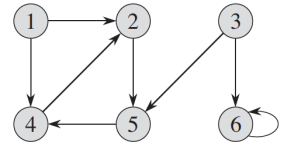


Graphs 1
Lecture 19

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Representations
2. Adjacency list and matrix
3. Breadth-first search
4. Depth-first search

Today’s outline

1. Representations
2. Adjacency list and matrix
3. Breadth-first search
4. Depth-first search

Definition

A graph, *G*, consists of a set of vertices, *V*, and a set of edges, *E*. This is often denoted as $\mathrm{G} = (\mathrm{V}, \mathrm{E})$.

Vertices are specified by unique labels, and an edge is an ordered pair of vertex labels.

For example:

$$\mathrm{G} = (\mathrm{V}, \mathrm{E})$$
$$\mathrm{V} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$$
$$\mathrm{E} = \{(\mathrm{b}, \mathrm{a}), (\mathrm{b}, \mathrm{c}), (\mathrm{a}, \mathrm{d}), (\mathrm{d}, \mathrm{c})\}$$

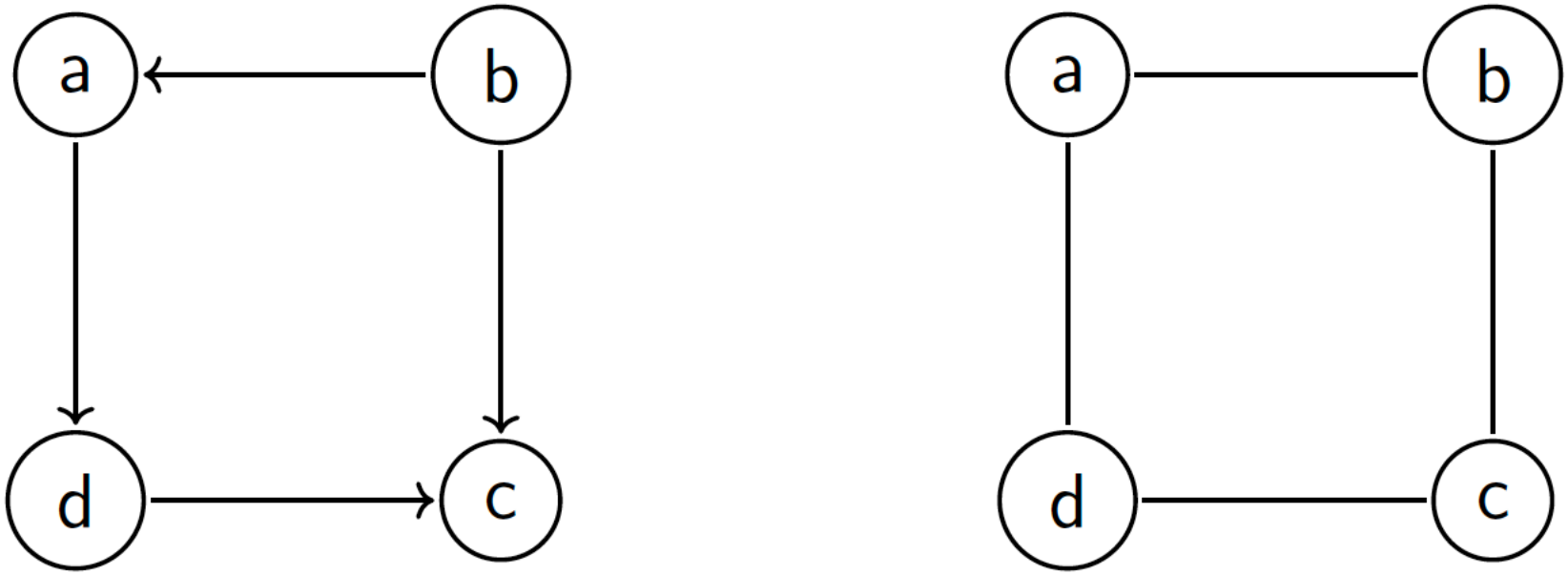
Definition

A graph may be directed or undirected. The definition given previously suffices for both, or we can explicitly define all edges in an undirected graph:

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

Visual representation

Graphs are often represented visually. For example, the above graph, with directed edges on the left, and undirected on the right, looks like:



Applications – Just a few

- the web, internet and all computer networks are graphs
- network routing
- web search (e.g. via pagerank)
- a map can be represented as a graph
- shortest path algorithms are graph algorithms
- automatically timetabling classes to minimise clashes
- social networks are graphs
- Neural pathways in the brain can be modelled as a graph.
- molecules can be modelled as graphs and used to simulate physical processes
- biological systems can be modelled as graphs (predator, prey, etc)
- projects can be modelled as graphs. Graph algorithms are used to determine
- critical paths or critical actions in a project

Today’s outline

1. Representations
2. Adjacency list and matrix
3. Breadth-first search
4. Depth-first search

Adjacency matrix

Let $G = (V, E)$ with $V = \{a_1, a_2, \ldots, a_n\}$. We assume that vertices are numbered 1, 2, …, n. Where $n = |V|$ is the number of vertices in the graph.

