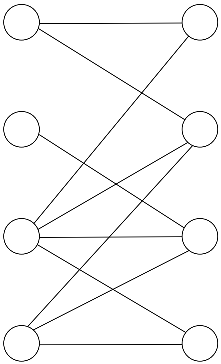
Testing Bipartiteness: Applications of BFS

Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.



Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$

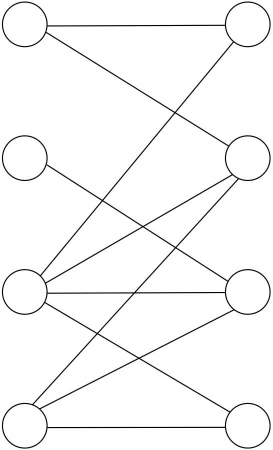
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$
- Report "not a bipartite graph" if contradiction was found

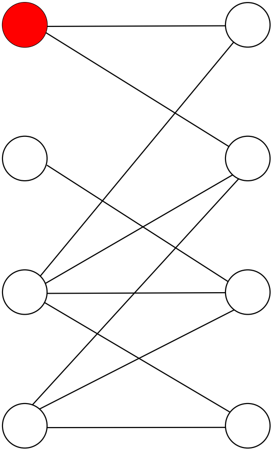
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$
- Report "not a bipartite graph" if contradiction was found
- If G contains multiple connected components, repeat above algorithm for each component

