CSC 225 - Summer 2019

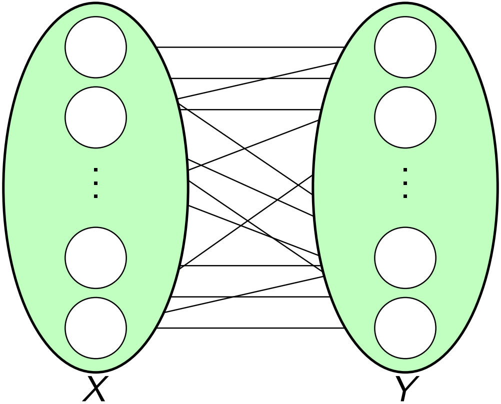
Graphs II

Bill Bird

Department of Computer Science
University of Victoria

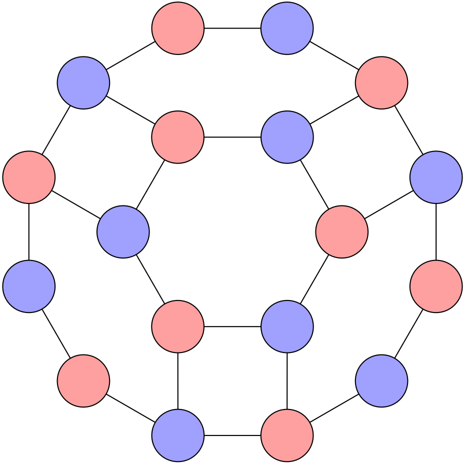
July 8, 2019

Bipartite Graphs (1)



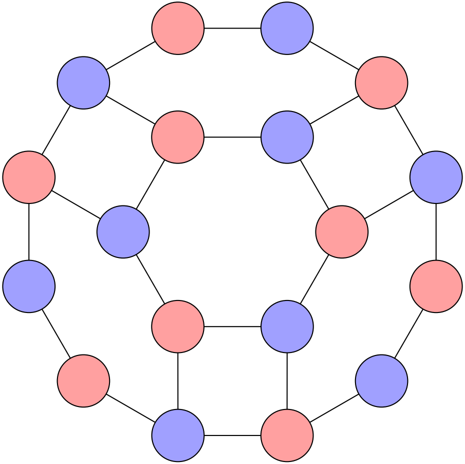
- A graph is **bipartite** if the vertices of G can be partitioned into sets X and Y such that no edge connects two vertices in X or two vertices in Y.
- The pair (X, Y) is called a **bipartition**.

Bipartite Graphs (2)



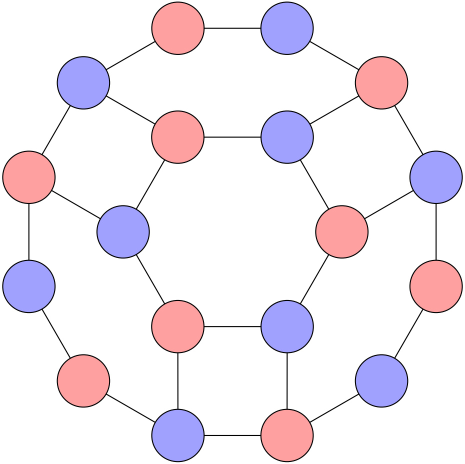
- The graph above is bipartite, since no pair of blue vertices is adjacent and no pair of red vertices is adjacent.
- There may be more than one possible bipartition.

Bipartite Graphs (3)



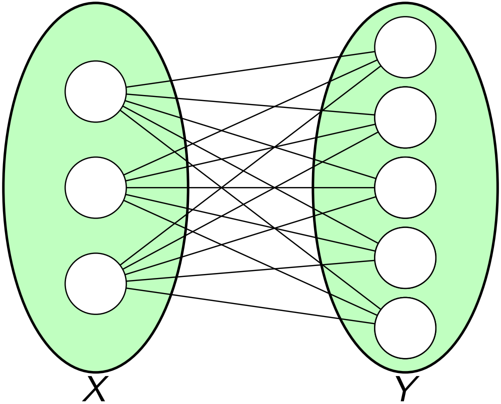
- A bipartite graph is also called **2-colourable**, since its vertices can be coloured with two colours such that no edge connects two vertices of the same colour.

Bipartite Graphs (4)



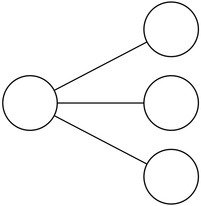
- Similarly, a bipartition can be called a '2-colouring'.

Bipartite Graphs (5)

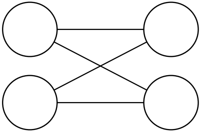


- A **complete bipartite graph**, denoted $K_{m,n}$, consists of a set X of m vertices and set Y of n vertices, with all possible edges between the two sets.
- The graph above is $K_{3,5}$.

