# Graph (I)

Christy Cheng {christycty}

2023-03-25

# Pre-Requisites

Data Structure (I):

- Queue
- Stack
- Linked lists

Recursion, divide and conquer:

- Recursion

# Contents


## Introduction to Graph

- What is a graph?
- Types of graphs
- Modelling problems as graphs

## Graph Representation

- Adjacency Matrix
- Edge List
- Adjacency List
- Grid graphs

## Graph Traversal

- Depth-First Search (DFS)
- Breadth-First Search (BFS)

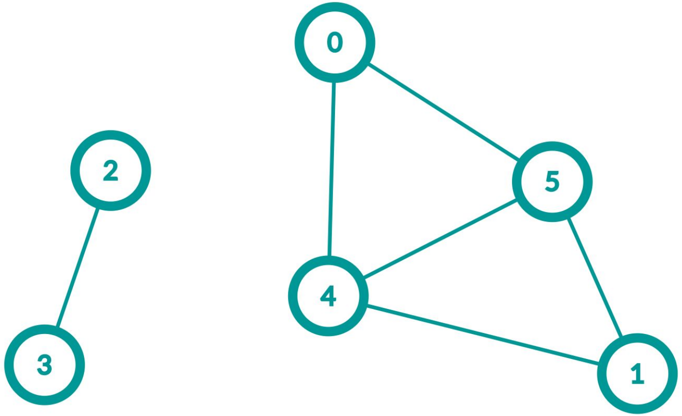
## Special Graphs

- Cycles
- Trees
  - Chains
  - Stars
- Bipartite graph
- Connected graph
- Planar graph

# Introduction to Graph

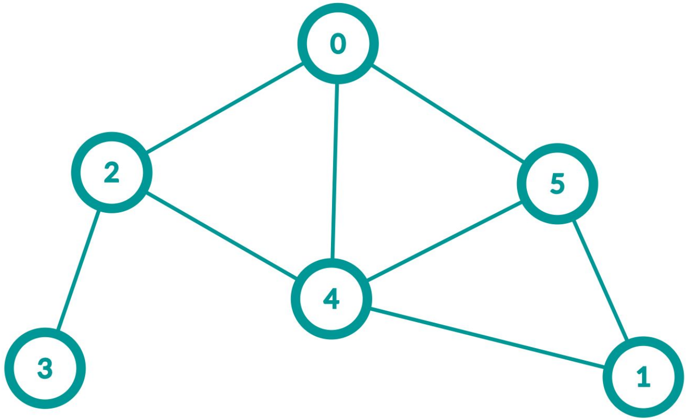
# Types of Graphs

## Disconnected Graph




{2, 3} and {0, 1, 4, 5} are connected components

## Connected Graph



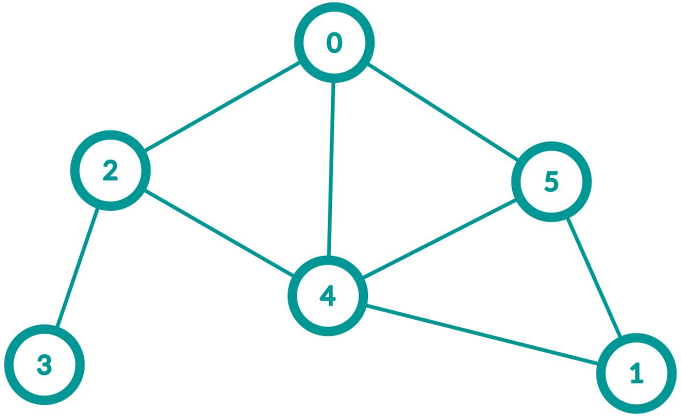

undirected edge = two oppositely directed edges

# What is a Graph?

A set of vertices/nodes linked by edges

We call a graph (G) = (V,E) where
V = set of vertices ➔ |V| = number of vertices
E = set of edges ➔ |E| = number of edges

Path = sequence of edges which joins a sequence of vertices (usually distinct)

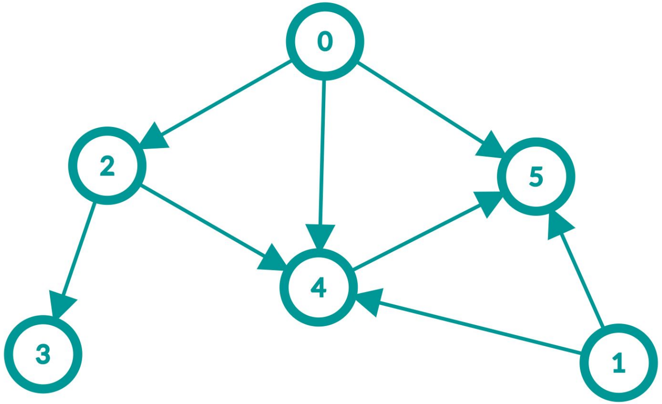


graph with 6 vertices and 8 edges

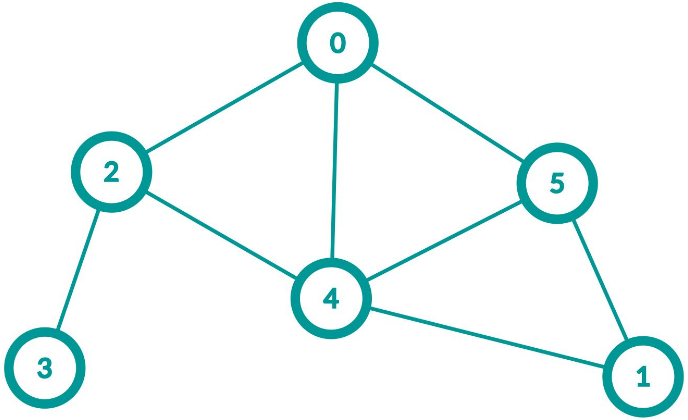
valid path: 4 → 5 → 0 → 2
invalid path: 1 → 2 → 5

# Types of Graphs

Directed Graph



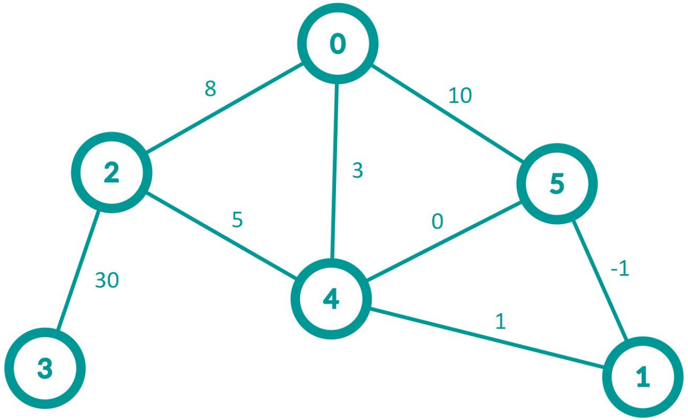

Undirected Graph




undirected edge = two oppositely directed edges

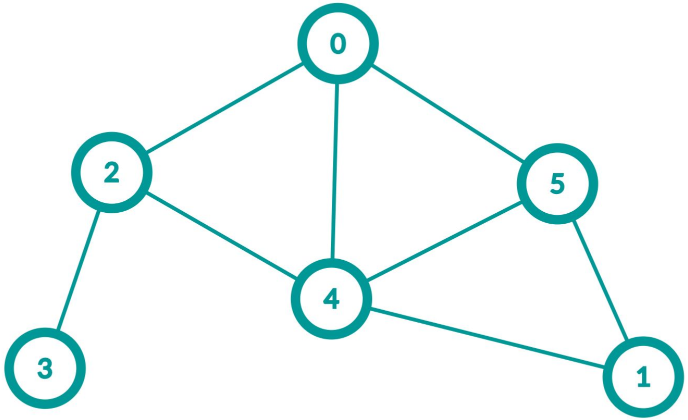
# Types of Graphs

## Weighted Graph




a value / cost is assigned to each edge

## Unweighted Graph




consider as all edges having same weight

# Modelling Problems as Graphs

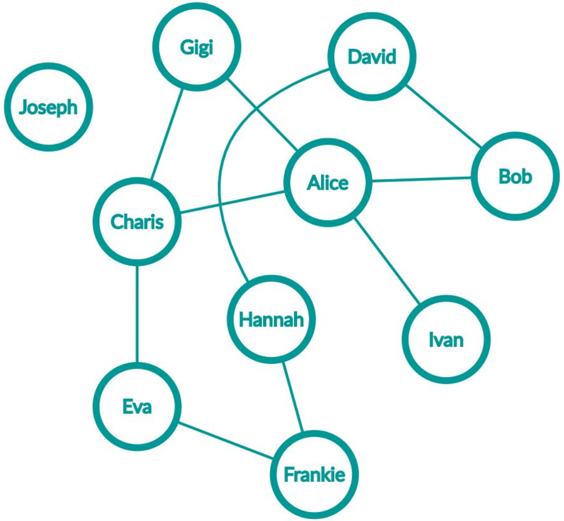
A graph can show relations/transitions between nodes (objects) using edges.

Examples:

- Map
- Maze
- Social networks
- Game states
- Conflict graph

A social network

- nodes = people
- edges = friendship




We can ask:
Do Bob and Charis have any common friends?

# Modelling Problems as Graphs

Represent :

- States/ Objects as a vertices
- Possible transitions/ relations as edges
- Transitional costs as weights

It forms a graph and makes things easier to handle.

# Modelling Problems as Graphs

Common use cases:

- Find possible paths
- Find shortest path
- Find longest path
- Count number of connected components
- Find nodes reachable with cost less than k
- Find cycles
- ...

# States - Water Jug Problem

There are 2 water jugs with capacities N and M litres respectively.

Initially, both of them are empty.

You can perform the following operations for infinitely many times (one operation a time)

1. Empty a jug
2. Fully fill a jug
3. Pour water from one jug to another until either one jug is empty / full

How to get a specific volume K in one of the jugs?

# States

The states in this problem would be the volumes of the two jugs (*v1*, *v2*)

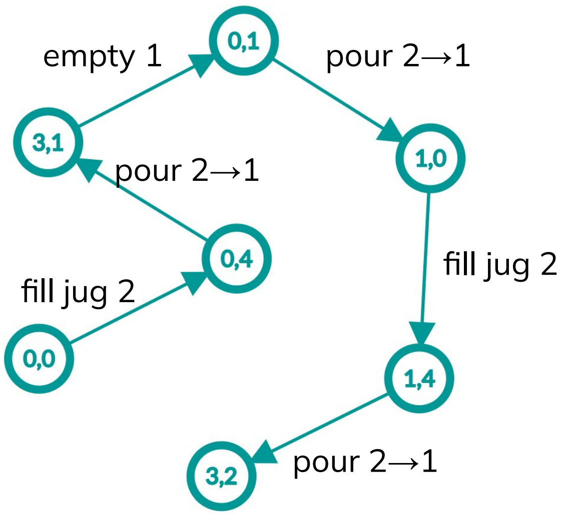
How to we represent them as a graph?

N = 3 *(jug 1)*, M = 4 *(jug 2)*, K = 2 *(goal)*

Initial state = (0, 0); Final state = (3, 2)

We can represent the solution as a path in the graph

We will further discuss this problem later.



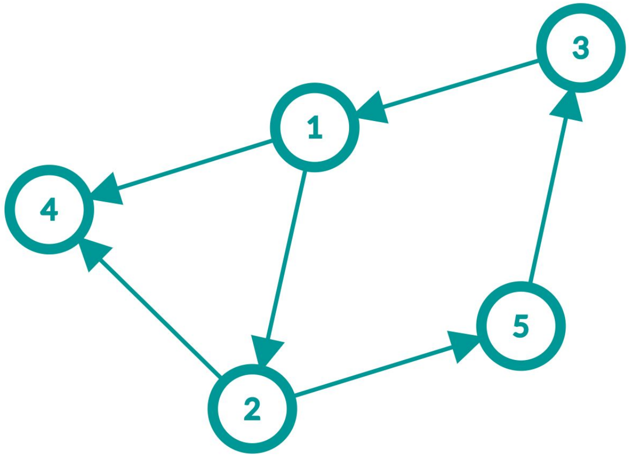
# Graph Representation

# How can we store a graph?

1. Adjacency Matrix
2. Edge List
3. Adjacency List

Methods needed:

1. Add an edge between two nodes
2. Get a node’s neighbors (or parent/child)



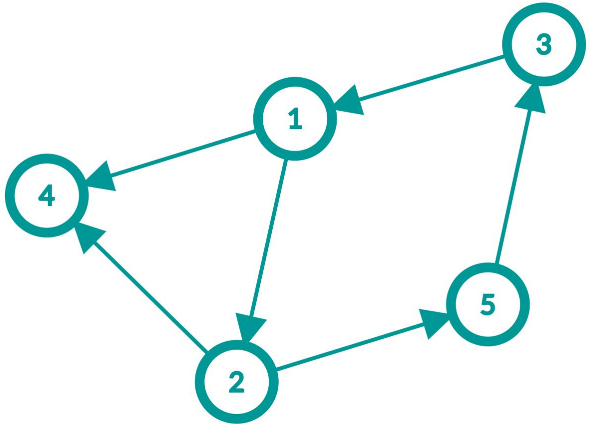

example: can we go from node 2 to node 1?

# Adjacency Matrix

A |V|x|V| array representing the relation / weight of every pair of vertices

`adj[i][j] = 1` *(or weight)* if there is an edge from vertex *i* to vertex *j*

else `adj[i][j] = 0`




| adj | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 |

# Adjacency Matrix - Implementation

```
adj[V][V]

for i ← 1 to V
for j ← 1 to V
    adj[i][j] = 0
```

Memory: $O(|V|^2)$

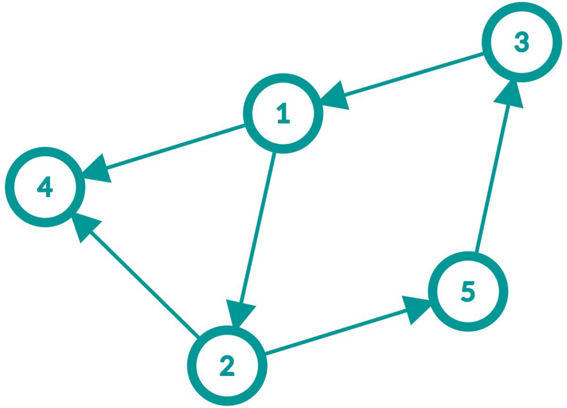
$O(|V|^2)$

```
for all edges from u to v:
    adj[u][v] = 1
```

$O(|E|)$

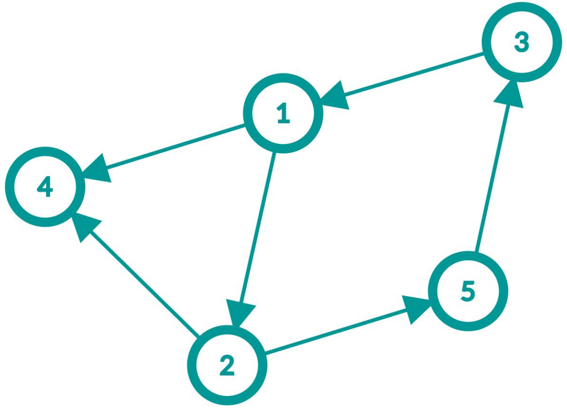
```
for all query if u → v exist:
    output adj[u][v]
```

O(1) per query




| adj | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 0/1 | 0 | 0/1 | 0 |
| 2 | 0 | 0 | 0 | 0/1 | 0/1 |
| 3 | 0/1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0/1 | 0 | 0 |

# Adjacency Matrix - Implementation




How to output all children of node x?

*all nodes y that exist an edge x → y*

```
for i ← 1 to V
    if adj[x][i] == 1
        output i
```

O(|V|) per query

How to output all parents of node x?

*all nodes y that exist an edge y → x*

```
for i ← 1 to V
    if adj[i][x] == 1
        output i
```

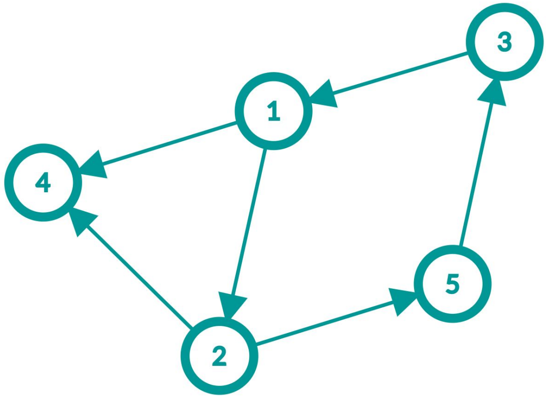
O(|V|) per query

| adj | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 |

# Edge List

Array of size `|E|` to store each edge:

`from[i]`, `to[i]` and `weight[i]` represent the the origin, the destination and the path weight of the i-th edge respectively.




| ID | from | to | weight |
|---|---|---|---|
| 1 | 1 | 2 | 1 |
| 2 | 1 | 4 | 1 |
| 3 | 2 | 4 | 1 |
| 4 | 2 | 5 | 1 |
| 5 | 3 | 1 | 1 |
| 6 | 5 | 3 | 1 |

# Adjacency Matrix - Implementation

```
edge[E]
edge_no = 0
```

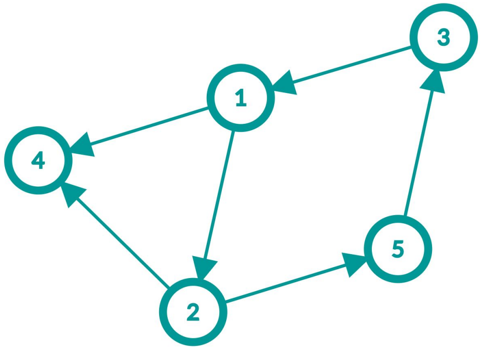
Memory: O(|E|)

```
for all edges from u to v:
    edge[edge_no] = edge(u,v)
    edge_no++
```

O(|E|)

```
for all query if u → v exist:
    for i ← 0 to edge_no - 1:
        if edge[i] is edge(u,v):
            output YES
```

O(|E|) per query




| ID | from | to |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 1 | 4 |
| 3 | 2 | 4 |
| 4 | 2 | 5 |
| 5 | 3 | 1 |
| 6 | 5 | 3 |

# Adjacency Matrix - Implementation

**Implement with C++ struct and vector**

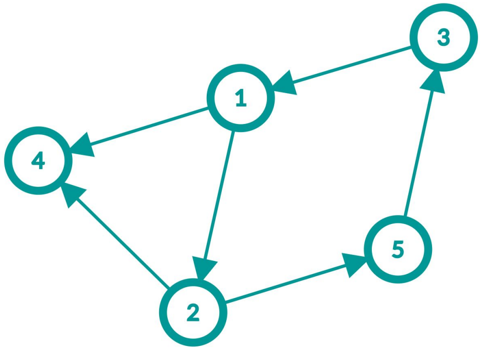
```
struct Edge {
    int from, to, weight;
};

vector<Edge> edge_list;
```

*(easier memory allocation)*

**Insert edge u → v**

```
Edge edge;
edge.from = u, edge.to = v, edge.w = 1;
edge_list.push_back(edge);
```




| ID | from | to |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 1 | 4 |
| 3 | 2 | 4 |
| 4 | 2 | 5 |
| 5 | 3 | 1 |
| 6 | 5 | 3 |

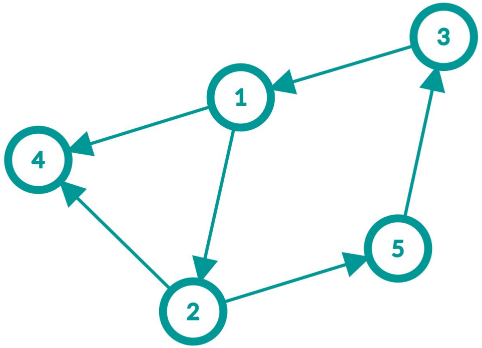
# Adjacency List

Idea: for each node, maintain a list of nodes/edges connected to it

*in practice, it is usually neighbours in undirected edges, or child for directed edges*

The list may be:

- Arrays of size |E|
- Linked lists
- Vectors




| | | |
|---|---|---|
| **1** | 2 | 4 |
| **2** | 4 | 5 |
| **3** | 1 | |
| **4** | | |
| **5** | 3 | |

# Adjacency List - Implementation

```
neighbour[V]

for all edges from u to v:
    neighbour[u].push_back(v)
    neighbour[v].push_back(u)
```

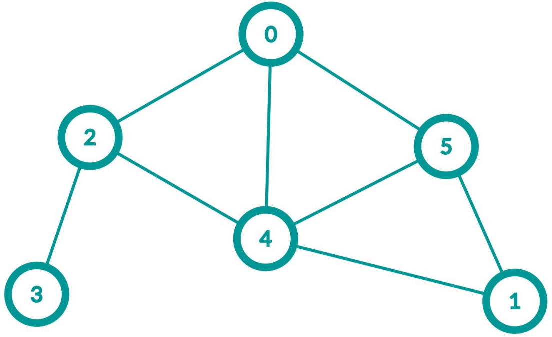
O(|E|)

```
all neighbours of node u
    for x in neighbour[u]:
        output x
```

O(|E|)

```
is v a neighbour of node u
    for x in neighbour[u]:
        if x == v:
            output YES
```

