CSC 225 - Summer 2019

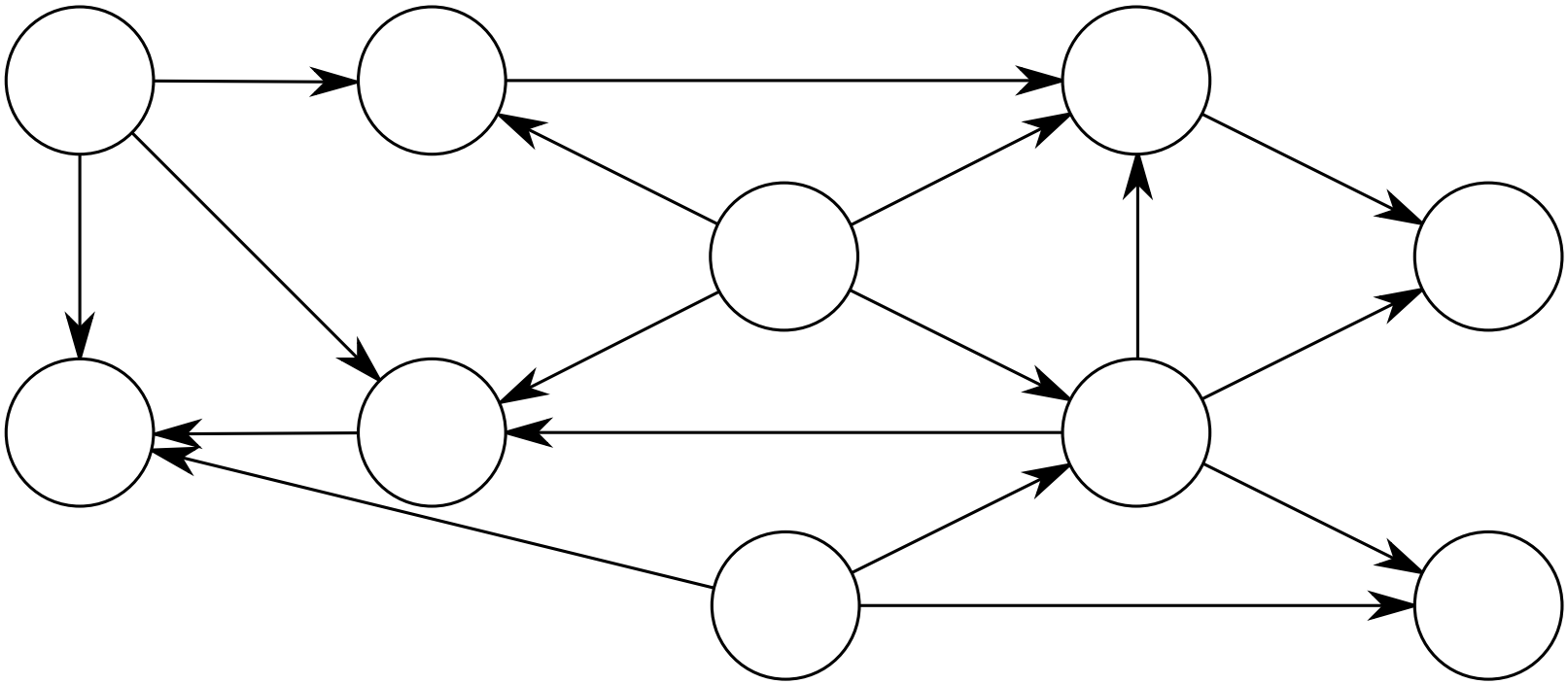
Topological Sorting

Bill Bird

Department of Computer Science
University of Victoria

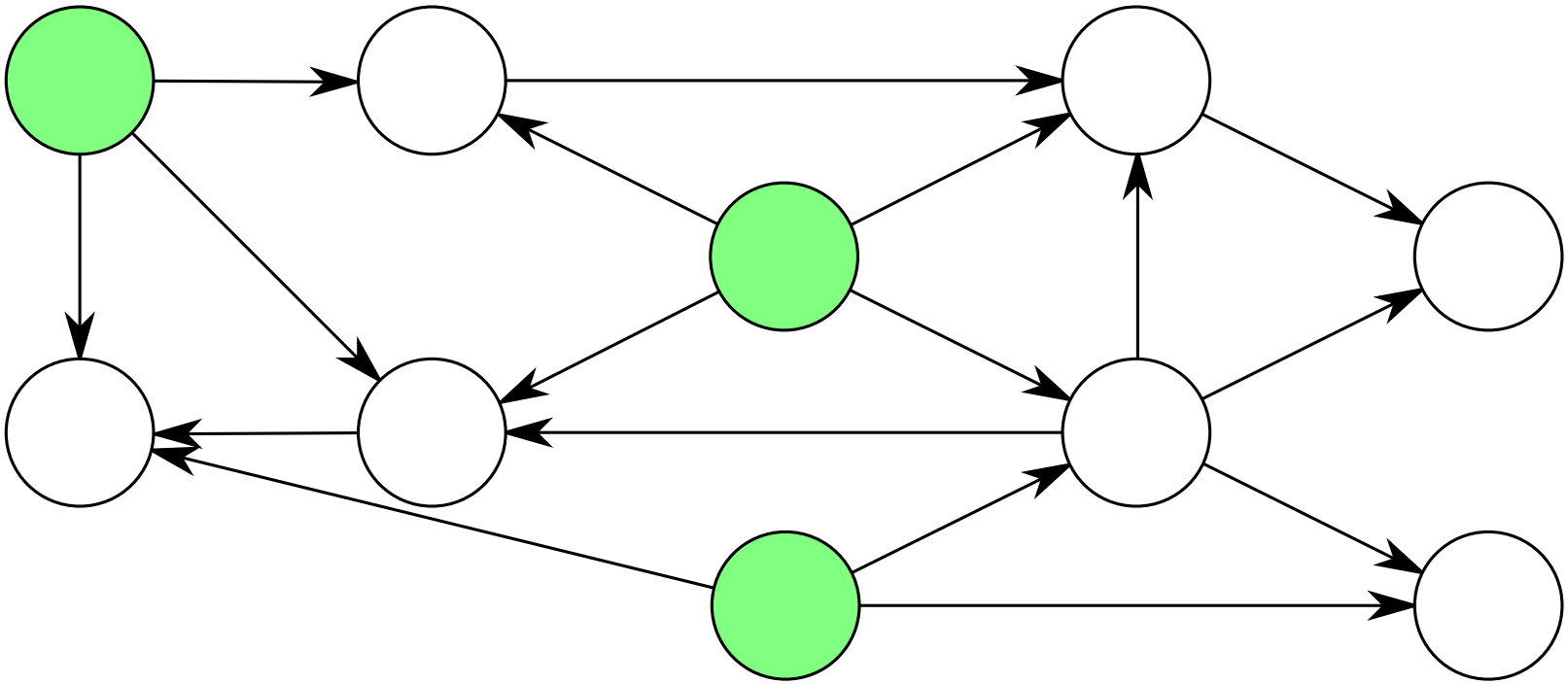
July 24, 2019

Directed Acyclic Graphs (1)



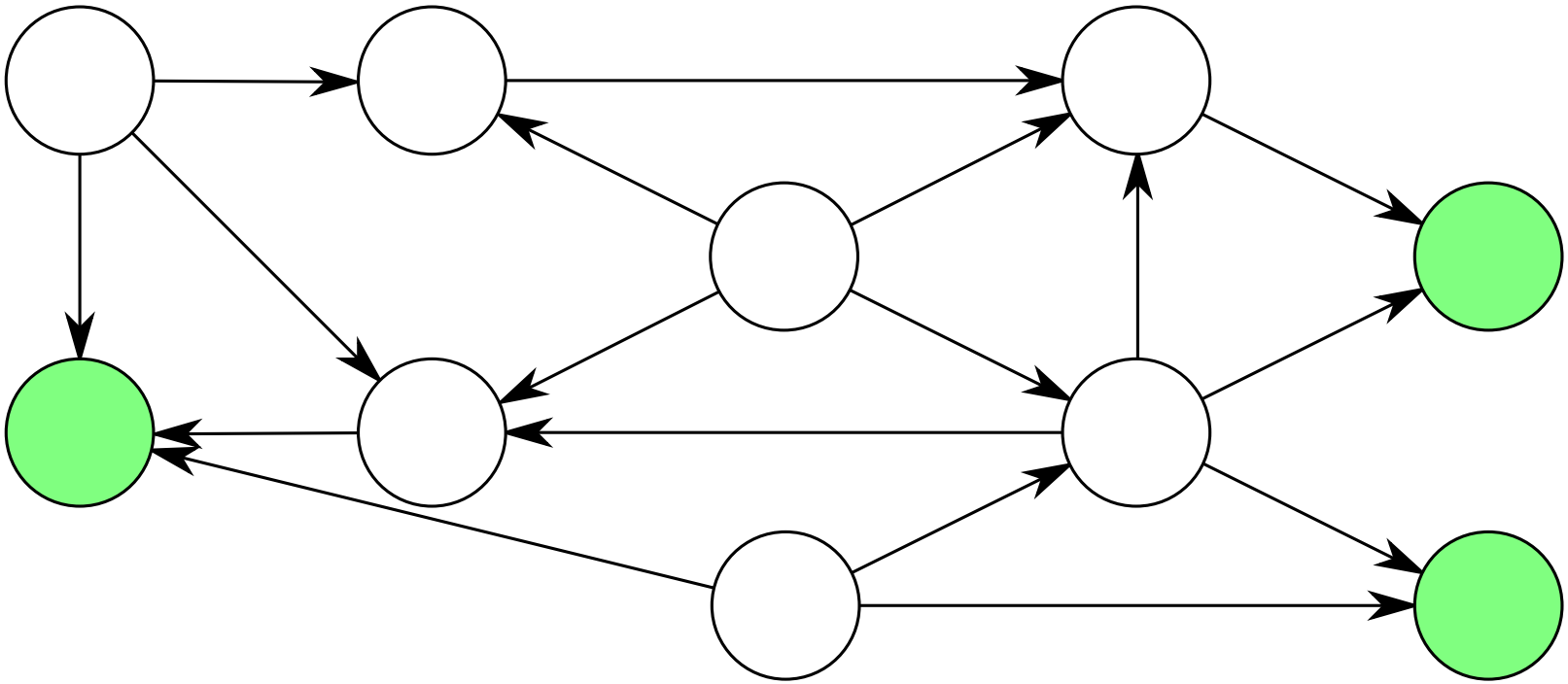
A **directed acyclic graph** (DAG) is a directed graph with no directed cycles.

Directed Acyclic Graphs (2)



In a DAG, a **minimum** is a vertex with no incoming edges.

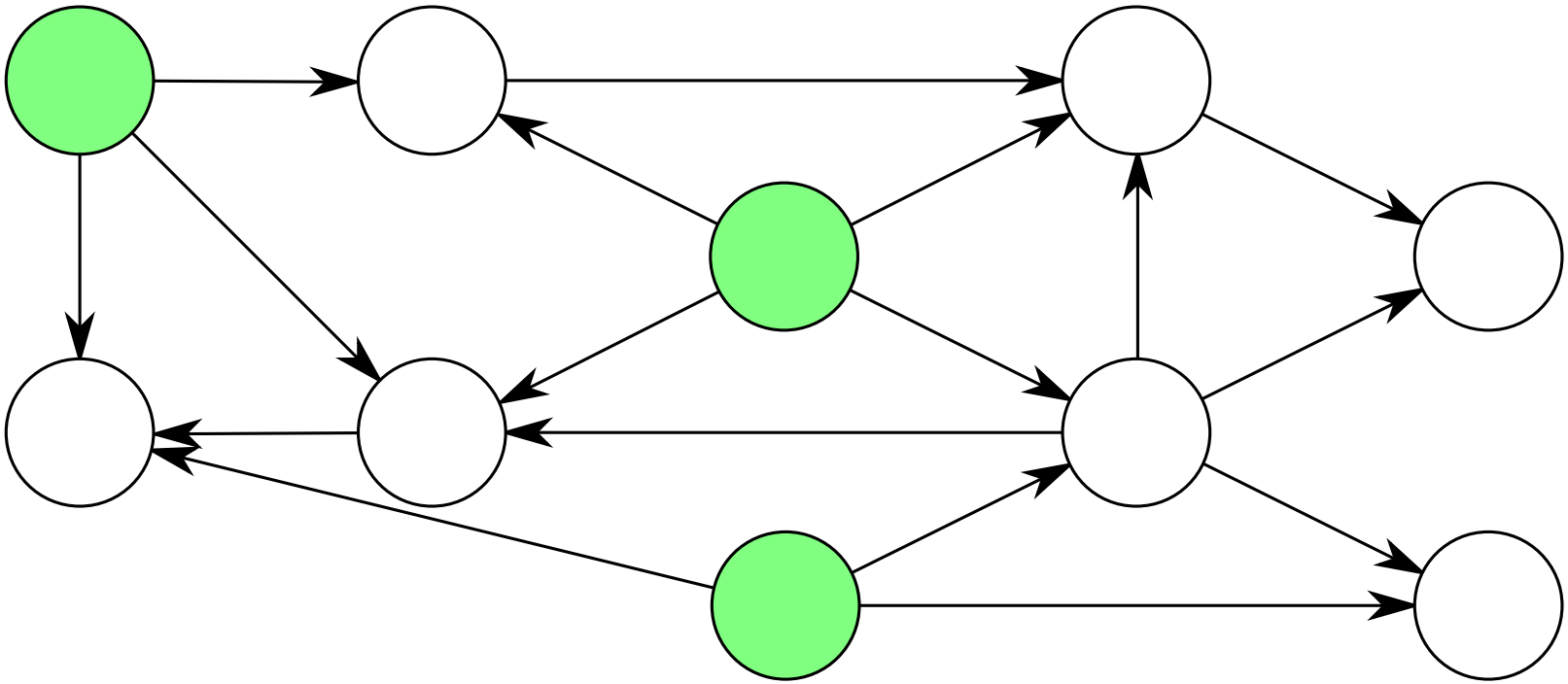
Directed Acyclic Graphs (3)



A **maximum** is a vertex with no outgoing edges.

(Note that some sources may swap the definitions of minimum and maximum)

Directed Acyclic Graphs (4)



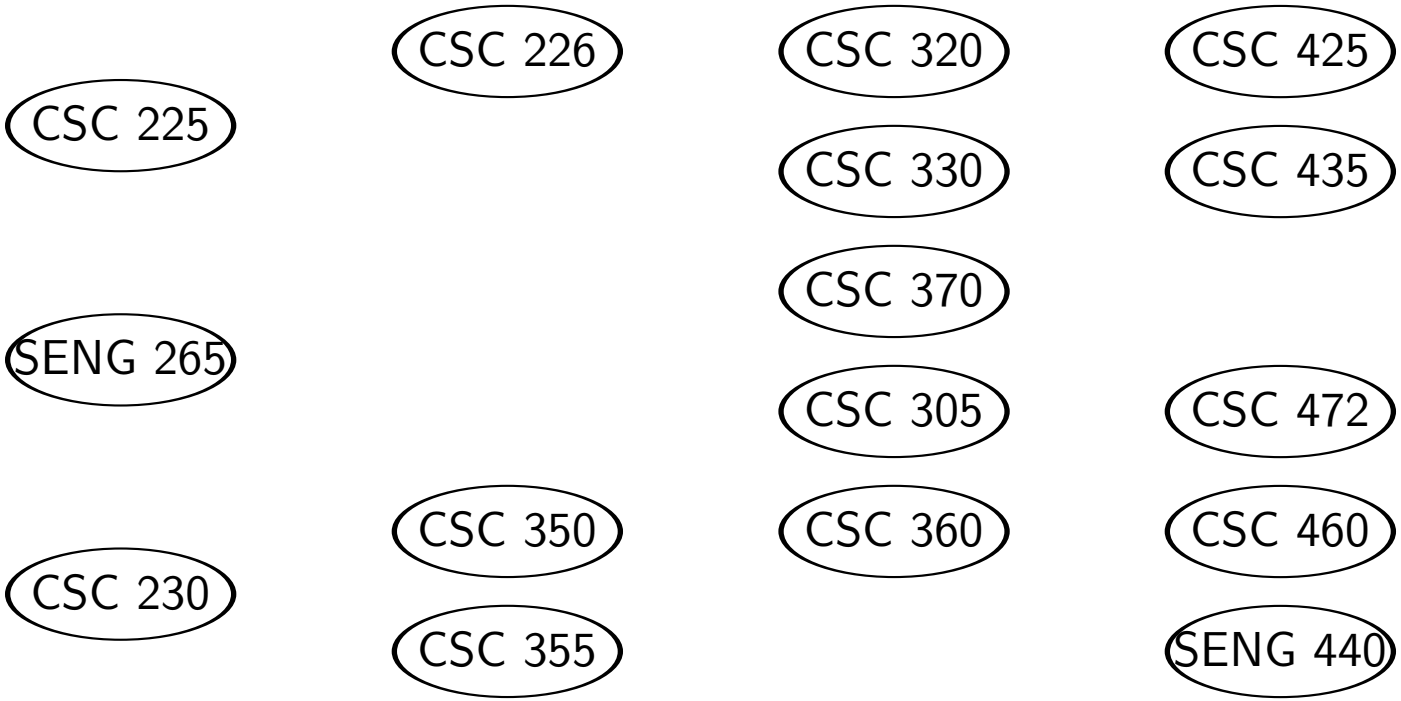
Exercise: Prove that every DAG contains at least one minimum vertex.

Scheduling Classes (1)

Course	Prerequisites
CSC 226	CSC 225
CSC 305	CSC 226, SENG 265
CSC 320	CSC 226
CSC 330	CSC 225, CSC 230, SENG 265
CSC 350	CSC 225, CSC 230
CSC 355	CSC 230
CSC 360	CSC 226, CSC 230, SENG 265
CSC 370	CSC 226, SENG 265
CSC 425	CSC 320
CSC 435	CSC 320, CSC 330
CSC 460	CSC 355, CSC 360
CSC 472	CSC 305
SENG 440	CSC 355

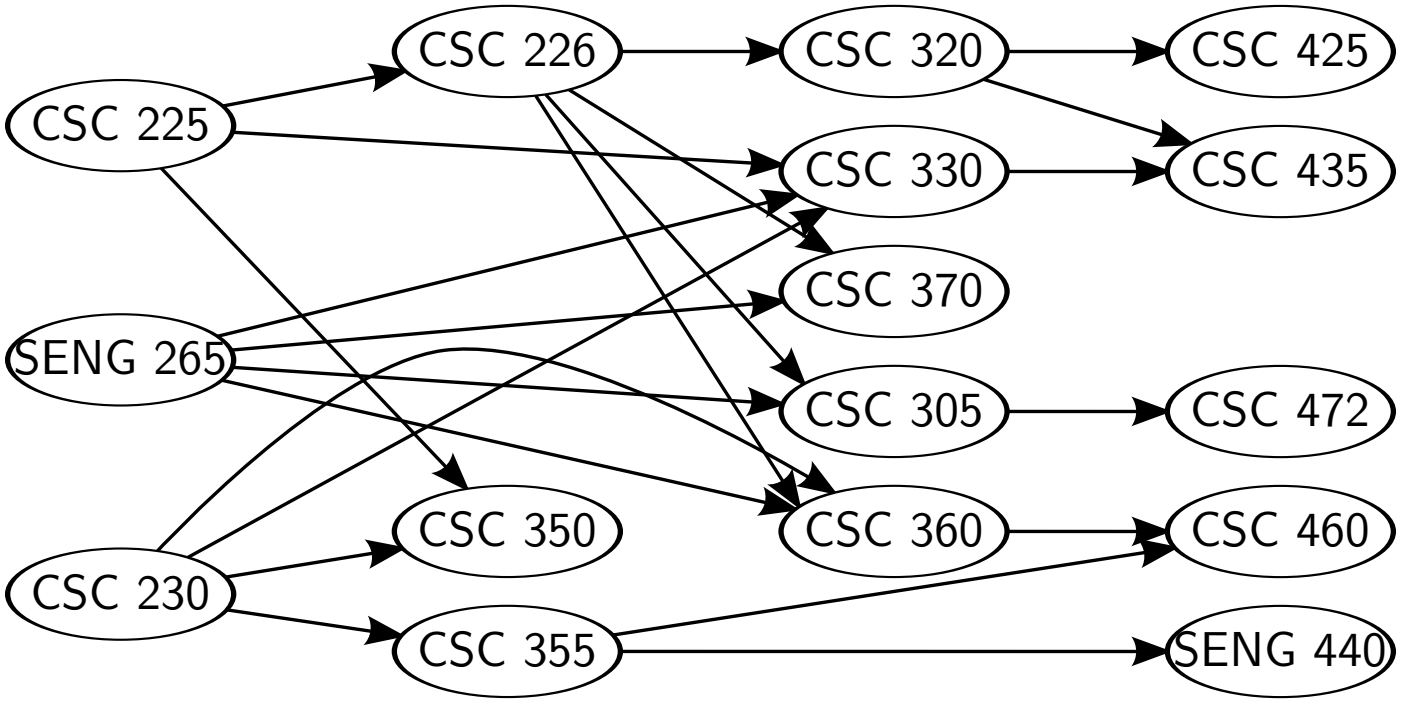
Problem: Find an ordering of the courses above such that no course is taken before its prerequisite.

Scheduling Classes (2)



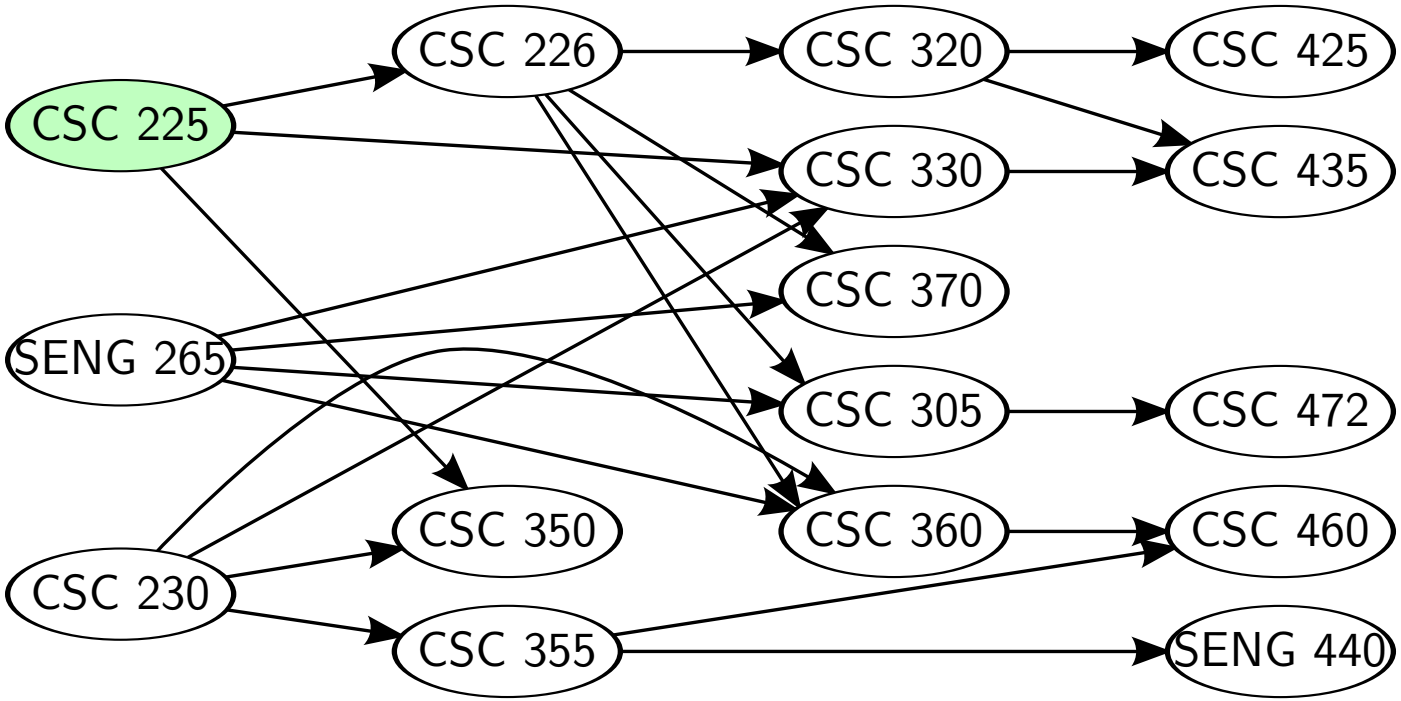
The prerequisite constraints can be modelled with a directed graph, where each vertex represents a course and arcs connect each course to the courses that require it.

Scheduling Classes (3)



It is illogical for a course to be its own prerequisite (directly or indirectly), so the graph must be acyclic.