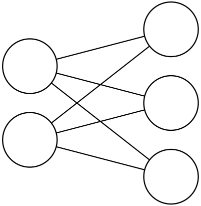
Bipartite Graphs (6)



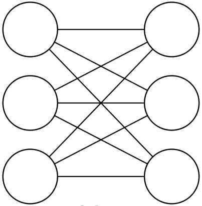
$K_{1,3}$



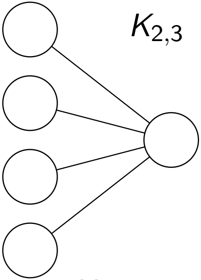
$K_{2,2}$



$K_{2,3}$



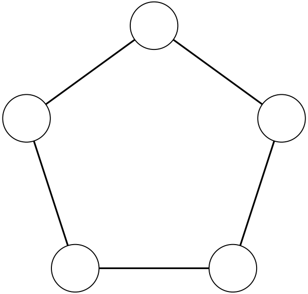
$K_{3,3}$



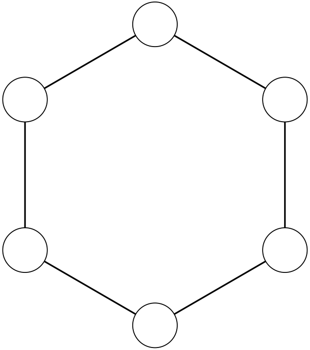
$K_{4,1}$

- The number of edges in $K_{m,n}$ is mn.

Cycles (1)



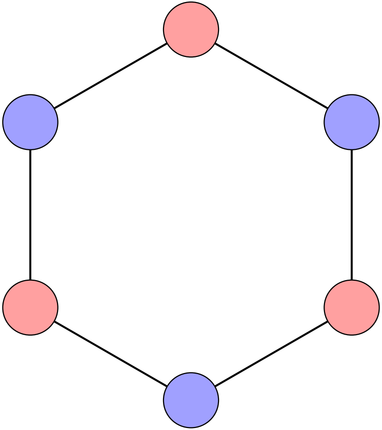
C_5



C_6

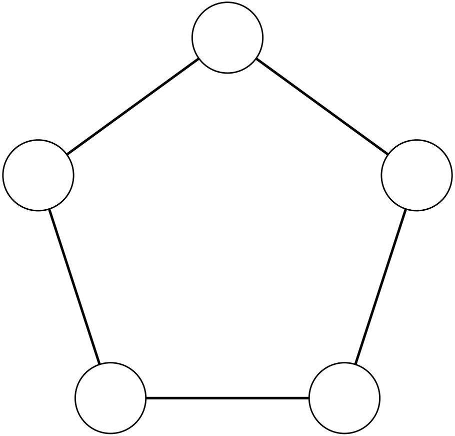
- A **cycle graph**, denoted C_n, consists of a cycle with n vertices (and n edges).

Cycles (2)



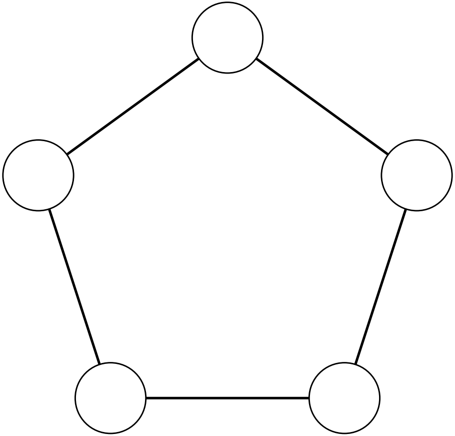
- The cycle C_6 is bipartite.
- It is easy to demonstrate that a graph is bipartite by finding a bipartition.

Cycles (3)



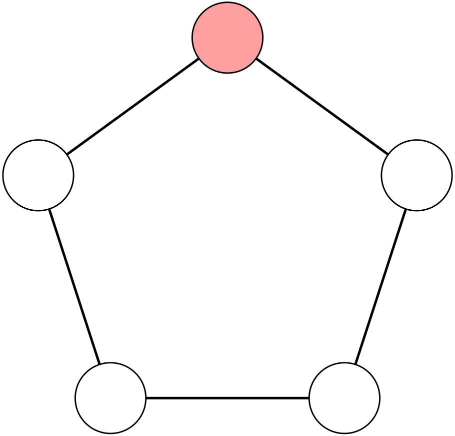
- **Exercise**: Prove that C_5 is not bipartite.
- This requires showing that it is impossible for a bipartition to exist.