O(|E|) per query



| | | | | |
|---|---|---|---|---|
| **0** | 2 | 4 | 5 | |
| **1** | 4 | 5 | | |
| **2** | 0 | 3 | 4 | |
| **3** | 2 | | | |
| **4** | 0 | 2 | 5 | |
| **5** | 0 | 1 | 4 | |

fill the matrix

# Adjacency List - Implementation

### Implement with C++ vector

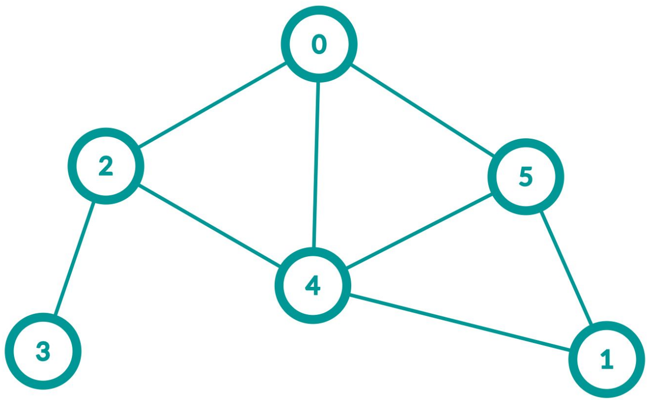
```
vector<int> neighbour[V];
```

### Insert edge u - v

```
neighbour[u].push_back(v);
neighbour[v].push_back(u);
```

### All neighbours of node u

```
for auto i in neighbour[u]:
     cout << i << " ";
```



# Adjacency List - Implementation

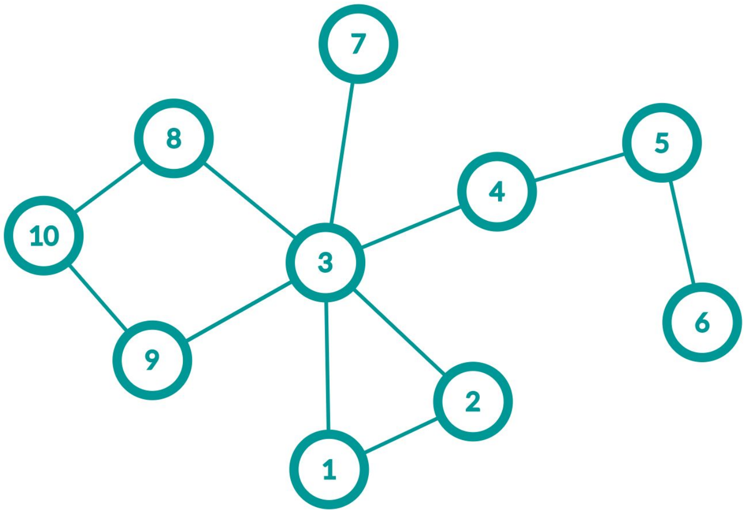
| | Array | Linked List | Vector |
|---|---|---|---|
| Declaration | int neighbour[V][E] | Node* neighbour[V] | vector<int> neighbour[V] |
| Space Efficiency | O(\|V\|\|E\|) | O(\|V\|+\|E\|) | O(\|E\|) |

for more implementation detail, refer to data structure (I)

in practice, just use vector

# Practice

Adjacency list for this graph?




| | | | | | | |
|---|---|---|---|---|---|---|
| **1** | 2 | 3 | | | | |
| **2** | 1 | 3 | | | | |
| **3** | 1 | 2 | 4 | 7 | 8 | 9 |
| **4** | 3 | 5 | | | | |
| **5** | 4 | 6 | | | | |
| **6** | 5 | | | | | |
| **7** | 3 | | | | | |
| **8** | 3 | 9 | 10 | | | |
| **9** | 3 | 10 | | | | |
| **10** | 8 | 9 | | | | |

# Pros and Cons

| | Adjacency Matrix | Edge List | Adjacency List |
|---|---|---|---|
| Ease of Implement | High | Middle | Low |
| Space Efficiency | $O(\|V\|^2)$ | $O(\|E\|)$ | $O(\|V\|+\|E\|)$ |
| Retrieve 1 edge | $O(1)$ | $O(\|E\|)$ | *faster than* $O(\|E\|)$ |
| Retrieve all neighbour | $O(\|V\|)$ | $O(\|E\|)$ | $O(\|V\|+\|E\|)$ for all nodes |
| Multiple Edges | No | Yes | Yes |

# Which to choose?

| Adjacency Matrix | Adjacency List |
|---|---|
| \|V\| is small (e.g. <= 500) | \|V\| is large (most cases) |
| dense edges | sparse edges |
| query random and unrelated edges | query all edges connected to a node |
| **Use Cases** | |
| Floyd-warshall algorithm | Dijkstra's *(and other BFS-based)* algorithm |
| | Searching (DFS, BFS) |

*In practice, we almost never use edge list*

# Break + Practice

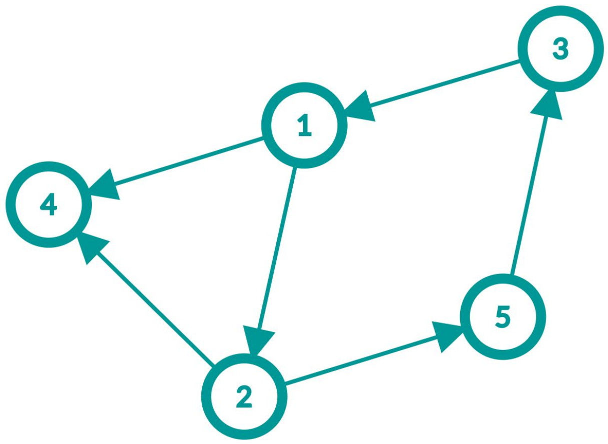
Given a graph of 5 nodes, which has no edges initially.

We want to support the following online queries:

1. Add an edge from node a to node b
2. Remove the edge from node a to node b
3. Output if there is an edge from node a to node b

Each line (representing a query) contains 3 integers: cmd a b

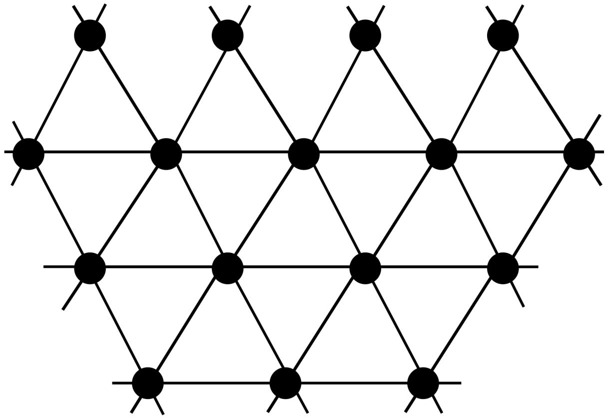
1 4 5 *add an edge from node 4 to node 5*
2 2 5 *remove an edge from node 2 to node 5*
3 3 4 *output if there is an edge from node 3 to node 4*

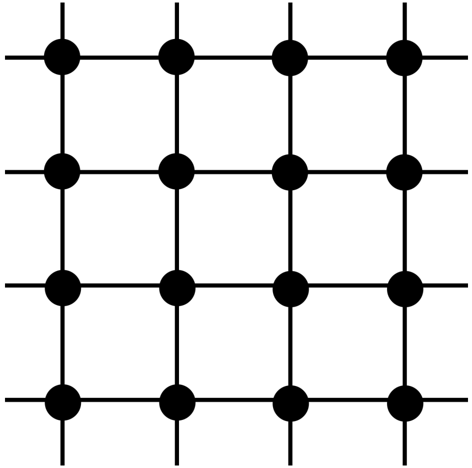



Try to test with this graph

# Grid graphs

In some problems, vertices form a regular tiling (square, triangle...)





# Grid graphs

vertices represented by coordinates (row, column)

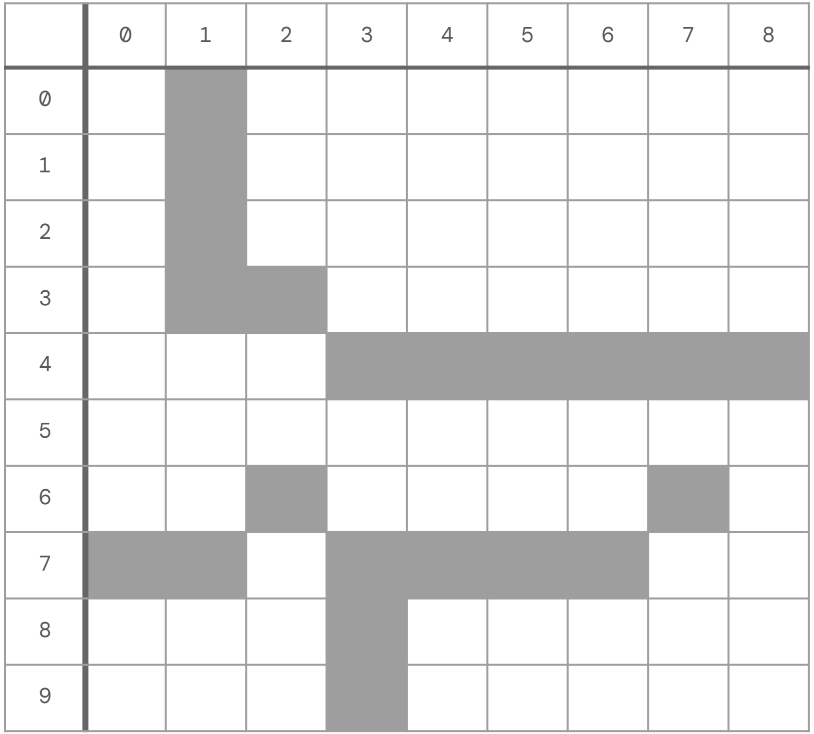
edges are often implicitly given

e.g. in a maze you could only go up/down/left/right

→ edge between each grid and all grids it can visit

*in practice, usually touching edges (and corners)*

assume that every node has these edges
→ check if these edges are valid as we traverse the graph (e.g. the grid is blocked?)




▨ blocked

# Grid graphs - Implementation

1. get neighbors
2. scan all the directions
3. check if the vertex is out-of-bound or invalid

```
N = 3 M = 4
..#.
.#..
....
Neighbors of (1, 2) :(2, 2) (1, 3)
Neighbors of (0, 3) :(1, 3)
```

```
const int MAX_N = 10;
char grid[MAX_N][MAX_N];
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, 1, -1};

void print_neighbors(int r, int c) {
    cout << "Neighbors of (" << r <<  ", " << c << "):";
    for (int i = 0; i < 4; i++){
        int nx = dx[i] + r;
        int ny = dy[i] + c;
        bool out_of_bound = nx < 0 || ny < 0 || nx >= N
             || ny >= M;
        bool valid = (grid[nx][ny] == '.');

        if (!out_of_bound && valid) {
            cout << "(" << nx <<  ", " << ny << ") ";
        }
    }
    cout << "\n";
}
```

# Graph Traversal

# Depth-First Search

Idea:

1. Choose an unvisited path
2. Go as far as possible
3. Go back
4. Explore a new path

Repeat until there is no unvisited path left.

**Time complexity**

- $O(|V|^2)$ for adjacency matrix
- $O(|V|+|E|)$ for adjacency list

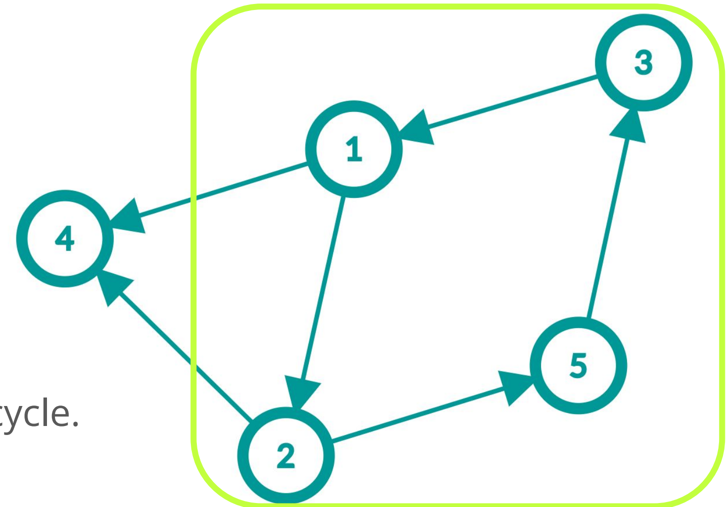
# Depth-First Search

There can be cycles - there exists a path that starts from and ends at the same vertex.

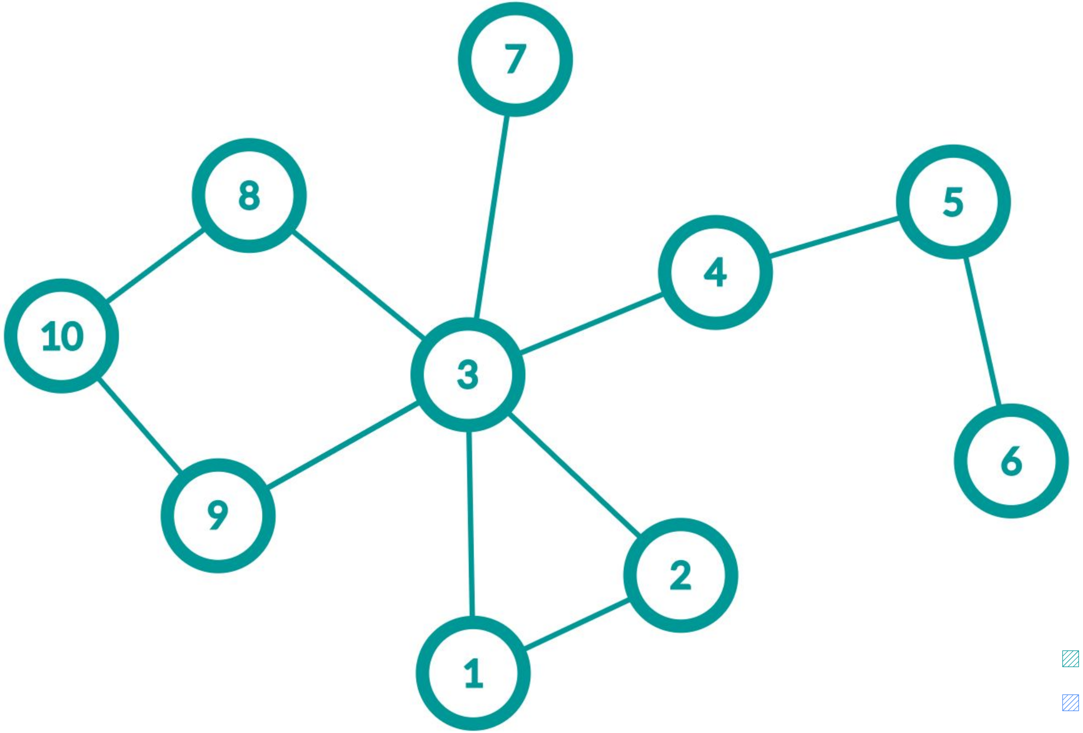
Recall that we do not go on a visited path, so we can stop going forward and treat the current vertex as a 'dead end'

In this case, not all edges are used
but all vertices are travelled

{1, 2, 3, 5} forms a cycle.

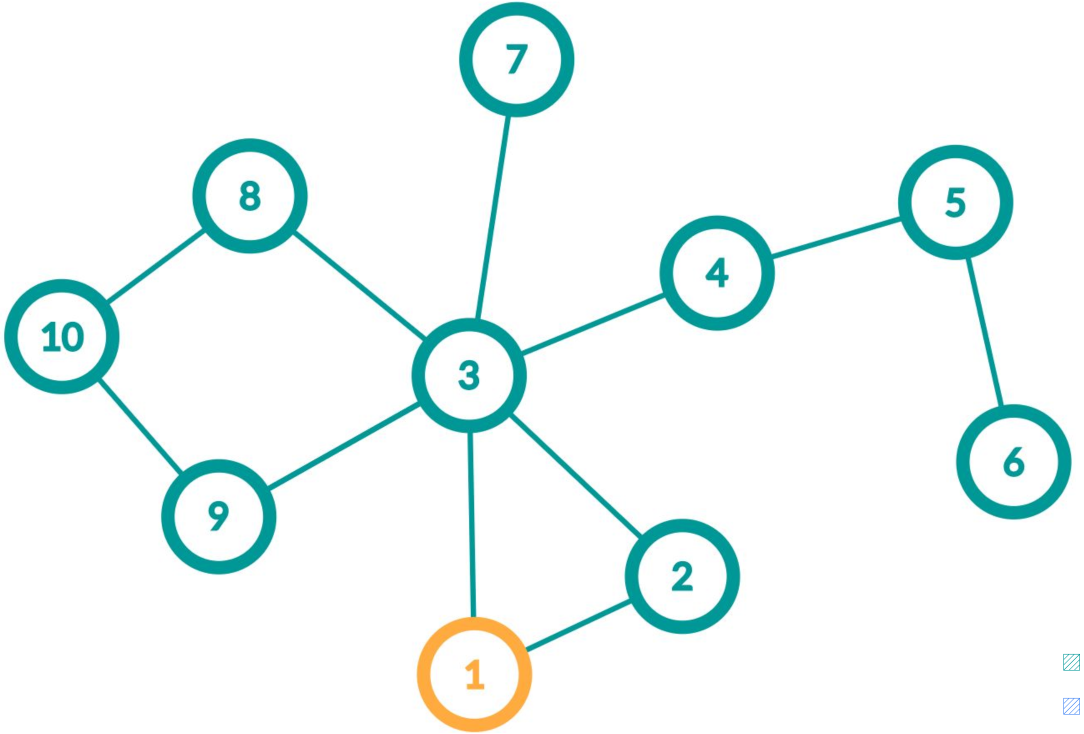


| |
|---|
| |
| |
| |
| |
| |
| |
| |
| |
| Function call |



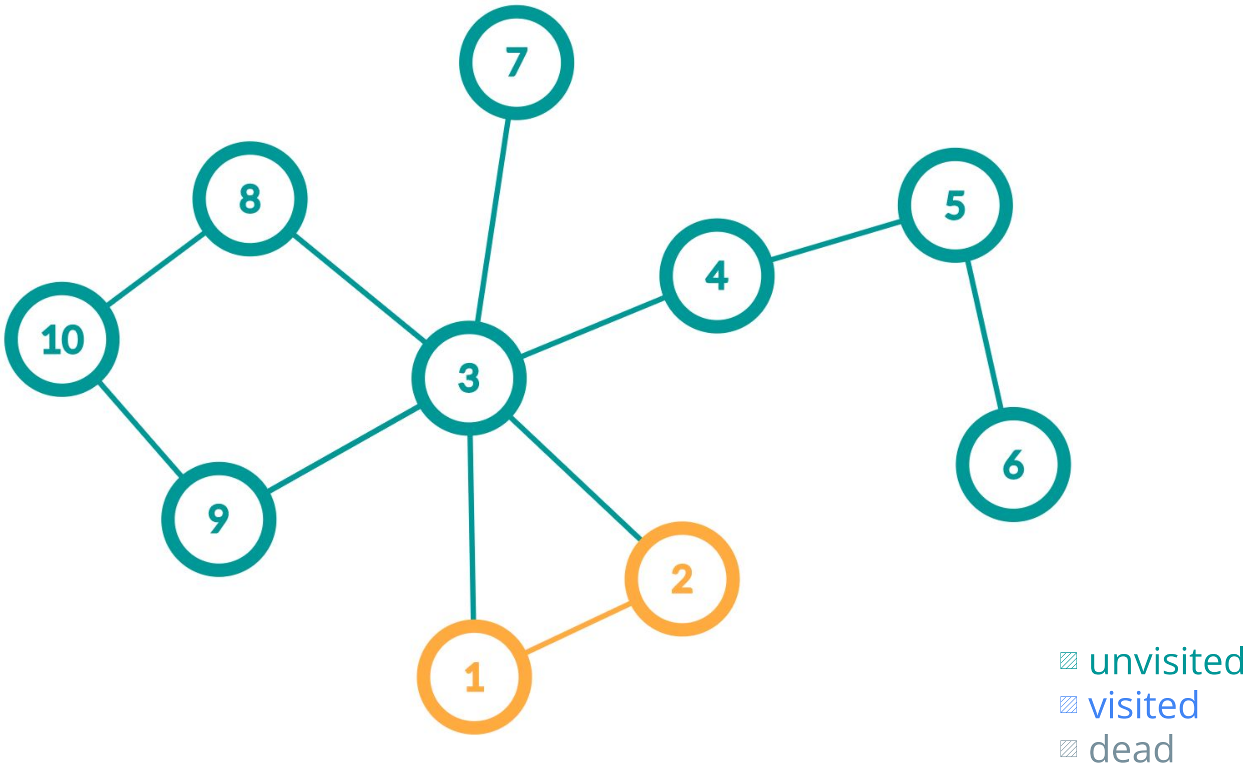

▨ unvisited
▨ visited
▨ dead

| |
|---|
| |
| |
| |
| |
| |
| |
| |
| DFS(1) |
| Function call |

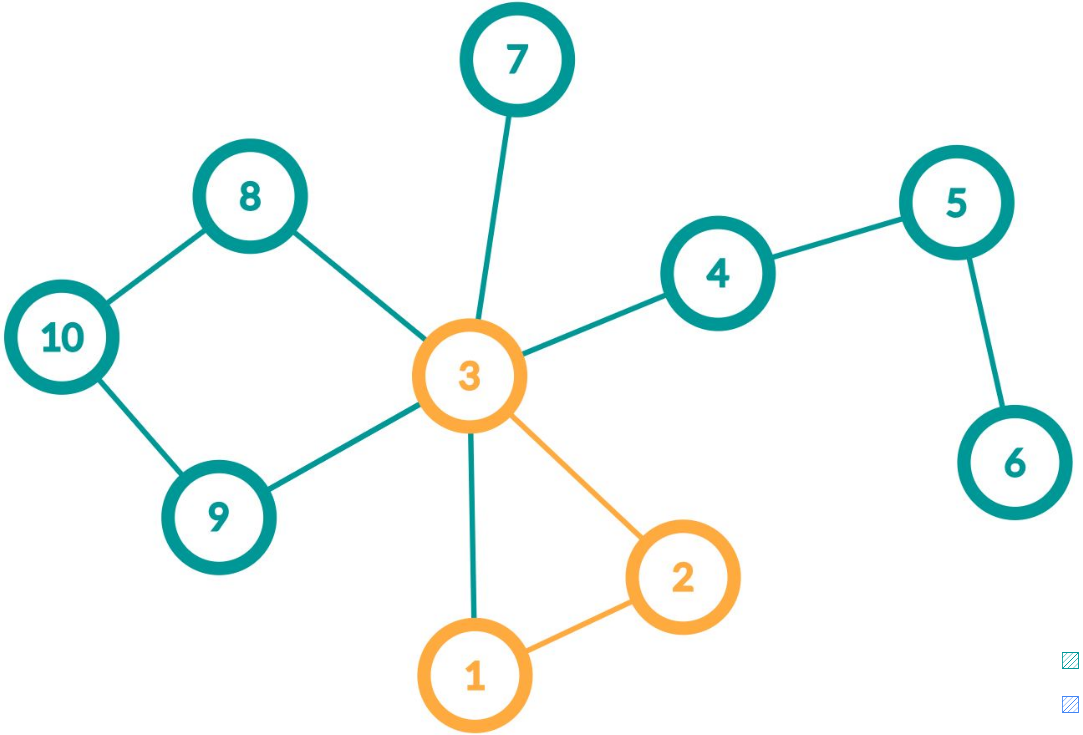



unvisited
visited
dead

| |
|---|
| |
| |
| |
| |
| |
| |
| DFS(2) |
| DFS(1) |
| Function call |

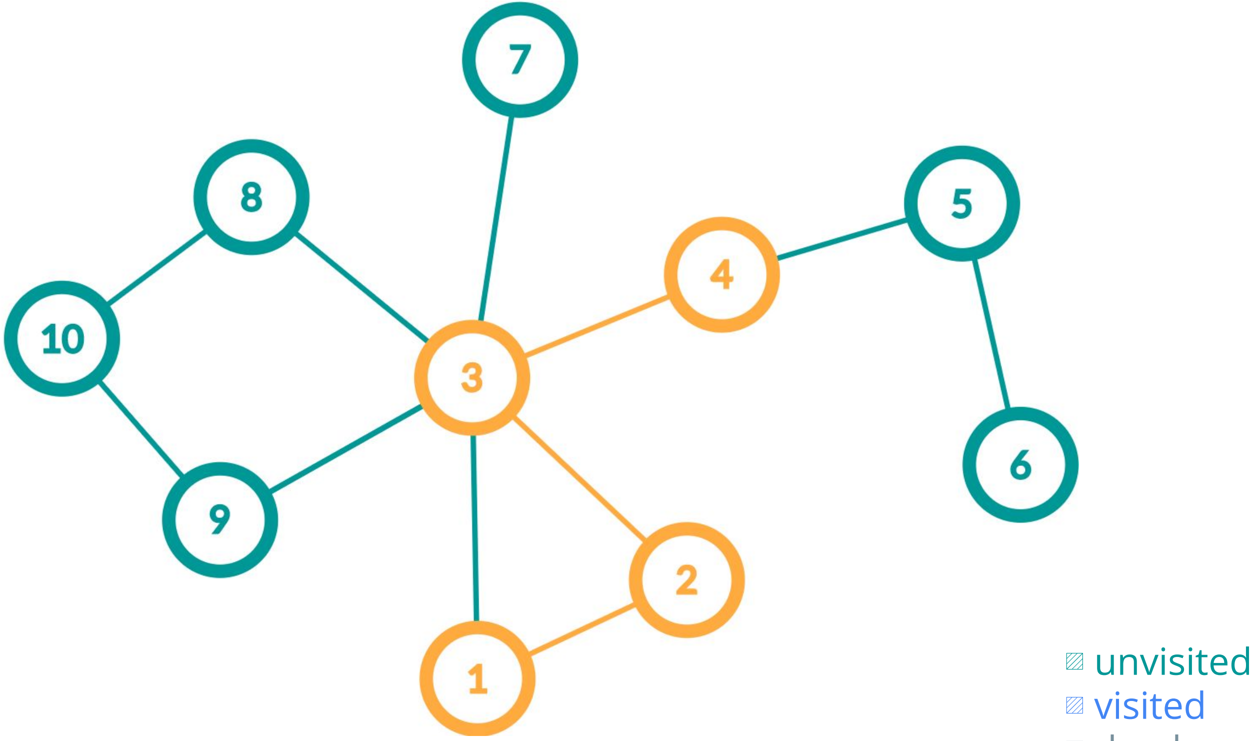


| |
|---|
| |
| |
| |
| |
| |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |

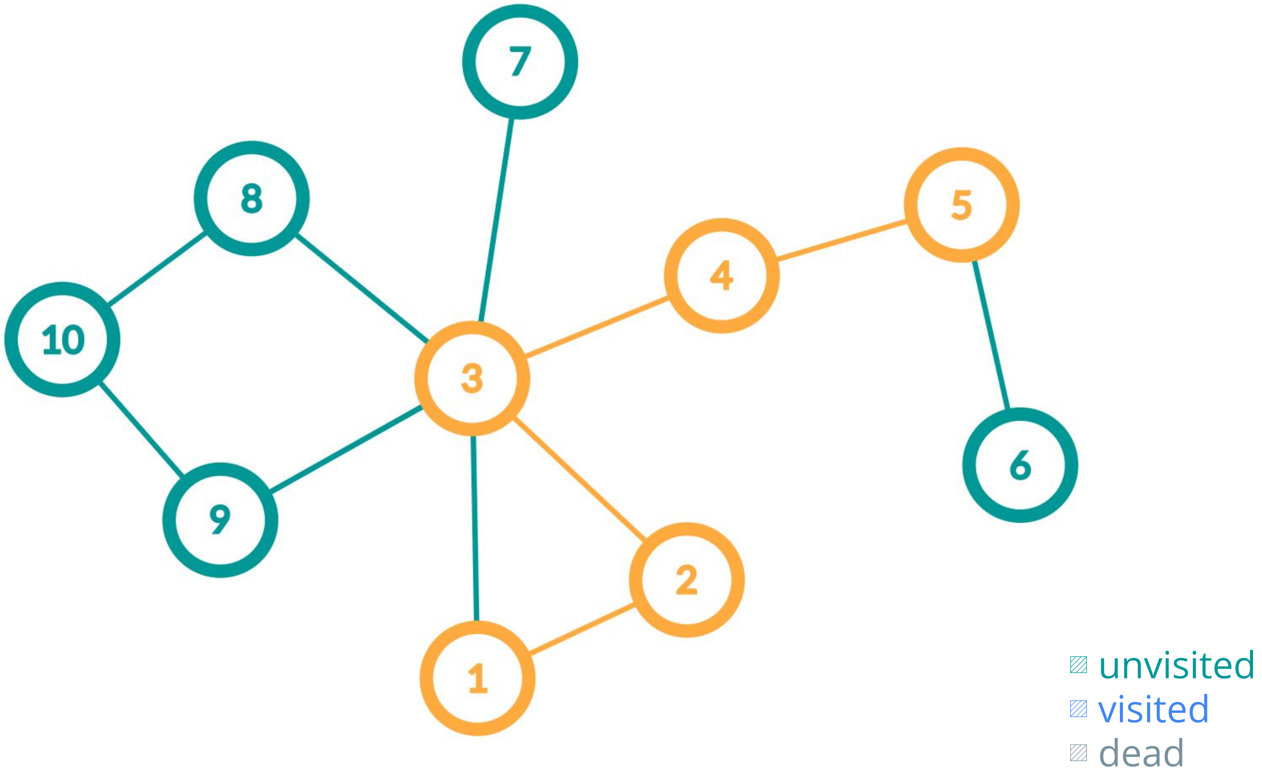



unvisited
visited
dead

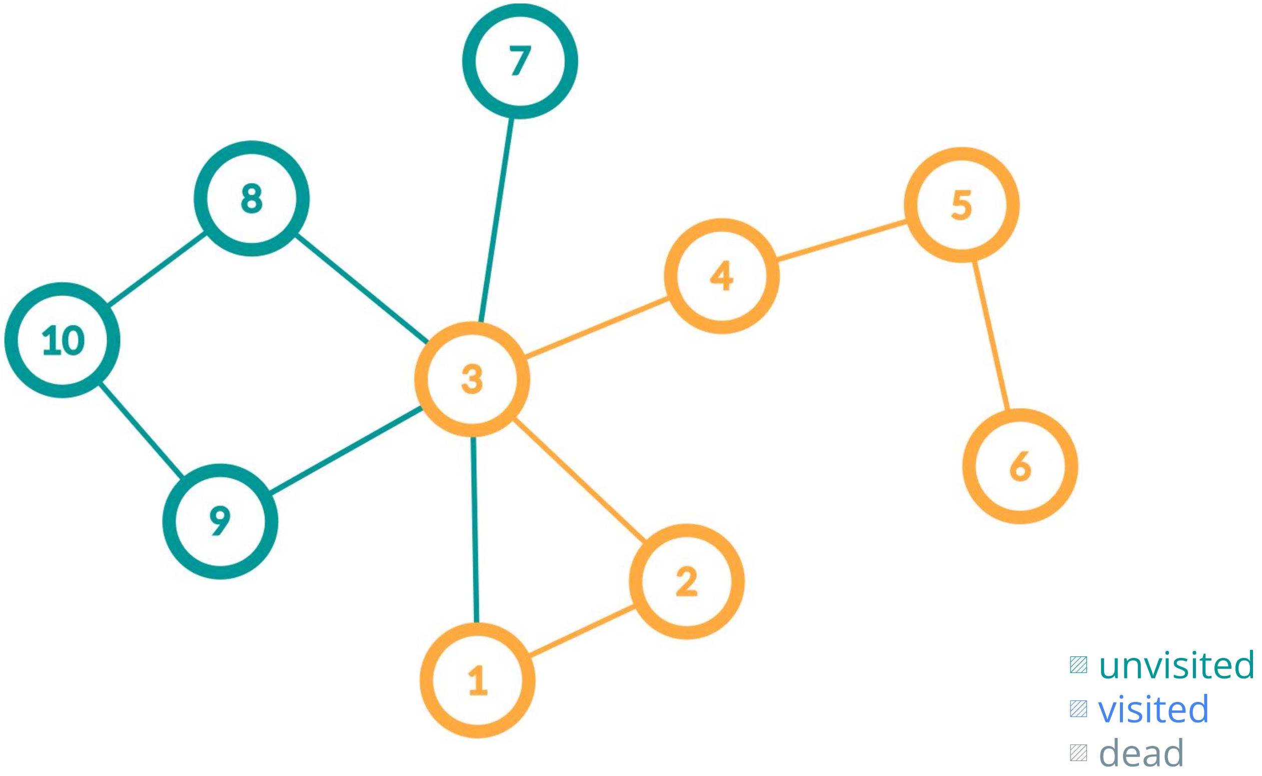
| |
|---|
| |
| |
| |
| |
| DFS(4) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



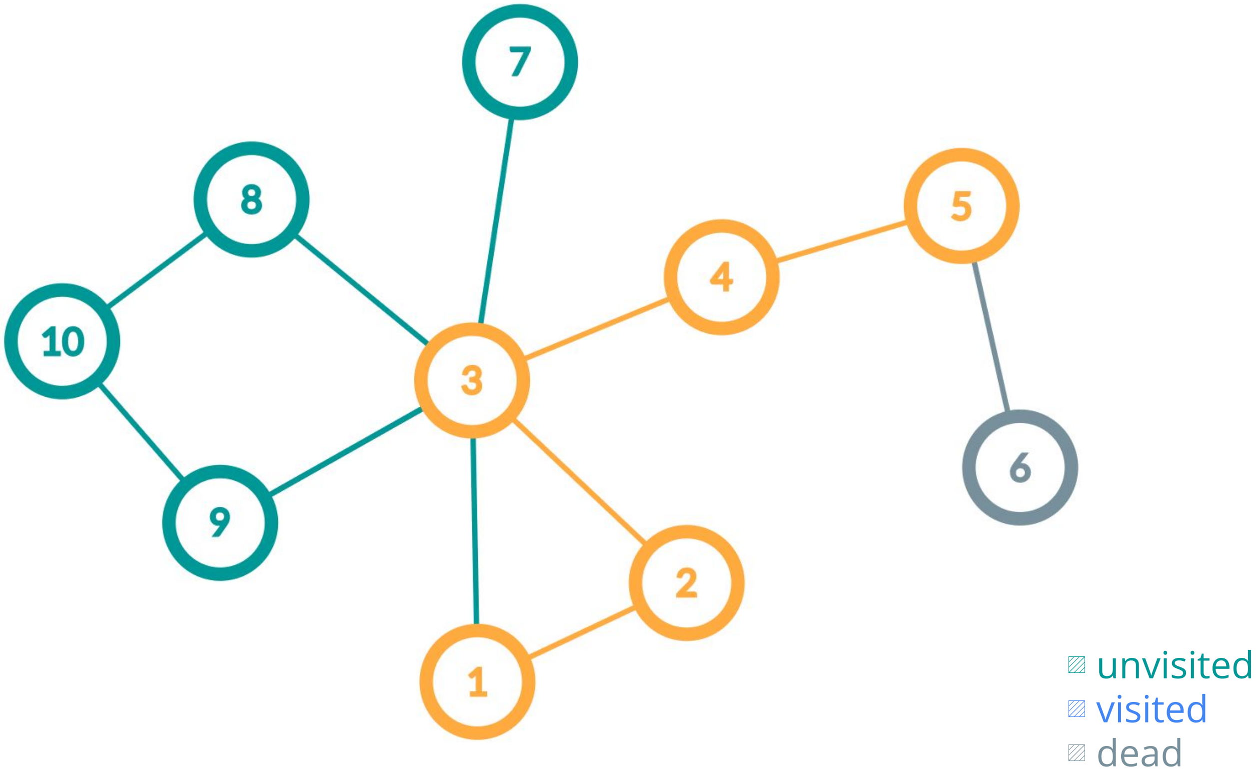
| |
|---|
| |
| |
| |
| DFS(5) |
| DFS(4) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



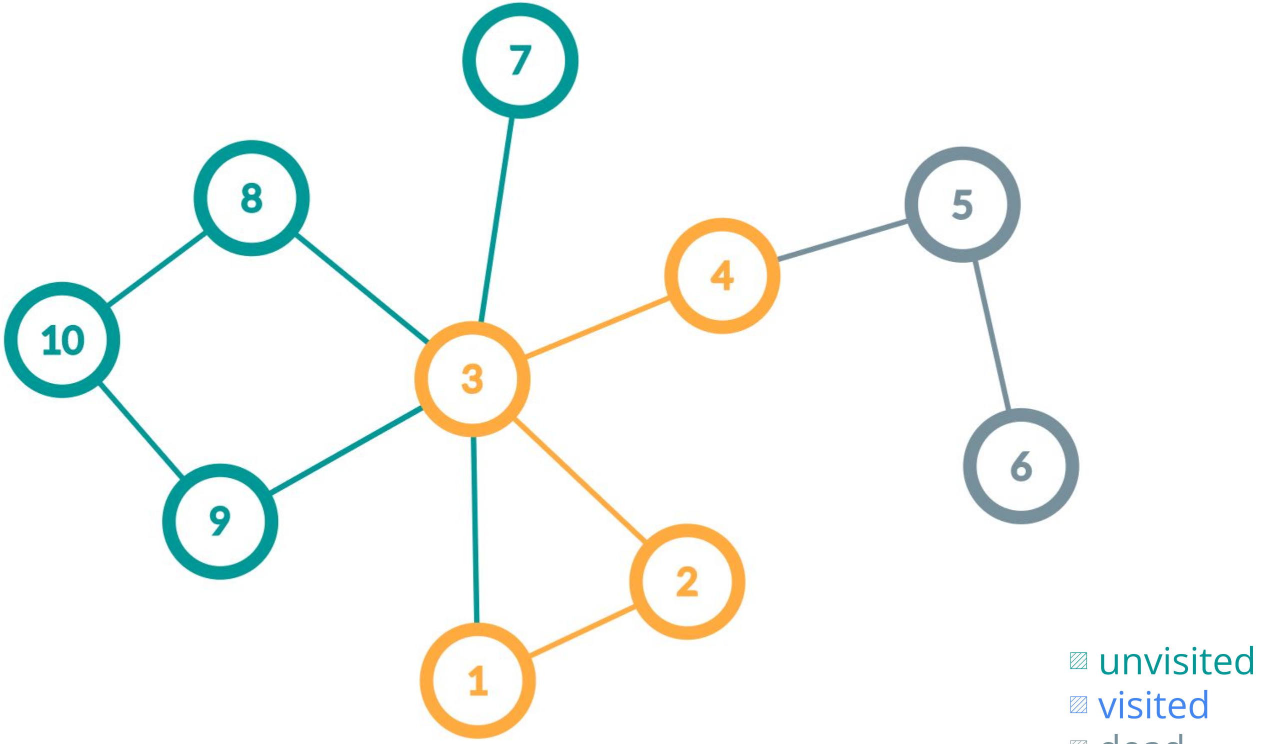
| |
|---|
| |
| |
| DFS(6) |
| DFS(5) |
| DFS(4) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



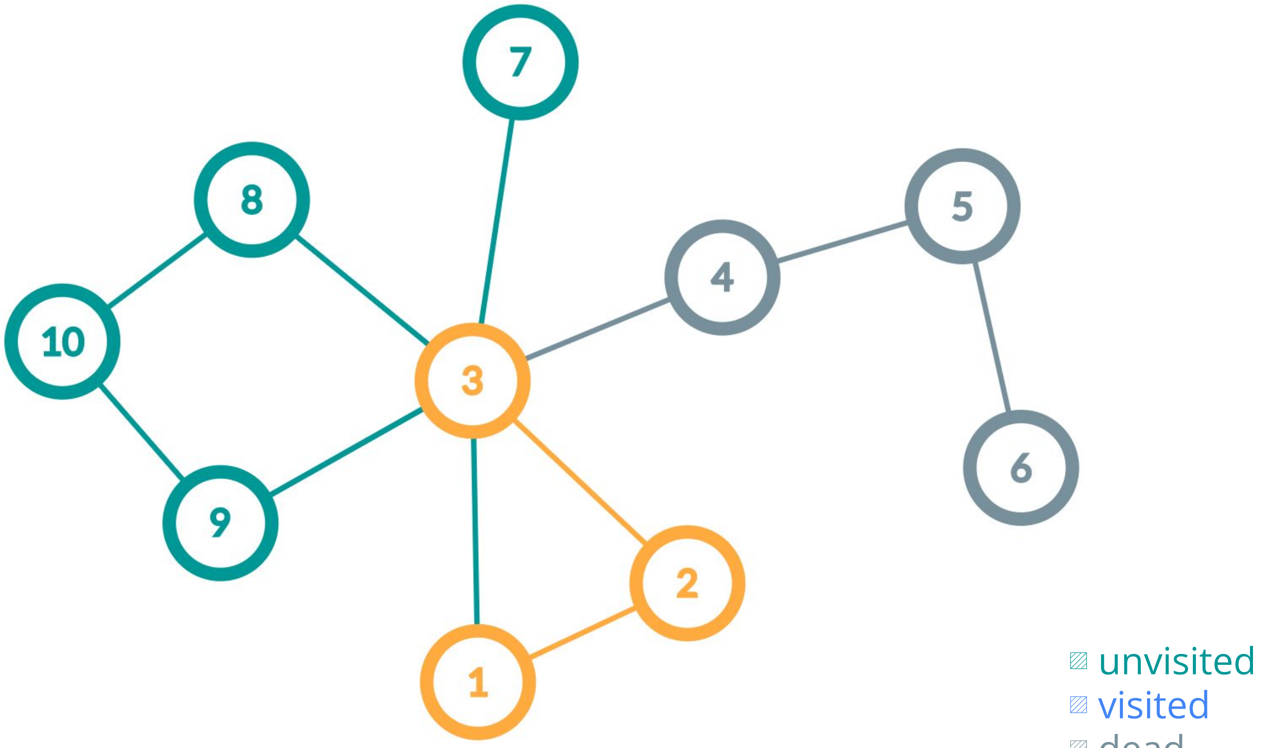
| |
|---|
| |
| |
| |
| DFS(5) |
| DFS(4) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



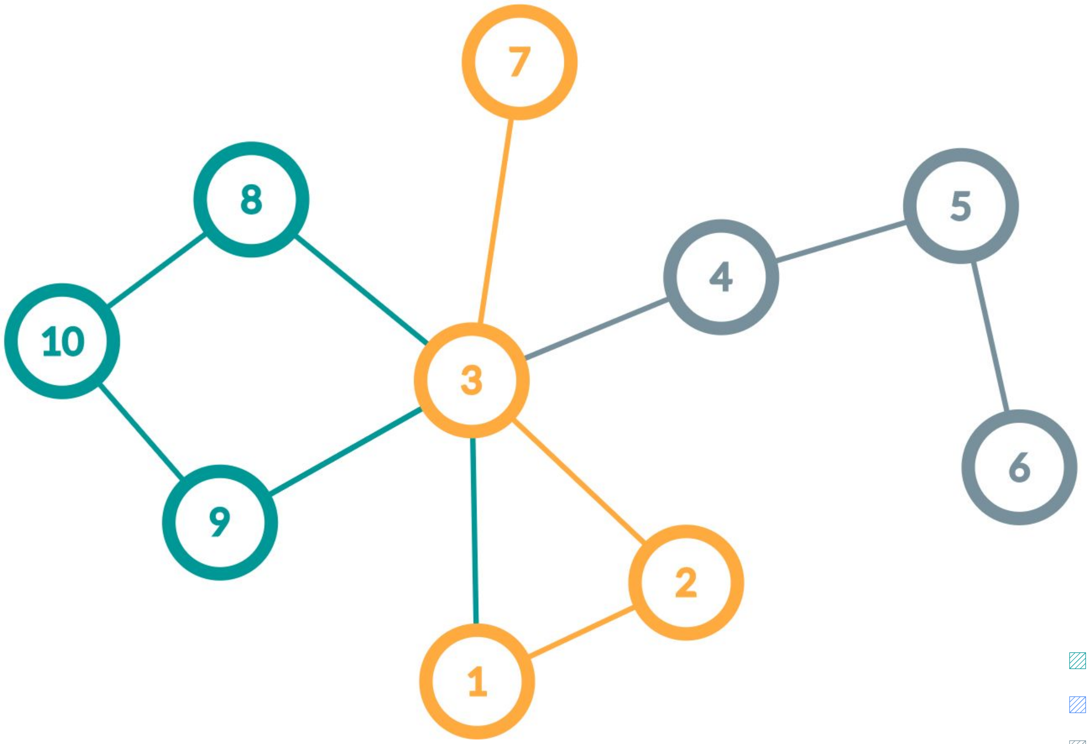
| |
|---|
| |
| |
| |
| |
| DFS(4) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



| |
|---|
| |
| |
| |
| |
| |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |

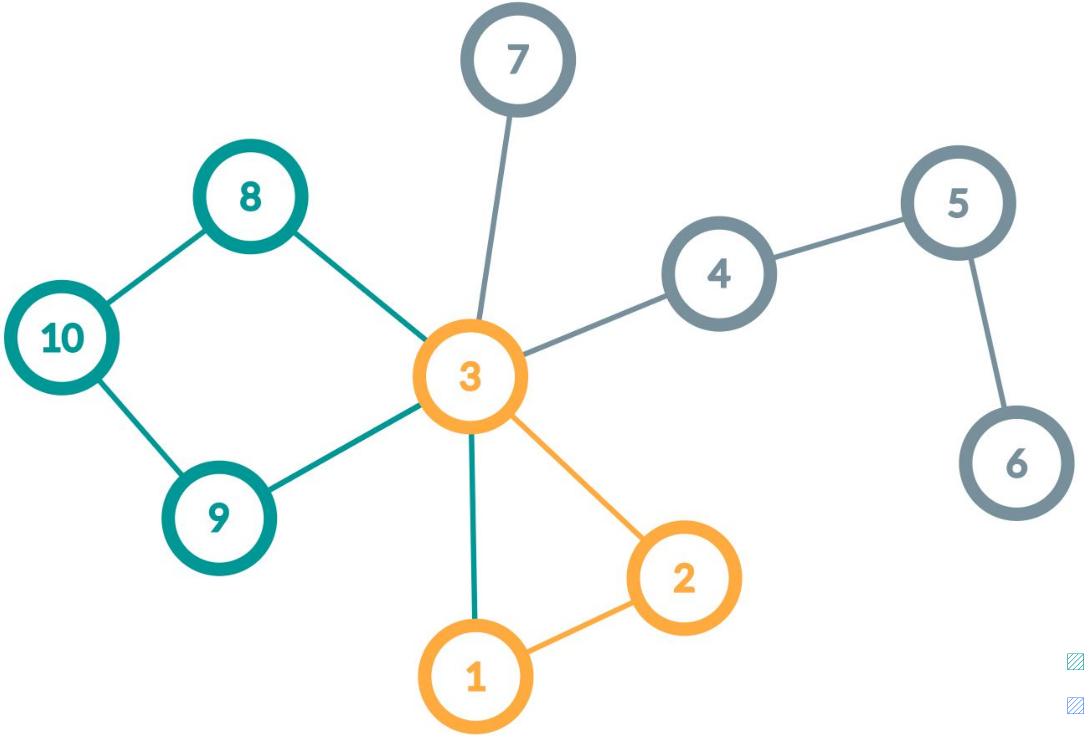


| |
|---|
| |
| |
| |
| |
| DFS(7) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



