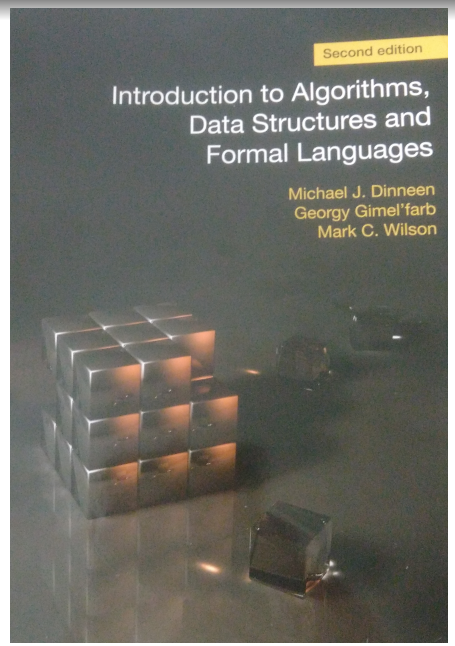
COMPSCI 220: Introduction to Graph Algorithms

Lecturer: **Michael J. Dinneen**[1]

Email: mjd@cs.auckland.ac.nz
Office: 303.425
Web: http://www.cs.auckland.ac.nz/∼mjd

Semester 2, 2023



[1] Thanks to Georgy Gimel'farb for improvements to slides.

Graph ADT (Chapter 4)

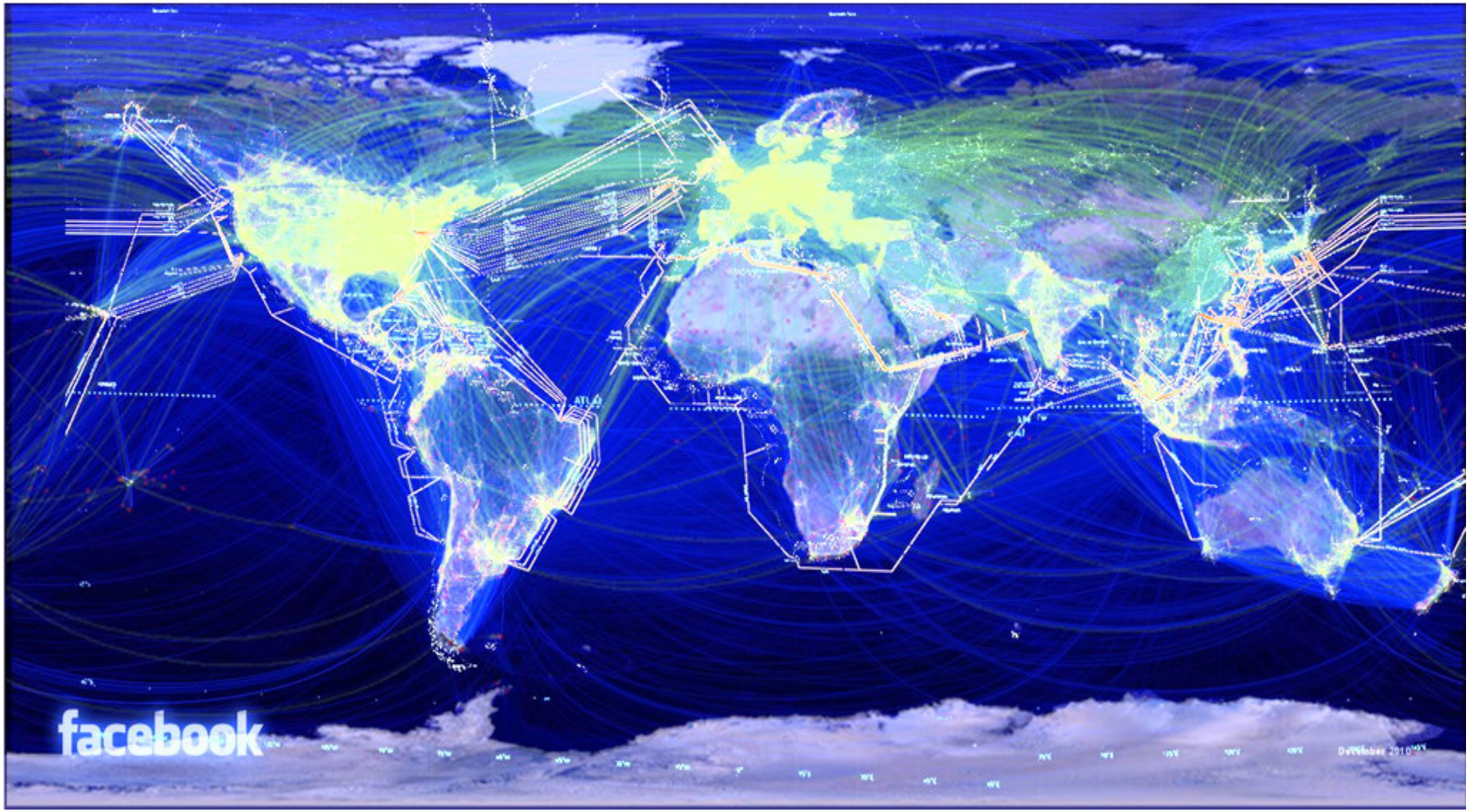
Graphs in Life: World Air Routes

http://milenomics.com/2014/05/partners-alliances-partner-awards/

Graphs in Life: Global Internet Connections

http://www.opte.org/maps/

Graphs in Life: Social Networks (Facebook)

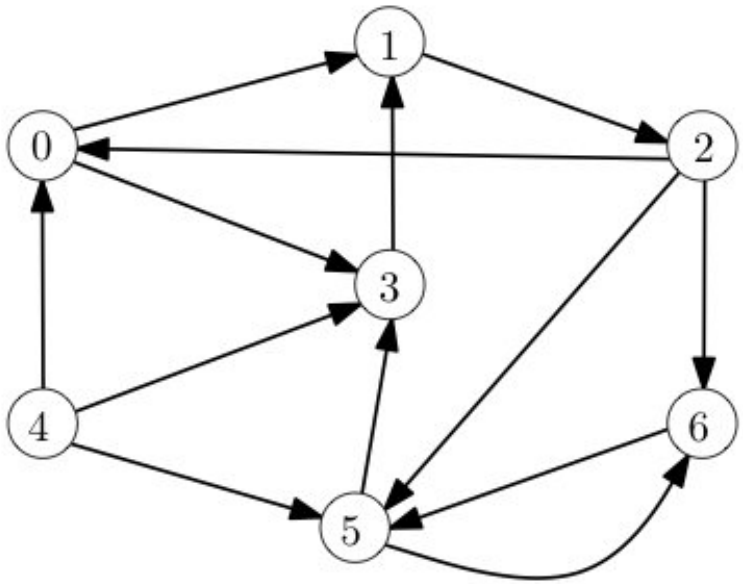


http://robotmonkeys.net/wp-content/uploads/2010/12/social-nets-then-and-now-fb-cities-airlines-data

Directed Graph, or Digraph: Definition

A **digraph** $G = (V, E)$ is a finite nonempty set V of **nodes** together with a (possibly empty) set E of ordered pairs of nodes° of G called **arcs**.