Cycles (4)



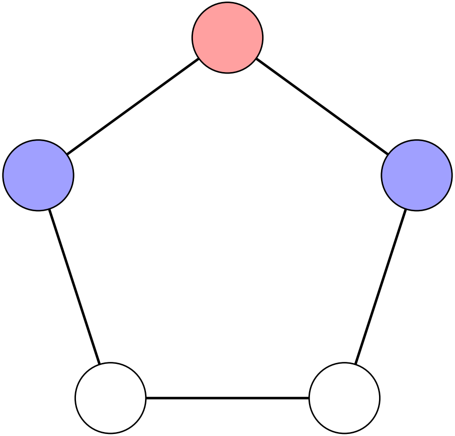
- To prove that C_5 is not bipartite, we will show that any possible 2-colouring leads to a contradiction.

Cycles (5)



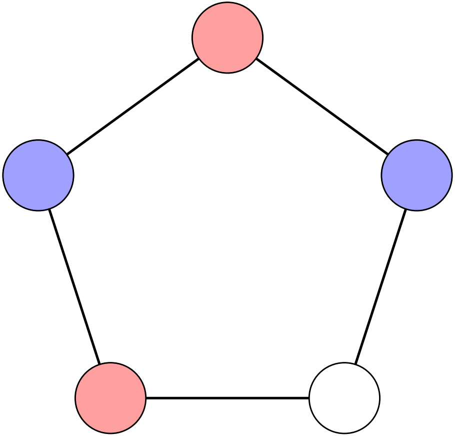
- First, we can assume that at least one vertex must be red. Without loss of generality, assume that it is the vertex shown.
- This is a reasonable assumption because all of the vertices are equivalent with respect to the structure of the graph.

Cycles (6)



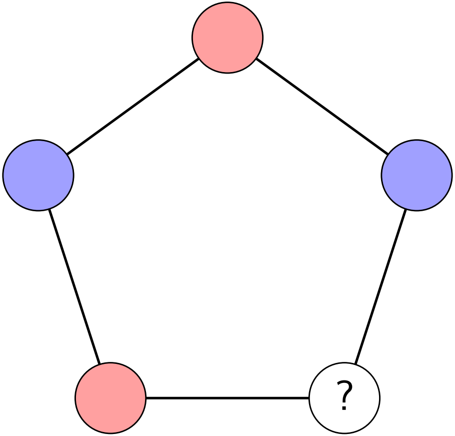
- Since the top vertex is red, both of its neighbours must be blue in a 2-colouring.

Cycles (7)



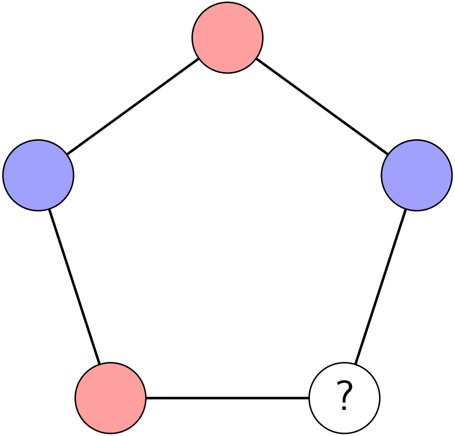
- The neighbour of the left blue vertex must be red.

Cycles (8)



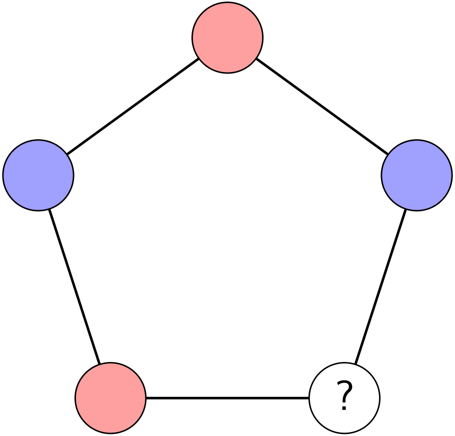
- The remaining vertex cannot be assigned a colour without breaking the rules, since it is adjacent to both a red and a blue vertex.
- Therefore, no 2-colouring exists.

Cycles (9)



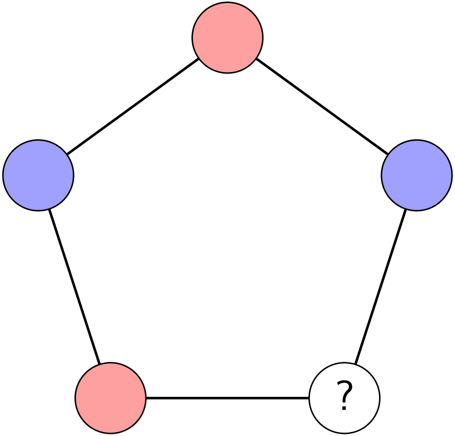
- The proof made exactly one assumption: There must be at least one red vertex.

