7SENG010W Data Structres & Algorithms

Week 9 Lecture

Graphs

Overview of Week 9 Lecture: Graphs

PART I

Introduction to Maths for Graphs:

Sets, Mappings, Ordered Pairs

What is a Set?

Definition: Set

A set is a collection of values called elements or members & all the members of a set must be of the same type.

Where in the above definition the *"same type"* means the *"same kind of thing"*.

For example, you can define *sets* of basically anything by listing the *elements* between set brackets – "{" & "}" separated by commas – "," as follows:

- *"numbers"* — $\{ 0, 3, 45, 999, \ldots \}$
- *"people"* — $\{ Bill, Sue, Joe, Mary, Ian, Jim, \ldots \}$
- *"students"* — $\{ JimJones, MarySmith, \ldots \}$
- *"MSc Level 7 modules"* — $\{ 7SENG010W, 7SENG013W, \ldots \}$
- *"colours"* — $\{ \mathbf{red}, \mathbf{blue}, \mathbf{green}, \mathbf{purple}, \mathbf{orange}, \mathbf{black}, \ldots \}$
- *"days of the week"* — $\{ Monday, Tuesday, Wednesday, \ldots \}$

Sets Ignore: Order of Elements & Duplicate Values

Two important properties of a set are:

- *No Duplicate Values:* an individual *element* is **in** a set or it is **not in** a set.

  It **does not** record *how many times an element occurs in a set*.

  So sets **do not include duplicate values**.

- *No Ordering of Elements:* in a set the *order* in which the elements in a set are listed does not matter.

  Any order will do for the elements.

So when we write out a set we do **not include duplicate values** & do not care about the **order of the elements**, so the following sets are all equal:

$$
\begin{aligned}
&\{\, 1,\ 2,\ 2,\ 3,\ 3,\ 3,\ 4,\ 4,\ 4,\ 4,\ 5,\ 5,\ 5,\ 5,\ 5 \,\} \\
=&\{\, 1,\ 5,\ 3,\ 2,\ 3,\ 4,\ 2,\ 4,\ 5 \,\} \\
=&\{\, 5,\ 3,\ 1,\ 2,\ 4 \,\} \\
=&\{\, 1,\ 2,\ 3,\ 4,\ 5 \,\} \qquad [\text{Standard form}]
\end{aligned}
$$

Sets: Checking Membership & Size

The following general standard set definition operators are available.

Symbol	Description
$x \in A$	*membership* – x is an *element* (or *member*) of A
$x \notin A$	*non-membership* – x is *not* an element (member) of A
$\{\ \}$	*Empty set*
$\{\ 1,\ 2,\ 3\ \}$	Set of *elements*: 1, 2, 3
$\mathrm{card}(A)$	*cardinality* – *number of elements* in set A

Examples

$$A = \{\ a,\ b,\ c,\ d,\ e,\ f,\ g,\ h\ \} \qquad B = \{\ a,\ e,\ i,\ o,\ u\ \}$$
$$C = \{\ x,\ y,\ z\ \} \qquad D = \{\ f,\ o,\ r,\ m,\ a,\ l,\ e,\ t,\ h,\ d,\ s\ \}$$

We have:

$$1 \in \{\ 1\ \} \qquad 3 \notin \{\ 1,\ 2,\ 4,\ 9\ \} \qquad 5 \notin \{\ \}$$
$$a \in A \qquad x \notin B \qquad f \in D$$
$$\mathrm{card}(A) = 8 \qquad \mathrm{card}(B) = 5 \qquad \mathrm{card}(C) = 3 \qquad \mathrm{card}(D) = 11$$

Comparing Sets

The following allow sets to be compared, provided they are the same type.

Symbol	Description
$A = B$	*equality* – A is *equal to* B
$A \neq B$	*non-equality* – A is *not equal to* B
$A \subseteq B$	*subset* – A is a *subset of or equal to* B

Examples: Where X is any set we have:

$$\{ 1,\ 2\ ,3 \} = \{ 3,\ 2,\ 1 \}$$

$$\{ 77,\ 88,\ 99 \} \neq \{ 33,\ 66,\ 99 \}$$

$$\{ 77,\ 88,\ 99 \} \subseteq \{ 77,\ 88,\ 99 \} \qquad [\ Equal\ \text{"}=\text{"}\ ]$$

$$\{ 3,\ 4\ ,5 \} \subseteq \{ 1,\ 2,\ 3,\ 4\ ,5 \} \qquad [\ Subset\ \text{"}\subset\text{"}\ ]$$

$$X \subseteq X$$

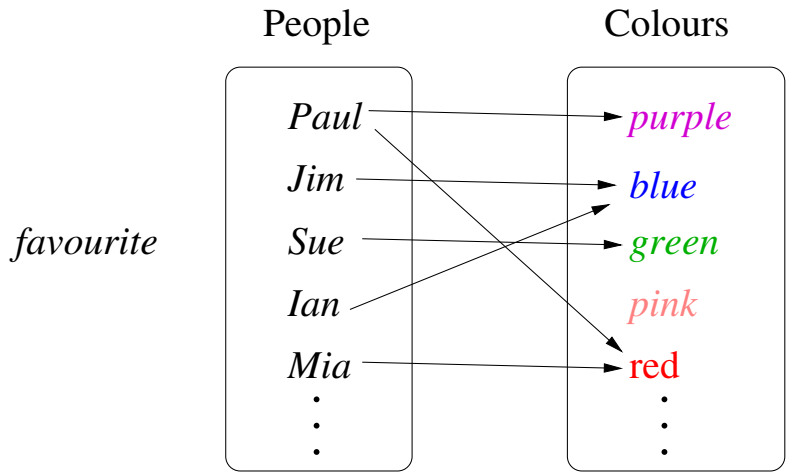
$$\{\ \} \subseteq X$$

Mappings

A *mapping* "*maps*" the elements of one set to the elements of another set.

The two sets can be *different sets* or the *same set*.

For example, we can create a $favourite$ *mapping* that maps *people* to their favourite *colour* as follows:



The $favourite$ mapping can be represented by a *set of pairs of values*:

$$favourite = \{ (Paul, purple),\ (Paul, red),\ (Jim, blue),\ (Sue, green),\ (Ian, blue),\ (Mia, red),\ \ldots \}$$

An *element* of this mapping (set) is a *pair of values*, e.g. "$(Paul,\ purple)$", meaning that $Paul$ & $purple$ are *"related"* or *"connected"*.