$$
\begin{aligned}
V &= \{\, 0, 1, 2, 3, 4, 5, 6 \,\} \\
E &= \{\, (0,1),\ (0,3), \\
&\quad\ (1,2), \\
&\quad\ (2,0),\ (2,5),\ (2,6), \\
&\quad\ (3,1), \\
&\quad\ (4,0),\ (4,3),\ (4,5), \\
&\quad\ (5,3),\ (5,6), \\
&\quad\ (6,5) \,\}
\end{aligned}
$$

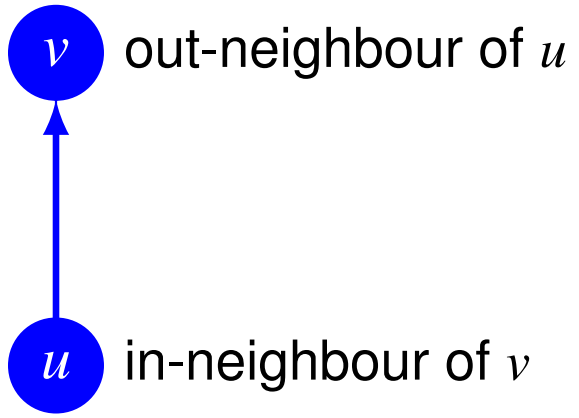


°) Set E is a neighbourhood, or adjacency *relation* on V.

Digraph: Relations of Nodes

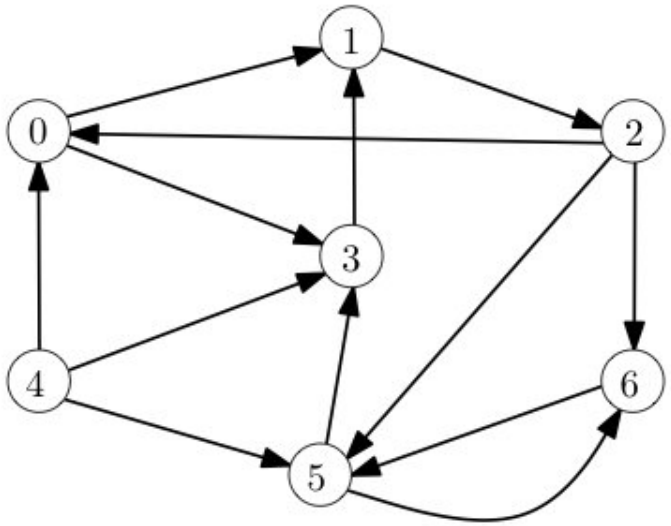
If $(u, v) \in E$,

- v is **adjacent** to u;
- v is an **out-neighbour** of u, and
- u is an **in-neighbour** of v.



Examples:

- Nodes (points) 1 and 3 are adjacent to 0.
- 1 and 3 are out-neighbours of 0.
- 0 is an in-neighbour of 1 and 3.
- Node 1 is adjacent to 3.
- 1 is an out-neighbour of 3.
- 3 is an in-neighbour of 1. . . .
- 5 is an out-neighbour of 2, 4, and 6.

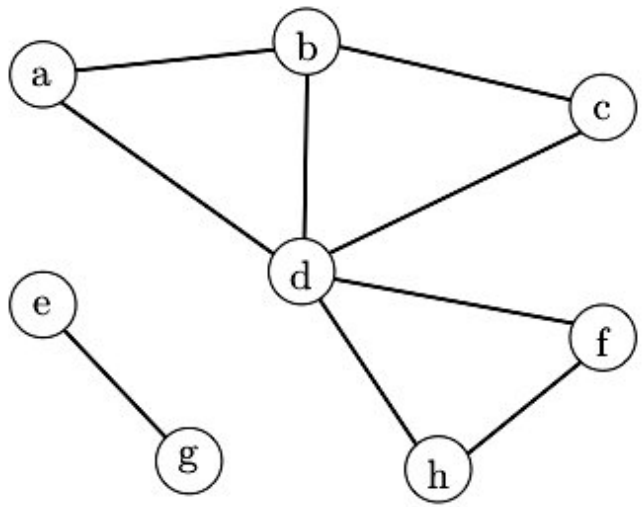


(Undirected) Graph: Definition

A **graph**° $G = (V, E)$ is a finite nonempty set V of **vertices** together with a (possibly empty) set E of unordered pairs of vertices of G called **edges**.

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

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



°) The symmetric digraph: each arc (u, v) has the opposite arc (v, u).

Such a pair is reduced into a single undirected edge that can be traversed in either direction.

Order, Size, and In- / Out-degree

The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.

The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.

For a given n, $m = 0$ **Sparse** digraphs: $|E| \in O(n)$ **Dense** digraphs: $|E| \in \Theta(n^2)$ $n(n-1)$

The **in-degree** or **out-degree** of a node v is the number of arcs entering or leaving v, respectively.

- A node of in-degree 0 – a **source**.
- A node of out-degree 0 – a **sink**.
- This example: the order $|V| = 6$ and the size $|E| = 9$.

Order, Size, and In- / Out-degree

The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.

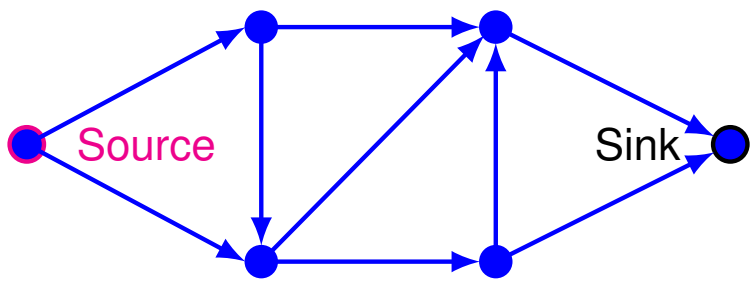
The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.

For a given n, **Sparse** digraphs: $|E| \in \mathrm{O}(n)$ **Dense** digraphs: $|E| \in \Theta(n^2)$

$m = 0$ $\qquad$ $n(n-1)$

The **in-degree** or **out-degree** of a node v is the number of arcs entering or leaving v, respectively.

- A node of in-degree 0 – a **source**.
- A node of out-degree 0 – a **sink**.
- This example: the order $|V| = 6$ and the size $|E| = 9$.



Walk, Path, and Cycle

A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0\, v_1\, \ldots\, v_n$, such that (v_i, v_{i+1}) is an arc in G, i.e., $(v_i, v_{i+1}) \in E$, for each i; $0 \leq i < n$.

- The **length** of the walk $v_0\, v_1\, \ldots\, v_n$ is the number n of arcs involved.
- A **path** is a walk, in which no node is repeated.
- A **cycle** is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from u to v, then there is at least one path from u to v.

Walk, Path, and Cycle

A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0\, v_1\, \ldots\, v_n$, such that (v_i, v_{i+1}) is an arc in G, i.e., $(v_i, v_{i+1}) \in E$, for each i; $0 \leq i < n$.

- The **length** of the walk $v_0\, v_1\, \ldots\, v_n$ is the number n of arcs involved.
- A **path** is a walk, in which no node is repeated.
- A **cycle** is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from u to v, then there is at least one path from u to v.