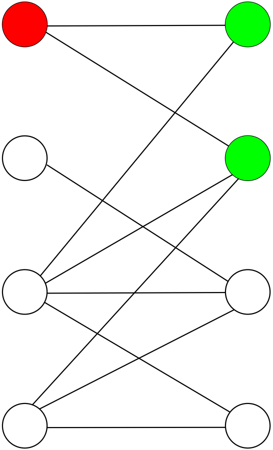
Test Bipartiteness



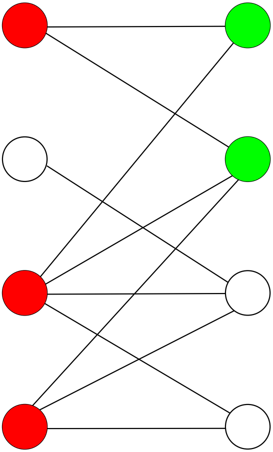
Test Bipartiteness



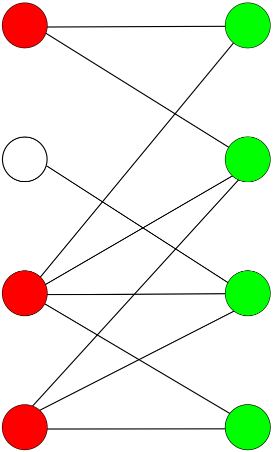
Test Bipartiteness



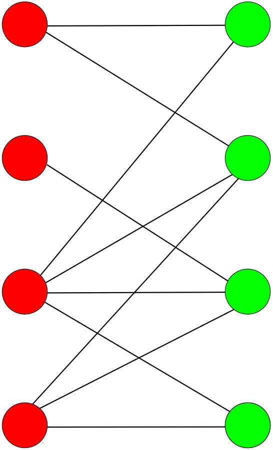
Test Bipartiteness



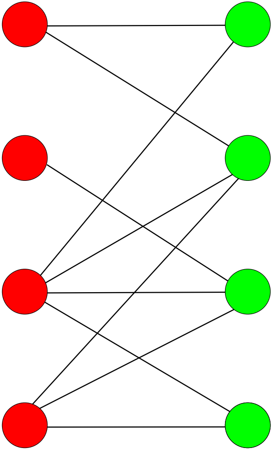
Test Bipartiteness

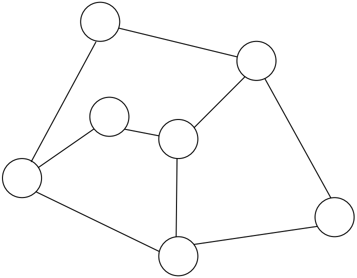


Test Bipartiteness

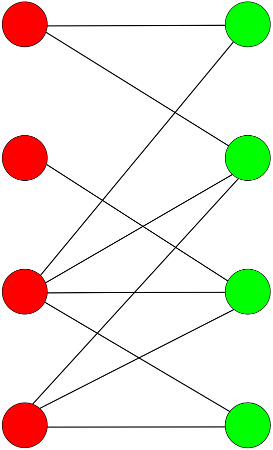


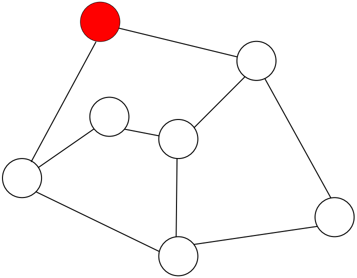
Test Bipartiteness



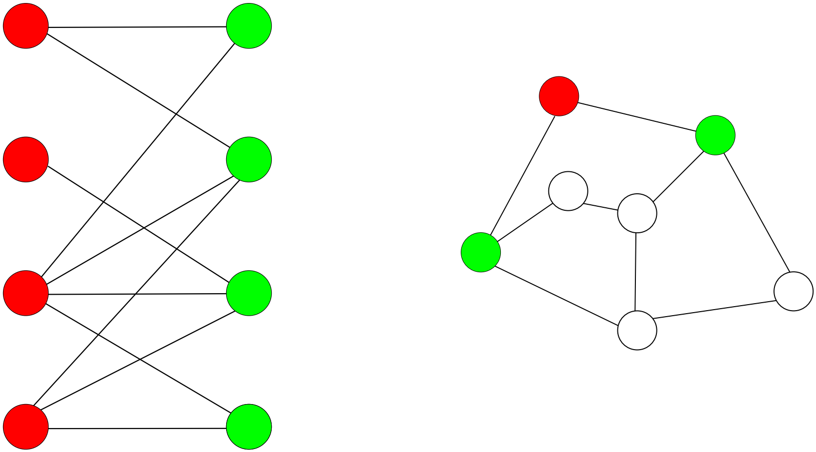


Test Bipartiteness

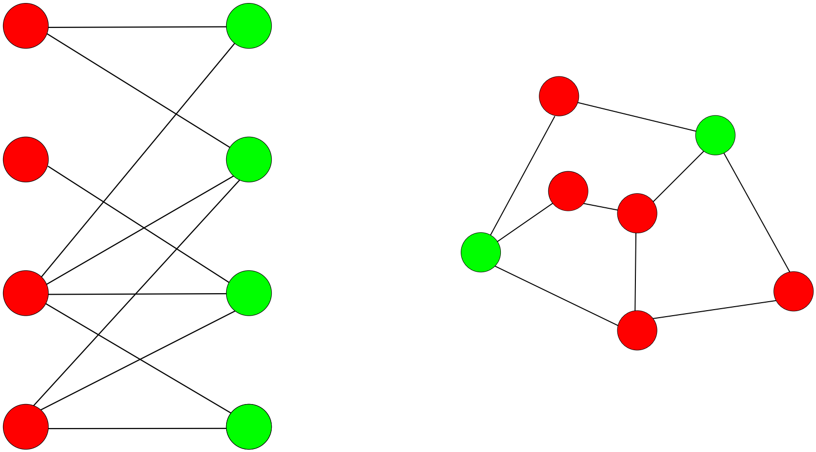




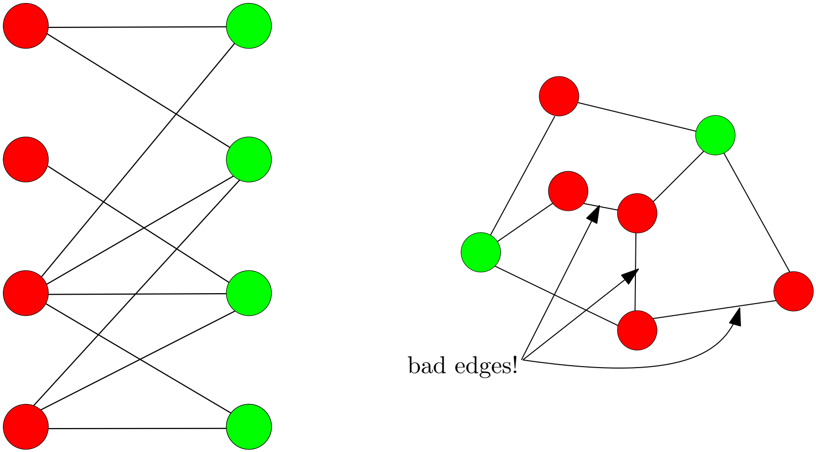
Test Bipartiteness



Test Bipartiteness



Test Bipartiteness



Testing Bipartiteness using BFS

BFS(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"

Testing Bipartiteness using BFS

test-bipartiteness(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
$color[s] \leftarrow 0$
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"
            $color[u] \leftarrow 1 - color[v]$
        else if $color[u] = color[v]$ **then**
            print("G is not bipartite") and exit

Testing Bipartiteness using BFS

mark all vertices as "unvisited"
for each vertex $v \in V$ **do**
    if v is "unvisited" **then**
        test-bipartiteness(v)
print("G is bipartite")

Testing Bipartiteness using BFS

1: mark all vertices as "unvisited"
2: **for** each vertex $v \in V$ **do**
3:     **if** v is "unvisited" **then**
4:         test-bipartiteness(v)
5: print("G is bipartite")

Obs. Running time of algorithm $= O(n + m)$

Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

1: mark all vertices as "unvisited"
