

MAT 7003 : Mathematical Foundations

(for Software Engineering)

J Paul Gibson, A207

paul.gibson@it-sudparis.eu

<http://www-public.it-sudparis.eu/~gibson/Teaching/MAT7003/>

Graphs and Trees

<http://www-public.it-sudparis.eu/~gibson/Teaching/MAT7003/L9a-GraphsAndTrees.pdf>

Graphs

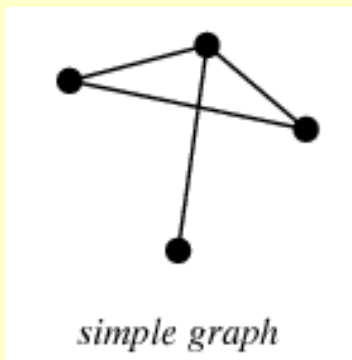
A graph is a:

- set of *points*,
- and *lines* connecting some (possibly empty) subset of the points.

The points of a graph are most commonly known as graph **vertices**, but may also be called "nodes" or simply "points." (NOTE: every vertex in a graph must be distinguishable from every other vertex in that graph.)

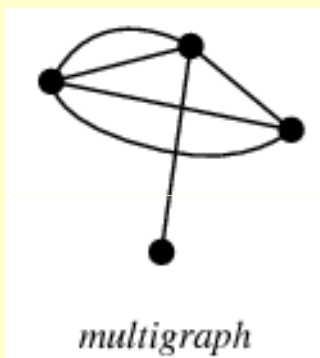
Similarly, the lines connecting the vertices of a graph are most commonly known as graph **edges**, but may also be called "arcs" or "lines."

Graphs come in a wide variety of different sorts



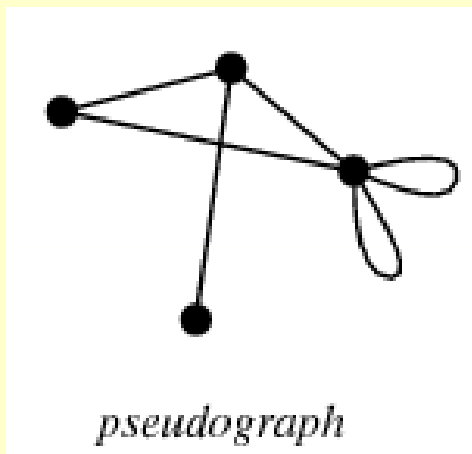
The most common type is graphs in which at most one edge (i.e., either one edge or no edges) may connect any two vertices. Such graphs are called **simple graphs**.

Graphs come in a wide variety of different sorts



If multiple edges are allowed between vertices, the graph is known as a **multigraph**.

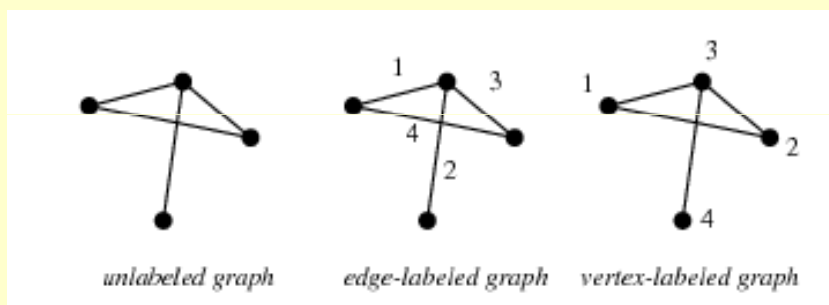
Graphs come in a wide variety of different sorts



Vertices are usually not allowed to be self-connected, but this restriction is sometimes relaxed to allow such "graph loops." A graph that may contain multiple edges and graph loops is called a pseudograph.

Graphs come in a wide variety of different sorts: labellings

The edges, vertices, or both of a graph may be assigned specific values, labels, or colors, in which case the graph is called a labeled graph.