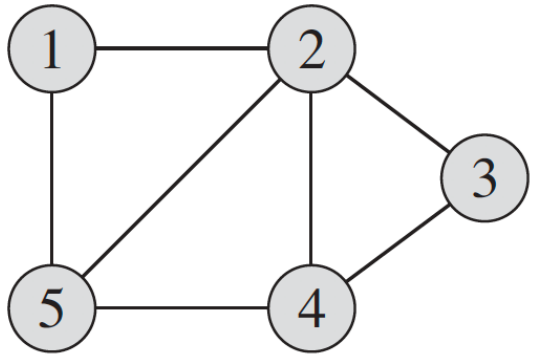
An adjacency matrix for G is an $|V| \times |V|$ Boolean array $A = (a_{ij})$, such that $a[i, j] = 1$ if there is a (directed) edge from i to j, else a[i, j] = 0.

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

For each $u \in V$ and $v \in V$, the matrix will contain 1 at row u and column v, if the edge $(u, v) \in E$, otherwise 0.

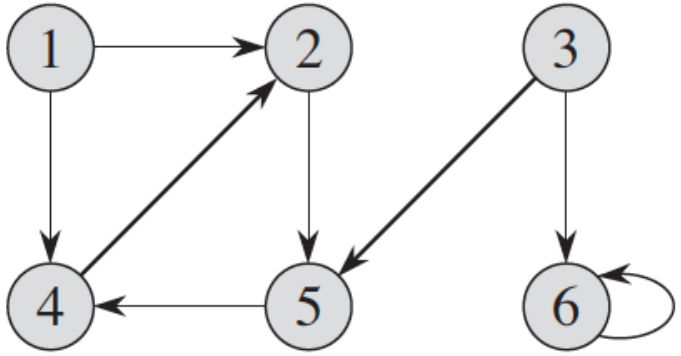
Examples: Adjacency matrix

Undirected



	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

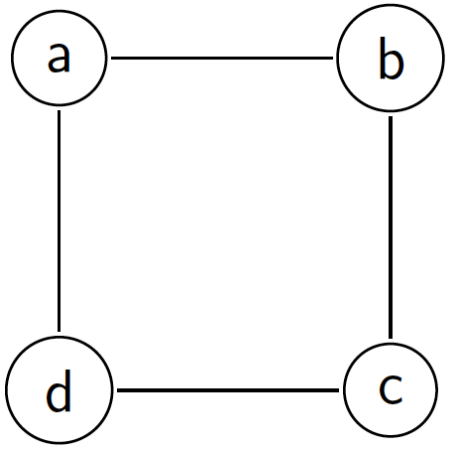
Directed

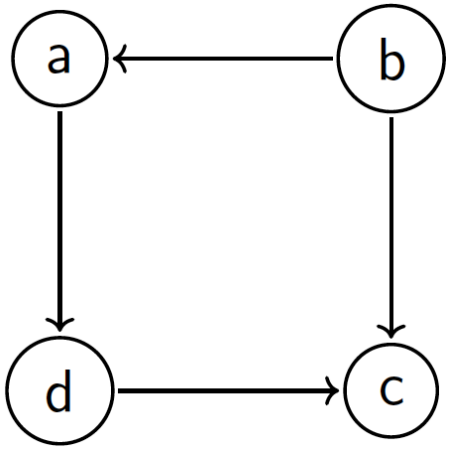


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Class challenge

Construct adjacency matrices

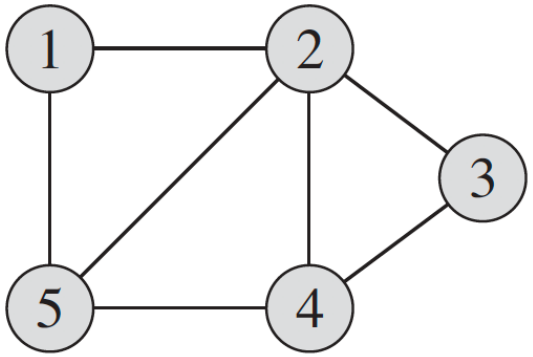




Observations

The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

In an undirected graph, (u,v) and (v,u) represent the same edge, and so the adjacency matrix *A* is its own transpose: $A = A^T$. That is, reflecting elements along its main diagonal.



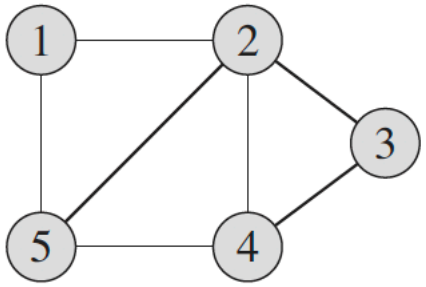
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

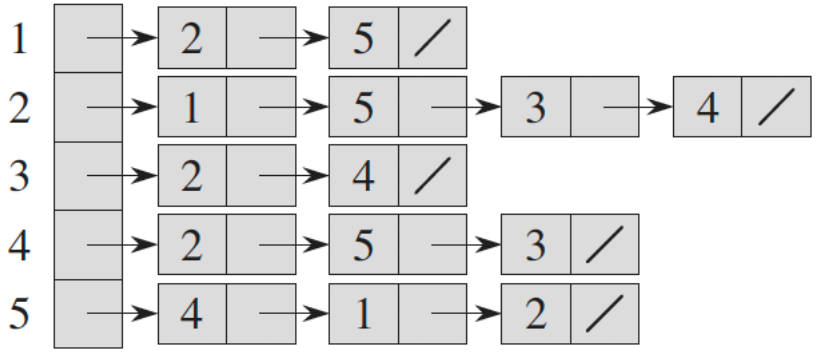
Adjacency lists

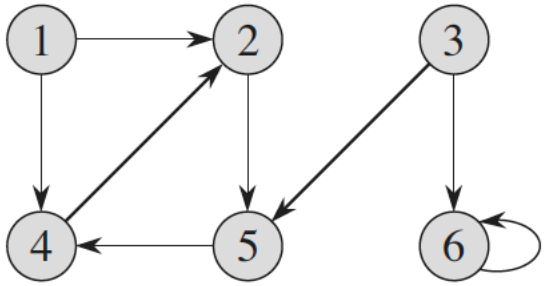
Let $G = (V, E)$ with $V = \{a_1, a_2, \ldots, a_n\}$.