2: recursive-test-DFS(s)

recursive-test-DFS(v)

1: mark v as "visited"
2: **for** all neighbors u of v **do**
3:     **if** u is unvisited **then** , recursive-test-DFS(u)

Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

```
mark all vertices as "unvisited"
color[s] ← 0
recursive-test-DFS(s)
```

recursive-test-DFS(v)

```
mark v as "visited"
for all neighbors u of v do
    if u is unvisited then
        color[u] ← 1 − color[v], recursive-test-DFS(u)
    else if color[u] = color[v] then
        print("G is not bipartite") and exit
```

Testing Bipartiteness using DFS

mark all vertices as "unvisited"
for each vertex $v \in V$ **do**
    if v is "unvisited" **then**
        test-bipartiteness-DFS(v)
print("G is bipartite")

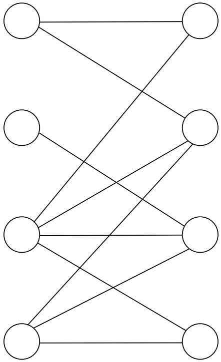
Testing Bipartiteness using DFS

mark all vertices as “unvisited”
for each vertex $v \in V$ **do**
    if v is “unvisited” **then**
        test-bipartiteness-DFS(v)
print(“G is bipartite”)

Obs. Running time of algorithm $= O(n+m)$

Bipartite Graph

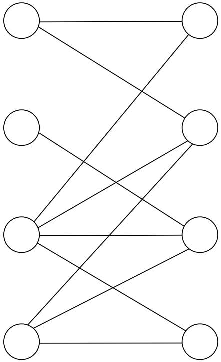
Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.



Bipartite Graph

Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.

Obs. Bipartite graph may contain cycles.

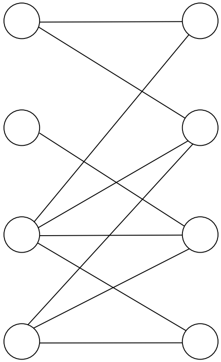


Bipartite Graph

Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.

Obs. Bipartite graph may contain cycles.

Obs. If a graph is a tree, then it is also a bipartite graph.



BFS and DFS

Obs. BFS and DFS naturally induce a tree.

BFS and DFS

Obs. BFS and DFS naturally induce a tree.

Obs. If G is a tree, then BFS tree $=$ DFS tree.

