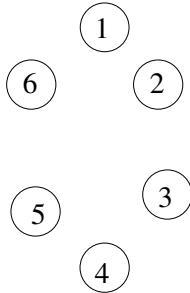


Chapter 20: Elementary Graph Algorithms

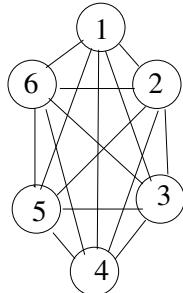
Introduction

- 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 $|V| = n$ and the number of edges $|E| = m$.
- A **network** is made up on **nodes** and **links** between the nodes. Nodes and links are equivalent concepts to vertices and edges. There's no real difference between graphs and networks. However, one can think of a graph as a formal representation (abstraction) of a network in mathematical terms.
- 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.
- Graphs can have unweighted edges (we assume a weight of 1 for each edge in this case) or have weighted edges. A graph with weighted edges is known as a **weighted graph**.

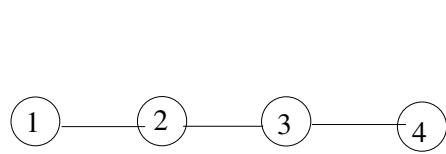
Here are some examples of undirected graphs.



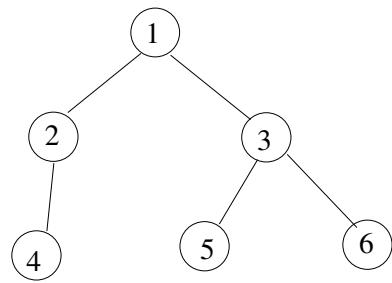
An Empty Graph



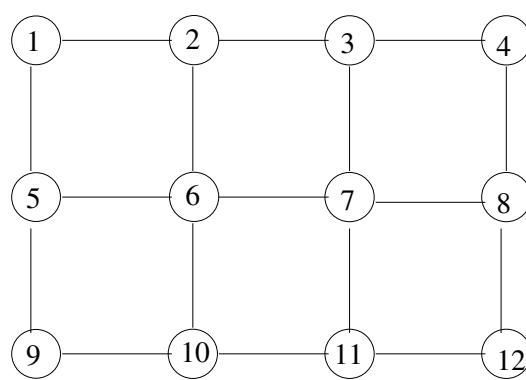
A Complete Graph



A List is a graph

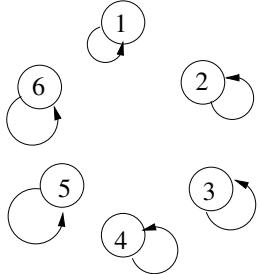


A Tree is a graph

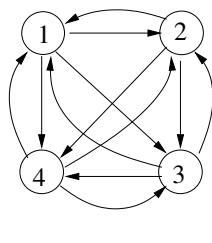


A grid or city map is a graph!

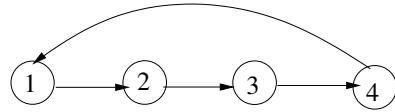
Here are some examples of directed graphs.



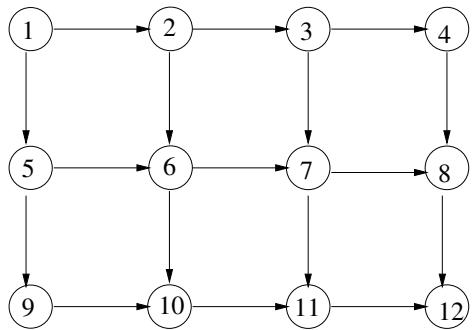
A party of narcissists



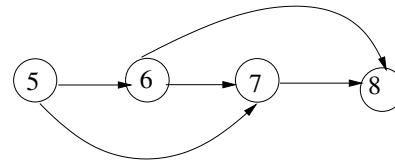
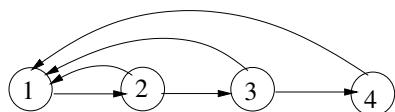
A Complete Graph