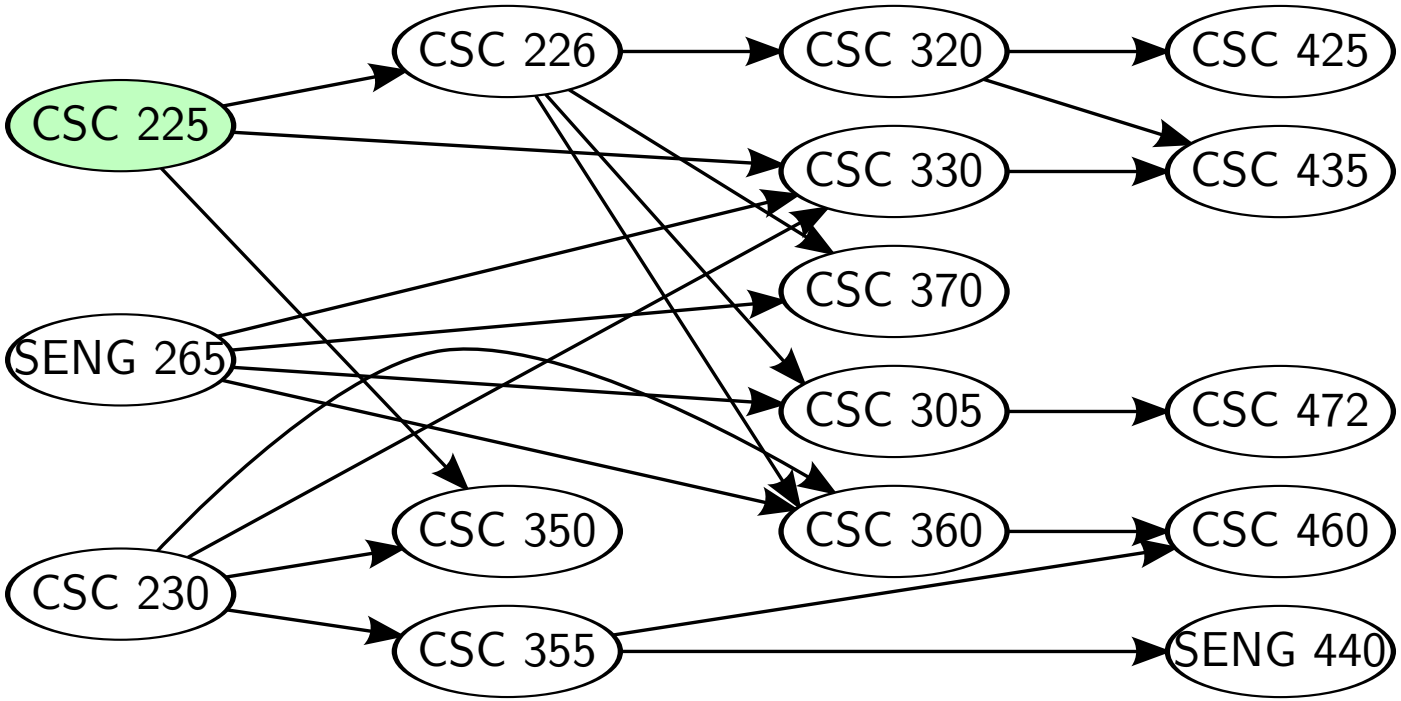
Scheduling Classes (4)



Ordering: CSC 225

A minimum vertex in the graph corresponds to a course with no prerequisites. Therefore, any minimum vertex can appear first in a valid ordering.

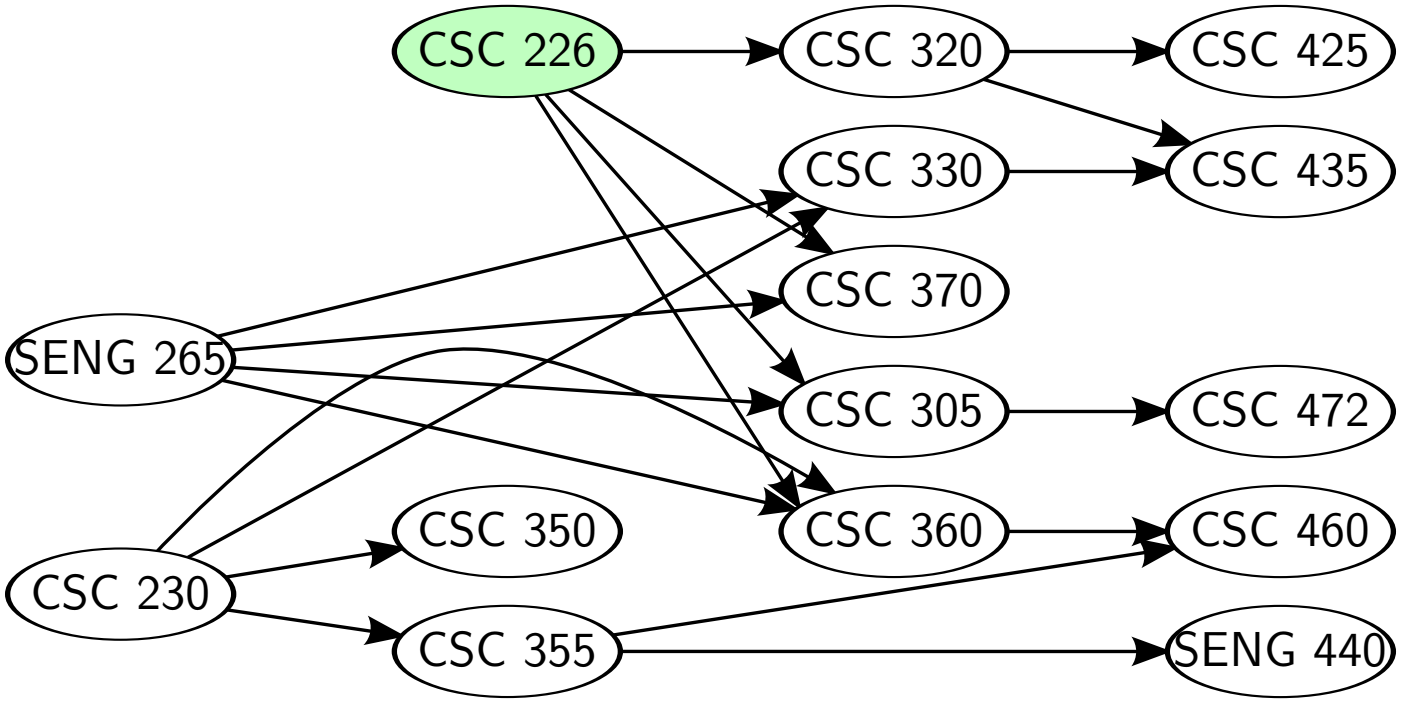
Scheduling Classes (5)



Ordering: CSC 225

Once a course C has been taken, no prerequisites involving C can be violated. Therefore, once a vertex has been added to the ordering, it (and all incident edges) can be deleted.

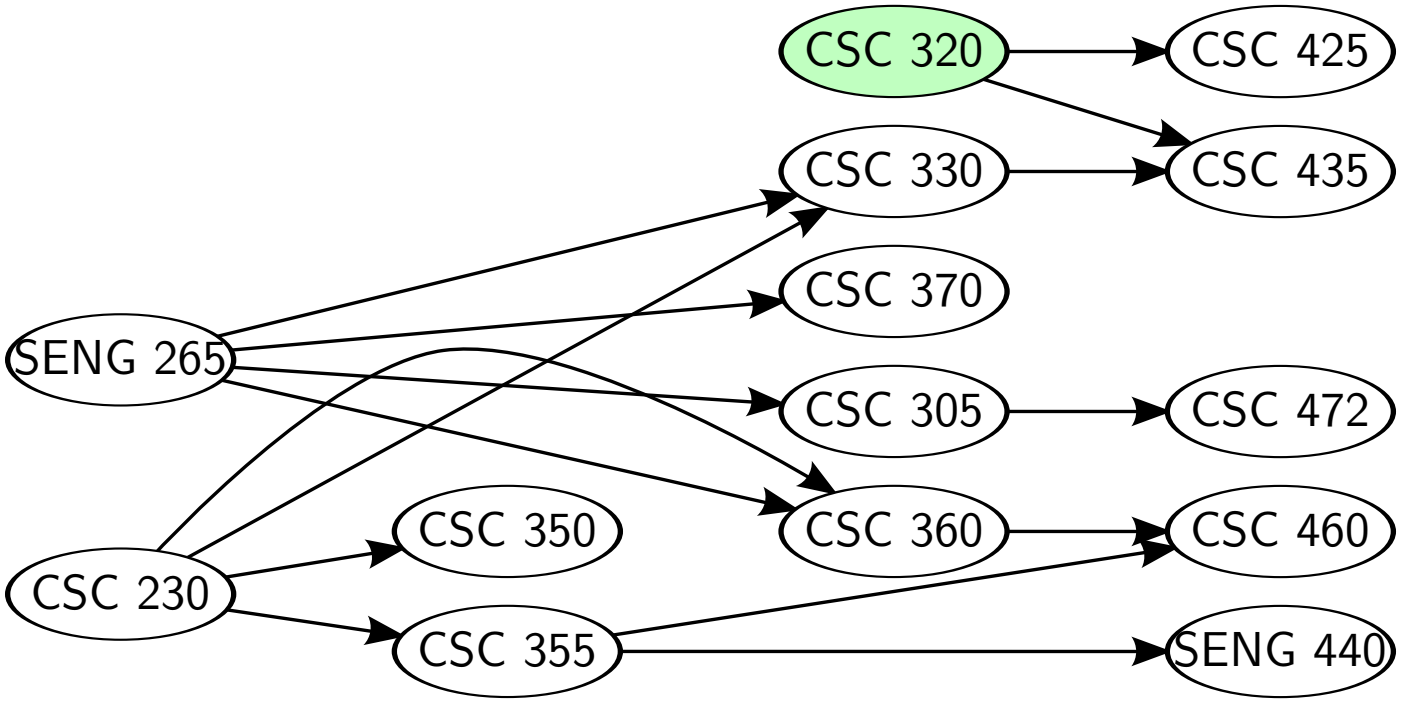
Scheduling Classes (6)



Ordering: CSC 225, CSC 226

After deleting CSC 225 from the graph, CSC 226 becomes a minimum vertex and can be added to the ordering.

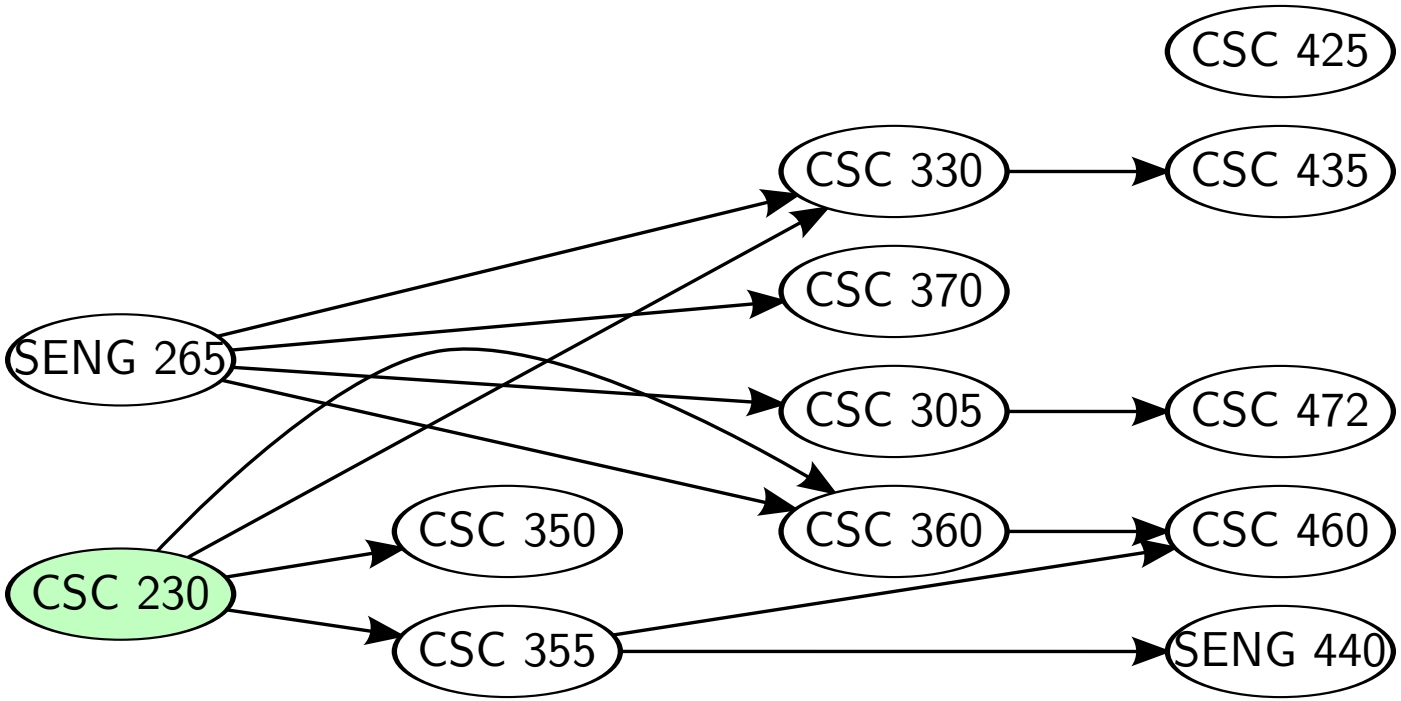
Scheduling Classes (7)



Ordering: CSC 225, CSC 226, CSC 320

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

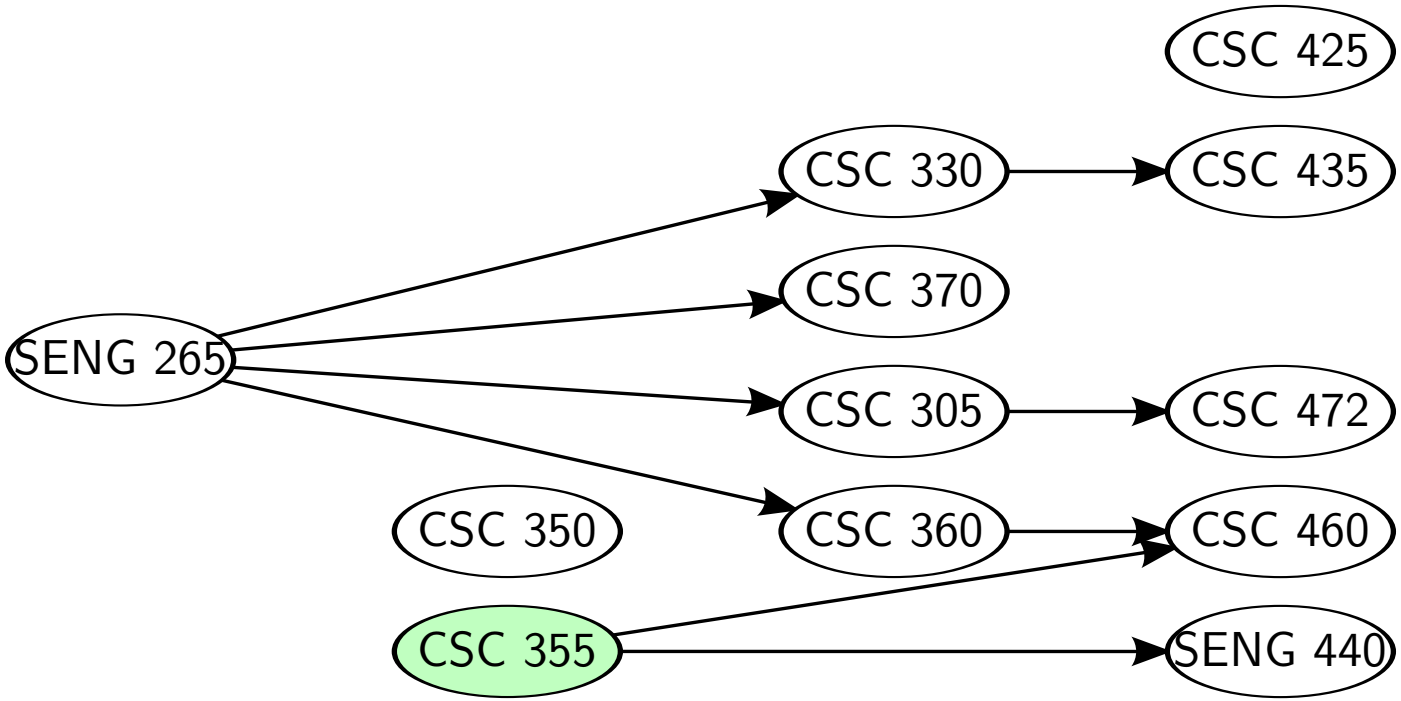
Scheduling Classes (8)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

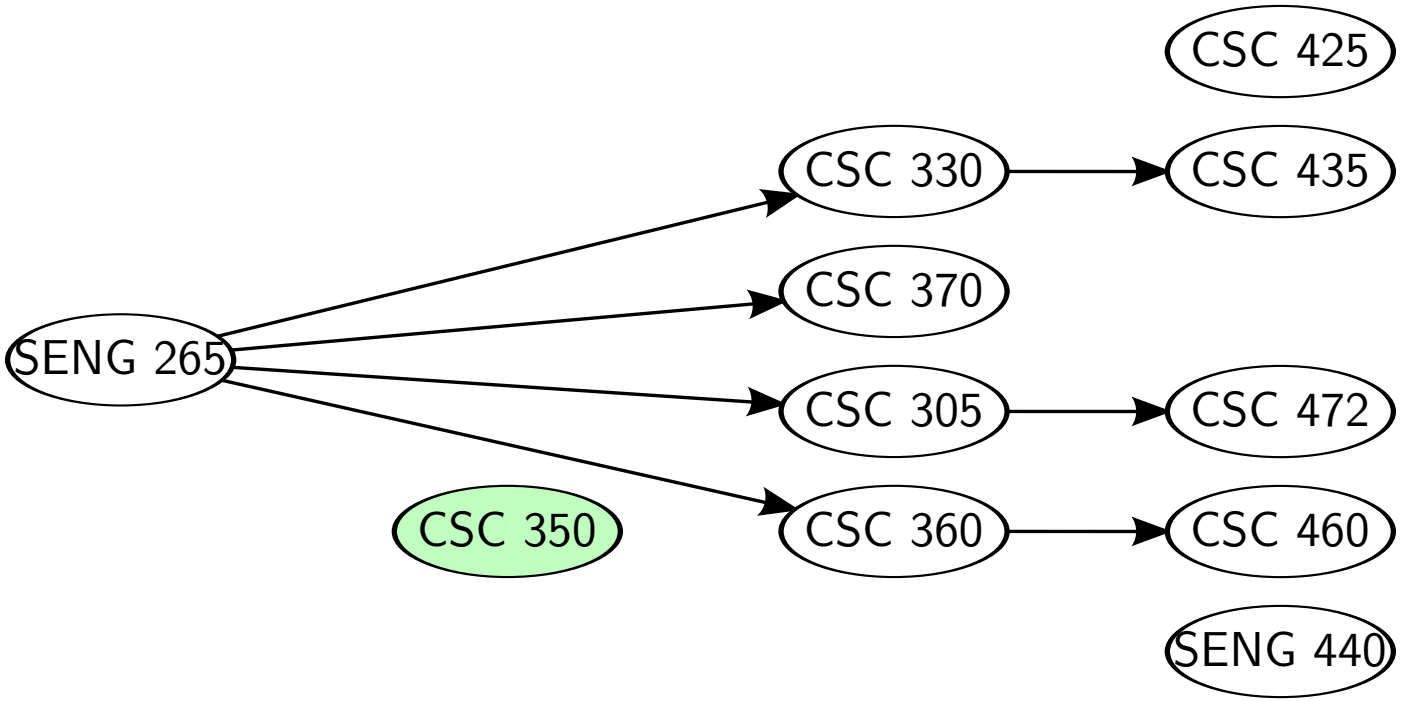
Scheduling Classes (9)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

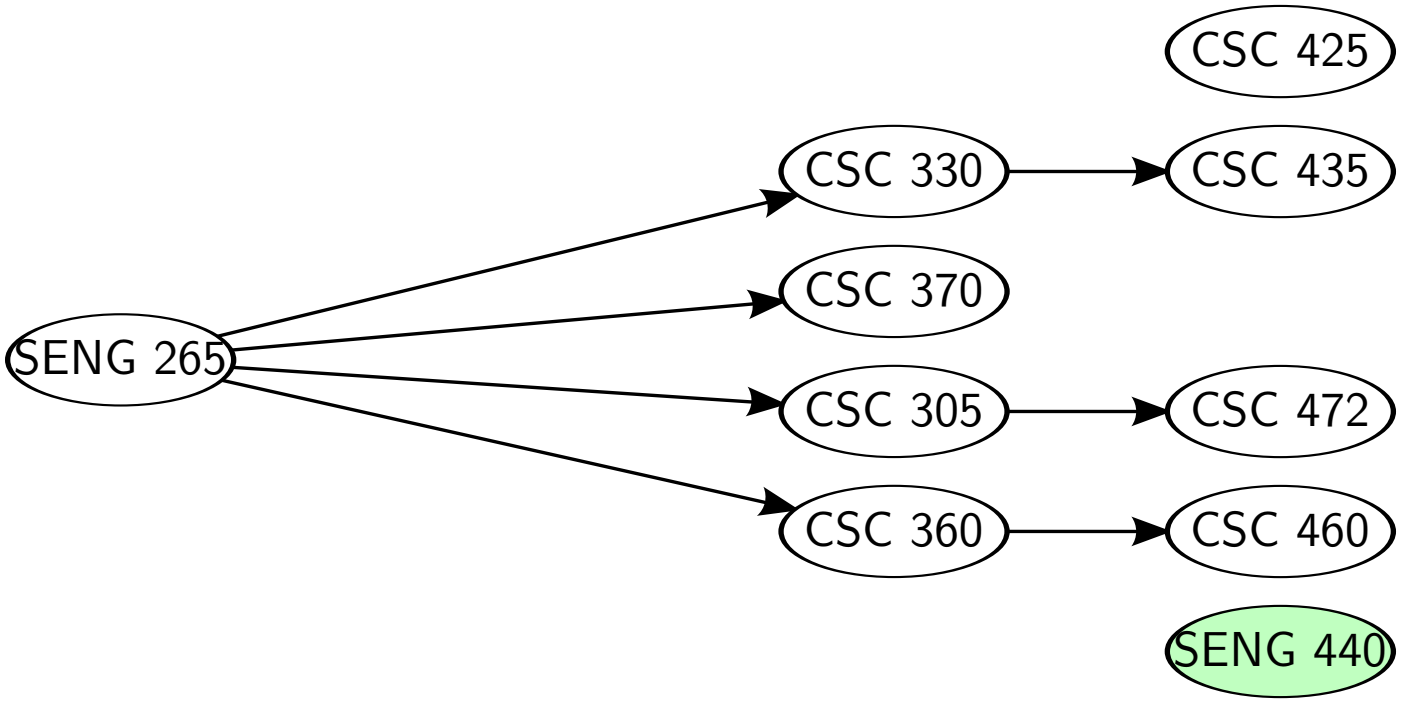
Scheduling Classes (10)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

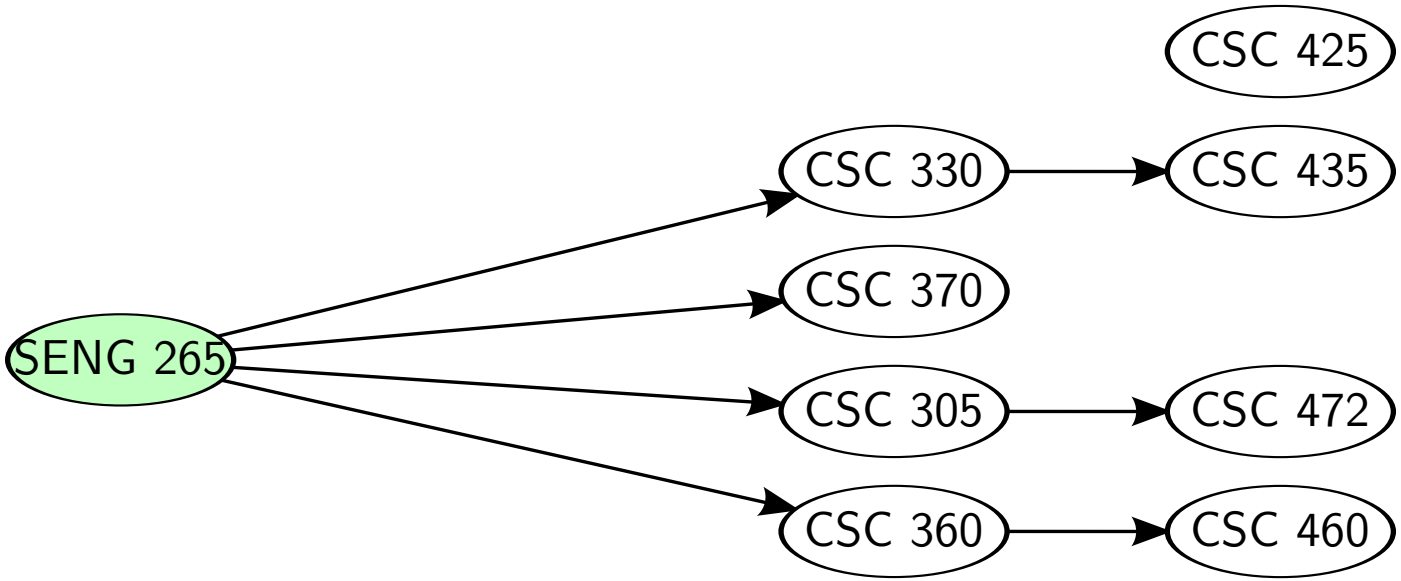
Scheduling Classes (11)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

