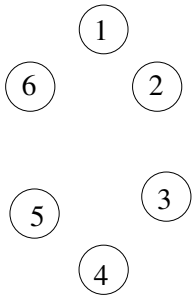
Chapter 22: Elementary Graph Algorithms

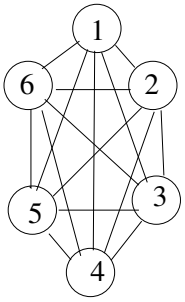
Introduction

- A graph $G = (V, E)$, is defined as a set of vertices V and a set of edges E that connect the vertices. We will denote the number of vertices $\mid V \mid = n$ and the number of edges $\mid E \mid = m$. A graph is a superset of a network. Nodes and links are equivalent concepts to verices and edges.
- The vertices of a graph can represent servers in a distributed system, servers on the internet, cities in a road map, users in a friend network and so on. The edges represent connections between the vertices.
- Graphs can be undirected or directed, based on whether edges have direction or not.

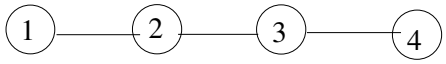
Here are some examples of undirected graphs.



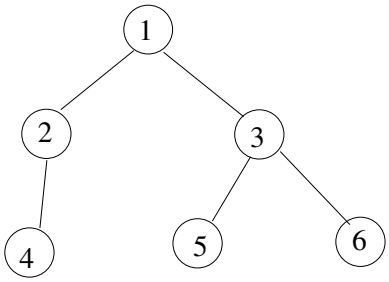
An Empty Graph



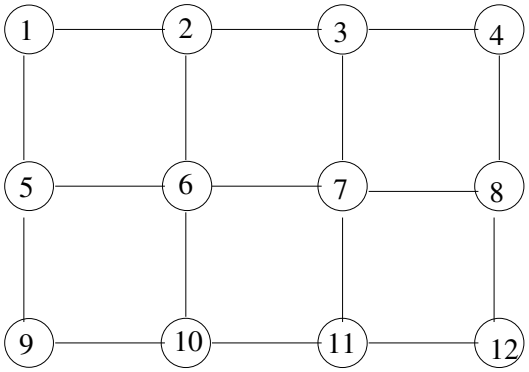
A Complete Graph



A List is a graph

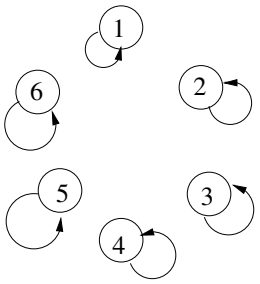


A Tree is a graph

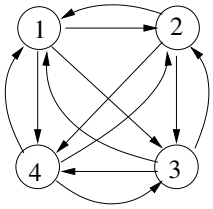


A grid or city map is a graph!

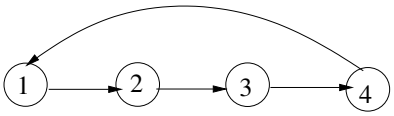
Here are some examples of directed graphs.