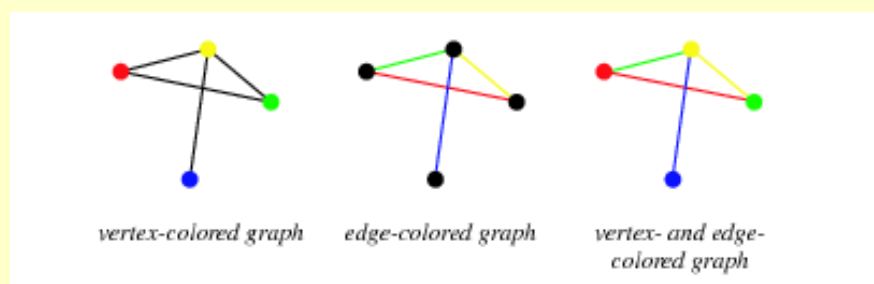


Graphs come in a wide variety of different sorts: colourings

A vertex coloring is an assignment of labels or colors to each vertex of a graph such that no edge connects two identically colored vertices.

An edge coloring is an assignment of labels or colors to each edge of a graph such that adjacent edges (or the edges bounding different regions) must receive different colors.

The assignment of labels or colors to the edges or vertices of a graph based on a set of specified criteria is known as graph coloring.

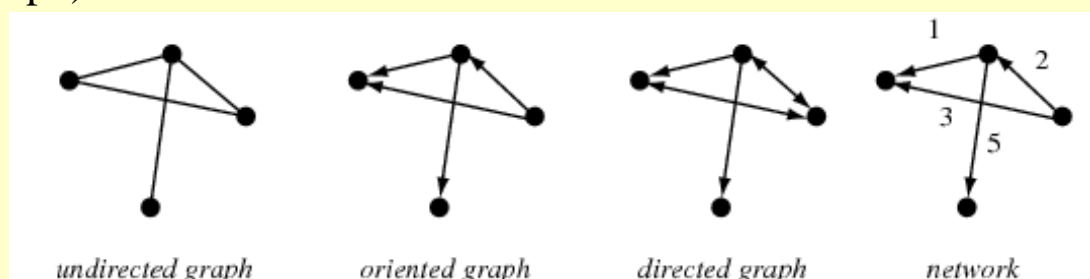


Graphs come in a wide variety of different sorts - directedness

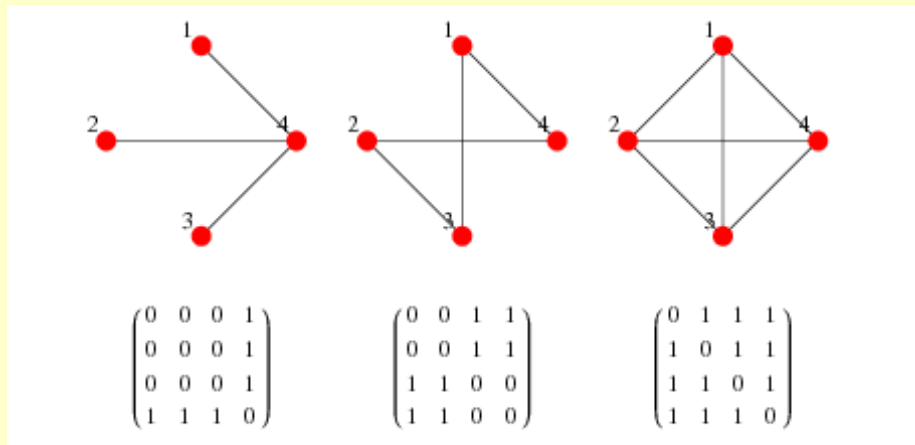
If arrows may be placed on one or both endpoints of the edges of a graph to indicate directedness, the graph is said to be *directed* (or a *digraph*); otherwise it is *undirected*.

A directed graph in which each edge is given a unique direction (i.e., edges may not be bidirected and point on both directions as once) is called an *oriented graph*.