Ordered Pairs, Maplets & Mappings

An *ordered pair*, can also be defined using the *maplet* notation:

$$x \mapsto y \quad = \quad (x, y)$$

meaning that *"x maps to y"*, e.g. in $favourite$:

$$Paul \mapsto purple \quad = \quad (Paul,\ purple)$$

Also note that **order of the values in an ordered-pair/maplet is important**:

$$(Paul,\ purple) \quad \neq \quad (purple,\ Paul)$$
$$Mia \mapsto red \quad \neq \quad red \mapsto Mia$$

Can check if a pair is "*mapped*" by a mapping using set membership:

$$(Paul,\ purple) \quad \in \quad favourite \qquad [\ \textit{Is mapped}\ ]$$
$$Paul \mapsto orange \quad \notin \quad favourite \qquad [\ \textit{Is not mapped}\ ]$$

Can use this notation to represent that to travel from $Oxford\ Circus$ to $Green\ Park$ on the Victoria tube line takes 2 minutes as follows:

$$(Oxford\ Circus,\ Green\ Park) \mapsto 2$$

Here the *ordered pair* of tube stations *maps to* the time taken to travel between them.

Defining a Set using a Condition

A set can be constructed by giving a *condition* that must hold for all the members of the set.

This is called *set comprehension* & has the general form:

$$\{\ variables \mid Condition(variables)\ \}$$

- $variables$ represents arbitrary *elements* of the set being defined.
- The $Condition$ is the *condition* that elements of the set *must satisfy* to be included in the set being defined.

  So if the $Condition$ is **not true** for a particular variable or variables then they do **not get included in the set**.

For example, the set of numbers 0 to 10:

$$\{\ x \mid x \leq 10\ \} = \{\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10\ \}$$

- "x" is the $variable$ part.
- "$x \leq 10$" is the $Condition(variable)$ part, i.e. the $Condition$ is "$\leq\ 10$".

The *elements of the set* are the *numbers* that satisfy the condition ($x \leq 10$).

Examples of Set Comprehension

Given sets: $A = \{ 1, 2, 3 \}, \quad B = \{ 77, 88, 99 \}$

$C \subseteq \{ \{x, y\} \mid x \in A \,\&\, y \in B \}$ [*Set of Sets*]

$C = \{ \{1, 77\}, \{88, 1\}, \{1, 99\}, \{77, 2\}, \{88, 2\}, \{2, 99\}, \{77, 3\}, \{3, 88\}, \{99, 3\} \}$

$D \subseteq \{ (x, y) \mid x \in A \,\&\, y \in B \}$ [*Set of Ordered-Pairs*]

$D = \{ (1, 77), (1, 88), (1, 99), (2, 77), (2, 88), (2, 99), (3, 77), (3, 88), (3, 99) \}$

$E \subseteq \{ (x, y) \mid x, y \in \boldsymbol{A} \}$ [*Same Set*]

$E = \{ (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3) \}$

$F \subseteq \{ (x, y) \mid x, y \in A \,\&\, x \neq y \}$ [*Different* x & y]

$F = \{ (1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2) \}$

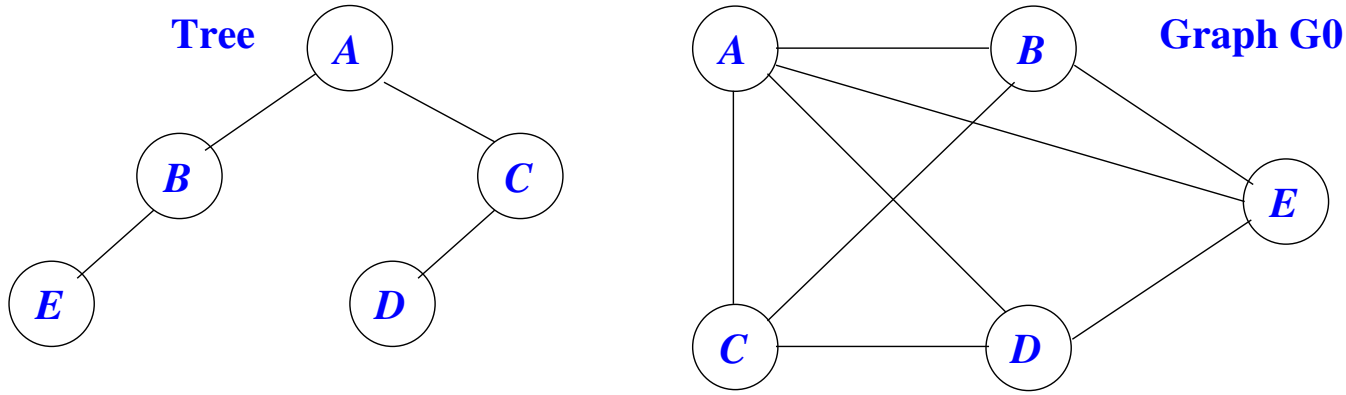
E.g. can represent different pairs of stations on the Victoria line:

$\{ (st1, st2) \mid st1, st2 \in VictoriaLine \,\&\, st1 \neq st2 \}$

PART II

Introduction to Graphs

Introduction to Graphs



A **graph** is a *more general non-linear data structure* than a **tree**, because each item (node) may have *zero or more successors and predecessors*.

With a *tree* each item (node) can have any number of successors, i.e. its children nodes, but either no predecessor, i.e. the root, or just one i.e. its parent node for all non-root nodes.