Walk, Path, and Cycle

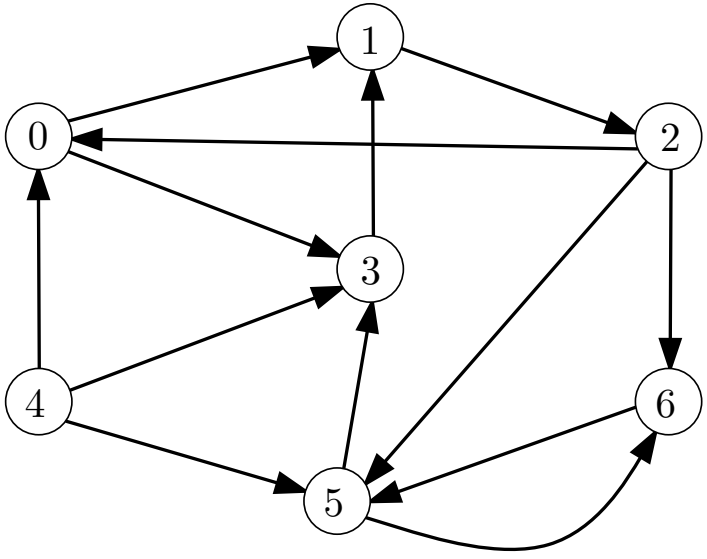
A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0\, v_1\, \ldots\, v_n$, such that (v_i, v_{i+1}) is an arc in G, i.e., $(v_i, v_{i+1}) \in E$, for each i; $0 \leq i < n$.

- The **length** of the walk $v_0\, v_1\, \ldots\, v_n$ is the number n of arcs involved.
- A **path** is a walk, in which no node is repeated.
- A **cycle** is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

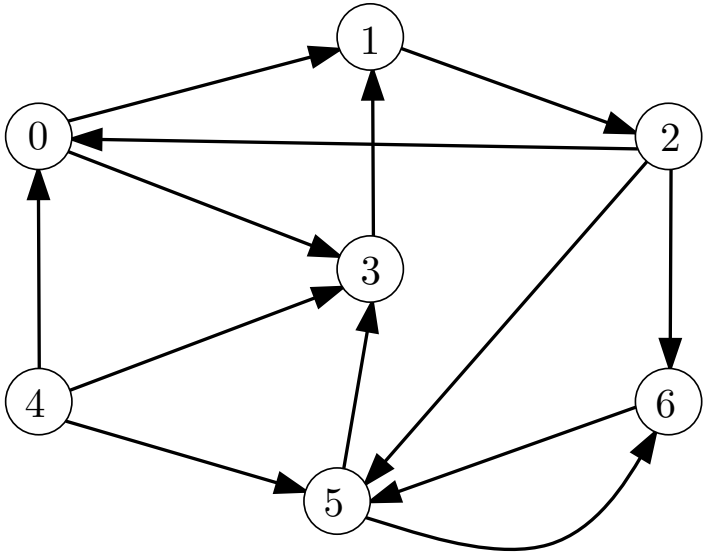
- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from u to v, then there is at least one path from u to v.

Walks, Paths, and Cycles in a Digraph: an Example



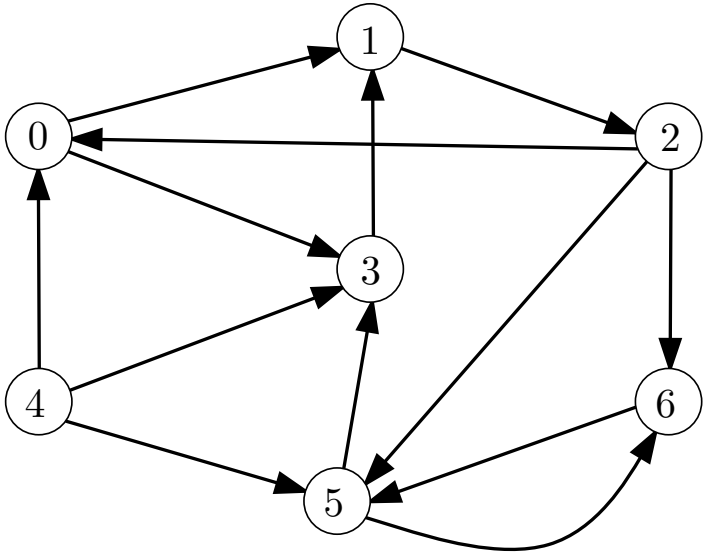
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



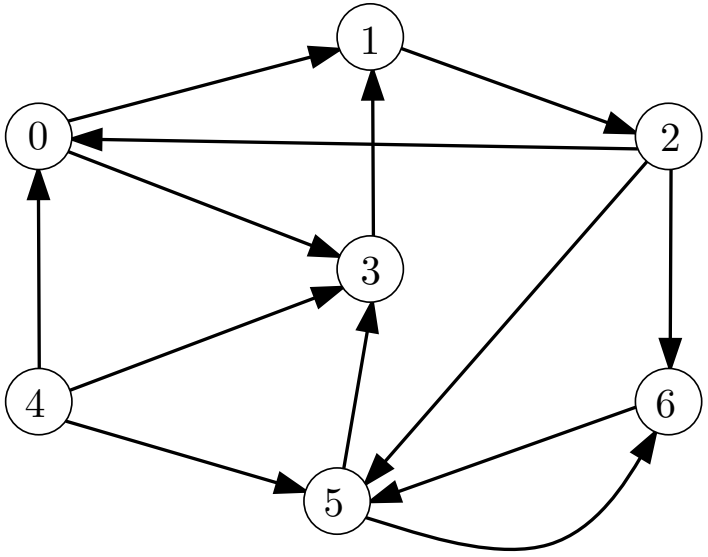
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



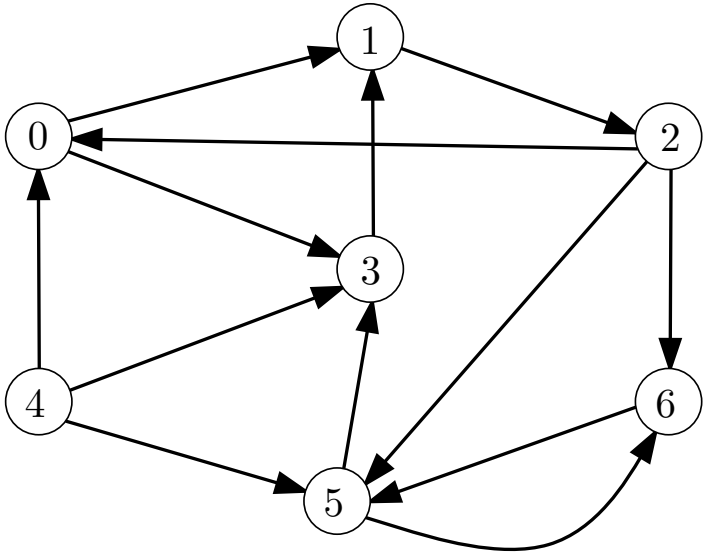
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



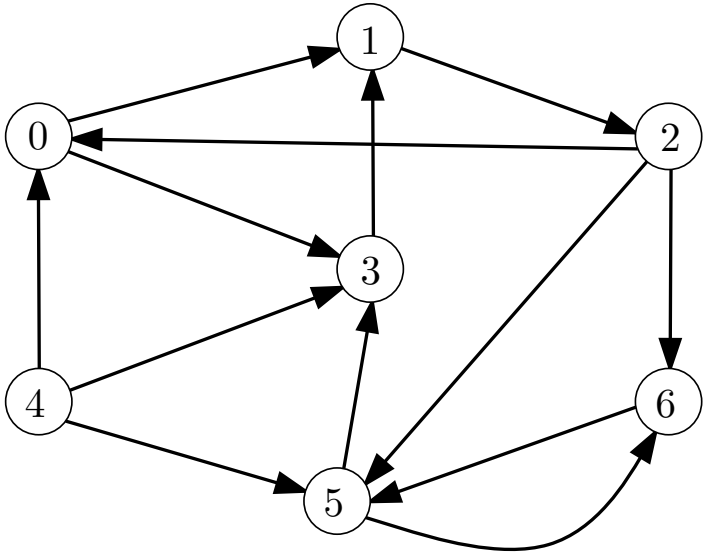
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