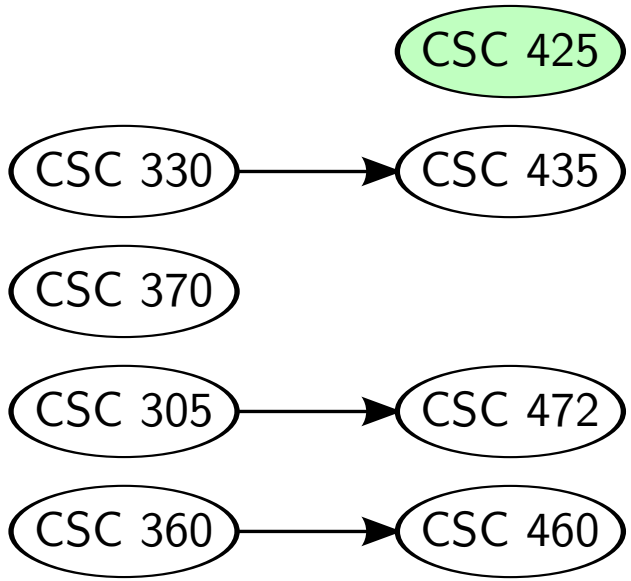
Scheduling Classes (12)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

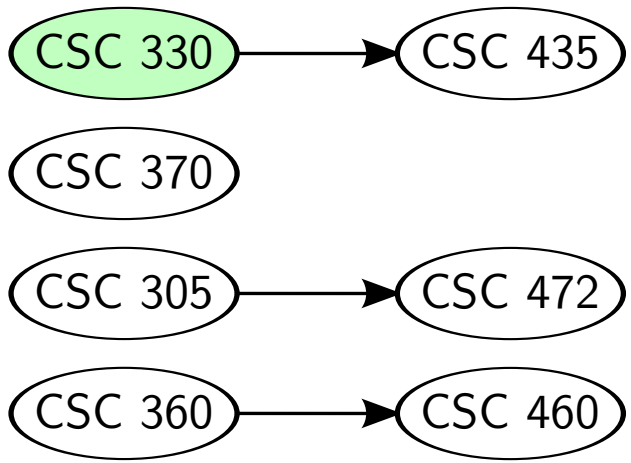
Scheduling Classes (13)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

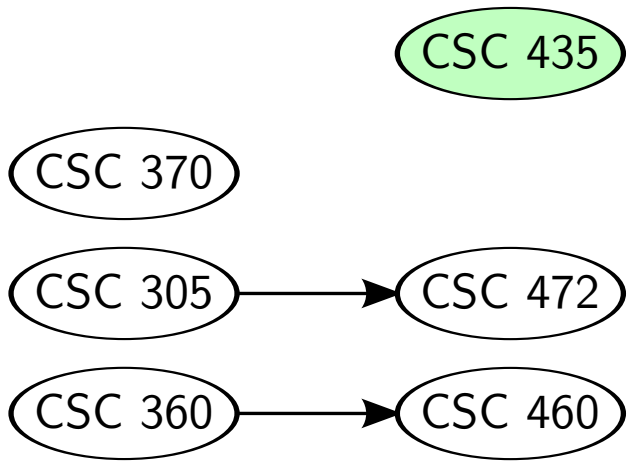
Scheduling Classes (14)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425, CSC 330

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

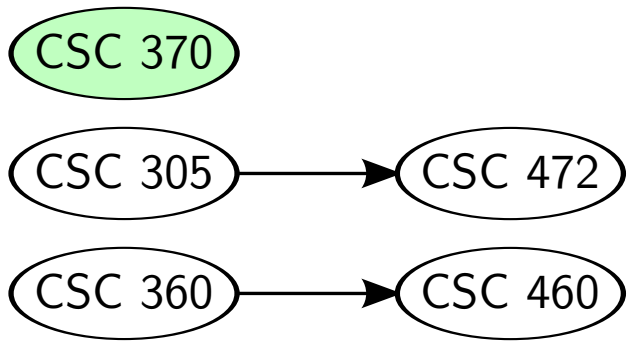
Scheduling Classes (15)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425, CSC 330, CSC 435

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

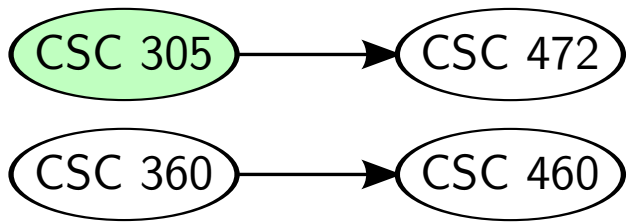
Scheduling Classes (16)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425, CSC 330, CSC 435, CSC 370

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

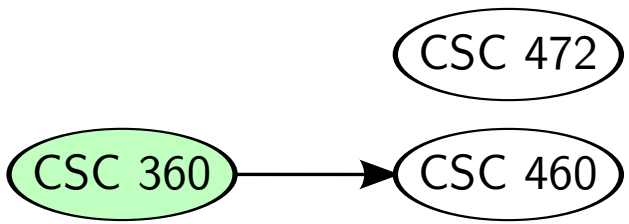
Scheduling Classes (17)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425, CSC 330, CSC 435, CSC 370, CSC 305

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

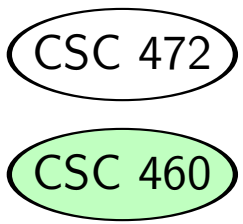
Scheduling Classes (18)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425, CSC 330, CSC 435, CSC 370, CSC 305, CSC 360

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

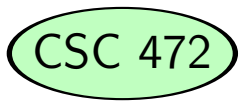
Scheduling Classes (19)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425, CSC 330, CSC 435, CSC 370, CSC 305, CSC 360, CSC 460

By repeatedly selecting minimum vertices, adding them to the ordering and deleting them from the graph, a valid ordering containing all courses can be constructed.

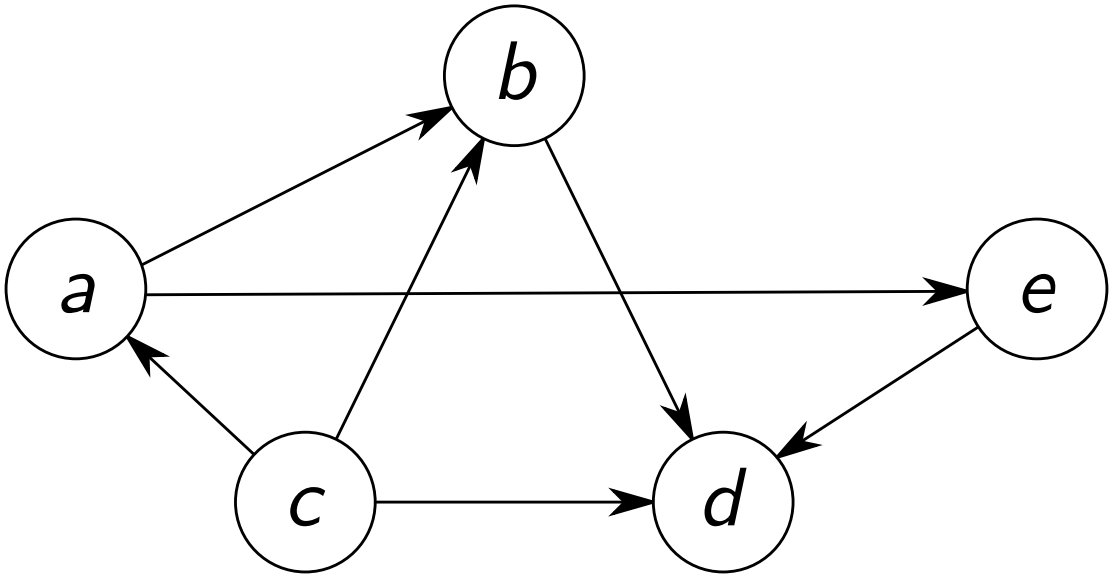
Scheduling Classes (20)



Ordering: CSC 225, CSC 226, CSC 320, CSC 230, CSC 355, CSC 350, SENG 440, SENG 265, CSC 425, CSC 330, CSC 435, CSC 370, CSC 305, CSC 360, CSC 460, CSC 472

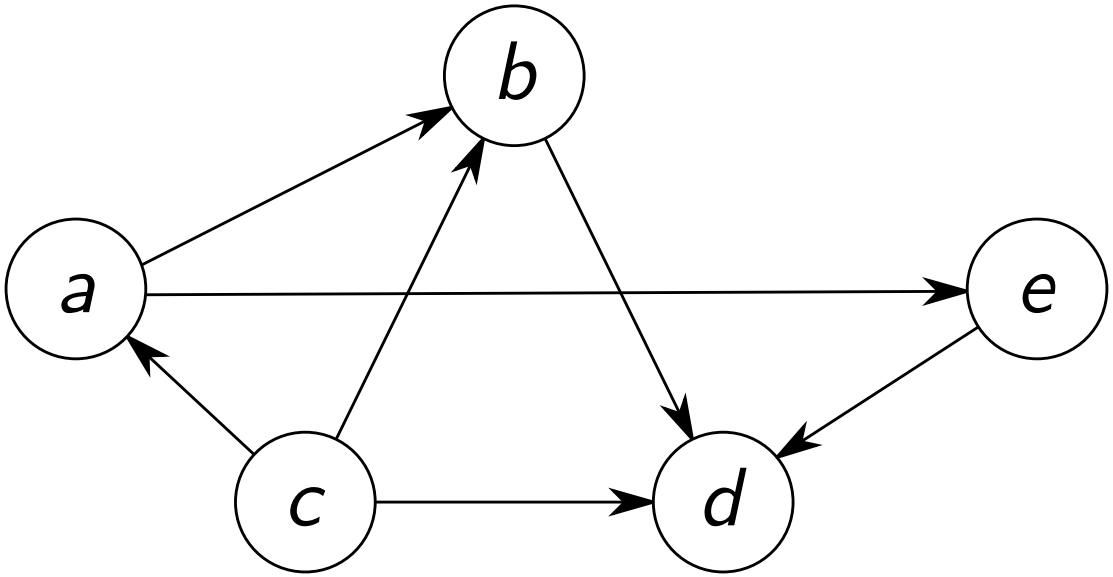
Exercise: Prove that either every vertex will eventually be added to the ordering and deleted, or the graph contains a cycle.

Topological Sorting (1)



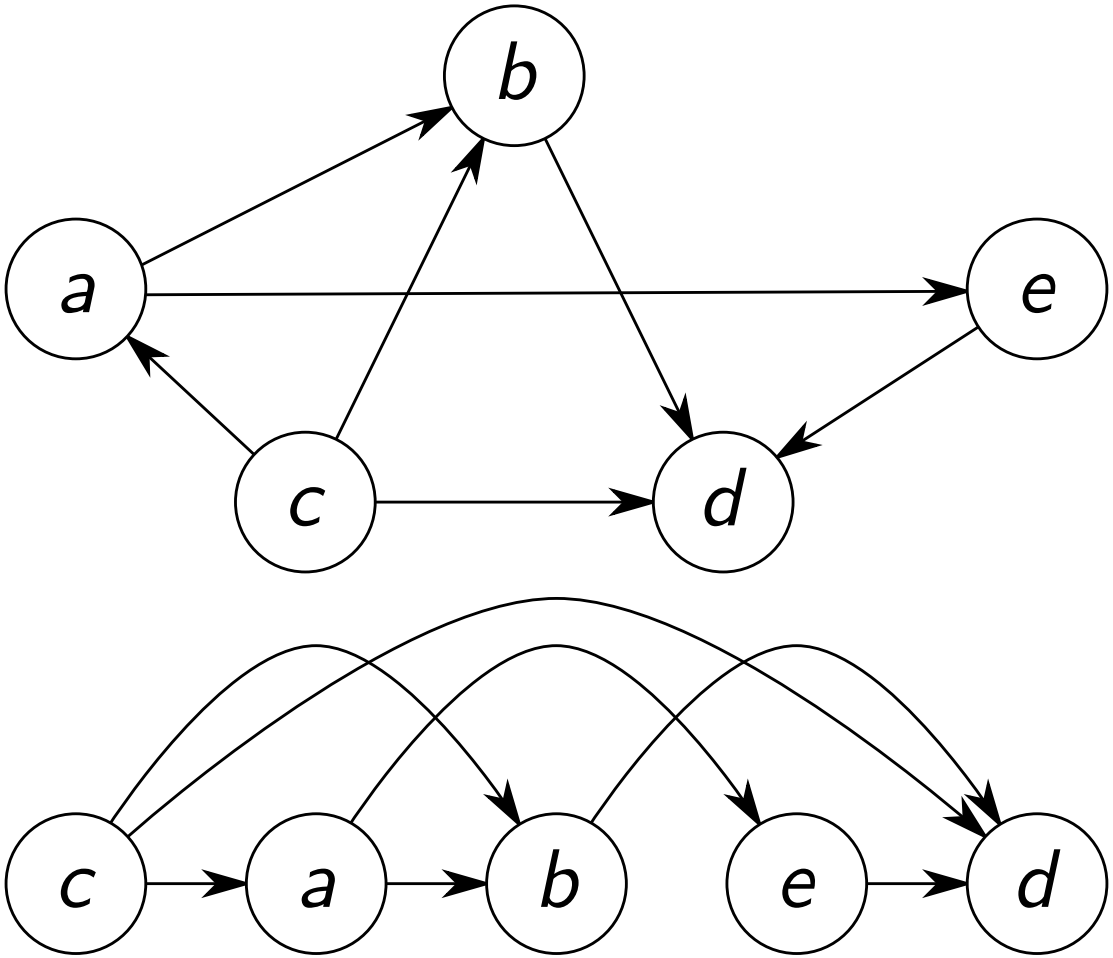
A **topologically sorted ordering** of the vertices of a directed acyclic graph G is a listing of vertices such that if $(u, v) \in E(G)$, then u appears before v in the listing.

Topological Sorting (2)



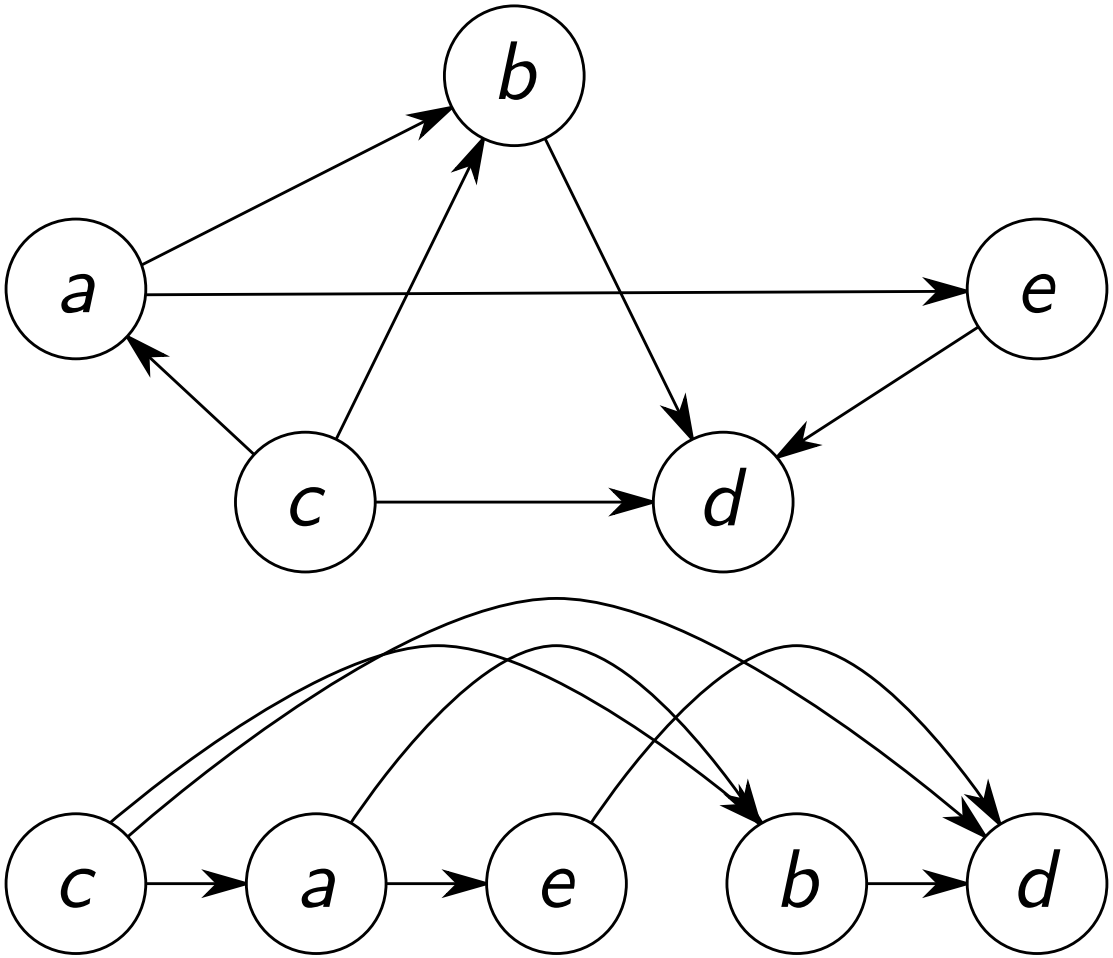
In the graph above, the ordering c, a, b, e, d is topologically sorted.

Topological Sorting (3)



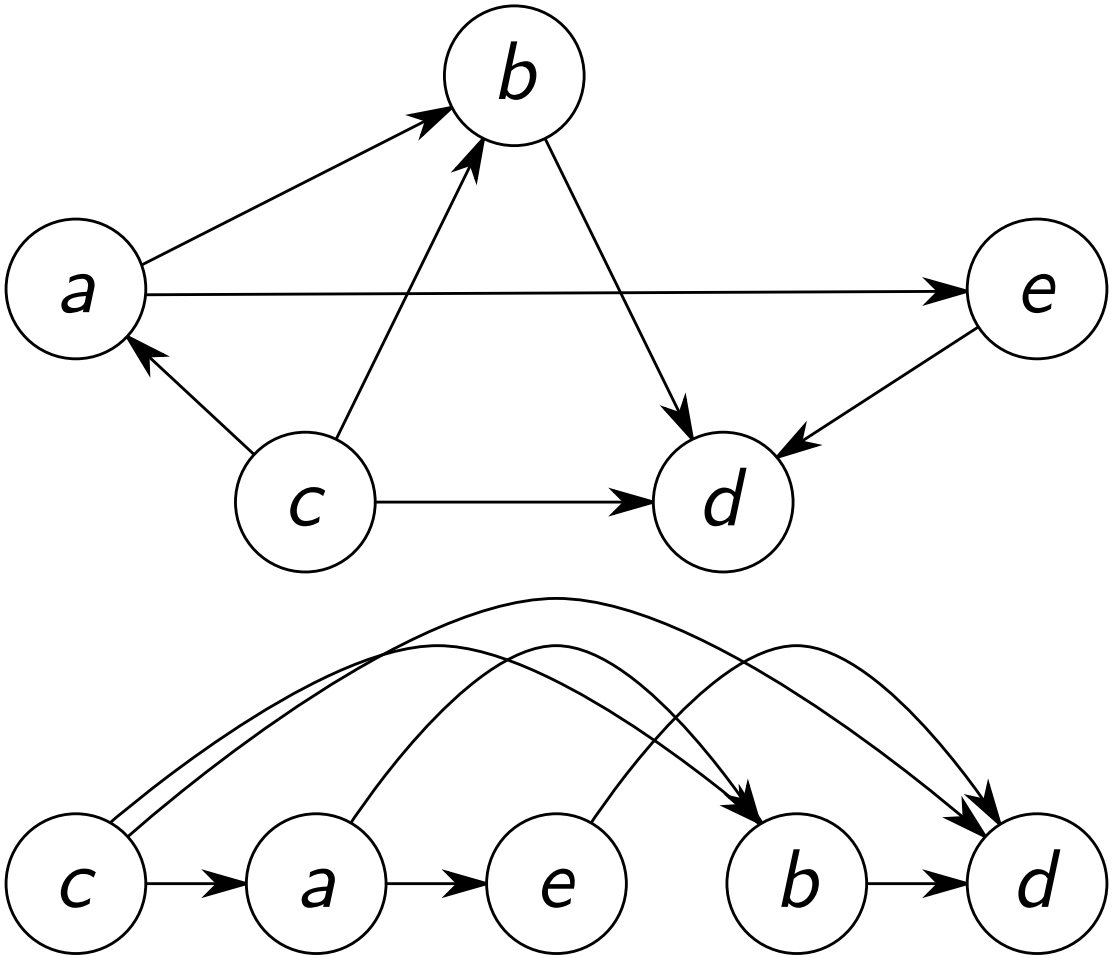
Intuitively, a topologically sorted ordering is an arrangement of vertices from left to right such that all edges point to the right.

Topological Sorting (4)



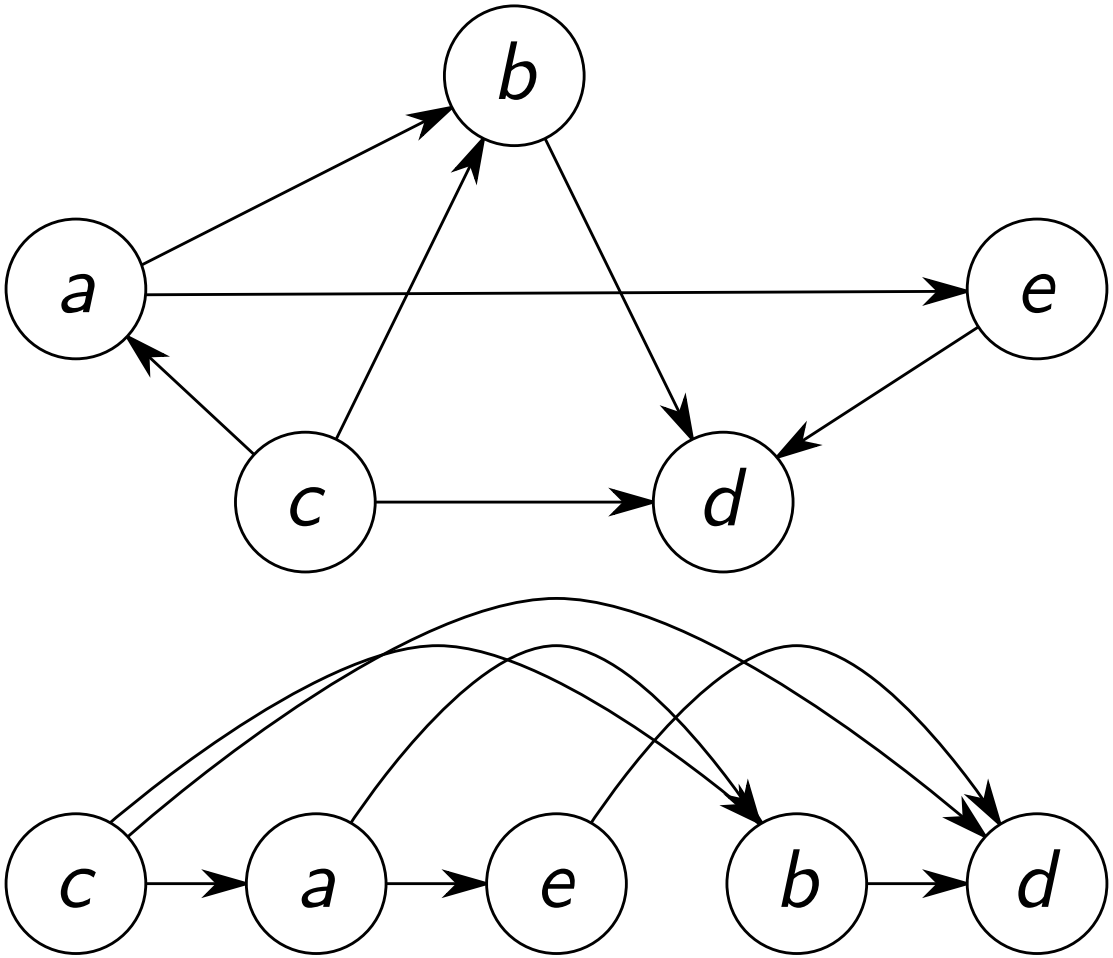
A graph may have multiple topologically sorted orderings.

