**Graph Theory**

**Graph theory** is the mathematical theory of the properties and applications of graphs (networks).

**Types of Graphs**

**Undirected Graph:** a graph in which edges have no orientation. The edge (*u*, *v*) is identical to the edge (*v*, *u*).

**Directed Graph (Digraph):** a graph in which edges have orientation. The edge (*u*, *v*) is the edge *from* node u *to* node *v*.

**Weighted Graphs:** a graph that in which edges have a certain weight to represent an arbitrary value such as cost, distance, quantity, etc. Weighted graphs come in both directed and undirected flavours.

**Special Types of Graphs**

**Trees:** A tree is an undirected graph with no cycles. Equivalently, it is a connected graph with N nodes and N-1 edges.

**Rooted Trees:** A tree with a designated root note where every edge either points away from or towards the root node. When edges point away from the root the graph is called arborescence (out-tree) and anti-arborescence (in-tree) otherwise.

**Directed Acyclic Graphs (DAGs):** Are directed graphs with no cycles. These graphs play an important role in representing structures with dependencies. Several efficient algorithms exist to operate on DAGs. **Note:** All out-trees are DAGs but not all DAGs are out-trees.

**Bipartite Graph:** A graph whose vertices can be split into two independent groups *U*, *V* such that every edge connects between *U* and *V*. Other definitions exist such as: The graph is two colourable or there is no odd length cycle.

**Complete Graphs:** A graph where there is a unique edge between every pair of nodes.

**Representing Graphs**

Adjacency Matrix

An adjacency matrix *m* is a very simple way to represent a graph. The idea is that the cell m[i][j] represents the edge weight of going from node i to node j.

**Note:** It is often assumed that the edge of going from a node to itself has a cost of zero.

|  |  |
| --- | --- |
| Pros | Cons |
| Space efficient for representing dense graphs. | Requires **O(V²)** space. |
| Edge weight lookup is O(1). | Iterating over all edges takes **O(V²)** time. |
| Simplest graph representation. |  |

Adjacency List

An adjacency list is a way to represent a graph as a map from nodes to lists of edges.

|  |  |
| --- | --- |
| Pros | Cons |
| Space efficient for representing sparse graphs. | Less space efficient for denser graphs. |
| Iterating over all edges is efficient. | Edge weight lookup is **O(E)**. |
|  | Slightly more complex graph representation. |

Edge List

An edge list is a way to represent a graph simply as an unordered list of edges. Assume the notation for any triplet (u, v, w) means: “the cost from node u to node v is w”.

This representation is seldomly used because of its lack of structure. However, it is conceptually simple and practical in a handful of algorithms.

**Common Graph Theory Problems**

For the upcoming problems ask yourself:

* Is the graph directed or undirected?
* Are the edges of the graph weighted?
* Is the graph I will encounter likely to be sparse or dense with edges?
* Should I use an adjacency matrix, adjacency list, and edge list or other structure to represent the graph efficiently?

**Shorted path problem**: Given a weighted graph, find the shortest path of edges from node A to node B. Algorithms: BFS (unweighted graph), Dijkstra’s, Bellman-Ford, Floyd-Warshall, A\*, and many more.

**Connectivity:** Does there exist a path between node A and node B? Typical solution: use union find data structure or any search algorithm (e.g. DFS).

**Negative cycles:** Does my weighted digraph have any negative cycles? If so, where? Algorithms: Bellman-Ford and Floyd-Warshall.

**Strongly Connected Components:** SCCs can be thought of as self-contained cycles within a directed graph where every vertex in a given cycle can reach every other vertex in the same cycle. Algorithms: Tarjan’s and Kosaraju’s algorithm.

**Traveling Salesman Problem:** Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? Algorithms: Held-Karp, branch and bound and many approximation algorithms.

**Bridges:** A bridge / cut edge is any edge in a graph whose removal increases the number of connected components. Bridges are important in graph theory because they often hint at weak points, bottlenecks, or vulnerabilities in a graph.

**Articulation points:** An articulation point / cut vertex is any node in a graph whose removal increases the number of connected components. Articulation points are important in graph theory because they often hint at weak points, bottlenecks, or vulnerabilities in a graph.

**Minimum Spanning Tree (MST):** A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Algorithms: Kurskal’s, Prim’s & Boruvka’s algorithm.

**Network Flow:** With an infinite input source how much “flow” can we push through the network? Suppose the edges are roads with cars, pipes with water or hallways packed with people. Flow represents the volume of water allowed to flow through the pipes, the number of cars the roads can sustain in traffic and the maximum amount of people that can navigate through the hallways. Algorithms: Ford-Fulkerson, Edmonds-Karp & Dinic’s algorithm.

**Depth-First Search (DFS) Overview**

The DFS is the most fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of O(V+E) and is often used as a building block in other algorithms.

By itself, the DFS isn’t all that useful, but when augmented to perform other tasks such as count connected components, determine connectivity, or find bridges articulation points then DFS really shines.

A DFS plunges depth first into a graph without regard for which edge it takes next until it cannot go any further at which point it backtracks and continues.

# Global or class scope variables

n = 'number of nodes in the graph'

graph = 'adjacency list representing graph'

visited = [False, '...', False]  # size n

def dfs(at):

    if visited[at]:

        return

    visited[at] = True

    neighbours = graph[at]

    for next in neighbours:

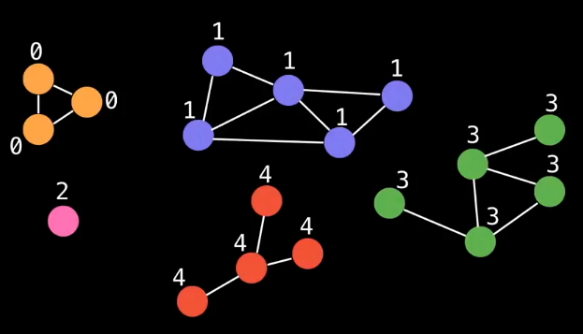
        dfs(next)

# start DFS at node zero

start\_node = 0

dfs(start\_node)

**Connected Components**

Sometimes a graph is split into multiple components. It’s useful to be able to identify and count these components. One way is to colour nodes so we can tell them apart. What does colouring nodes mean? It is equivalent to labelling each node in a component with an integer to be able to tell them apart.

33:20