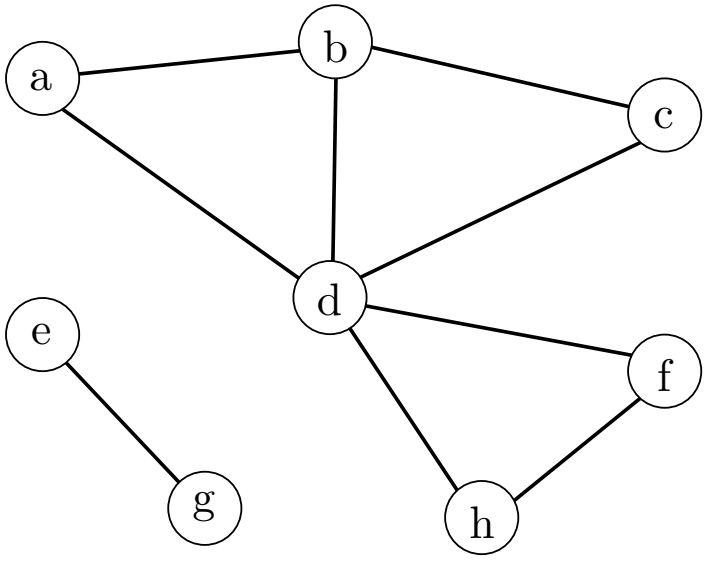
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Digraph: an Example



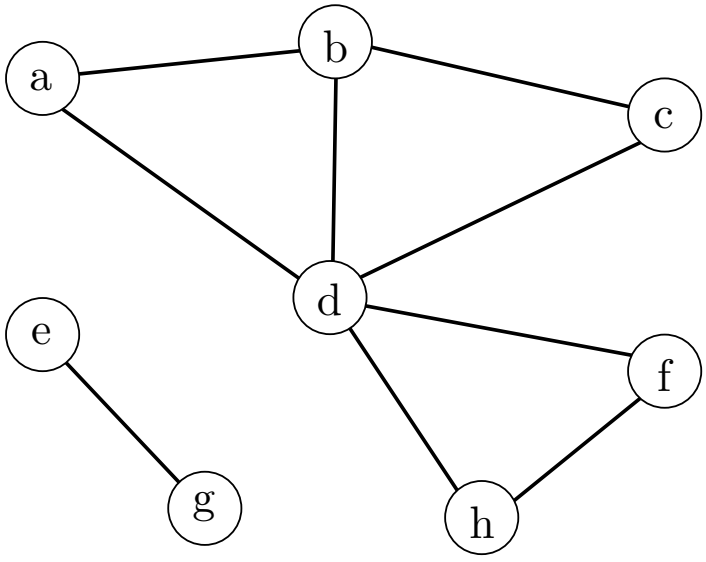
Sequence	Walk?	Path?	Cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, Paths, and Cycles in a Graph: an Example



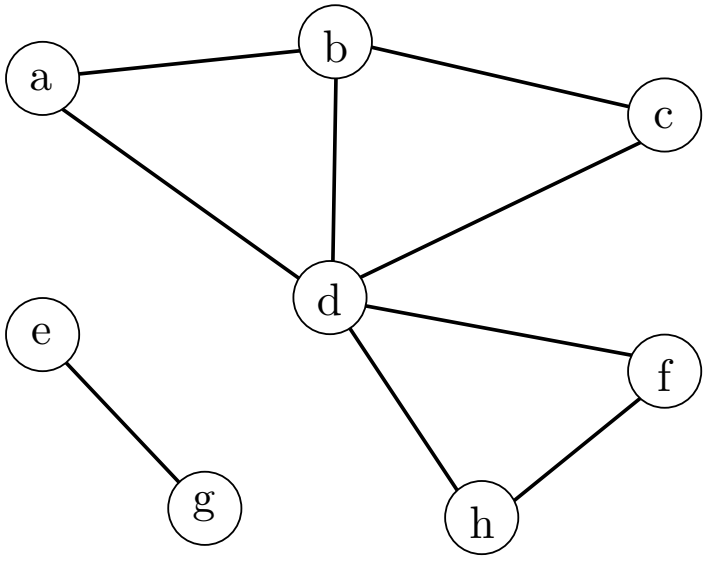
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



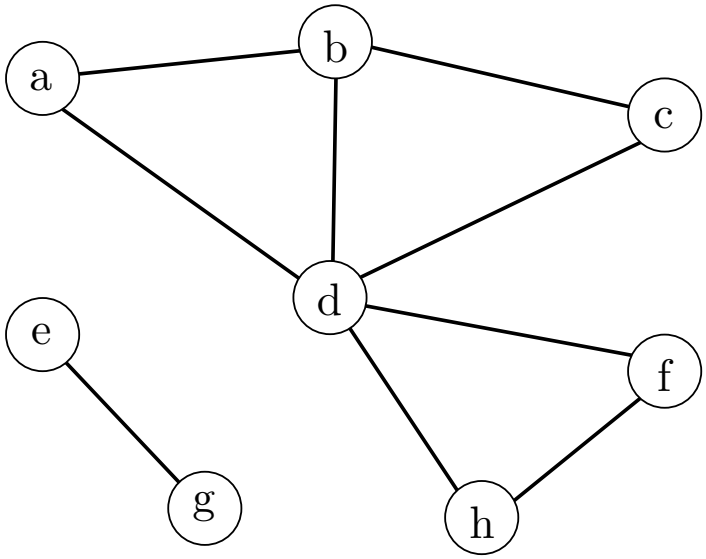
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



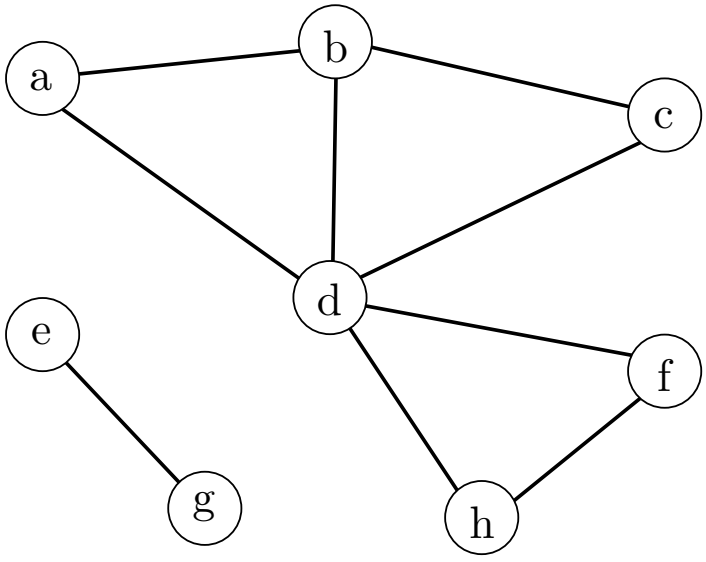
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