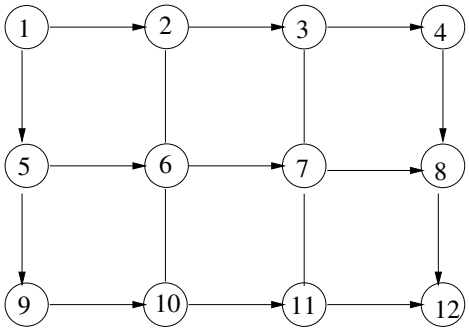
A party of narcissists



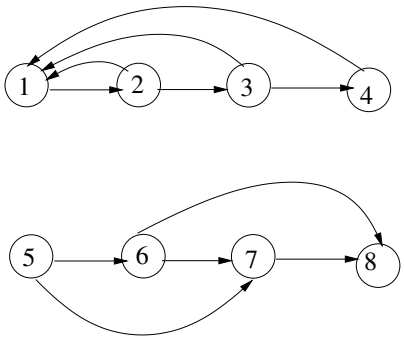
A Complete Graph



A circular list is a graph



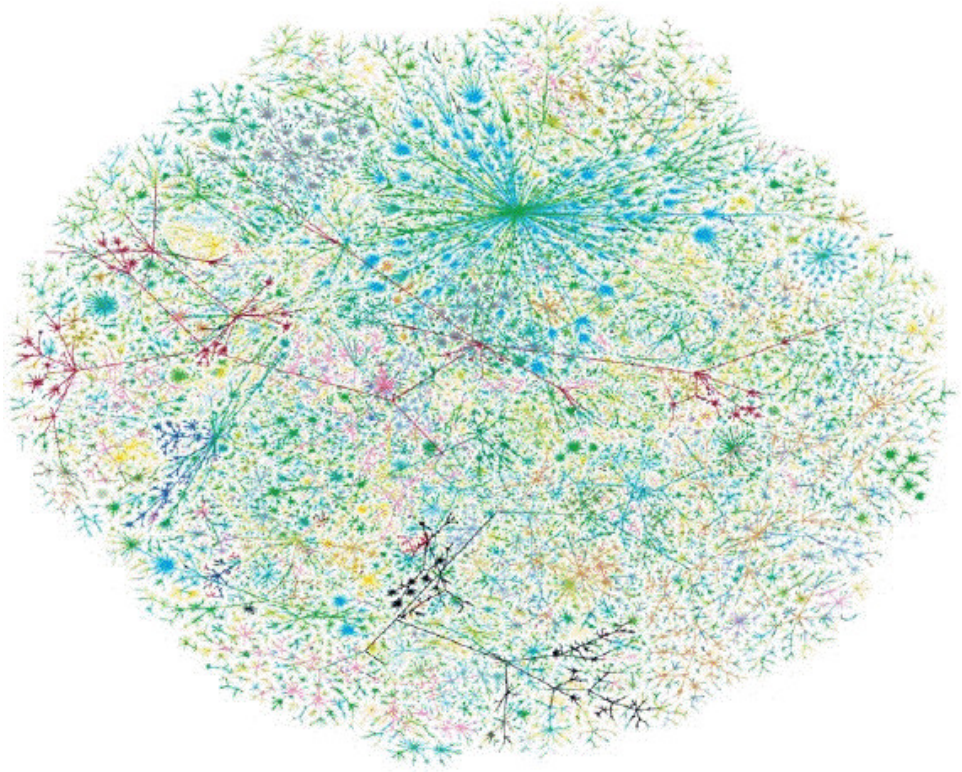
All roads lead to 12!



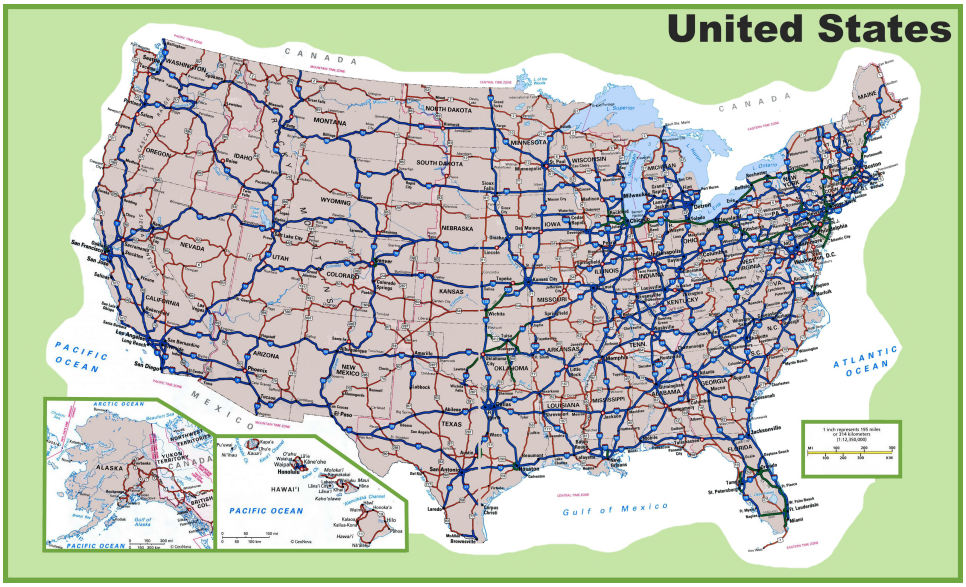
A graph with two connected components

A graph of Netflix servers!