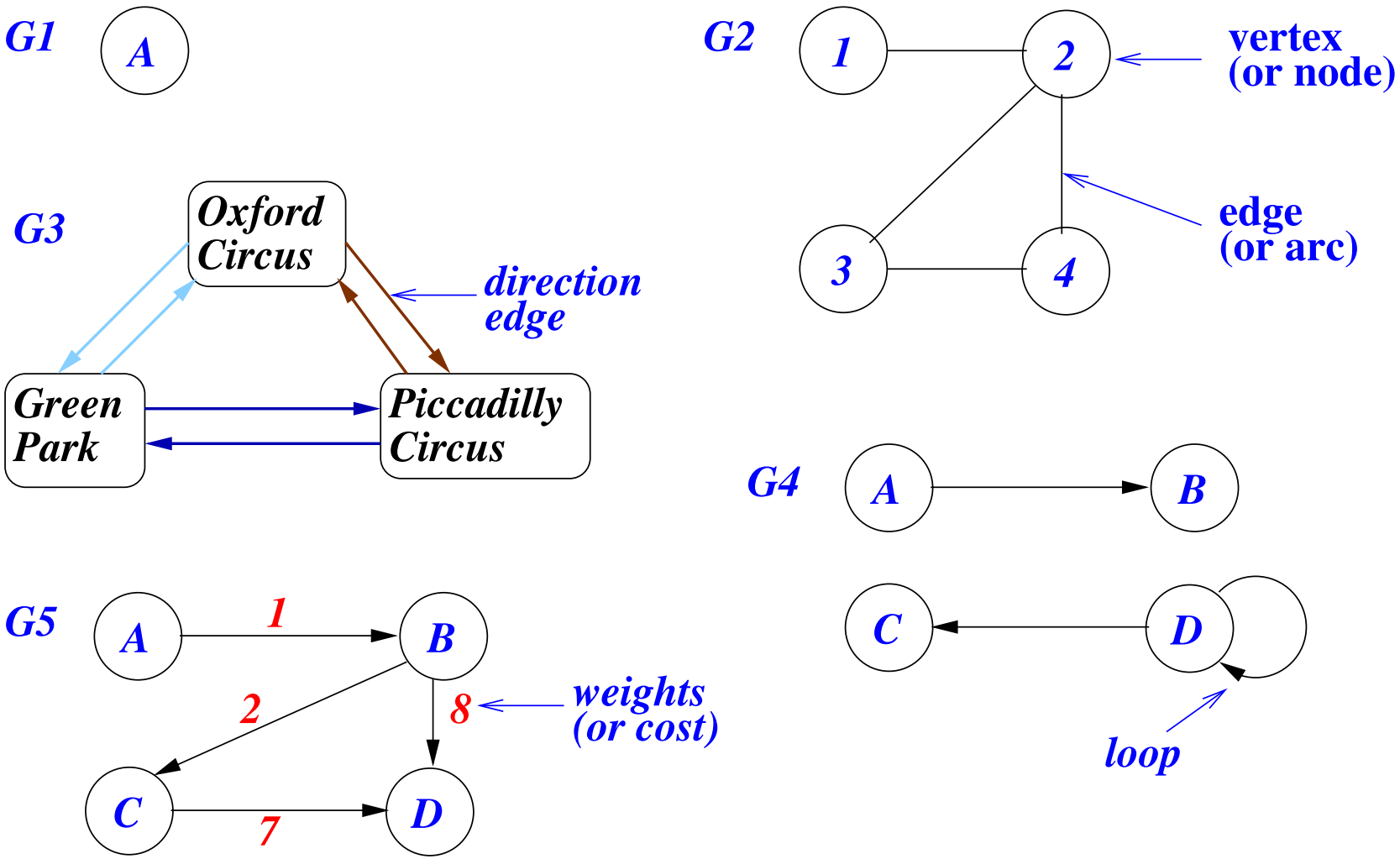
Whereas in a *graph* any item (node) may have **any number of** *predecessors* and/or *successors*.

Also a tree **cannot** have a "*cycle*" (a path from a tree node back to itself), but graphs do.

This means that a *tree* is a *special case of* a *graph*, i.e. all trees are graphs, but **not all graphs are trees**.

Examples of Graphs

Examples of different types of graphs *G1* – *G5*.



Example Graphs *G1* - *G5*

- *G1* has a single **vertex** *A*, but **no edges**.
- *G2* has 4 **vertices**: 1, 2, 3, 4, and 4 **edges** connecting: 1 & 2, 2 & 3, 2 & 4, 3 & 4.
- *G3* has 3 tube stations as vertices and 6 **directional edges**, i.e. each edge is assigned a **direction**.
  E.g. **two distinct edges** between stations: *Oxford Circus* **to** *Green Park* and *Green Park* **to** *Oxford Circus*.
- *G4* has 4 vertices: *A, B, C, D* and 3 directional edges: *A* **to** *B*, *D* **to** *C*, *D* **to** *D* (known as a **loop** (or **self-loop**).
- *G5* has 4 vertices: *A, B, C, D* and 4 directional edges: *A* **to** *B*, *B* **to** *C*, *B* **to** *D*, *C* **to** *D*.
  In addition, each edge has a value, know as its "**weight**", assigned to it, e.g. *A* **to** *B* has **weight 1**, *B* **to** *D* has **weight 8**.

So a *vertex* can be thought of as a point and an *edge* as a line or arrow connecting a pair of points.

Example Graph's Properties

Properties of the graphs:

- *G1* and *G2* are **undirected** graphs, as their edges do not have a direction, i.e. no arrows on the edges.
- But *G3*, *G4* and *G5* are **directed** graphs (or **digraphs**), as their edges all have a direction, i.e. arrows on the edges.
- *G1*, *G2*, *G3* and *G5* are examples of **connected** graphs, meaning all vertices can be visited by using an edge.
- The two *directed* graphs *G3* and *G5* are still considered to be *connected*, because the **direction of an edge is ignored** when determining if a graph is connected.
- *G4* is an example of a **disconnected graph**, as *A* and *B* are connected, as are *C* and *D* but there is no edge connecting these two sub-graphs.

Definition of a Graph

The formal definition of a **graph** is as follows.

Definition: Graph
A **graph** G, where $G = (V,\ E)$, is defined as follows:

- V is a set of **vertices** (or **nodes**), where $V = \{\ v_1,\ v_2,\ \ldots,\ v_m\ \}$.
- E is a set of **edges** (or **arcs**), where $E = \{\ e_1,\ e_2,\ \ldots,\ e_n\ \}$.
  Where an **edge** represents a *connection* between pairs of *vertices*.

So formally an *edge* $e_k \in E$ is defined as a *pair of vertices* $e_k = (v_i,\ v_j)$, where v_i and v_j are members of the vertices set V, i.e. $v_i,\ v_j \in V$.