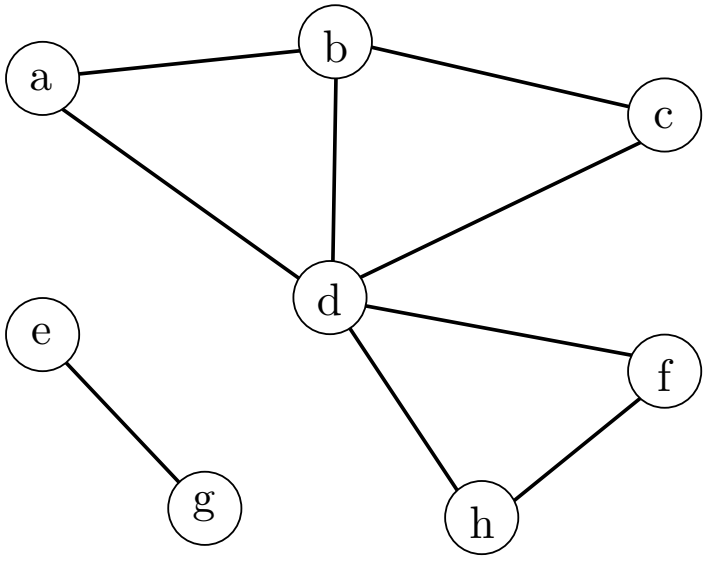
Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Walks, Paths, and Cycles in a Graph: an Example



Sequence	Walk?	Path?	Cycle?
$a\,b\,c$	yes	yes	no
$e\,g\,e$	yes	no	no
$d\,b\,c\,d$	yes	no	yes
$d\,a\,d\,f$	yes	no	no
$a\,b\,d\,f\,h$	yes	yes	no

Digraph $G = (V, E)$: Distances and Diameter

The **distance**, $d(u, v)$, from a node u to a node v in G is the *minimum* length of a path from u to v.

- If no path exists, the distance is undefined or $+\infty$.
- For graphs, $d(u, v) = d(v, u)$ for all vertices u and v.

The **diameter** of G is the *maximum* distance $\max_{u,v \in V}[d(u, v)]$ between any two vertices.

The **radius** of G is $\min_{u \in V} \max_{v \in V}[d(u, v)]$.

A graph is **connected** if it has finite radius and diameter.

Digraph $G = (V, E)$: Distances and Diameter

The **distance**, $d(u, v)$, from a node u to a node v in G is the *minimum* length of a path from u to v.

- If no path exists, the distance is undefined or $+\infty$.
- For graphs, $d(u, v) = d(v, u)$ for all vertices u and v.

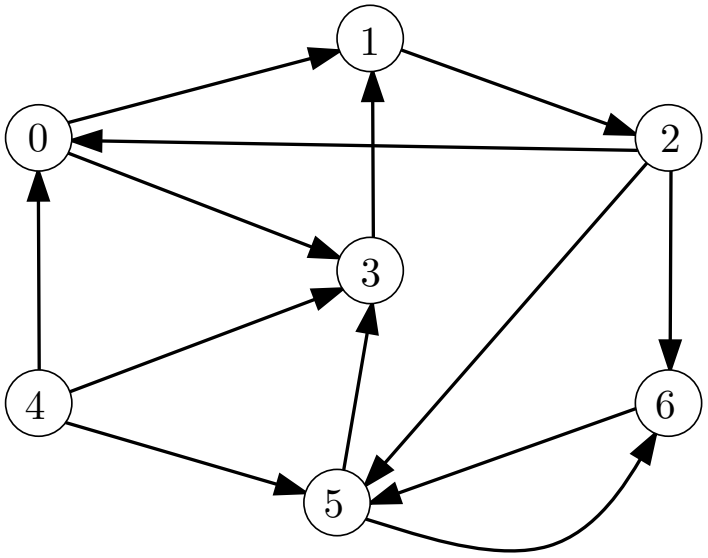
The **diameter** of G is the *maximum* distance $\max_{u,v \in V}[d(u, v)]$ between any two vertices.

The **radius** of G is $\min_{u \in V} \max_{v \in V}[d(u, v)]$.

A graph is **connected** if it has finite radius and diameter.

Path Distances in Digraphs: Examples

$$\begin{aligned} d(0,3) &= \min\{\text{length}_{\text{of } 0,3}; \text{length}_{\text{of } 0,1,2,6,5,3}; \text{length}_{\text{of } 0,1,2,5,3}\} \\ &= \min\{1;5;4\} = 1 \end{aligned}$$



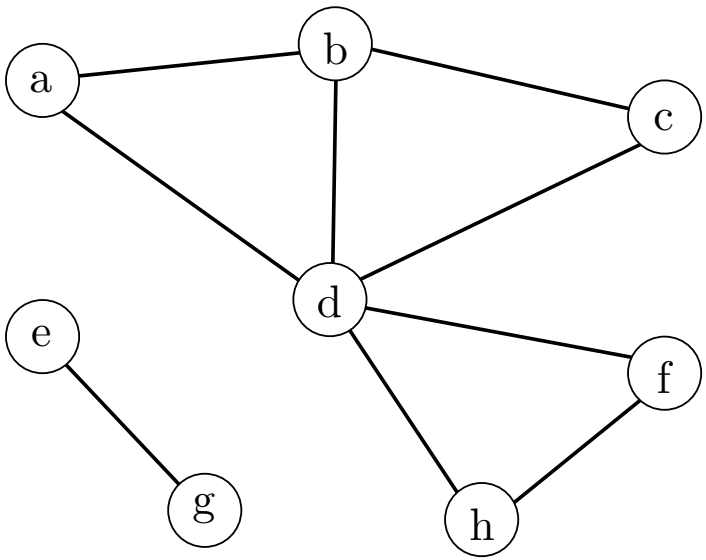
	v						
	0	1	2	3	4	5	6
u=0	–	1	2	1	∞	3	3
u=1	2	–	1	3	∞	2	2
u=2	1	3	–	2	∞	1	1
u=3	3	1	2	–	∞	3	3
u=4	1	2	3	1	–	1	2
u=5	4	2	3	1	∞	–	1
u=6	5	3	4	2	∞	1	–

$d(0,1) = 1$, $d(0,2) = 2$, $d(0,5) = 3$, $d(0,4) = \infty$, $d(5,5) = 0$, $d(5,2) = 3$,
$d(5,0) = 4$, $d(4,6) = 2$, $d(4,1) = 2$, $d(4,2) = 3$