The graph (network) representing the Internet:



US Road Map, where cities are vertices and roads are edges.



Facebook Friend Graph

Representations of Graphs

Adjacency-list

A graph $G = (V, E)$ can be represented by an array Adj of $n = |V|$ lists, one for each vertex in V, where V and E are the sets of vertices and edges, respectively.

For each vertex $u \in V$, $Adj[u]$ is a list containing all the vertices v that $(u, v) \in E$.

If G is an undirected graph, the sum of the lengths of all lists is equal to $2|E| = 2m$.

If G is a directed graph, the sum of the lengths of all lists is equal to $|E| = m$.

The amount of memory required for adjacency-list representation is $O(| V | + | E |) = O(n+m)$ (both directed and undirected graphs).

To determine if an edge (u, v) is in a given graph requires $O(|V|)$. It's not efficient. Why?

A *weighted graph* has a weight function $w : E \to \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. Each weight $w(u, v)$ of the edge $(u, v) \in E$ is stored with vertex v in u's list.

Adjacency-matrix

A graph $G =< V, E >$ can be represented by a $|V| \times |V|$ matrix $A = (a_{ij})$, where

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

If G is a directed graph, the number of 1's in the matrix is equal to $|E|$.

If G is an undirected graph, the number of 1's in the matrix is equal to $2|E|$.

Memory requirement is $|V|^2$.

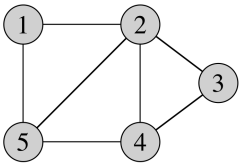
The weighted graph in an adjacency-matrix representation:

Store the weight $w(u, v)$ of the edge $(u, v) \in E$ in the entry a_{uv}. If $(u, v) \notin E$, then

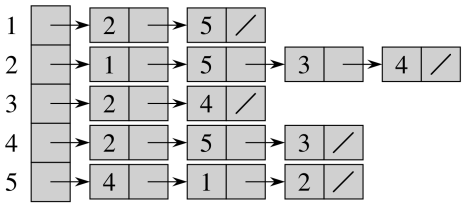
$$a_{uv} = \begin{cases} \text{nil} \\ 0 \quad \text{depends on the applications} \\ \infty \end{cases}$$

To determine whether an edge (u, v) is in a given graph takes only constant time.

Examples



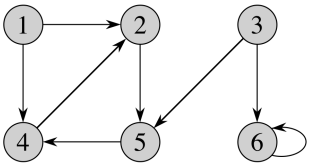
(a)



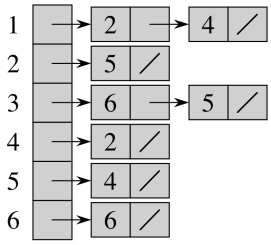
(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Comparison

Adjacency-list better representation for sparse graphs: $|E| \ll |V|^2$.

Adjacency-matrix better representation for dense graphs: $|E|$ is close to $|V|^2$.

Breadth-First Search

Given a graph $G =< V, E >$ and a source vertex s, breadth-first search discover every vertex that is reachable from s through the edges in E.

It can compute the shortest path (in terms of minimal # of edges) from s to all reachable vertices and produce a "breadth-first tree" with root s that contains all reachable vertices.

It works for both directed and undirected graphs.

Search strategy: The algorithm discovers all vertices at distance k from s before discovering any vertex at distance $k+1$.

```
BFS(G, s)
1.  for each vertex u in G.V - {s}
2.      u.color = white
3.      u.p = NIL           // p[u]: parent of u
4.      u.d = infinity      // d[u]: distance from s to u
5.  s.color = gray
6.  s.p = NIL
7.  s.d = 0
8.  Q = empty  //Q is a queue. It will contain all gray vertices and nothing else
9.  EnQueue(Q, s)
10. while Q is not empty
11.     u = DeQueue(Q)
12.     for each v in G.Adj[u]
13.         if v.color == white
14.             v.color = gray
15.             v.p = u
16.             v.d = u.d + 1
17.             EnQueue(Q, v)
18.     u.color = black
```

Three colors are used in the algorithm to help keeping track of the status of each vertex.

- **white:** not yet discovered.
- **gray:** discovered but not yet finished.
- **black:** finished.

A vertex is black (finished) if all its neighbors are discovered.

