Parallel And Sequential Data Structures and Algorithms

Graph Search: DFS

Learning Objectives

- Review **graph** definitions and representations
- Understand **depth-first search** and its applications

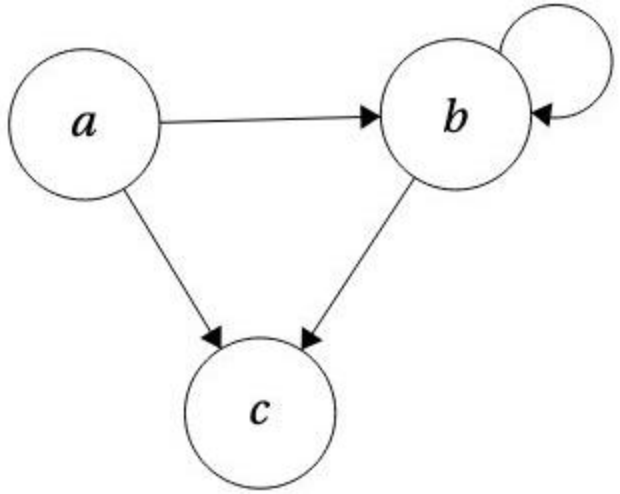
Definitions

Definition (Graph): a graph *G = (V, E)* is defined as a pair of sets, a set of vertices *V*, and a set of edges *E*. Each element of *E* is a pair of elements from *V*, possibly with an associated weight.

- We frequently use *n* to refer to the number of vertices, |*V* |, and *m* to refer to the number of edges, |*E* |

Example: we can represent this graph with $V = \{a, b, c\}$ and $E = \{(a, b), (b, b), (b, c), (a, c)\}$

note that this is a directed graph, so the edge from *a* to *b* corresponds to the tuple *(a, b)*



Number of Edges

Question: Let's say we have a directed graph on n vertices, where edges from a vertex to itself are not allowed. What is the maximum number of edges that we can create in this graph?

- If we allow multiple copies of an edge to exist, our graph can have infinitely many edges (we can just duplicate any edge)
- Otherwise, for each edge, we need to choose a source and destination. There are n possible sources and n possible destinations, therefore there are n^2 possible edges.

Number of Edges

Question: Let's say we have a directed graph on n vertices, where edges from a vertex to itself are not allowed. What is the maximum number of edges that we can create in this graph?

Question: If we don't allow multiple copies of an edge to exist, how many possible graphs are there on n vertices?

- Each of the n^2 possible edges are either in or not in any graph with n vertices; there are 2^{n^2} such graphs

Graph Classification

- The graphs we have seen so far have been **directed** (every edge has a source and a destination vertex)
- In an **undirected** graph, E should not contain both (u, v) and (v, u), since these tuples represent the same edge

Definition (Simple Graph): a graph is **simple** if it contains no **self-loops** (edges that start and end at the same vertex) or **multi-edges** (an edge that appears more than once in a graph)

Graph Substructures

Definition (Neighbor): a vertex is a neighbor of another vertex in an undirected graph if the two vertices are connected by an edge

- We define the set $N_G(v)$ for a vertex *v* in an undirected graph *G* to be the set of *v*'s neighbors, or all u such that $(u, v) \in E$
- For a directed graph, we say u is an **in-neighbor** of v if there is an edge from u to v. Here, v is an **out-neighbor** of u
- The set of all in-neighbors of *v* in graph *G* is denoted $N_G^-(v)$ and the set of all out-neighbors as $N_G^+(v)$

Graph Substructures

Definition (Degree): the degree of a vertex v in an undirected graph G is the number of neighbors it has. That is,

$$d_G(v) = |N_G(v)|$$

- We can similarly define the **in-degree** and **out-degree** of a vertex v in a directed graph G, denoted as $d_G^-(v)$ and $d_G^+(v)$ respectively

Graph Substructures

Definition (Path): a path is a sequence of vertices such that each adjacent pair in the sequence has an edge between them in the graph. The length of a path is the number of edges in the path.

Definition (Simple Path): a path that does not reuse any vertices

Definition (Reachability): A vertex u is reachable from a vertex v if there exists a path that starts at u and ends at v.