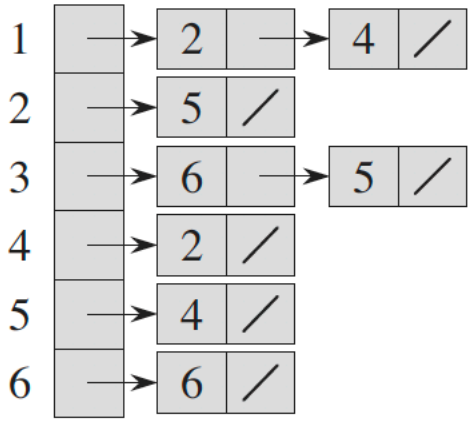
An adjacency list consists of an array *Adj* of $|V|$ lists, one for each vertex in V. For each $u \in V$, $Adj[u]$ contains all the vertices v, such that there is an edge $(u, v) \in E$.

Adjacency lists



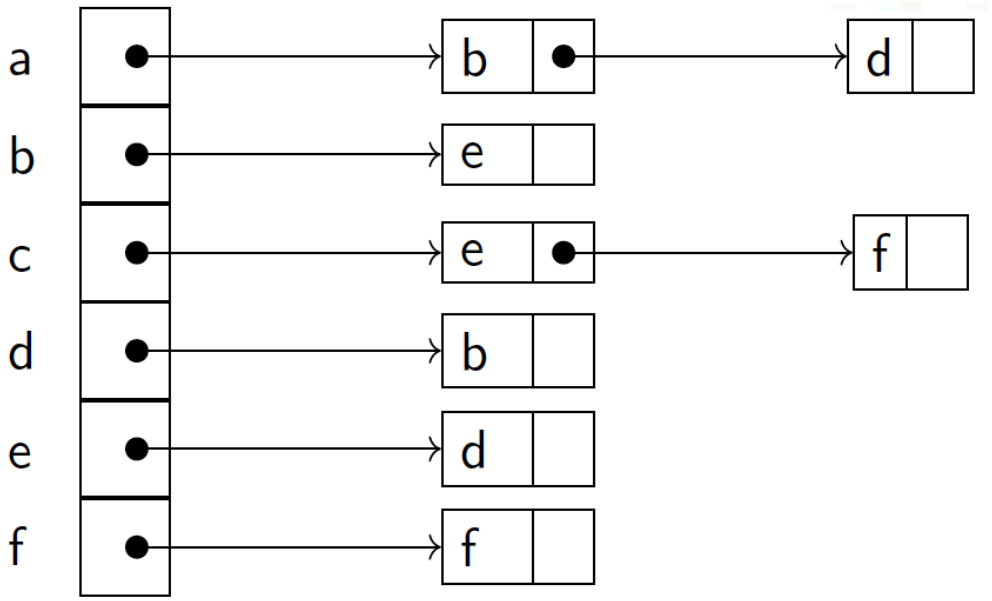


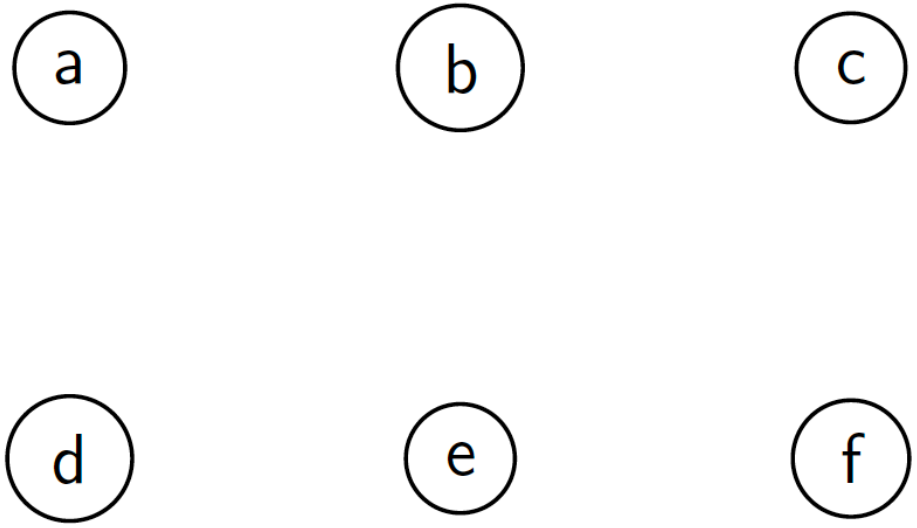




Class challenge

Visualise the graph that corresponds to the following adjacency list:





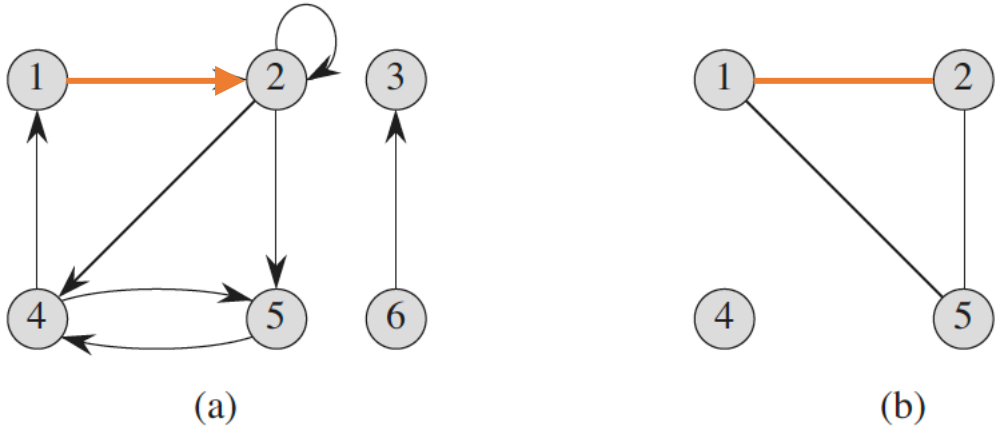
Adjacency

If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is **adjacent** to vertex u.

When the graph is undirected, the adjacency relation is symmetric.

When the graph is directed, the adjacency relation is not necessarily symmetric.

In (a) and (b) of the Figure, vertex 2 is adjacent to vertex 1, since the edge (1, 2) belongs to both graphs. Vertex 1 is not adjacent to vertex 2 Figure (a), since the edge (2, 1) does not belong to the graph. That is, we cannot reach 1 from 2.



Comparing implementations

Suppose *G* is a graph with *n* vertices in *V*. Which is better, an adjacency matrix or adjacency list?

The space requirements of the adjacency matrix are high - if the graph is dense this space is utilised, if the graph is sparse an adjacency list would be more economical.

Dense means the number of edges in E is close to n^2, sparse means *n* or fewer edges.

Comparing implementations

It also depends on the operations we want to use. Apart from insertion and deletion, the three most common operations on graphs are:

1. Given two vertices *i* and *j*, determine whether there is an edge connecting them.
2. Given vertex *i*, find all vertices adjacent to *i*.
3. Given a vertex *i* as starting point, traverse the graph.

Discussion

The first operation is supported best by the adjacency matrix, the second and third by adjacency lists. Why?

Today’s outline

1. Representations
2. Adjacency list and matrix
3. Breadth-first search
4. Depth-first search

BFS Overview

Given a graph *G = (V, E)* and a **source** vertex *s*, BFS explores the edges of *G* to discover every vertex that is reachable from *s*. It computes the distance (smallest number of edges) from *s* to each reachable vertex.

The algorithm works on both directed and undirected graphs.

BFS explores all of the neighbour nodes at the present depth prior to moving on to nodes at the next depth level.

That is, it discovers all vertices at distance *k* from *s*, before discovering vertices at distance *k* + 1.

BFS Overview

Every vertex starts off coloured white. A vertex is then coloured grey when it is “discovered” for the first time (adjacent to the current node).

The node is finally coloured black when its adjacency list has been fully examined (i.e. all the vertices adjacent to it have been coloured grey).

If $(u, v) \in E$ and vertex u is black, then vertex v is either grey or black. That is, all vertices adjacent to black vertices have been “discovered”.

Grey vertices may have some adjacent white vertices, as these represent the frontier between discovered and undiscovered vertices.

BFS Distances

For every vertex *u*, we compute the distance from the source vertex *s*, Initially, the distance from source *s* to *u* is set to ∞. We store this in a variable *u.d*.

We could simplify the algorithm to do without colours (instead of testing whether *u.c* = white we can test whether u.d = 1).

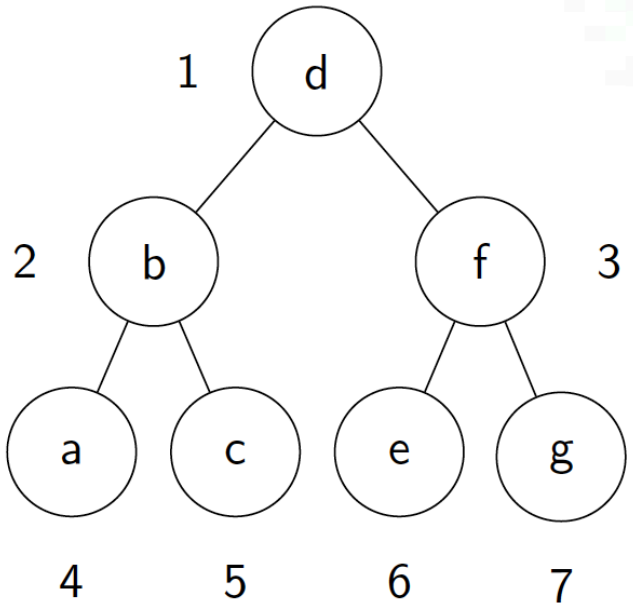
The predecessor of *u* is stored in *u.π.* The algorithm uses a first-in, first-out queue (FIFO).

The results of BFS depends upon the order in which the neighbours of a given vertex are visited in L12, but the distances *d* computed will not.

