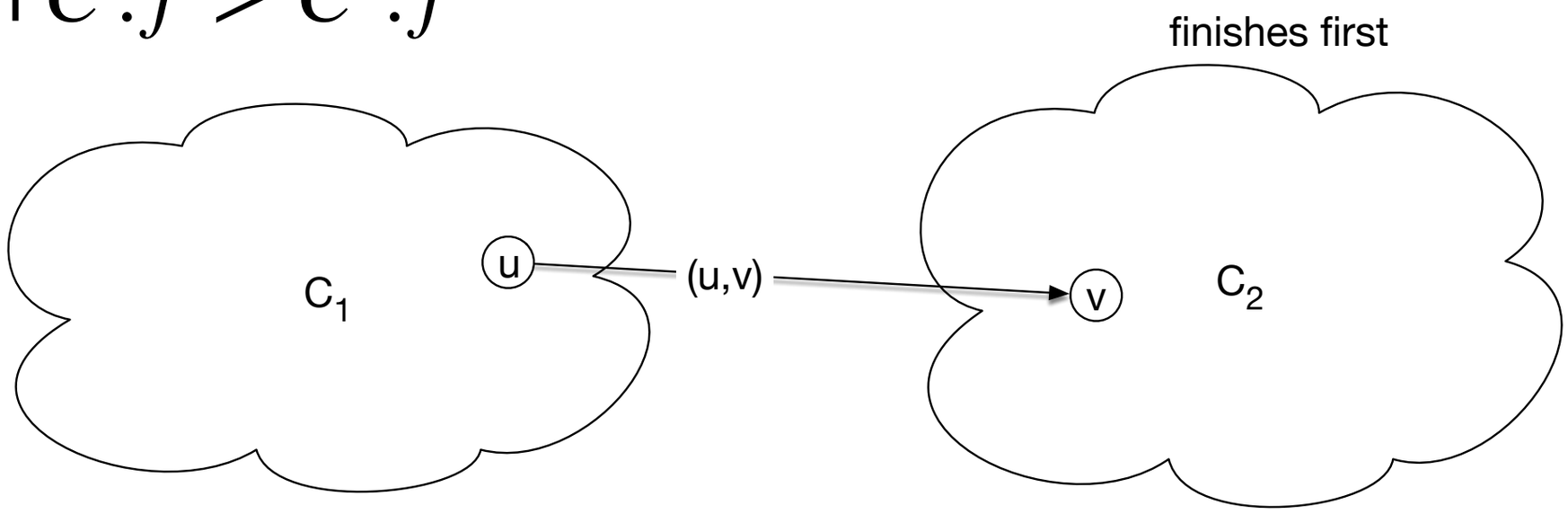
- This example shows
  - how the first pass DFS identifies SCC that are high up in the SCC meta-graph
  - how the DFS in the reverse graph identifies then SCC, but because of the choice of the starting points, any SCC that can be reached are already done and all blackened

Strongly Connected Components

- For any set of nodes $U \subset V$, we set
  - $U.d = \min(\{u.d \mid u \in U\})$
  - $U.f = \max(\{u.f \mid u \in U\}$
- This means:
  - the discovery time of U is the earliest discovery time in U
  - the finish time of U is the latest finish time of a node in U

