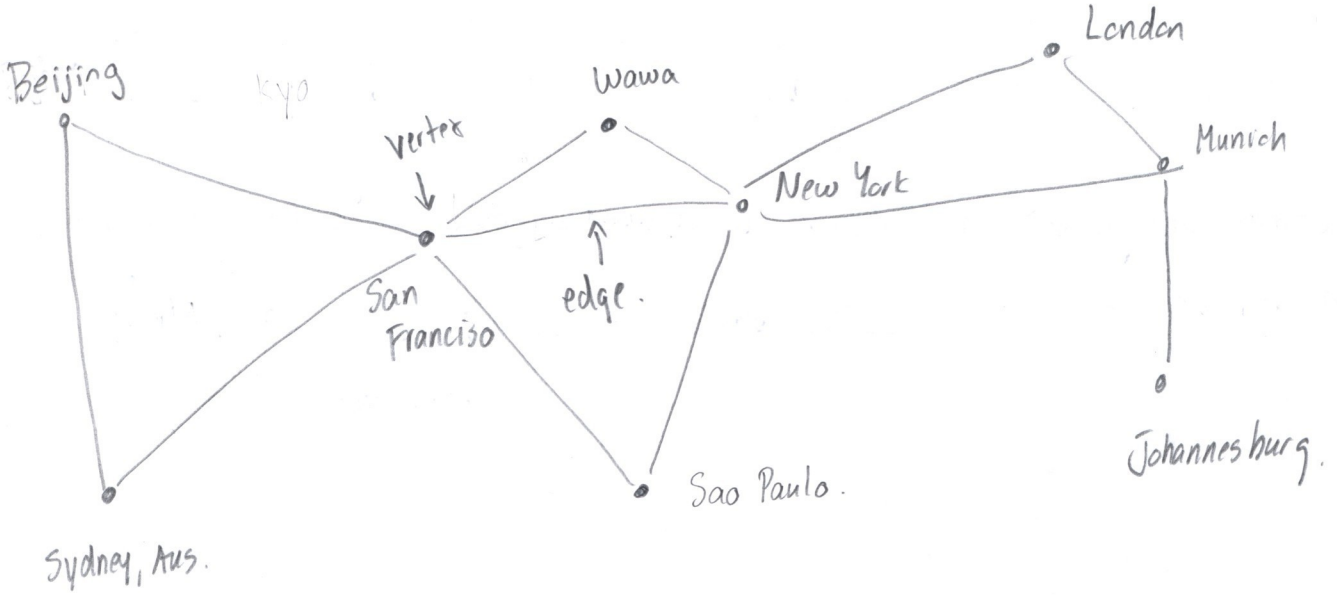
## Graphs

Graphs can be used to model problems from virtually any field. We have already used them in describing relations.

A graph is a pair $(V, E)$ where $V$ is a set called the vertex set and $E$ is a set called the edge set. Each edge describes a 'connection' between vertices of $V$.

For example let $V$ be a set of major world cities and let the edges in $E$ represent connections (direct flights between them).




This particular graph is a simple graph - no loops (connections between a vertex and itself.)

- only one edge connecting any two vertices.

The graph is also undirected since edges do not have direction.

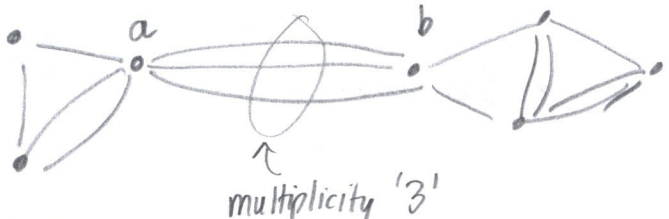
One way to represent such a graph is to represent each edge as a set (order doesn't matter) of size 2.

Eg:

$V = \{$ Beijng, Sydney, San Franciso, Wawa, New York, São Paulo, London, Munich, Johannesburg $\}$.

$E = \{$ $\{$Beijing, Sydney$\}$, $\{$Beijng, San Francisco$\}$, $\{$Sydney, San Francisco$\}$, $\{$San Francisc, Wawa$\}$ $\{$San Francisco, New York$\}$ $\{$San Francisco, Sao Paulo$\}$ $\{$Wawa, New York$\}$ $\{$New York, London$\}$ $\{$New York, Munich$\}$ $\{$Munich, Johannesburg$\}$ $\}$.