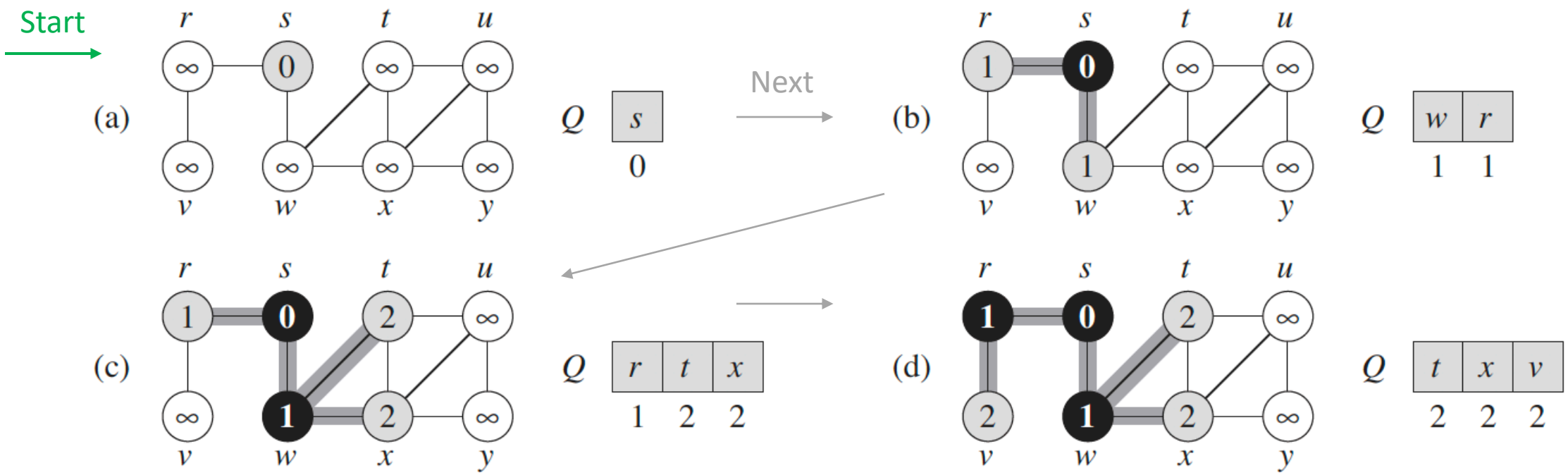
```
procedure BFS_Search(G, s)
1:   for each vertex u ∈ G.V − {s}   // Initialise all vertices except source
2:       u.colour = WHITE
3:       u.d = INF
4:       u.π = NIL
5:   s.colour = GREY                 // Initialise source
6:   s.d = 0
7:   s.π = NIL
8:   Q = Ø
9:   Enqueue(Q, s)                   // Add source to queue
10:  while Q = Ø                     // While we have vertices to explore
11:      u = Dequeue(Q)              // Get next vertex
12:      for each v ∈ G.Adj[u]       // For-each vertex that is adjacent to u
13:          if v.colour == WHITE
14:              v.colour = GREY
15:              v.d = u.d + 1
16:              v.π = u
17:              Enqueue(Q, v)       // Add to queue, to explore later
18       u.colour = BLACK            // Explored all adjacent vertices, make self black
end procedure
```

BFS and binary trees

When applied to a binary tree, BFS gives a new kind of traversal - a level-order traversal.