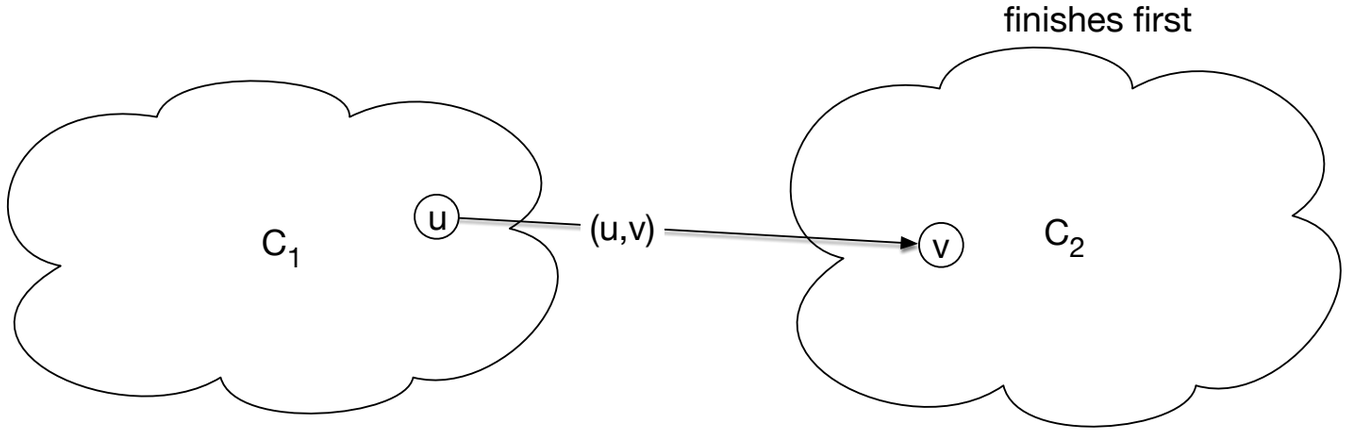
Strongly Connected Components

- Lemma:
  - Let C and C' be two strongly connected components of the directed graph $G = (V, E)$.
  - Suppose there is an edge (u, v) with $u \in C$ and $v \in C'$
  - Then $C.f > C'.f$



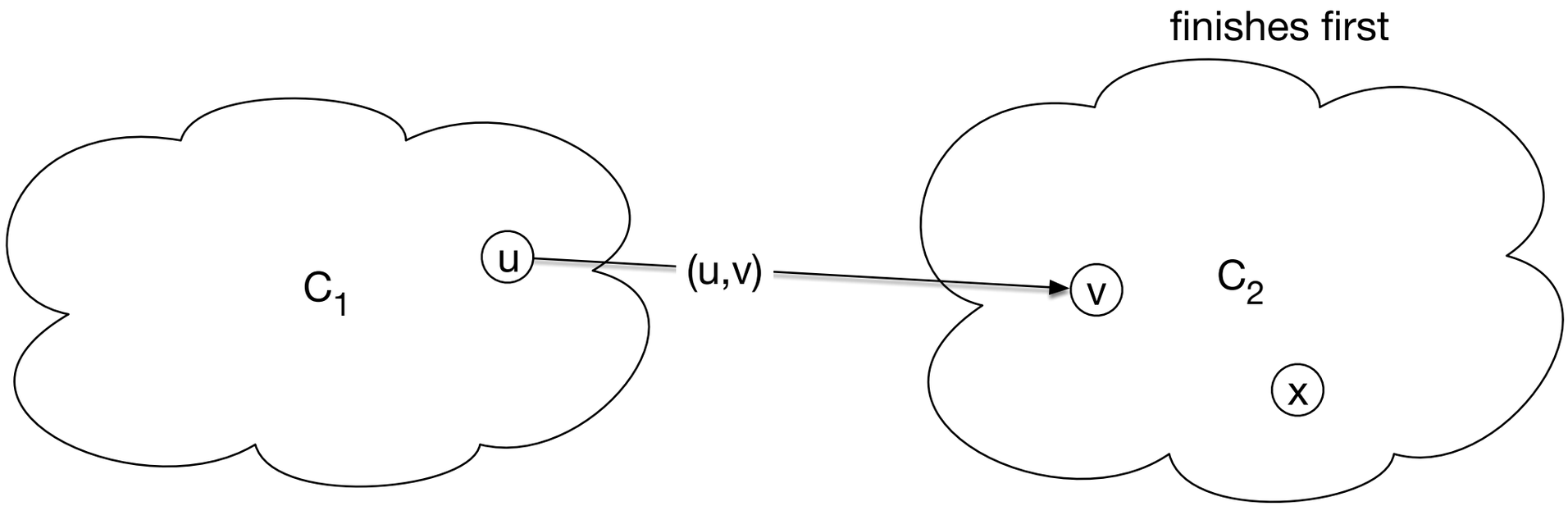
Strongly Connected Components

- Proof:
- Assume that we discover a vertex in C_1 first
- Let x be that vertex
- Then there is a white path from x to any vertex in C_2 (maybe via v)
- So x becomes an ancestor of all nodes in C_2
- Therefore, x finishes after all nodes in C_2 and has a later finishing time