BFS and DFS

Obs. BFS and DFS naturally induce a tree.

Obs. If G is a tree, then BFS tree $=$ DFS tree.

Obs. If BFS tree $=$DFS tree, then G is a tree.

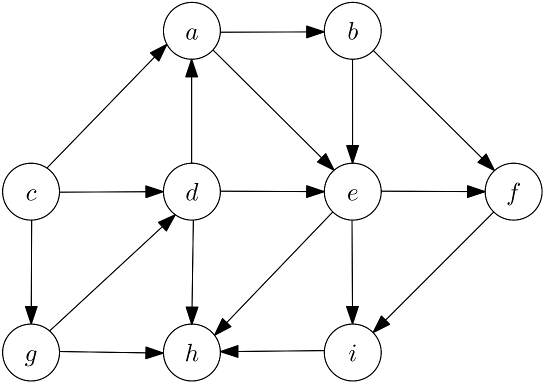
Outline

Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$

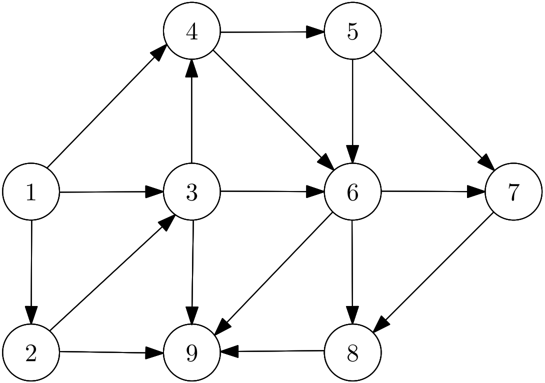


Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

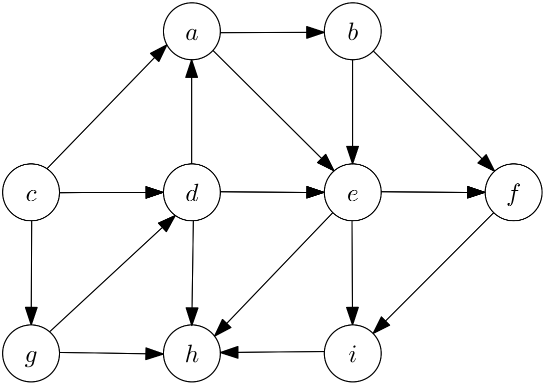
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$



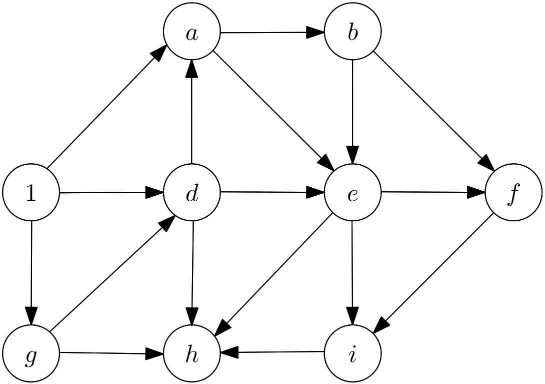
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



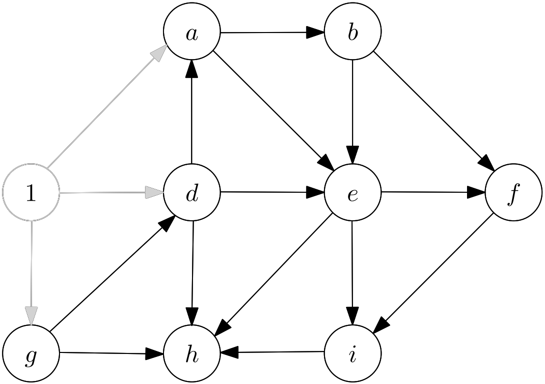
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



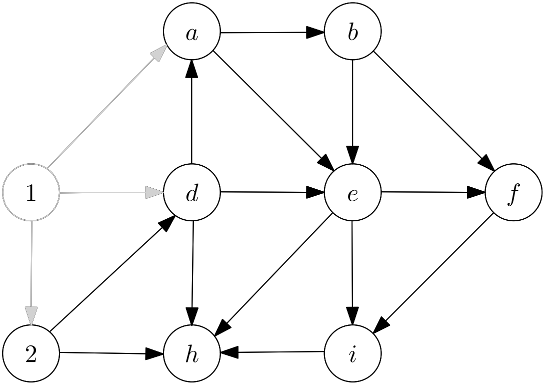
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



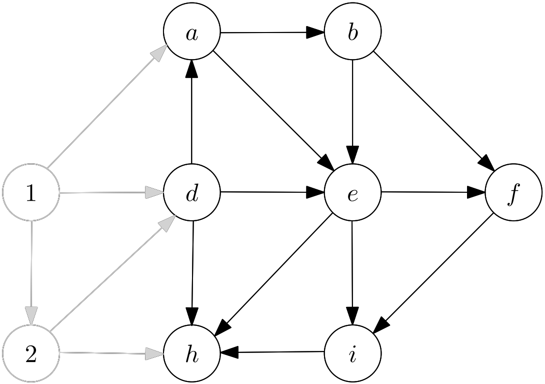
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



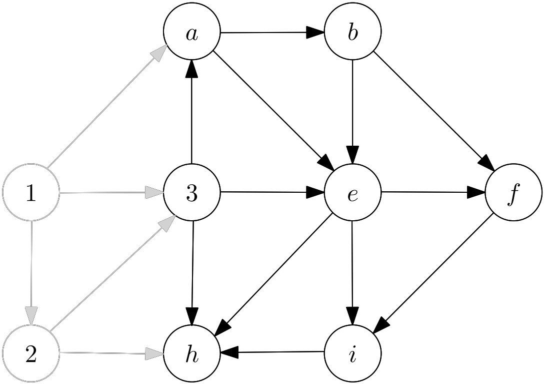
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



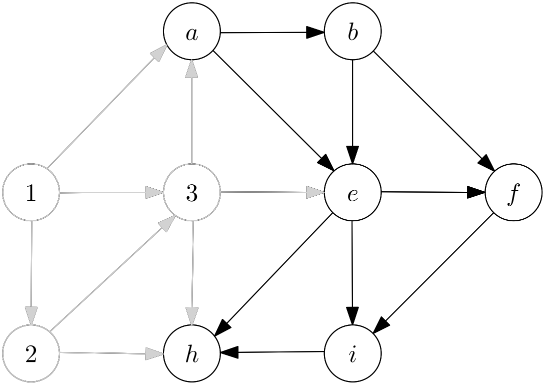
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



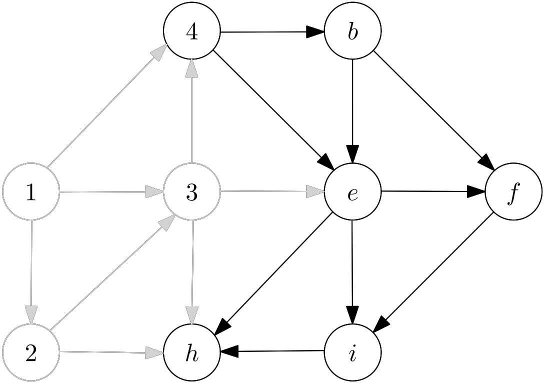
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



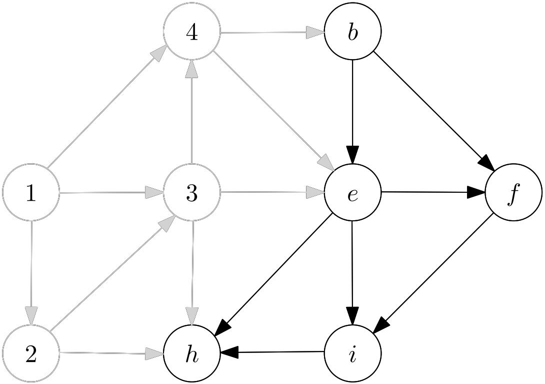
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