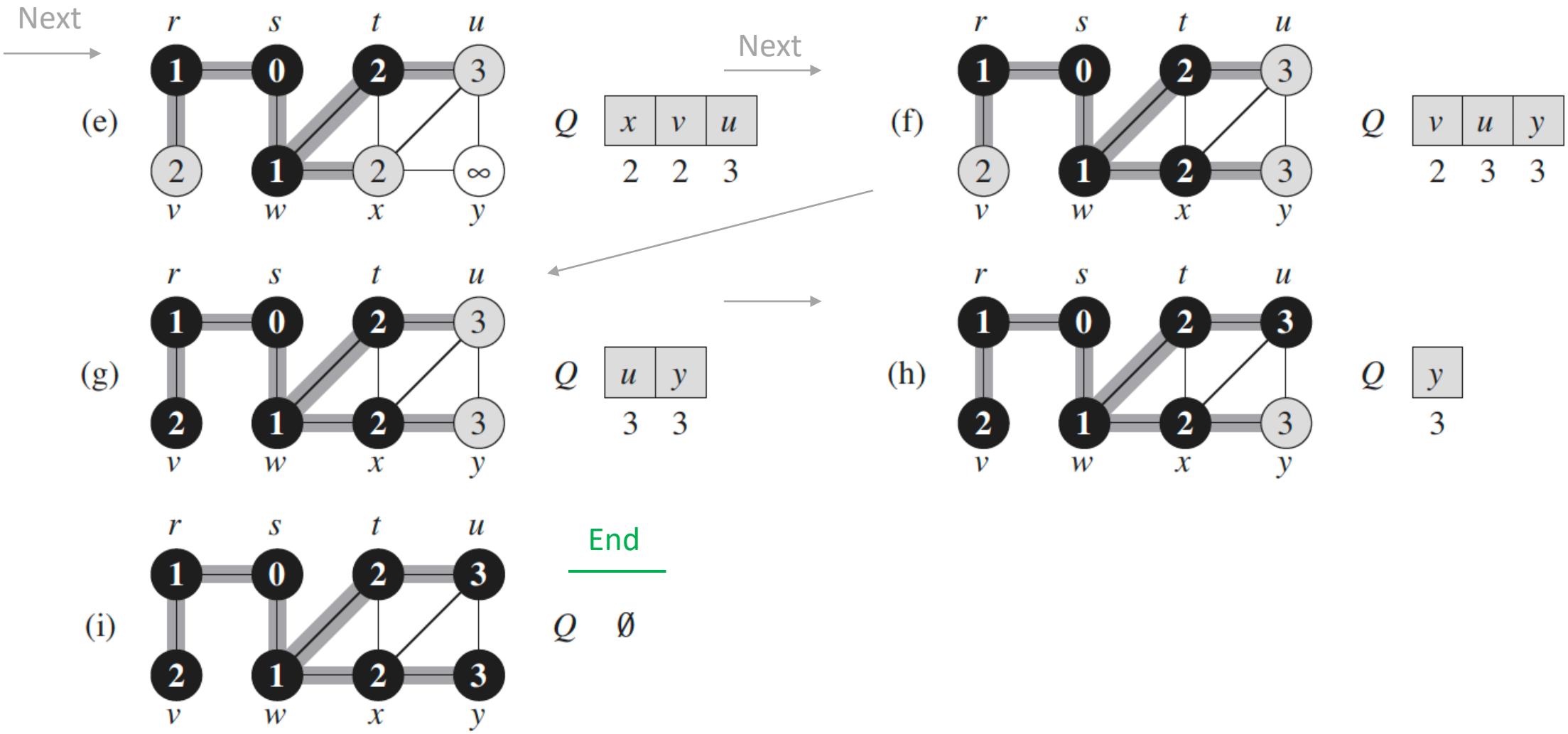


Example



Q = Queue

Example


Next
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
Next
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
End
Q Ø

Today’s outline

1. Representations
2. Adjacency list and matrix
3. Breadth-first search
4. Depth-first search

Depth-First Search

The idea of depth-first search (DFS) is that one goes deeper whenever possible. That is, we explore the edges from the most recently discovered vertex *v*.

Once all of *v*'s edges have been explored, the search "backtracks" to vertex *u*, (*v*'s parent from which *u* was discovered), and continue exploring edges leaving vertex *u*.

As in BFS, vertices start off white, are made grey when first discovered, and are made black when finished, i.e. their adjacency list has been examined completely.

DFS Overview

DFS also timestamps each vertex. Each vertex *v* has two timestamps: the first *v.d* records when *v* was first discovered (i.e. coloured grey), and the second timestamp *v.f* when the search finishes examining *v*'s adjacency list (i.e. blackens *v*).

Timestamps provide important information about tree structure. Timestamps are integers that range between 1 and 2|V|.

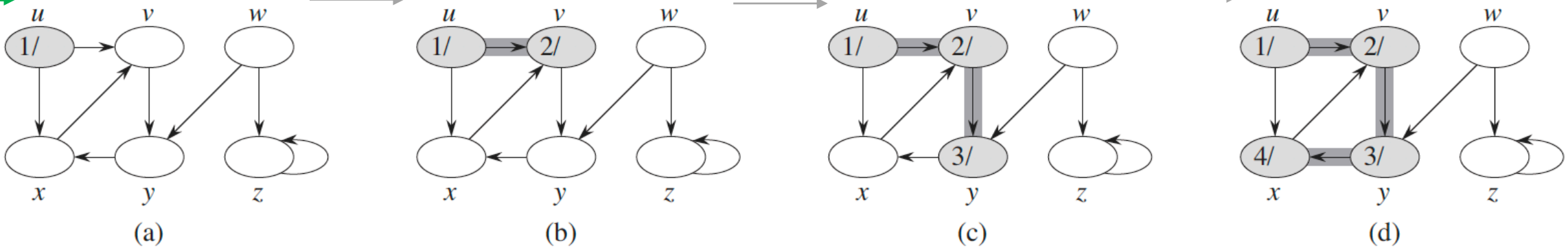
For each vertex, $u.d < u.f$

The variable time is a **global variable** for timestamping.

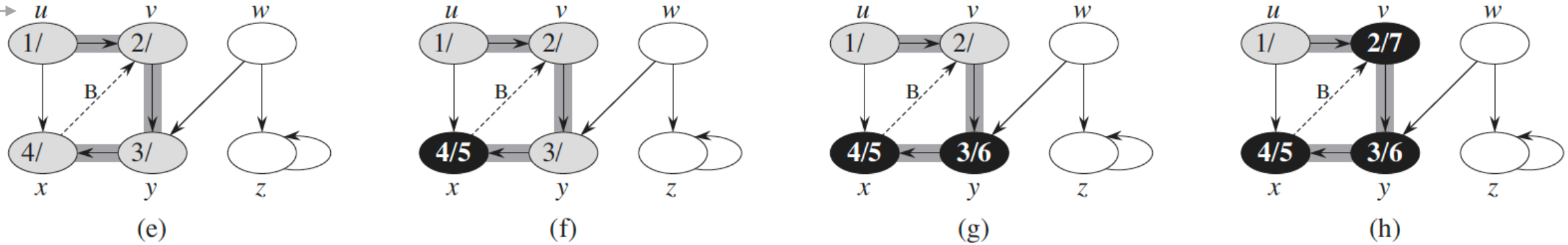

```
procedure DFS(G)
1:   for each vertex u ∈ G.V   // Initialise all vertices
2:       u.colour = WHITE
3:       u.π = NIL
4:   time = NIL                 // Initialise time (global variable)
5:   for each vertex u ∈ G.V
6:       if u.colour == WHITE
7:           DFS-Visit(G, u)   // Visit each white vertex in Graph

procedure DFS-Visit(Q, u)
1:   time = time + 1                  // Increment global time value
2:   u.d = time                       // Vertex discovered time
3:   u.colour = GREY
4:   for each v ∈ G.Adj[u]            // For each vertex v adjacent to vertex u
5:       if v.colour == WHITE         // if white, then it's unexplored and we need to visit
6:               v.π = u
7:               DFS-Visit(Q, v)
8:   u.colour = BLACK                 // Explored all adjacent vertices, make self black
9:   time = time + 1                  // Increment time (global)
10:  u.f = time                       // Set finish time for self
```