Diameter: $\max\{1,2,1,\infty,3,\ldots,4,\ldots,5,\ldots,1\} = \infty$

Radius: $\min\{\infty,\infty,\ldots,3,\infty,\infty\} = 3$

Path Distances in Graphs: Examples



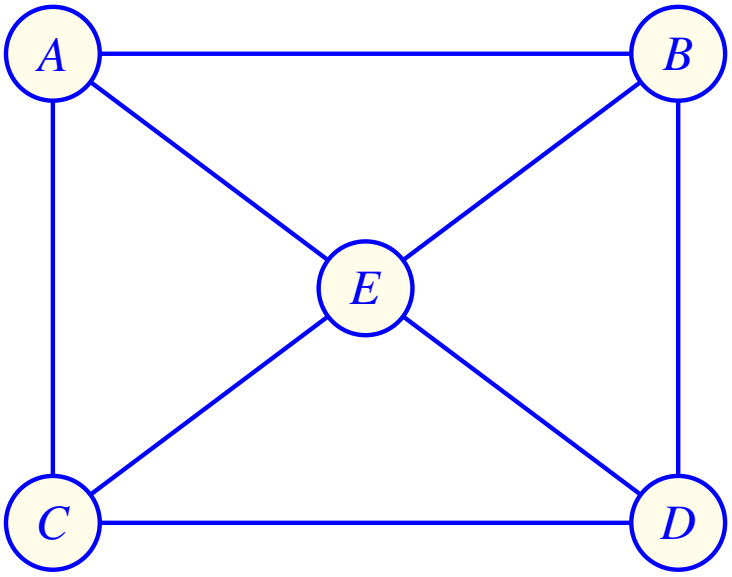
	v a	b	c	d	e	f	g	h
u=a	0	1	2	1	∞	2	∞	2
u=b	1	0	1	1	∞	2	∞	2
u=c	2	1	0	1	∞	2	∞	2
u=d	1	1	1	0	∞	1	∞	1
u=e	∞	∞	∞	∞	0	∞	1	∞
u=f	2	2	2	1	∞	0	∞	1
u=g	∞	∞	∞	∞	1	∞	0	∞
u=h	2	2	2	1	∞	1	∞	0

$d(\mathrm{a},\mathrm{b}) = d(\mathrm{b},\mathrm{a}) = 1$, $d(\mathrm{a},\mathrm{c}) = d(\mathrm{c},\mathrm{a}) = 2$, $d(\mathrm{a},\mathrm{f}) = d(\mathrm{f},\mathrm{a}) = 2$,
$d(\mathrm{a},\mathrm{e}) = d(\mathrm{e},\mathrm{a}) = \infty$, $d(\mathrm{e},\mathrm{e}) = 0$, $d(\mathrm{e},\mathrm{g}) = d(\mathrm{g},\mathrm{e}) = 1$, $d(\mathrm{h},\mathrm{f}) = d(\mathrm{f},\mathrm{h}) = 1$,
$d(\mathrm{d},\mathrm{h}) = d(\mathrm{h},\mathrm{d}) = 1$

Diameter: $\max\{0, 1, 2, 1, \infty, 2, \ldots, 2, \ldots, 2, \ldots, 0\} = \infty$

Radius: $\min\{\infty, \ldots, \infty\} = \infty$

Diameter / Radius of an Unweighted Graph

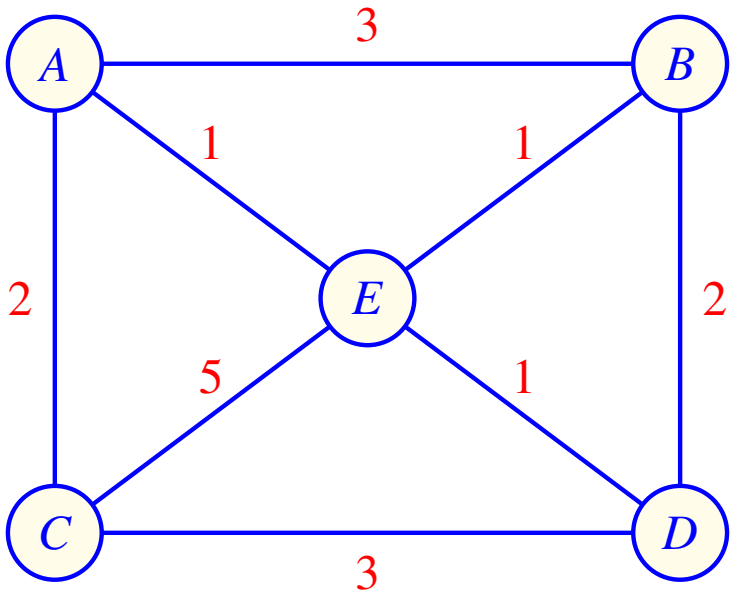


	A	B	C	D	E	$\max_v d(u,v)$
A	0	1	1	2	1	2
B	1	0	2	1	1	2
C	1	2	0	1	1	2
D	2	1	1	0	1	2
E	1	1	1	1	0	1

$$d(C,E) = d(E,C) = 1$$
$$d(B,C) = d(C,B) = 2$$

Radius = 1; diameter = 2.

Diameter / Radius of a Weighted Graph



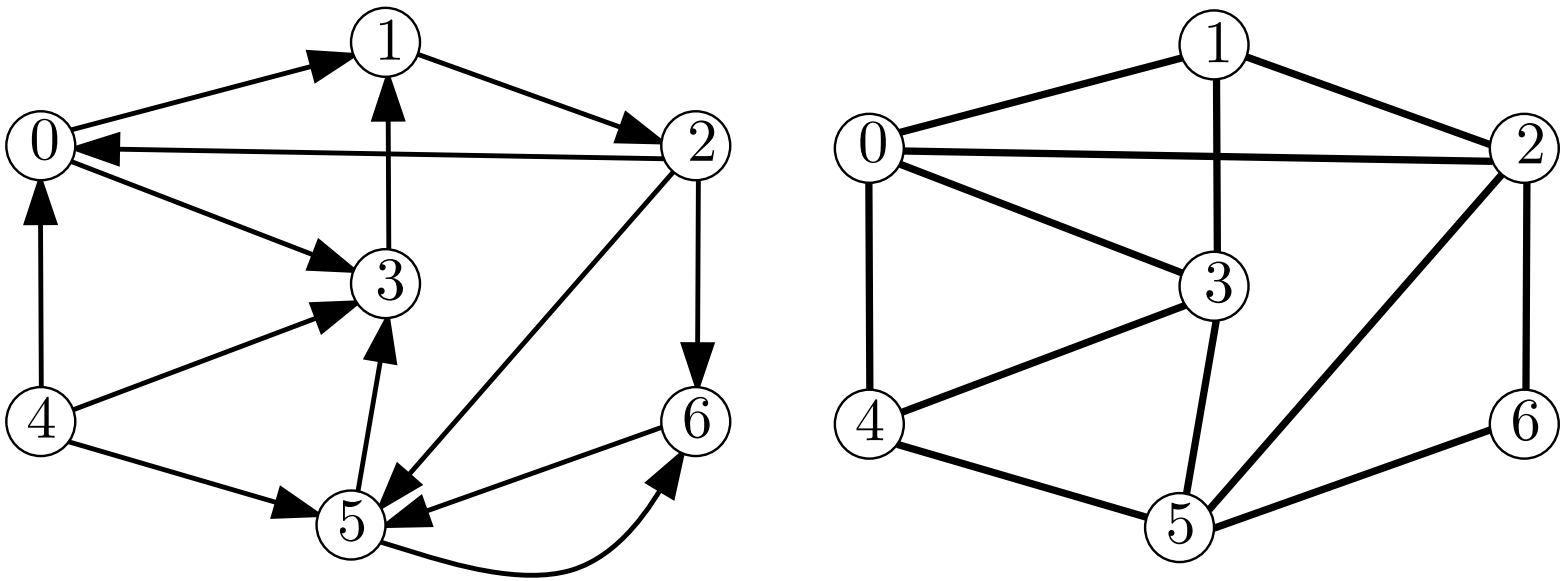
	A	B	C	D	E	$\max_v d(u,v)$
A	0	2	2	2	1	2
B	2	0	4	2	1	4
C	2	4	0	3	3	4
D	2	2	3	0	1	3
E	1	1	3	1	0	3

$$
\begin{aligned}
d(C,E) &= d(E,C) \\
&= \min\{5, 2+1, 3+1, 2+3+1, 3+2+1\} = 3 \\
d(B,C) &= d(C,B) \\
&= \min\{3+2, 1+1+2, 1+5, 1+1+3, 2+3, 2+1+5\} = 4
\end{aligned}
$$

Radius = 2; diameter = 4.