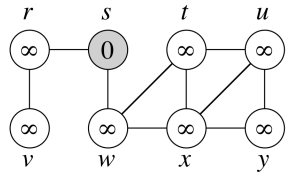
To generate a breadth-first tree, the edges used to discover white vertices will be added to the initial empty tree. That is, a white vertex v is discovered from a gray vertex u through an edge $(u,v) \in E$, we add v and (u,v) to the tree.

A vertex is discovered at most once (from white to gray), it has at most one parent.

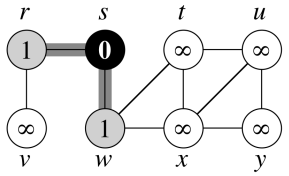
Run-time analysis:

- Initialization (line 1 to line 4): $O(V)$
- The adjacency-list of each vertex is examined at most once: $O(E)$
- Each vertex is EnQueued and DeQueued at most once: $O(V)$
- Totally, the running time is $O(V+E)$

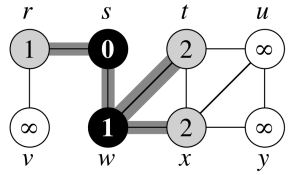
Example:

(a) 

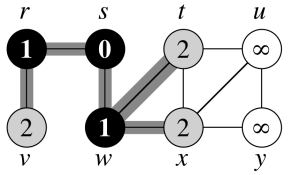
Q | s | 0

(b) 

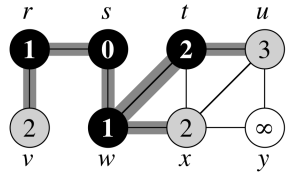
Q | w | r | 1 1

(c) 

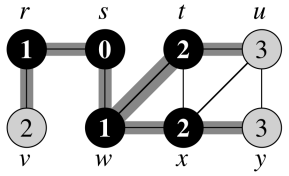
Q | r | t | x | 1 2 2

(d) 

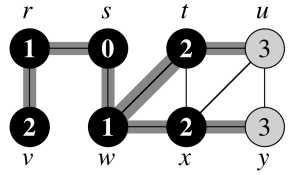
Q | t | x | v | 2 2 2

(e) 

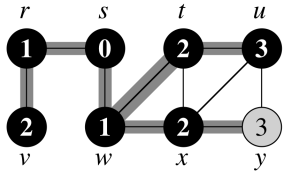
Q | x | v | u | 2 2 3

(f) 

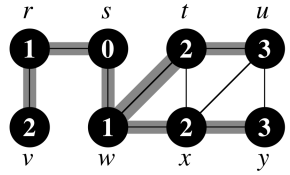
Q | v | u | y | 2 3 3

(g) 

Q | u | y | 3 3

(h) 

Q | y | 3

(i) 

Q $\emptyset$

Question: Why the breadth-first tree is a shortest tree (in terms of # of edges) ?
Answer: because the breadth-first search discovers all vertices at the distance k from s before discovering any vertex at distance $k+1$ from s.
$\Longrightarrow$ Any vertex will be discovered as early as possible. That is, if there are multiple paths from s to a vertex u, u will be discovered through the shortest one.

Depth-First Search

Given a graph $G =< V, E >$, depth-first search discovers all vertices in G to form a depth-first forest (a set of depth-first trees) because the search may be repeated from multiple sources.

Search strategy: The algorithm discovers vertices as deep as possible, if there is no deeper vertices to be discovered, then the search "backtracks" to the predecessor from the current vertex.

Three colors again to indicate the status of each vertex during search: white (not yet discovered), gray (discovered but not yet finished), and black (finished).

Two **timestamps** may be generated for each vertex during search to keep track of the relative ordering of events occurred to each vertex. Two events for each vertex: discover and finish.

- 1st timestamp $d[u]$ for each vertex u records the time when u is discovered (from white to gray).
- 2nd timestamp $f[u]$ for each vertex u records the time when u is finished (from gray to black).
- The time recorded is not a real time. A timer, with initial value 0, is incremented by 1 when an event occurred. The timestamps store the value of the timer when events occurred.