Cycles (10)



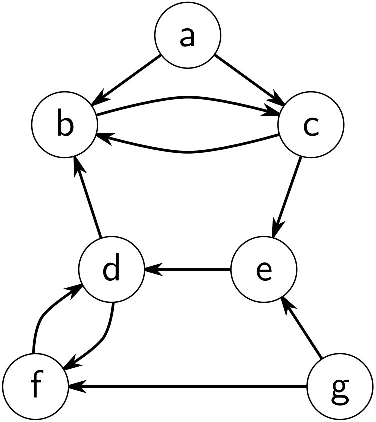
- All of the other steps in the proof were the result of mathematical requirements based on the definition of a 2-colouring.

Cycles (11)



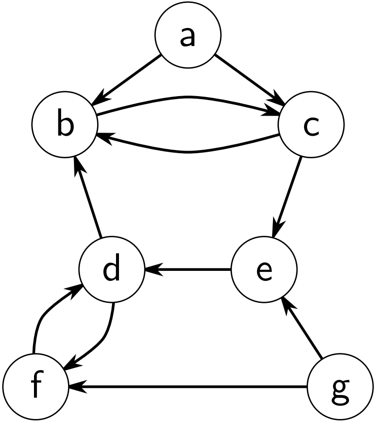
- **Warning**: The proof is only correct because the assumption is justified and sound. Without justifying the assumption, the 'proof' would just be an example, which proves nothing.

Directed Graphs (1)



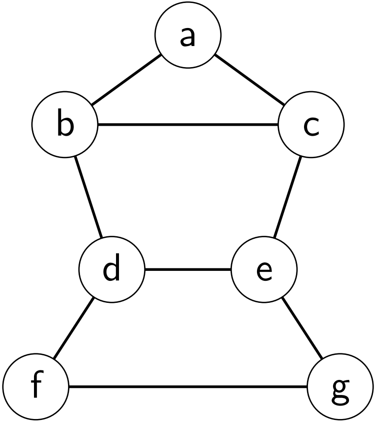
- A **directed graph** (or digraph) is a graph in which edges have a direction.
- Formally, edges in a directed graph correspond to ordered pairs of vertices.

Directed Graphs (2)



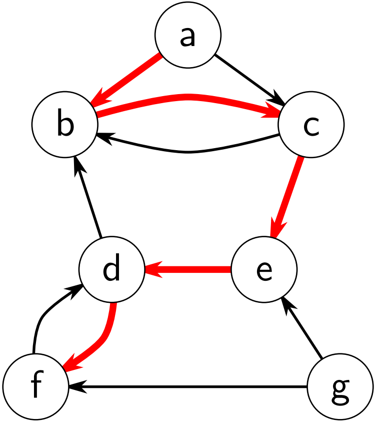
- Since edges are ordered pairs, the edges (b, c) and (c, b) are distinct.
- Edges in directed graphs are often called **directed edges** or **arcs**.

Directed Graphs (3)



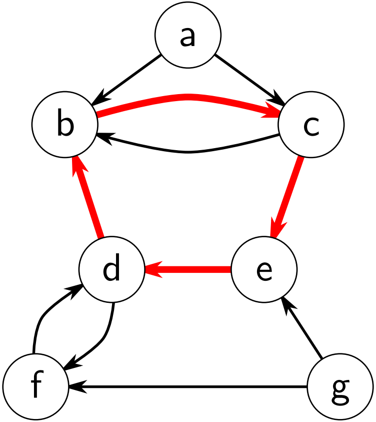
- Every directed graph can be collapsed into an undirected graph by removing the directions from edges (and combining bi-directional edge pairs into single undirected edges).

Directed Graphs (4)



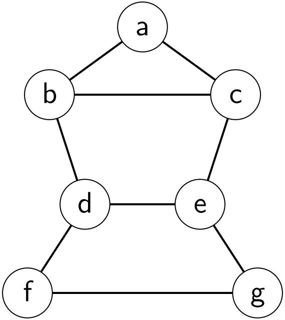
- A **directed path** is a path in a directed graph which respects the direction of edges.
- It is natural to refer to the endpoints of a directed path as the 'source' and the 'destination'.

Directed Graphs (5)



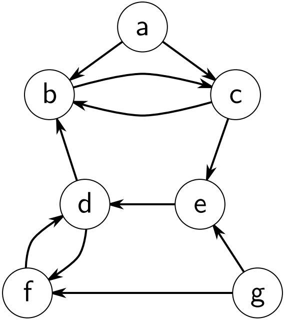
- A **directed cycle** is a directed path whose source and destination are the same.

