Graphs describe how things are connected to other things

Graphs consist of a set of nodes, and a set of edges which link some of the nodes.

**Directed graphs** aka digraphs. In a directed graph, the edges have associated directions.

**Multigraph** - two or more edges possible on a single node

**Undirected graph** - no associated edge direction

**Weighted graph** - Weighting applied to edges

**Examples**

Wired computer network - undirected multigraph

Maps (Geographical Data) - directed and undirected multigraph

Plan of a building (edges = doors, rooms = nodes) - undirected

Social Networks

Logical relations

Pipelines

Finite automata

Electrical Circuits

Trees (file storage/ syntax)

Two linked nodes are called **adjacent**.

an edge from A to B is said to have A as it’s **source** node, and B as it’s **target** node.

an edge is **incident** on a node if it has the node as it’s source or target

For an undirected graph, the number of edges incident at the node is called the “**degree**” of the node

For directed graphs, similar but with **in-degree** and **out-degree**

A **path** is a sequence of adjacent nodes.

If there is a path from A to B, B is **reachable** from A

A path from a node to itself is a **cycle**, a **loop** is an edge from a node to itself

In an undirected graph, two nodes are connected if there is a path between them

In a directed graph, two nodes are connected if there is a **sequence of adjacent nodes** between them

A **graph is connected** if all pairs of nodes are connected

**Subgraph -** A subset of the nodes and edges of a graph G

**Connected Component -** A largest connected subgraph. Just like little clusters

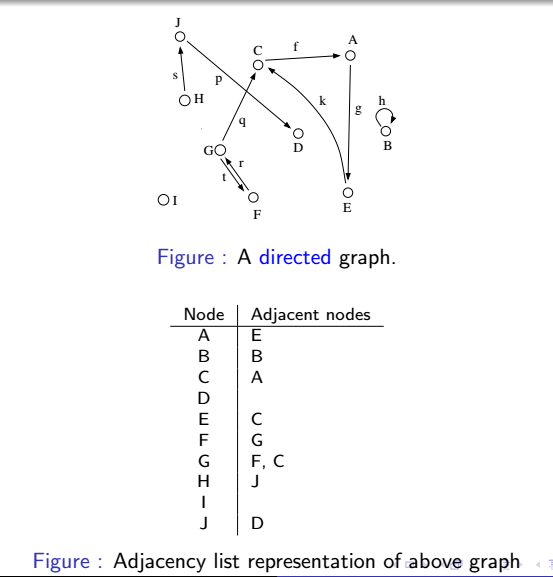
**Representing graphs using primitive programming types**

Depends on certain things

* Properties of the graph
* The algorithms we wish to implement
* The programming language

**Adjacency List**

A list of a nodes, and with each node, a list of all adjacent nodes can be used to represent a graph.



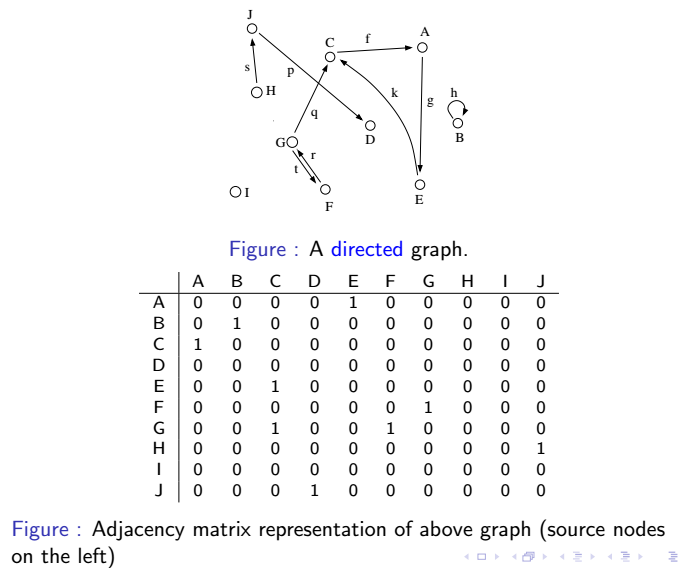
**Adjacency Matrix**

A 2-D matrix, each dimension is indexed by the nodes, giving an nxn matrix.

contains a 1 at (m,n) if there is an edge from m to n

Not particularly spatially efficient for a small number of edges.

For an undirected graph, the matrix will be symmetrical.



Both of the above are **tabular representations**

The choice of implementation tends to depend on the application.