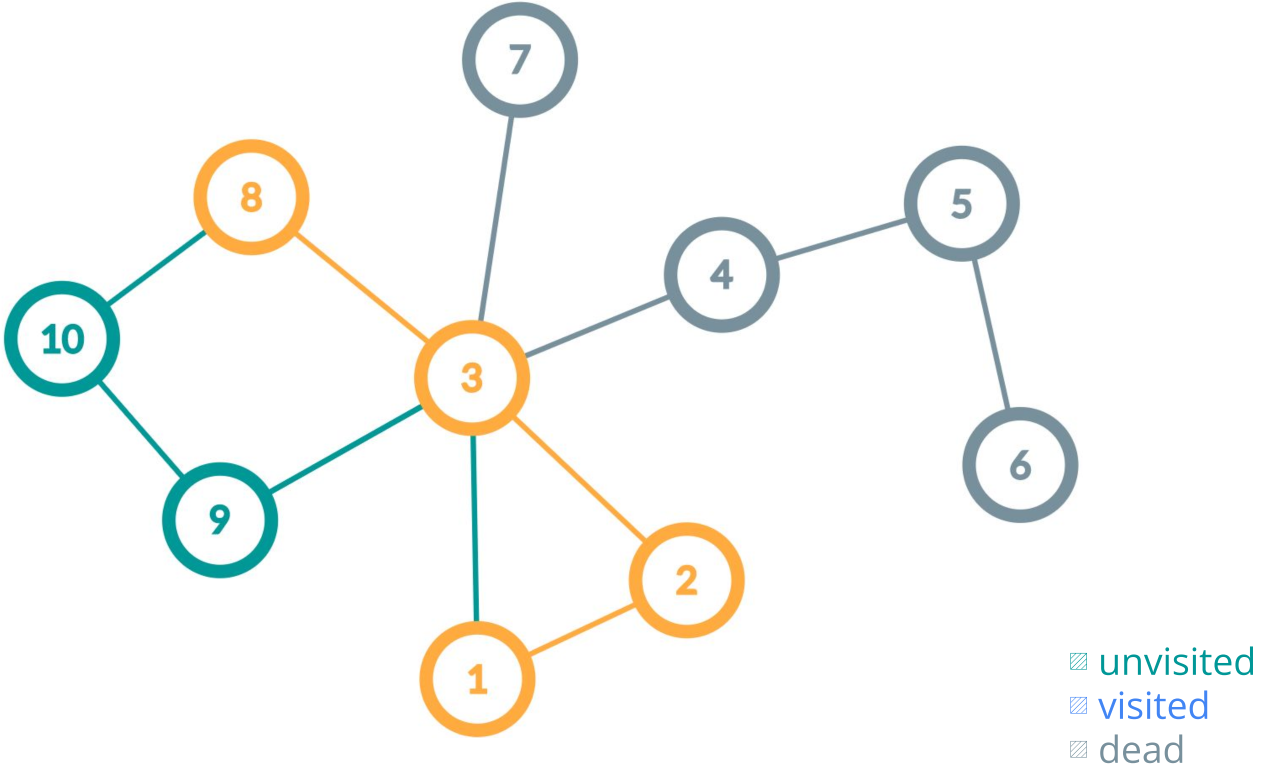

unvisited
visited
dead

| |
|---|
| |
| |
| |
| |
| |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |

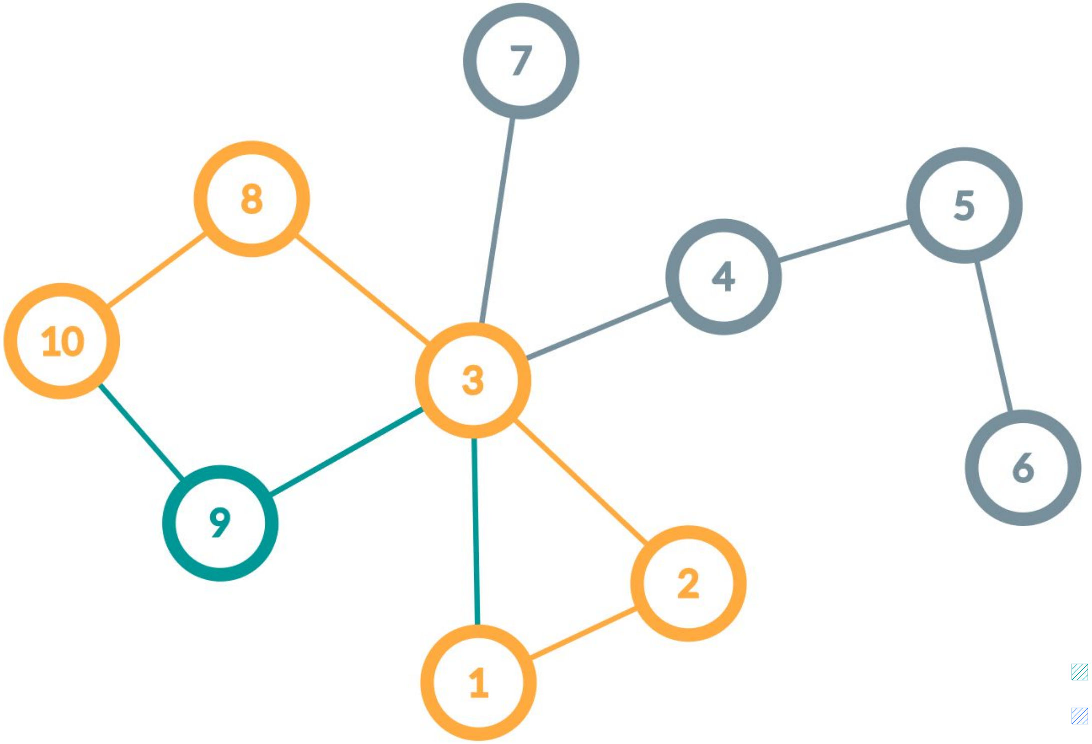



unvisited
visited
dead

| |
|---|
| |
| |
| |
| |
| DFS(8) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |

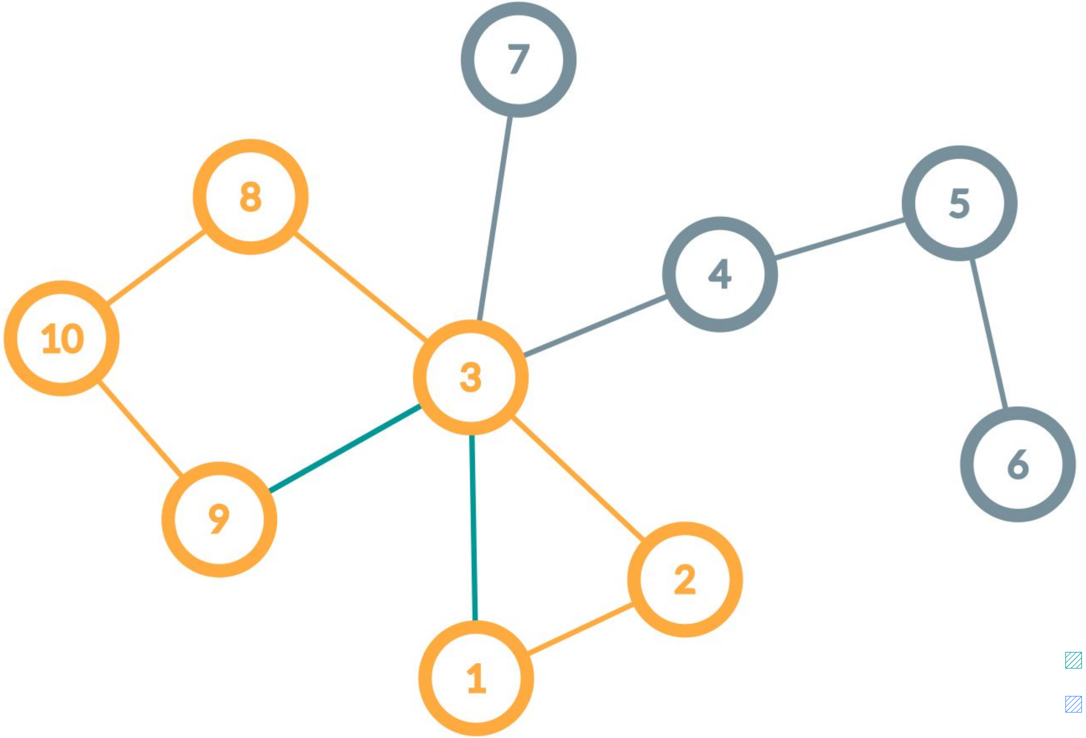


| |
|---|
| |
| |
| |
| DFS(10) |
| DFS(8) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



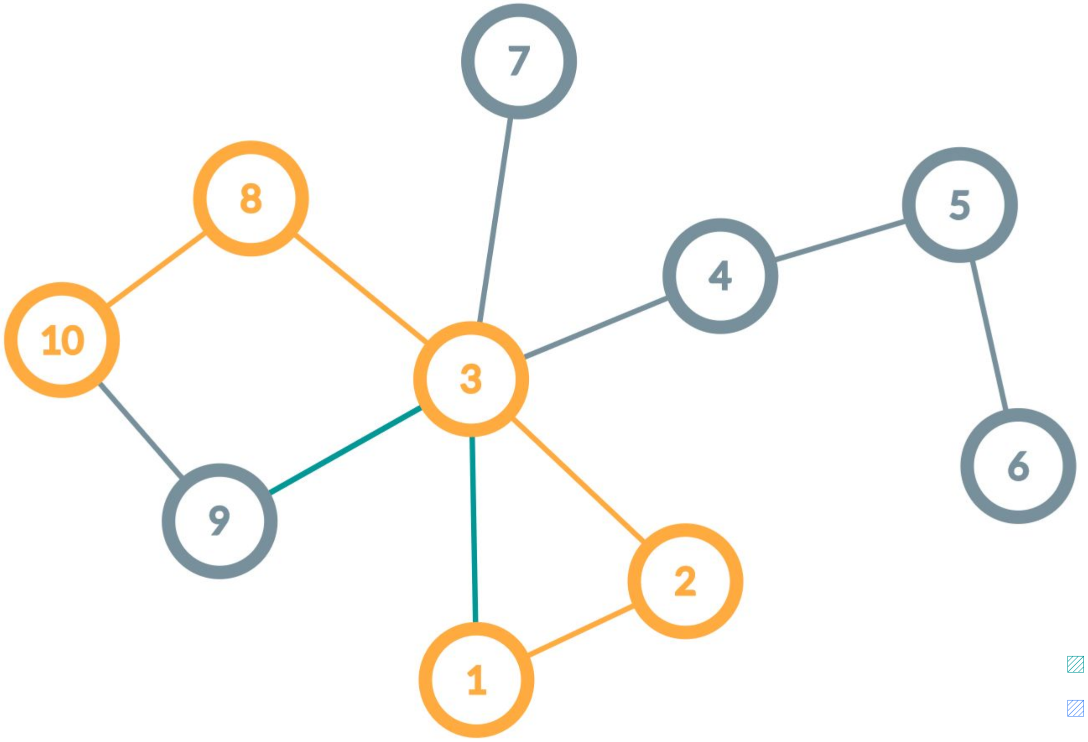

- unvisited
- visited
- dead

| |
|---|
| |
| |
| DFS(9) |
| DFS(10) |
| DFS(8) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



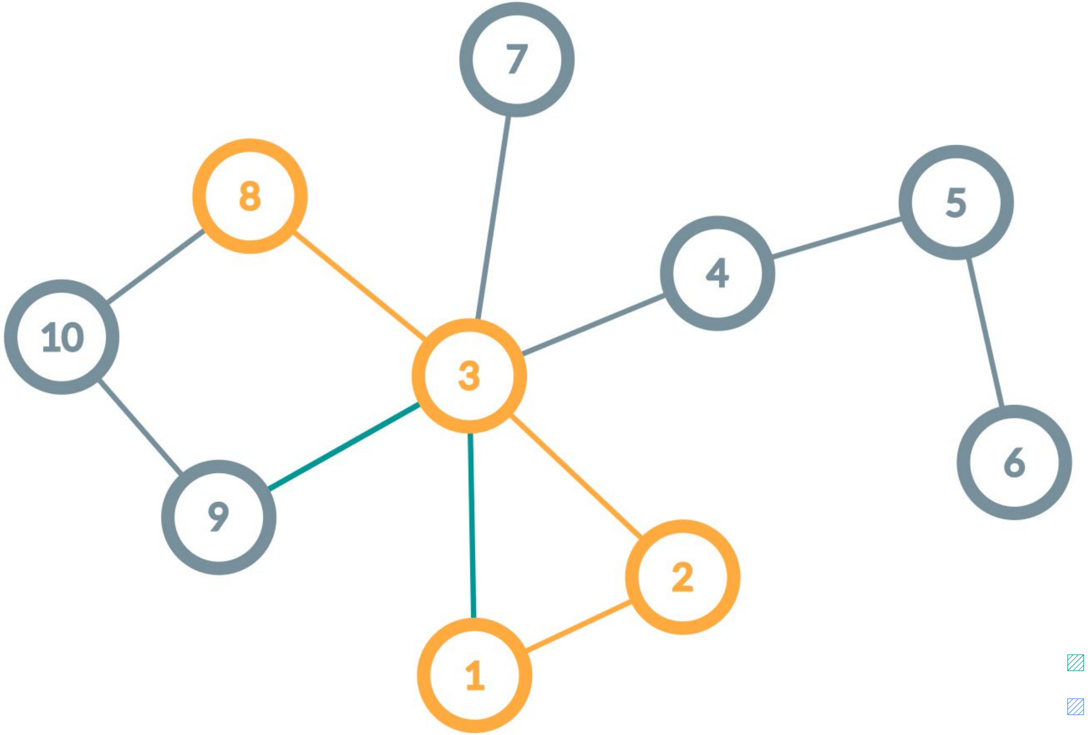

unvisited
visited
dead

| |
|---|
| |
| |
| |
| DFS(10) |
| DFS(8) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |



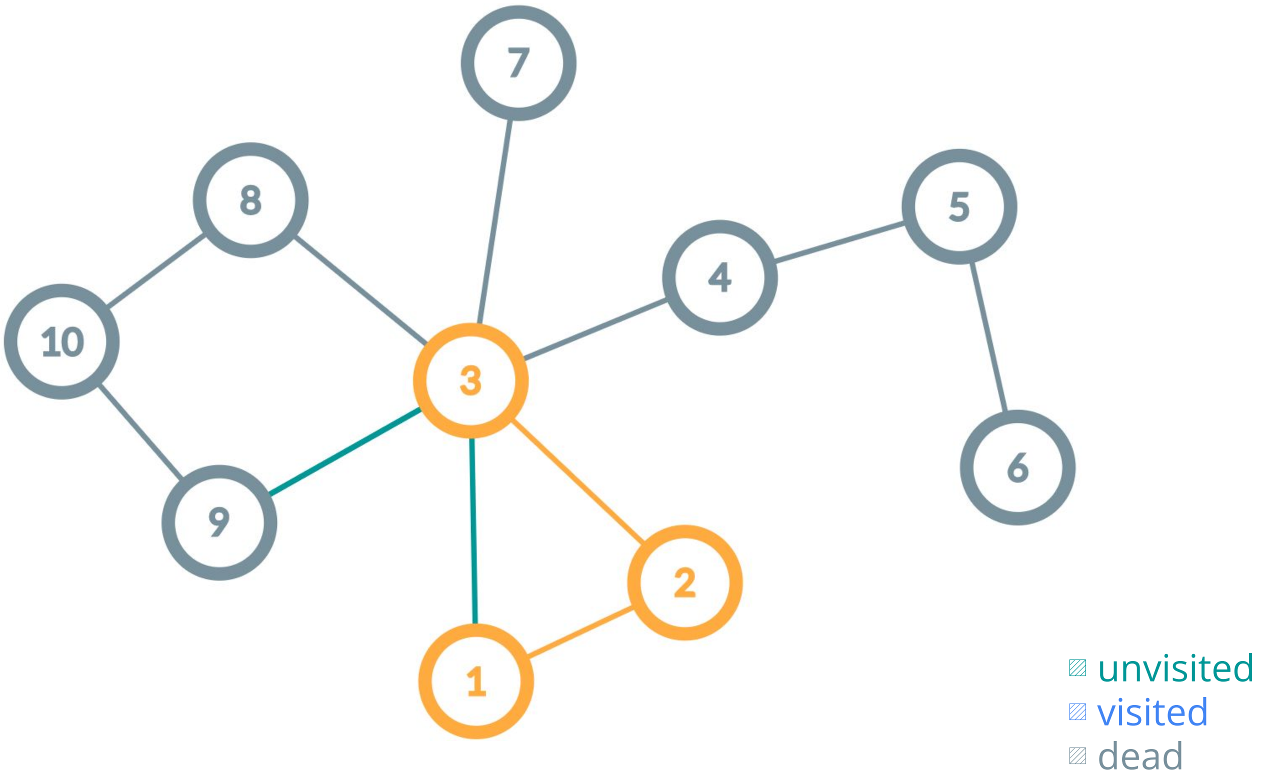

unvisited
visited
dead

| |
|---|
| |
| |
| |
| |
| DFS(8) |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |

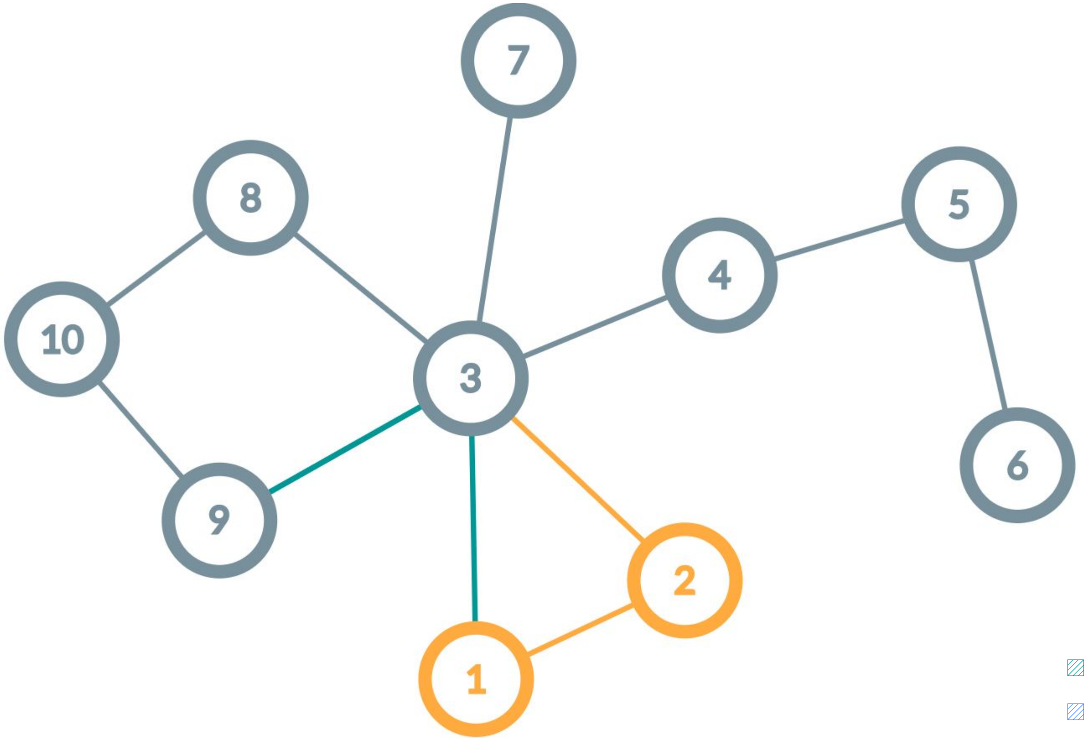



- unvisited
- visited
- dead

| |
|---|
| |
| |
| |
| |
| |
| DFS(3) |
| DFS(2) |
| DFS(1) |
| Function call |

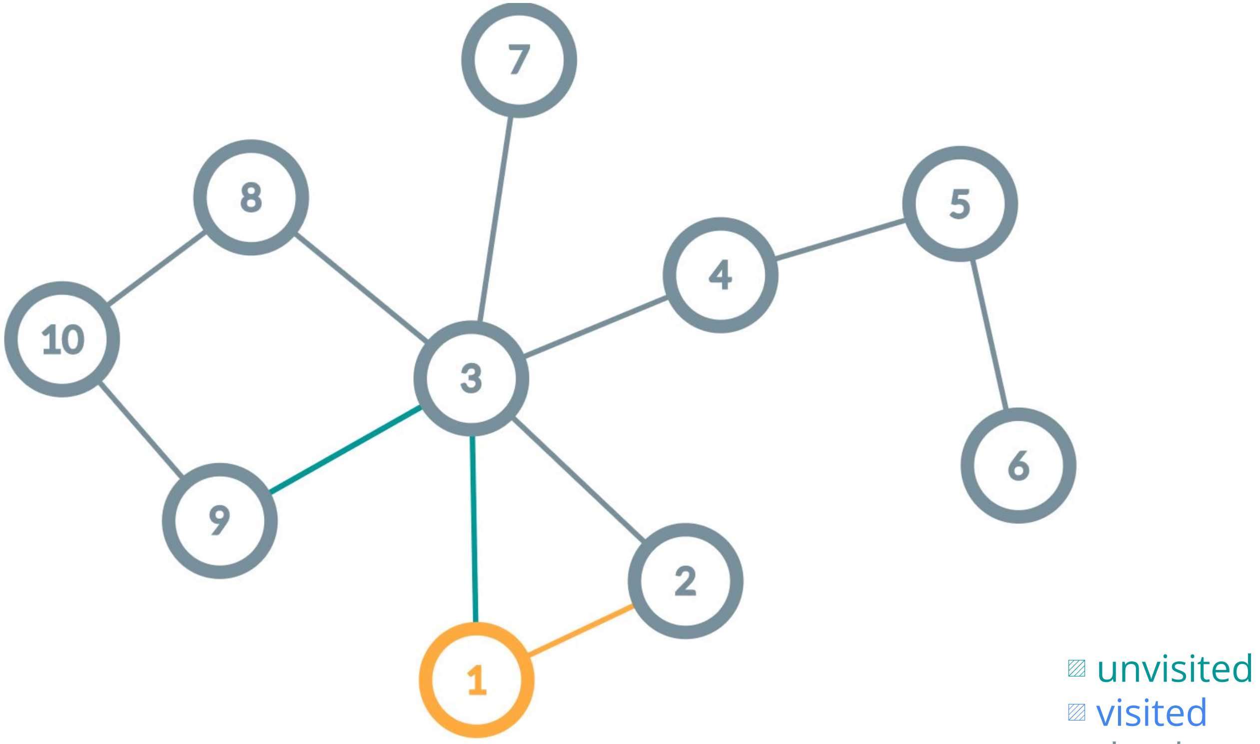


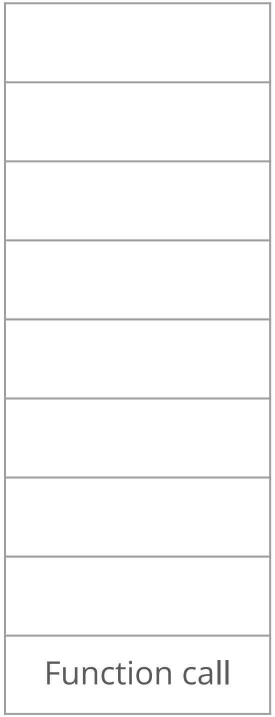
| |
|---|
| |
| |
| |
| |
| |
| |
| DFS(2) |
| DFS(1) |
| Function call |