Topological Sorting (5)



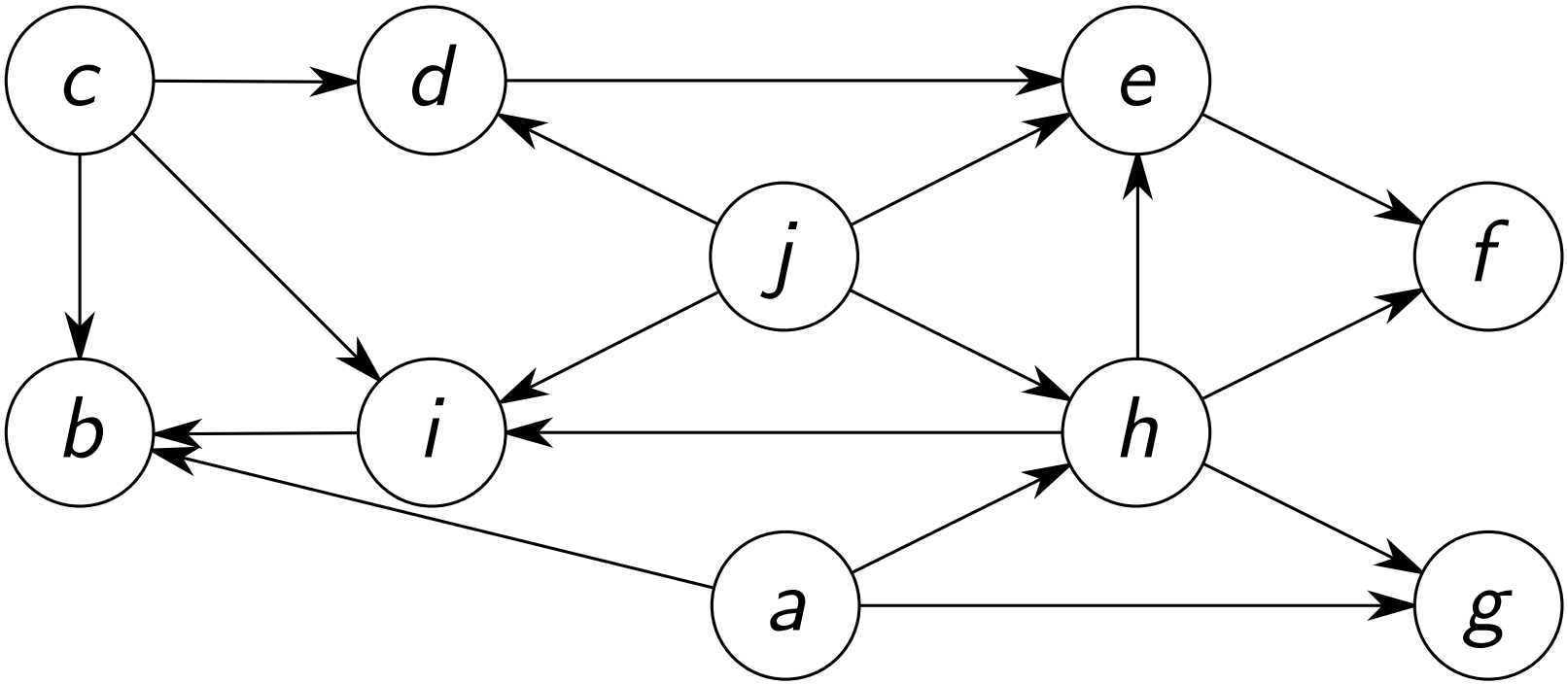
Exercise: Prove that a graph with cycles cannot be topologically sorted.

Topological Sorting (6)



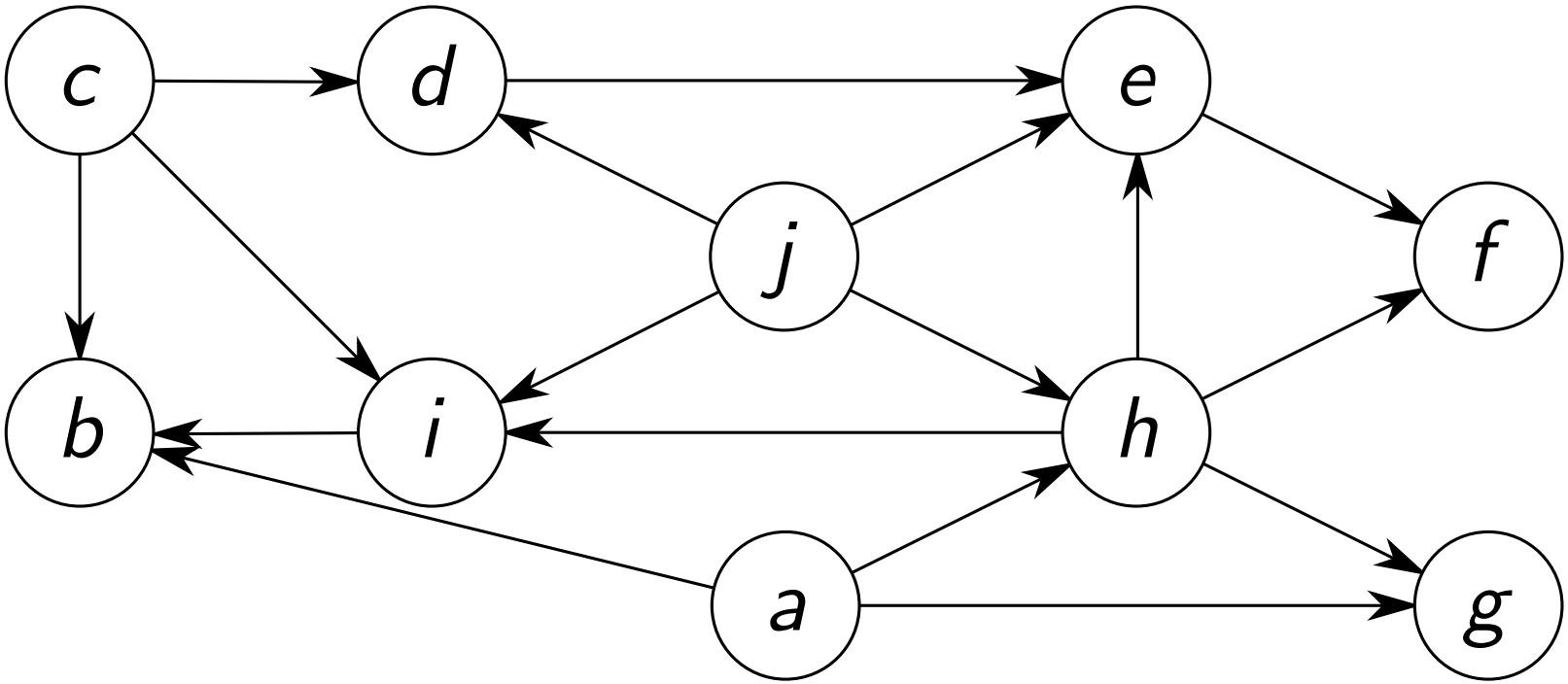
Exercise: Prove that a directed acyclic graph must have at least one topologically sorted ordering.

Topological Sorting Algorithms (1)



Problem: Design an algorithm to output a topologically sorted ordering of the vertices of a graph G.

Topological Sorting Algorithms (2)



Idea: Remove minimum vertices iteratively until no vertices remain (similar to the course prerequisite example).

Topological Sorting Algorithms (3)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

The pseudocode above topologically sorts G by repeatedly searching for and removing minimum vertices.

Topological Sorting Algorithms (4)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

One vertex is deleted on each iteration of the outer loop (lines 3 - 14).

Topological Sorting Algorithms (5)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

Therefore, the outer loop runs for n iterations on a graph with n vertices.

Topological Sorting Algorithms (6)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

The inner loop (lines 6 - 11) is essentially linear search over the vertices of G, and requires $\Theta(|V(G)|)$ time.

Topological Sorting Algorithms (7)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

The total running time of TopologicalSortNaive is $\Theta(n^2)$ time in the worst case.

Topological Sorting Algorithms (8)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

Question: Can we do better than $\Theta(n^2)$ time?

Topological Sorting Algorithms (9)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

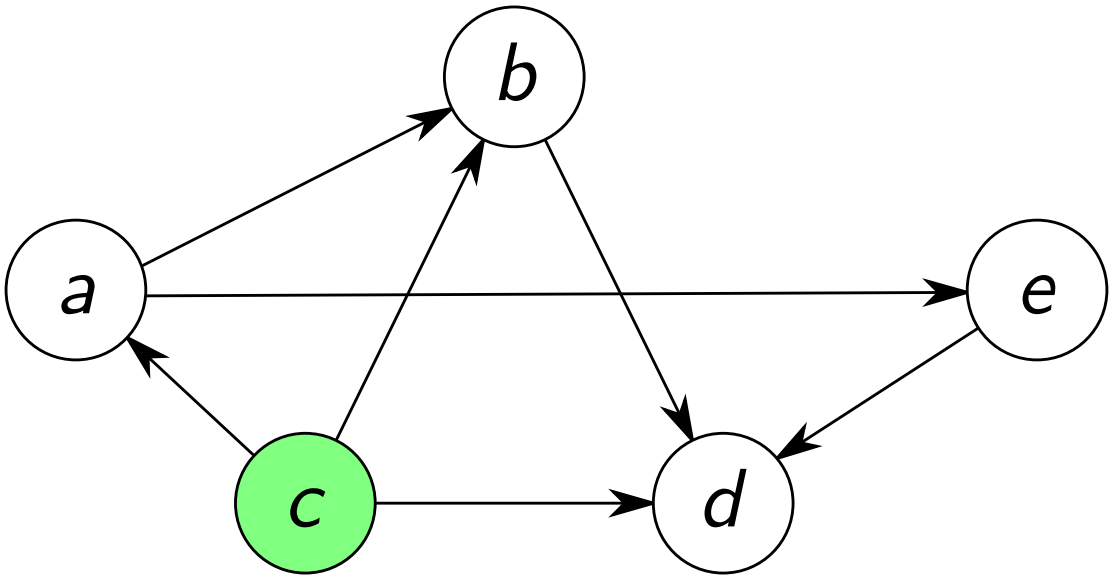
Any solution based on the above algorithm will require the same outer loop (which deletes vertices until none remain).

Topological Sorting Algorithms (10)

```
 1: procedure TopologicalSortNaive(G)
 2:     L ← Empty list
 3:     while |V(G)| > 0 do
 4:         //Find a minimum vertex v
 5:         v ← null
 6:         for each vertex w ∈ V(G) do
 7:             if w has no incoming edges then
 8:                 v ← w
 9:                 Break
10:             end if
11:         end for
12:         Add v to the end of L
13:         Delete v from G
14:     end while
15:     return L
16: end procedure
```

The most obvious area to improve is then the inner loop, which finds the next minimum vertex.

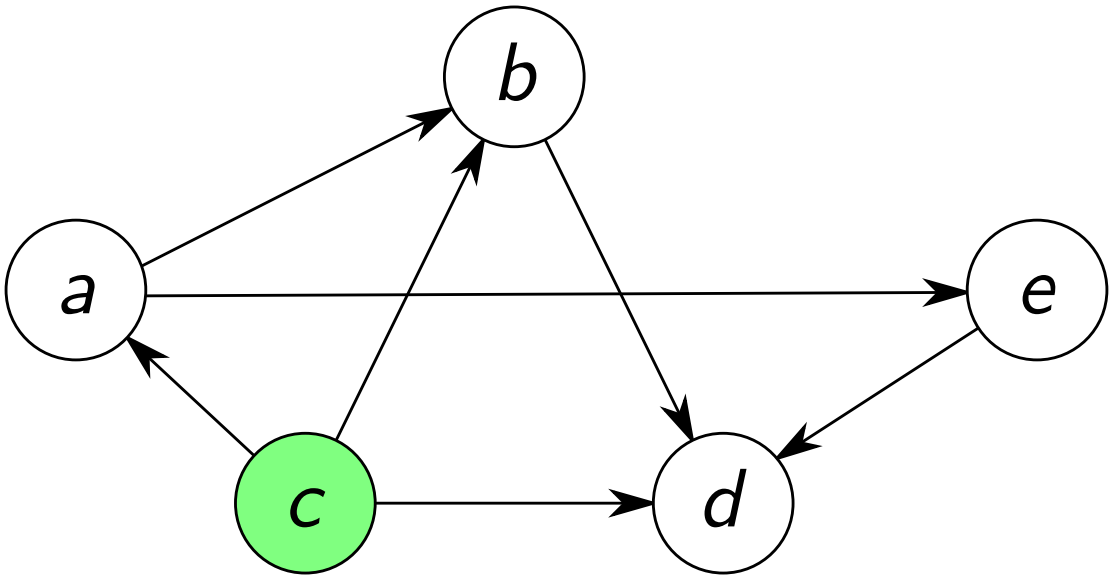
Topological Sorting Algorithms (11)



Observation: Except for the initial minimum vertices, each vertex will only become minimum when one of its neighbours is deleted.

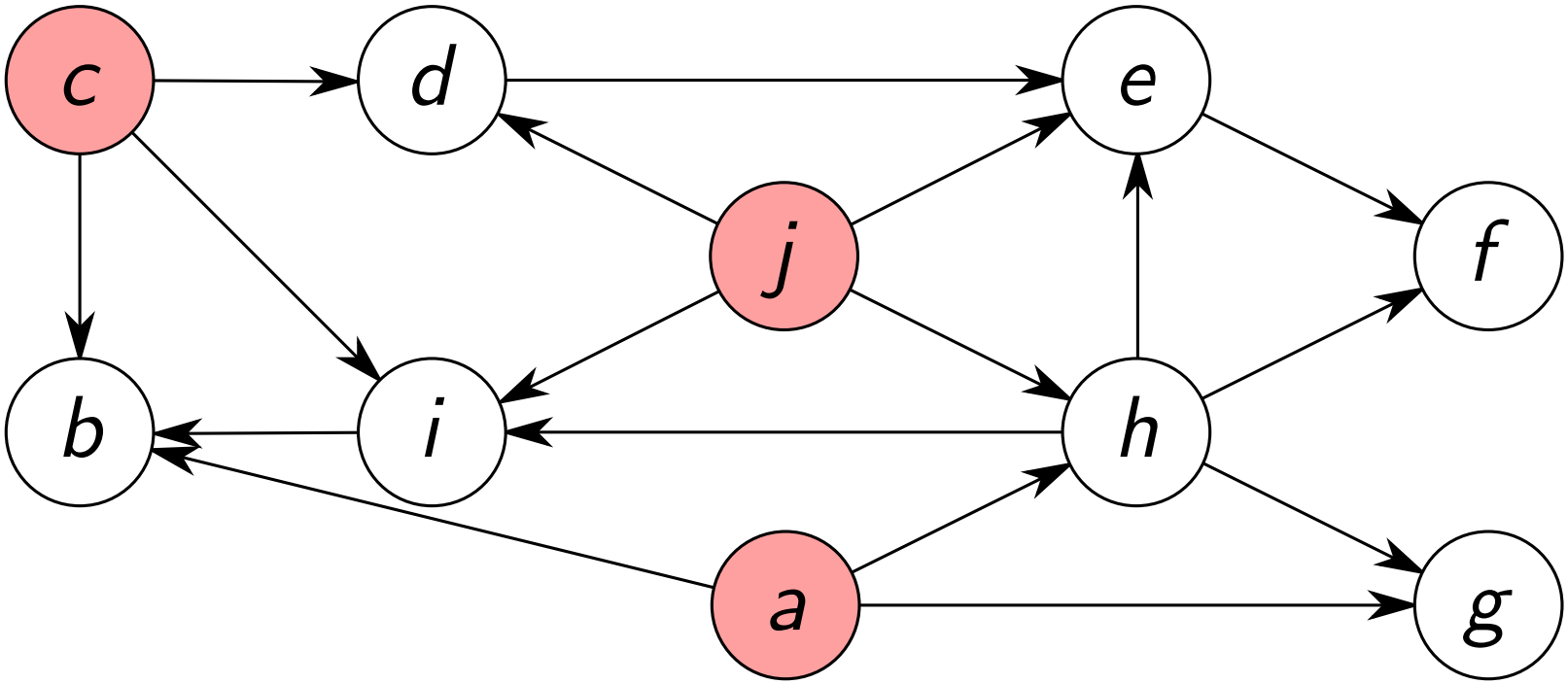
For example, vertex *a* above will become minimum when *c* is deleted.

Topological Sorting Algorithms (12)



Idea: When each vertex is deleted, check whether any neighbours became minimum vertices and if so, add them to a queue for processing.

Topological Sorting Algorithms (13)

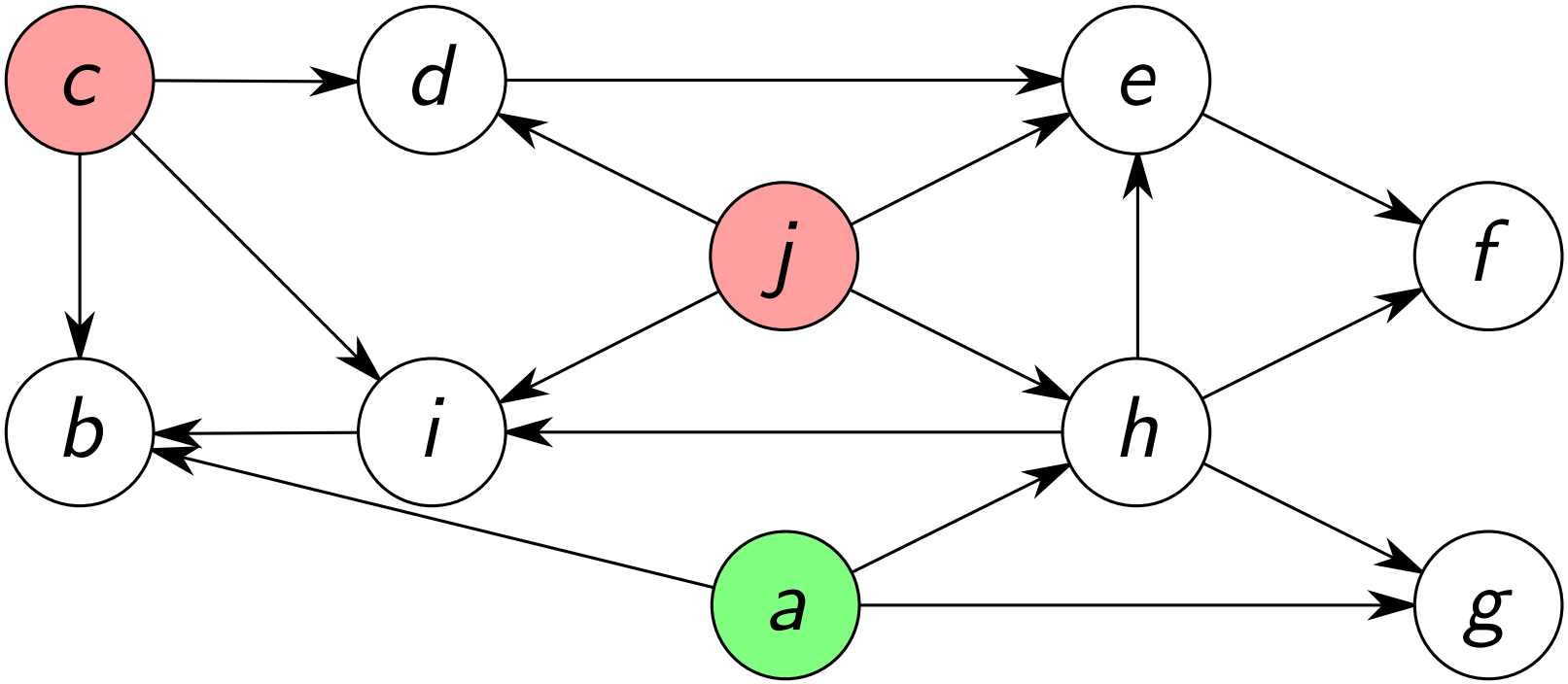


Queue: *a* *c* *j*

Ordering:

Start by adding all minimum vertices to the queue.

Topological Sorting Algorithms (14)

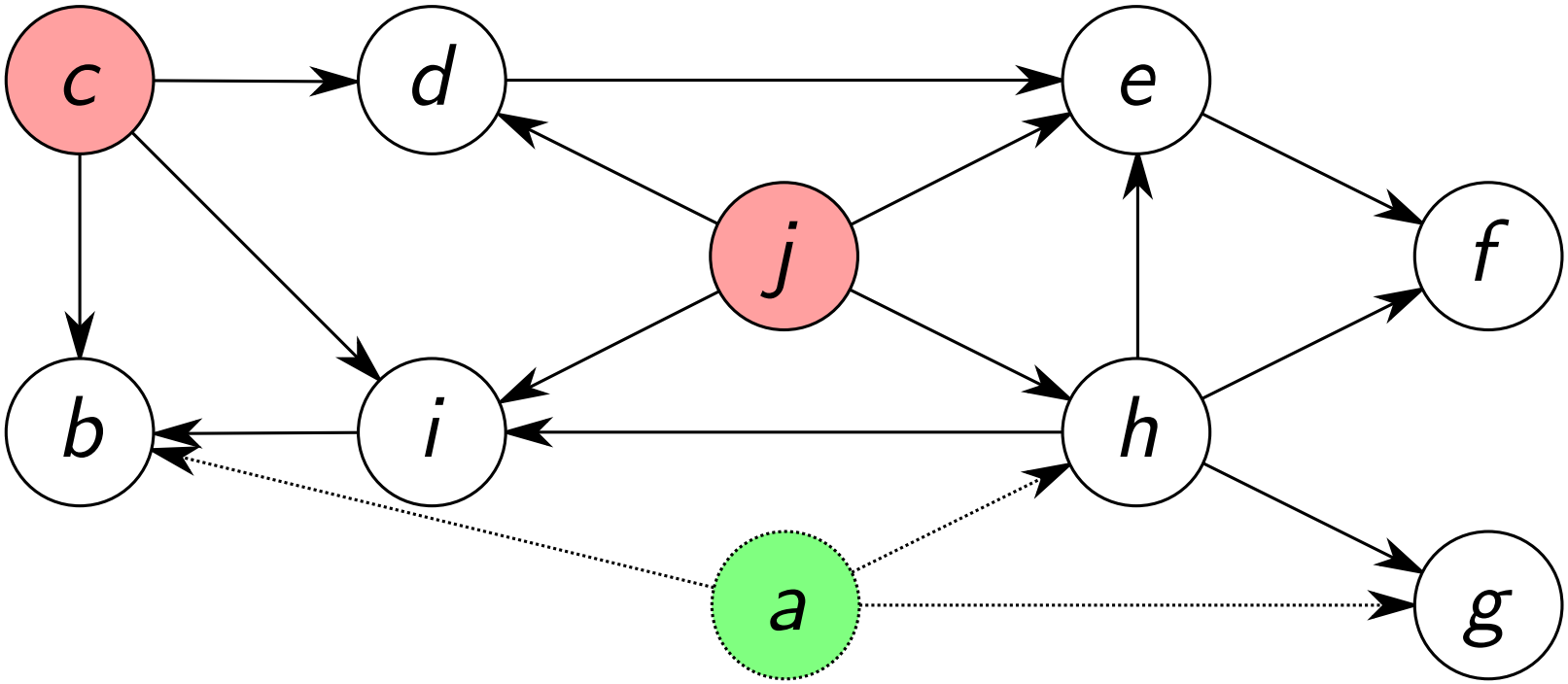


Queue: c j

Ordering: a

At each step, remove a vertex from the front of the queue and add it to the topologically sorted ordering.

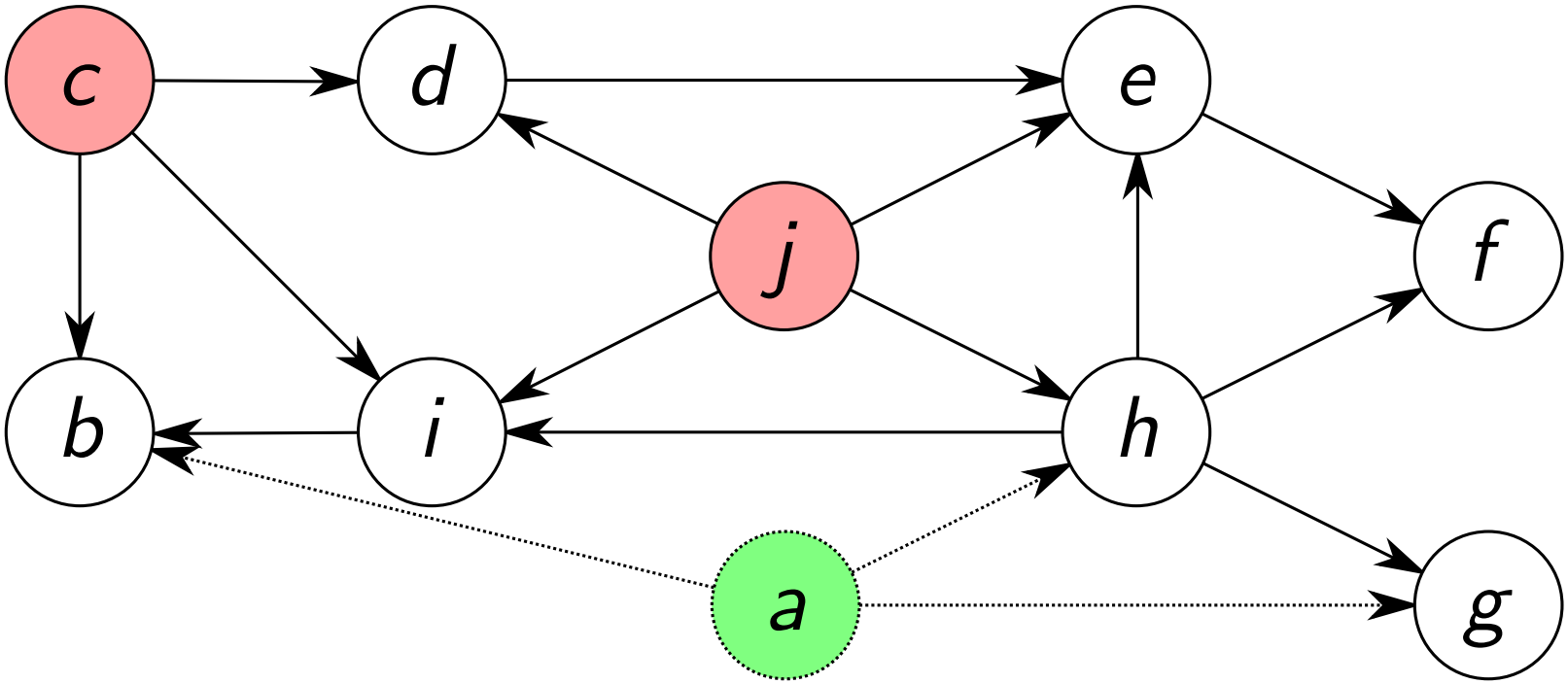
Topological Sorting Algorithms (15)



Queue: *c* *j*
Ordering: *a*

As each vertex is processed, delete it and all incident edges from the graph (or simulate deletion with a boolean flag).