This means that the set of edges E is a **binary relation** between *vertices* that represents the **adjacency relation** between *vertices*.

$$V = \{\ v_1,\ v_2,\ \ldots,\ v_m\ \} \qquad [\ \mathrm{card}(V) = m\ ]$$
$$E = \{\ e_1,\ e_2,\ \ldots,\ e_n\ \} \qquad [\ \mathrm{card}(E) = n\ ]$$
$$E \subseteq \{\ (v_i,\ v_j)\ |\ v_i,\ v_j \in V\ \} \qquad [\ where:\ e_k\ =\ (v_i,\ v_j)\ ]$$

(**Note:** $\mathrm{card}(X)$ is the number of elements in a set X.)

Formal Definitions of Example Graphs

$$
\begin{aligned}
G1: \quad V &= \{ A \} \\
E &= \{ \}
\end{aligned}
$$

$$
\begin{aligned}
G2: \quad V &= \{ 1, 2, 3, 4 \} \\
E &= \{ (1, 2), (2, 1), (2, 3), (3, 2), (2, 4), (4, 2), (3, 4), (4, 3) \}
\end{aligned}
$$

$$
\begin{aligned}
G3: \quad V &= \{ OxfordCircus, GreenPark, PiccadillyCircus \} \\
E &= \{ (OxfordCircus, GreenPark), (GreenPark, OxfordCircus), \\
&\qquad \ldots, (PiccadillyCircus, GreenPark) \}
\end{aligned}
$$

$$
\begin{aligned}
G4: \quad V &= \{ A, B, C, D \} \\
E &= \{ (A, B), (D, C), (D, D) \}
\end{aligned}
$$

For *G5* as well as the *vertices* (V) & *edges* (E) it also includes a mapping (W) from each edge to its *weight*.

$$
\begin{aligned}
G5: \quad V &= \{ A, B, C, D \} \\
E &= \{ (A, B), (B, C), (B, D), (C, D) \} \\
W &= \{ (A, B) \mapsto 1, (B, C) \mapsto 2, (B, D) \mapsto 8, (C, D) \mapsto 7 \}
\end{aligned}
$$

Undirected Graphs

The formal definition of an **undirected graph** is:

Definition: Undirected Graph

With an **undirected graph no direction is given to its edges**.

So for any edge $e_k \in E$ that connects the pair of vertices $v_i, v_j \in V$, then in effect the *ordered pair* $(v_i,\ v_j)$ and the *ordered pair* $(v_j,\ v_i)$ both represent the same edge e_k:

$$e_k \ = \ (v_i,\ v_j) \ = \ (v_j,\ v_i).$$

and E can be *redefined* as follows[3]:

$$E \ \subseteq \ \{\ \{v_i,\ v_j\} \mid v_i,\ v_j \in V\ \}$$

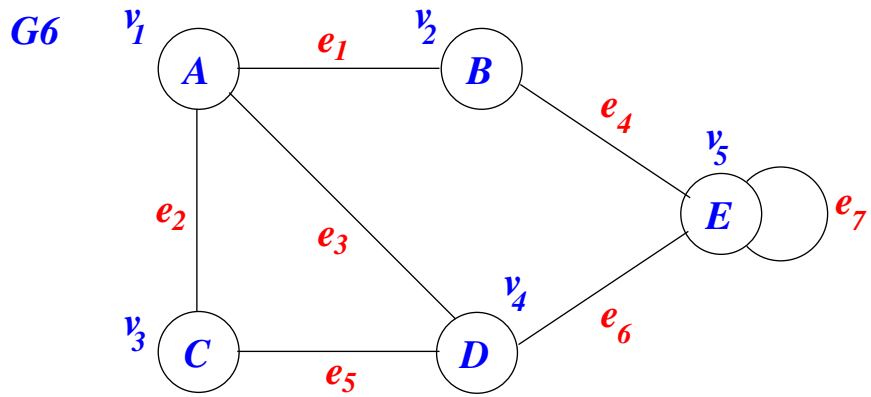
Using *sets* "$\{-,-\}$" instead of *ordered pairs* "$(-,-)$" means the order of the vertices does not matter as **there is no direction involved**, e.g. *G2*'s edges:

$$E \ = \ \{\ \{1,\ 2\},\ \{2,\ 3\},\ \{2,\ 4\},\ \{3,\ 4\}\ \}$$

[3] Since for **sets** $\{\ a,\ b\ \} = \{\ b,\ a\ \}$, whereas for **ordered pairs** in general $(a,\ b) \neq (b,\ a)$, & there are **no loops**.

Undirected Graphs

Below is *G6* an *undirected* graph.



Where its component *vertices* and *complete set of edges* are as follows:

$$V = \{\ A,\ B,\ C,\ D,\ E\ \}$$
$$v_1 = A,\ v_2 = B,\ v_3 = C,\ v_4 = D,\ v_5 = E$$

$$E = \{\ (A,B),(B,A),\ (A,C),(C,A),\ (A,D),(D,A),$$
$$(B,E),(E,B),\ (C,D),(D,C),\ (D,E),(E,D),\ (E,E)\ \}$$

$$e_1 = (A,B) = (B,A), \qquad e_2 = (A,C) = (C,A), \qquad e_3 = (A,D) = (D,A),$$
$$e_4 = (B,E) = (E,B), \qquad e_5 = (C,D) = (D,C), \qquad e_6 = (D,E) = (E,D),$$
$$e_7 = (E,E)$$

Directed Graph

The formal definition of a **directed graph** is:

Definition: Directed Graph

With a **directed graph** (or **digraph**) **directions are given to all its edges**.

So for any edge e_k that connects the pair of vertices v_i and v_j, then the pair (v_i, v_j) represents the edge e_k from vertex v_i to vertex v_j.

Where v_i is the *source vertex*, and v_j is the *destination vertex*.

Note that the edge e_l with pair $e_l = (v_j, v_i)$ from the *source vertex* v_j to the *destination vertex* v_i, **represents a different edge** to e_k.