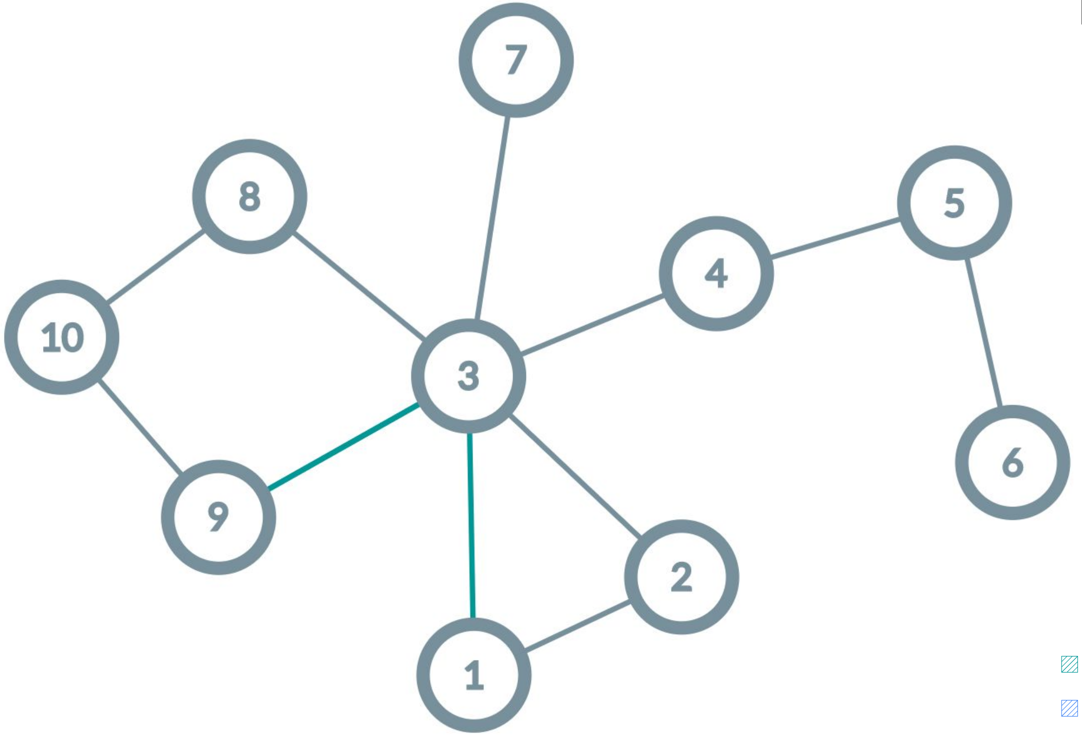



unvisited
visited
dead

| |
|---|
| |
| |
| |
| |
| |
| |
| |
| DFS(1) |
| Function call |






DONE!!

- unvisited
- visited
- dead

# Depth-First Search

(1) Iterative (stack)

```
push source vertex (denote as u) into stack S
while (S is not empty)
    pop the top element (x) in S
    if x is not visited
        push all the unvisited vertices that are neighbors of x into S
        mark x as visited
```

# Depth-First Search

(2) Recursive *(more common)*

```
Procedure DFS(vertex x)
    mark x as visited
    For all vertices that are neighbors of x and are unvisited (v)
        Call DFS(v)
```

`To start the search, call DFS(u) (source vertex)`

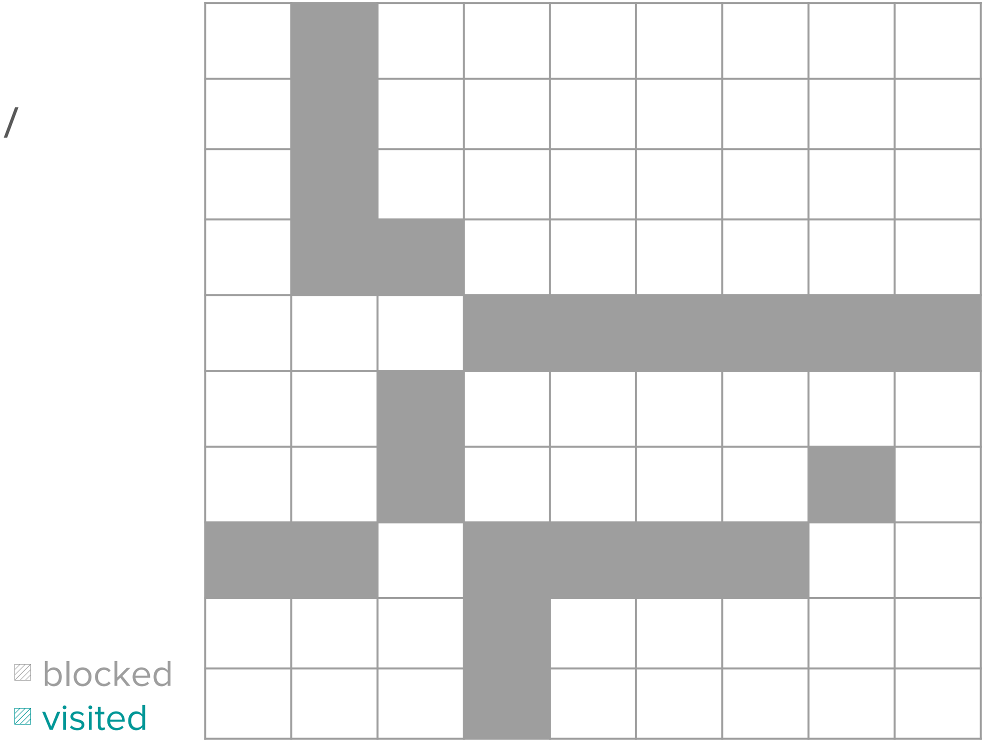
# Flood Fill - DFS

From a empty cell, you can go up/down/left/right to reach another empty cell.

Mutually reachable cells form a region.

How many regions are there?

How large is each region?

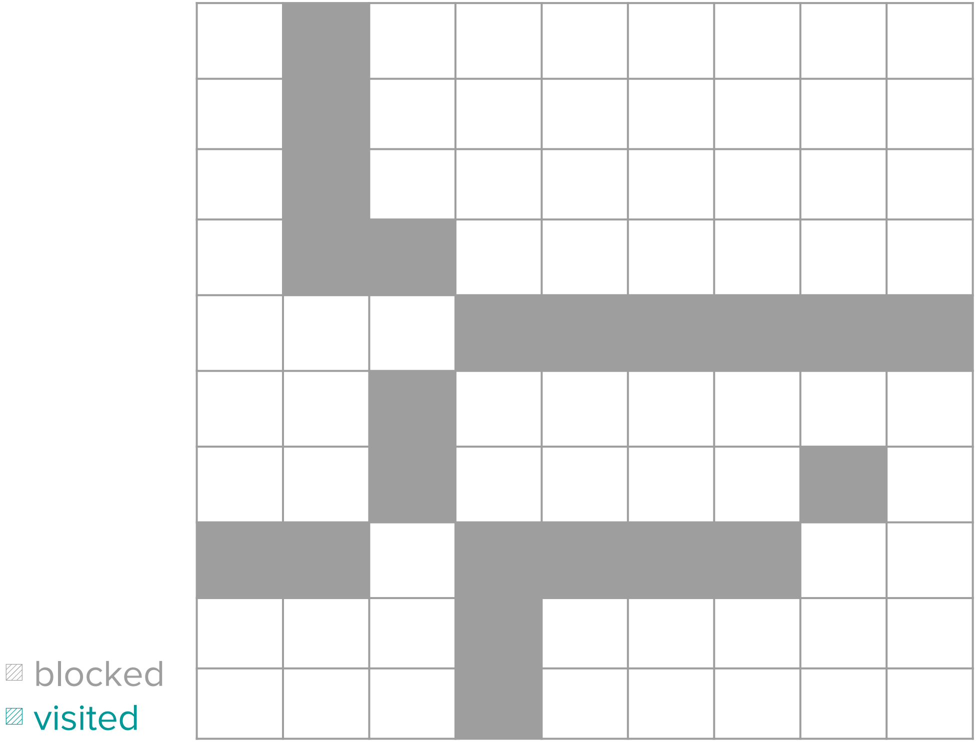


# Flood Fill - DFS

We name each cell by its coordinate (row, column)

For example, top-left cell: (1, 1)
bottom-right: (10, 9)

When we detect an unvisited empty cell, we run DFS on it and count the number of cells in this region.

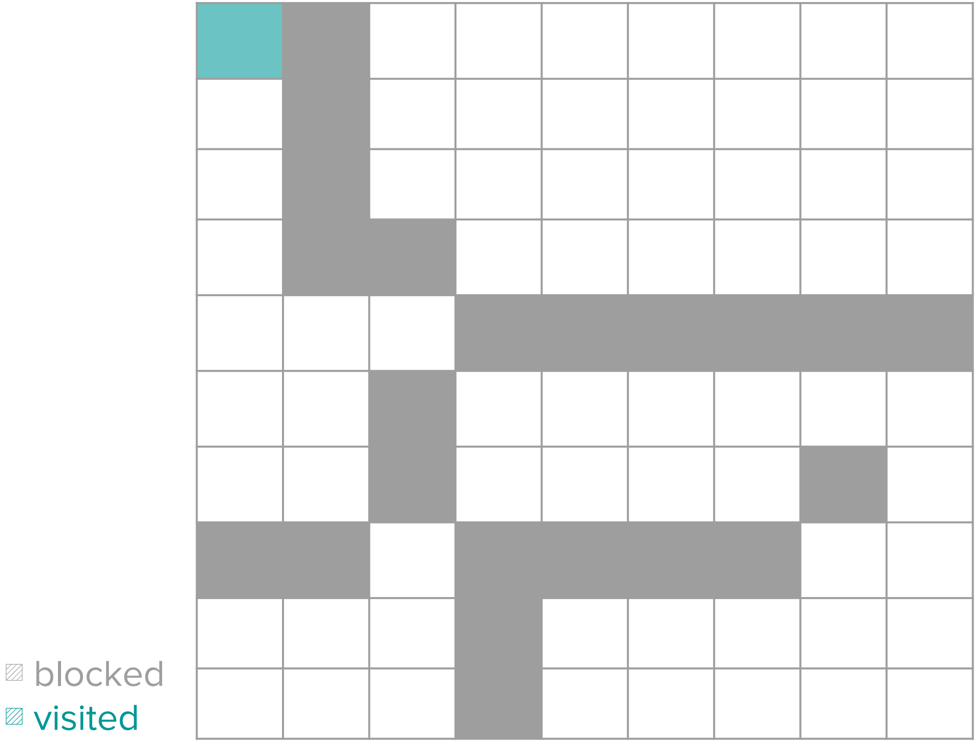


# Flood Fill - DFS

We name each cell by its coordinate (row, column)

For example, top-left cell: (1, 1)
bottom-right: (10, 9)

When we detect an unvisited empty cell, we run DFS on it and count the number of cells in this region.

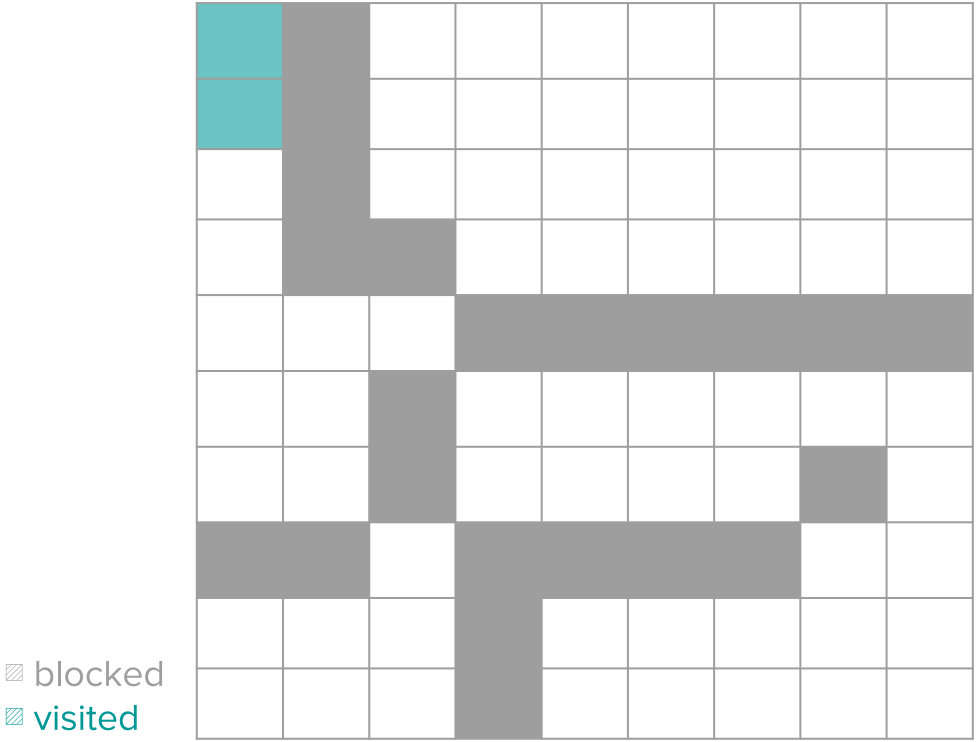


# Flood Fill - DFS

We name each cell by its coordinate (row, column)

For example, top-left cell: (1, 1)
bottom-right: (10, 9)

When we detect an unvisited empty cell, we run DFS on it and count the number of cells in this region.

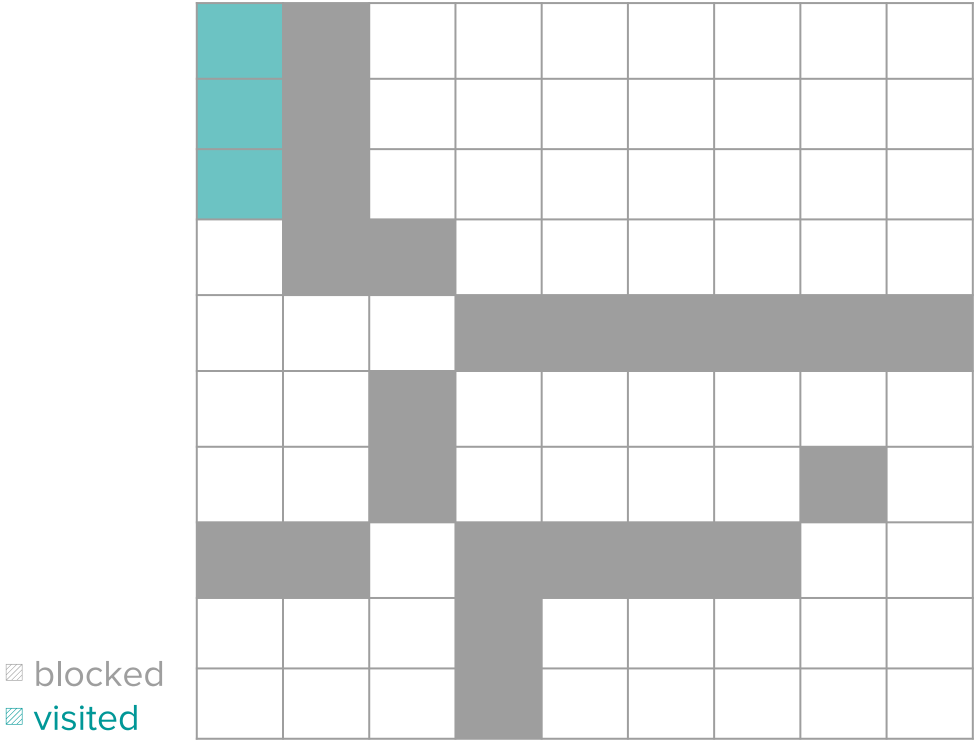


# Flood Fill - DFS

We name each cell by its coordinate (row, column)

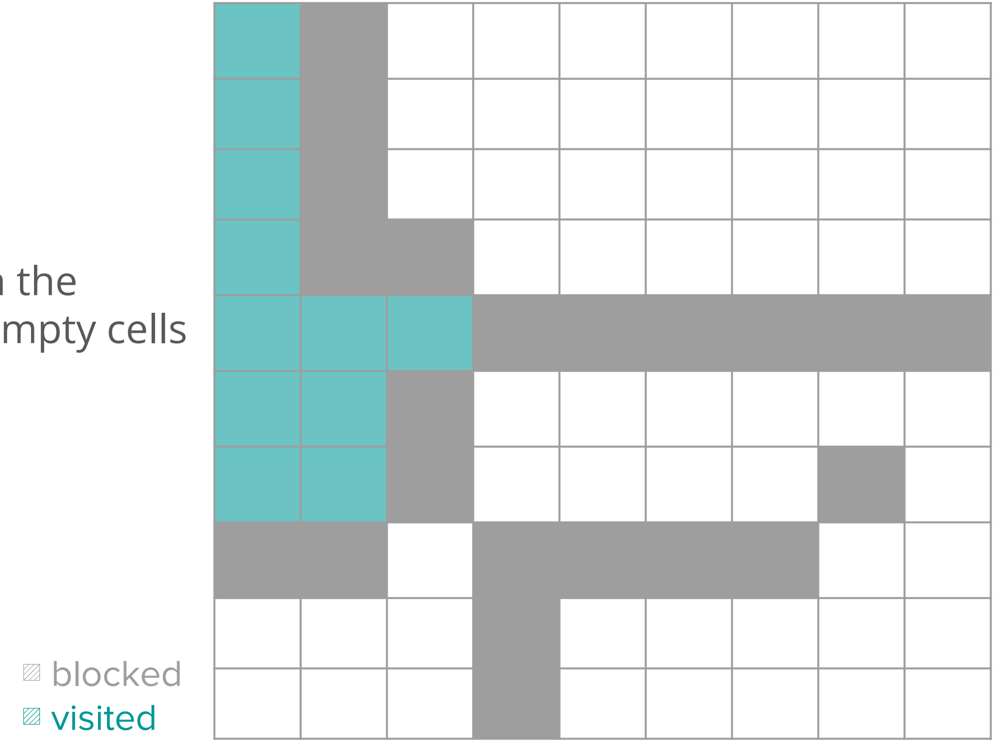
For example, top-left cell: (1, 1)
bottom-right: (10, 9)

When we detect an unvisited empty cell, we run DFS on it and count the number of cells in this region.



# Flood Fill - DFS

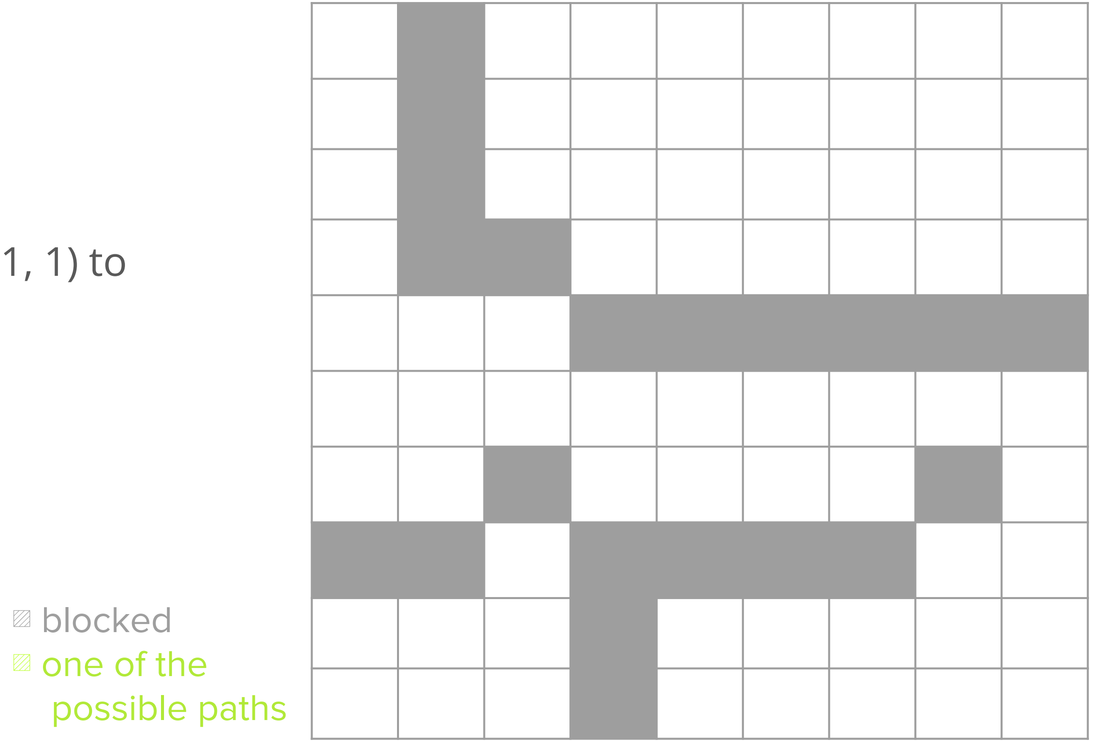
After visiting a region, we perform DFS on the remaining unvisited empty cells until all empty cells are visited.