Graph Substructures

Definition (Cycle): a path that starts and ends at the same vertex

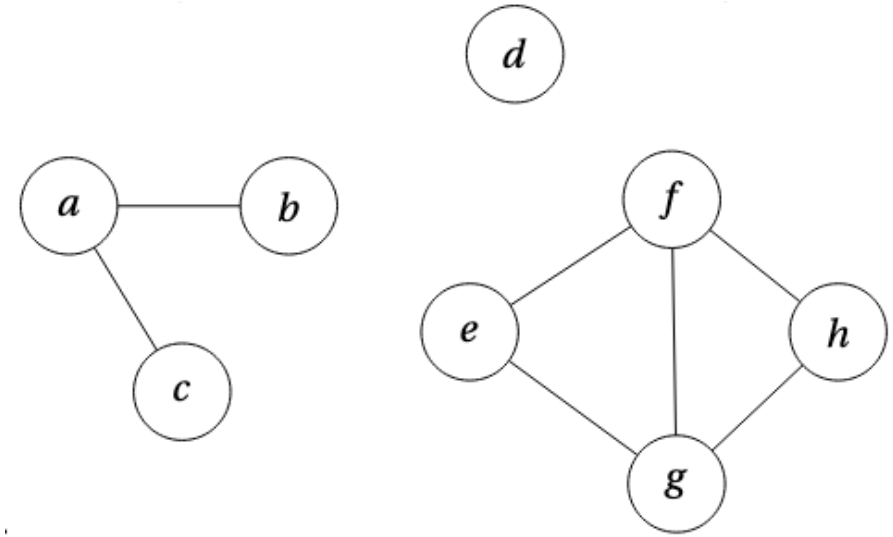
Definition (Simple Cycle): a cycle that does not reuse any vertices or edges (excluding the starting/ending vertex)

Definition (DAG): a directed acyclic graph is a directed graph with no cycles

Graph Substructures

Definition (Connected Component): a connected component of an undirected graph is a subset C of the vertices such that, given any vertex v in C, all other vertices are reachable from v

Example: in this graph, the connected components are $\{a, b, c\}, \{d\}, \text{and } \{e, f, g, h\}$

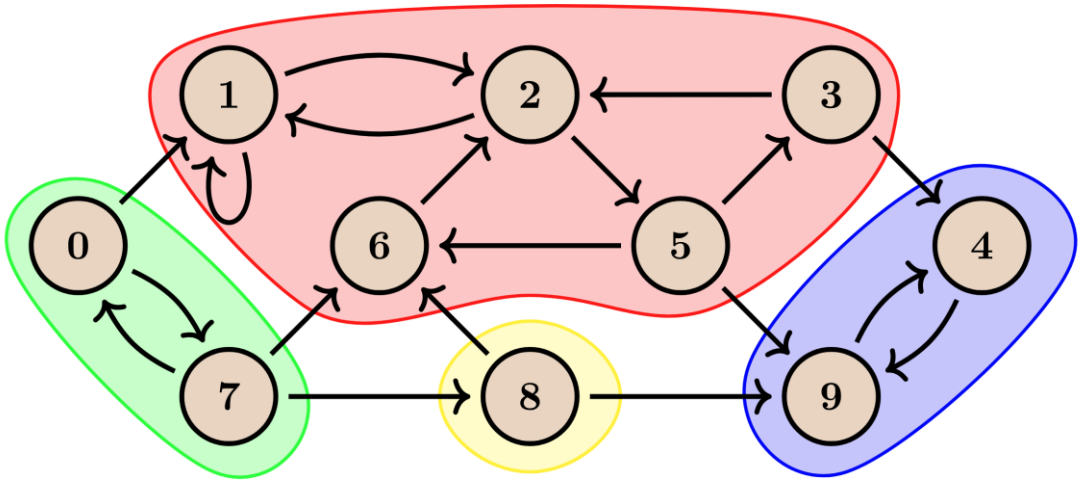


- Note: this idea is a little more complicated for directed graphs…

Graph Substructures

Definition (Strongly Connected Component): In a directed graph, two vertices u and v are in the same strongly connected component (SCC) if u is reachable from v and v is reachable from u.

Example: in this graph, the strongly connected components are $\{0,7\}, \{8\}, \{4,9\}, \text{and } \{1,2,3,6,5\}$



Note: The SCCs form a DAG.