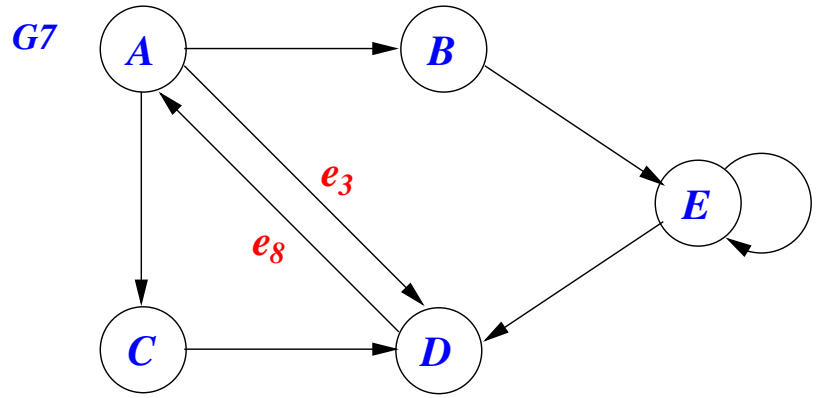
That is, the two pairs (v_i, v_j) and (v_j, v_i) **do not represent the same edge**, but two different edges, i.e. $e_k \neq e_l$, that is:

$$e_k = (v_i, v_j) \neq (v_j, v_i) = e_l$$

With directed graphs (v_i, v_j) is **not the same as** (v_j, v_i) since, even though the edges connect the same pair of vertices, **their directions are different**.

Directed Graphs

G7 is a *directed* graph, (similar to *G6*) where now all the edges have been assigned a **direction** as represented by the arrows.



Note: that *G6*'s *undirected* edge e_3 has been separated into two edges going in opposite directions between A and D, e.g. e_3 and e_8, that are *different edges* since they *go in different directions* between A and D:

$$e_3 = (A, D) \neq (D, A) = e_8$$

This graph has the same *vertices* V as *G6*, but has different *edges*:

$$E = \{ (A,B), (A,C), (A,D), (B,E), (C,D), (D,A), (E,D), (E,E) \}$$

The result is that some vertices are now **no longer adjacent** to each other in a particular direction as there is no edge connecting them, e.g. no direct route from C to A, but only via $C \rightarrow D \rightarrow A$.

Weighted Graph

The formal definition of a **weighted graph** is:

Definition: Weighted Graph
With a **weighted graph** each *edge* has a number, called a "**weight**" (or "**cost**") assigned to it.

A **weighted graph** can be either a *directed* or an *undirected* graph.

So all the edges $e_k \in E$, i.e. $e_k = (v_i,\ v_j)$, have a weight w_k assigned to them using a $weight$ function, e.g. $weight(e_k) = w_k$, defined by:

$$weight\ =\ \{\ e_1 \mapsto w_1,\ e_2 \mapsto w_2,\ e_3 \mapsto w_3,\ \ldots\ \}$$

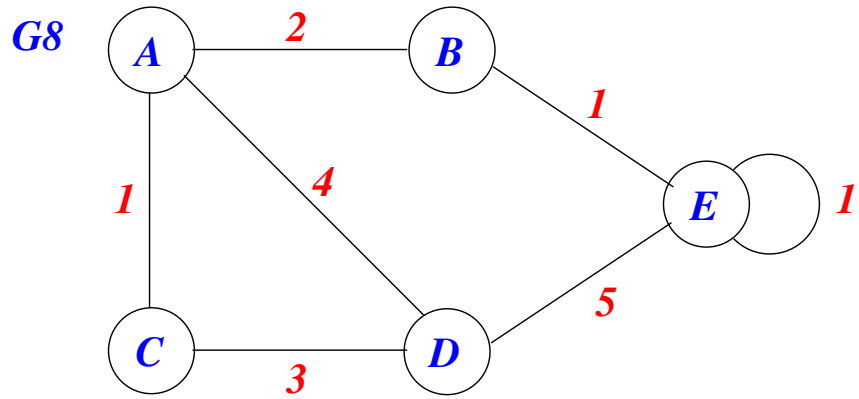
Note: that since in a *directed* graph there can be more than one edge between any two vertices, that each edge will have its own weight attached to it, even though they connect the same vertices.

E.g. think of how much *energy* ("weight/cost") it takes to ride a bike up a hill compared to riding it down the same hill.

The *weight* can represent any type of attribute or value that one wants to assign to an edge, e.g. distances between two locations in kilometres or time; capacity of a route through a system, e.g. network capacity, number of people in a lift or on a tube; etc.

Weighted Undirected Graphs

G8 is a *weighted* undirected graph:



Where its *vertices* & *edges* are as follows:

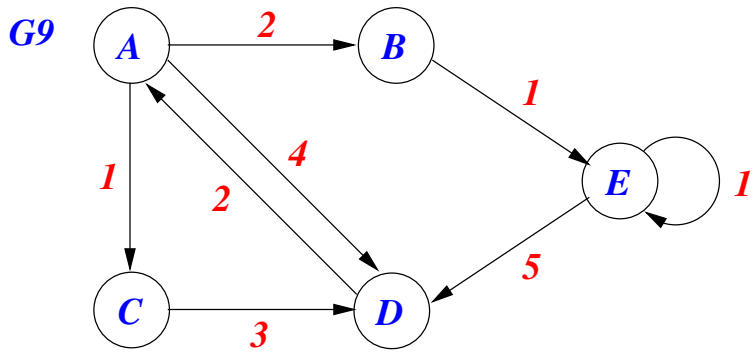
$$V = \{\ A,\ B,\ C,\ D,\ E\ \}$$
$$E = \{\ (A,B),(B,A),\ (A,C),(C,A),\ (A,D),(D,A),$$
$$(B,E),(E,B),\ (C,D),(D,C),\ (D,E),(E,D),\ (E,E)\ \}$$

Its *edge weights* are defined using $weight$ as follows:

$$weight(\ (A,\ B)\ ) = weight(\ (B,\ A)\ ) = 2, \quad weight(\ (A,\ C)\ ) = weight(\ (C,\ A)\ ) = 1,$$
$$weight(\ (A,\ D)\ ) = weight(\ (D,\ A)\ ) = 4, \quad weight(\ (B,\ E)\ ) = weight(\ (E,\ B)\ ) = 1,$$
$$weight(\ (C,\ D)\ ) = weight(\ (D,\ C)\ ) = 3, \quad weight(\ (D,\ E)\ ) = weight(\ (E,\ D)\ ) = 5,$$
$$weight(\ (E,\ E)\ ) = 1$$

Weighted Directed Graphs

G9 is a *weighted* directed graph.