For example, an adjacency matrix is probably not a good choice if you have strict memory limitations, due to the amount of redundancy

However, adjacency matrices allow us to do arithmetic on the graph. For example, we can perform matrix addition/multiplication on an adjacency matrix

In general, adjacency lists tend to perform better in graph traversal algorithms, whereas the matrix representation has other applications.

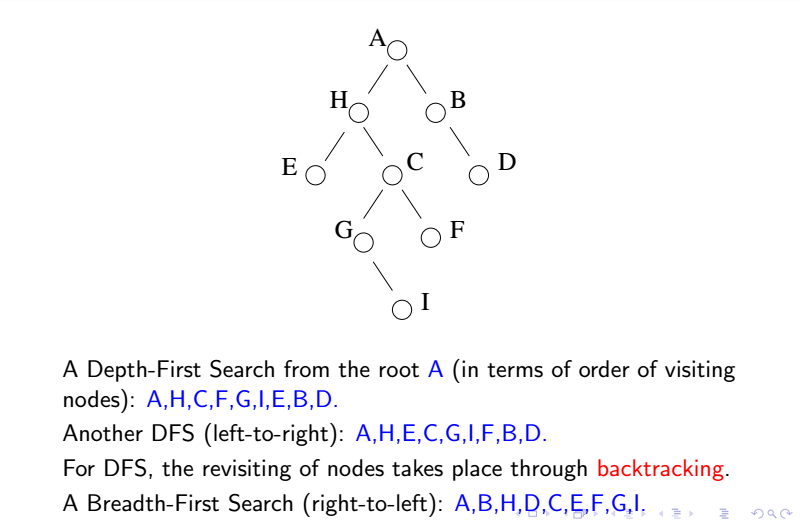
**Graph Traversal**

Graph traversal is the process of visiting nodes in the graph using edges and revisits of nodes.

Two main techniques for graph traversal:

Depth-First Search - Visit all descendents of a node before visiting it’s siblings

Breadth-First Search - Visit all children of a node, then all grandchildren etc

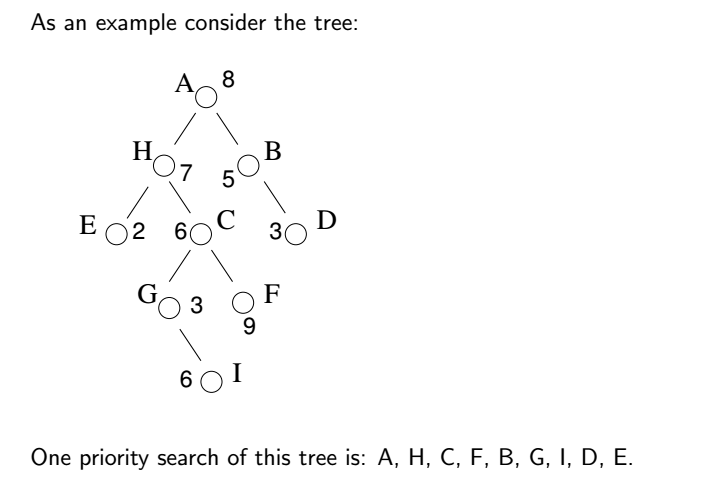


**Priority Search**

Let each node have an assigned priority

1. Visit the root
2. At each step, visit a node that has the highest priority amongst unvisited children of visited nodes.

Priority searches are often used to implement **heuristic** search methods, where the priority is calculated at each node, to give a prediction of whether a route through this node is likely to reach the required goal.



**Generic Search Routine For Trees**

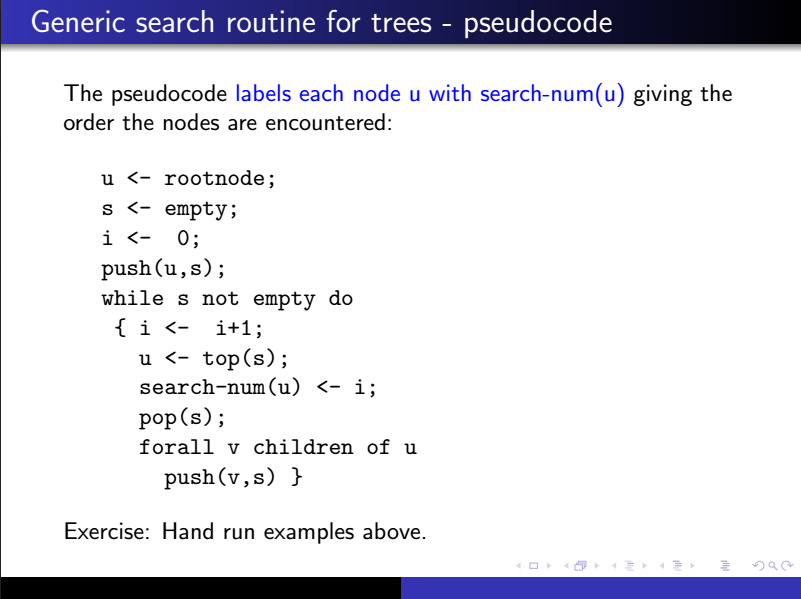
We can use an auxiliary data structure with operations, push(v), pop(), **top()**, empty() to implement all the search operations used in a tree.