Sometimes we want to allow multiple edges between vertices. Such graphs are called <u>multigraphs</u>.

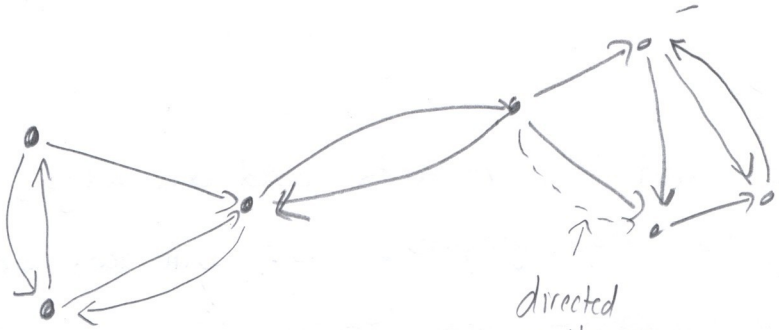



Edges are still undirected, so we can represent them as sets, however we can now have multiple instances of sets. If for a pair of vertices, say $a$, $b$ there are $m$ edges, then $\{a,b\}$ has multiplicity $m$.

$m=3$ in the diagram above.

Sometimes we may want to relax the restriction that there are no edges between a vertex and itself. Such edges are called <u>loops</u> (we have seen these in graphical depictions of reflexive relations), and the resulting graph is called a <u>pseudograph</u>. (Ex: loopbacks in a network)

We can also have <u>directed</u> versions of these graphs, where edges only go in one direction:



directed multigraph if we allow duplicates

In a directed graph we represent edges by ordered pairs $(a,b)$ instead of a set $\{a,b\}$.

<u>To Summarize</u>

| Graph Type | Edges | Multiple Edges Allowed? | Loops Allowed |
|---|---|---|---|
| a) Simple | Undirected | No | No |
| b) Multigraph | " | Yes | No |
| c) Pseudograph | " | Yes | Yes |
| d) Simple directed | Directed | No | No |
| e) Directed Multigraph | " | Yes | Yes |
| f) Mixed | Both Directed / Undirected | Yes | Yes |

## Uses of Graphs

Social Networks: (aka Acquaintanceship Graphs)

Hollywood Graphs: which actors/actresses have played rolls in the same movie.

Collaboration Graphs: like Hollywood G., among researchers.

Call Graph: Calls made on a network.

Web Graph: Link structure of the web.

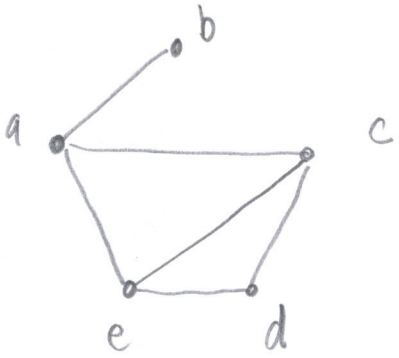
## Representing Graphs

One way is to list all vertices and all edges. This is time consuming for large graphs. We will look at two ways we can represent graphs - these 'techniques' are used in computing to represent graphs as well.

## Adjacency Lists:

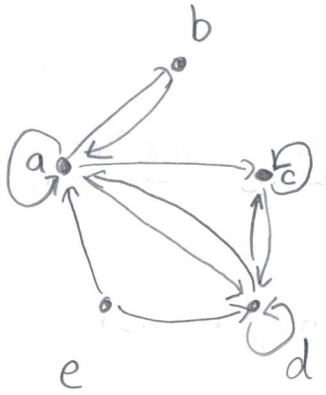
For each vertex, list the vertices that are 'adjacent' (i.e. connected by an edge)

This representation may be used for both directed, and undirected, graphs, and with loops.




Adjacency List

| Vertex | adjacent verts |
|---|---|
| a | b, c, e |
| b | a |
| c | a, d, e |
| d | c, e |
| e | a, c, d. |



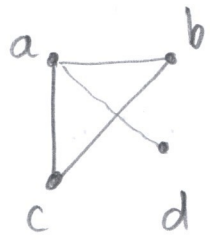

| Vertex | adjacent vertices |
|---|---|
| a | a, b, c, d |
| b | a |
| c | c, d |
| d | a, c, d |
| e | a, d |

## Matrix Representation / Adjacency Matrix:

One row and column for every vertex, $v_1, v_2, \ldots v_n$.