Where *G9*'s *vertices* & *edges* are as follows:

$$V = \{\ A,\ B,\ C,\ D,\ E\ \}$$

$$E = \{\ (A,B),\ (A,C),\ (A,D),\ (B,E),\ (C,D),\ (D,A),\ (E,D),\ (E,E)\ \}$$

With the *weights* for *G9*'s directed edges are as follows:

$$weight(\ (A,B)\ ) = 2, \quad weight(\ (A,C)\ ) = 1, \quad weight(\ (A,D)\ ) = 4,$$
$$weight(\ (B,E)\ ) = 1, \quad weight(\ (C,D)\ ) = 3, \quad weight(\ (D,A)\ ) = 2,$$
$$weight(\ (E,D)\ ) = 5, \quad weight(\ (E,E)\ ) = 1$$

$$weight\ =\ \{\ (A,\ B) \mapsto 2,\ (A,\ C) \mapsto 1,\ (A,\ D) \mapsto 4,\ (B,\ E) \mapsto 1,$$
$$(C,\ D) \mapsto 3,\ (D,\ A) \mapsto 2,\ (E,\ D) \mapsto 5,\ (E,\ E) \mapsto 1\ \}$$

The *edge weights* for *G8* & *G9* are similar except that *G8*'s A & D edge is split into two edges (A, D) & (D, A) in *G9*, each with a distinct weight.

Formal Definitions of Graph Properties

Definition: Degree

The **degree of a vertex**, is the number of edges connected to it.

The **degree of a graph**, is the maximum degree of any vertex.

In a *directed* graph the **in-degree of a vertex**, is the number of edges *coming into a vertex*, for which it is the *destination* vertex.

In a *directed* graph the **out-degree of a vertex**, is the number of edges *going out of a vertex*.

Examples:

For a vertex, in *G1* the degree(A) = 0, in *G2* the degree(2) = 3.

For a graph, degree(G1) = 0, degree(G2) = 3, degree(G3) = 4, degree(G4) = 3, degree(G5) = 3.

For *directed* graphs vertices in:
G3: in-degree(OxfordCircus) = 2, out-degree(OxfordCircus) = 2,
G4: in-degree(A) = 0, out-degree(D) = 2,
G5: in-degree(D) = 2, out-degree(A) = 1.

Paths & Cycles in a Graph

Definition: Paths

A **path** in a graph is a sequence of vertices connected by edges.

A **simple path** is a path with no repeated vertices.

Examples: In *G9*: $A \to C \to D \to A \to D$ is a path.

And $A \to B \to E \to D$ is a *simple path*.

Definition: Cycles

A **cycle** is a path whose first and last vertices are the same.

A **simple cycle** is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

The **length** of a *path* or a *cycle* is the number of edges it contains.

Examples: In *G9*: $A \to D \to A \to D \to A$ is a *cycle*.

And $E \to D \to A \to B \to E$ is a *simple cycle*.

The length of the above paths and cycles are 4, 3, 4, 4.

Connected Graphs

Definition: Connected

A graph is **connected** if there is a path from every vertex to every other vertex in the graph. (*Note: edge direction is ignored.*)

A graph that is **not connected** consists of a set of connected components, which are maximal connected sub-graphs.

In terms of vertices, one vertex is *connected* to another vertex if there *exists a path that contains both of them*.

A *directed* graph is **strongly connected** if there is a path from every vertex to every other vertex in the graph, respecting the direction of the edges.

Examples: *G1*, *G2*, *G3* and *G5* are **connected** graphs.

G4 is a **disconnected** graph.

G3 and *G7* are *directed* graphs and are **strongly connected**.

PART III

Representing Graphs

Ways of Representing Graphs

From simply looking at a graph it strongly suggests that the most obvious way to represent it would be to use a "*linked*" type data structure.

Something based on a generalisation of a *linked list* or an *n-ary tree*?

For example, *vertices* represented by nodes & *edges* represented by nodes having a link to each of its adjacent vertices.

However, this is very rarely if ever done in practice.

This is because the vast majority of operations & algorithms applied to graphs involve traversing the graph, i.e. moving around it from vertex to vertex.

But, having to do this by following the *links* between the vertices is usually very inefficient, especially for large graphs.

So more efficient alternative ways of representing a graph are use.

The two most common alternative data structures used to do this are:

- an *adjacency matrix* – a 2-dimensional array, or
- an *adjacency list* – a list of lists (or an array of lists).

Adjacency Matrix Representation of a Graph (1/2)

An **adjacency matrix** (or **connection matrix**) represents a graph *G* by representing the presence or absence of *edges between its vertices*.

In an *adjacency matrix AM(G)* the *rows* and *columns* represent the two **end vertices** (or **end points**) of the *edges*, i.e. the *source* and *destination* vertices of an *edge*.

So if *G* has $\mathrm{card}(V) = m$ vertices then *AM(G)* will be an $m \times m$ matrix, that is implemented by a 2-dimensional array AM[m][m].

Each element AM[r][c] records the presence or absence of an *edge* between the two vertices v_r and v_c.

So can just use **0** and **1** or Boolean, as follows:

- AM[r][c] = **1** (or TRUE)
  Then there **is an edge** between v_r and v_c, i.e. $(v_r,\ v_c) \in E$.

- AM[r][c] = **0** (or FALSE)
  Then there is **no edge** between v_r and v_c, i.e. $(v_r,\ v_c) \notin E$.

Adjacency Matrix Representation of a Graph (2/2)

In an adjacency matrix for a *weighted* graph the *edge's weight* is used instead of 0 & 1 or Booleans to indicate the presence or absence of an edge.

And if an edge is not present then use a value that is **never used as a weight**, e.g. *-1*, *infinity* ∞ or some other "*safe*" value.

- AM[r][c] = $weight((v_r,\ v_c))$

  There is an edge between v_r and v_c, with $weight((v_r,\ v_c))$.

- AM[r][c] = -1

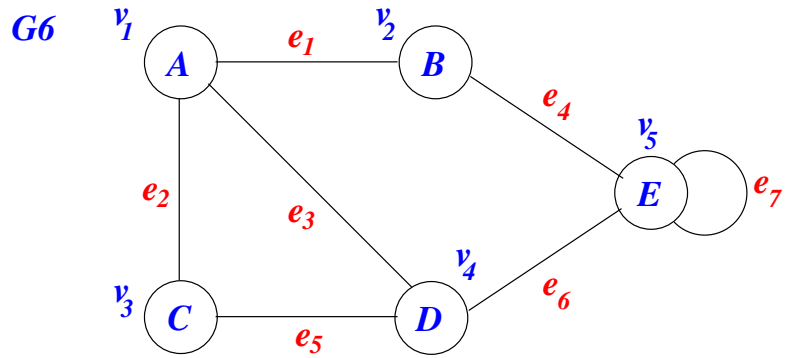
  Then there is **no edge** between v_r and v_c.

