Graphs

- A graph is a collection of nodes/vertices and edges
- Edges connect the nodes

Often defined as:

- $G = \{V, E\}$
  Where graph G is made of 2 sets V vertices and E edges.

Other notes:

- There are directed and undirected graphs.
- V is a finite set.
- E can be thought of as a binary relation.

Definitions:

- **Cycle**: A path that starts and stops at the same vertex, but contains no other repeated vertices.
- **Acyclic**: A graph with no edges.
- **Incident**: The connection between two nodes.
- **Adjacent**: If nodes are incident to each other.
- **Node Degree**: The number of edges incident to a node.
- **Regular Graphs**: Only allows one edge between any pair of nodes*. Each edge only connects two nodes.
- **Sparse Graph**:

*Directed graphs can have multiple edges between two nodes as long as they are in different directions

Edges:

- Minimum number of edges is 0 (if it is not a connected graph)
- For a connected graph, the minimum number of edges is $\approx n$ and the max is $\approx n^2$

Path (of length k)

- from vertex a to vertex b is a sequence
- $V_0, V_1, V_2, V_3, \ldots, V_k$

- Where $a = V_0$ and $b = V_k$
- $(V_i, V_{i+1}) \in E$ for all $i = 0, \ldots, k-1$
- If there is a path from a to b, then b is reachable from a
- A path is **simple** if all the nodes on the path are distinct

Connected Components:

- Any pair of nodes are mutually reachable

For graph G where $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, d\}\}$

- All nodes are connected
  For graph G where $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{c, d\}\}$
- a and b are connected and c and d are connected, but a and d are not

Graph Representation:

(2 standard methods and 1 custom)

- Adjacency Matrix
- Adjacency List
- custom method (ex. our maze)

Adjacency List:

- (Often preferred)
- Especially for sparse graphs.

Complexity:

- To traverse all adjacencies it will be $O(n)$
  - (Bad for checking a specific adjacency)
- Holds $O(E)$ memory (where E is the number of edges)

Properties:

- One list for each node
- In the list are the adjacent nodes

For graph G where $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{a, d\}, \{a, c\}, \{b, d\}, \{b, c\}, \{c, d\}\}$

1. A: b, c, d
2. B: a, c, d

3. C: a, b, d
4. D: a, b, c

Adjacent Matrix Nodes

	a	b	c	d
a		1	1	1
b	1		1	1
c	1	1		
d	1	1	1	

Where a, b, c, d are nodes.

Complexity:

- Time: $O(n^2)$

Use Case:

- Only really efficient if the graph is very dense.
- Traversal is pretty much the same as the adjacency list to check all adjacent
- To find if one node is adjacent to another, the matrix is faster
- If you know if your graph is very dense, it could be a good idea
- Easy to overuse memory with this

Custom (Maze) Representation

1. Very sparse
2. Node adjacency are not arbitrary

In our graph, a node can only be adjacent to the node to its left, right, top, or bottom.

You can make an adjacency list on the fly! (and then discard them)

What About Searching Graphs?

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

Breadth First Search

Given graph G:

- Start node s ($s \in V$)
- Explore the edges of G to discover every node that is reachable from for S (shortest path to...)
- Computes distance
- Generates a BFS Tree that has paths
- Expands a frontier:
    - It discovers all nodes of some distance k from the node before finding all the nodes of distance $k+1$ or $k+2$
    - Uses a queue to record the nodes that will be explored later on.
- Finds shortest path from starting node to every other node

Data Inside Nodes:

Each node will be colored to record its status
1. White - undiscovered
2. Grey - in progress
3. Black - complete

- Can be simplified to discovered or not discovered (bool)

Each Node Will Contain:

- distance `d[]`
- parent `p[]` (predecessor)
- color `color[]` or `is_discovered`

Vertex can only be discovered once. Once it is discovered it cannot be discovered again

Here's the code:

```
// O(V + E)
BFS(Graph g, Node s) {
        // initialization O(V)
        for each node u ∈ V − {s}
                color[u] = white
                distance[u] = infinity  // flag value
                p[u] = null             // flag value

        // initialization of the node we will search from (root)
        color[s] = gray
        d[s] = 0
```

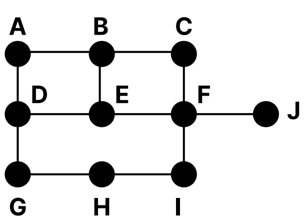
```
        p[s] = null or -1  // ...some flag that there is no parent
        q.insert(s)        // insert the starting node to the queue

        // main section O(E)
        while (!q.is_empty()) {
                u = q.dequeue()
                for each v ∈ Adjacent[u] {   // adjacency list or matrix
                        if (color[v] = white) {  // non-discovered node
                                color[v] = gray
                                d[v] = d[u] + 1
                                p[v] = u
                                q.insert(v)
                        }
                }
                color[u] = black  // done with it
        }
}
```

We will pick the flag value. There are no perfect values. Maybe it could be `null` or `-1`.