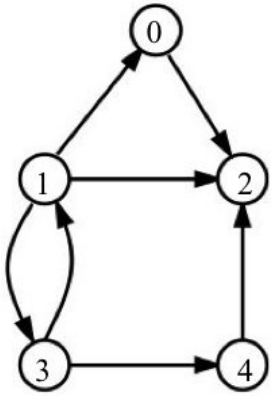
Underlying Graph of a Digraph

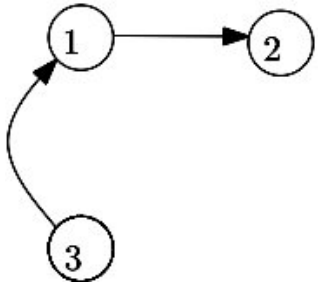
The underlying graph of a digraph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \big\{\{u, v\} \mid (u, v) \in E\big\}$.



Sub(di)graphs

A **subdigraph** of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.

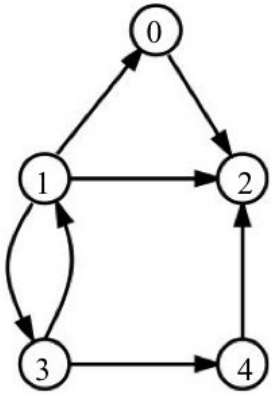


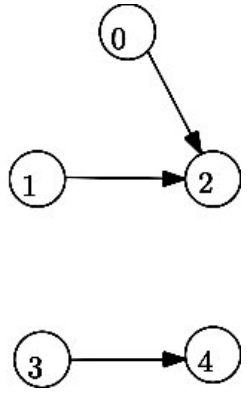


$$G = \begin{pmatrix} V = \{0, 1, 2, 3, 4\}, \\ E = \left\{ \begin{matrix} (0,2), (1,0), (1,2), \\ (1,3), (3,1), (4,2), \\ (3,4) \end{matrix} \right\} \end{pmatrix} \qquad G' = \begin{pmatrix} V' = \{1, 2, 3\}, \\ E' = \{(1,2),\ (3,1)\} \end{pmatrix}$$

Spanning Sub(di)graphs

A **spanning** subdigraph contains all nodes, that is, $V' = V$.

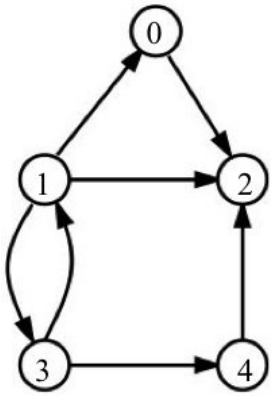


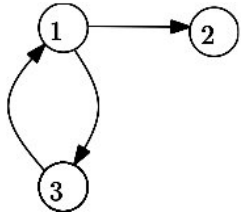


$$G = \begin{pmatrix} V = \{0,1,2,3,4\}, \\ E = \begin{Bmatrix} (0,2),(1,0),(1,2), \\ (1,3),(3,1),(4,2), \\ (3,4) \end{Bmatrix} \end{pmatrix} \qquad G' = \begin{pmatrix} V' = \{0,1,2,3,4\}, \\ E' = \begin{Bmatrix} (0,2),\ (1,2), \\ (3,4) \end{Bmatrix} \end{pmatrix}$$

Induced Sub(di)graphs

The subdigraph **induced** by a subset V' of V is the digraph $G' = (V', E')$ where $E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$.





$$G = \begin{pmatrix} V = \{0, 1, 2, 3, 4\}, \\ E = \begin{Bmatrix} (0, 2), (1, 0), (1, 2), \\ (1, 3), (3, 1), (4, 2), \\ (3, 4) \end{Bmatrix} \end{pmatrix} \qquad G' = \begin{pmatrix} V' = \{1, 2, 3\}, \\ E' = \begin{Bmatrix} (1, 2), \ (1, 3), \\ (3, 1) \end{Bmatrix} \end{pmatrix}$$

Digraphs: Computer Representation

For a digraph G of order n with the vertices, V, labelled $0, 1, \ldots, n-1$:

The adjacency matrix of G:

The $n \times n$ boolean matrix (often encoded with 0's and 1's) such that its entry (i,j) is true if and only if there is an arc (i,j) from the node i to node j.

An adjacency list of G:

A sequence of n sequences, $L_0, \ldots, L_{n-1}$, such that the sequence L_i contains all nodes of G that are adjacent to the node i.

Each sequence L_i may not be sorted! But we often sort them.

Digraphs: Computer Representation

For a digraph G of order n with the vertices, V, labelled $0, 1, \ldots, n-1$:

The adjacency matrix of G:

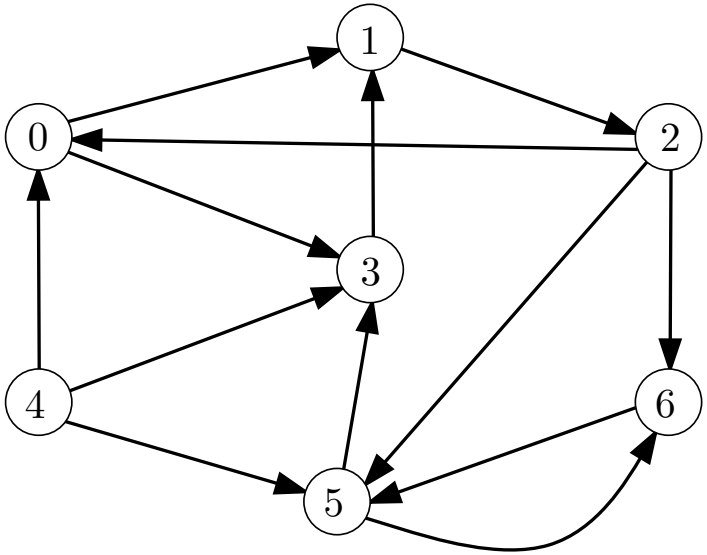
The $n \times n$ boolean matrix (often encoded with 0's and 1's) such that its entry (i, j) is true if and only if there is an arc (i, j) from the node i to node j.

An adjacency list of G:

A sequence of n sequences, $L_0, \ldots, L_{n-1}$, such that the sequence L_i contains all nodes of G that are adjacent to the node i.

Each sequence L_i may not be sorted! But we often sort them.