Representing Graphs

Graph Representation

- How do we store the graph in memory?
- By default, we focus on **enumerable graphs**, where the n vertices of the graph are labeled with the integers 0 to $n-1$
- Different representations support different operations more efficiently
  - Most graph algorithms primarily need one fundamental operation: given a vertex v, obtain its neighbors

Adjacency Lists

Definition (Adjacency List): for each vertex *v*, store the collection of edges leaving *v*

- If the graph is enumerable, our adjacency list has type

```
sequence<sequence<Edge>>
```

  - Given an adjacency list `adj`, the neighbors of *u* are stored in `adj[u]`
  - Allows for O(1) access to a vertex's adjacency list
- Otherwise (if the type of the vertex labels is not integers), we can store a dictionary mapping vertices to their adj list:

```
dictionary<V, sequence<Edge>>
```

```
type Edge = {to : V, weight : W}   (weighted graph)
```

Adjacency Lists

Claim (Cost of Adjacency Lists): For both the enumerable and non-enumerable version, an adjacency list uses $\Theta(n + m)$ space.

For obtaining the neighbors of a vertex u, the cost is $O(1)$ for an enumerable graph. If the graph is not enumerable, we can either use:

- A hashable dictionary: lookup takes $O(1)$ expected time
- A balanced-BST dictionary: lookup takes $O(\log n)$ time

Adjacency Matrices

Definition (Adjacency Matrix): If the vertices are numbered $0, \ldots, \mathrm{n} - 1$, we store an $n \times n$ matrix

```
sequence<sequence<W>>
```

where entry *(u, v)* indicates the weight of the edge from *u* to *v*. In an unweighted graph, the weights can simply be booleans.

- Allow constant-time edge existence queries
- However, finding all the neighbors of a vertex requires searching an entire row and takes $\Theta(n)$ time
- The space usage is $\Theta(n^2)$, which is inefficient for sparse graphs but often the best choice for dense graphs

Depth-First Search

Depth-First Search

- Explores all the edges and vertices in the graph in $O(n + m)$ work and span (linear in $|G|$)!
- Sequential algorithm
- From here, we will assume the graphs we are processing are enumerable and use the adjacency list representation of graphs

Depth-First Search

Do example here

Algorithm (DFS Skeleton):

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun skeleton(adj : adjacency_list):
  n = |adj|
  visited = [False for _ in 0...n-1]

  fun dfs(u : int):
    visited[u] ← True
    for v in adj[u]:
      if not visited[v]:
        dfs(v)
```

Depth-First Search

What does running dfs(s) do? Which vertices does it visit?

Answer: it visits all the vertices in the graph reachable from s.

Proof: By looking at the pseudo-code we see that if there is an edge (u, v) and u is visited, then v must also be visited.

Suppose there exists a path from s to v. Since s is visited, it follows that v must be visited. QED

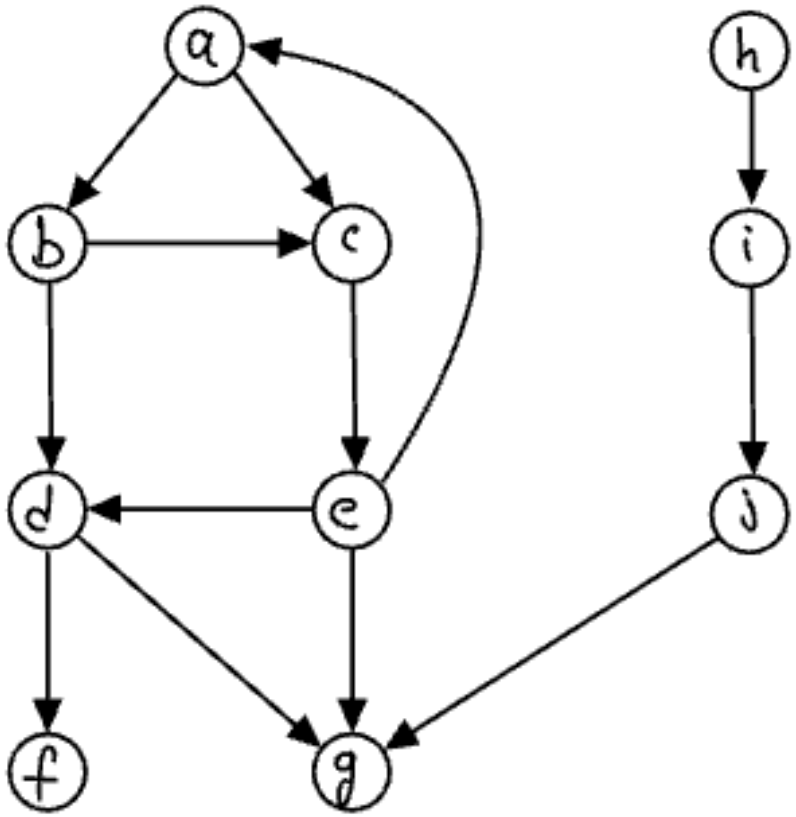
Vertex Numbering

Algorithm (DFS Vertex Numbering):

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun dfs_numbers(adj : adjacency_list)
                -> (sequence<int>,
                    sequence<int>):
  n = |adj|
  visited = [False for _ in 0...n-1]
  start =   [0 for _ in 0...n-1]
  finish =  [0 for _ in 0...n-1]
  i = 0
```