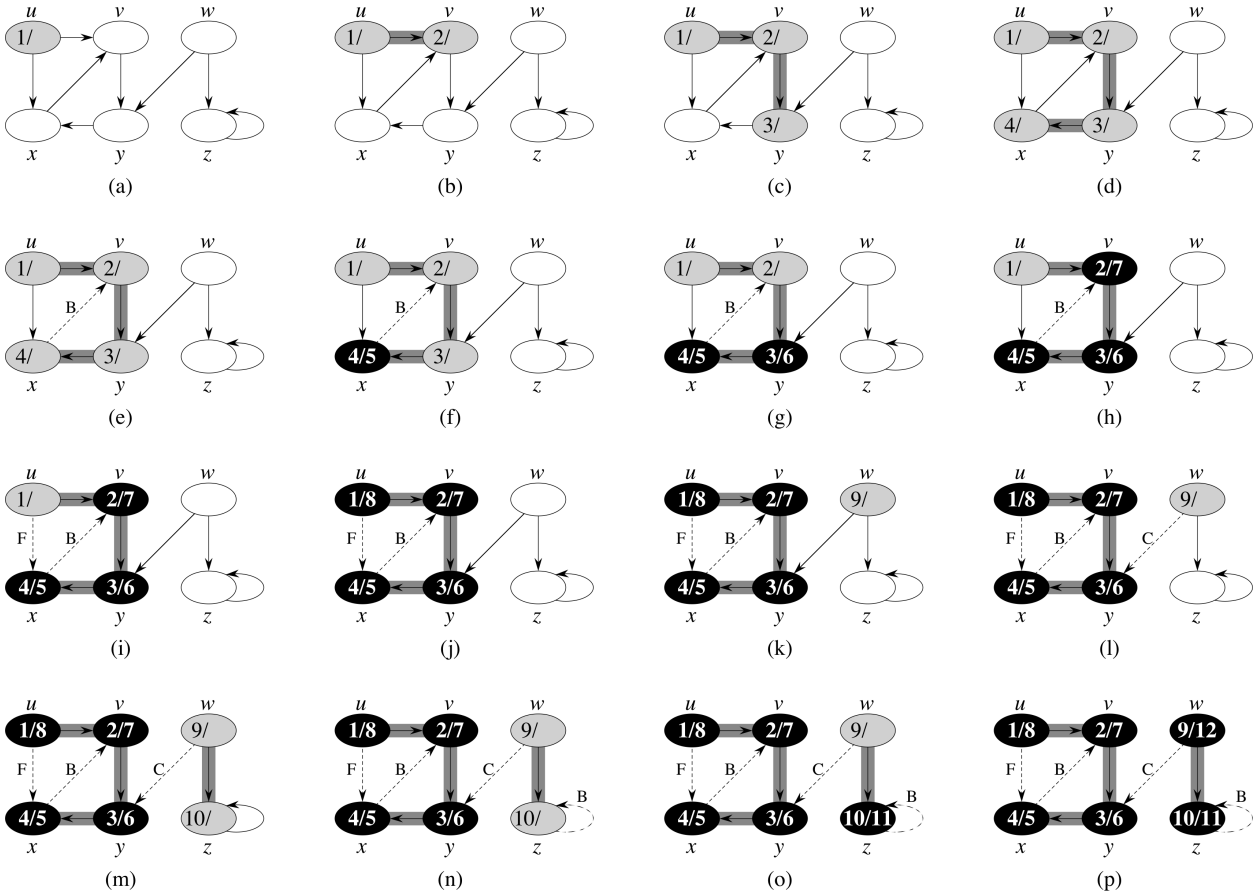
```
DFS(G)
1.  for each vertex u in G.V
2.      u.color = WHITE
3.      u.p = NIL          // u.p: parent of u
4.  time = 0
5.  for each vertex u in G.V
6.      if u.color == WHITE
7.          DFS-Visit(u)

DFS-Visit(G, u)
1.  time = time + 1 // white vertex u has just been discovered
2.  u.d = time
3.  u.color = GRAY
4.  for each v in G.Adj[u] // explore edge (u, v)
5.      if v.color == WHITE
6.            v.p = u
7.            DFS-Visit(G, v)
8.  u.color = BLACK //blacken u, it's finished
9.  time = time + 1
10.  u.f = time
```

Run-time analysis:

- There are two `for` loops in DFS, each takes $\Theta(|V|)$ iterations.
- DFS-Visit is called exactly once for each vertex $v \in V$. Each DFS-Visit(v) examines all vertices in the list $Adj[v]$. Thus, total # of vertices examined is $\sum_{v \in G} Adj[v] = \Theta(|E|)$.
- The running time is $\Theta(|V| + |E|) = \Theta(n+m)$.