Adjacency Matrix of a Digraph



Digraph $G = (V, E)$

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

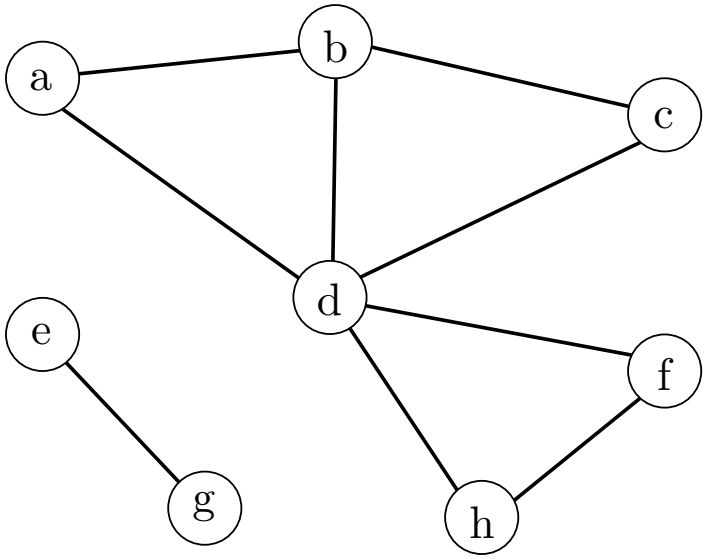
Adjacency matrix of G:

0 – a non-adjacent pair of vertices: $(i, j) \notin E$

1 – an adjacent pair of vertices: $(i, j) \in E$

The number of 1's in a row (column) is the out-(in-) degree of the related node.

Adjacency Lists of a Graph



Graph $G = (V, E)$

symbolic	**numeric**
	8
0 = a: b d	1 3
1 = b: a c d	0 2 3
2 = c: b d	1 3
3 = d: a b c f h	0 1 2 5 7
4 = e: g	6
5 = f: d h	3 7
6 = g: e	4
7 = h: d f	3 5

Here the first line of the **numeric** representation is the graph's order n. Vertices are usually indexed from 0 to $n - 1$.

Digraph Operations w.r.t. Data Structures

Operation	**Adjacency Matrix**	**Adjacency Lists**
arc (i,j) exists?	is entry (i,j) 0 or 1	find j in list i
out-degree of i	scan row and sum 1's	size of list i
in-degree of i	scan column and sum 1's	for $j \neq i$, find i in list j
add arc (i,j)	change entry (i,j)	insert j in list i
delete arc (i,j)	change entry (i,j)	delete j from list i
add node	create new row/column	add new list at end
delete node i	delete row/column i and shuffle other entries	delete list i and for $j \neq i$, delete i from list j

Adjacency Lists / Matrices: Comparative Performance

$$G = (V, E) \quad \longrightarrow \quad n = |V|; \quad m = |E|$$

Operation	**adj. matrix**	**adj. lists**
arc (i, j) exists?	$\Theta(1)$	$\Theta(\alpha)$
out-degree of i	$\Theta(n)$	$\Theta(1)$
in-degree of i	$\Theta(n)$	$\Theta(n + m)$
add arc (i, j)	$\Theta(1)$	$\Theta(1)$
delete arc (i, j)	$\Theta(1)$	$\Theta(\alpha)$
add node	$\Theta(n)$	$\Theta(1)$
delete node i	$\Theta(n^2)$	$\Theta(n + m)$

Here, α denotes size of the adjacency list for vertex i.

Other Graph Representations

Special cases can be stored more efficiently or convenient for algorithms:

- A complete binary tree or a heap: in an array.
- A general rooted tree: in an array `pred` of size n; `pred[i]` – a pointer to the parent of node i.
- Matematical: Set of vertices and set of edges (pairs/tuples).
- Lists of dictionaries (of neighbours).
- Edge incidence matrix (see COMPSCI 320).
- Sparse graphs of bounded pathwidth, treewidth or branchwidth (see COMPSCI 720).

Graph Adjacency Matrix in Python

```
n = 3 # order of graph
M = [[0]*n for _ in range(n)] # can use False/True
M[0][2],M[1][2] = 1,1 # add two arcs (directed edges)
print("M1:", M)
M[1][0],M[1][2],M[2][1] = 1,0,1 # add two, remove one
print("M2:",M)
M.append([0]*n) # add new node/vertex (row)
n += 1
for i in range(n): M[i].append(0) # extend all rows
print("M3:",M)
```

Output should be:

```
M1: [[0, 0, 1], [0, 0, 1], [0, 0, 0]]
M2: [[0, 0, 1], [1, 0, 0], [0, 1, 0]]
M3: [[0, 0, 1, 0], [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
```

Graph Adjacency Lists in Python

```
n = 3 # order of graph
M = [[] for _ in range(n)]       # n emtpy lists
M[0].append(2); M[1].append(2)  # add two arcs
print("M1:", M)
M[1].append(0); M[2].append(1)  # add two more arcs
M[1].remove(2)                  # and remove one
print("M2:",M)
n += 1              # add new node/vertex (list)
M.append([])
print("M3:",M)
```

Output should be:

```
M1: [[2], [2], []]
M2: [[2], [0], [1]]
M3: [[2], [0], [1], []]
```

Simple Graph ADT in Python

```
class AdjacencyLists:
'''
  AdjacencyLists stores the state of vertices and arcs
  as a list of lists.  The arc (i,j) is represented by
  the list at element i containing the value j.
'''
def __init__(self):
    self._adj = [] # Creates an internal empty array

def empty(self): self._adj = []

def order(self):
    return len(self._adj)

def size(self):
    return reduce(
        lambda x, y : x + y,   # Add up the lengths of
        map (len, self._adj), # every row's list
        0 )                    # Initial value 0
```

```
def addVertices(self, n):
    if not 0 <= n:
        raise ValueError('Argument is negative!')

    for i in range(n): self._adj.append([])

def removeVertex(self, i):
    if not 0 <= i < self.order():
        raise ValueError('Arguments out of bounds')

    del self._adj[i] # Delete this vertex's list

    for otherVertex in range(self.order()):
        current = self._adj[otherVertex]
        try:
            current.remove(i) # Try to remove node i
        except ValueError:
            pass # It didn't appear

        for j in range(len(current)): # Relabel
            if current[j] > i: current[j] -= 1
```

```
def addArc(self, i, j):
    if not (0 <= i < self.order() and \
            0 <= j < self.order()):
        raise ValueError('Arguments out of bounds')

    if not self.isArc(i, j):    # Avoid multi-edges
        self._adj[i].append(j)

def removeArc(self, i, j):
    if not (0 <= i < self.order() and \
            0 <= j < self.order()):
        raise ValueError('Arguments out of bounds')
    try:
        self._adj[i].remove(j)
    except ValueError:
        pass # Not found

def addEdge(self, i, j):
    self.addArc(i,j); self.addArc(j,i)
def removeEdge(self, i, j):
    self.removeArc(i,j); self.removeArc(j,i)
```

```
def isArc(self, i, j):
    if not (0 <= i < self.order() and \
            0 <= j < self.order()):
        raise ValueError('Arguments out of bounds')
    return j in self._adj[i]

def isEdge(self, i, j):
    return self.isArc(i,j) and self.isArc(j,i)

def degree(self, i):
    if not 0 <= i < self.order():
        raise ValueError('Argument out of bounds')
    return len(self._adj[i])

def inDegree(self, i):
    if not 0 <= i < self.order():
        raise ValueError('Argument out of bounds')
    retval = 0
    for j in range(self.order()):
        if self.isArc(j, i): retval += 1
    return retval
```

```
def neighbors(self, i):
    if not 0 <= i < self.order():
        raise ValueError('Argument out of bounds')

    # Returning a copy that is safe for modification
    return sorted(self._adj[i])

neighbours = neighbors # Permit the Old English /
                              British spelling

def read(self, stream): # CompSci 220 format
    self.empty()
    order = int(stream.readline().strip())
    self.addVertices(order)
    for i in range(order):
        for neighbor in map(int, \
                            stream.readline().strip().split()):
            self.addArc(i, neighbor)
```