A graph or directed graph together with a function which assigns a positive real number to each edge (i.e., an oriented edge-labeled graph) is known as a *network*.

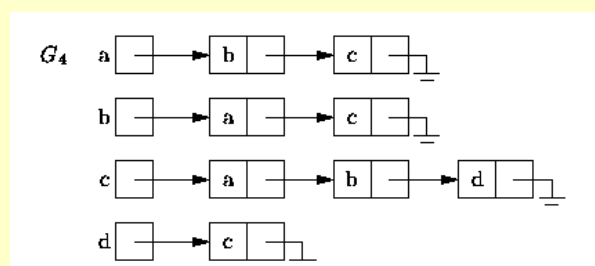
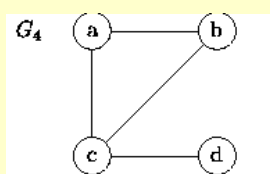


Graphs as adjacency matrices



We can use a matrix to represent a graph's structure.

Graphs as adjacency lists



We can also use lists to represent a graph's structure.

QUESTION: How to chose between matrix and list representation?

Graphs: more terminology and definitions

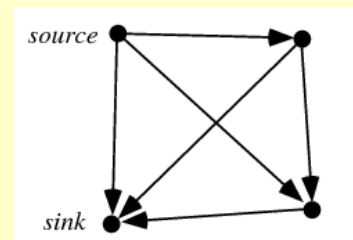
The *degree* (or *valence*) of a vertex is the number of edge *ends* at that vertex.

The *out-degree* of a node in a digraph is: the number of edges emanating from that node

The *in-degree* of a node in a digraph is: the number of edges incident on that node

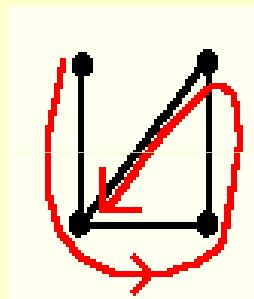
A *source* has *in-degree*=0

A *sink* has *outdegree*=0



Graphs: more terminology and definitions

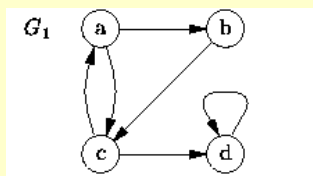
A *path* is a sequence of consecutive edges in a graph and the *length* of the path is the number of edges traversed.



Graphs: more terminology and definitions

Consider the path $P = \{v_1, v_2, \dots, v_k\}$ in a directed graph $G = (V, E)$.

- Vertex v_{i+1} is the *successor* of vertex v_i for $1 \leq i < k$. Each element v_i of path P (except the last) has a *successor*.
- Vertex v_{i-1} is the *predecessor* of vertex v_i for $1 < i \leq k$. Each element v_i of path P (except the first) has a *predecessor*.
- A path P is called a *simple* path if and only if $v_i \neq v_j$ for all i and j such that $1 \leq i < j \leq k$. However, it is permissible for v_1 to be the same as v_k in a simple path.
- A *cycle* is a path P of non-zero length in which $v_1 = v_k$. The *length of a cycle* is just the length of the path P .
- A *loop* is a cycle of length one. That is, it is a path of the form $\{v, v\}$.
- A *simple cycle* is a path that is both a *cycle* and *simple*.



QUESTION: find a path, cycle, loop and simple cycle in graph G_1

Graphs: more terminology and definitions

Two vertices are *adjacent* if they are connected by an edge.

A *complete* graph with n vertices (denoted K_n) is a graph with n vertices in which each vertex is connected to each of the others (with one edge between each pair of vertices)

A graph is *connected* if there is a path connecting every pair of vertices. A graph that is not connected can be divided into connected components (disjoint connected subgraphs).