Note: the **big disadvantage of using an adjacency matrix** is that if the graph has a very large number N of vertices, e.g. in the millions, then the storage space needed for the values is $O(N^2)$, is seen as prohibitive.

This becomes even more the case if the number of edges in the graph is very low in relation to the number of vertices, this type of graph is known as a **sparse graph**.

Adjacency Matrix for an Undirected Graph

Note that the Adjacency Matrix for *G6* an *undirected* graph is **symmetric**.



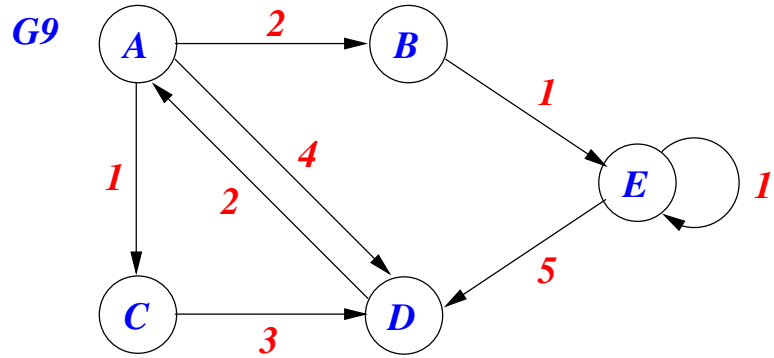
$$E = \{\ (A,B),(B,A),\ (A,C),(C,A),\ (A,D),(D,A),\ (B,E),(E,B),\ (C,D),(D,C),\ (D,E),(E,D),\ (E,E)\ \}$$

$AM(G6)$

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	1
C	1	0	0	1	0
D	1	0	1	0	1
E	0	1	0	1	1

Adjacency Matrix for a Directed Weighted Graph

Note that the Adjacency Matrix for *G9* a *weighted directed* graph is **not symmetric**.



$AM(G9)$

	A	B	C	D	E
A	-1	2	1	4	-1
B	-1	-1	-1	-1	1
C	-1	-1	-1	3	-1
D	2	-1	-1	-1	-1
E	-1	-1	-1	5	1

NOTE: *G9*'s edge weights, e.g. $weight((A,\ B)) = 2$, are used to indicate the presence of an edge, & -1 if an edge is not present.

Adjacency List Representation of a Graph

The disadvantage of an *adjacency matrix* is that it always requires $O(N^2)$ space, irrespective of the number of edges.

An alternative that does not have this problem is the *Adjacency List*.

Because it **only stores the edges**, and therefore it uses much less memory space when the corresponding adjacency matrix is **sparse**.

If the graph is not sparse, i.e. **dense**, then the adjacency matrix is probably a better choice.

In an adjacency list each vertex v is a *header* (i.e. points to the head) of a *linked list of nodes*.

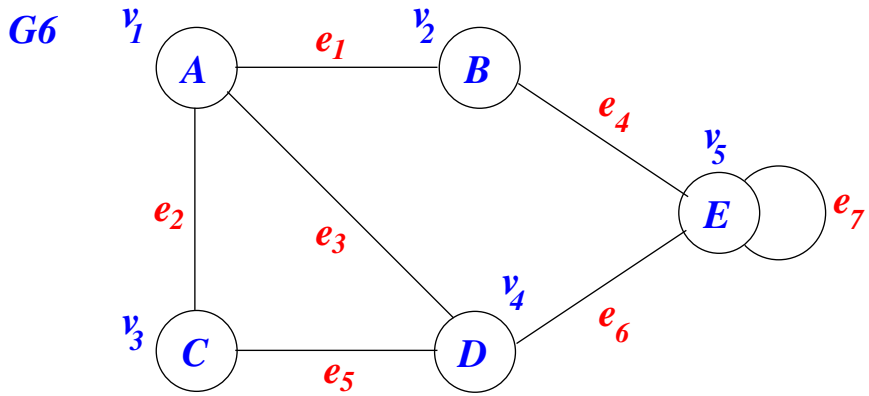
Where a *node* stores the *destination vertices* for the edges leaving vertex v.

For a *weighted graph* the nodes will also contain the *weight* assigned to that edge.

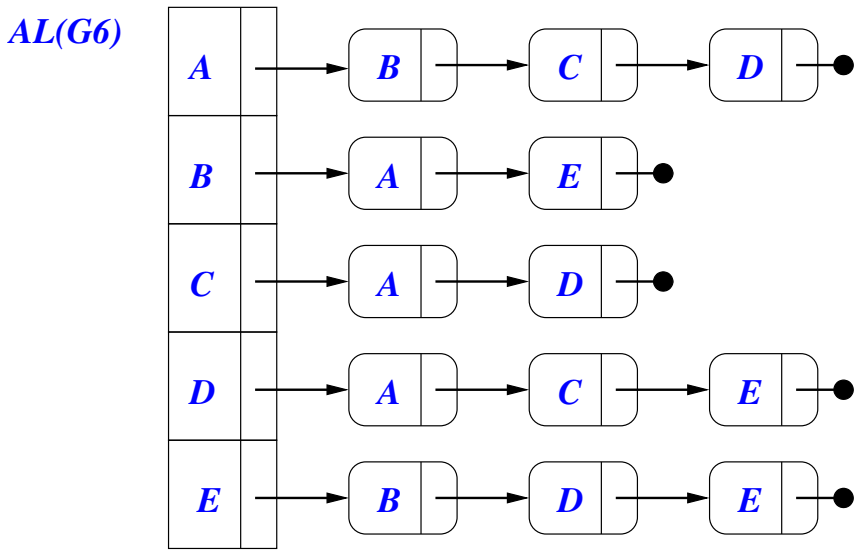
Note: In some implementations the node will contain all of the edge information, i.e. *[source vertex, destination vertex, weight]*.

The *headers* can be stored either in an *array* or a *list*, giving a list of lists.