Row $i$ column $j$ is 1 if the edge $\{i, j\}$ is in $E$, otherwise the entry has a value of 0.

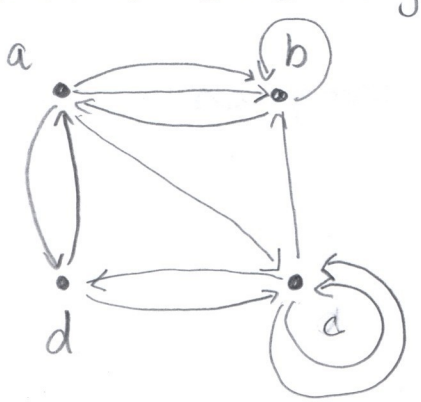
In order for this to work you must have some fixed ordering on the vertices.




$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

For simple, undirected, graphs the adjacency matrix is symmetric ($a_{ij} = a_{ji}$) and the main diagonal is all 0's ($a_{jj} = 0$) since loops are not permitted.

In general we call allow for multigraphs by putting the multiplicities as entries instead of just a 0 or 1.




$$\begin{bmatrix} 0 & 2 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

← directed graph, matrix is not symmetric.

When to use which representation?

Some 'operations' are faster for adjacency matrices some for adjacency lists.

i.e.

Are vertices $u, v$ adjacent — $O(1)$ time in matrix; $O(\deg(v))$ in adjacency list.

To list the vertices adjacent to $u$ requires — $O(n)$ time in matrix; $O(\deg(u))$ in adjacency list.

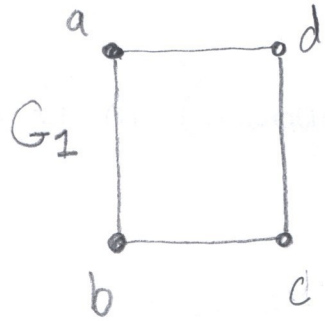
## Graph Isomorphism (Cover after graph terminology)

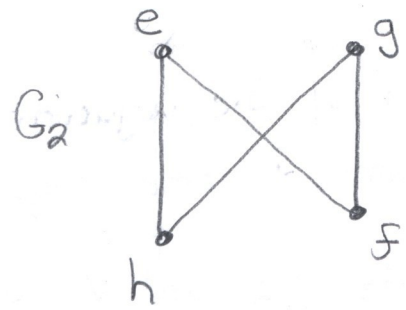
We often want to know if it is possible to draw two graphs in the same way. If it is possible to do so we say the graphs are isomorphic.

Formally we define isomorphism as:

Defn: The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there is a one-to-one and onto function $f$ from $V_1$ to $V_2$ with the property that $a$ and $b$ are adjacent in $G_1$ if and only if $f(a)$ and $f(b)$ are adjacent in $G_2$ for all $a$ and $b$ in $V_1$. Such a function is called an isomorphism.

But it is a bit easier to understand by seeing an example.




$f(a) = e \quad f(b) = f \quad f(c) = g \quad f(d) = h$

The function is a bijection which preserves adjacency in the new graph.

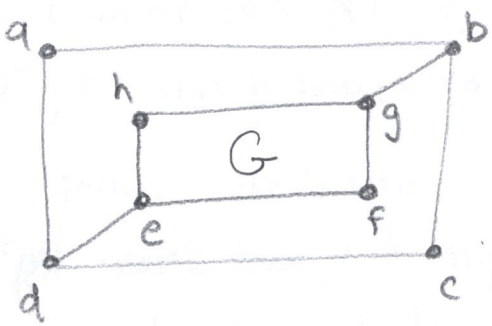
It can be difficult to determine if simple graphs are isomorphic, as there are $n!$ one-to-one correspondences between vertex sets of two simple graphs - so testing all possibilities is time consuming.

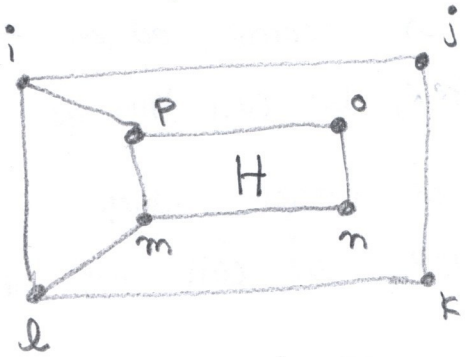
We can sometimes use properties that are preserved during the transformation - such properties are called <u>graph invariants</u>.