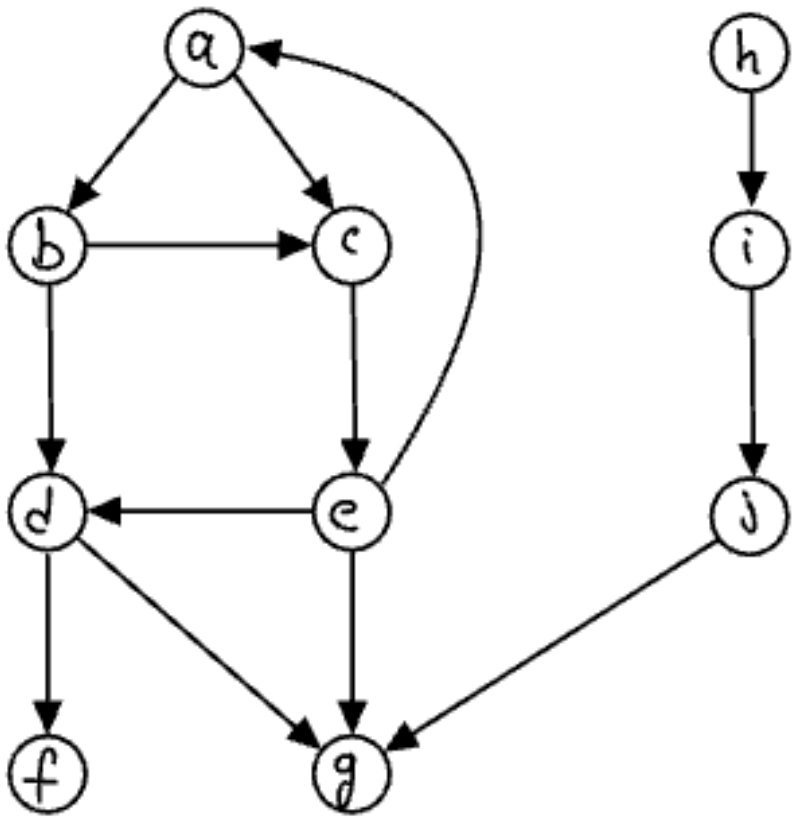
```
fun dfs(u : int):
  visited[u] ← True
  start[u] ← i;
  i ← i+1;
  for v in adj[u]:
    if not visited[v]:
      dfs(v)          // tree edge
  finish[u] ← i
  i ← i+1

for u in 0...n-1:
  if not visited[u]:
    dfs(u)

return (start, finish)
```