DFS Example

Start



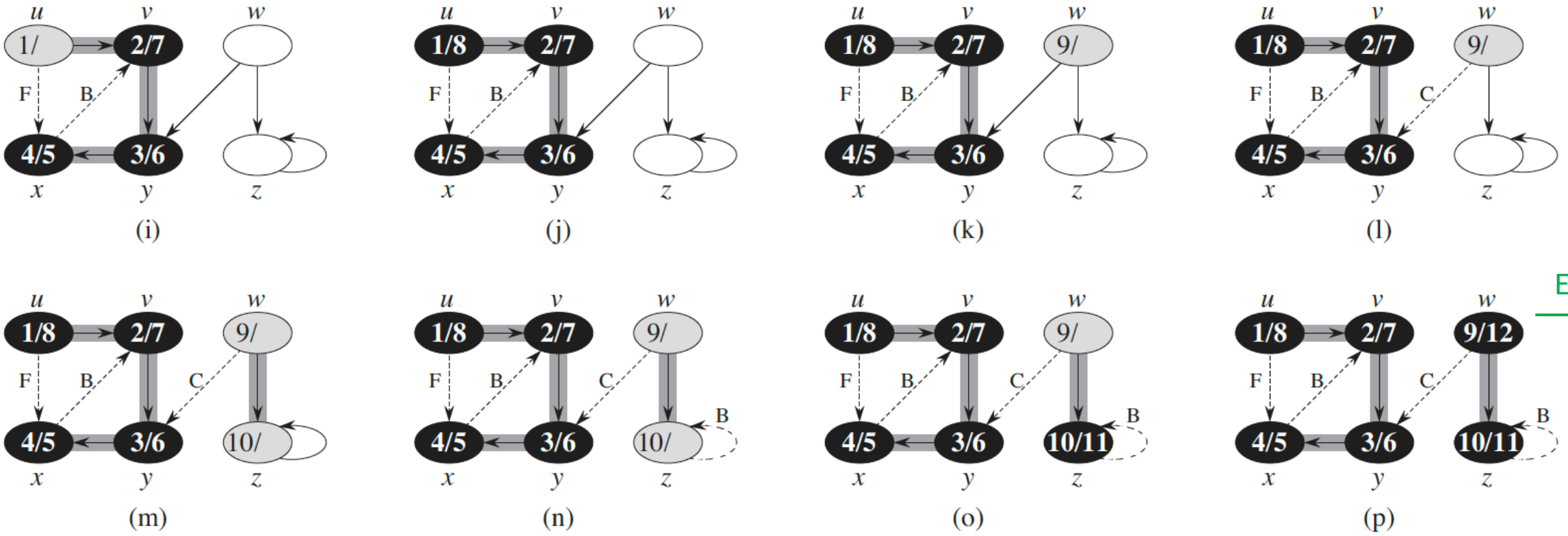
Next



(e) We rediscover vertex v, but it's not white, so this is a back edge.
(f) We've now explored all vertices in x, colour black, and recurse up to (g)

DFS Example

Next



End

(j) Explored all vertices in u, colour black.
(k) Explore next white vertex in Graph (Liens 5-7 of DFS)

(l) Rediscover node y, but it's not white, so cross edge.
(p) All vertices fully explored, colour last vertex black.

Suggested reading

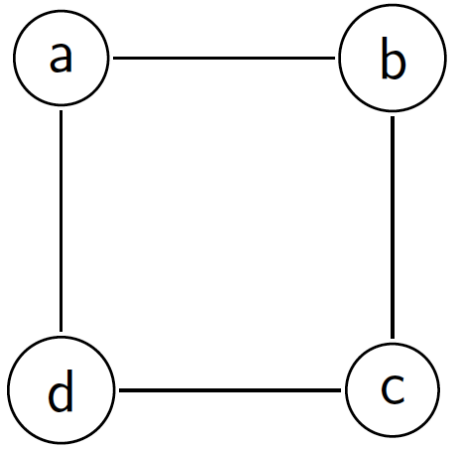
Today's material on elementary graph algorithms is discussed in Section 22 of the textbook.

A primer on graphs and graph terminology is covered in Appendix B.4 of the textbook.

