**Graph**

*Nodes and Vertices* - both can carry data

**Connectivity - "Graph Theory"**

- Connected graph has no disconnected nodes

- Connectivity measures the ammount of element that can be removed to get a connected graph

+ Disconnected: some vertex or group of vertices that have no connection with the rest

+ Weakly Connected: A directed graph is weakly connected when only by replacing all of the directed edges with undirected edges can cause it to be connected.

+ Connected: Here we only use "connected graph" to refer to undirected graphs. In a connected graph, there is some path between one vertex and every other vertex.

+ Strongly Connected: Strongly connected directed graphs must have a path from every node and every other node. So, there must be a path from A to B **AND** B to A.

**Graph Representations**

OOPs would use objects to represent nodes and graphs. However:

- Edge List: 2D List, each showing which nodes IDs that edge connect

- Adjacency List: 2D List, each index is node id IDs, the list containing IDs of reachable nodes

- Adjacency Matrix: Similar to adjacency list, but each list is decoder for reachable nodes [0, 1, 0, 1]

🡪 This node can reach node id = 1 and id = 3

A picture containing text, screenshot, font, diagram

Description automatically generated

(*if we want the index to be the node id 🡪 Also need the 0th row)*

**Graph Traversal**

1. **Depth First Search (keep going deeper)**

- Use **Stack** to mark "exploring" nodes, a method to check "seen" nodes (could be a bool in node)

|  |
| --- |
| push start node to stack  while stack is not empty:  current node = top of stack (no popping)  Mark current node as seen. Pick current node's edges:  If there is unexplored edge (how to know?): mark explored edge and get next node  If next node has been seen, continue;  Else, push next node to stack.  else if there are no more edges, pop the current node off the stack |

🡪 O(2e + v) *explore each edge twice*

- Recursion: dfs\_helper(self, start\_node)

|  |
| --- |
| ret\_list = [start\_node.value]  start\_node.visited = True  for adj\_edge in start\_node.edges:       if (start\_node.value == adj\_edge.node\_from.value and not adj\_edge.node\_to.visited):             ret\_list = ret\_list + self.dfs\_helper(adj\_edge.node\_to)  return ret\_list |

1. **Breath First Search (by levels)**

- Use **Queue** to mark "exploring" nodes, a method to check "seen" nodes (could be a bool in node)

from collections import deque

ret\_list = [node.value]

# Your code here

queue = deque()

node.visited = True

queue.append(node)

while(len(queue) != 0):

curr = queue.popleft()

for edge in curr.edges:

if edge.node\_from.value == curr.value and not edge.node\_to.visited:

            edge.node\_to.visited = True

                queue.append(edge.node\_to)

                ret\_list.append(edge.node\_to.value)

return ret\_list