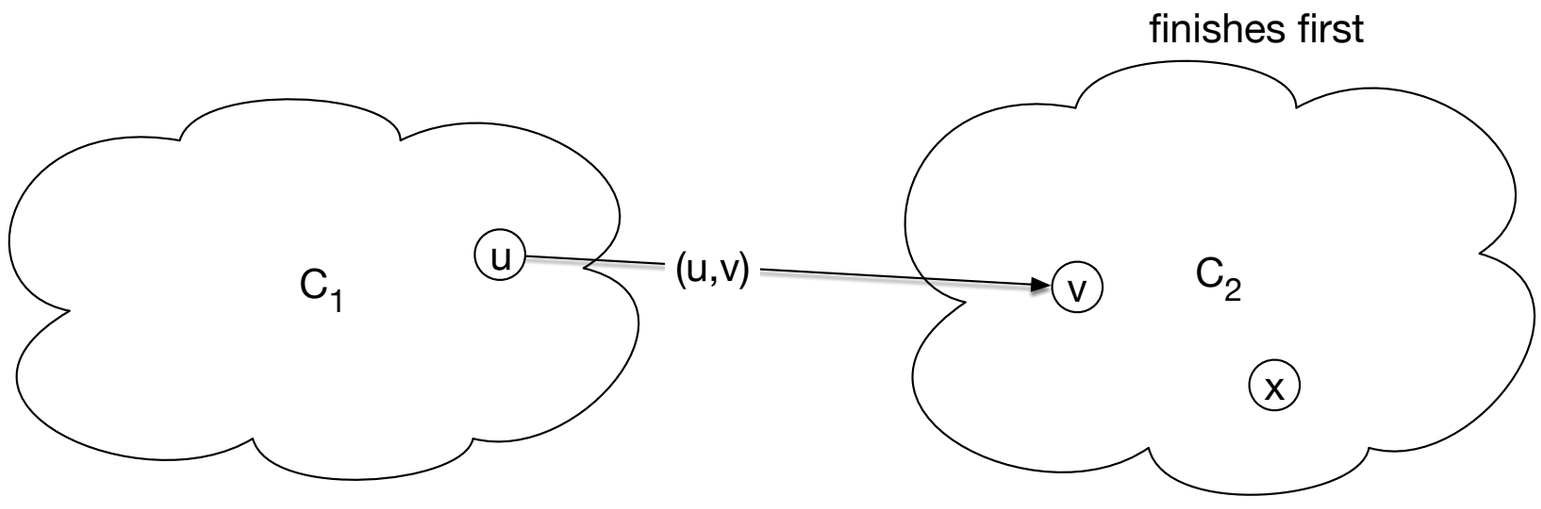
Strongly Connected Components

- Assume that we discover a vertex x in C_2 first
  - There cannot be a path from x to any node in C_1 or by the previous lemma, the two SCC would be the same



Strongly Connected Components

- By the white path theorem, x is not going to be an ancestor of any node in C_1, but it is going to be an ancestor of everything in C_2
- Therefore, x finishes before anything in C_1 can be discovered.
- qed



Strongly Connected Components

- Applied to the reverse graph, we get
  - Let C and C' be two strongly connected components of the directed graph $G = (V, E)$.
  - Suppose there is an edge (u, v) with $u \in C$ and $v \in C'$
  - Then $C.f < C'.f$ in the DFS running on the reverse graph of G

Strongly Connected Components

- Theorem
  - The algorithm correctly calculates the SCC of a graph

- Proof proceeds by induction on the number k of depth first trees found in the second pass of DFS
  - Induction base: $k = 0$ is trivial
  - Induction step: Assume the first k SCCs have been correctly generated

Strongly Connected Components

- We start out from a vertex u
  - $u \in C$ for a strongly connected component C
  - because we pick u in order of finishing time, any other SCC C' has a finishing time in the first pass smaller than u
  - by the consequence of the lemma and induction hypothesis, all edges that leave C in the reverse graph can only be to SCC already emitted
  - All nodes in C must be white by induction hypothesis
  - Thus, the descendants of u are exactly the nodes in C