Topological Sorting Algorithms (16)



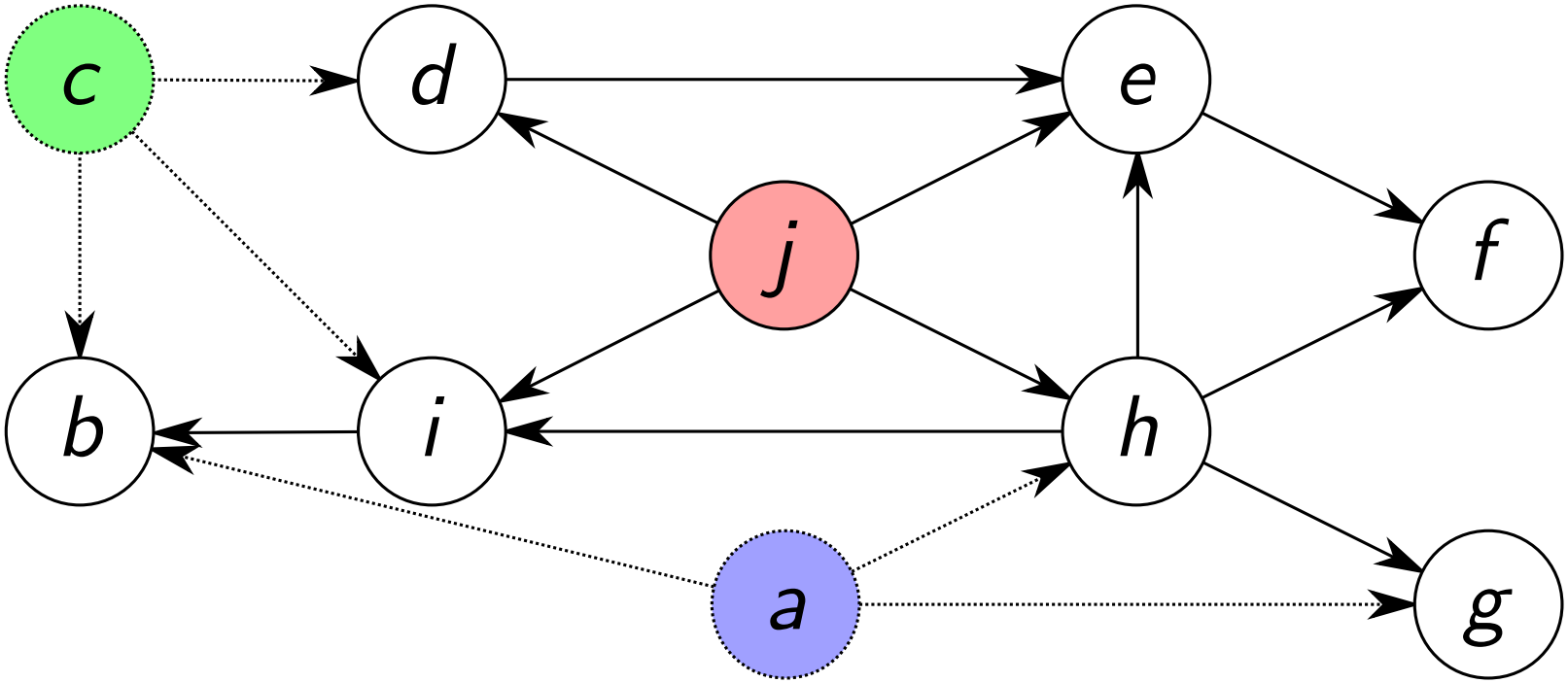
Queue: c j

Ordering: a

When a vertex is deleted, inspect all of its neighbours and add any new minimum vertices to the queue.

In this case, no new minimum vertices were created.

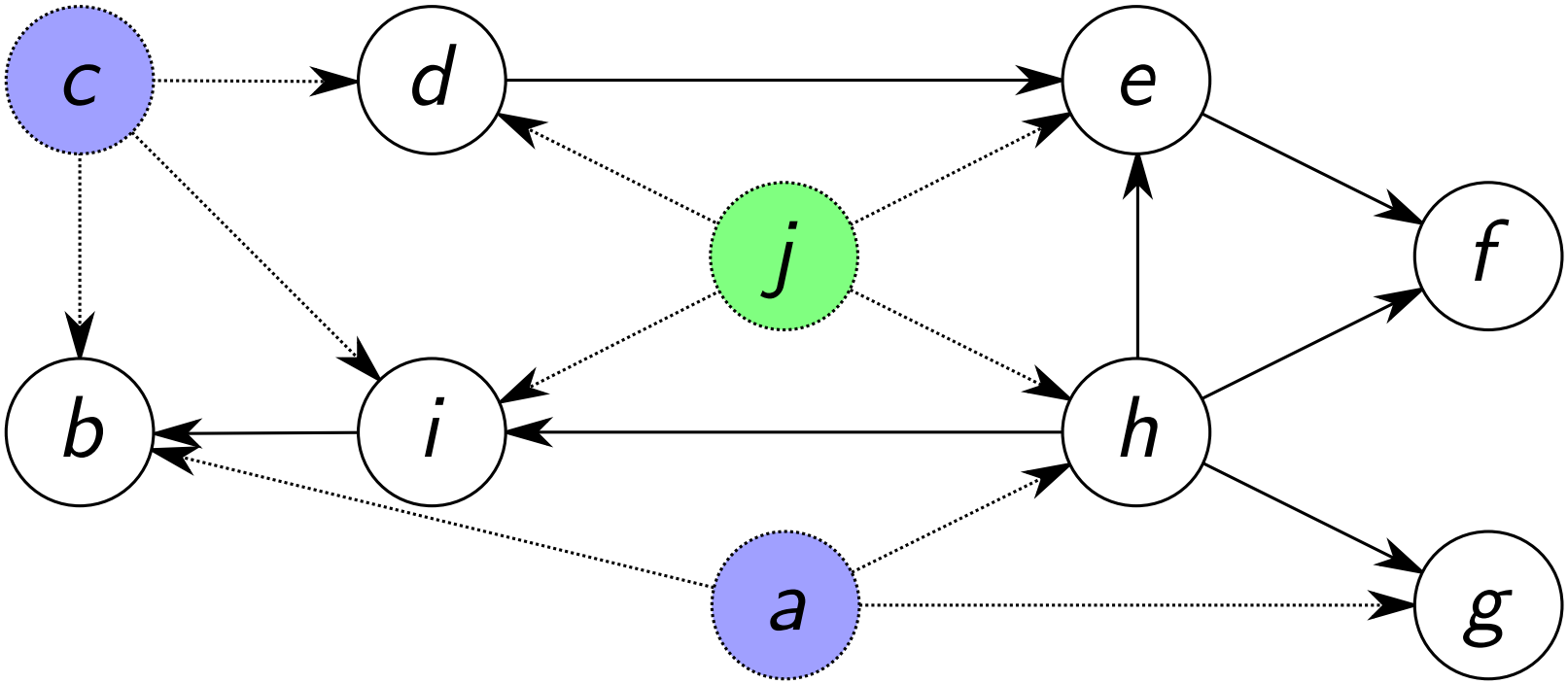
Topological Sorting Algorithms (17)



Queue: j

Ordering: a c

Topological Sorting Algorithms (18)

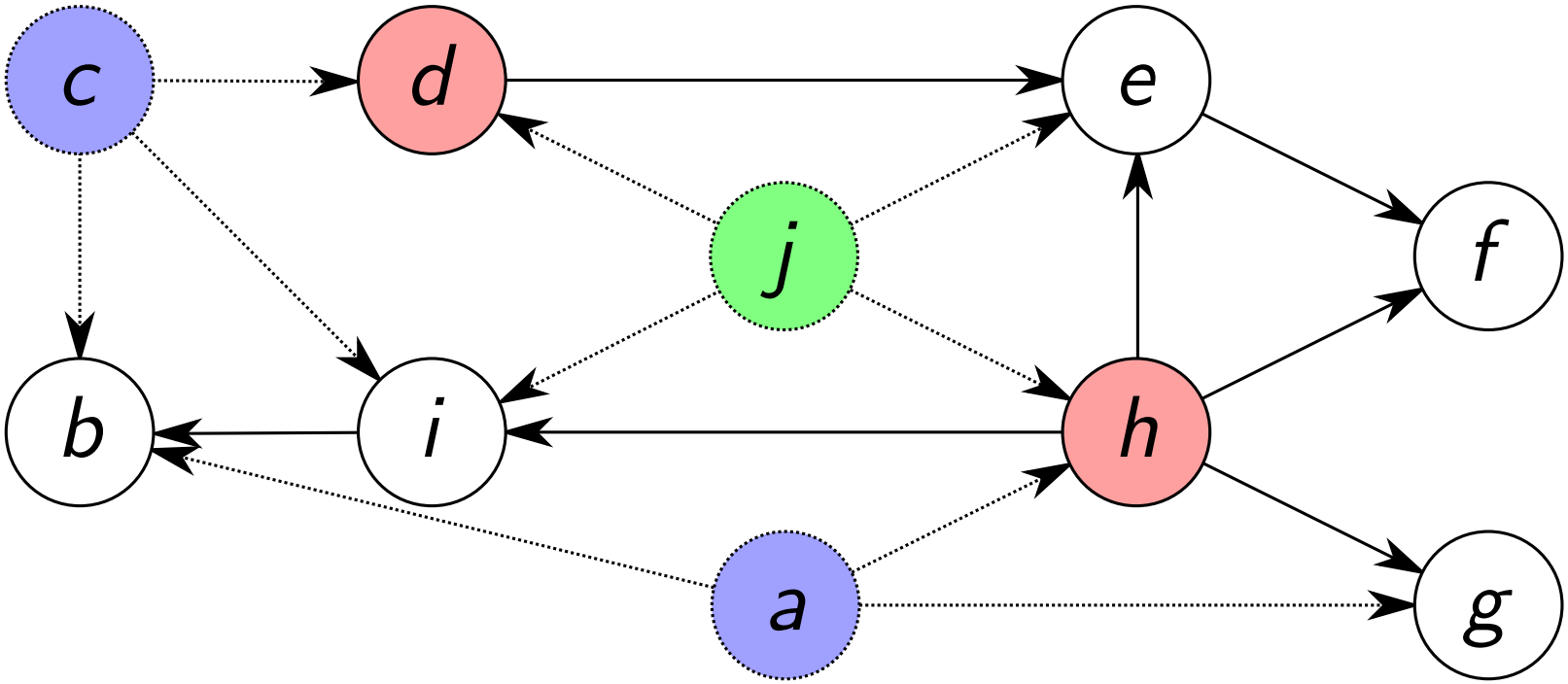


Queue:

Ordering: a c j

Deleting vertex j results in d and h becoming minimum vertices.

Topological Sorting Algorithms (19)

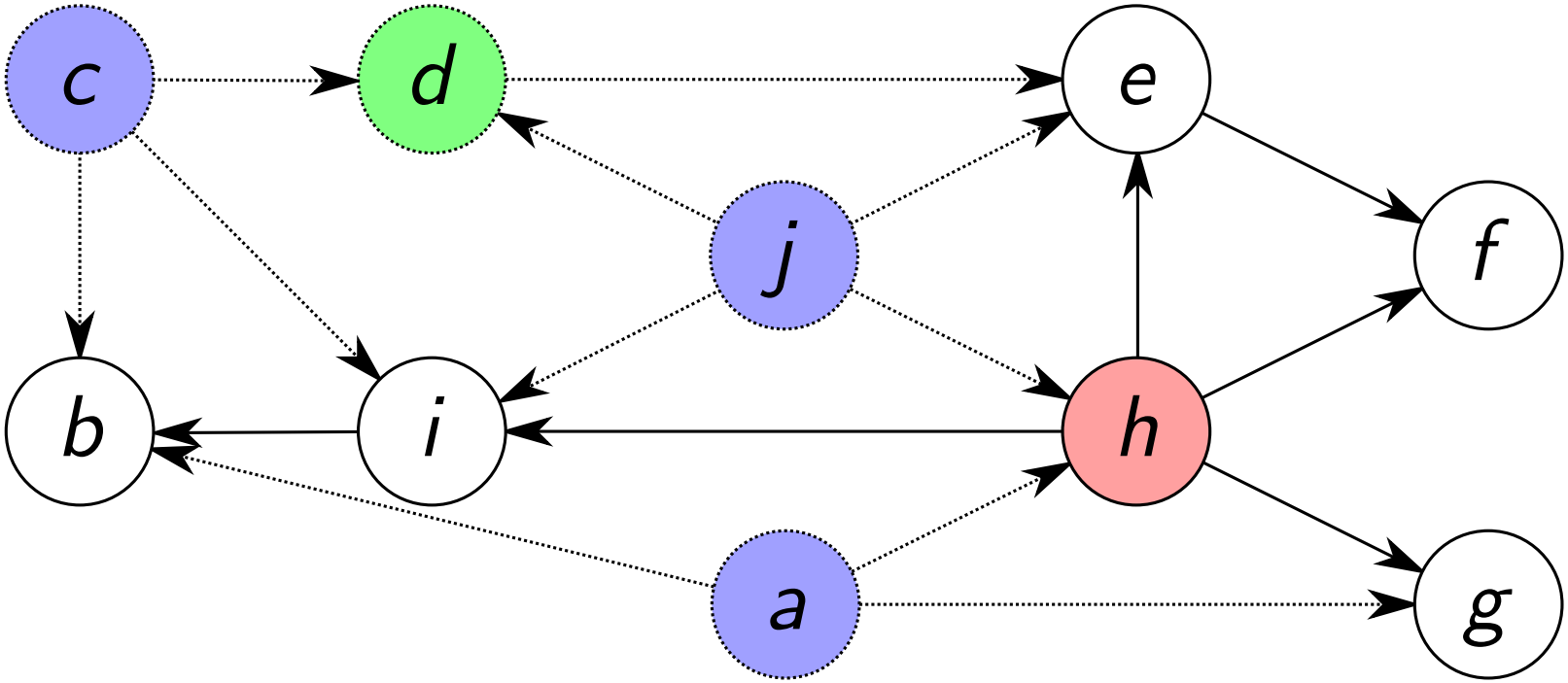


Queue: d h

Ordering: a c j

Vertices d and h are added to the queue.

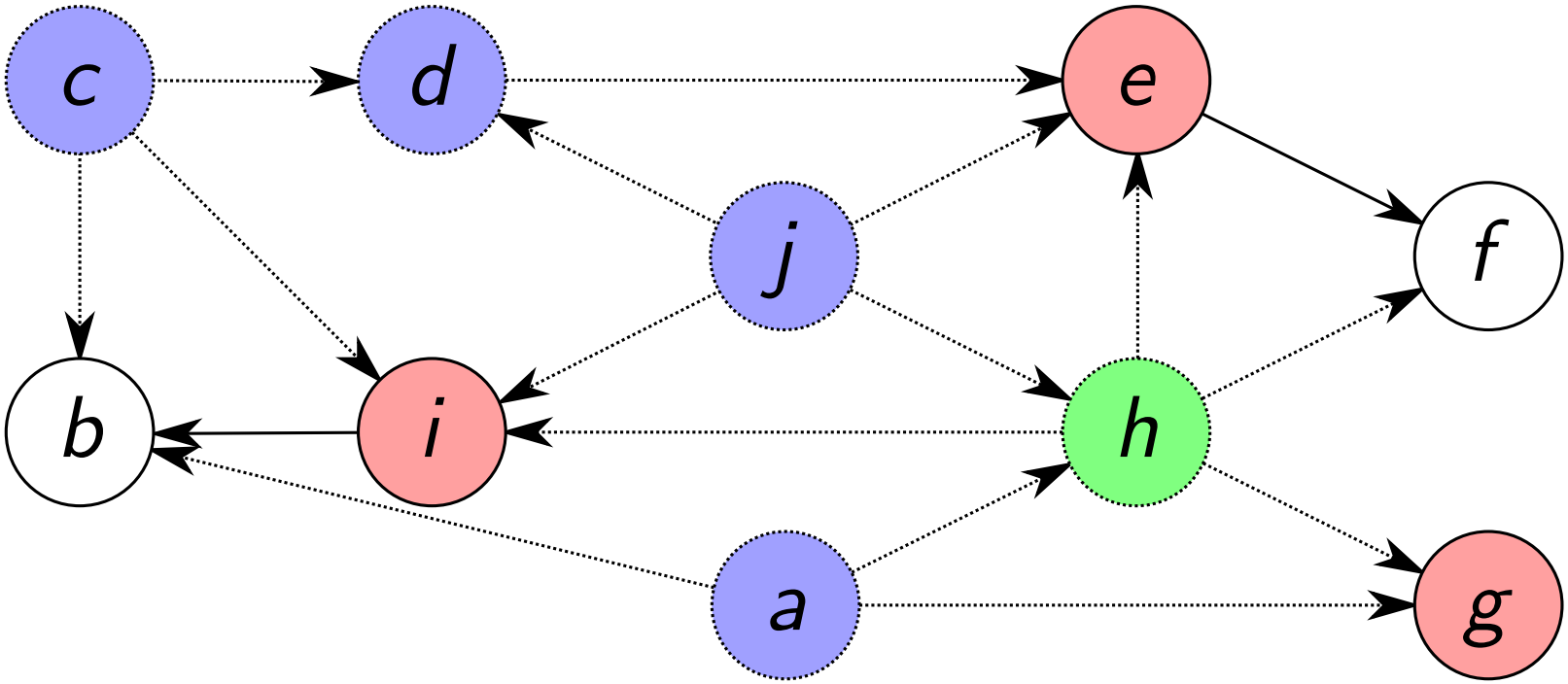
Topological Sorting Algorithms (20)



Queue: *h*

Ordering: *a* *c* *j* *d*

Topological Sorting Algorithms (21)

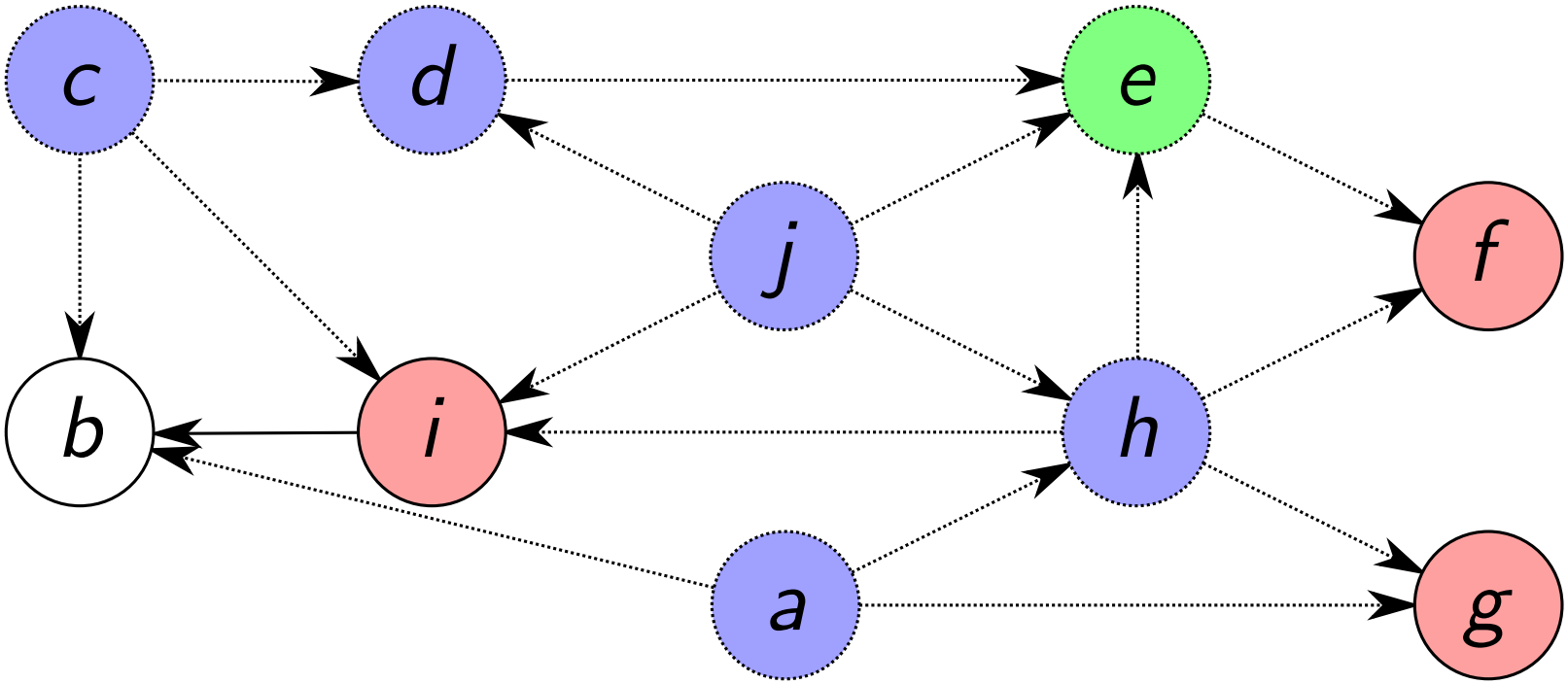


Queue: e i g

Ordering: a c j d h

When h is deleted, vertices e, i and g become minimum.

Topological Sorting Algorithms (22)

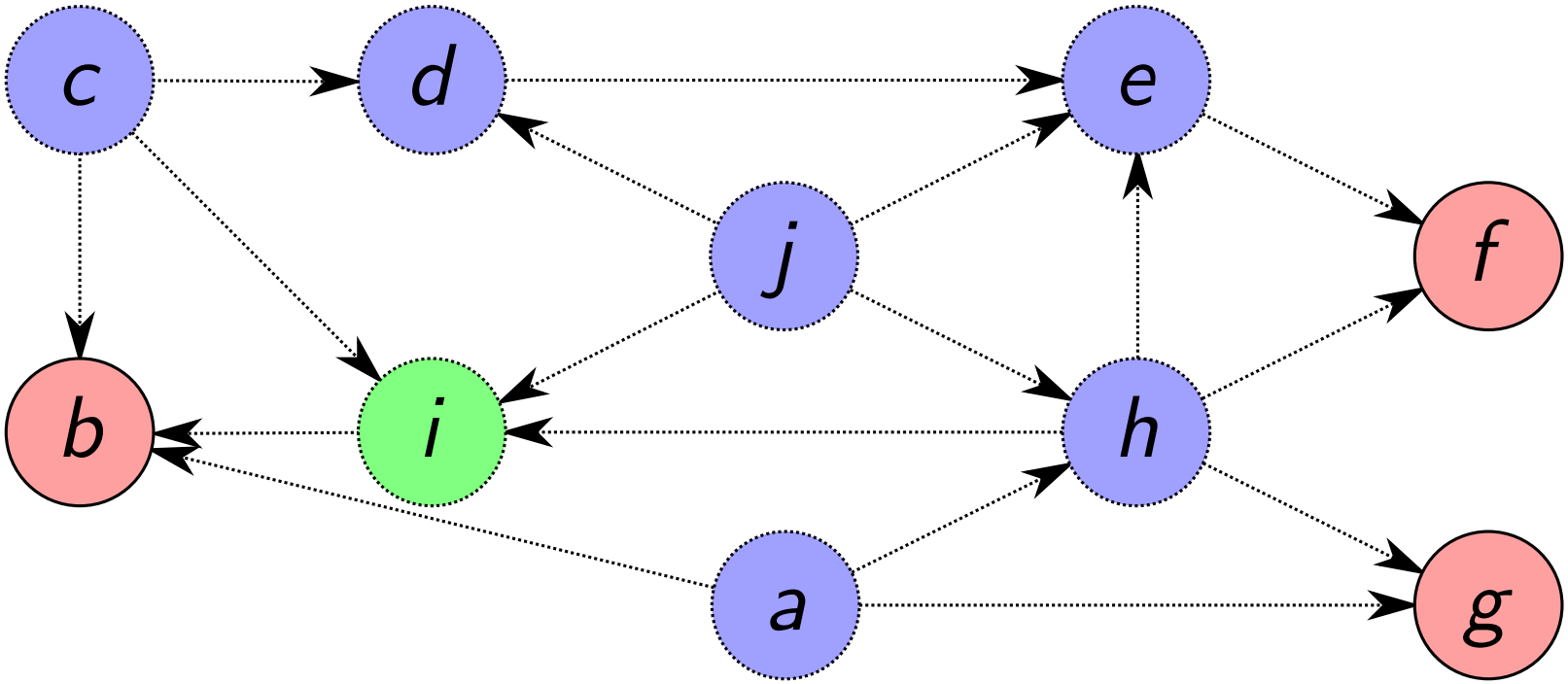


Queue: i g f

Ordering: a c j d h e

When e is deleted, vertex f becomes a minimum.