## Maze - DFS

Find a way from the top-left corner (1, 1) to the bottom-right corner (10, 9)

You can go up / down / left / right.

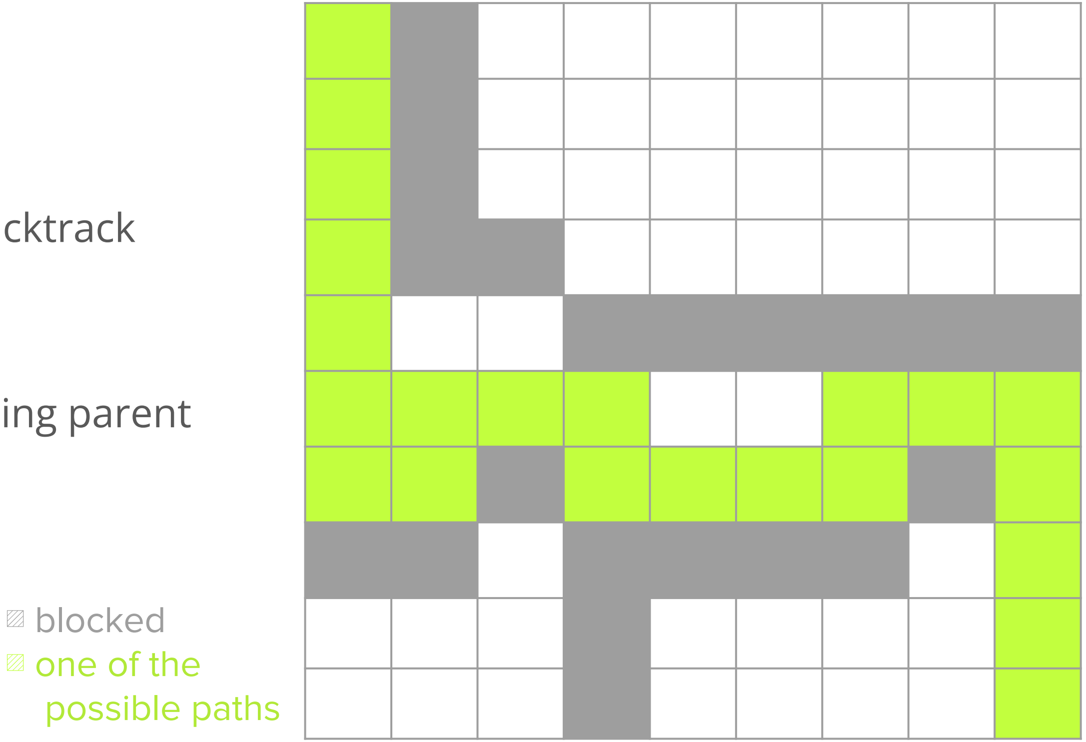


# Maze - DFS

Call DFS(1, 1)

When the destination is reached, backtrack the path.

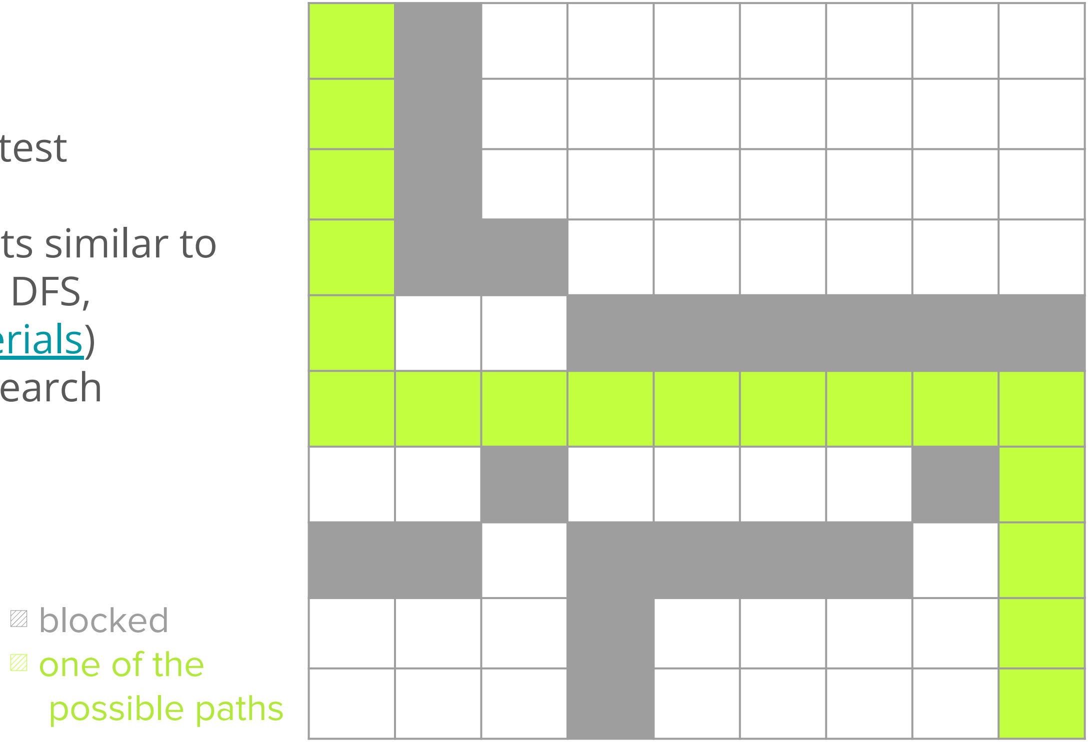
To backtrack, maintain a list containing parent of each nodes in the current path



# Maze - DFS

Note that the path **may not** be shortest

(To find shortest paths using concepts similar to DFS, read about Iterative Deepening DFS, it was covered in [2018 Graph II materials])
– Fixing depth of traversal for each search



# Breadth-First Search

Though Depth-First Search can provide us a path from *u* (source node) to *v* (any node reachable), it may not be the shortest path.

Given the graph is **unweighted**, we can use Breadth-First Search (BFS) to find the shortest path from *u* to *v*.

For weighted graph shortest path, refer to Graph (II).

# Breadth-First Search

Starting from the source vertex, we visit all reachable nodes using minimum of *i* steps.

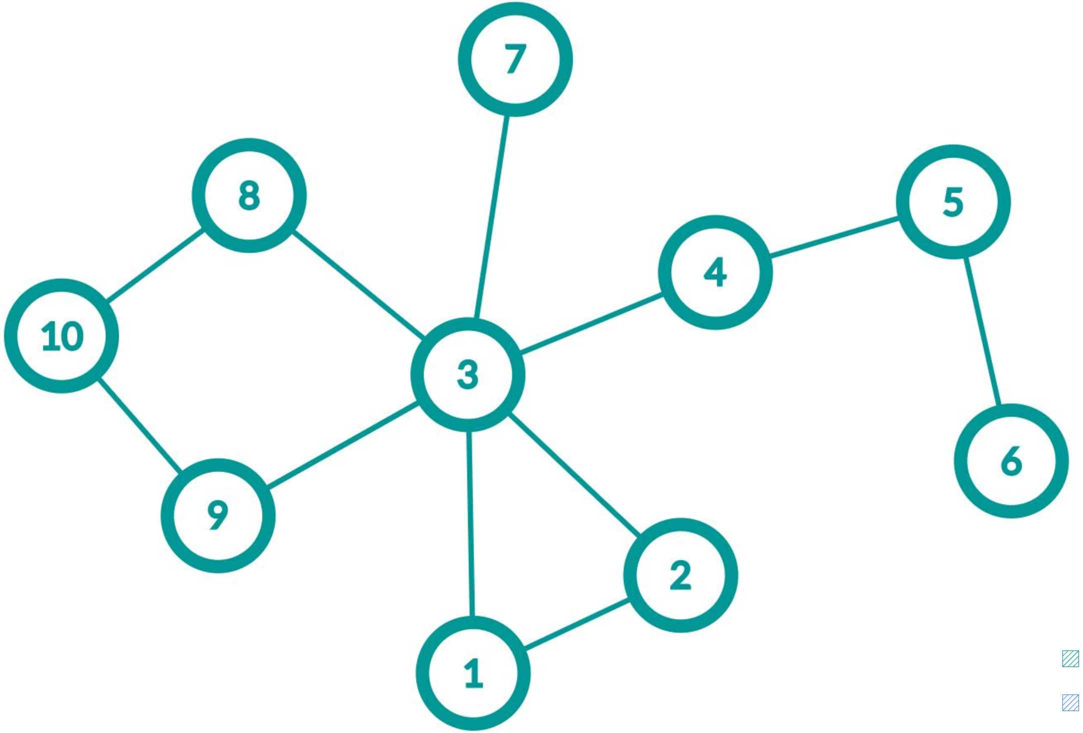
(1) Visit all the neighbors of *u* — {x : set of vertices that are 1-step reachable from *u*}

(2) Visit all the unvisited neighbors of {x} — {y : set of vertices that are 2-step reachable from *u*}

(3) Visit all the unvisited neighbors of {y} — {z : set of vertices that are 3-step reachable from *u*}

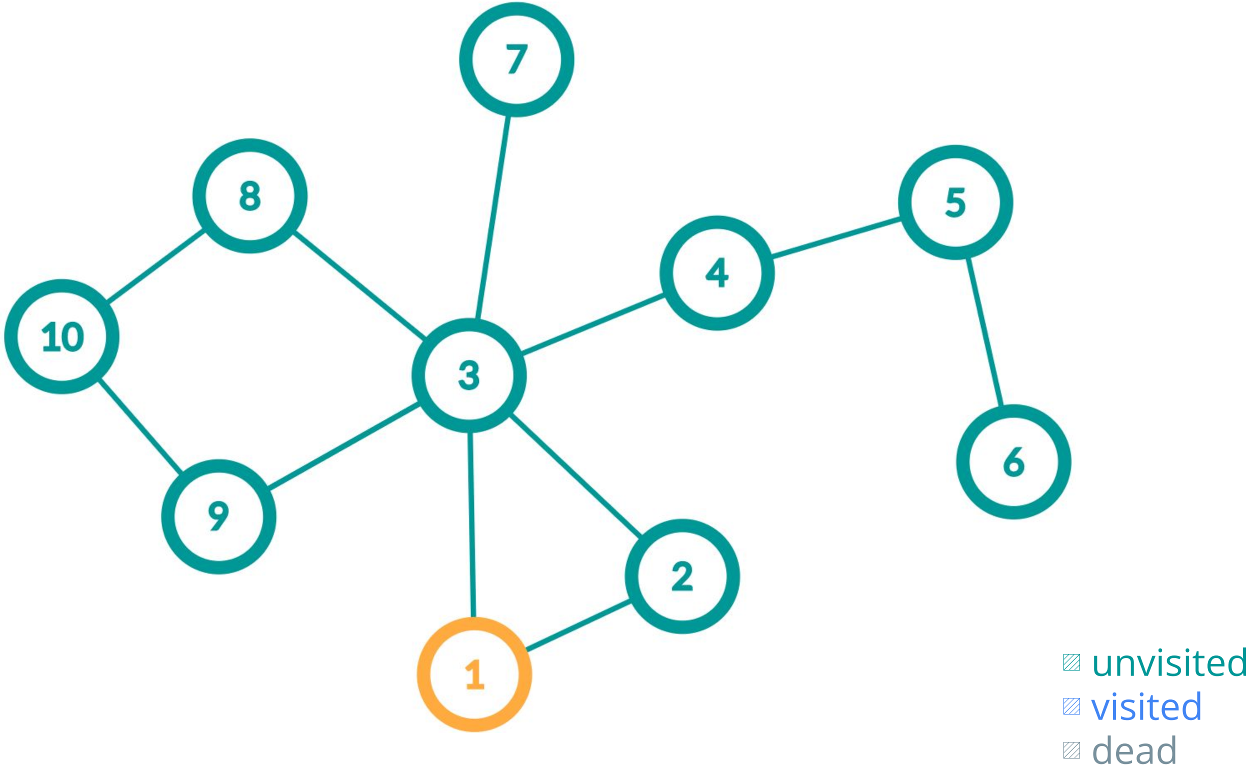
… until no new vertices are visited

| Queue |
| --- |
| |
| |
| |
| |
| |
| |
| |
| |

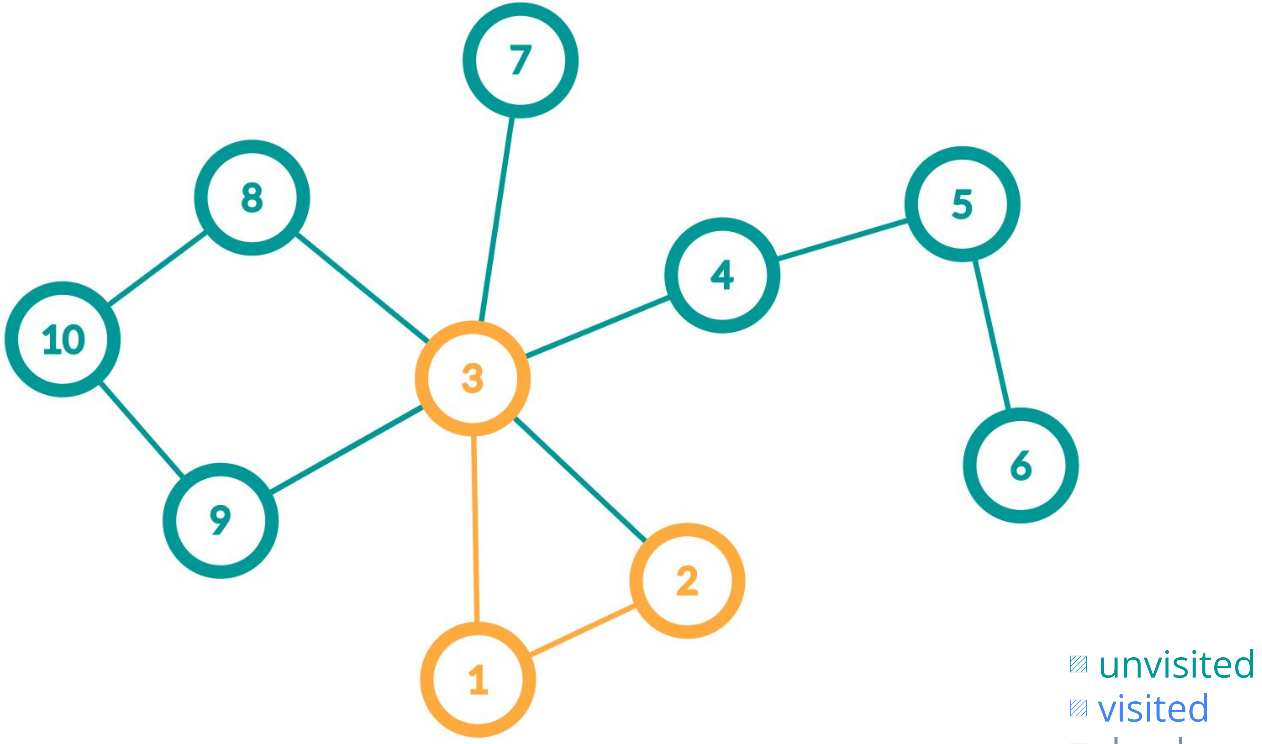



unvisited
visited
dead

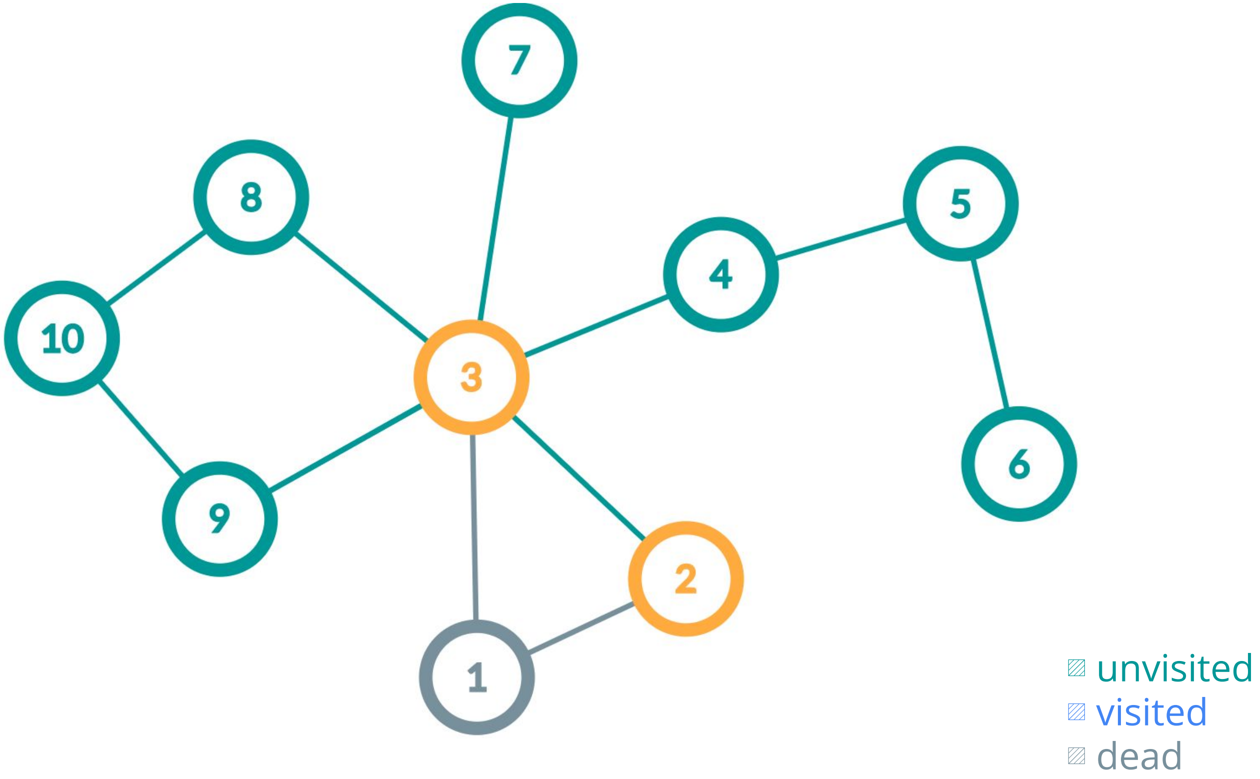
| Queue |
| --- |
| 1 |
| |
| |
| |
| |
| |
| |
| |



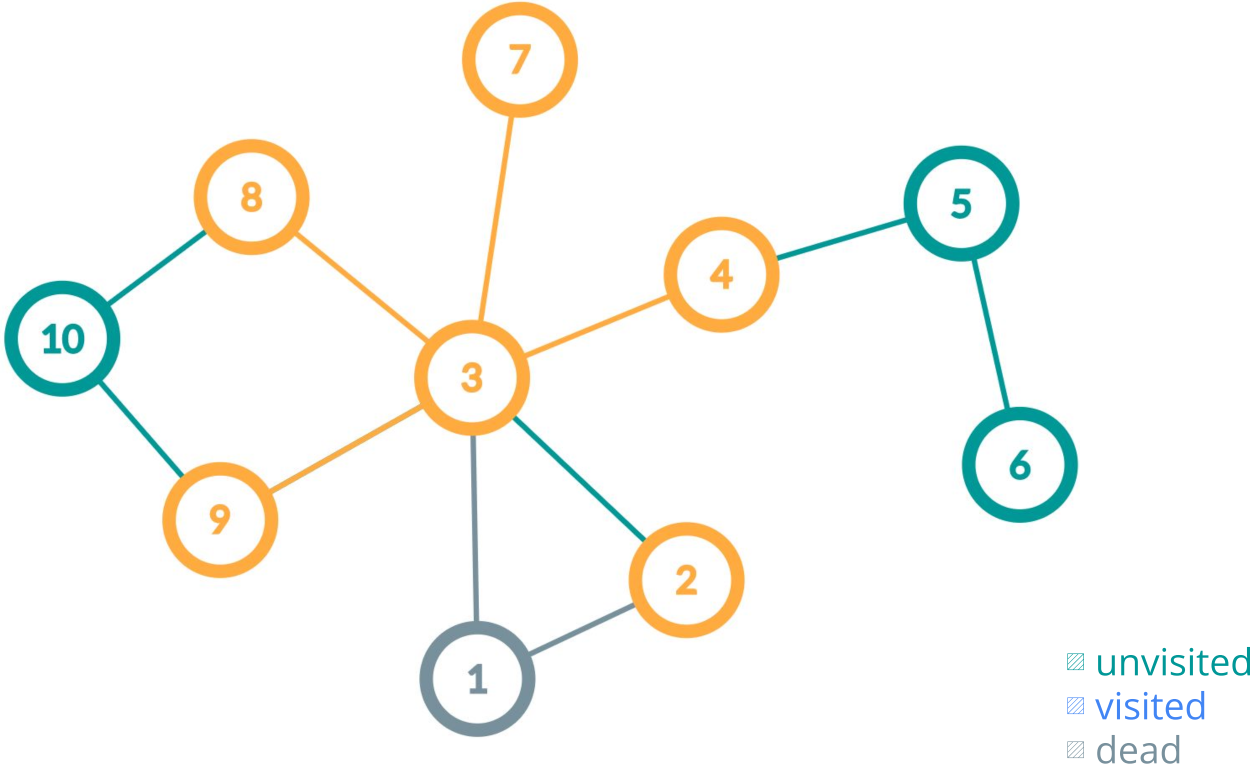
| Queue |
| --- |
| 1 |
| 2 |
| 3 |
| |
| |
| |
| |
| |