When the data structure is a stack, the traversal is DFS.

When the data structure is a queue the traversal is BFS,

When the data structure is a priority queue, the traversal is a priority search.

1. Start by initially pushing root node of the tree onto the structure
2. For each iteration, visit the **top** element of the structure, pop it and push it’s children onto the structure



This means the structure will always store the unvisited children of visited nodes in the order that we encounter them.

In a priority queue, top returns the highest priority element, and pop pops this element.

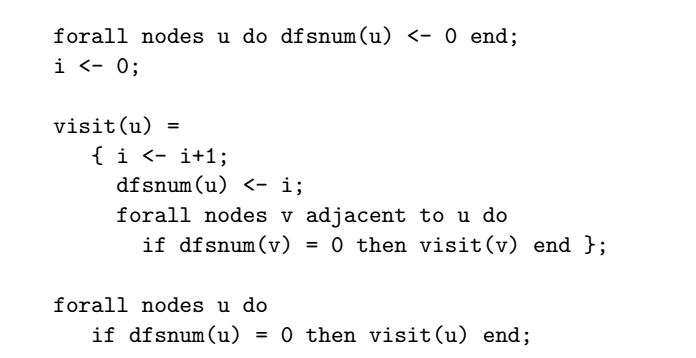
To modify Tree Traversal for graphs:

1. We may revisit nodes, so we must mark nodes as visited/unvisited
2. There may not be a single node from which all others are reachable. So we choose a node, perform traversal, then start traversal again from any unvisited nodes.

**Depth First Search**

It is possible to implement a DFS recursively by allocating a number to each node in a graph.

We can use this numbering to define whether a node has been previously visited (1 = visited, 0 = not visited)



In the above pseudo code, the first step allocates the number zero to each node, and sets a constant “i” to 0.

The next step defines the function visit(), which locally increments “i” to 1, and assigns this to the node, marking it as visited. We then check each adjacent unvisited node and repeat the step by calling the function on the new node.

The last step calls visit() on every node in the tree. Most nodes will be visited in the first few iterations, but we call visit() on every node to make sure we visit any disconnected nodes or clusters

DFS - For a graph with N nodes and E edges

For an adjacency list, we visit each node once, plus we iterate through the list of adjacent nodes (number of edges), so the complexity is linear = **O(N + E)**

For an adjacency matrix representation, the complexity is **O(N^2)**

Stacks and Recursion go hand in hand.

DFS ordering achieved in the search is that of a stack-based principle.

BFS does **not** have a recursive implementation.

**Path Finding**

Different types of pathfinding

* Find all paths between all nodes - **Transitive Closure**
* Find all paths between fixed pair of nodes
* Find all paths from one node to all others - **Single Source Problem**

When edges are labelled we can find things such as the “shortest path”, where the length of a path is the sum of all it’s edge labels.

**There is an optimization problem concerning shortest paths:**

If p is a shortest path from v to u via w, vw and wu are also shortest paths.

This is the basis of both Floyd’s algorithm and Dijkstra's Algorithm.

To accumulate shortest paths, we need combine only other shortest paths, and not consider any other paths.

**Planarity**

Planarity concerns how to represent graphs in 2D space. i.e how do we draw graphs, and can we do so without any edges crossing?

Embedding is basically drawing all points/edges on a plane, without any two edges crossing.

Not all graphs can be embedded in a plane. Graphs that can are called planar graphs.

There is an algorithm (Hopcroft and Tarjan) to deduce the planarity of a graph in linear time (based on DFS)

**Graph Colouring**

A colouring of a graph with k colours is an allocation of colours to the nodes of the graph so that each node has just one colour, and nodes linked by an edge have different colours.

This problem is NP-Complete, meaning the only way to calculate it is by using exhaustive backtracking algorithms, and hence takes exponential time.