Topological Sorting Algorithms (23)

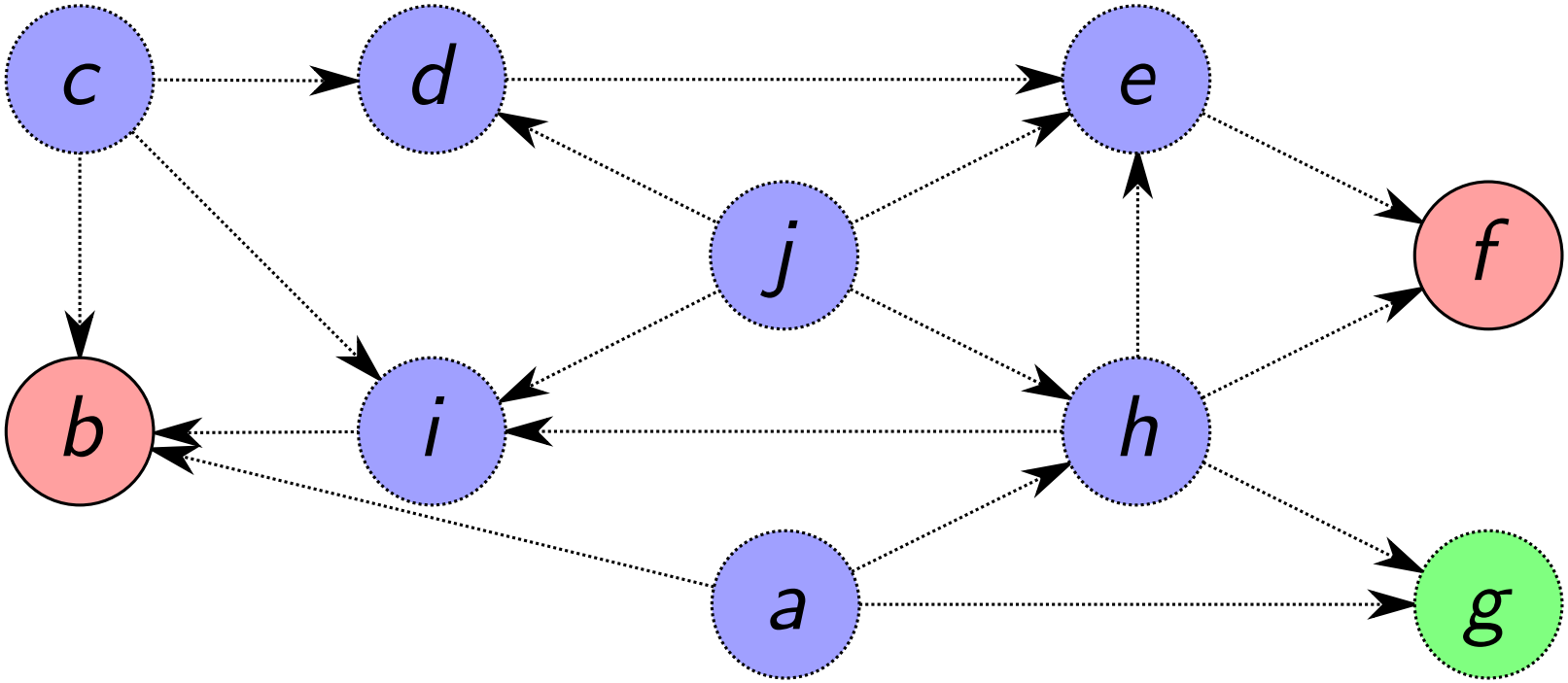


Queue: g f b

Ordering: a c j d h e i

When i is deleted, vertex b becomes a minimum.

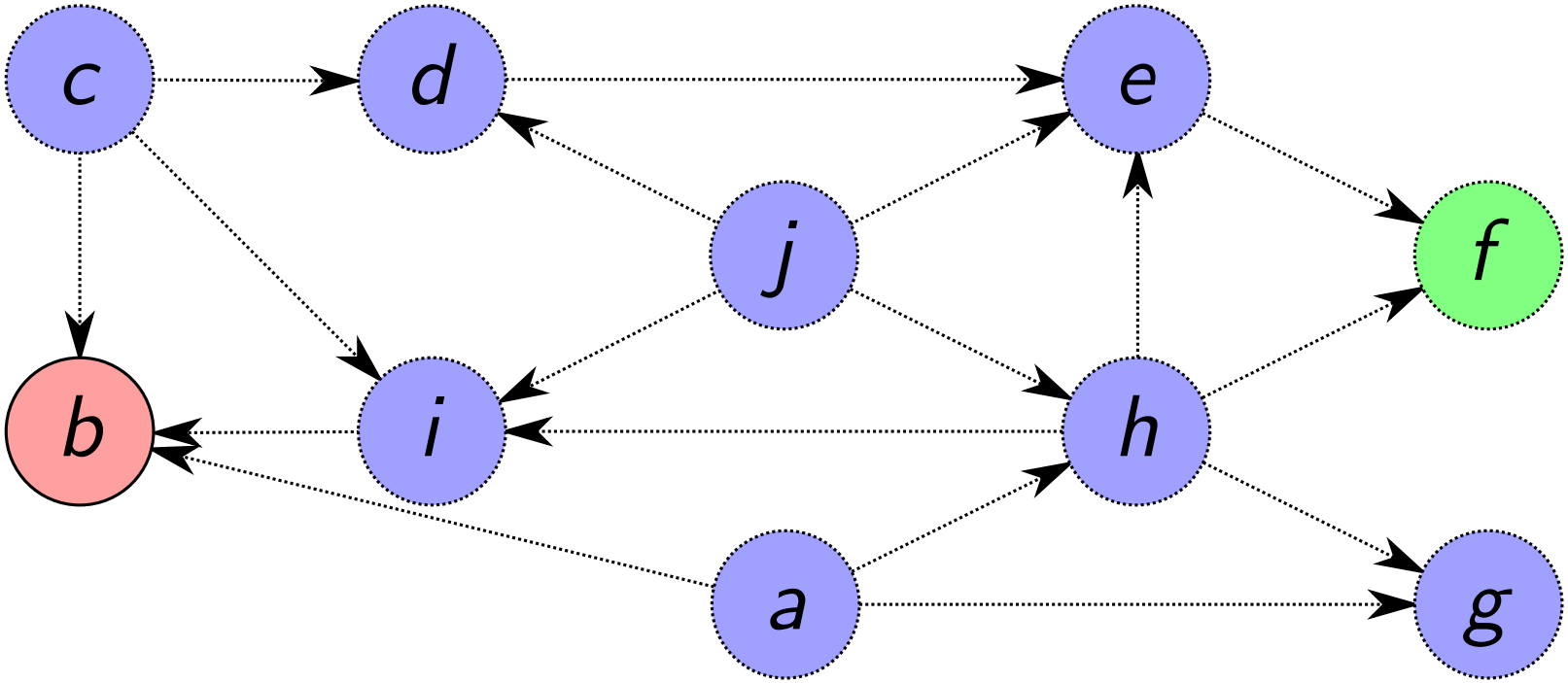
Topological Sorting Algorithms (24)



Queue: *f* *b*

Ordering: *a* *c* *j* *d* *h* *e* *i* *g*

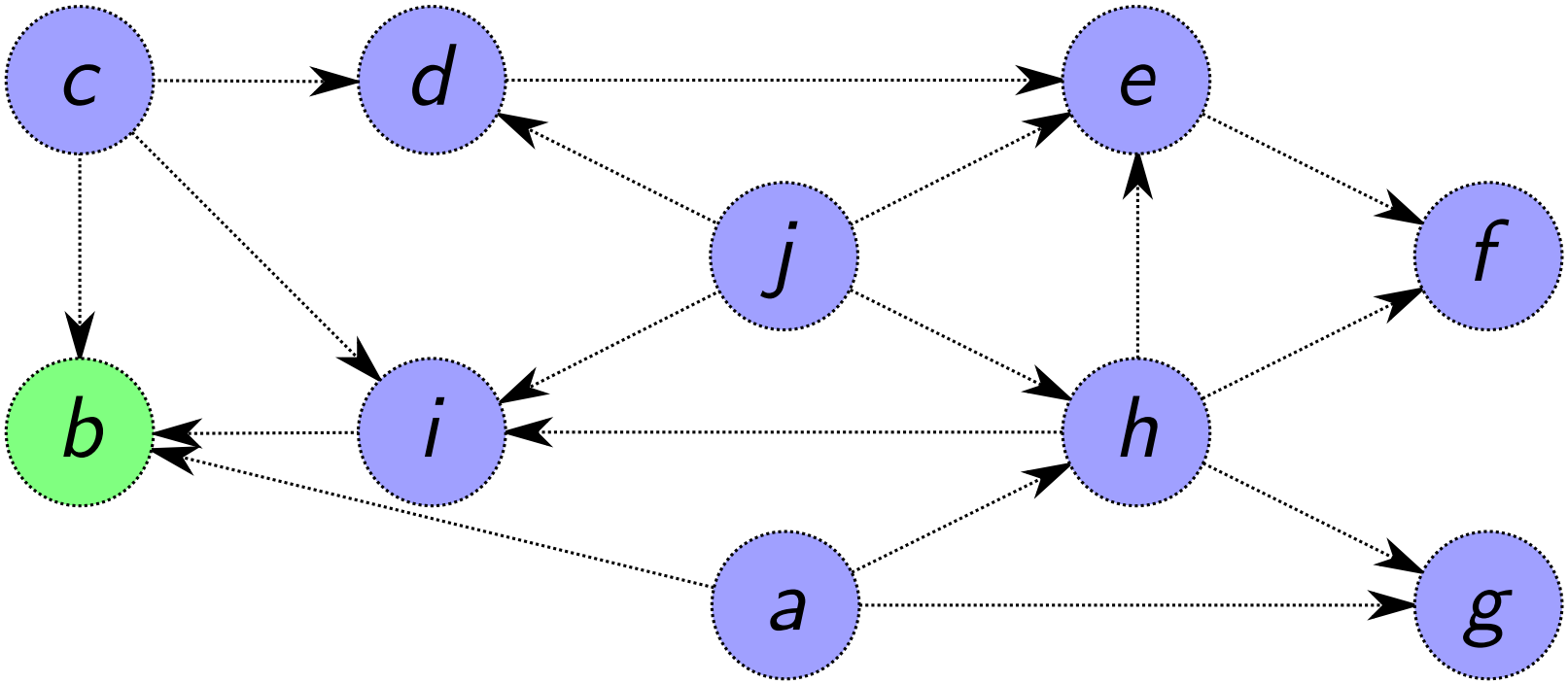
Topological Sorting Algorithms (25)



Queue: *b*

Ordering: *a* *c* *j* *d* *h* *e* *i* *g* *f*

Topological Sorting Algorithms (26)



Queue:

Ordering: *a* *c* *j* *d* *h* *e* *i* *g* *f* *b*

This algorithm is essentially BFS with one extra condition governing when a vertex can be visited.

Topological Sorting Algorithms (27)

```
 1: procedure TopologicalSortBFS(G)
 2:     L ← Empty list
 3:     Q ← Empty queue
 4:     for each vertex v ∈ V(G) do
 5:         if v has no incoming edges then
 6:             Enqueue v in Q
 7:         end if
 8:     end for
 9:     while Q is non-empty do
10:         v ← Dequeue(Q)
11:         for each neighbour w of v do
12:             if w has incoming degree 1 then
13:                 Enqueue w in Q
14:             end if
15:         end for
16:         Add v to the end of L
17:         Delete v from G
18:     end while
19:     return L
20: end procedure
```

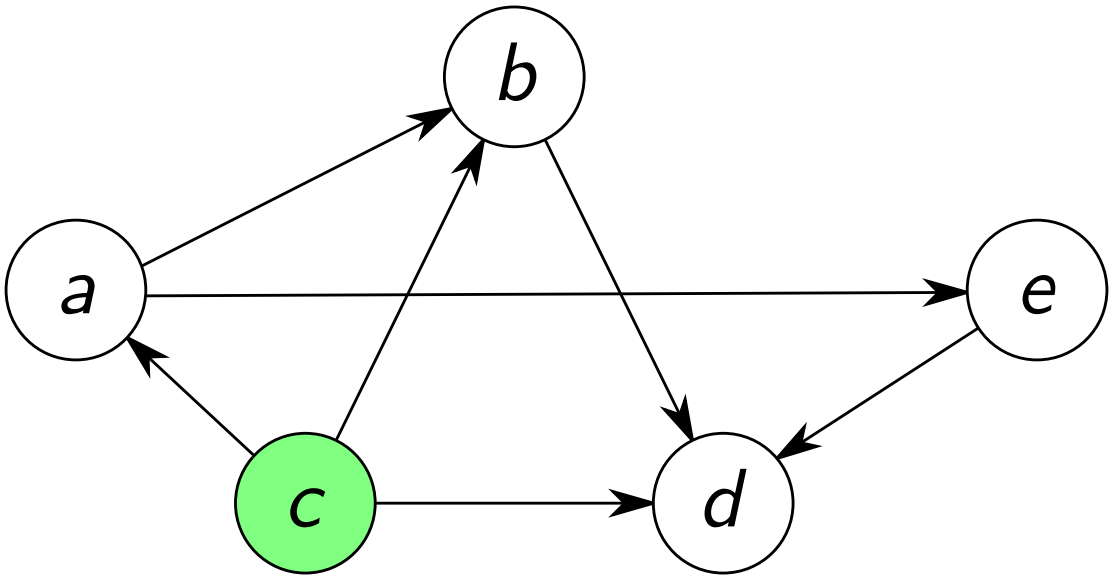
The BFS-like topological sorting algorithm is given by the pseudocode above.

Topological Sorting Algorithms (28)

```
 1: procedure TopologicalSortBFS(G)
 2:     L ← Empty list
 3:     Q ← Empty queue
 4:     for each vertex v ∈ V(G) do
 5:         if v has no incoming edges then
 6:             Enqueue v in Q
 7:         end if
 8:     end for
 9:     while Q is non-empty do
10:         v ← Dequeue(Q)
11:         for each neighbour w of v do
12:             if w has incoming degree 1 then
13:                 Enqueue w in Q
14:             end if
15:         end for
16:         Add v to the end of L
17:         Delete v from G
18:     end while
19:     return L
20: end procedure
```

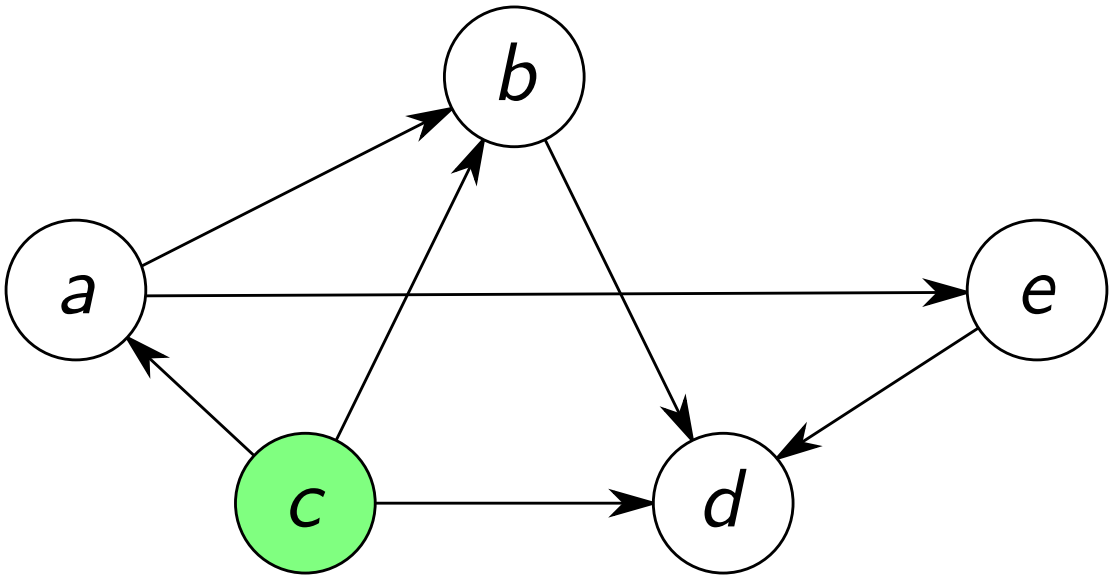
The running time of this variant is $\Theta(n + m)$, like BFS.

Topological Sorting With DFS (1)



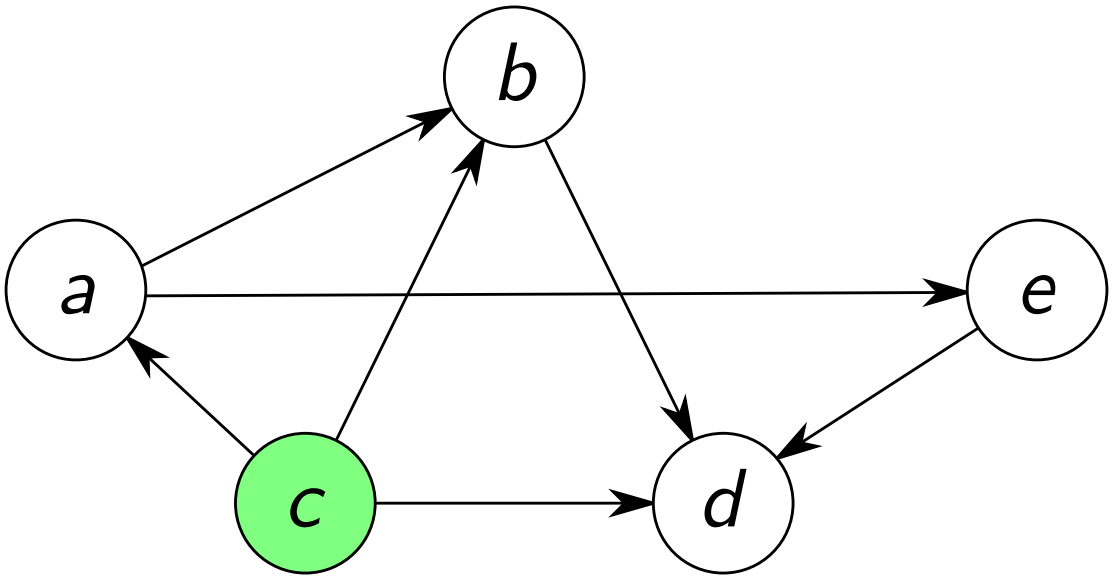
Topological sorting is also possible with DFS, by leveraging an observation about post-order numberings.

Topological Sorting With DFS (2)



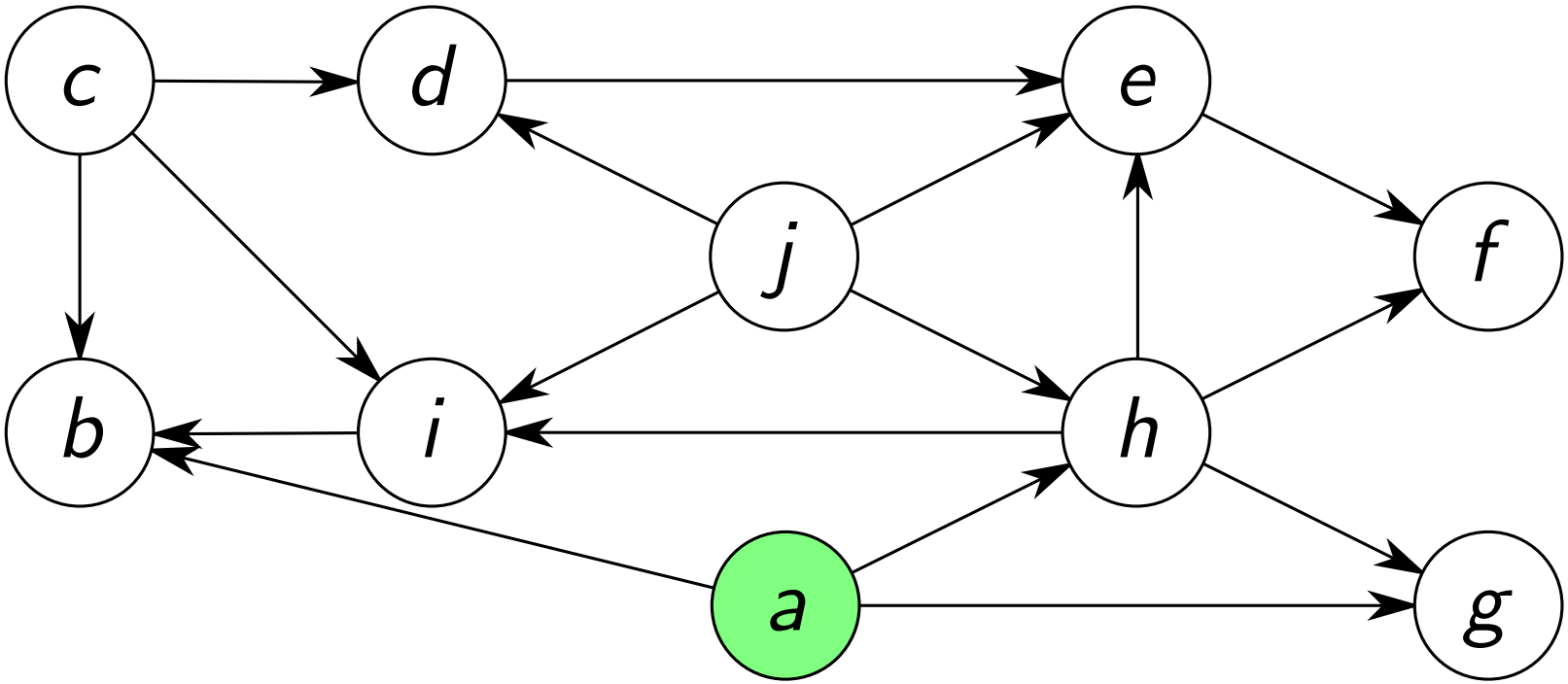
Observation: A DFS traversal will not unwind from a vertex v until all vertices reachable from v have been visited.

Topological Sorting With DFS (3)



Therefore, when DFS unwinds from a vertex v, all vertices which should follow v in a topologically sorted order have been visited.

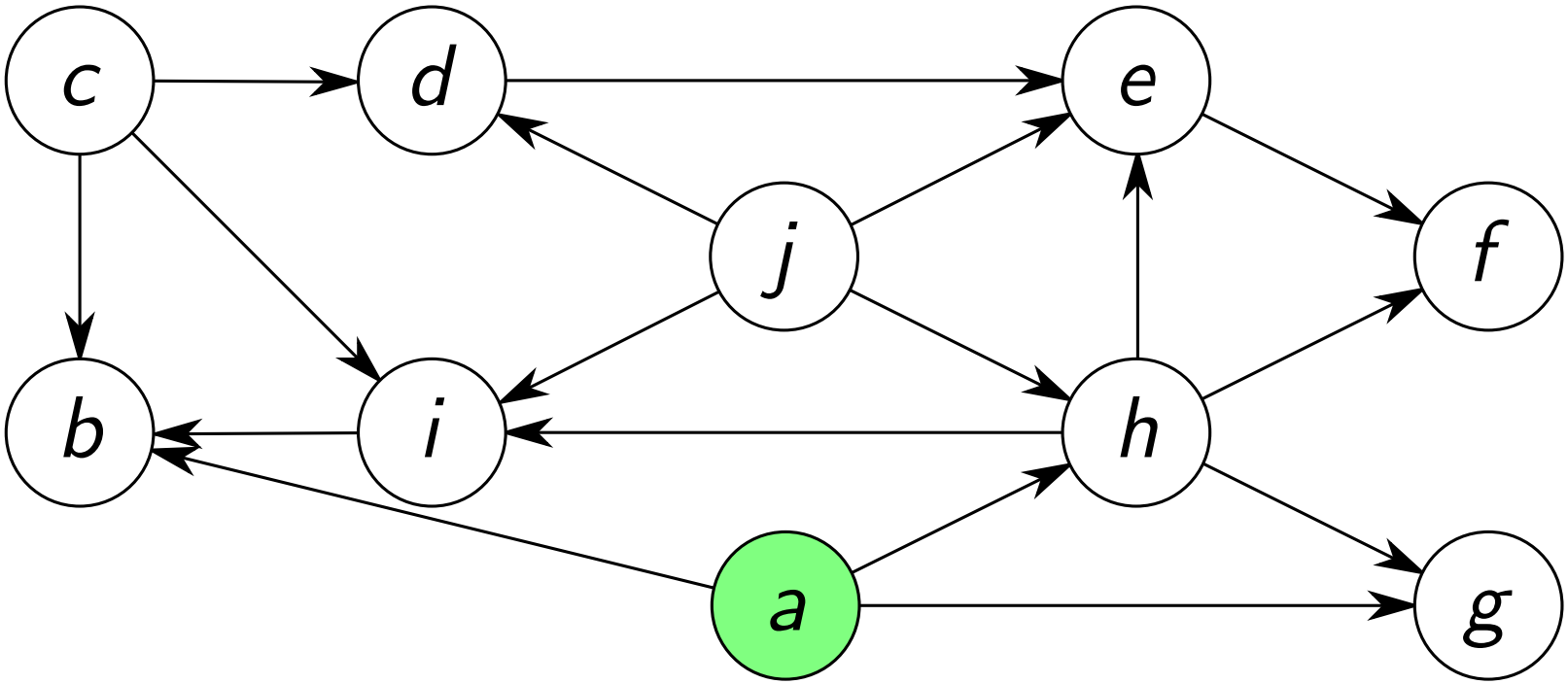
Topological Sorting With DFS (4)



Ordering:

The DFS-based topological sorting algorithm constructs the ordering in reverse.