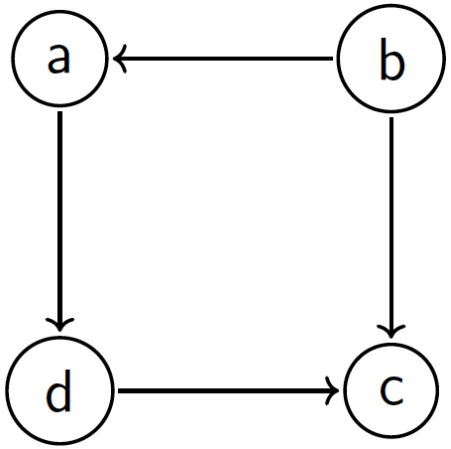
Solutions

Class challenge

Construct adjacency matrices



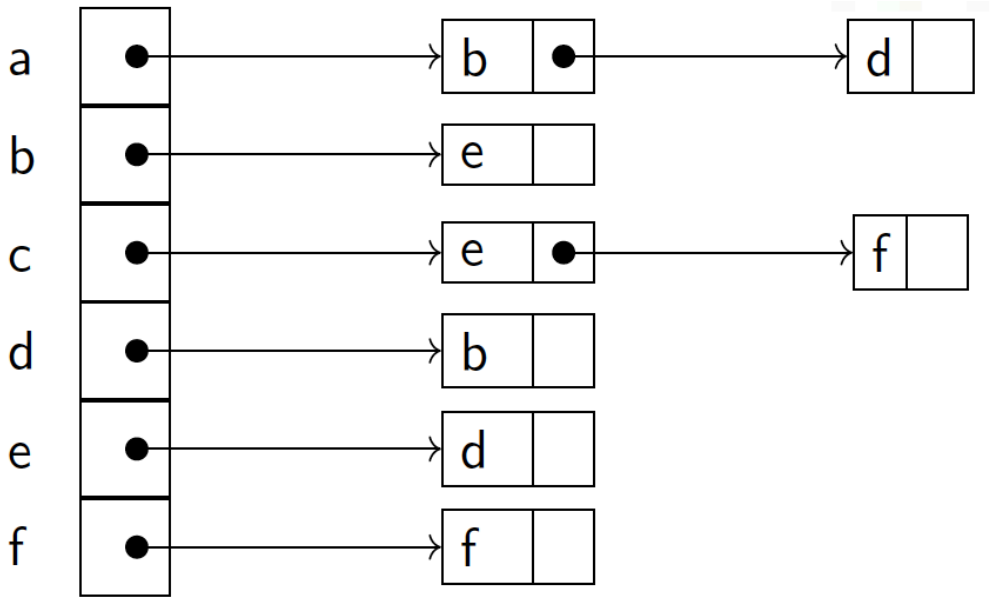
	a	b	c	d
a	0	1	0	1
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0

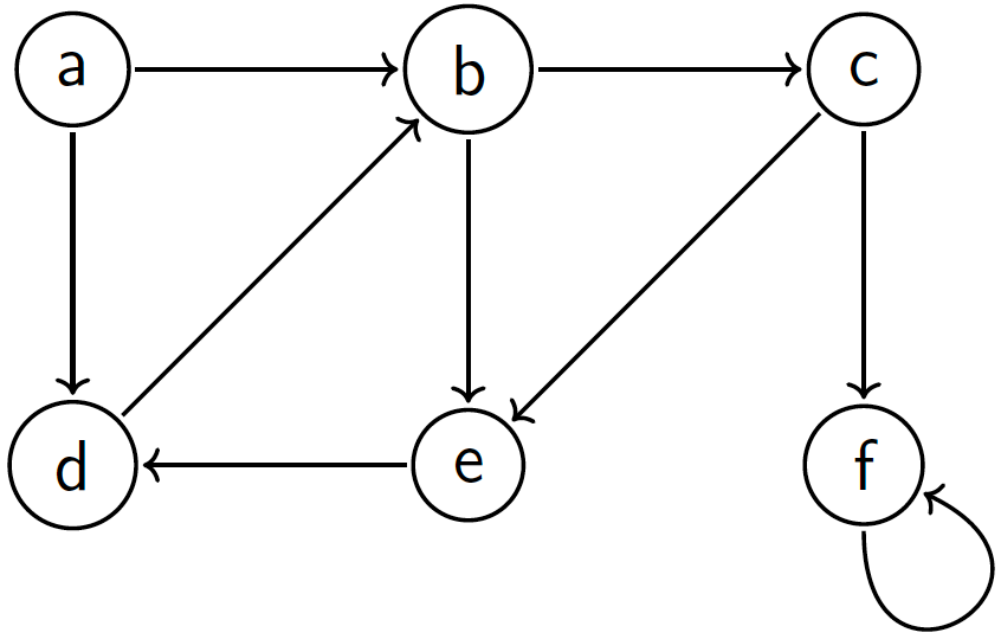


	a	b	c	d
a	0	0	0	1
b	1	0	1	0
c	0	0	0	0
d	1	0	1	0

Class challenge

Visualise the graph that corresponds to the following adjacency list:





Comparing implementations

1. Given two vertices *i* and *j*, determine whether there is an edge connecting them.

   In an undirected graph, a matrix allows for a single lookup operation. In contrast, a list would require traversal of up to d elements for vertex i. This list is length d (degree of vertex).

2. Given vertex *i*, find all vertices adjacent to *i*.

   With a list, we need only look at and traverse the list of length d at vertex i. But for a matrix, we need to look at all |V| in row i. However, with a dense graph, matrix and list are equivalent.

Comparing implementations

3. Given a vertex *i* as starting point, traverse the graph.

   With a list, we only traverse traverse the number of elements = |E|, the number of edges in graph. For a matrix, we need to traverse V^2 elements (wasted steps due to 0 edge coding). However, with a dense graph, matrix and list are equivalent.

Image attributions

This Photo by Unknown Author is licensed under CC0

Disclaimer: Images and attribution text provided by PowerPoint search. The author has no connection with, nor endorses, the attributed parties and/or websites listed above.