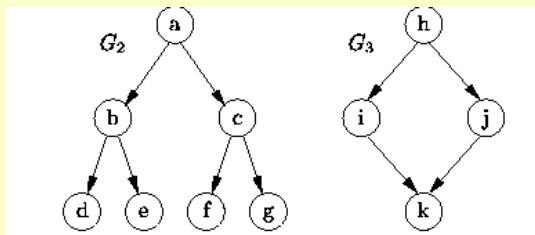
A vertex of degree zero (with no edges connected) is *isolated*

A vertex of degree one (with only one edge connected) is a *pendant* edge.

Directed Acyclic Graphs (DAGs)

For certain applications it is convenient to deal with graphs that contain no cycles. For example, a tree is a special kind of graph that contains no cycles.

DEFINITION: A *directed, acyclic graph* is a directed graph that contains no cycles.



NOTE: G_2 is a particular type of DAG – a *tree*

QUESTION: Can you define a tree?

Graph Traversals

There are many different applications of graphs. As a result, there are many different algorithms for manipulating them.

Many of the different graph algorithms systematically visit all the vertices in the graph. That is, the algorithm walks through the graph data structure and performs some computation at each vertex in the graph. This process of walking through the graph is called a *graph traversal*.

While there are many different possible ways in which to systematically visit all the vertices of a graph, the most common are:

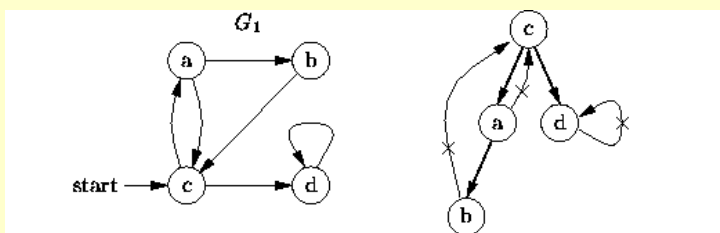
- depth-first traversal,
- breadth-first traversal and
- topological sort.

Graph Traversals: *depth-first*

Depth-first traversal of a graph visits a *start-vertex* and then recursively visits all the vertices adjacent to that node.

The graph may contain cycles, but the traversal must visit every vertex at most once.

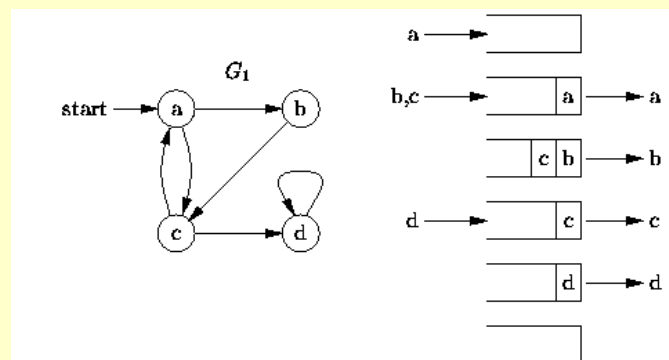
The solution to the problem is to keep track of the nodes that have been visited, so that the traversal does not suffer the fate of infinite recursion. For example:



Graph Traversals: *breadth first*

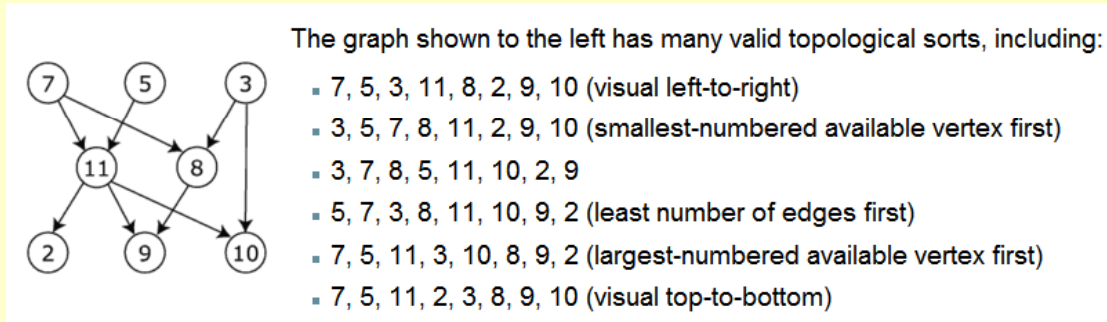
Since a graph has no root, when we do a breadth-first traversal, we must specify the vertex at which to start the traversal.