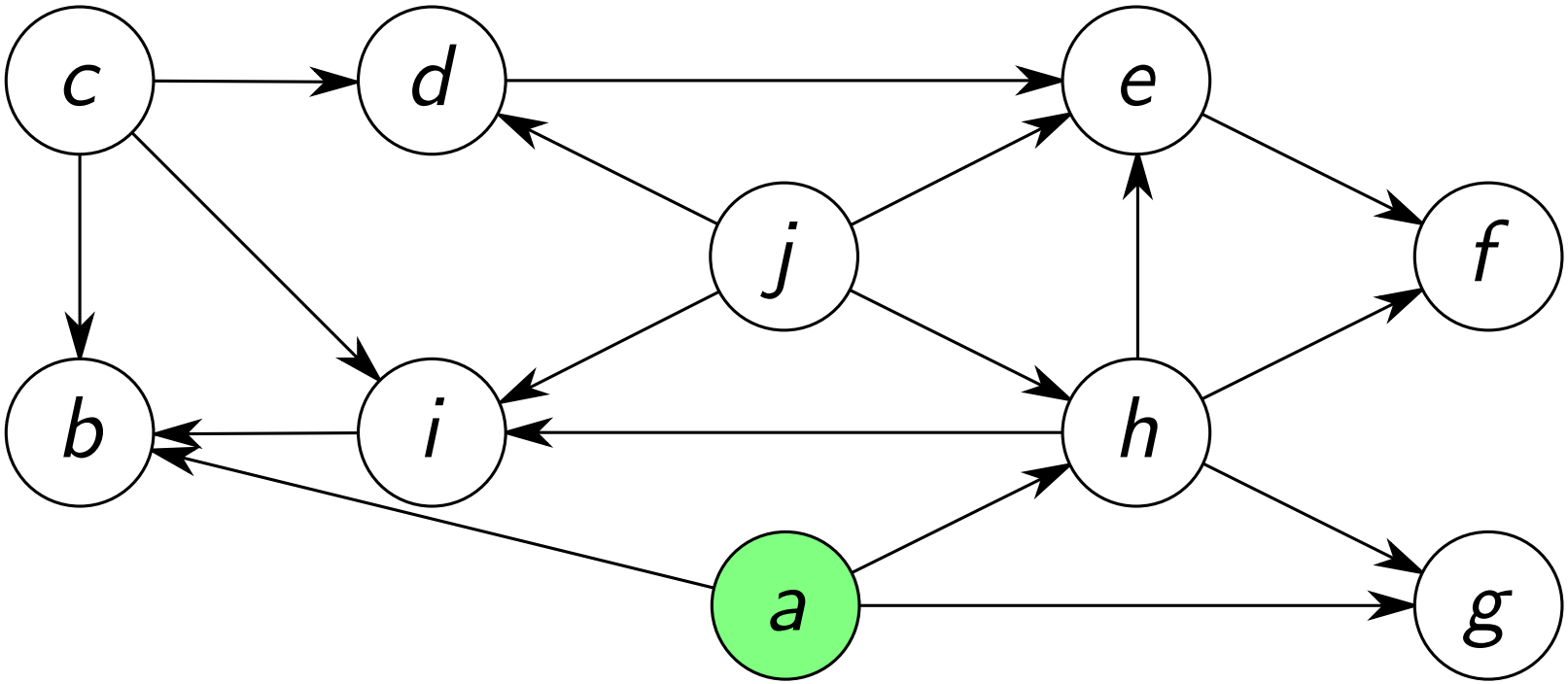
Topological Sorting With DFS (5)



Ordering:

First, DFS is started on an arbitrary vertex (not necessarily a minimum).

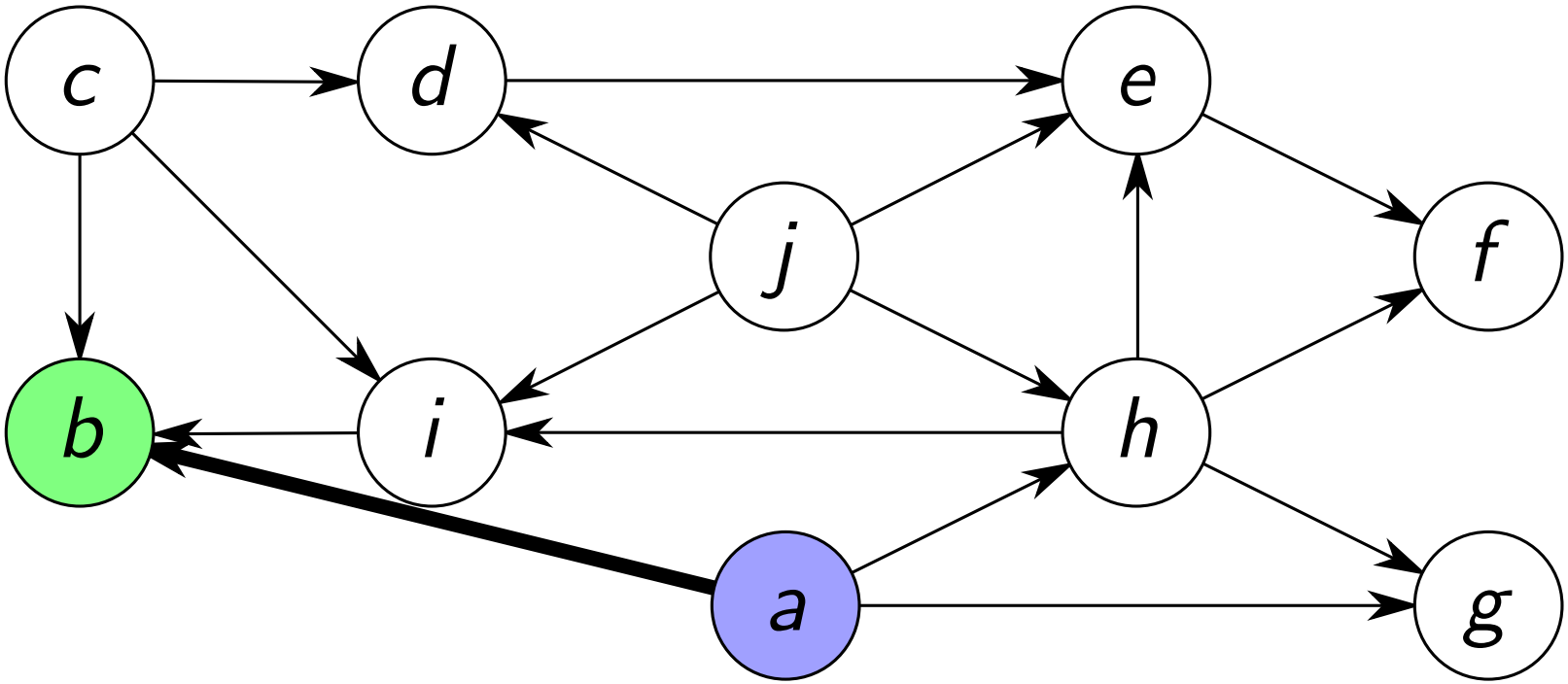
Topological Sorting With DFS (6)



Ordering:

First, DFS is started on an arbitrary vertex (not necessarily a minimum).

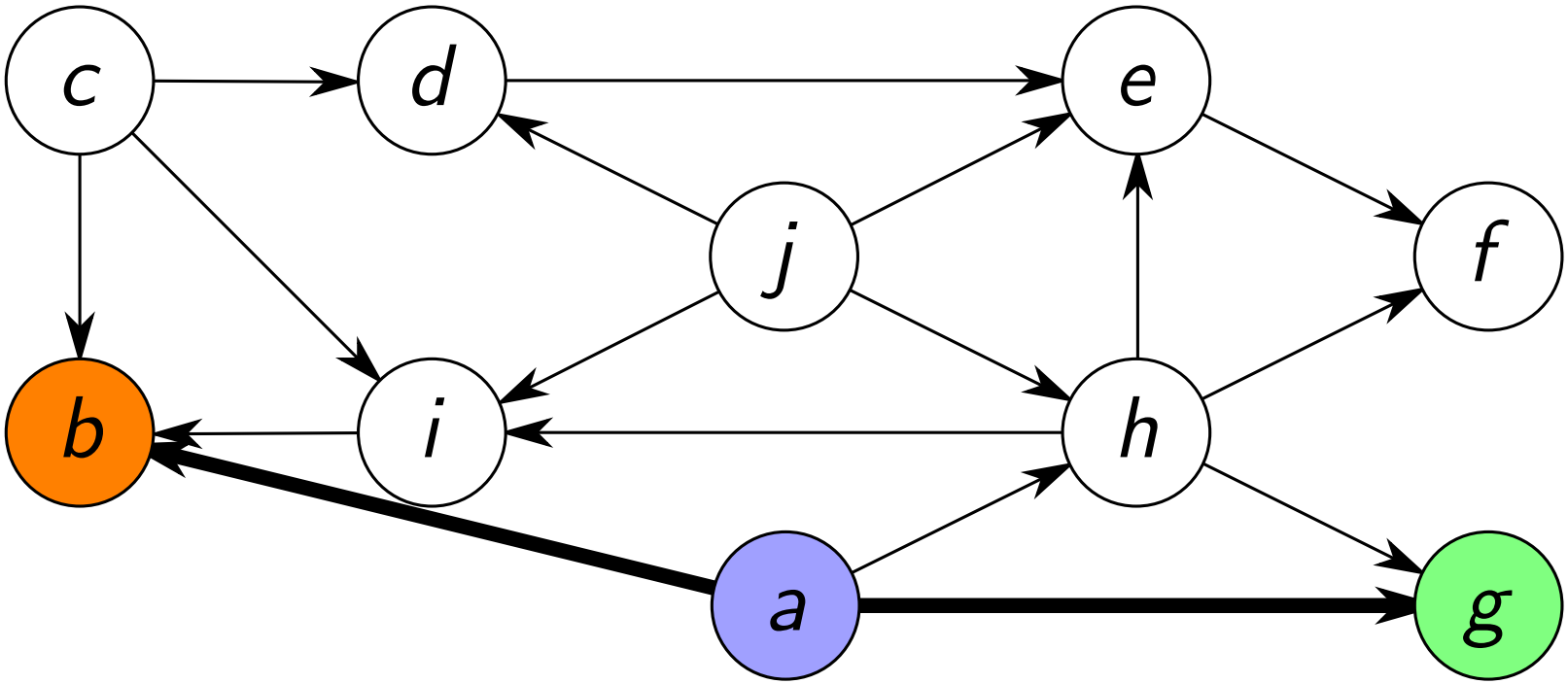
Topological Sorting With DFS (7)



Ordering:

In the diagram above, the vertex being visited is green and other vertices in the same branch of the DFS tree are blue.

Topological Sorting With DFS (8)

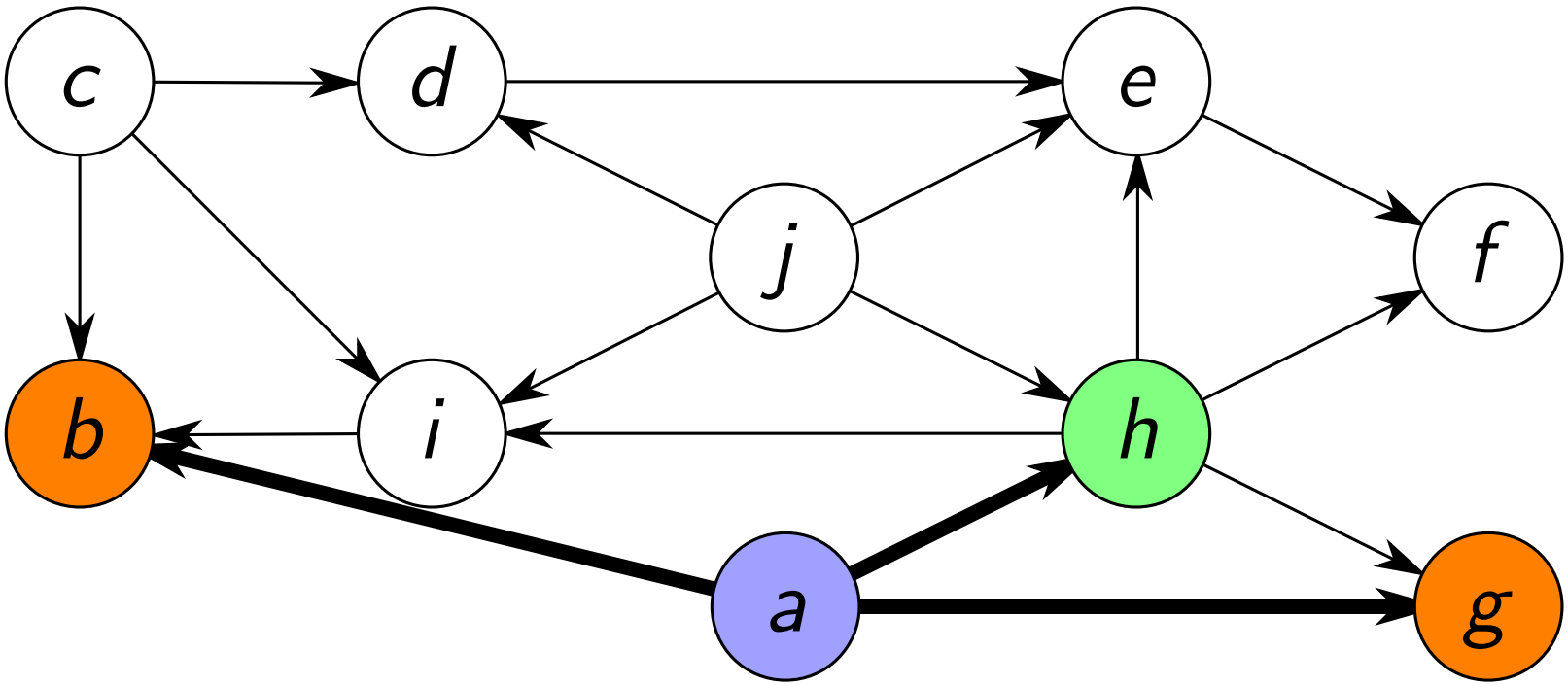


Ordering: b

Vertices are added to the **beginning** of the ordering when DFS unwinds from them.

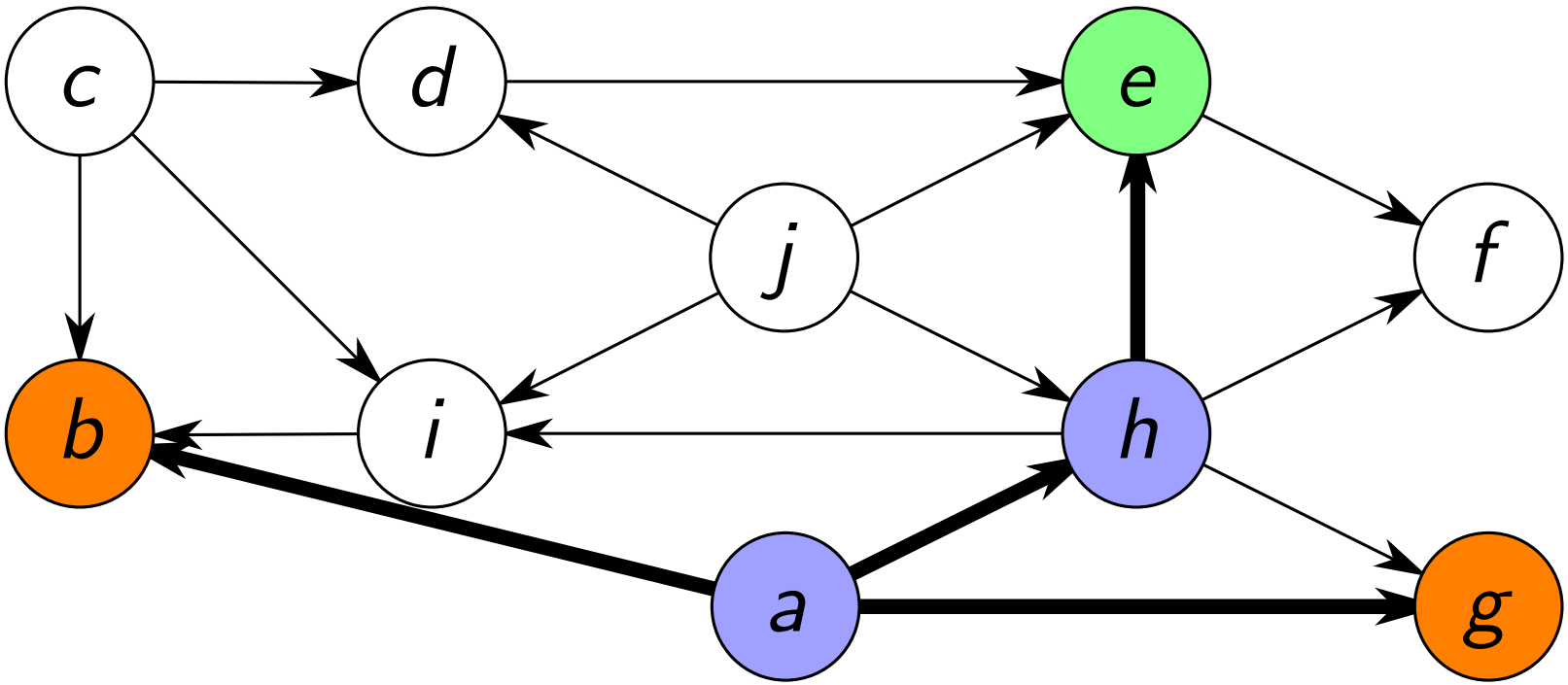
In the diagram, vertices which have been visited and added to the ordering are coloured orange.

Topological Sorting With DFS (9)



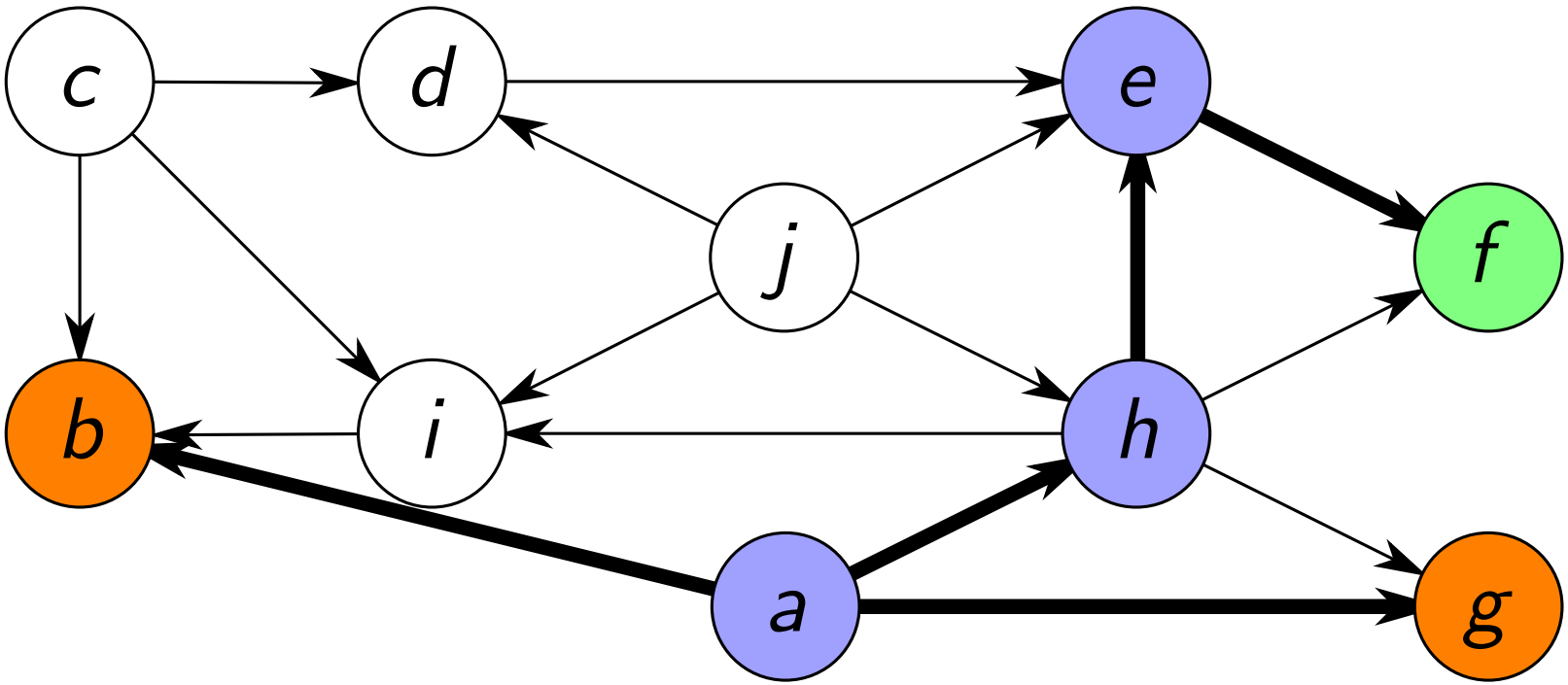
Ordering: g b

Topological Sorting With DFS (10)



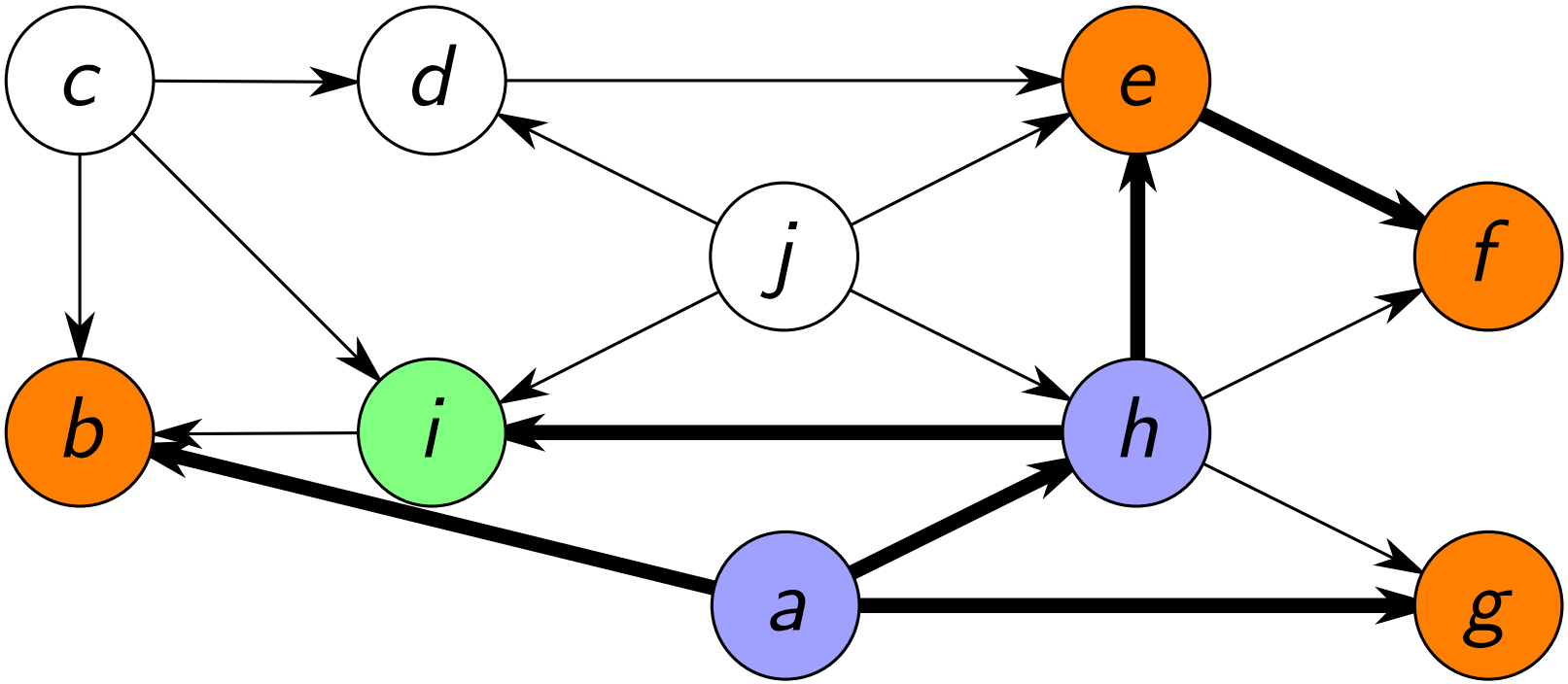
Ordering: g b

Topological Sorting With DFS (11)