Adjacency List for Undirected Graph *G6*

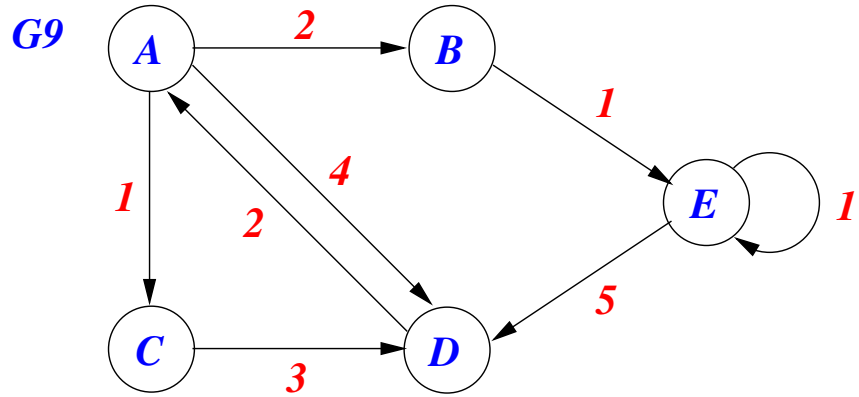


Adjacency list representation of *G6*, using an *array* for the vertices headers.

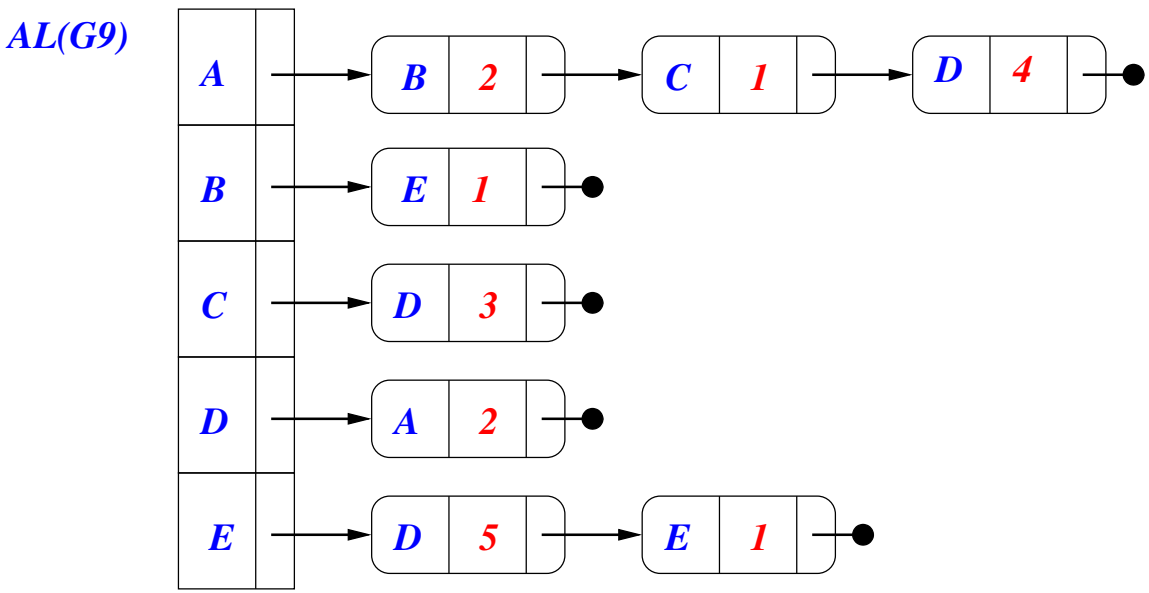


Since *G6* is undirected, each edge "(*A*, *B*)" is added to *A*'s & *B*'s lists.

Adjacency List for Weighted Directed Graph *G9*



Adjacency list representation of *G9*, using an *array* for the vertices headers, including the edge weights.



APPENDIX A

Glossary: Graph Terminology

Glossary: Graph Terminology 1

Taken from:
`http://www.btechsmartclass.com/data_structures/introduction-to-graphs.html`

Vertex:	Individual data element of a graph is called as *vertex*. A vertex is also known as *node*.
Edge:	An *edge* is a *connecting link between two vertices*. An edge is also known as *arc*. An edge is represented as *(startingVertex, endingVertex)*. There are several *edges types*.
Undirected Edge:	An *undirected edge* is a *bidirectional edge* type. **If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A)**.
Directed Edge:	A *directed edge* is a *unidirectional edge*. **If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A)**.
Weighted Edge:	A *weighted edge* is an edge with a *value (or cost)* on it.
Undirected Graph:	A *graph with only undirected edges* is said to be an *undirected graph*.
Directed Graph:	A *graph with only directed edges* is said to be *directed graph*.
Mixed Graph:	A *graph with both undirected and directed edges* is said to be a *mixed graph*.

Glossary: Graph Terminology 2

End Vertices: (or Endpoints)	The two vertices joined by an *edge* are the *end vertices (or endpoints) of that edge*.
Source: (or Origin)	If an *edge is directed* then its *first endpoint is its source (or origin)*.
Destination: (or Target)	If an *edge is directed* then its *second endpoint is its destination (or target)*.
Adjacent:	If there is an *edge* between *vertices* A and B then *both A and B are said to be adjacent*.
Incident:	An *edge* is an *incident on a vertex* if the *vertex* is one of the *endpoints of that edge*.
Outgoing Edge:	A *directed edge* is said to be an *outgoing edge* on its origin vertex.
Incoming Edge:	A *directed edge* is said to be an *incoming edge* on its destination vertex.

Glossary: Graph Terminology 3

Degree:	The *total number of edges connected to a vertex* is the *degree of that vertex*.
In-degree:	The *total number of incoming edges* connected to a vertex is the *in-degree of that vertex*.
Out-degree:	The *total number of outgoing edges* connected to a vertex is said to be out-degree of that vertex.
Parallel edges: (or Multiple edges)	If there are *two undirected edges* with the *same end vertices*; or there are *two directed edges* with the *same origin and destination*, such edges are called *parallel edges or multiple edges*.
Self-loop:	An *edge* (undirected or directed) is a *self-loop if its two endpoints coincide with each other*.
Simple Graph:	A *graph* is *simple* if there are *no parallel and self-loop edges*.
Path:	A *path* is a *sequence of alternate vertices and edges that starts at a vertex and ends at another vertex* such that each *edge is incident to its predecessor and successor vertex*.