- # of vertices.
- # of edges.
- # of vertices of each degree.

These still are not always enough, but they can help.

Eg: Are the two graphs G and H isomorphic.



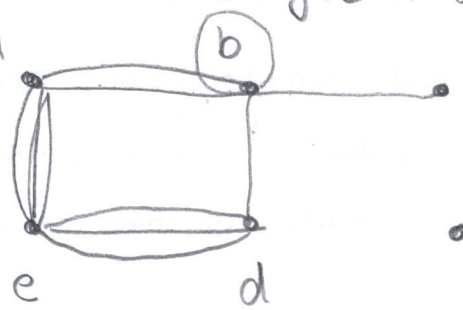

No. For example b has degree 3 and is adjacent to 1 degree 3 vertex g and 2 degree 2 vertices (a & c). All degree 3 vertices in H have only a single degree 2 vertex among thier neighbours.

## Graph Terminology

Two vertices $u$ and $v$ are adjacent (neighbours) in $G$ if $u$ and $v$ are endpoints of some edge of $G$.

If edge $e$ connects $u$ and $v$ we say $e$ is incident on $u$ (and $v$) or incident with $u$ and $v$.

The degree of a vertex in an undirected graph is the number of edges incident with it (loops are counted twice). We denote the degree of a vertex $u$ by $\deg(u)$.




Eg: $\deg(a) = 5$
$\deg(b) = 6$
$\deg(f) = 0$.

### The Handshaking Theorem

$$\sum_{v \in V} \deg(v) = 2|E|$$

For directed graphs for edge $(u,v)$ $u$ is the initial vertex and $v$ is the terminal vertex.

The # of incoming edges = # of edges with $v$ as terminal vertex, we call this the in-degree of $v$, and denote $\deg^-(v)$

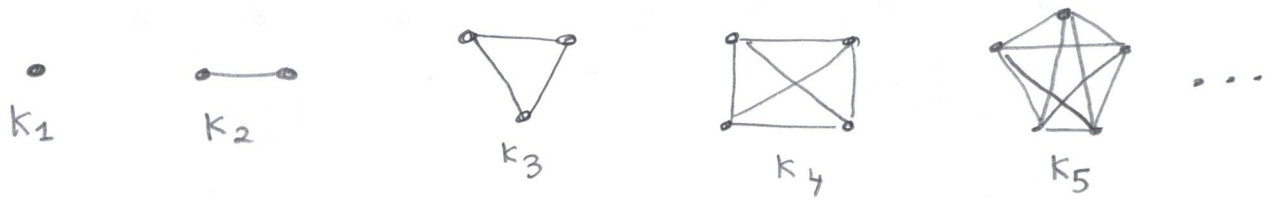
The # of outgoing edges = # of edges with $v$ as initial vertex, we call this the out-degree of $v$, and denote $\deg^+(v)$

Note that: $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$

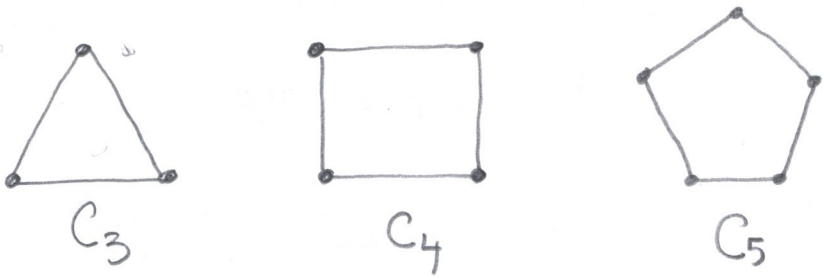
## Special Graphs

Some graphs appear frequently, or have special properties and thus have been named.

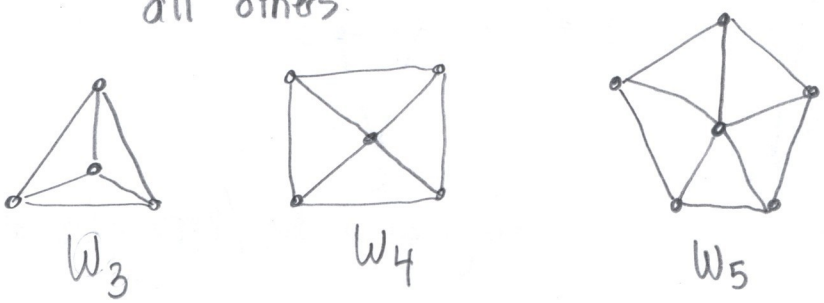
The complete graph: $K_n$ on $n$ vertices, each connected to all others.