A circular list is a graph

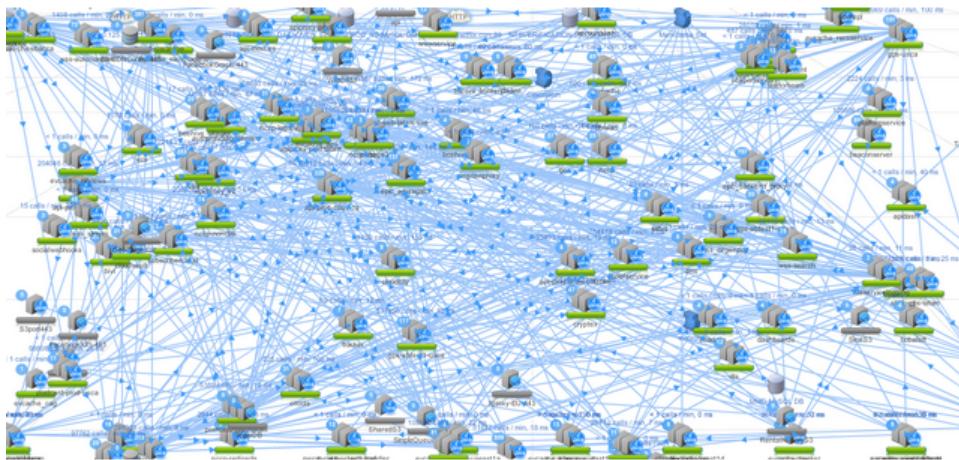


All roads lead to 12!

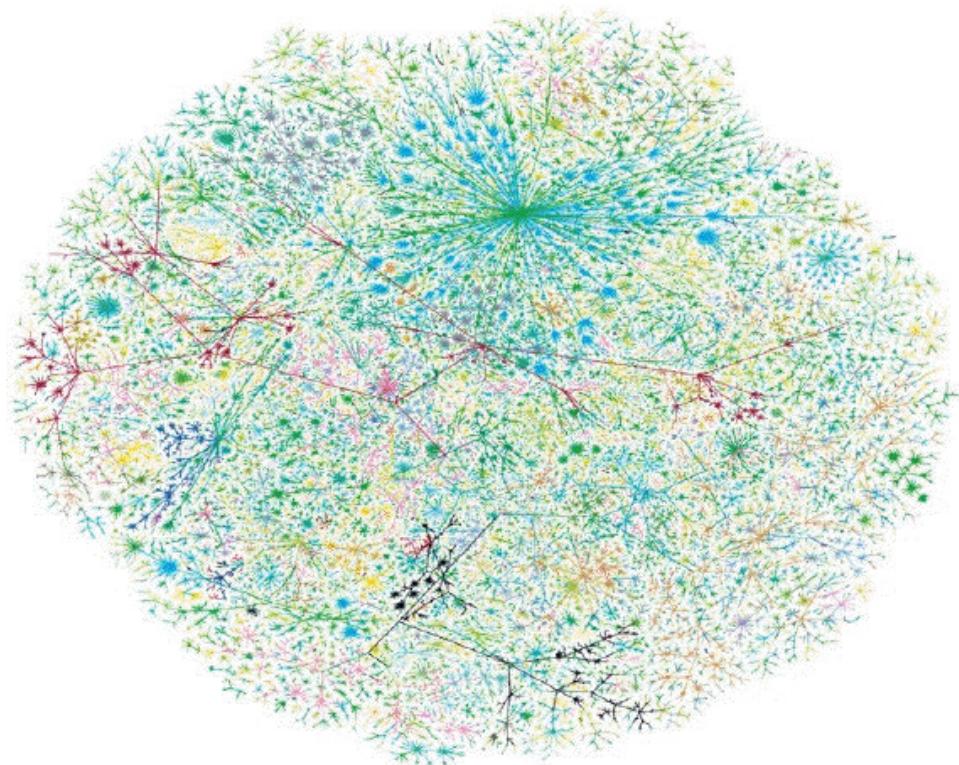


A graph with two connected components

A graph of Netflix servers!