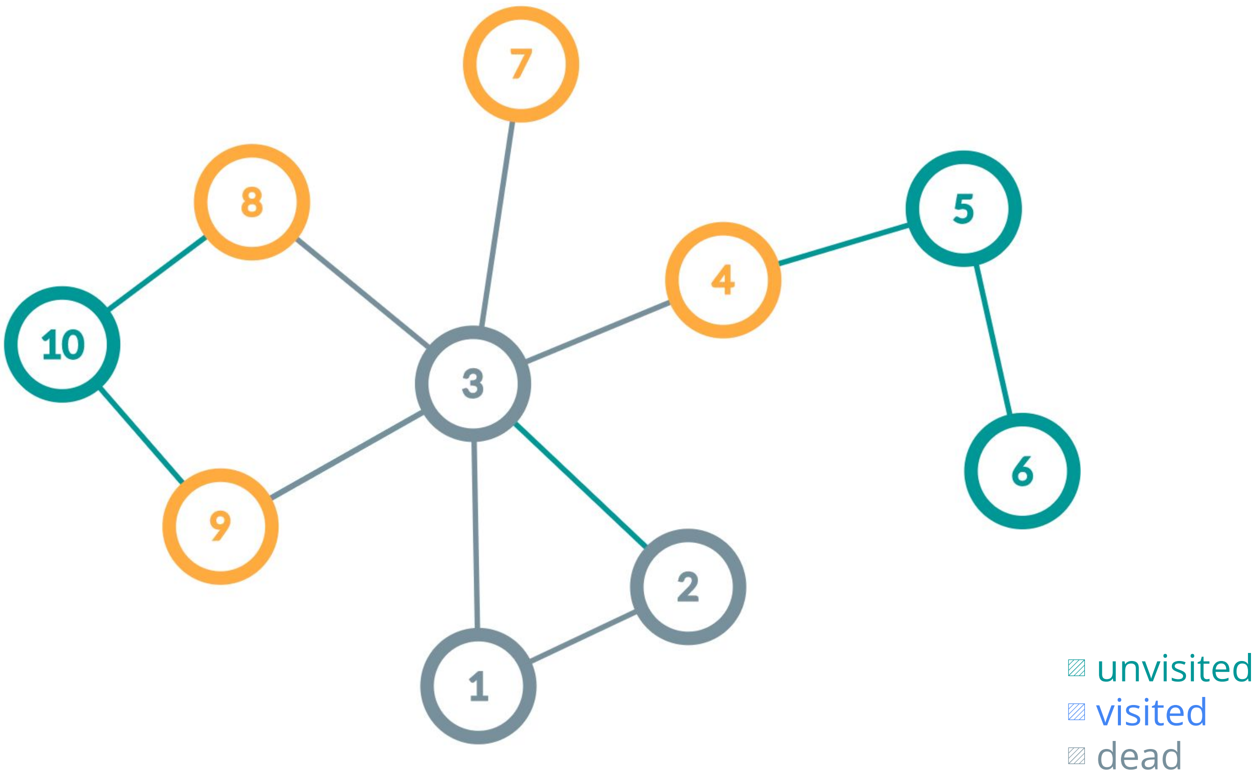
| Queue |
| --- |
| 2 |
| 3 |
| |
| |
| |
| |
| |
| |



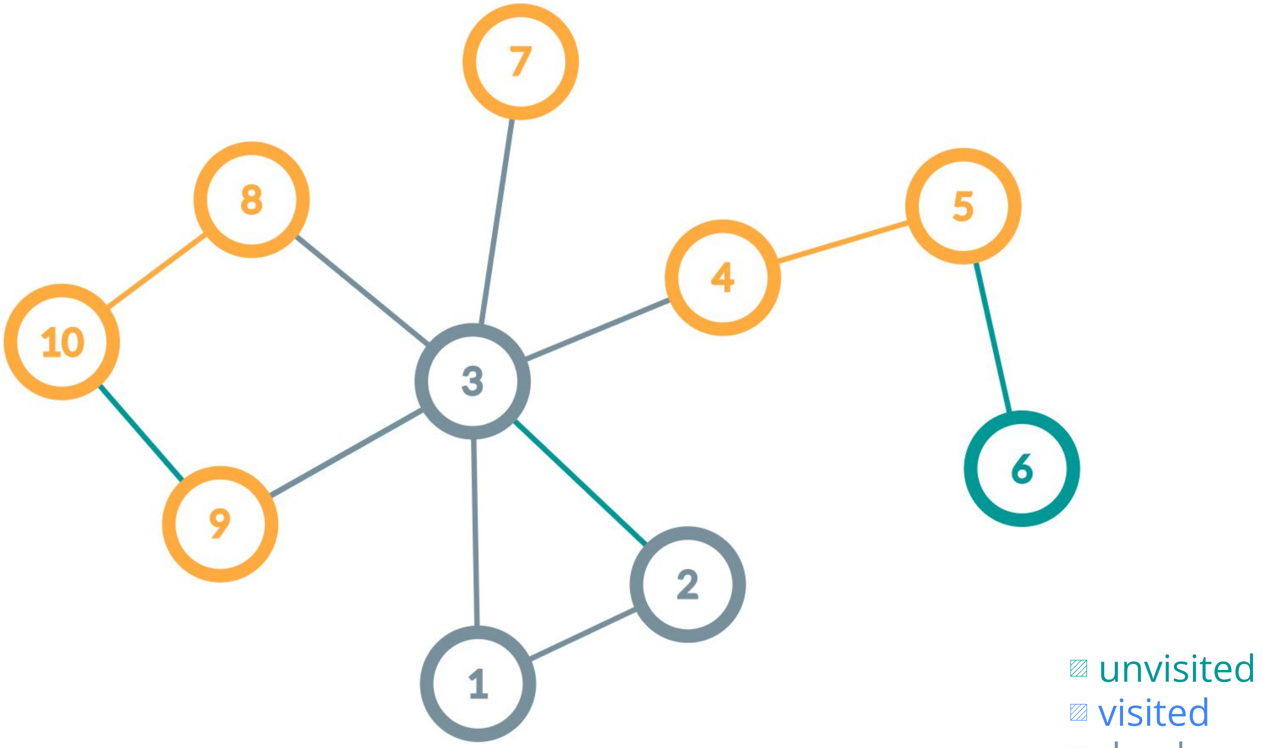
| Queue |
|---|
| 2 |
| 3 |
| 4 |
| 7 |
| 8 |
| 9 |
| |
| |



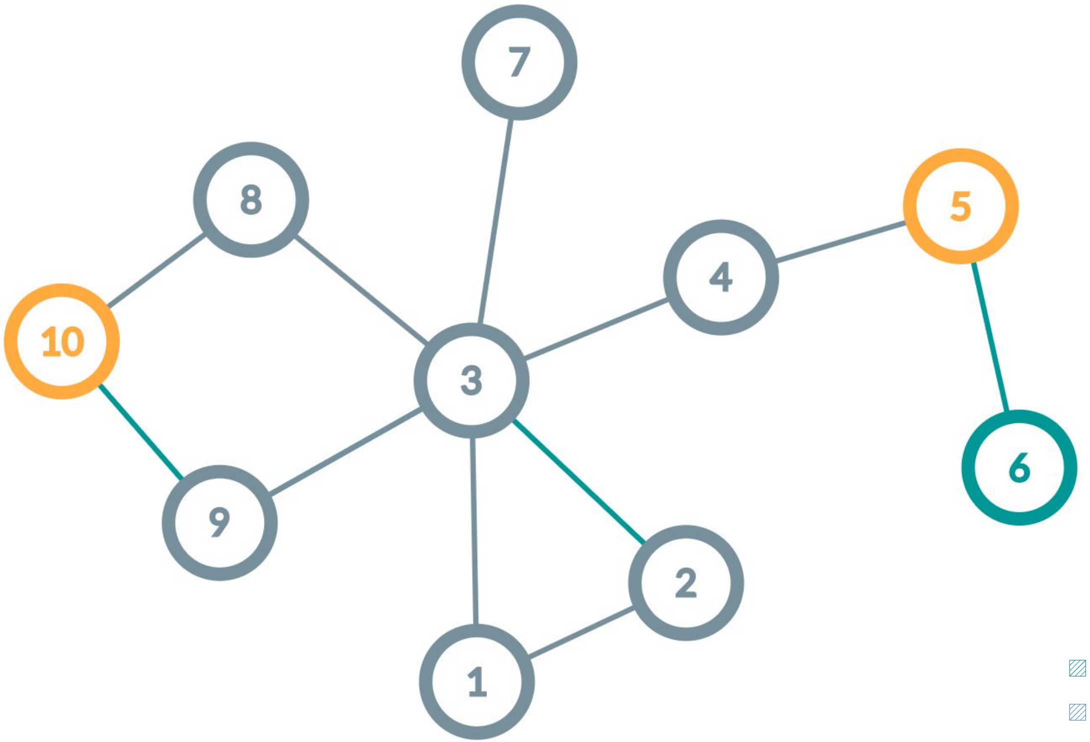
| Queue |
|---|
| 4 |
| 7 |
| 8 |
| 9 |
| |
| |
| |
| |



| Queue |
| --- |
| 4 |
| 7 |
| 8 |
| 9 |
| 5 |
| 10 |
| |
| |

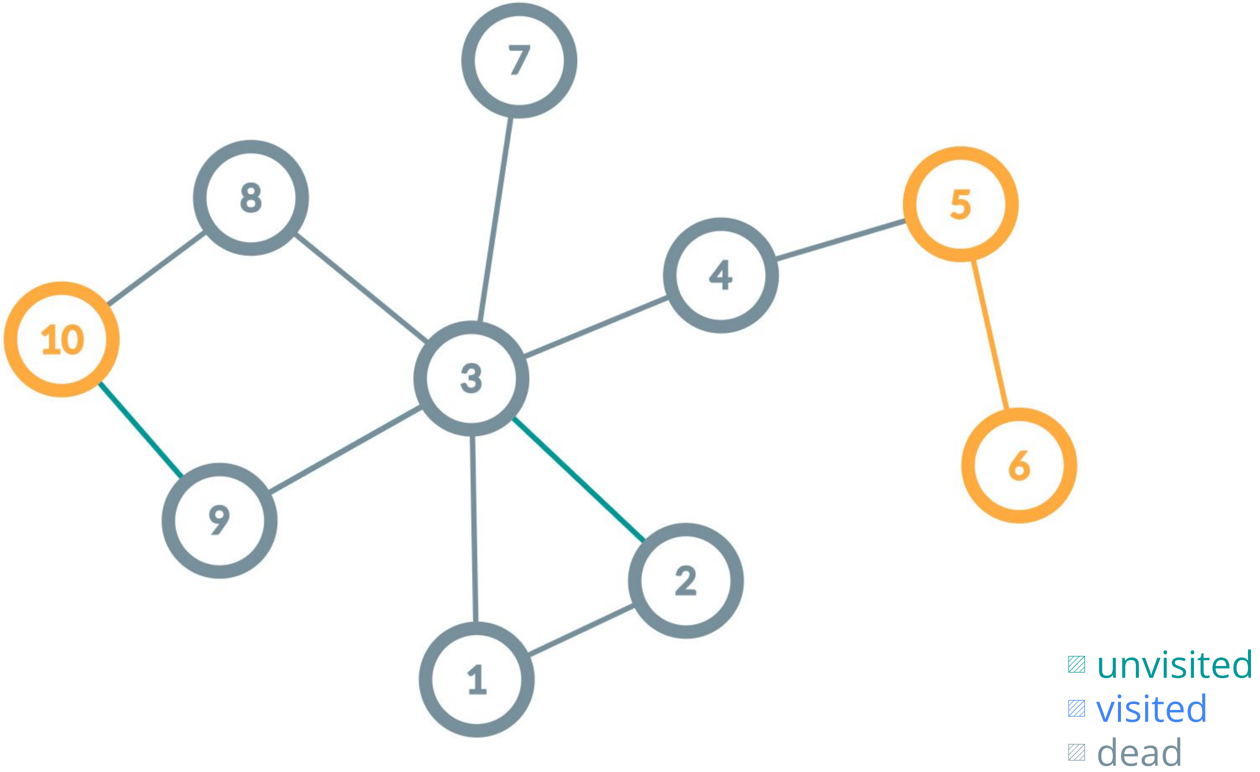


| Queue |
| --- |
| 5 |
| 10 |
| |
| |
| |
| |
| |
| |

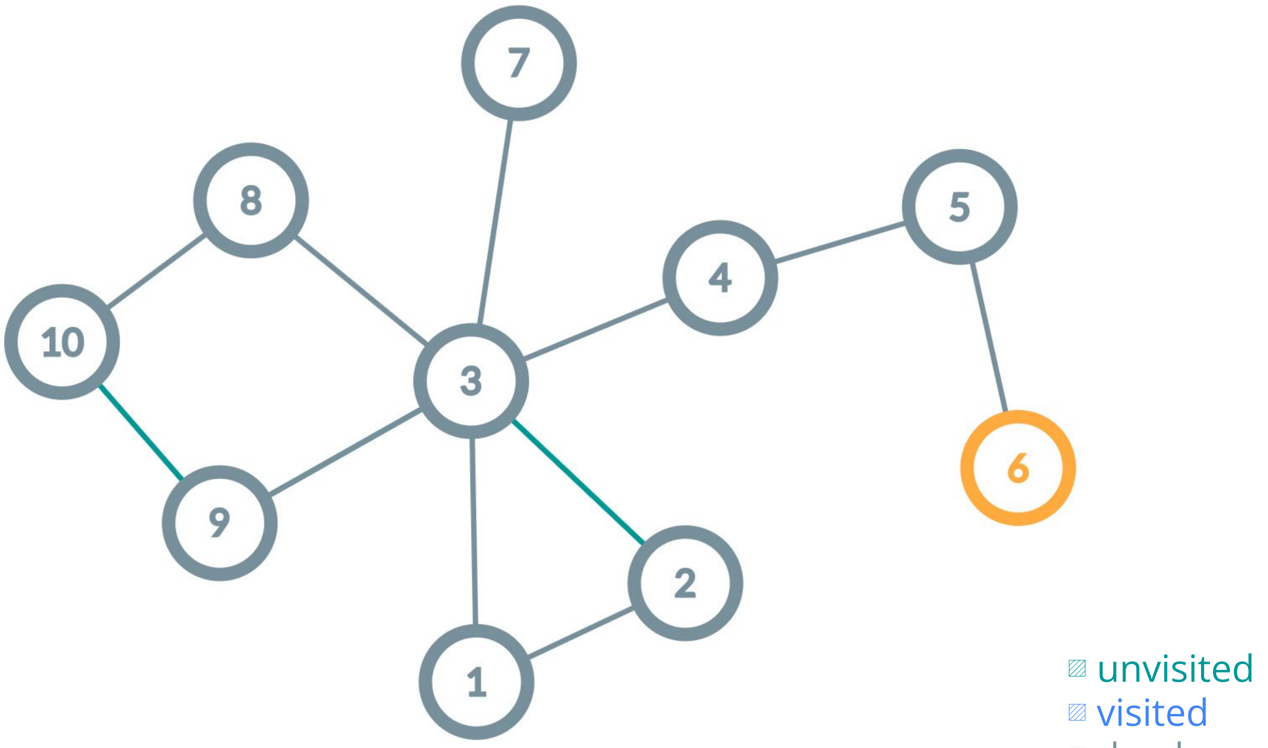



unvisited
visited
dead

| Queue |
|---|
| 5 |
| 10 |
| 6 |
| |
| |
| |
| |
| |

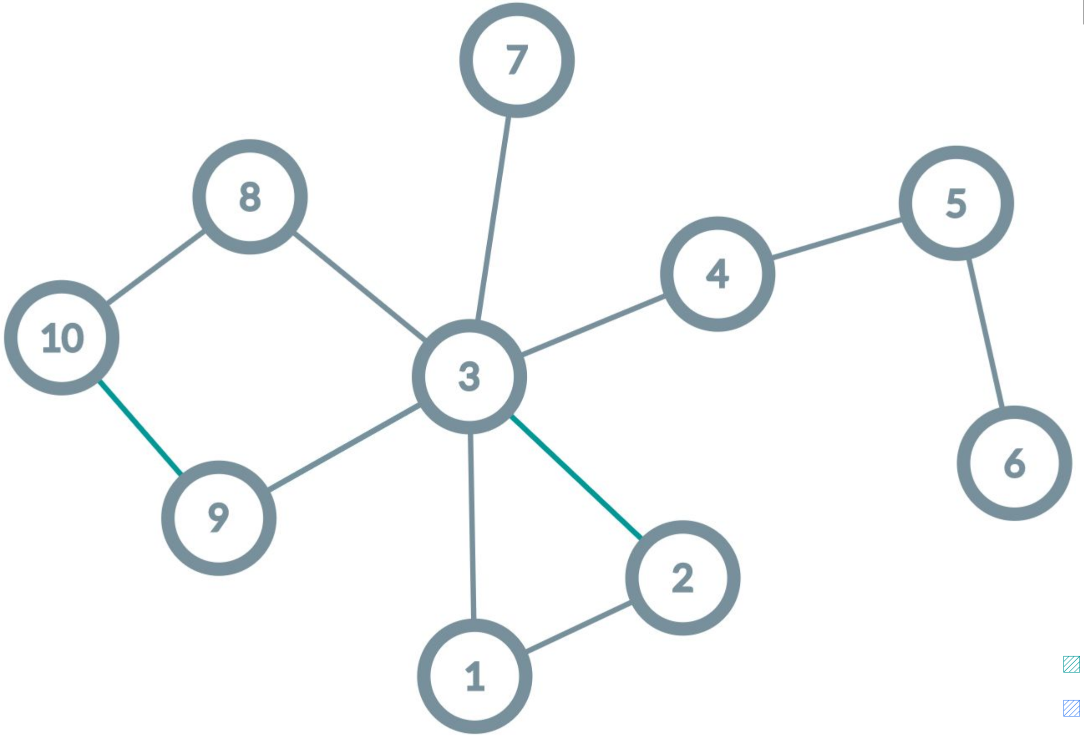


| Queue |
| --- |
| 6 |
| |
| |
| |
| |
| |
| |
| |



| Queue |
| --- |
| |
| |
| |
| |
| |
| |
| |
| |

DONE!!




unvisited
visited
dead

# Breadth-First Search

```
Push source vertex (denote as u) into queue Q
Mark u as visited
While (Q is not empty)
    Pop the front element (x) in Q
    Push all the unvisited vertices that are neighbors of x into Q
    Mark them as visited
```

# BFS - Water Jug Problem

There are 2 water jugs with capacities N and M litres respectively.

Initially, both of them are empty.

You can perform the following operations for infinitely many times (one operation a time)

1. Empty a jug
2. Fully fill a jug
3. Pour water from one jug to another until either one jug is empty / full

How to get a specific volume K in one of the jugs?

# BFS - Water Jug Problem

N = 3, M = 4, K = 2

States:

| | | | | |
|---|---|---|---|---|
| (0, 0) | (0, 1) | (0, 2) | (0, 3) | (0, 4) |
| (1, 0) | (1, 1) | (1, 2) | (1, 3) | (1, 4) |
| (2, 0) | (2, 1) | (2, 2) | (2, 3) | (2, 4) |
| (3, 0) | (3, 1) | (3, 2) | (3, 3) | (3, 4) |

initial state
target states

Run BFS from (0,0)

# BFS - Water Jug Problem

**Transitions from (x, y)**

1. Empty a jug (x, 0) and (0, y)
2. Fully fill a jug (x,M) and (N, y)
3. Pour water from the first jug to the second (x + y - M, M) or (0, x+ y)
4. Pour water from the second jug to the first (N, x+ y - N) or (x + y, 0)

# BFS - Maze Problem with Bomb

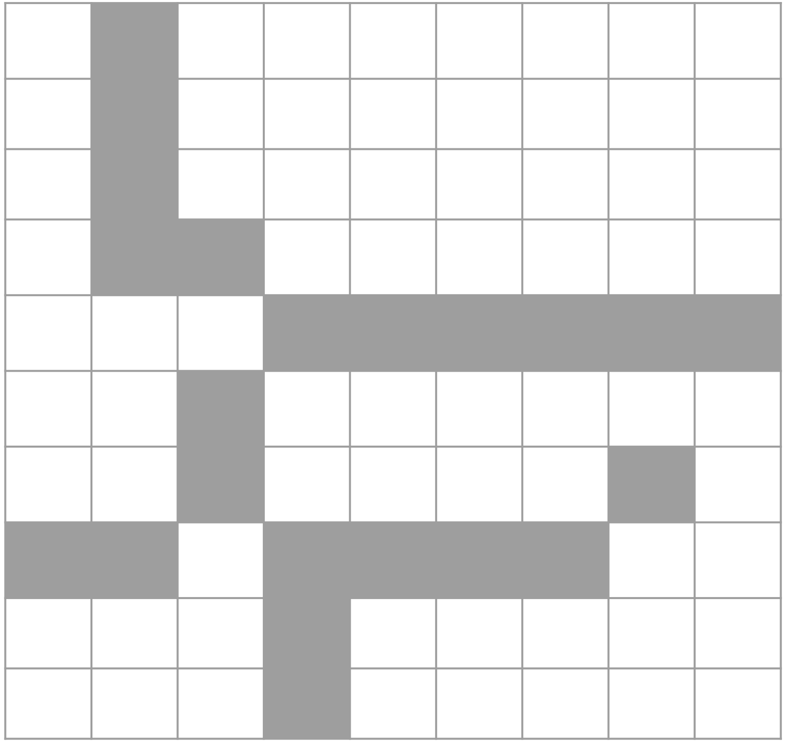
There is a N x M maze.

You can go U/ D/ L/ R one cell each move.

You have B bombs to destroy a wall of one cell

Find the **shortest path** from (1, 1) to (N, M)

blocked
visited



# BFS - Maze Problem with Bomb

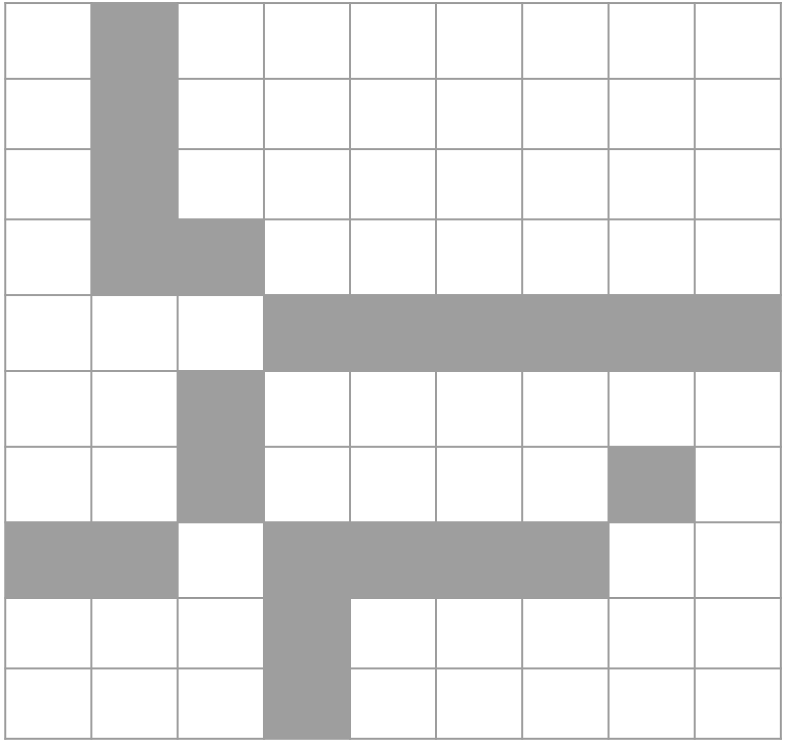
States: (x, y, b) [you are in cell (x, y) and you have b bombs]

Transitions

- (x, y + 1, b) if (x, y + 1) is not blocked
- (x, y + 1, b - 1) if (x, y + 1) is blocked and b is greater than 0

Same for other directions

Run BFS.