Graph Representation (1)



$$V = \{a, b, c, d, e, f, g\}$$

$$E = \{\{a,b\},\ \{a,c\},\ \{b,c\},\ \{b,d\},\ \{c,e\},\ \{d,e\},\ \{d,f\},\ \{e,g\},\ \{f,g\}\}$$

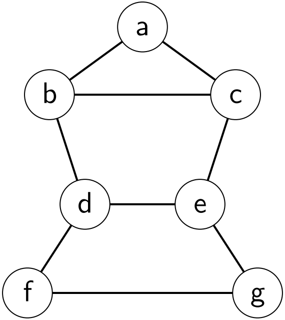


$$V = \{a, b, c, d, e, f, g\}$$

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

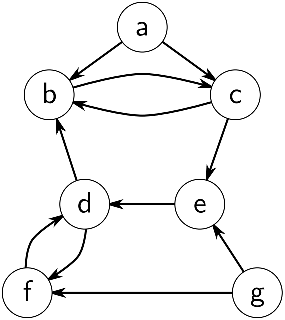
- The most obvious way to represent a graph is to explicitly store the sets V and E.

Graph Representation (2)



$$V = \{a, b, c, d, e, f, g\}$$

$$E = \{\{a,b\}, \quad \{a,c\}, \quad \{b,c\}, \quad \{b,d\}, \quad \{c,e\}, \quad \{d,e\}, \quad \{d,f\}, \quad \{e,g\}, \quad \{f,g\}\}$$

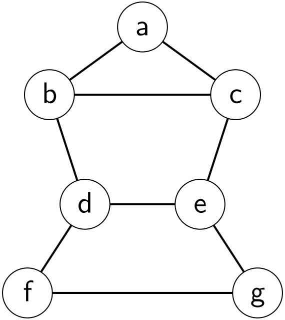


$$V = \{a, b, c, d, e, f, g\}$$

$$E = \{(a,b), \quad (a,c), \quad (b,c), \quad (c,b), \quad (c,e), \quad (d,b), \quad (d,f), \quad (e,d), \quad (f,d), \quad (g,e), \quad (g,f)\}$$

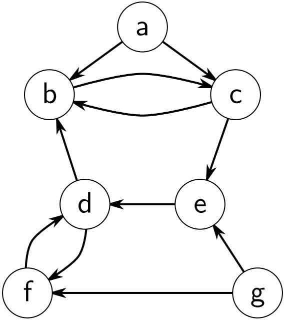
- For lack of a better term, this representation is called an **edge list**.

Graph Representation (3)



$$V = \{a, b, c, d, e, f, g\}$$

$$\begin{aligned} E = \{&\{a, b\}, \quad \{a, c\}, \quad \{b, c\}, \\ &\{b, d\}, \quad \{c, e\}, \quad \{d, e\}, \\ &\{d, f\}, \quad \{e, g\}, \quad \{f, g\}\} \end{aligned}$$



$$V = \{a, b, c, d, e, f, g\}$$

$$\begin{aligned} E = \{&(a, b), \quad (a, c), \quad (b, c), \\ &(c, b), \quad (c, e), \quad (d, b), \\ &(d, f), \quad (e, d), \quad (f, d), \\ &(g, e), \quad (g, f)\} \end{aligned}$$

- In an implementation, each edge could be represented by an object containing pointers to its endpoint vertices.

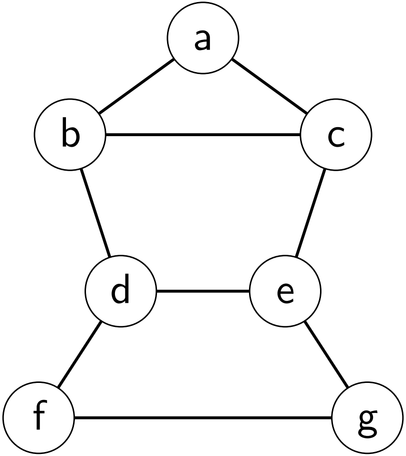
Graph Representation (4)

	Edge List	Adj. List (unsorted)	Adj. List (sorted)	Adjacency Matrix
Count edges in G	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$	$\Theta(n^2)$
Find all neighbours of v	$\Theta(m)$	$\Theta(\deg(v))$	$\Theta(\deg(v))$	$\Theta(n)$
Test if edge vu exists	$\Theta(m)$	$\Theta(\deg(v))$	$\Theta(\log(\deg(v)))$	$\Theta(1)$
Add edge vu	$\Theta(1)$	$\Theta(1)$	$\Theta(\deg(v))$	$\Theta(1)$
Delete edge vu	$\Theta(1)$[1]	$\Theta(\deg(v))$	$\Theta(\log(\deg(v)))$	$\Theta(1)$

Space Complexity	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n^2)$

[1] If it is necessary to search for edge *vu* first, deletion takes $\Theta(m)$ time.