The graph (network) representing the Internet:



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



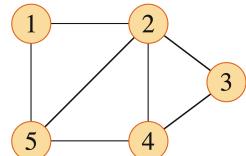
Facebook Friend Graph



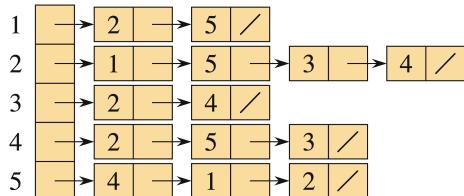
Representations of Graphs

Examples

Adjacency lists and Adjacency Matrix representations:



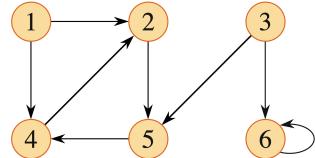
(a)



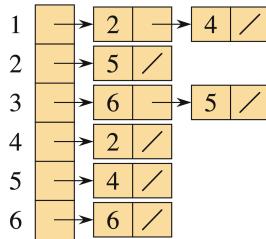
(b)

	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	0	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Adjacency-list

Adjacency list representation:

1. 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.
2. For each vertex $u \in V$, $Adj[u]$ is a list containing all the vertices v that $(u, v) \in E$.

Some properties of adjacency list representation:

- 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?

- **Adjacency-list:** better representation for **sparse graphs**: when $|E| \ll |V|^2$. Typical examples will have $|E| = m = O(n)$.
- A *weighted graph* has a weight function $w : E \rightarrow \mathbb{R}$, 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

Adjacency matrix representation: A graph $G = \langle V, E \rangle$ can be represented by a $|V| \times |V|$ (or $n \times n$) matrix $A = (a_{ij})$, where

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

Some properties of the adjacency matrix representation:

- 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 $|V|^2 = \Theta(n^2)$.
- To determine whether an edge (u, v) is in a given graph takes $\Theta(1)$ time.
- **Adjacency-matrix:** better representation for **dense graphs**: $|E|$ is close to $|V|^2$.
- 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

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

Recommended Exercises: 20.1-1, 20.1-2, 20.1-3, 20.1-5, and 20.1-8.

Breadth-First Search

Given a graph $G = \langle V, E \rangle$ 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.d = infinity      // d[u]: distance from s to u
4.     u.p = NIL            // p[u]: parent of u
5. s.color = gray
6. s.d = 0
7. s.p = NIL
8. Q = empty    //Q is a queue. It will contain only gray vertices
9. EnQueue(Q, s)
10. while Q is not empty
11.     u = DeQueue(Q)
12.     for each v in G.Adj[u]    //search the neighbors of u
13.         if v.color == white // is V being discovered now?
14.             v.color = gray
15.             v.d = u.d + 1
16.             v.p = u
17.             EnQueue(Q, v)    //v is now on the frontier
18.     u.color = black        //u is now behind the frontier

```

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.

Loop invariant: At the test in line 10, the queue Q consists of the set of gray vertices.

Initialization: Prior to the start of the loop, the only gray vertex is the source vertex s and it is in the queue.

Maintenance: The loop invariant is maintained because whenever a vertex is painted gray (in line 14), it is also enqueued (in line 17) and whenever a vertex is dequeued (in line 110, it is also painted black (in line 18).

Termination: The loop terminates when the queue is empty and there are no more gray vertices.

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.

We can terminate a BFS early, before the queue becomes empty. Because each vertex is discovered at most once and receives a finite d value only when it is discovered, the algorithm can terminate once every vertex has a finite d value. This can be done by keeping count of how many vertices have been discovered.

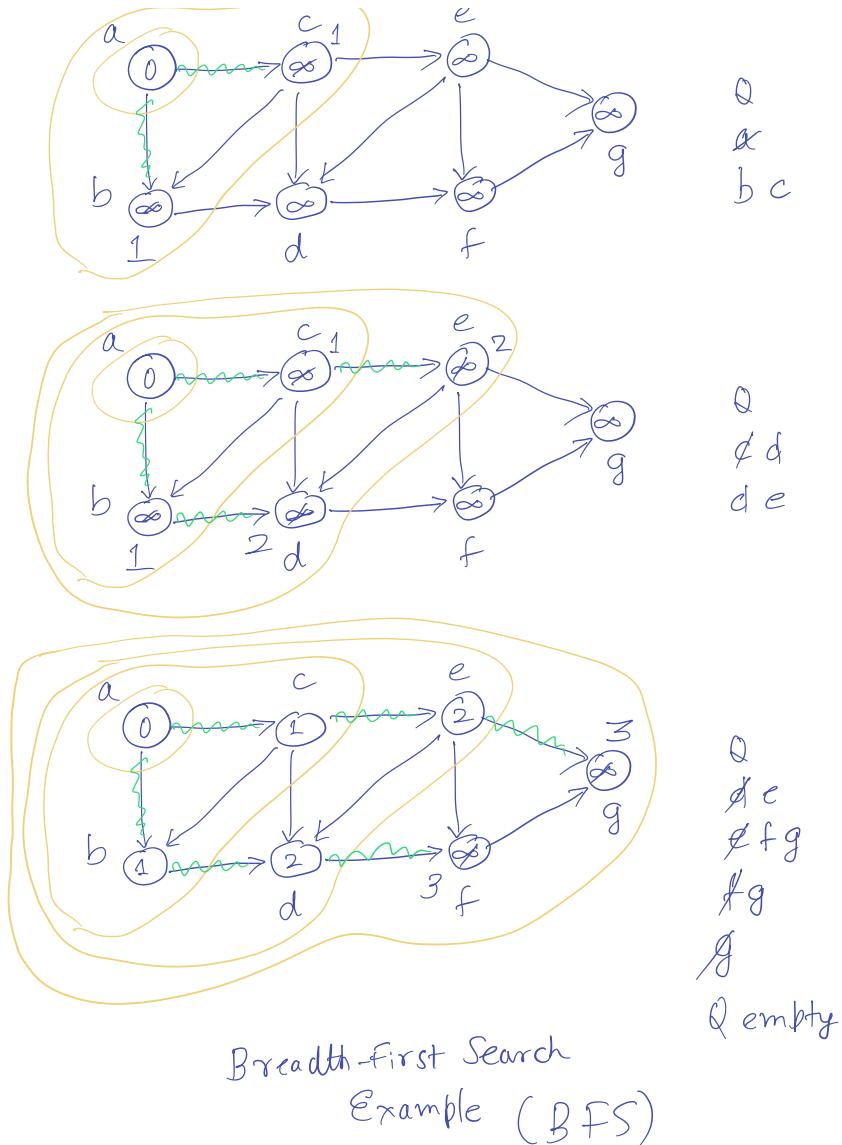
Run-time analysis:

- Initialization (line 1 to line 9): $O(V)$
- The adjacency-list of each vertex is examined at most once: $O(E)$
- Each vertex is Enqueued and Dequeued at most once: $O(V)$
- The total running time is $O(V + E) = O(m + n)$

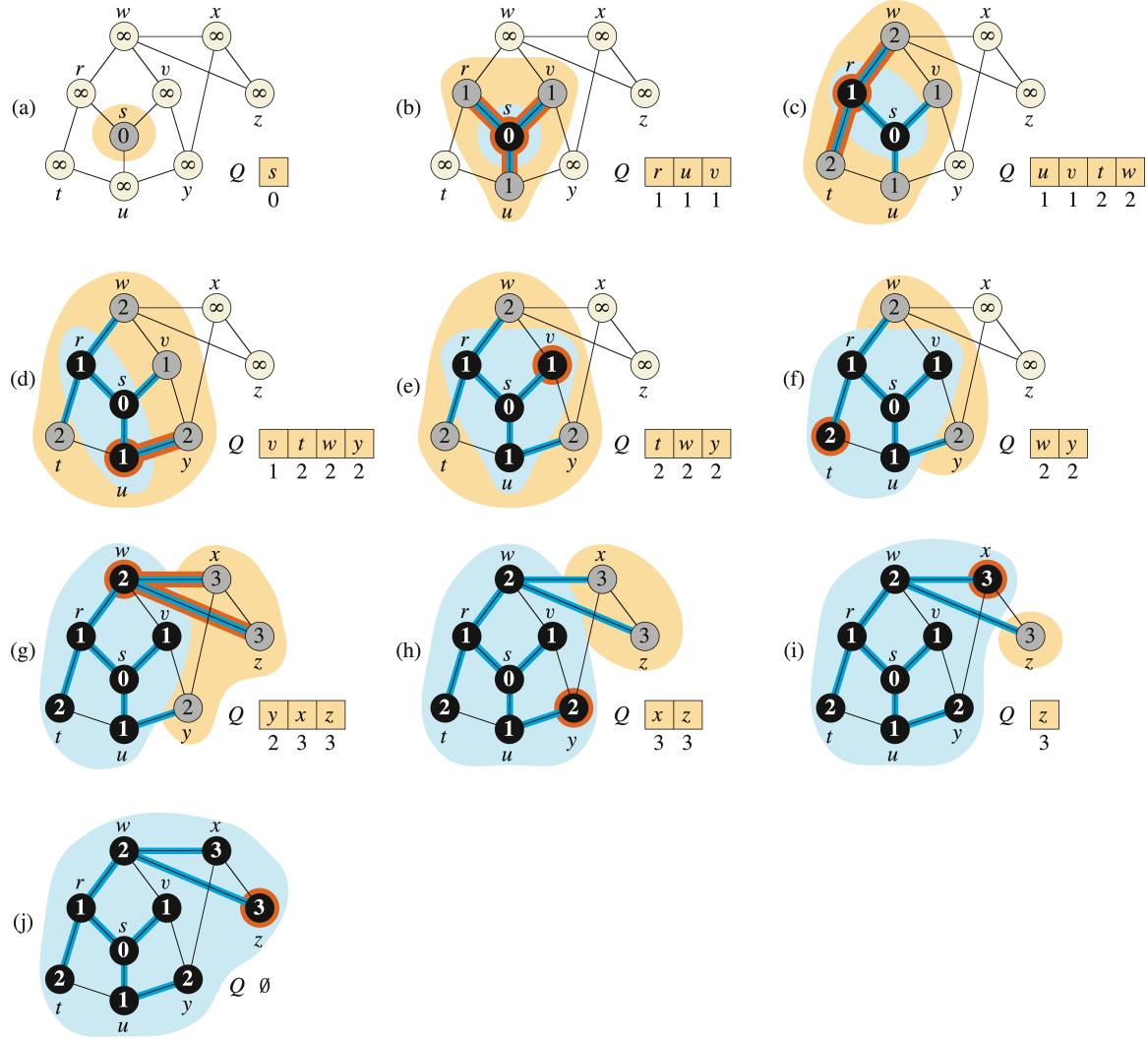
Once BFS has completed the BFS tree, the following recursive procedure can be used to print out the vertices on the shortest path from s to v .

```
PRINT-PATH(G, s, v)
1. if v == s
2.     print s
3. elseif v.p == NIL
4.     print "no path from " s "to " v " exists"
5. else PRINT-PATH(G, s, v.p)
6.     print v
```

Example 1:



Example 2:



Question: Why is the breadth-first tree 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 .

⇒ 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.

Exercise: What happens when we do a BFS on a binary tree? Create a binary tree with 4 levels and run BFS on it. What happens when we run BFS on any arbitrary tree?

Recommended Exercises: 20.2-1, 20.2-2, 20.2-4.

Depth-First Search

Given a graph $G = \langle V, E \rangle$, 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 **timesteps** 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)

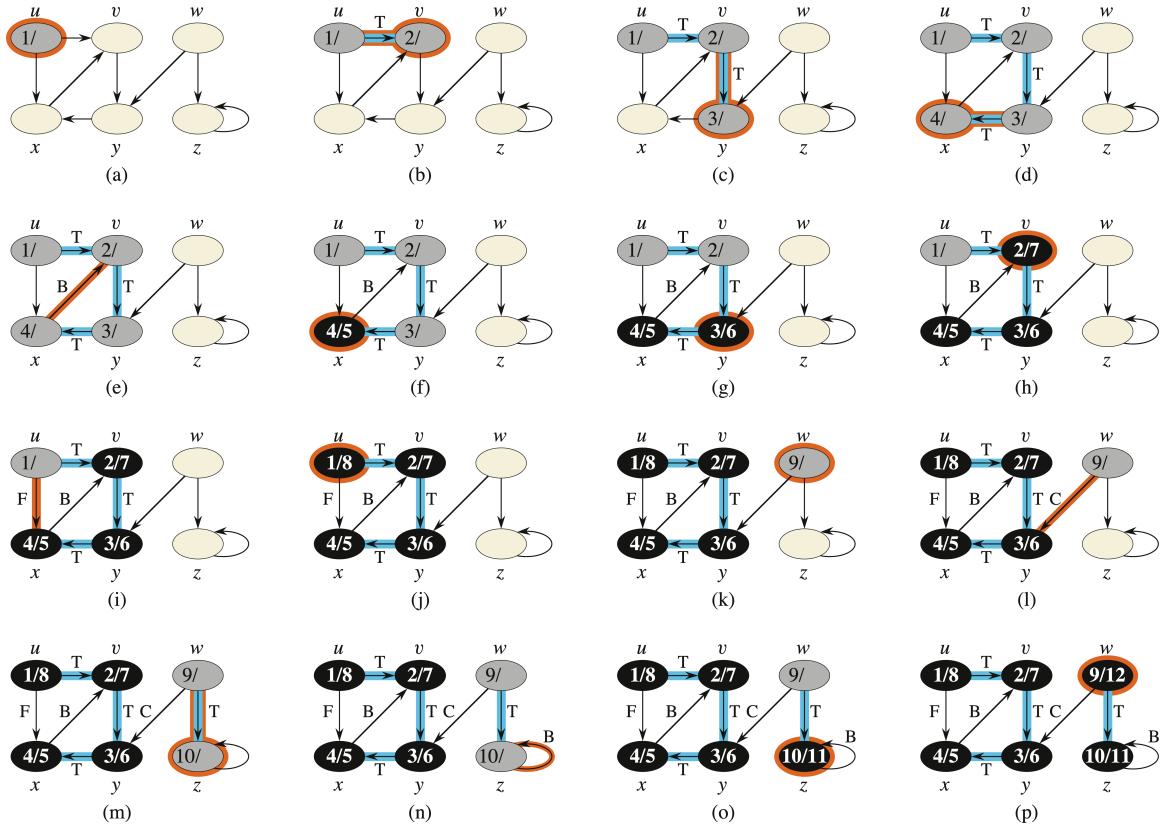
DFS-Visit (G, u)
1. time = time + 1 // white vertex u has just been discovered
2. u.d = time
3. u.color = GRAY
4. for each v in G.Adj[u] // explore edge (u, v)
5.     if v.color == WHITE
6.         v.p = u
7.         DFS-Visit (G, v)
8. time = time + 1
9. u.f = time
10. u.color = BLACK //blacken u, it's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

```

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 $\text{DFS-Visit}(v)$ examines all vertices in the list $\text{Adj}[v]$. Thus, total # of vertices examined is $\sum_{v \in G} \text{Adj}[v] = \Theta(|E|)$.
- The running time is $\Theta(|V| + |E|) = \Theta(n + m)$.

Example 1:



Example 2:

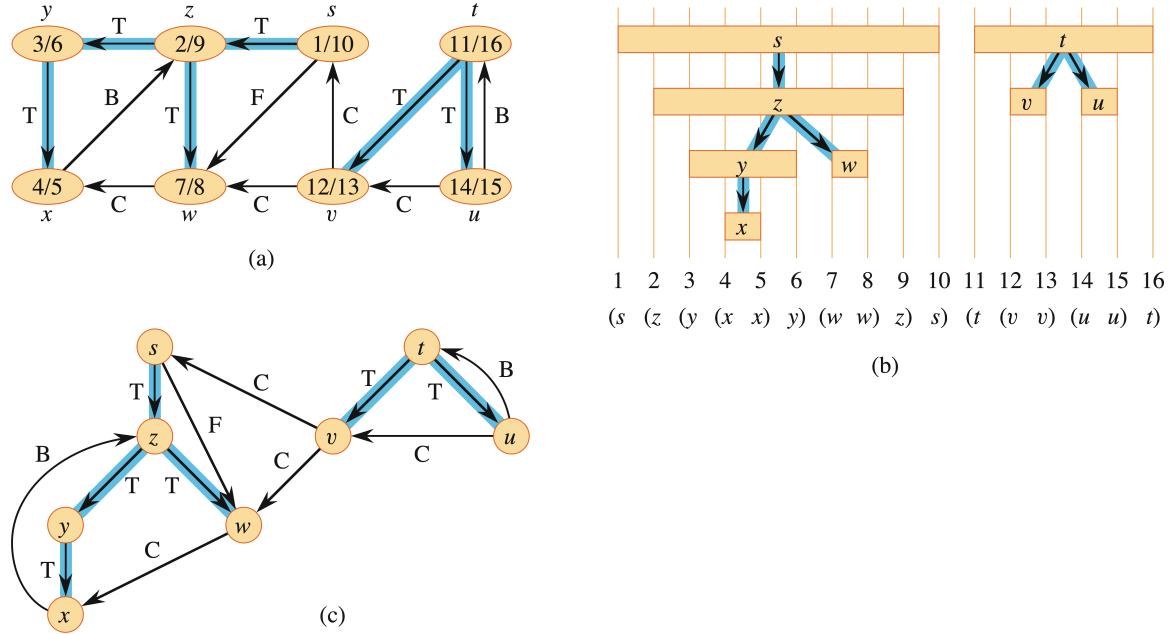
The timestamps have the parenthesis structure:

- 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, $d[u] < d[v] < f[v] < f[u]$



Recommended Exercises: 20.3-2, 20.3-3, 20.3-5, 20.3-9, 20.3-12.

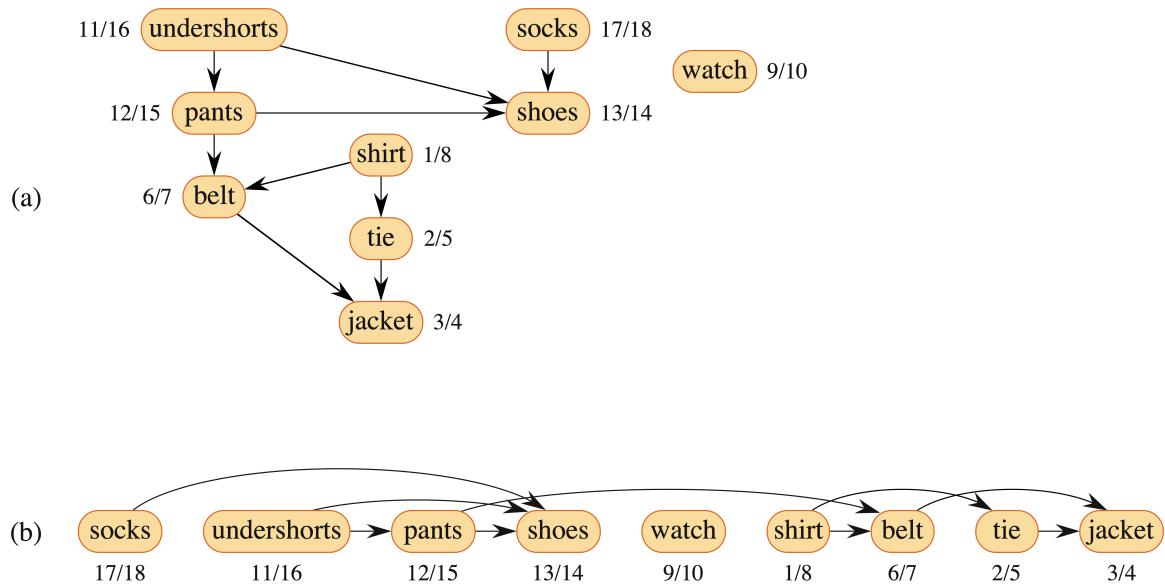
Topological Sort

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

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

Topological sorting forms the basis of linear-time algorithms for finding the critical path of the project, a sequence of milestones and tasks that controls the length of the overall project schedule. Other examples include ordering of formula cell evaluation when recomputing formula values in spreadsheets, determining the order of compilation tasks to perform in build tools (such as make), and data serialization. It is also used to decide in which order to load tables with foreign keys in databases.

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



Topological-Sort (G)

1. call $\text{DFS}(G)$ 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 shown in part (b) of the figure above.

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