Furthermore, we can define the *depth* of a given vertex to be the length of the shortest path from the starting vertex to the given vertex. Thus, breadth-first traversal first visits the starting vertex, then all the vertices adjacent to the starting vertex, and then all the vertices adjacent to those, and so on.



Graph Traversals: *topological sort*

A *topological sort* or *topological ordering* of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.



Graph Algorithms

There are many interesting graph problems and algorithmic solutions, e.g:

- Traversal
- Searching
- Shortest/Longest Paths
- Eulerian Hamiltonian Circuits
- Max/Min flow
- Flooding
- Isomorphisms
- Reduction
- Re-writing
- Color coding
- Spanning Trees

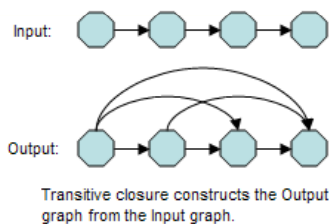
TO DO: You need to have a broad understanding of all of these problems

They arise in many different problem domains and you should be aware of classical solutions that can be re-use/adapted

We return to some of these in later lectures when we look at computational complexity

Graph Algorithms: *transitive closure*

In **computer science** the concept of transitive closure can be thought of as constructing a data structure that makes it possible to answer reachability questions. That is, can one get from node a to node d in one or more hops? A binary relation tells you only that node a is connected to node b, and that node b is connected to node c, etc. After the transitive closure is constructed, as depicted in the following figure, in an $O(1)$ operation one may determine that node d is reachable from node a. The data structure is typically stored as a matrix, so if $\text{matrix}[1][4] = 1$, then it is the case that node 1 can reach node 4 through one or more hops.



Trees

A *tree* is a graph in which any two vertices are connected by exactly one simple path.

In other words, any connected graph without cycles is a tree.

A *forest* is a disjoint union of trees.

A tree is called a *rooted tree* if one vertex has been designated the root, in which case the edges have a natural orientation, towards or away from the root. The *tree-order* is the *partial ordering on the vertices* of a tree with $u \leq v$ if and only if the unique path from the root to v passes through u .

In a rooted tree, the *parent* of a vertex is the vertex connected to it on the path to the *root*; every vertex except the root has a unique parent. A *child* of a vertex v is a vertex of which v is the parent. A *leaf* is a vertex without children.

Trees: some specialisations

An *n-ary tree* is a rooted tree for which each vertex which is not a leaf has at most n children.

2-ary trees (resp. 3-ary trees) are sometimes called *binary trees* (resp. *ternary trees*)

A *binary search tree (BST)* is a node based binary tree data structure (where each node has a data *value* or *key* that are ordered) which has the following properties:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

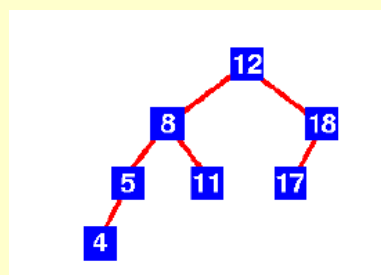
Trees: balanced binary search tree

Definition: A *balanced binary search tree* requires that:

The tree is a binary search tree, and it is *balanced*:

- The *height* of the two subtrees (*children*) of a node differs by at most one. (Where, the height is the maximum distance of any *leaf* from the *root* of a *tree*)
- Each subtree is *balanced*

For example:



Trees: AVL tree

The AVL tree is named after its two inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper “*An algorithm for the organization of information.*”