Adjacency Lists (1)

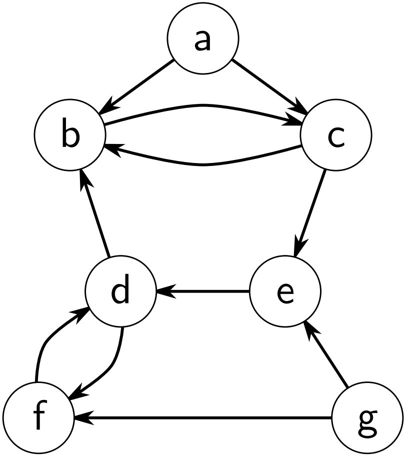


Adjacency List:

Vertex	Neighbours
a	b, c
b	a, c, d
c	a, b, e
d	b, e, f
e	c, d, g
f	d, g
g	e, f

- An **adjacency list** structure represents a graph with lists of each vertex's neighbours.

Adjacency Lists (2)

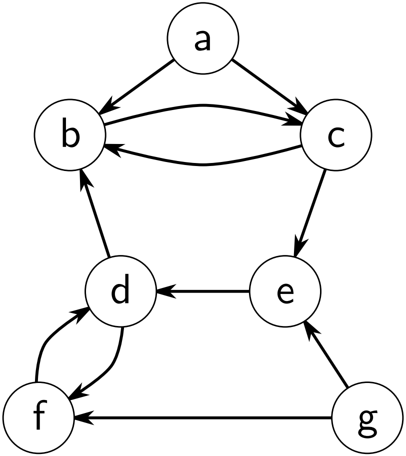


Adjacency List:

Vertex	Neighbours
a	b, c
b	c
c	b, e
d	b, f
e	d
f	d
g	e, f

- In a directed graph, the neighbour list normally only stores ‘outbound’ neighbours.
- A second set of lists can be created to store inbound neighbours if necessary.

Adjacency Lists (3)

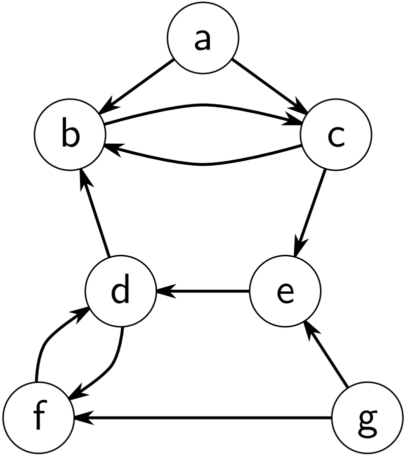


Adjacency List:

Vertex	Neighbours
a	b, c
b	c
c	b, e
d	b, f
e	d
f	d
g	e, f

- The total size of all neighbour lists together is $2m$, since each edge is represented twice (once for each endpoint).

Adjacency Lists (4)

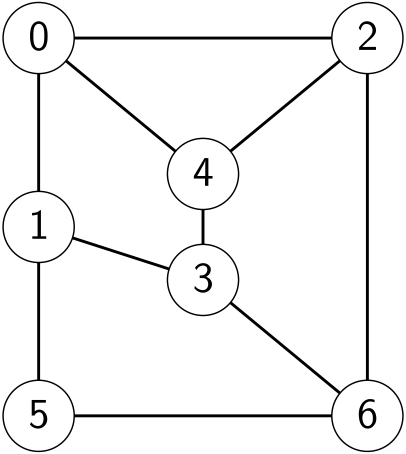


Adjacency List:

Vertex	Neighbours
a	b, c
b	c
c	b, e
d	b, f
e	d
f	d
g	e, f

- One advantage of adjacency lists is their compactness: Graphs with few edges will require less space than more dense graphs.

Adjacency Lists (5)

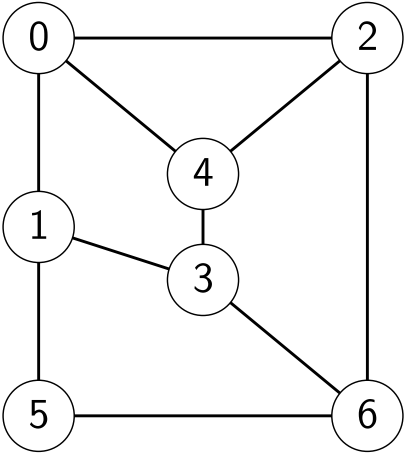


Adjacency List:

Vertex	Neighbours
0	1, 2, 4
1	0, 3, 5
2	0, 4, 6
3	1, 4, 6
4	0, 2, 3
5	1, 6
6	2, 3, 5

- In most cases, it is helpful to number vertices starting at 0 and use their indices to construct an adjacency list.

Adjacency Lists (6)

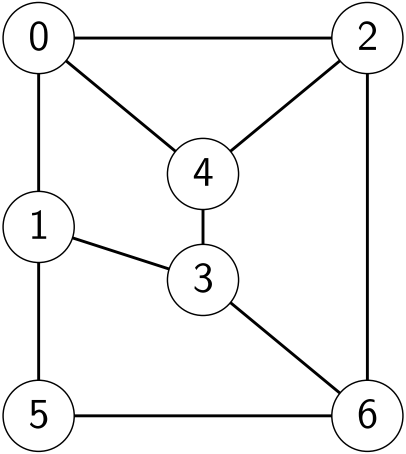


Adjacency List:

Vertex	Neighbours
0	1, 2, 4
1	0, 3, 5
2	0, 4, 6
3	1, 4, 6
4	0, 2, 3
5	1, 6
6	2, 3, 5

- Assigning an index to each vertex allows the data structure to be stored in an array of *n* lists.