Ordering: g b

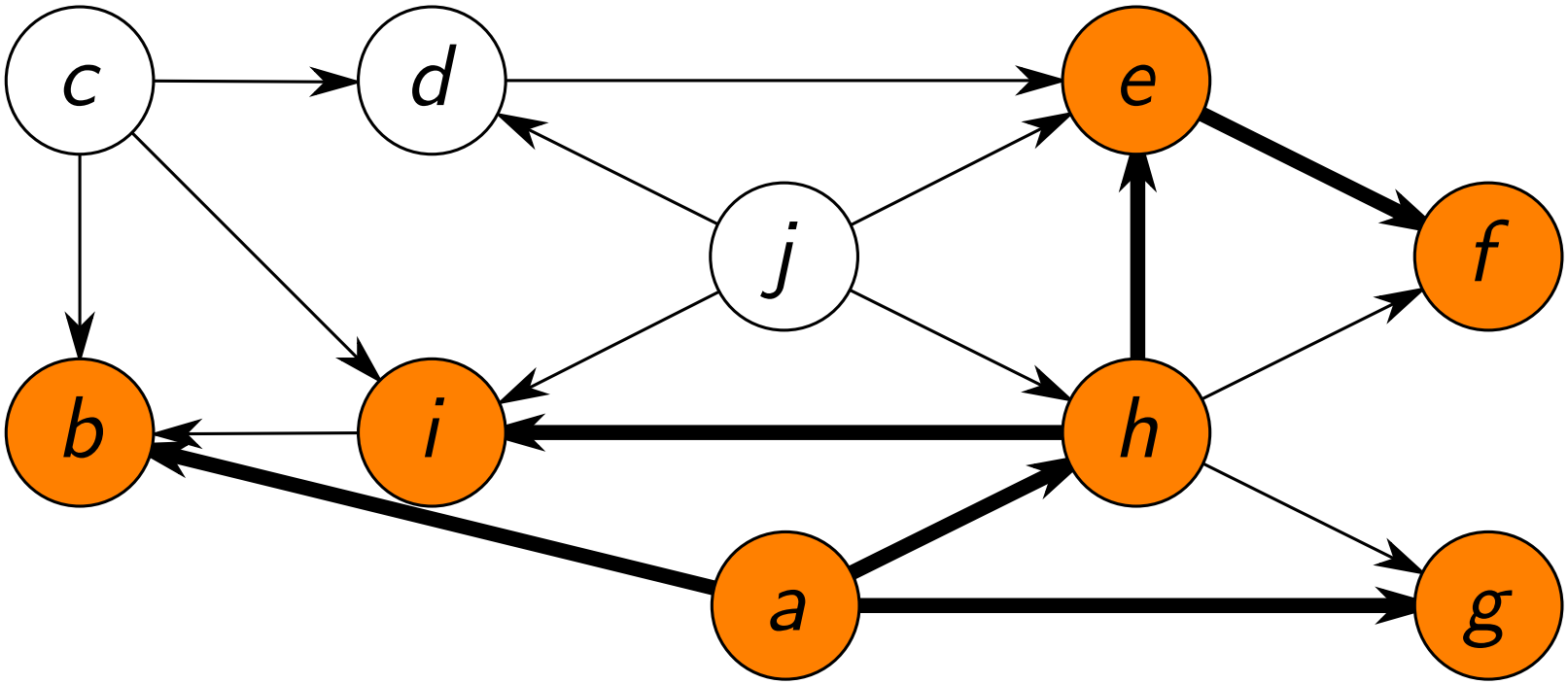
Topological Sorting With DFS (12)



Ordering: e f g b

After finishing at f, DFS unwinds (and f is added to the ordering). Since e has no unvisited neighbours left, DFS unwinds from e as well.

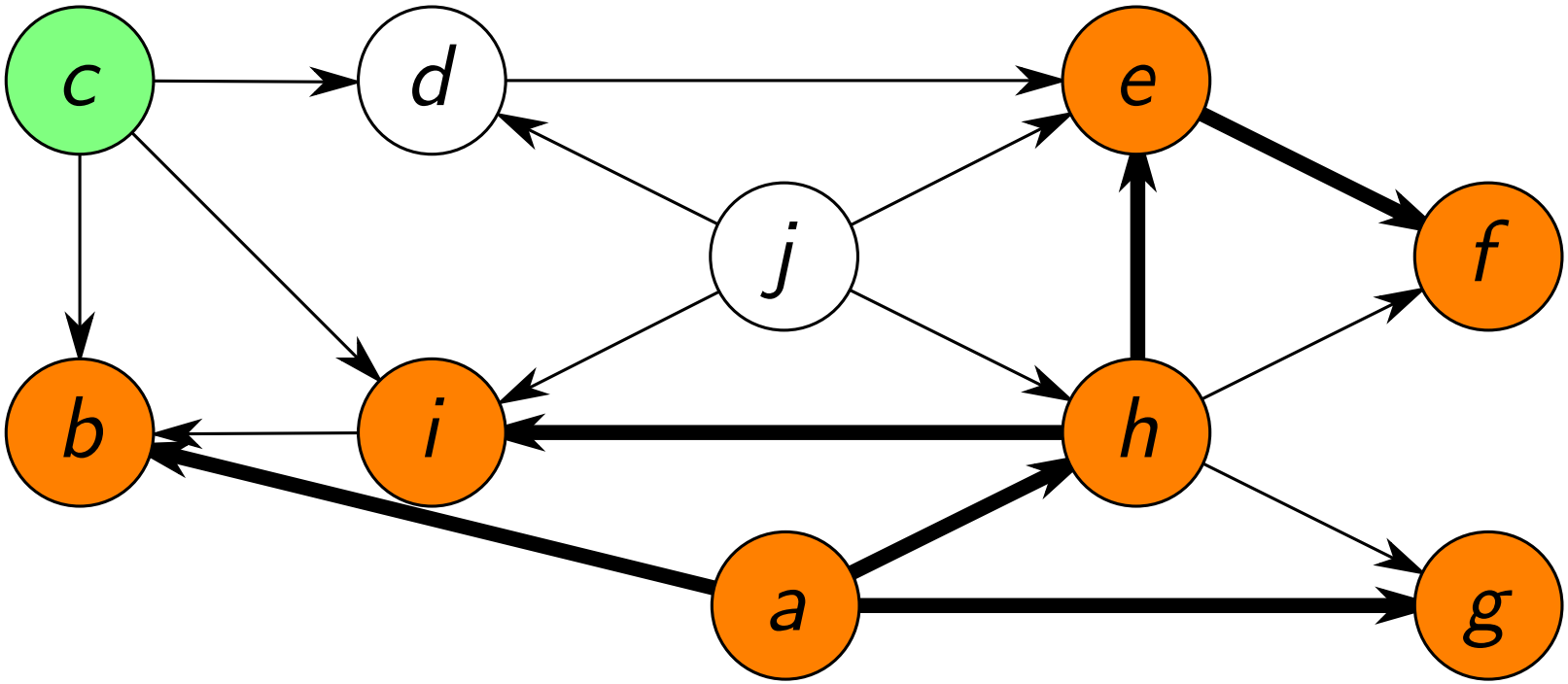
Topological Sorting With DFS (13)



Ordering: *a* *h* *i* *e* *f* *g* *b*

After *i* is processed, DFS unwinds through *h* and *a*, adding all three vertices to the ordering. Then, DFS terminates after unwinding from the initial vertex *a*.

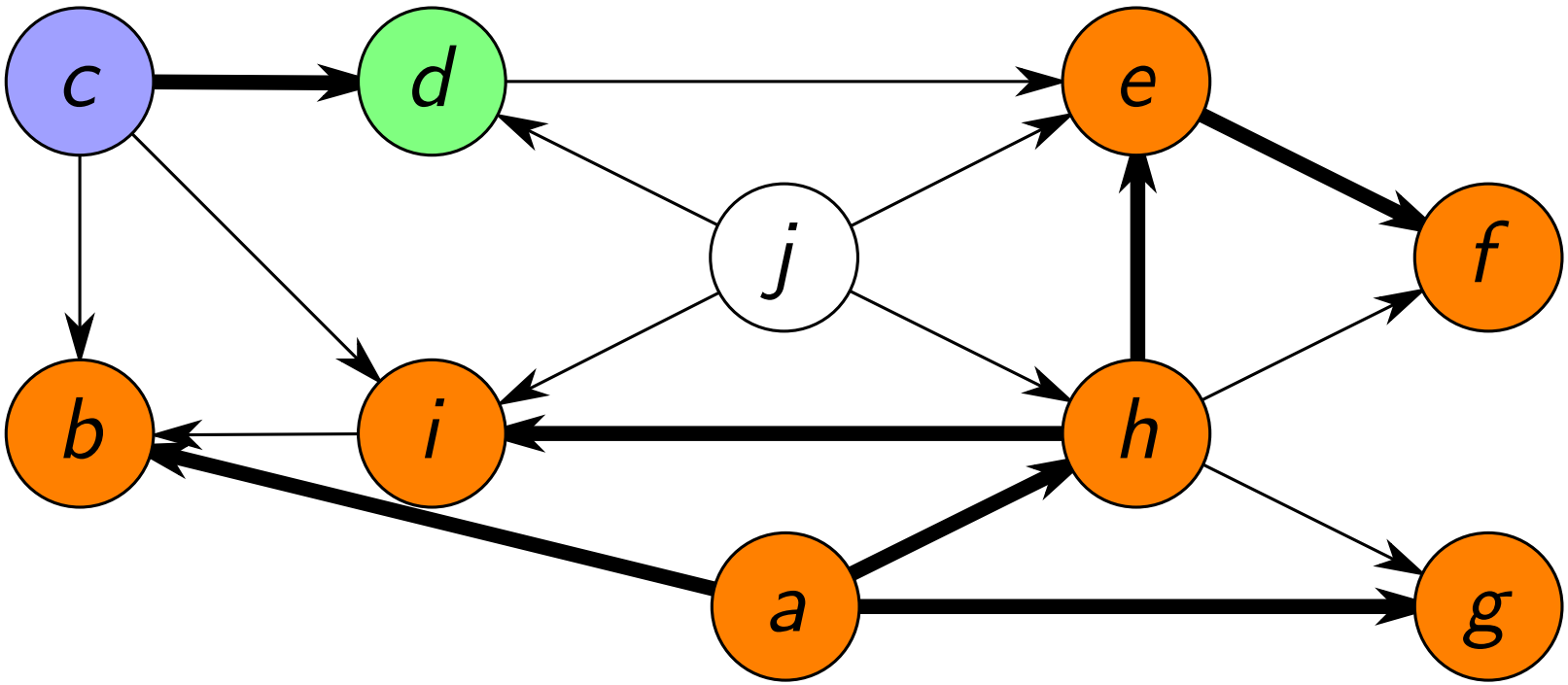
Topological Sorting With DFS (14)



Ordering: *a* *h* *i* *e* *f* *g* *b*

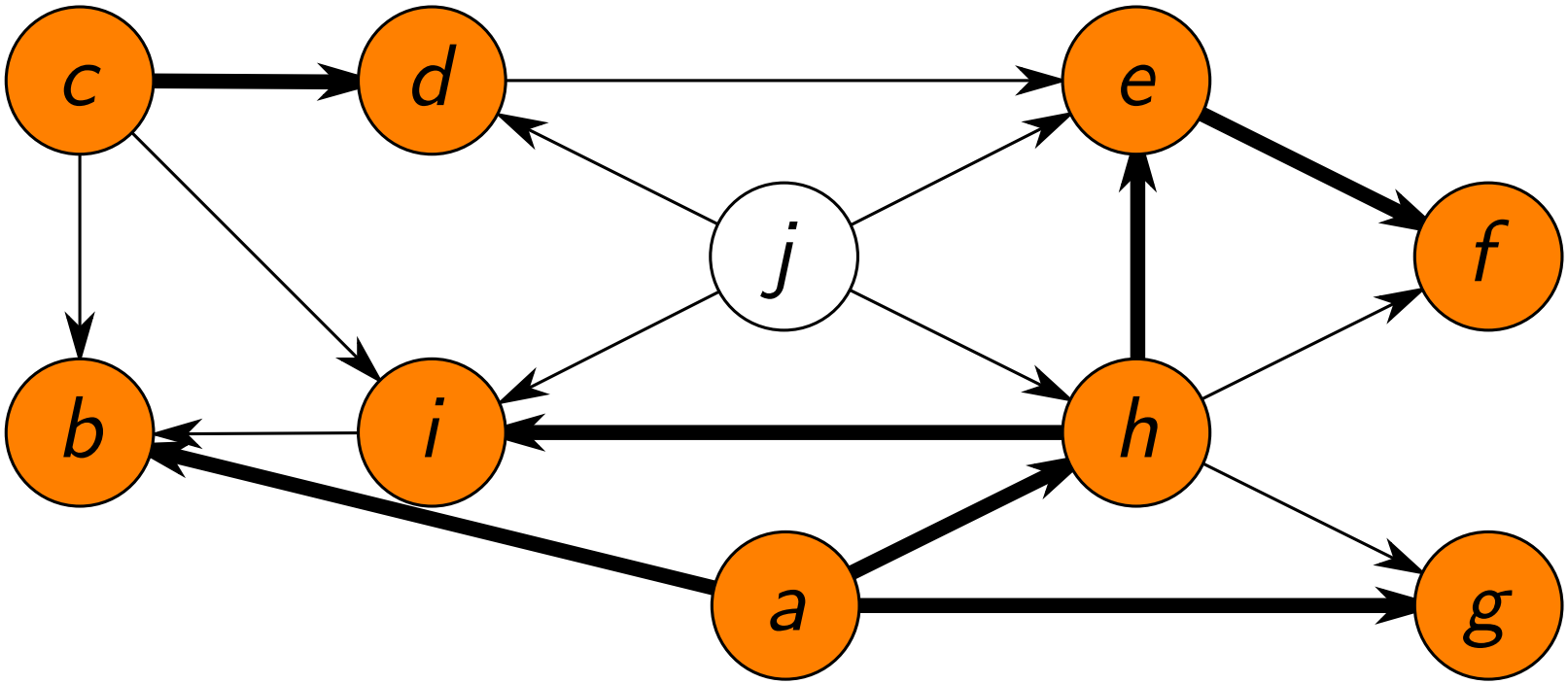
If any unvisited vertices remain, a new traversal must be started on a different initial vertex (using the same ordering).

Topological Sorting With DFS (15)



Ordering: *a* *h* *i* *e* *f* *g* *b*

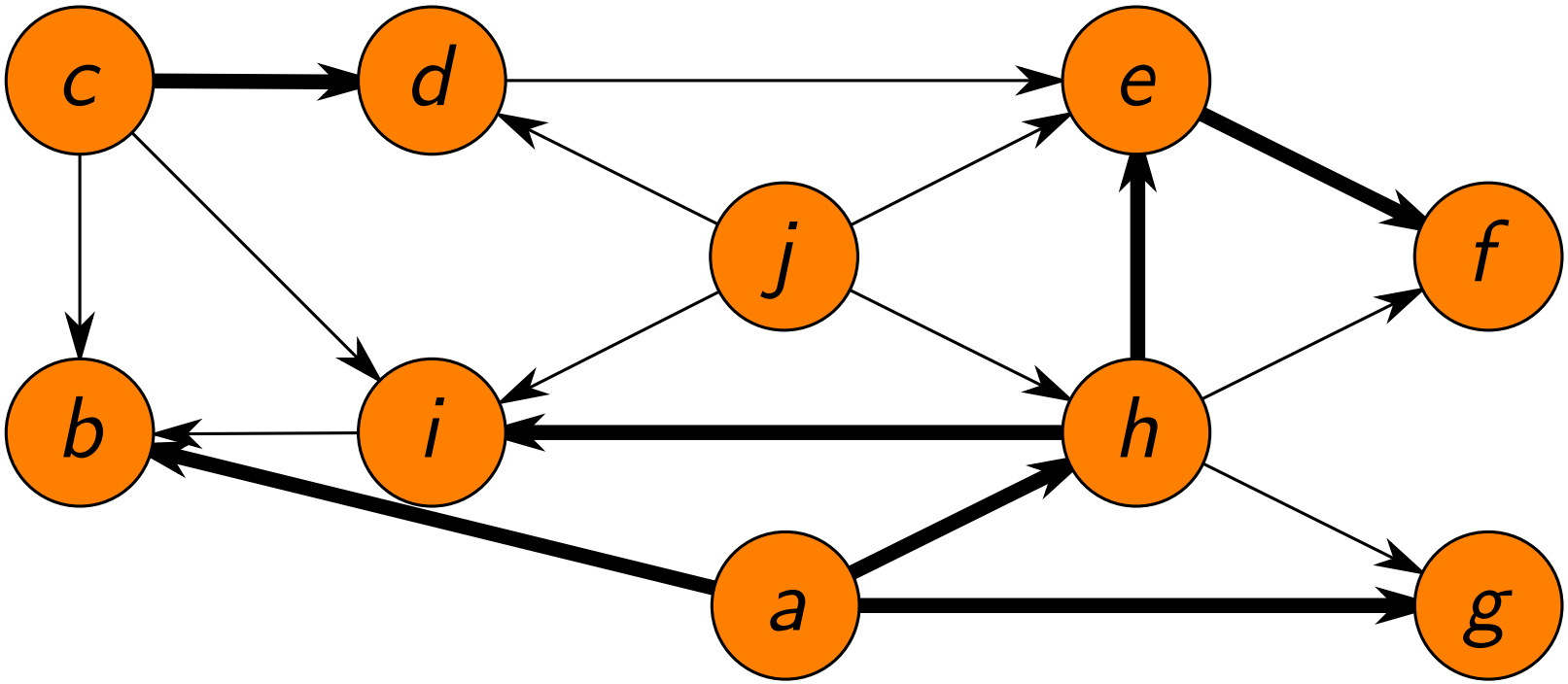
Topological Sorting With DFS (16)



Ordering: c d a h i e f g b

DFS may need to be restarted several times before the entire graph is covered.

Topological Sorting With DFS (17)



Ordering: j c d a h i e f g b

Although several separate traversals were required, each vertex was visited exactly once.

Topological Sorting With DFS (18)

```
 1: procedure RecursiveDFS(G, L, v)
 2:     Mark v as visited
 3:     for each neighbour w of v do
 4:         if w is unvisited then
 5:             RecursiveDFS(G, L, w)
 6:         end if
 7:     end for
 8:     Add v to the front of L
 9: end procedure
10: procedure TopologicalSortDFS(G)
11:     Mark all vertices of G unvisited
12:     L ← Empty list
13:     for each vertex v in G do
14:         if v is unvisited then
15:             RecursiveDFS(G, L, v)
16:         end if
17:     end for
18:     return L
19: end procedure
```

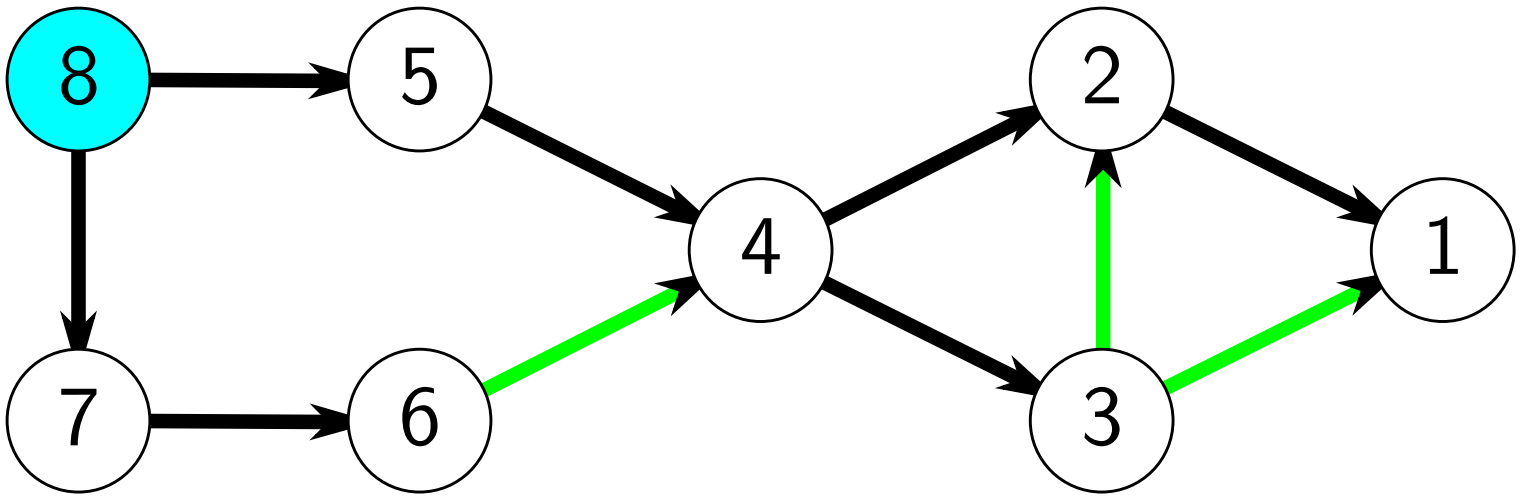
The DFS-like topological sorting algorithm is given by the pseudocode above.

Topological Sorting With DFS (19)

```
 1: procedure RecursiveDFS(G, L, v)
 2:     Mark v as visited
 3:     for each neighbour w of v do
 4:         if w is unvisited then
 5:             RecursiveDFS(G, L, w)
 6:         end if
 7:     end for
 8:     Add v to the front of L
 9: end procedure
10: procedure TopologicalSortDFS(G)
11:     Mark all vertices of G unvisited
12:     L ← Empty list
13:     for each vertex v in G do
14:         if v is unvisited then
15:             RecursiveDFS(G, L, v)
16:         end if
17:     end for
18:     return L
19: end procedure
```

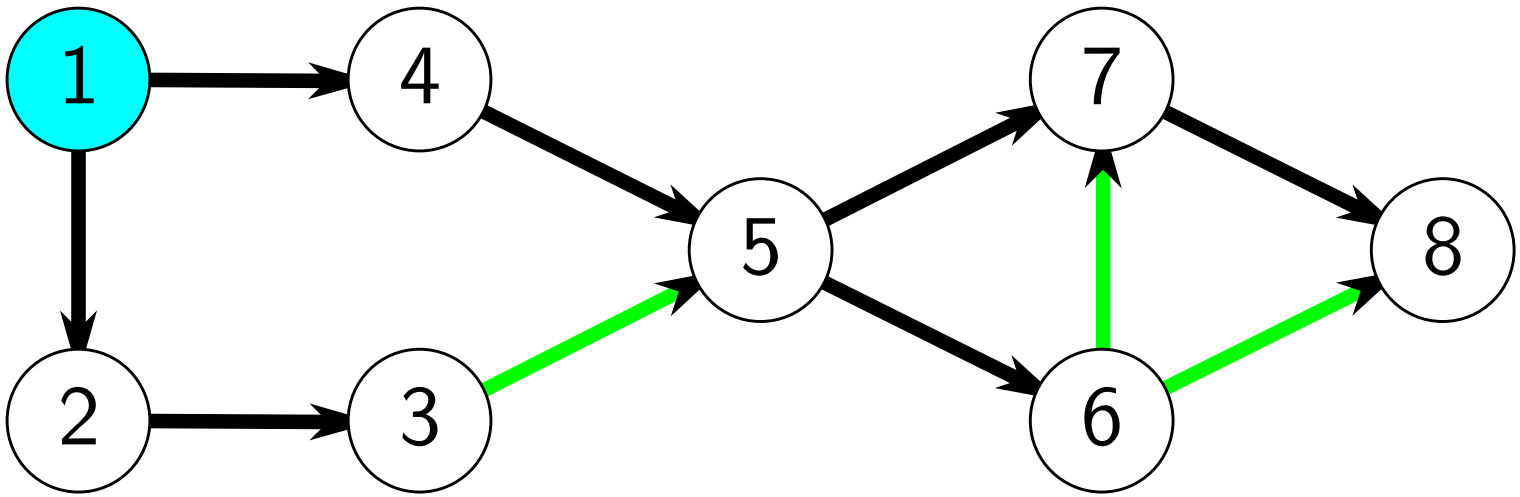
Since each vertex is visited exactly once, the running time of this algorithm is $\Theta(n + m)$ in the worst case.

Post Order DFS (1)



A topologically sorted ordering can also be produced with a post-order numbering of vertices. The numbering above is the post-order numbering of a DFS traversal starting at the blue vertex.

Post Order DFS (2)



Reversing the order of the post-order numbering produces a topologically sorted numbering.