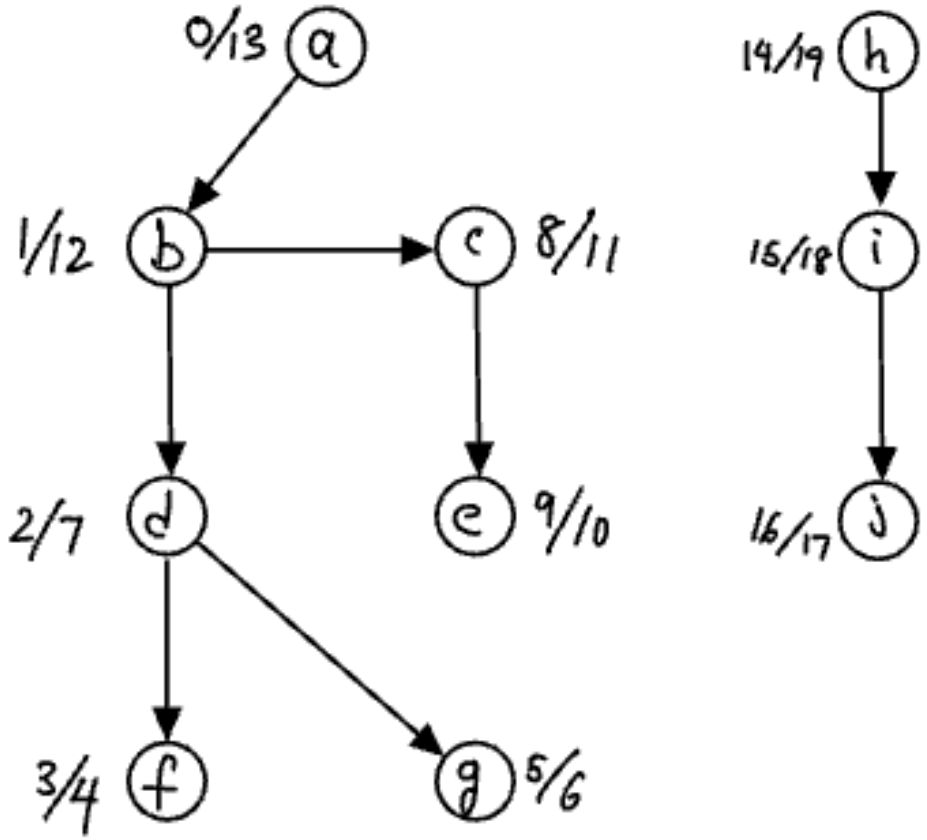
Vertex Numbering




```
fun dfs_numbers(adj : adjacency_list)
    -> (sequence<int>, sequence<int>):
  n = |adj|
  visited = [False for _ in 0...n-1]
  start =   [0 for _ in 0...n-1]
  finish =  [0 for _ in 0...n-1]
  i = 0

  fun dfs(u : int):
    visited[u] ← True
    start[u] ← i;
    i ← i+1;
    for v in adj[u]:
      if not visited[v]:
        dfs(v)      // tree edge
    finish[u] ← i
    i ← i+1

  for u in 0...n-1:
    if not visited[u]:
      dfs(u)

  return (start, finish)
```