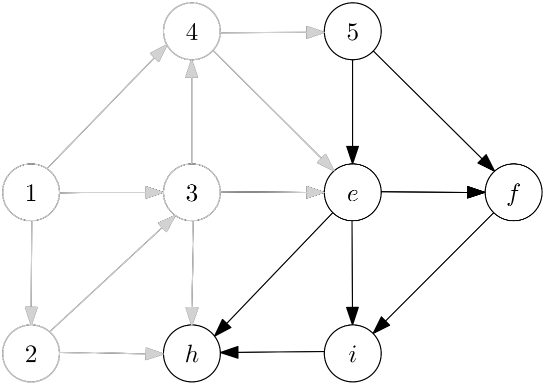
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



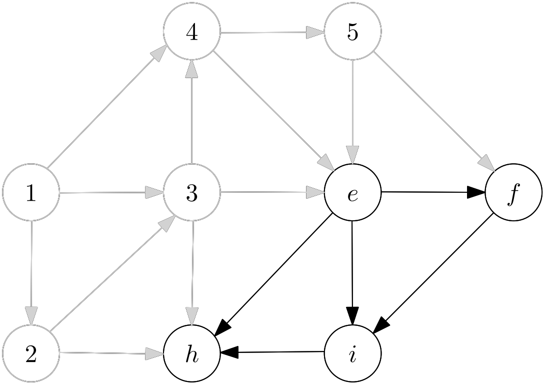
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



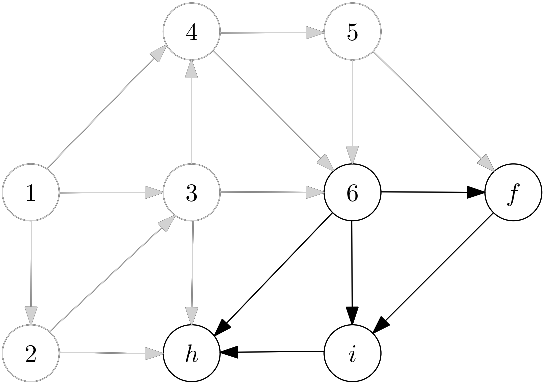
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



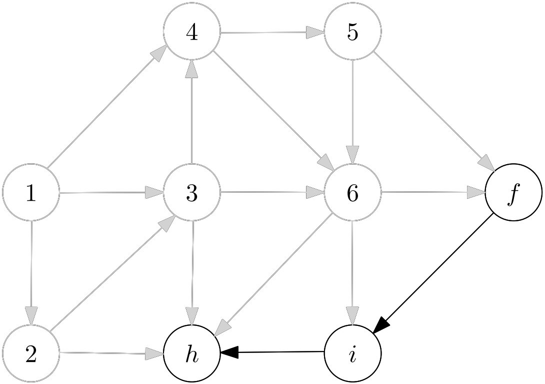
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



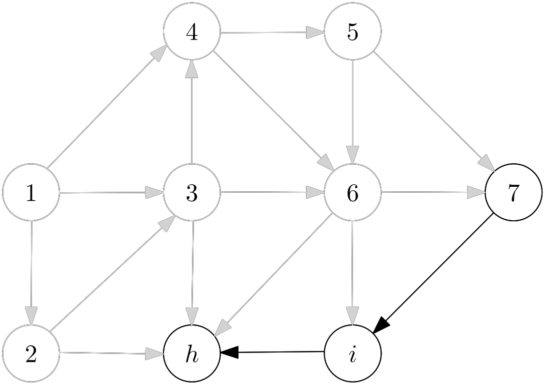
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