Adjacency Lists (7)

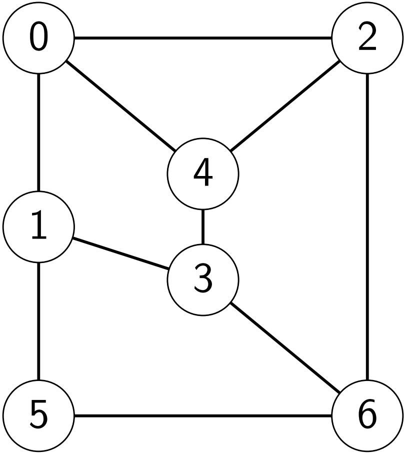


Adjacency List:

Vertex	Neighbours
0	1, 2, 4
1	0, 3, 5
2	0, 4, 6
3	1, 4, 6
4	0, 2, 3
5	1, 6
6	2, 3, 5

- Alternatively, the data structure can be represented by a set of vertex objects, each with a list of pointers to its neighbours.

Adjacency Lists (8)

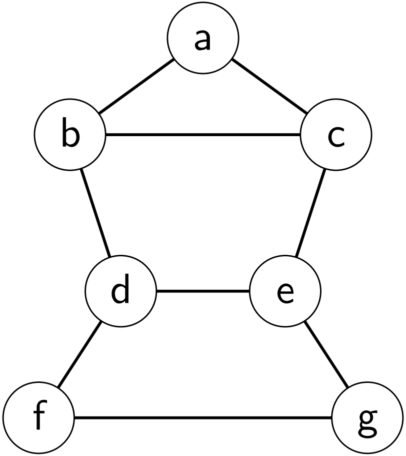


Adjacency List:

Vertex	Neighbours
0	1, 2, 4
1	0, 3, 5
2	0, 4, 6
3	1, 4, 6
4	0, 2, 3
5	1, 6
6	2, 3, 5

- The linked representation is preferable when vertices might be added to or removed from the graph after construction.

Adjacency Matrices (1)

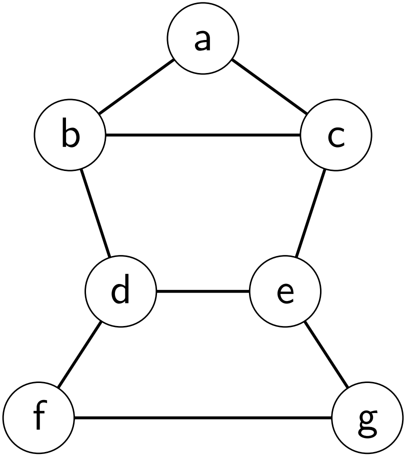


Adjacency Matrix:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	0	1	0	0
d	0	1	0	0	1	1	0
e	0	0	1	1	0	0	1
f	0	0	0	1	0	0	1
g	0	0	0	0	1	1	0

- An **adjacency matrix** structure represents a graph on n vertices with an $n \times n$ matrix of 0/1 values.

Adjacency Matrices (2)

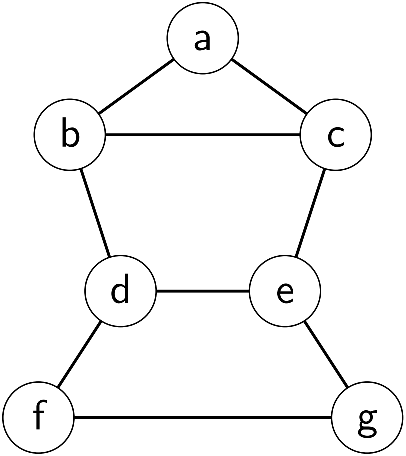


Adjacency Matrix:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	0	1	0	0
d	0	1	0	0	1	1	0
e	0	0	1	1	0	0	1
f	0	0	0	1	0	0	1
g	0	0	0	0	1	1	0

- Each vertex receives both a row and column of the matrix.
- If the entry in row u and column v of the matrix is 1, then there is an edge uv in the graph.