We don't know which will be longer $O(E)$ or $O(V)$ so we cannot simplify, therefore, our time complexity is $O(V+E)$.

Example:

If we were to have a graph G with vertex: $\{a,b,c,d,e,f,g,h,i,j\}$



With starting node e we would find the closes adjacent nodes and add them to the queue.

Here is the queue order grouped by distance d:

- $d=1:\{b,d,f\}$
- $d=2:\{a,c,i,g,j\}$
- $d=3:\{h\}$

We can now find the shortest path from e to another node.

Adjacency list:

node	adjacent
a	b, d
b	a, e, c
c	b, f
d	a, e, c
e	b, d, f
f	c, e, j
g	d, h
h	g, i
i	f, h
j	f

Directed Graphs:

- Cycles
- Self loops (not in our maze)
- We use the term strongly connected when the subgraph is mutually reachable:
  - $a \rightarrow b$ && $a \leftarrow b$
- We have a strongly connected component if there were something like $a \rightarrow b \rightarrow c \rightarrow a$ (a cycle)

Depth First Search

- No starting node.
- Discovers all nodes in the graph wether they are connected or not!
- Searches deeper into the graph first.
- Uses the stack and function calls to keep track of where we're exploring and where we need to go.

Data For Nodes

- Coloring `color[]`
- Timestamped:
  - each time is **unique** and is between 1 and 2 nodes
  - `d` discovery time (always less than finish time!)
  - `f` finishing time

- Parent `p`

We can code it recursively:

```
DFS(Graph g) {
	// color every node to white
	for each node v ∈ V {
		color[v] = white
		p[v] = null  // or some flag
		// note: d[v] and f[v] are uninitialized
	}
	time = 0
	for each node v ∈ V {
		if (color[v] = white) {
			DFS_visit(v)
		}
	}
}
```

We will need a helper function:

```
DFS_visit(Node u) {
	color[u] = gray
	d[u] = ++time;  // increment before setting value
	for each node v ∈ Adjacency[u] {
		if (color[v] == white) {
			parent[v] = u
			DFS_visit(v)
		}
	}
	color[u] = black  // we're done with it
	f[u] = ++time
}
```

We will need to find a way to break ties. For this, we will consider nodes in ascending order.

Given:

1. $a \to b$
2. $b \to c$

3. $c \rightarrow a$
4. $b \rightarrow d$
5. $d \rightarrow e$
6. $e \rightarrow f$
7. $e \leftarrow f$

Here's a table of the start and finish times:

Node	Discovery (time)	Finish (time)
a	1	12
b	2	11
c	3	4
d	5	10
e	6	9
f	7	8

Time between start and finish:
a : 1-12
b : 2-11
c : 3-4
d : 5-10
e : 6-9
f : 7-8

Used to classify edges:

1. Tree Edge - DFS first (a, b) where b was discovered.
2. Back Edge - edge (a, b) from a to an ancestor b
3. Forward Edge - are non-tree edges connecting node a to descendant b
4. Cross Edge - All other edges

White

Tree Edge: An edge that is white (in our case) when discovered

Gray

Back Edge: $c \rightarrow a$ (a discovered b which discovered c which goes back to a)

Black:

Forward Edge: $a \to d$
Cross Edge: $f \to c$ (non-descendants)

- If the discovery time of u is less than v, we have a forward edge
- If the discovery time of u is greater than v, we have a cross edge
-

More Use Cases:

- Topological sorting.
- To find Strongly connected components in the graph.

DFS To Identify Strongly Connected Components:

1. Identify strongly connected components.
2. DFS(G) "record finish times".
3. Generate G^T (transpose*).
4. DFS(G^T) considering nodes in decreasing finishing time.

*Transpose - Take edges and reverse them (directionally)!

Graph G

Just use this to find the times

Node	Discovery (time)	Finish (time)
d	1	6
e	2	5
f	3	4
b	8	11
c	9	10
a	7	12

Graph G_T

Used to find SCC

Node	Discovery (time)	Finish (time)
a	1	6
c	2	5

Node	Discovery (time)	Finish (time)
b	3	4
d	7	8
e	9	12
f	10	11

SCC:

$\{a, b, c\}$

$\{d\}$

$\{e, f\}$