The cycle: $C_n$ $n > 3$, $V_1 V_2, \ldots V_n$ with edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots \{v_n, v_1\}$.




The wheel $W_n$: take $C_n$ and add a vertex connected to all others.

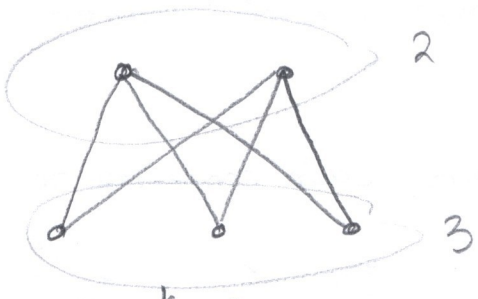



## Bipartite Graphs

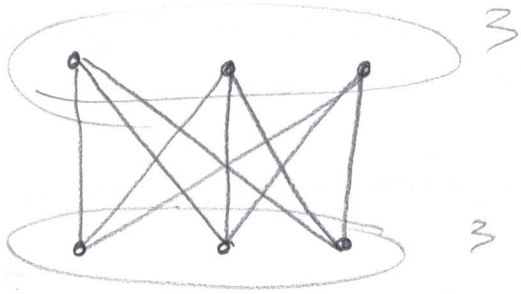
In a bipartite graph we can partition the vertex set $V$ into two subsets $V_1$ and $V_2$ such that all edges have one endpoint in $V_1$ and one endpoint in $V_2$.

- one way to do this is to assign one of two colows to every vertex such that no adjacent vertices have the same colour.

The complete bipartite graph: $K_{m,n}$




$K_{2,3}$



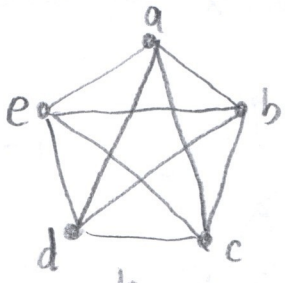

$K_{3,3}$

## Subgraphs

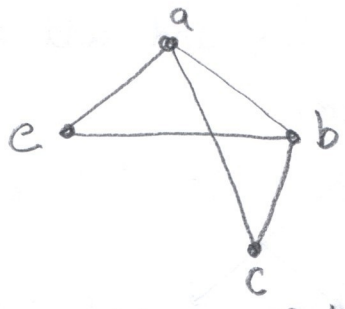
A <u>subgraph</u> of a graph $G=(V,E)$ is a graph $H=(W,F)$ such that $W \subseteq V$ and $F \subseteq E$.

A <u>subgraph</u> $H$ of $G$ is a <u>proper</u> subgraph of $G$ if $H \neq G$.

Ex:




$K_5$




Subgraph of $K_5$ / Proper subgraph of $K_5$.

A subgraph is spanning if it contains <u>all</u> vertices of the original graph.

## Connectivity

Sometimes we want to know if two vertices in a graph are connected by a sequence of edges that might visit other vertices on the way (eg. can 2 computers on a network talk).

Recall that a path is a sequence of edges that begins at a vertex and travels from vertex to vertex along the edges of the graph.

More formally, a path of length $n > 0$ from vertex $u$ to vertex $v$ in $G$ is a sequence of $n$ edges $e_1, e_2, \ldots e_n$ in $G$ such that $e_1 = \{x_0, x_1\}$, $e_2 = \{x_1, x_2\}$, $\ldots$ $e_n = \{x_{n-1}, x_n\}$. where $x_0 = u$ and $x_n = v$.

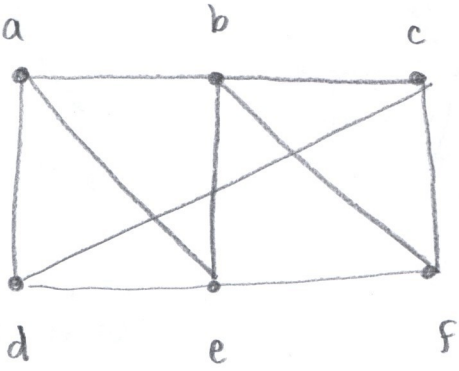
If the graph is simple we can just use the vertex sequence to label the path.

The path is a circuit (cycle) if $u = v$.