Vertex Numbering

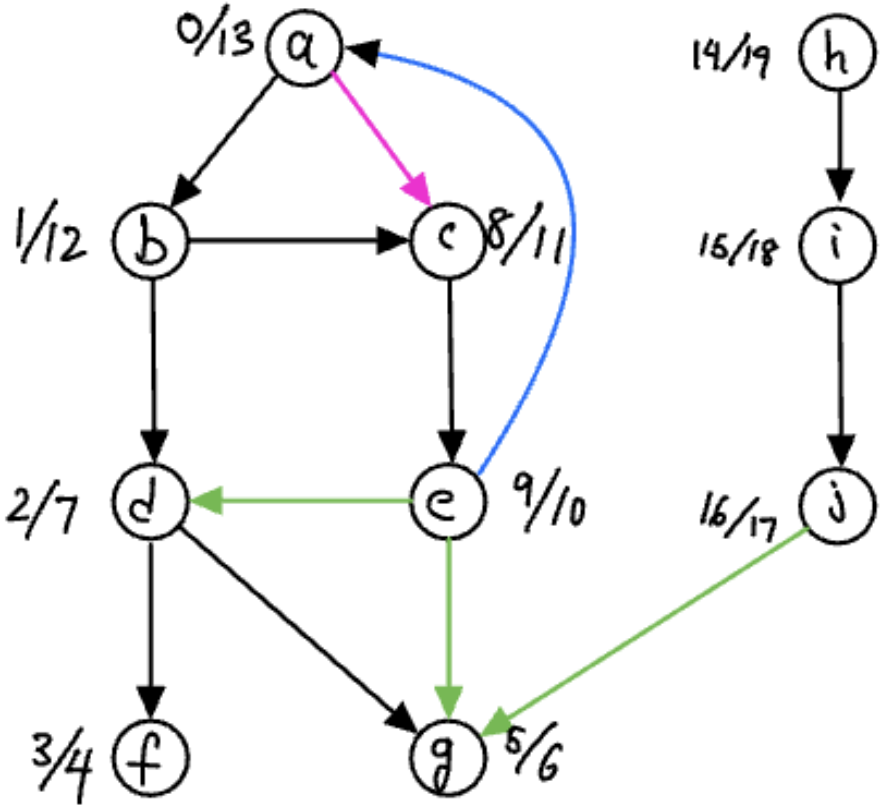




Classification of DFS Edges

Definition (DFS Edges):

- **Tree edges** occur when a node is first visited.
- **Forward edges** go from a node to one of its descendants in the DFS tree.
- **Back edges** go from a node to one of its ancestors in the tree.
- **Cross edges** connected two nodes that are unrelated by the ancestor relation.



Vertex Numbering

- Interesting properties:

Directed Cycles

Theorem (Directed Cycles): A directed graph contains a cycle if and only if its DFS discovers a back edge.

Proof:

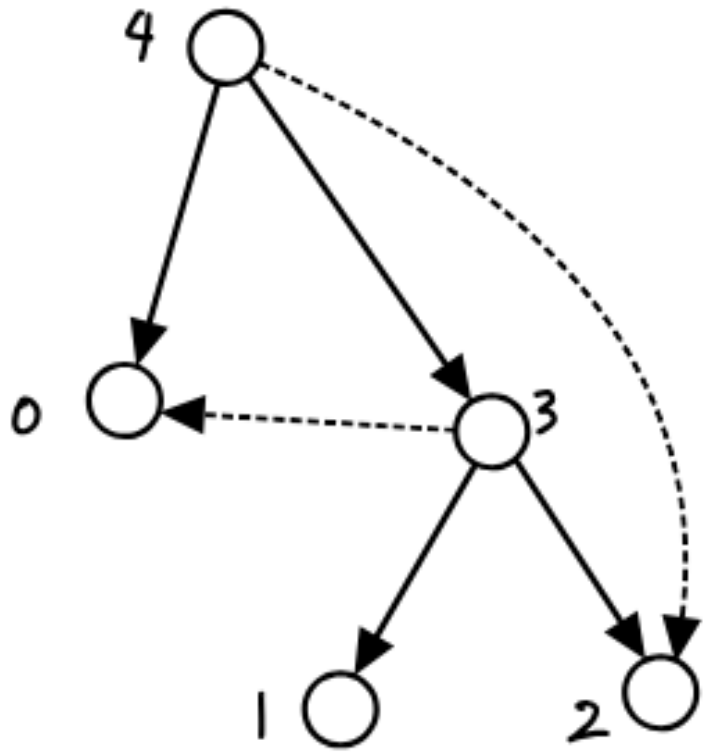
Computing a Topological Ordering

Algorithm (Topological Sorting):

```
// Assume an unweighted, enumerable
graph
type adjacency_list =
sequence<sequence<int>>

fun topological_sort(adj :
adjacency_list) -> sequence<int>:
  n = |adj|
  visited = [False for _ in 0...n-1]
  answer = [0 for _ in 0...n-1]

  i = 0
```

```
fun dfs(u : int):
    visited[u] ← True
    for v in adj[u]:
      if not visited[v]:
        dfs(v)
    answer[i] ← u
    i ← i+1

  for u in 0...n-1:
    if not visited[u]:
      dfs(u)

  return reverse(answer)
```

Computing a Topological Ordering




```
fun topological_sort(adj :
adjacency_list) -> sequence<int>:
  n = |adj|
  visited = [False for _ in 0...n-1]
  answer = [0 for _ in 0...n-1]

  i = 0
  fun dfs(u : int):
    visited[u] ← True
    for v in adj[u]:
      if not visited[v]:
        dfs(v)
    answer[i] ← u
    i ← i+1

  for u in 0...n-1:
    if not visited[u]:
      dfs(u)

  return reverse(answer)
```

Computing a Topological Ordering

Lemma: When the DFS Topological Ordering algorithm is run on a DAG, it orders the vertices in topological order.

Proof: