COMP20230: Data Structures & Algorithms
Lecture 16: Graphs (1)

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Outline

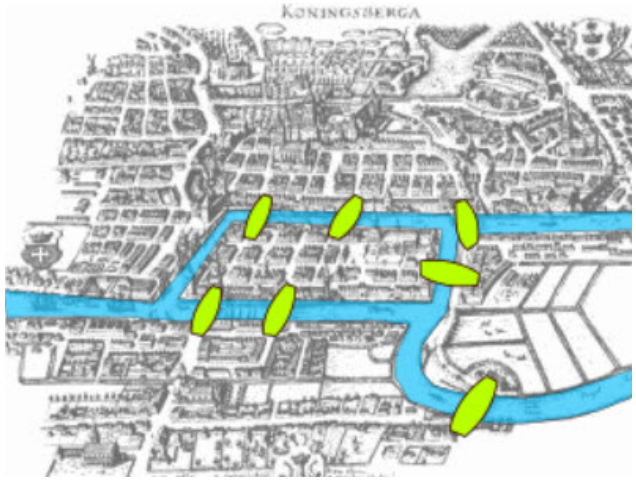
What is a graph?

- Definitions
- Graph ADT
- Array-based Representation
- Traversing a graph: DFS and BFS

Take home message

Like a Tree ADT, a graph is a *non-linear* data type but cycles and non-connected components are allowed.

Origin of Graphs



History

G. Boole pioneered the development of symbolic logic, and he introduced many of the basic set notations in a book published in 1854. Modern set theory was created by G. Cantor during the period 1874–1895. Cantor focused primarily on sets of infinite cardinality. The term "function" is attributed to G. W. Leibniz, who used it to refer to several kinds of mathematical formulas. His limited definition has been generalised many times. Graph theory originated in 1736, when L. Euler proved that it was impossible to cross each of the seven bridges in the city of Koñigsberg exactly once and return to the starting point.

Text from Cormen p.1182. Image from https://www.mathsisfun.com/activity/seven-bridges-konigsberg.html

Graphs

A collection of vertices (nodes), joined together by edges.
No definite beginning or end.

Searching

Many algorithms begin by searching (e.g. DFS, BFS) their input graph to obtain this structural information.

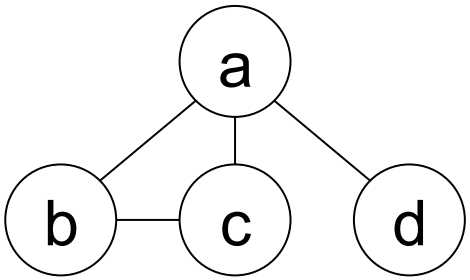
Representations of graphs

Two common methods: adjacency lists and adjacency matrices

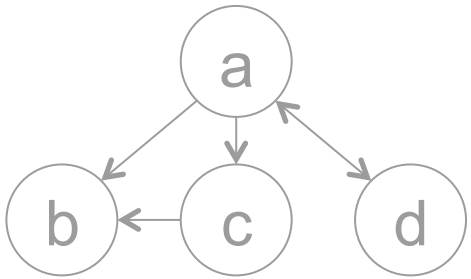
Graphs

The Graph ADT represents the mathematical concepts of:

- directed graph
- undirected graph (we will focus on this type today)



undirected graph

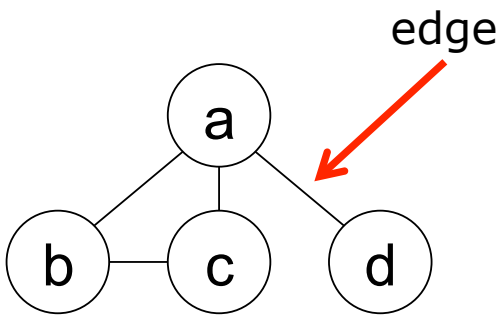


directed graph

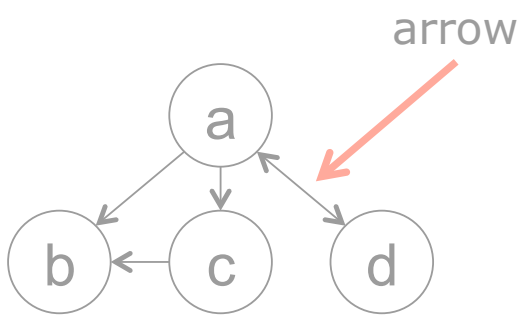
Graphs

Connections between vertices

- **undirected graph**: edges, arcs
- **directed graph**: arrows, directed edges, directed arcs



undirected graph



directed graph

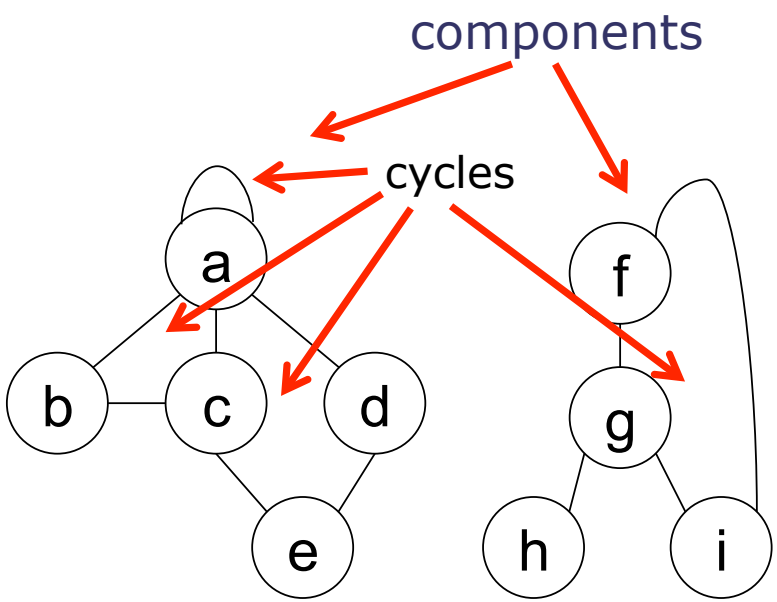
Graphs vs Trees

Unlike trees, graphs

can have cycles

can be composed of different components

Example undirected graph with cycles and components:

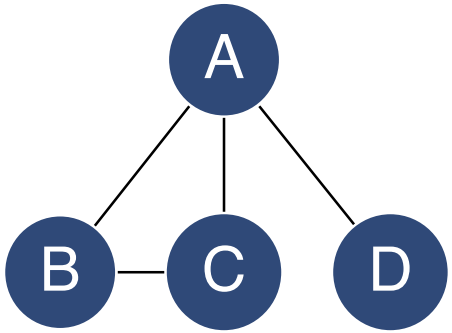


Operations of the Graph ADT

- `adjacent(G, x, y)`: tests whether there is an edge from the vertices x to y;
- `neighbours(G, x)`: lists all vertices y such that there is an edge from the vertices x to y;
- `add_vertex(G, x)`: adds the vertex x, if it is not there;
- `remove_vertex(G, x)`: removes the vertex x, if it is there;
- `add_edge(G, x, y)`: adds the edge from the vertices x to y, if it is not there;
- `remove_edge(G, x, y)`: removes the edge from the vertices x to y, if it is there;

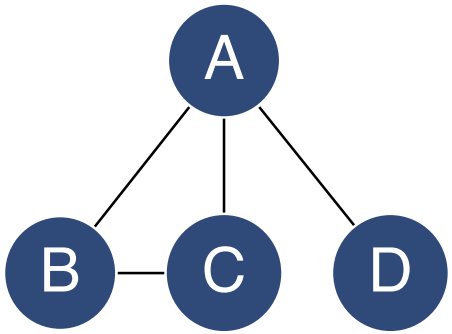
adjacent(G, x, y)

adjacent(G, b, c) = True
adjacent(G, c, d) = False



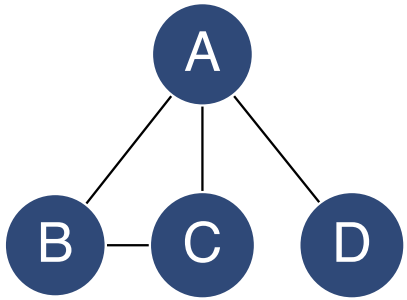
neighbours(G,x)

neighbours(G, a) = $\{b, c, d\}$
neighbours(G, b) = $\{a, c\}$
neighbours(G, d) = $\{a\}$

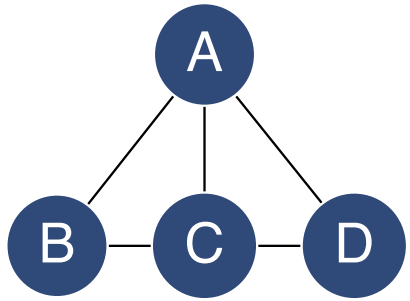


add_edge(G, x, y)

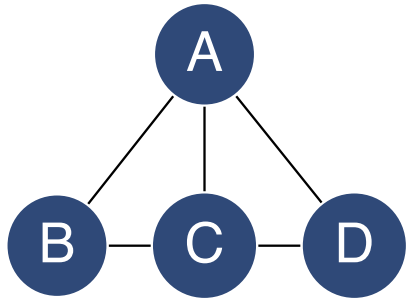
Original Graph:



add_edge(G, c, d)

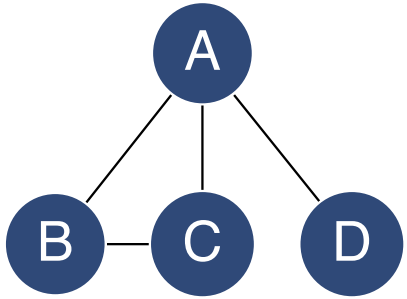


add_edge(G, a, b)

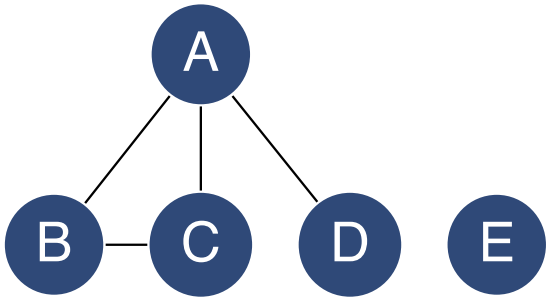


add_vertex(G, x)

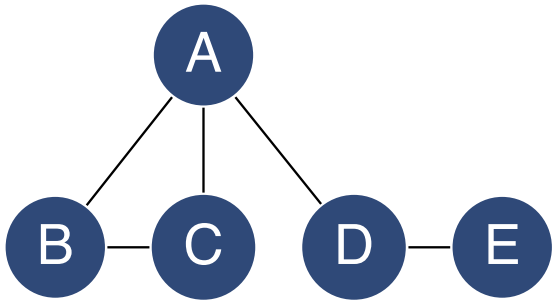
Original Graph:



add_vertex(G, e)

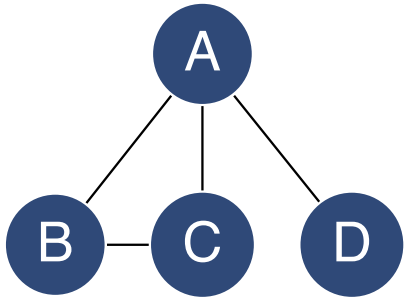


add_edge(G, d, e)

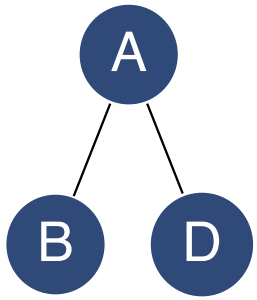


remove_vertex(G, x)

Original Graph:

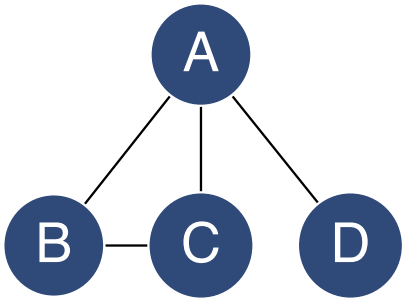


remove_vertex(G, c)

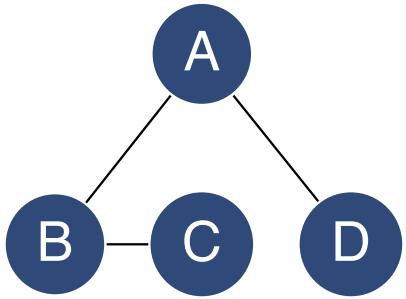


`remove_edge(G, x, y)`

Original Graph:



`remove_edge(G, a, c)`



Computational Representation of Graphs

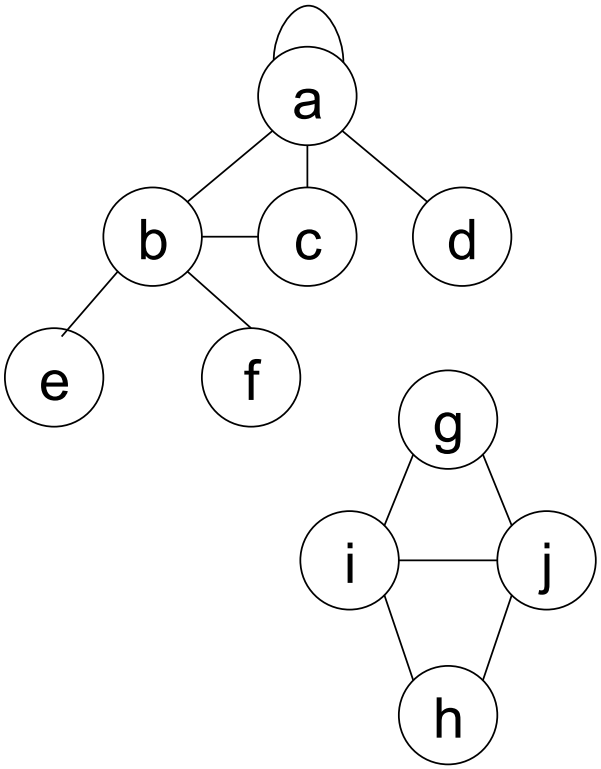
Computational Representation of Graphs

How to convert the visualisation into a computer data structure.
`G(V,E)`: Graphs are a function of their **Vertices** and **Edges**

(Again) data will dictate the best way to represent the graph

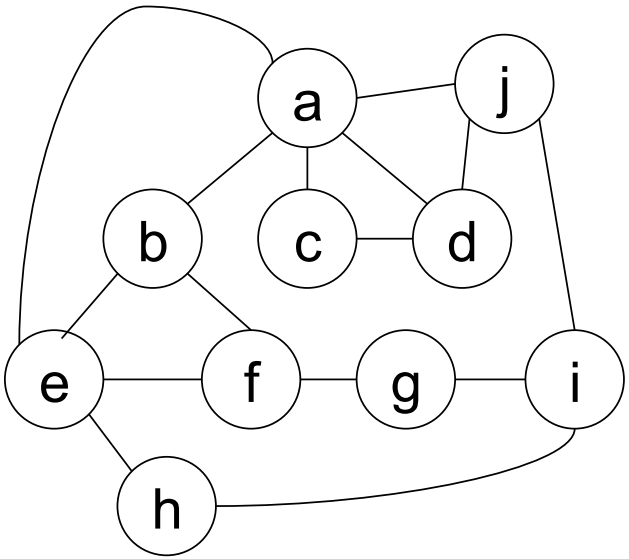
An **adjacency-list** provide a compact representation for sparse graphs, i.e. those for which $|E|$ is much less than $|V|^2$
An **adjacency-matrix** is better when a graph is dense, i.e. $|E|$ is close to $|V|^2$

Adjacency List Example



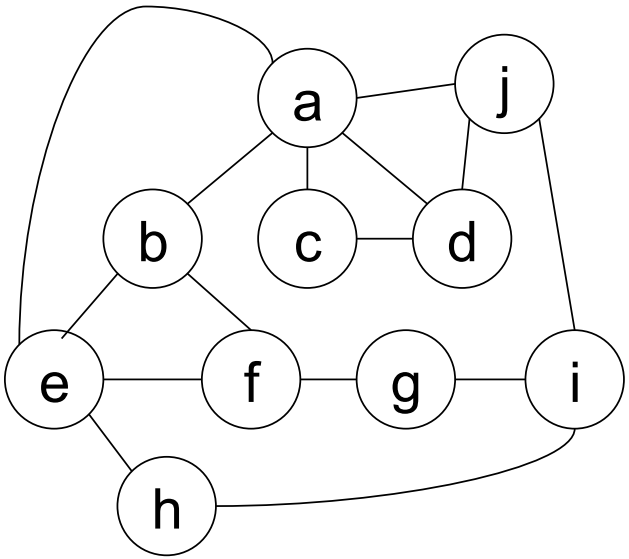
a	a, b, c, d
b	a, c, e, f
c	a, b
d	a
e	b
f	b
g	i, j
h	i, j
i	g, h, j
j	g, h, i

Adjacency List Exercise



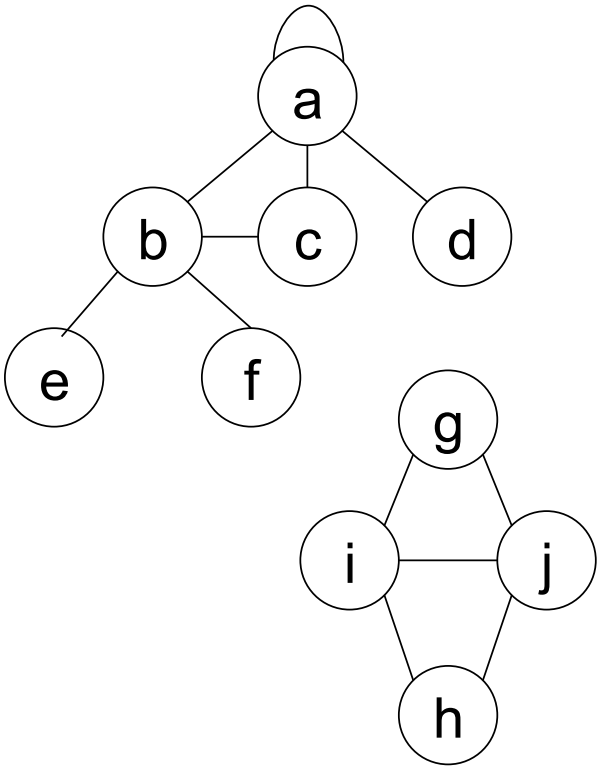
a	
b	
c	
d	
e	
f	
g	
h	
i	
j	

Adjacency List Exercise



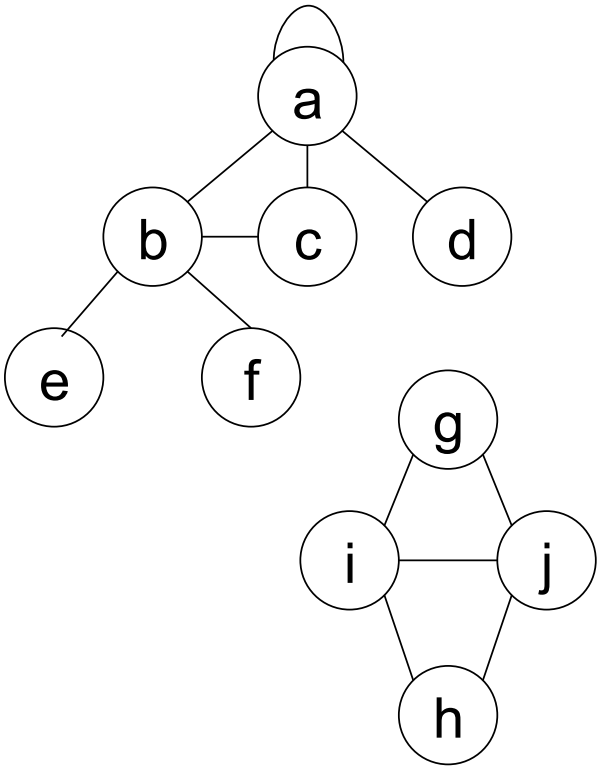
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Adjacency Matrix Example



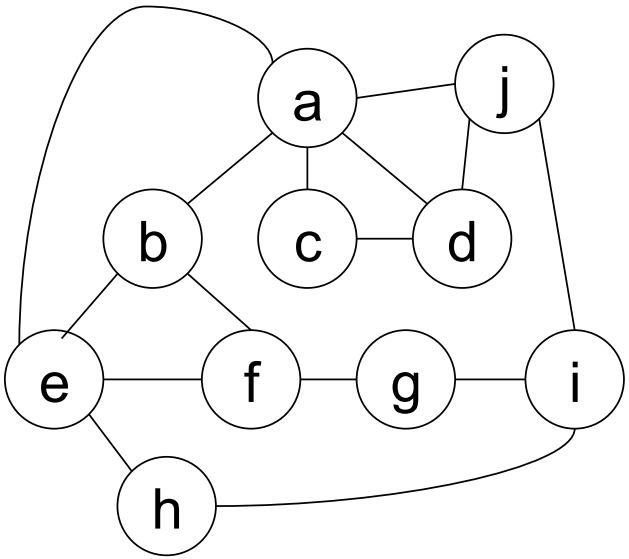
	a	b	c	d	e	f	g	h	i	j
a	1	1	1	1						
b	1		1		1	1				
c	1	1								
d	1									
e		1								
f		1								
g									1	1
h									1	1
i							1	1		1
j							1	1	1	

Adjacency Matrix Example – Note Symmetry



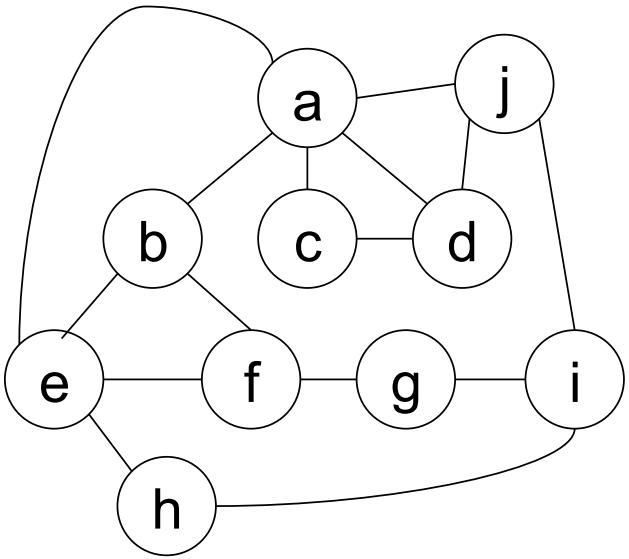
	a	b	c	d	e	f	g	h	i	j
a	1	1	1	1						
b			1		1	1				
c										
d										
e										
f										
g									1	1
h									1	1
i										1
j										

Adjacency Matrix Exercise



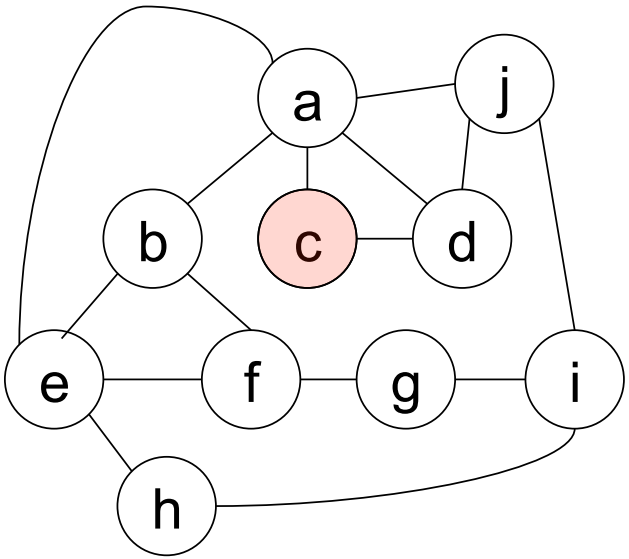
	a	b	c	d	e	f	g	h	i	j
a										
b										
c										
d										
e										
f										
g										
h										
i										
j										

Adjacency Matrix Exercise



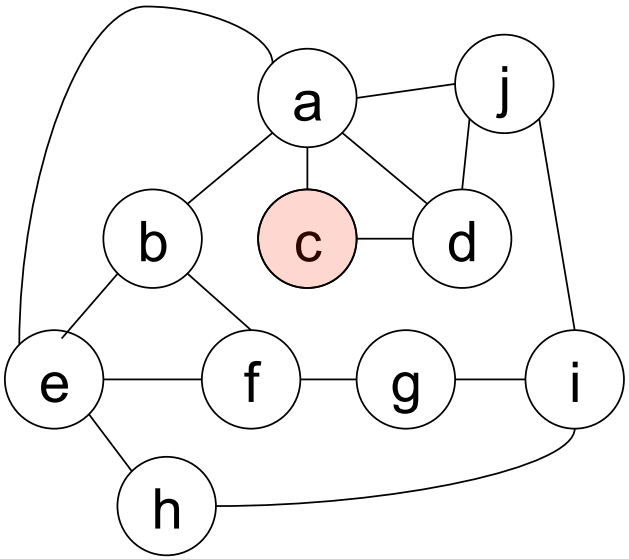
	a	b	c	d	e	f	g	h	i	j
a		1	1	1	1					1
b	1				1	1				
c	1			1						
d	1		1							1
e	1	1				1		1		
f		1			1		1			
g						1			1	
h					1				1	
i							1	1		1
j	1			1					1	

Depth First Search



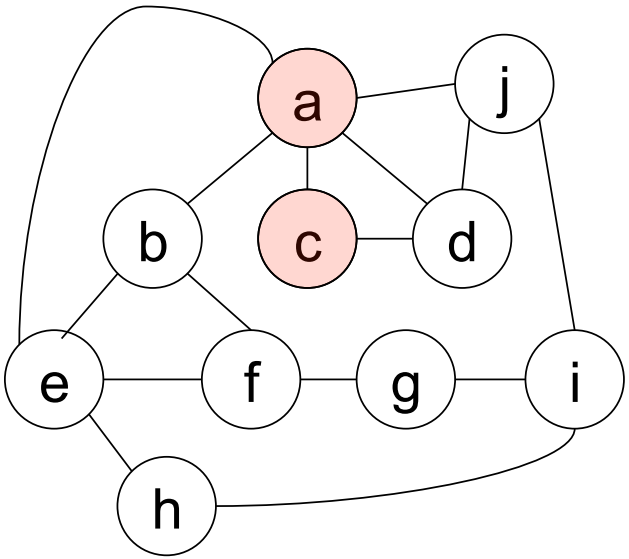
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



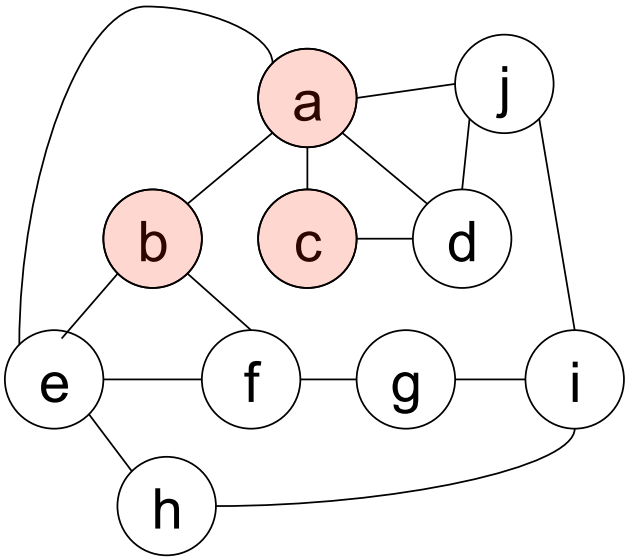
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



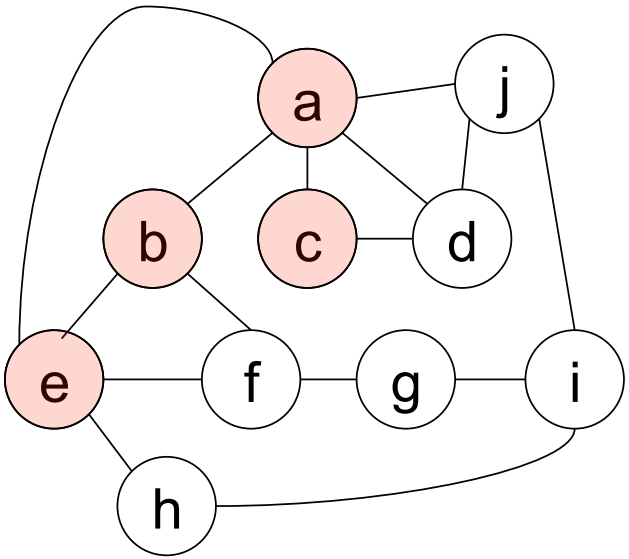
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



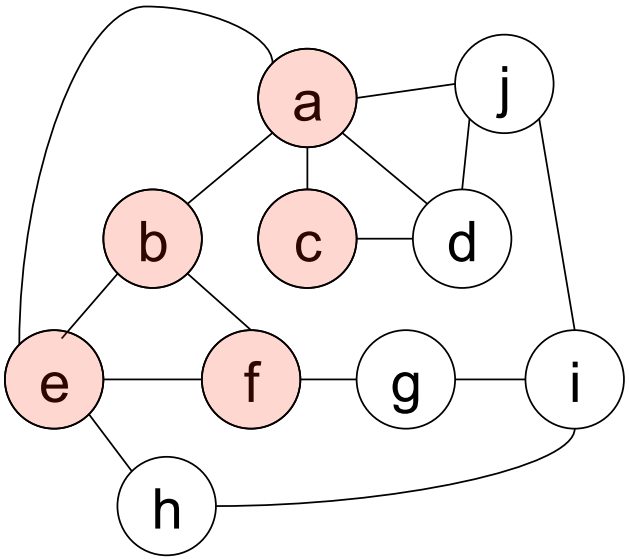
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



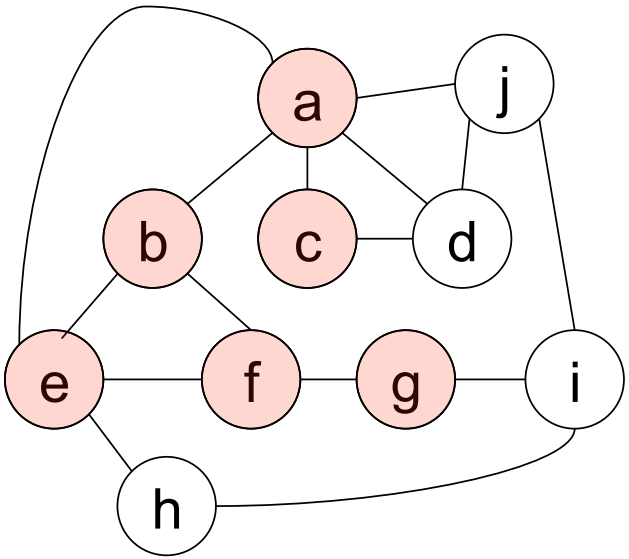
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



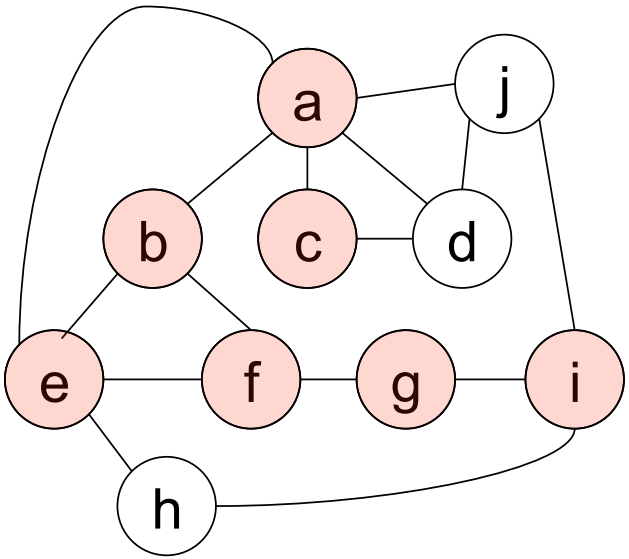
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



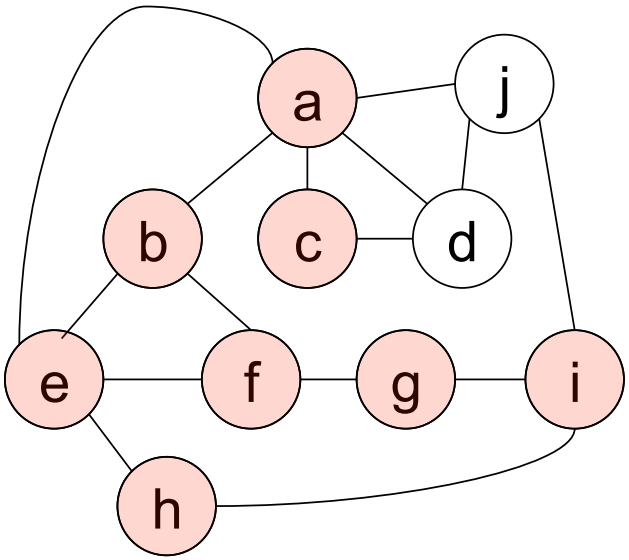
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



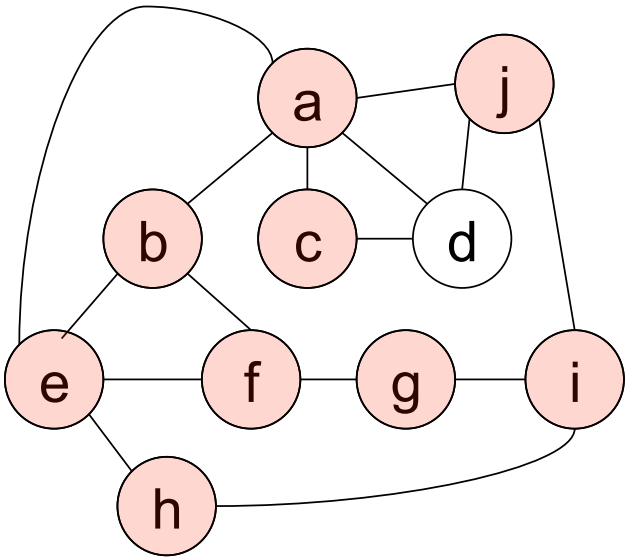
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



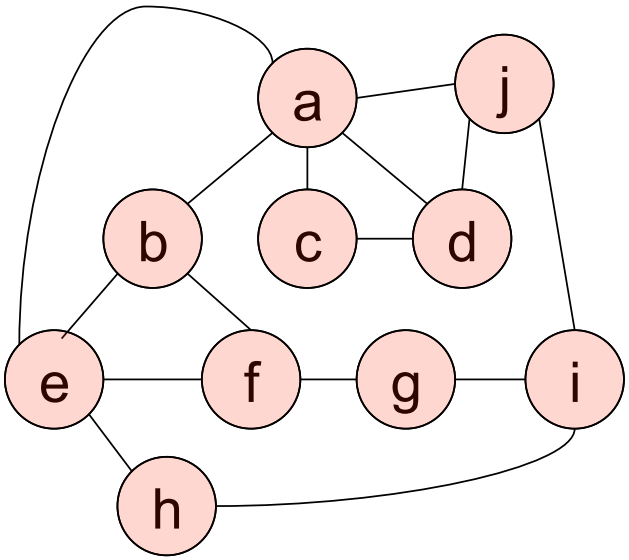
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search (non-recursive)

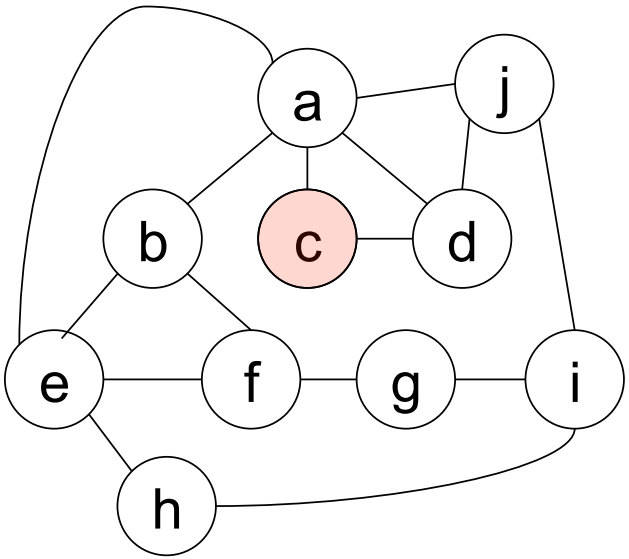
```
Algorithm dfs:
Input:  a Graph g and a node n
Output:  the procedure explores every node starting
from n
to_visit ← empty stack
add n to to_visit
visited ← empty sequence (?)
while to_visit is not empty do
    current ← pop to_visit # get the first element
    push all neighbours of current (not in visited)
                                        to to_visit
    do something on current
endfor
```

Depth First Search (recursive)

```
Algorithm dfs:
Input:  a Graph g and a node n
Output:  the function explores every node from n
flag n as visited
do something
for each neighbour n_c of of n which is not visited do
    dfs(n_c)
endfor
```

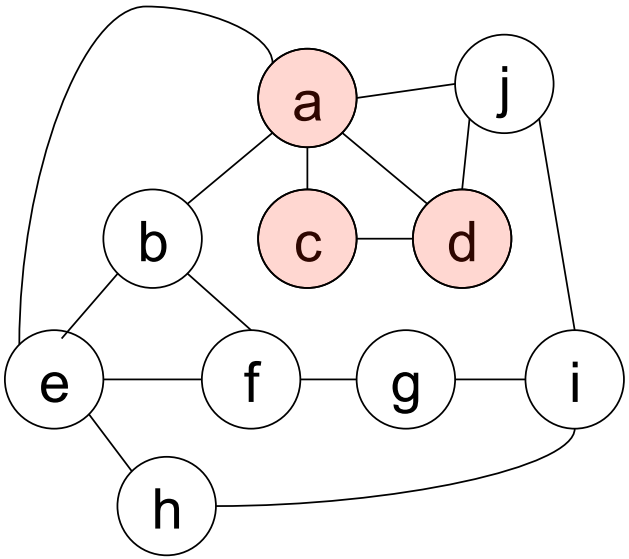
Compare the traversal (search) algorithms for graphs to those for trees. Starting point needs to be chosen for graph as there is no root node but otherwise similar.

Breadth First Search



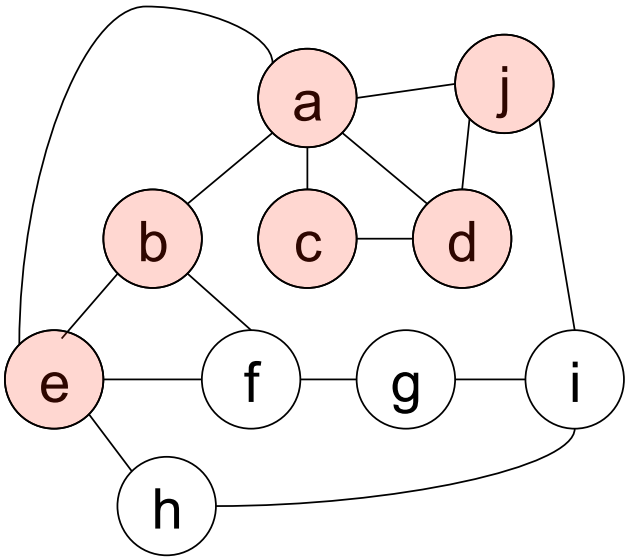
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



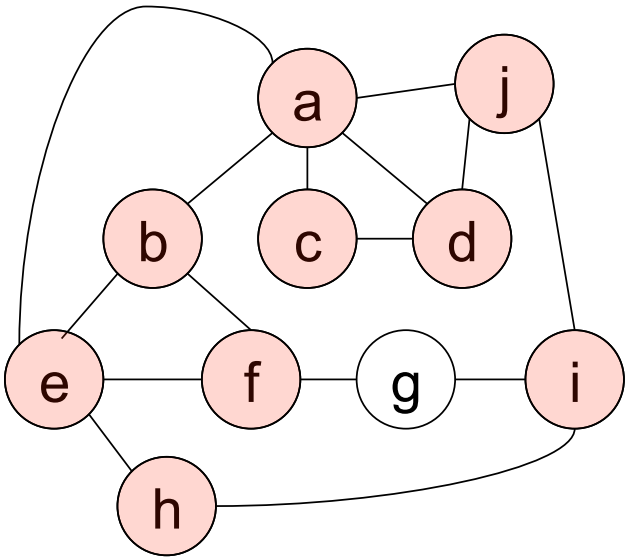
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



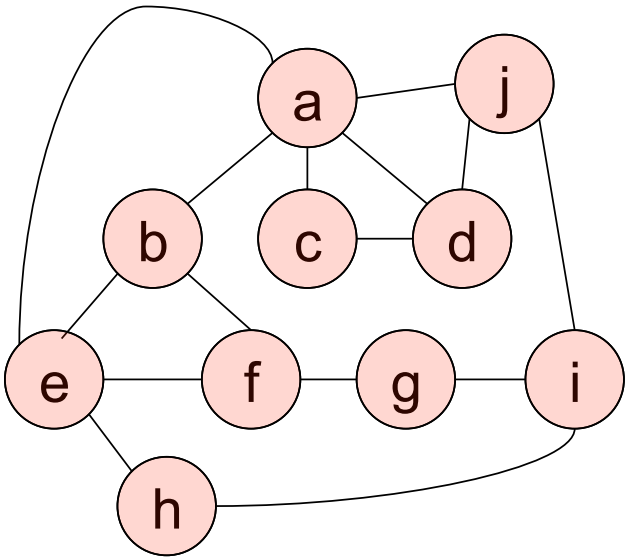
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

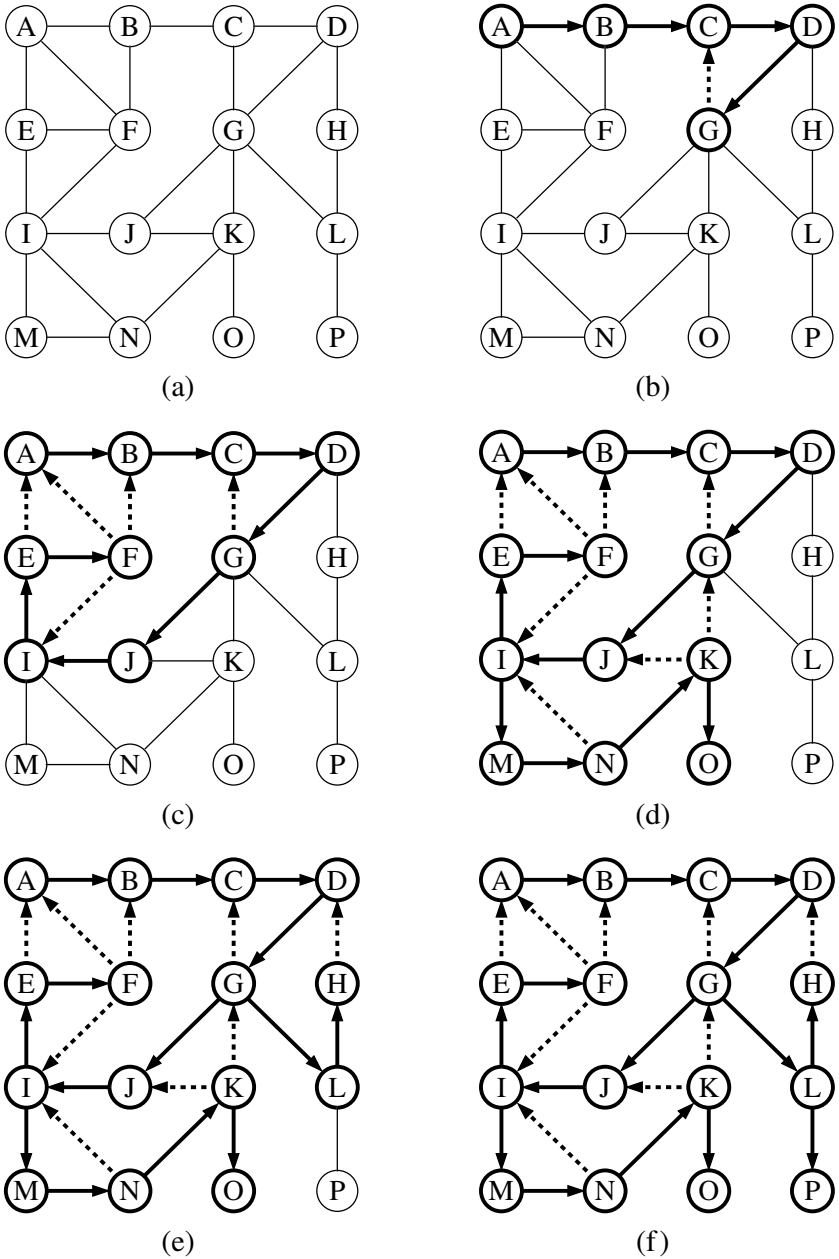
Breadth First Search (non-recursive)

```
Algorithm bfs:
Input:  a Graph g and a node n
Output:  the procedure explores every node of g from
n
to_visit is a queue
enqueue n
visited is a sequence (?)
while to_visit is not empty do
    n_current ← dequeue to_visit
    add n_current to visited
    for each neighbour n_c of n_current that is not
                                              visited do
        enqueue n_c to to_visit
    endfor
    do something on n_current
endwhile
```

Breadth First Search (recursive)

```
Algorithm bfs:
Input:  a Graph g a queue q (originally having a
starting node)
Output:  explores every node from the starting node
if q is empty then # base case
    do something (?)
else
    current ← dequeue q
    flag current as visited
    for each neighbour n_c of current not visited do
        enqueue n_c
    endfor
    do something
    bfs(q)
endif
```

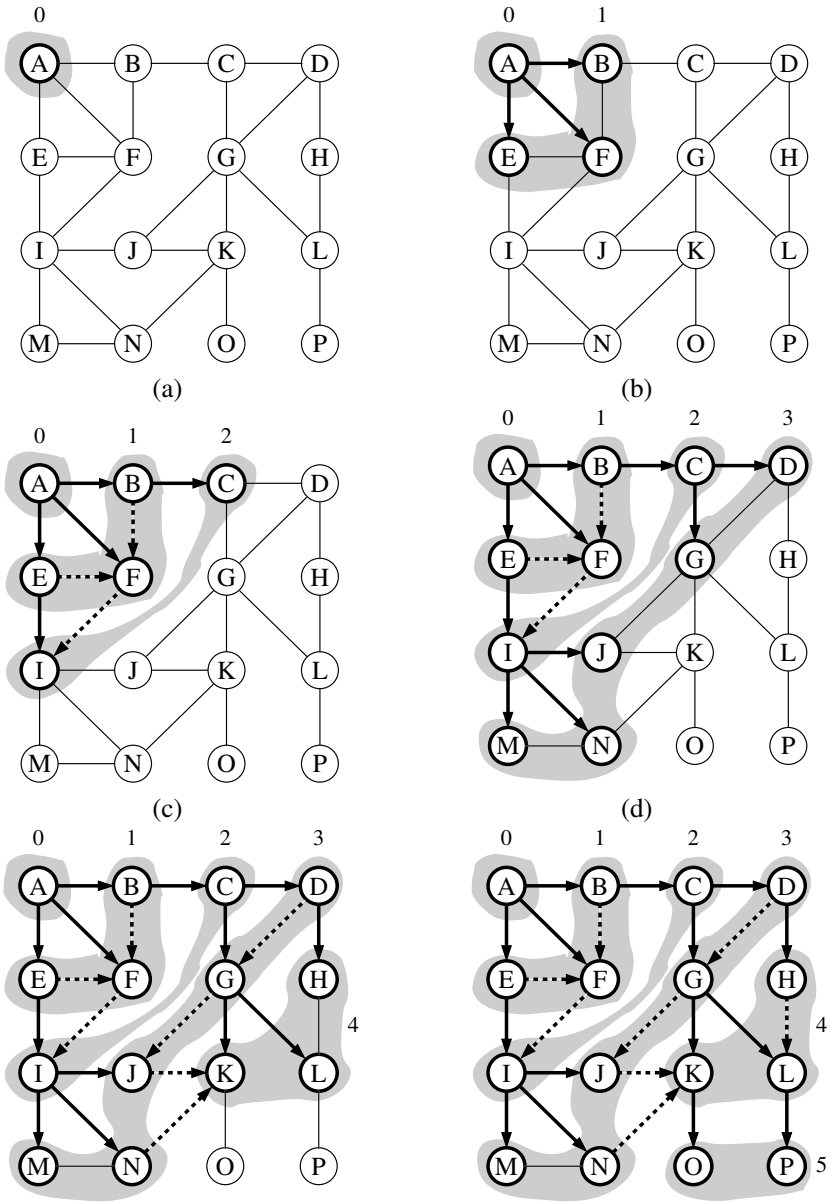
DFS example



Example from Goodrich of DFS traversal

Undirected graph starting at vertex A. We assume that a vertex's adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph; (b) path of tree edges, traced from A until back edge (G,C) is examined; (c) reaching F, which is a dead end; (d) after backtracking to I, resuming with edge (I,M), and hitting another dead end at O; (e) after backtracking to G, continuing with edge (G,L), and hitting another dead end at H; (f) final result.

BFS example



Example from Goodrich of BFS traversal

The edges incident to a vertex are considered in alphabetical order of the adjacent vertices. The discovery edges are shown with solid lines and the nontree (cross) edges are shown with dashed lines: (a) starting the search at A; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

Reflection

Question

Why might you choose a DFS rather than a BFS or vice-versa?

Can you think of an examples to explain your answer?

You will find a lot of answers if you Google it so have a think about it first!

Summary

Graph ADT is similar to Tree ADT but cycles and non-connected components allowed

- **Two array based representations:** adjacency list and adjacency matrix
- **Traversal methods:** DFS and BFS

Next: Weighted Graphs, but first an aside...

Remember trying to work out the order for brushing your teeth or inflating a tyre?

Directed Graph Topological Sort Example

See page 613 of Cormen for more detail if interested.

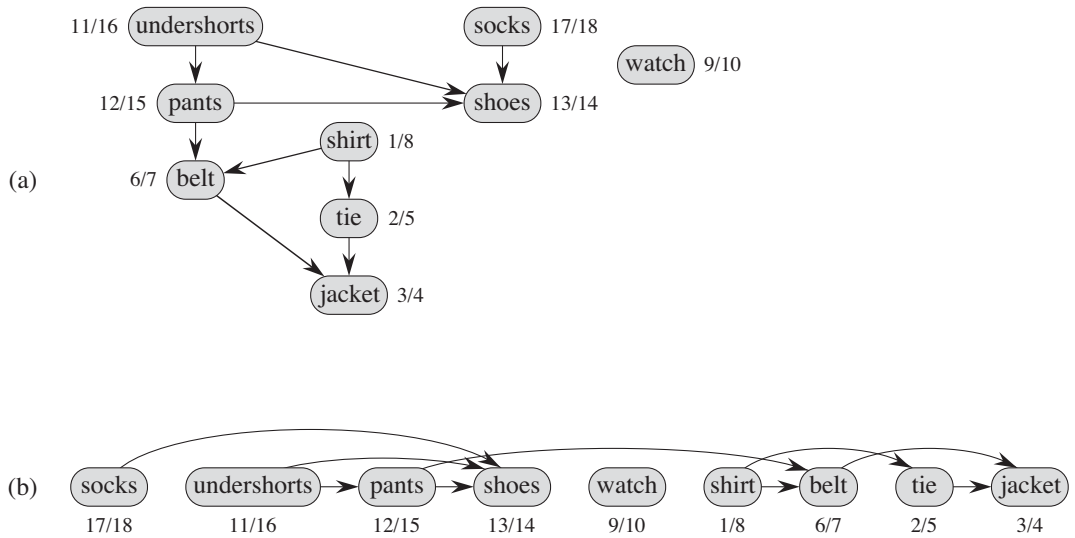


Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.