The path passes through $x_1, x_2, x_3, \ldots x_{n-1}$ and traverses edges $e_1, e_2, \ldots e_n$

A path is simple if it does not traverse an edge more than once. (same holds for circuits)

Ex:




a, d, c, f, e is a simple path of length 4.

d, e, c, a is NOT a path $\{e, c\} \notin E$

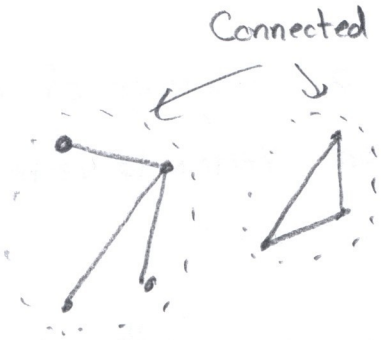
b, c, f, e, b is a simple circuit of length 4.

a, b, e, d, a, b is a non-simple path of length 5.

An undirected graph is connected if there is a path between every two distinct vertices in the graph.

Ex:

Connected




Not connected

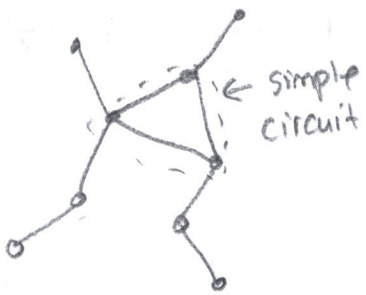
The different parts that are (maximally) connected are called connected components.

## Trees

One special type of connected (subgraph is called a tree.

A tree is a connected undirected graph with no simple circuits.

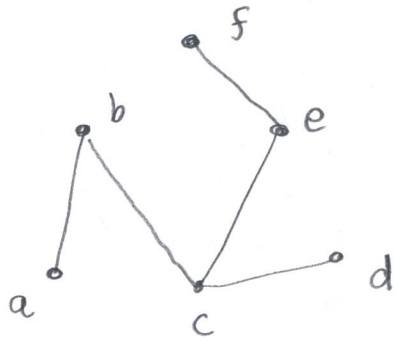
Tree



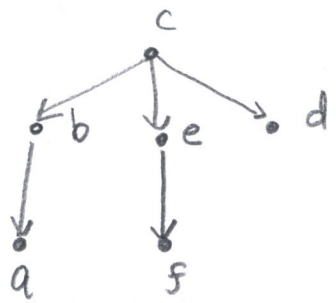

Not a tree (has circuit)

Not a tree (disconnected)
This is a Forest.

A rooted tree is a tree with one vertex specifically designated as the root and every edge directed away from the root.

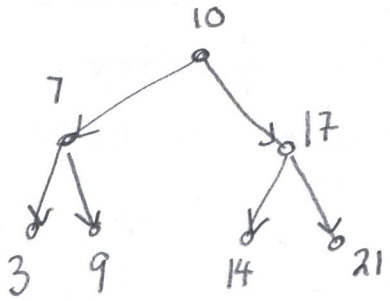



A tree




Tree rooted at "c"

Many data structures in computer science use some form of tree as the base for the structure. (Eg. Binary Search Tree).

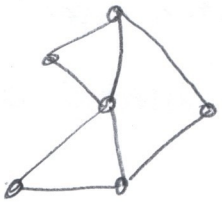



Spanning Trees

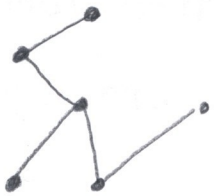
Because trees are "simple" in structure we often want to find a subgraph of a graph that forms a tree.

- we call such a tree a spanning subgraph.
  - spanning subgraph – because all vertices are present.
  - tree – because it is connected & no circuits

G



Spanning Subgraph (Tree) of G.



## Graph Traversals

### Depth-First Search

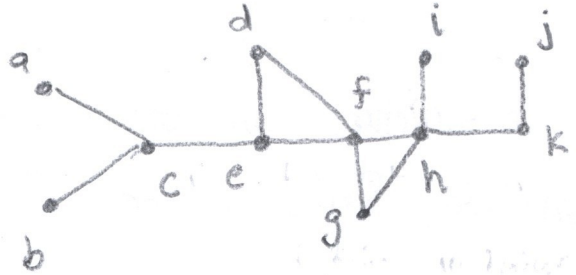
Suppose you are trying to explore a graph ie. to see what computers are connected to a network (find the connected components.