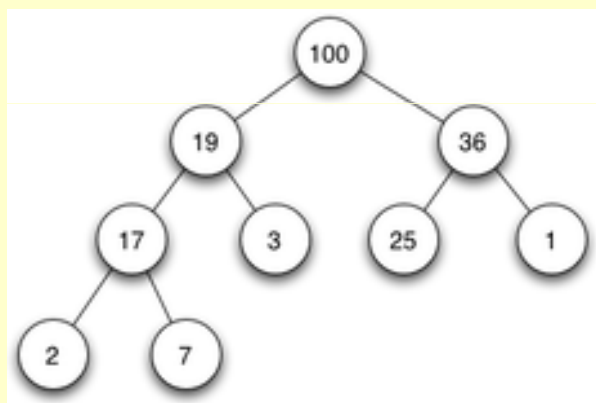
An AVL tree is a self-balancing binary search tree, and it is the first such data structure to be invented

TO DO: Check that you know how **insertions** and **deletions** work for such AVL trees

Trees: some specialisations

A *heap* is a specialized binary tree where if B is a child node of A , then $\text{key}(A) \geq \text{key}(B)$.

For example:



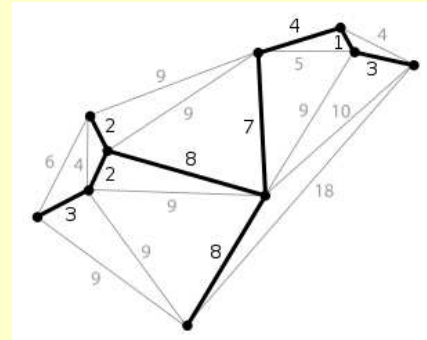
Trees from graphs – minimum spanning tree

Informally, a spanning tree of a graph G is a selection of edges of G that form a tree *spanning* every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed.

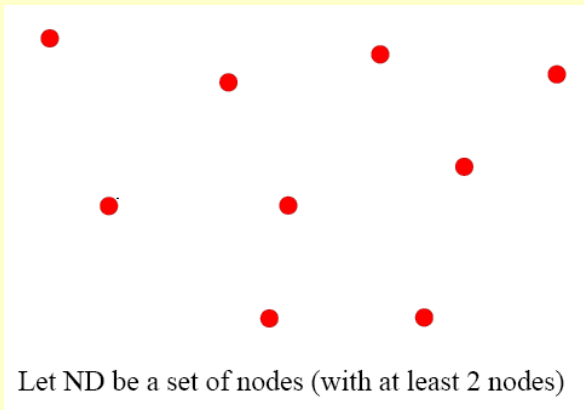
It is often useful to find a minimum spanning tree of a weighted graph, ie a spanning tree with weight less than or equal to the weight of every other spanning tree.

NOTE: The *weight* is defined as the sum of the weights of the edges in that spanning tree.

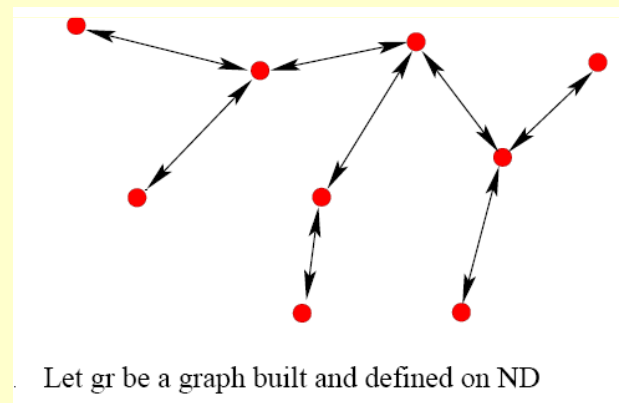
EXAMPLE:



Modelling a graph as the cartesian product of nodes



Modelling Step



gr is a symmetric and irreflexive graph

g is a graph built on N

$$g \subseteq N \times N$$

g is symmetric

$$g = g^{-1}$$

g is irreflexive

$$\text{id}(N) \cap g = \emptyset$$

TO DO: Specify - The *degree* (or *valence*) of a vertex is the number of edge *ends* at that vertex.

Degree: Graph, $N \rightarrow \text{Nat}$