Example


u v w
1/
x y z
(a)
1/ 2/
(b)
1/ 2/ 3/
(c)
1/ 2/ 4/ 3/
(d)
B
(e)
4/5
(f)
3/6
(g)
2/7
(h)
F
(i)
1/8
(j)
9/
(k)
C
(l)
10/
(m)
(n)
10/11
(o)
9/12
(p)

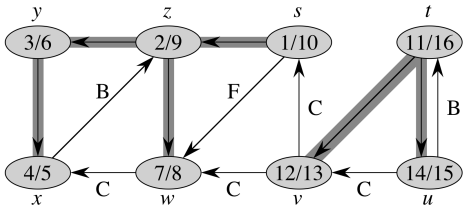
The timestamps have the parenthesis structure:

- For each $d[u]$, we write down a left parenthesis and then u, i.e., "$(u$".
- For each $f[u]$, we write down u and then a right parenthesis, i.e., "$u)$".
- Doing the above mapping, in the order of timestamp values.
  For the above example, the structure is

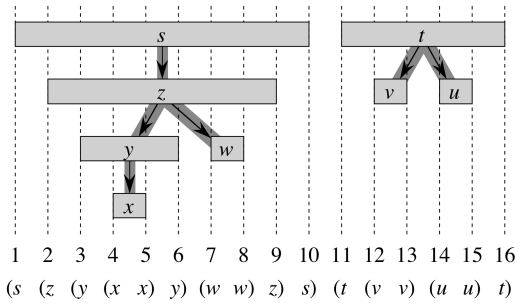
$$(1\ (2\ (3\ 3)\ (4\ (5\ 5)\ 4)\ 2)\ 1)\ (6\ (7\ 7)\ 6)$$

Vertex v is a proper descendant of vertex u in the depth-first forest if, and only if, $d[u] < d[v] < f[v] < f[u]$

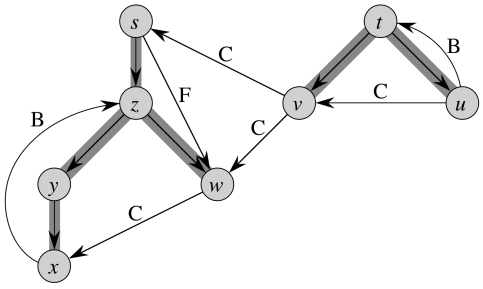
(a)



(b)



(c)



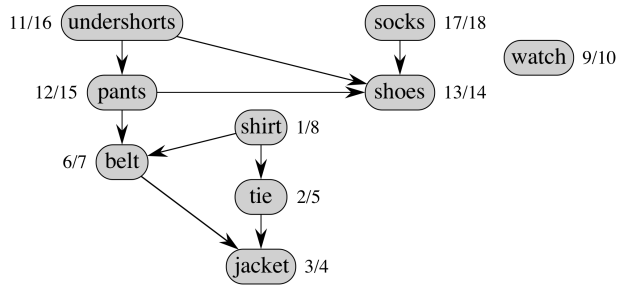
Topological Sort

An application for DFS search on a **directed acyclic graph** or "**DAG**".

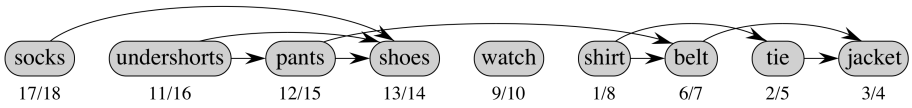
A topological sort of a DAG $G =< V, E >$ is a linear ordering of all vertices in V, where, for all edges $(u, v) \in E$, u appears before v in the ordering.

Example: A DAG shows how a person can get dressed. The topological order suggests a way to get dressed.

(a)



(b)




```
Topological-Sort(G)
1. call DFS(G) to compute f[v] for each vertex v.
2. insert a node at the front of a linked list when the node is finished.
3. return the linked list
```

For the above example, the topological sequence is in shown in part (b) of the figure above.

A given DAG may have many topological sequences. Each sequence corresponds to a different DFS search.