How do you do this in an orderly way, so that you don't end up getting lost (looping forever).

Idea: mark all vertices as <u>unvisited</u>.

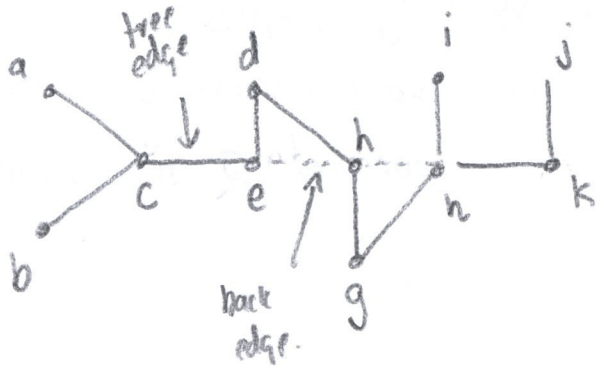
- start at an arbitrary vertex.
- go to an unvisited neighbour.
- repeat until you have no unvisited neighbours
- return to the vertex from which you were visited.

<u>Ex:</u>




Start at say f

- f, g, h, k, j
- backtrack to k
- backtrack to h
- i
- backtrack to g
- backtrack to f
- d, e, c, a
- backtrack to c
- b
- backtrack to c, e, d, f DONE.



This produces a spanning tree: all vertices are visited + if we retain just the edges we backtrack on (tree edges) we have a spanning tree.

- No cycles - this would mean revisiting a visited node.

Depth First Search (DFS) has <u>many</u> applications.

- find paths, circuits
- find connected components
- etc.

## Breadth First Search

Instead of always going as deep as possible, we try to explore gradually at specific distances from the starting vertex.

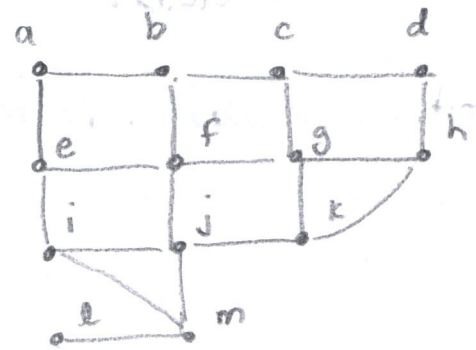
Keep a list of all vertices we've seen

Start at selected vertex (root).

Add all adjacent vertices to the list.

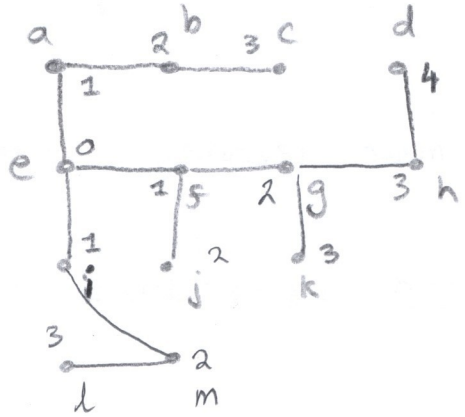
Take first vertex off the list, explore it } repeat until list is empty.
- add its vertices to the end of the list, if the have not yet been visited or added to the list.

Ex:




start w 'e' e=0
add {a, f, i} = 1
add {b, g, j, m} = 2
add {c, h, k, l} = 3
add {d} = 4

In this case we add edges from which each vertex was explored:




Again we get a spanning tree (rooted at e)

This also gives the shortest path from the start vertex 'e' to any vertex when we measure path length by # of edges visited.

## Exam Review

1. The exam will cover the entire course, but is weighted slightly to material appearing after the mid-term.
2. There are questions from every lecture.
3. You are responsible for <u>all</u> lecture / textbook material indicated on the course webpage - exception - Lecture 12 just worry about what we covered in class.
4. You will be given the definitions included on the mid-term.

Formula's You should memorize.

Summation Formulae p. 157 text. (Except last 2)

Pigeonhole Principle p. 347 Thm 1.
Gen. Pigeonhole Principle. p 349 Thm 2.

r-Permutation formula $P(n,r)$ p. 356

r-Combination formula $C(n,r)$ p. 358

The Binomial Theorem p. 363

r-permutations & r combinations with repetition. p. 375 (Table 1)

Algorithms You should know

Insertion and Bubble Sort

Merge Sort

Warshalls Algorithm (you will not be asked to calculate on the exam).

How to compute various closures on relations