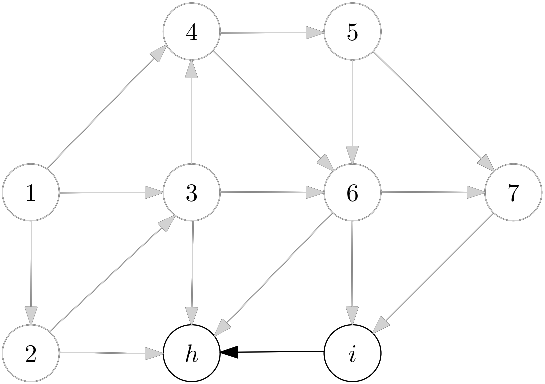
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



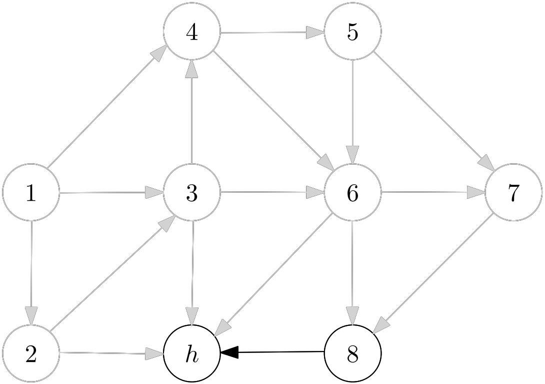
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



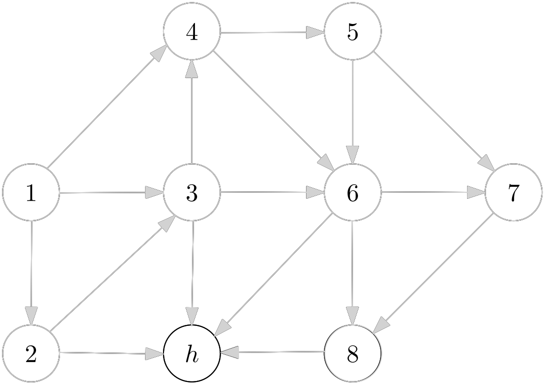
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



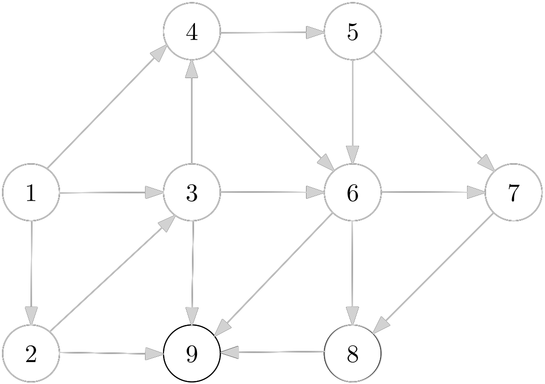
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



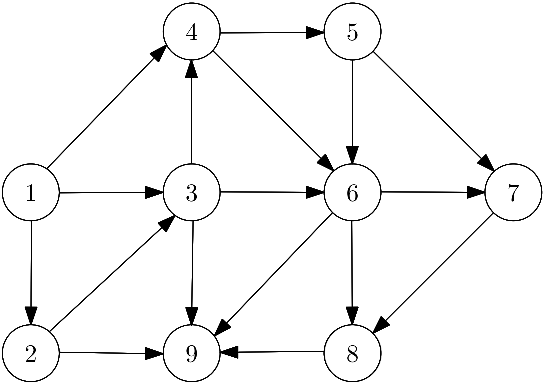
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:

- Use linked-lists of outgoing edges
- Maintain the in-degree d_v of vertices
- Maintain a queue (or stack) of vertices v with $d_v = 0$

topological-sort(G)

```
let d_v ← 0 for every v ∈ V
for every v ∈ V do
    for every u such that (v, u) ∈ E do
        d_u ← d_u + 1
S ← {v : d_v = 0}, i ← 0
while S ≠ ∅ do
    v ← arbitrary vertex in S, S ← S \ {v}
    i ← i + 1, π(v) ← i
    for every u such that (v, u) ∈ E do
        d_u ← d_u − 1
        if d_u = 0 then add u to S
if i < n then output "not a DAG"
```

- S can be represented using a queue or a stack
- Running time $= O(n + m)$