Adjacency Matrices (3)

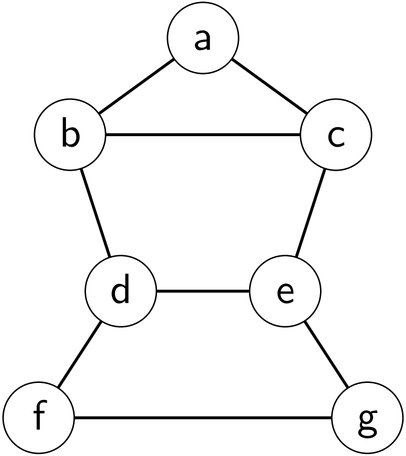


Adjacency Matrix:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	0	1	0	0
d	0	1	0	0	1	1	0
e	0	0	1	1	0	0	1
f	0	0	0	1	0	0	1
g	0	0	0	0	1	1	0

- The adjacency matrix of an undirected graph is always symmetric.
- The space complexity of an adjacency matrix is always $\Theta(n^2)$.

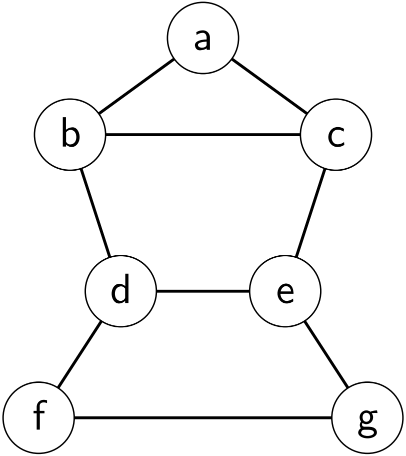
Adjacency Matrices (4)



Adjacency Matrix:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	0	1	0	0
d	0	1	0	0	1	1	0
e	0	0	1	1	0	0	1
f	0	0	0	1	0	0	1
g	0	0	0	0	1	1	0

- On graphs with a small number of edges, adjacency matrices can be inefficient.

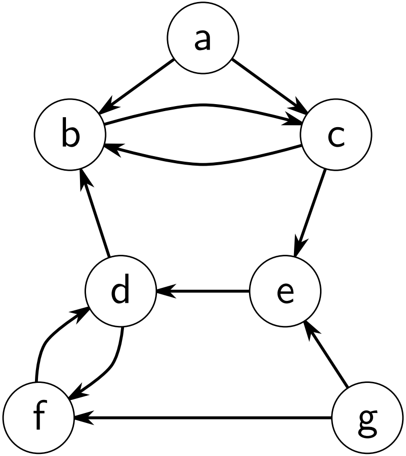


Adjacency Matrix:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	0	1	0	0
d	0	1	0	0	1	1	0
e	0	0	1	1	0	0	1
f	0	0	0	1	0	0	1
g	0	0	0	0	1	1	0

- Finding all of the neighbours of a given vertex requires $\Theta(n)$ operations, since every entry in a row (or column) must be inspected.

Adjacency Matrices (6)

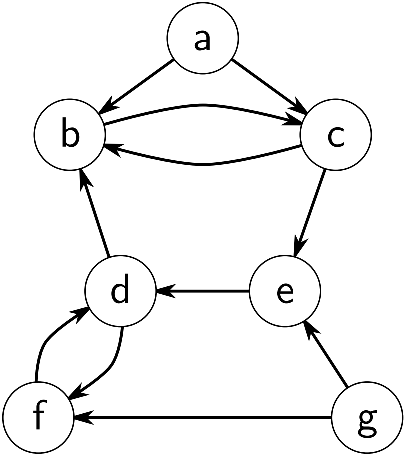


Adjacency Matrix:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	0	0	1	0	0	0	0
c	0	1	0	0	1	0	0
d	0	1	0	0	0	1	0
e	0	0	0	1	0	0	0
f	0	0	0	1	0	0	0
g	0	0	0	0	1	1	0

- For a directed graph, rows represent source vertices and columns represent destination vertices.

Adjacency Matrices (7)

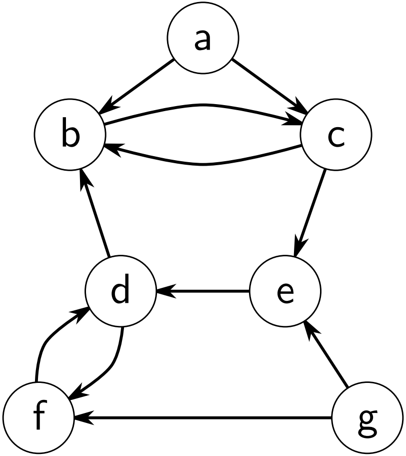


Adjacency Matrix:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	0	0	1	0	0	0	0
c	0	1	0	0	1	0	0
d	0	1	0	0	0	1	0
e	0	0	0	1	0	0	0
f	0	0	0	1	0	0	0
g	0	0	0	0	1	1	0

- If the entry at row u and column v is 1, then there is an edge from u to v.

Adjacency Matrices (8)



Adjacency Matrix:

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

- As with adjacency lists, it can be helpful to assign vertices an index starting at 0 to allow fast indexing into the matrix.