# Practice - CF1033A King Escape

https://codeforces.com/problemset/problem/1033/A

Alice and Bob are playing chess on a chessboard with dimensions NxN.

Alice has a queen at (ax, ay), Bob has a king at (bx, by)

Bob wins if he can move his king from (bx, by) to (cx, cy) without getting in check.

A king is in check if it is on the same row/column/diagonal as the enemy queen. “obstacle”

Remember that a king can move to any of the 8 adjacent squares. rule to form edge

Find whether Bob can win or not.

**Task: solve it with both DFS and BFS**

*(ignore the analytic O(1) solution for now)*

# Practice - CF1033A King Escape

Modify it as a graph problem:

- an edge exists between nodes sharing a corner/edge
- “obstacle” are grids on same row/column/diagonal line with (ax, ay)

1. start at (bx, by)
2. search all possible grids reachable
3. stop when reaching (cx, cy)

# Practice - CF1033A King Escape - Tricks

**Visit the surrounding grids efficiently** *(without hard-coding 8 directions)*

- observe that the dx, dy combinations are:

  {-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}

  → a combination of dx = {-1, 0, 1} and dy = {-1, 0, 1} *(except {0, 0})*

```
for dx = -1 to 1
for dy = -1 to 1
    if valid(cur_x + dx, cur_y + dy)
        visit(cur_x + dx, cur_y + dy)
```

# Practice - CF1033A King Escape - Tricks

## Check for diagonals

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | (1, 2) | | | | |
| 2 | | | (2, 3) | | | |
| 3 | (3, 1) | | | (3, 4) | | |
| 4 | | (4, 2) | | | (4, 5) | |
| 5 | | | (5, 3) | | | (5, 6) |
| 6 | | | | (6, 4) | | |

observe that all green grids are (1+k, 2+k)

→ generalize to (x0+k, y0+k),

*(x0, y0) = top left of each diagonal*

eliminate k:

- (x0 + k) - (y0 + k) = x0 - y0

all grids (x, y) in same top-left to bottom-right diagonal share the same (x - y)

# Practice - CF1033A King Escape - Tricks

## Check for diagonals

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | | (1, 3) | | | |
| 2 | | (2, 2) | | | | |
| 3 | (3, 1) | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |

observe that all green grids are (3 **-** k, 1 **+** k)

→ generalize to (x0 **-** k, y0 **+** k),

*(x0, y0) = bottom left of each diagonal*

eliminate k:

- (x0 - k) + (y0 + k) = x0 - y0

all grids (x, y) in same top-left to bottom-right diagonal share the same (x - y)

# Multi-source BFS - Flood Fill

Given N sources, find the distance between each cell from the nearest source.

Method 1: BFS from one source at a time

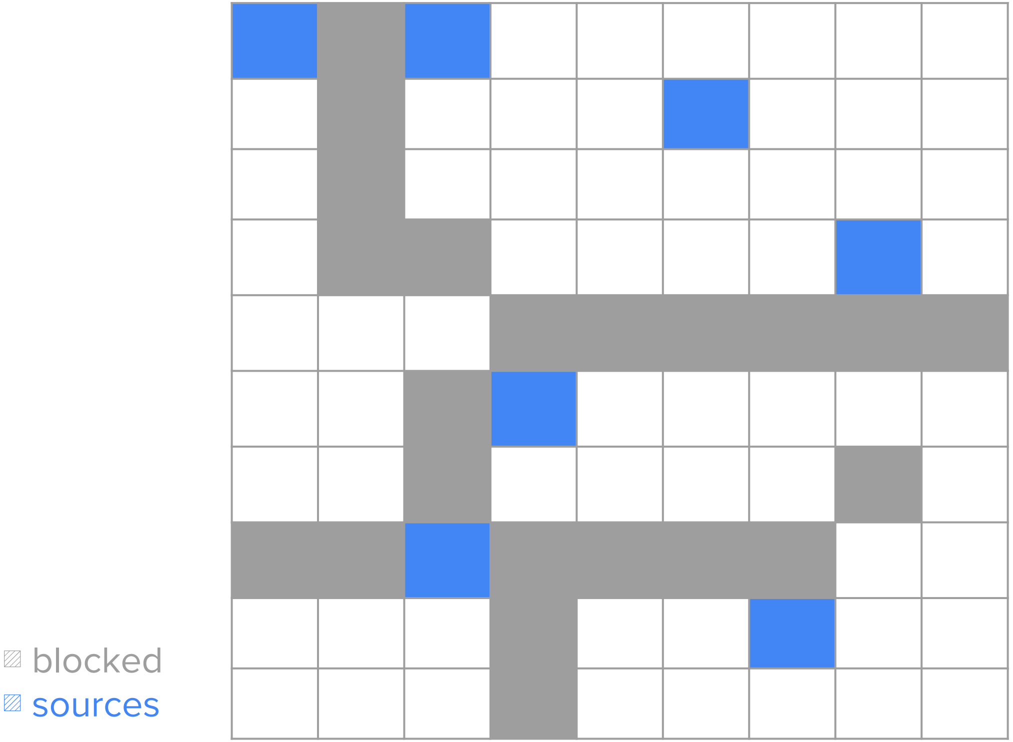
Time complexity: $O(N * (V + E))$

Method 2: BFS from all sources (perform BFS once)

Time complexity: $O(V + E)$

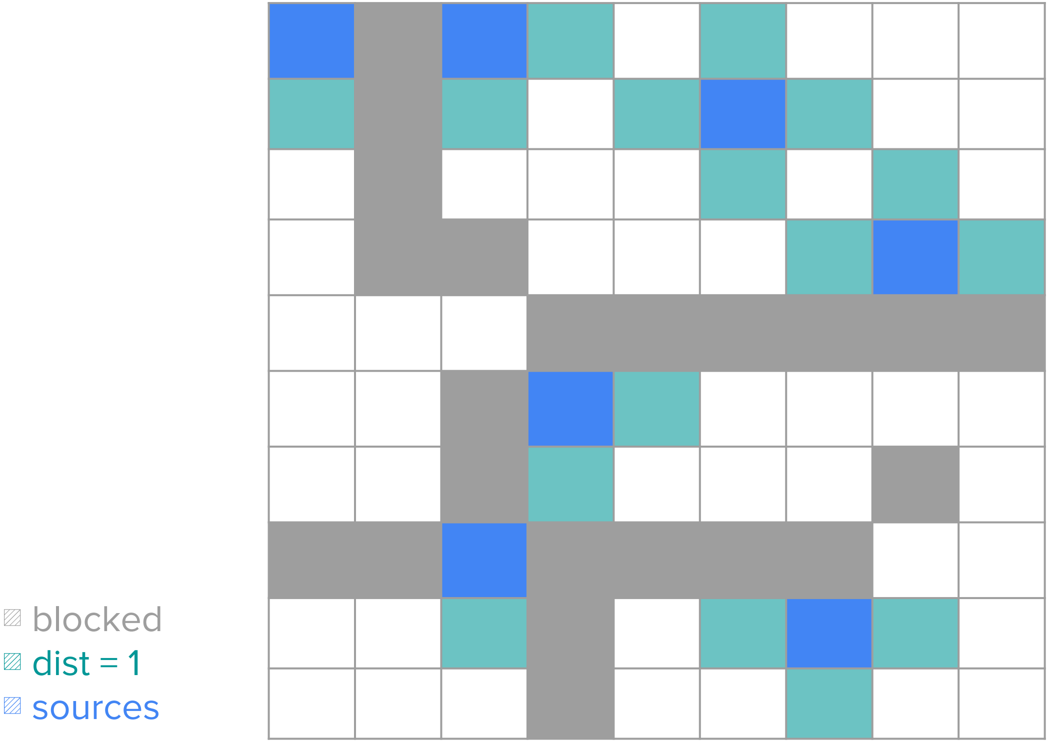
## Multi-source BFS

Multi-source BFS on flood fill…



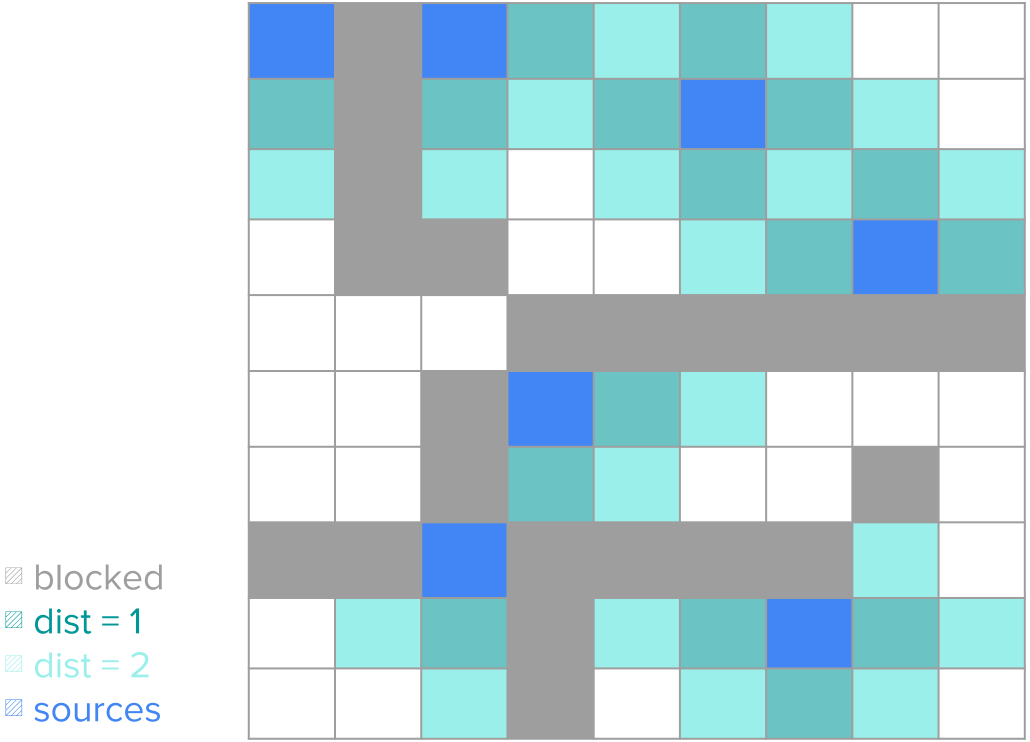
# Multi-source BFS

Multi-source BFS on flood fill...



# Multi-source BFS

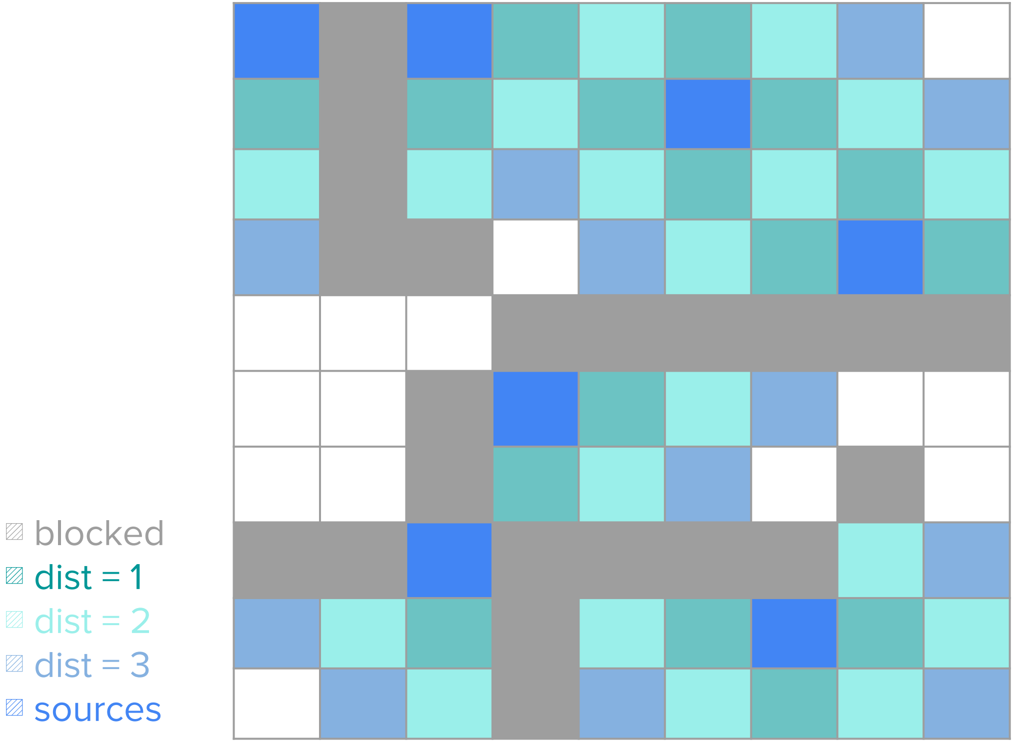
Multi-source BFS on flood fill...



# Multi-source BFS

Multi-source BFS on flood fill…

And so on…



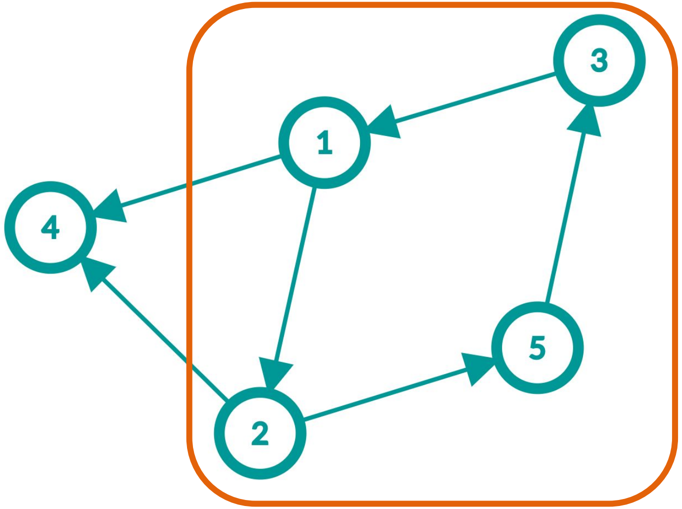
# Bidirectional search

To speed up finding unweighted shortest path from u to v
we can perform BFS with both u and v as sources
Stop when two paths meet in the middle

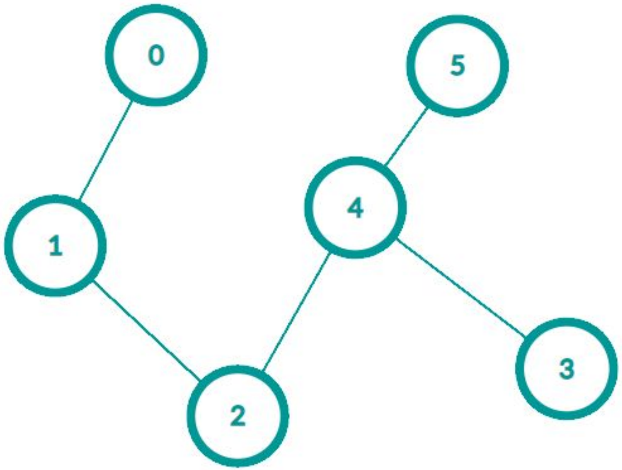
# Special Graphs

# Cycles

## Graph with cycles

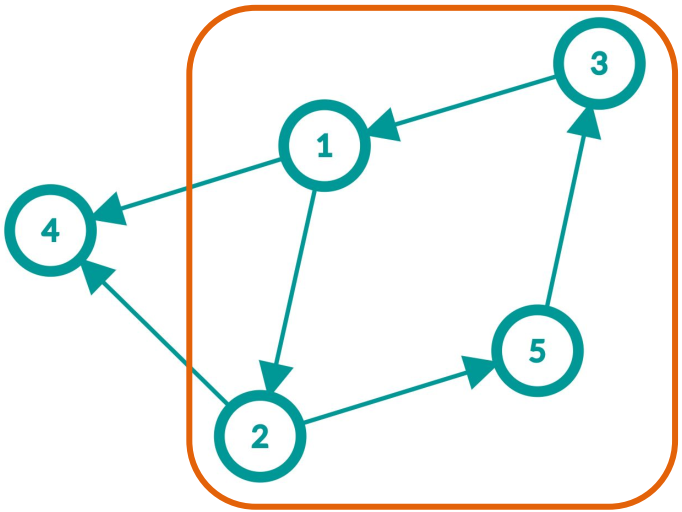



## Acyclic graph



# Cycles

Graph with cycles




**How to detect cycles in a graph?**

Recall searching, we avoid going to visited nodes again… why?

It indicates a cycle!

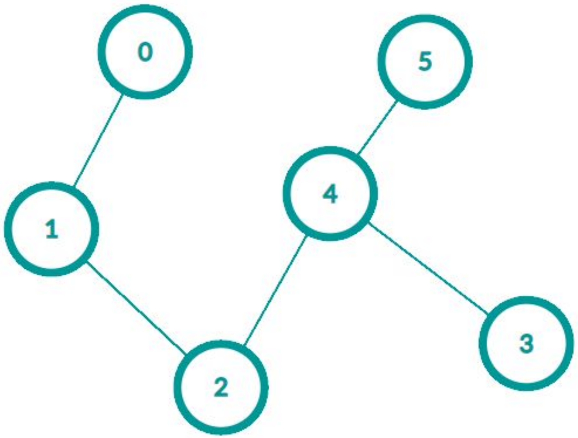
When we visit a node the second time, we know it is in a cycle

# Trees

- A connected graph with |V|-1 edges
- A connected graph without cycles
- A graph with exactly one path between every pair of vertices

You will learn more about trees in Graph(II) and (III)

- Minimum spanning tree
- Lowest common ancestor
- Tree traversal
- ...



# Chains

A tree with all vertices having 2 neighbors, except the two ends having only 1

Properties:

- Acyclic
- No branches

* Traverse from the ends


usually appear in earlier subtasks

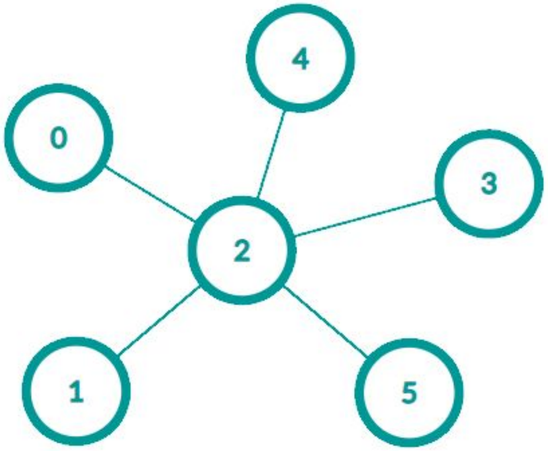
# Stars

A tree with one internal node and remaining nodes as leaves

Properties:

- Acyclic
- Max distance between 2 nodes = 2

* Special handling on the internal node



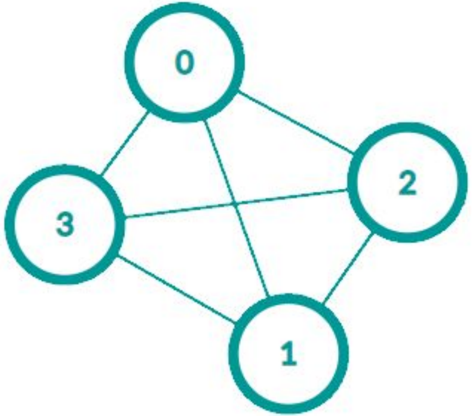

usually appear in earlier subtasks

# Complete Graph

Every pair of distinct vertices is connected by a unique edge

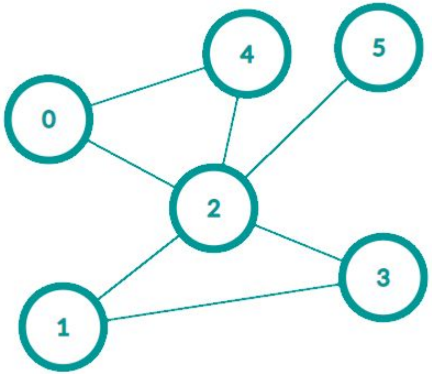
Properties:

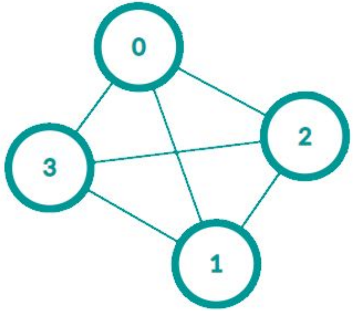
- `|E| = |V| * (|V| - 1) / 2`



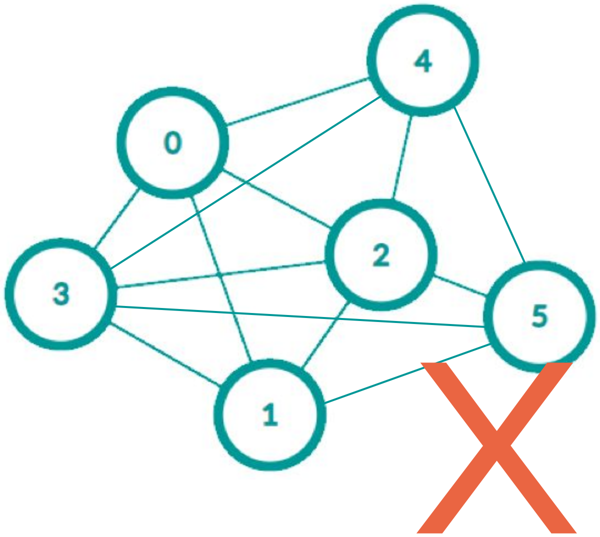
# Planar Graph

Vertices and edges in such graph can be drawn in a plane such that no two edges intersect




(can move node 3 inside 0,1,2)



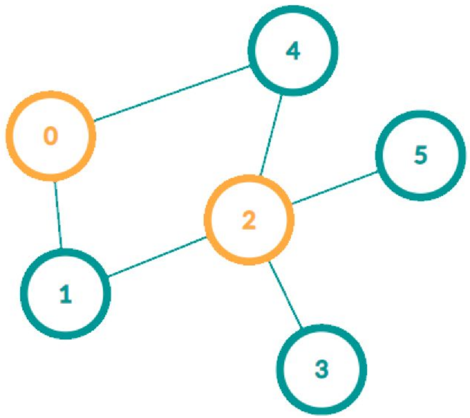
# Bipartite Graph

Vertices can be divided into two disjoint and independent sets U and V, such that every edge connects a vertex in U to a vertex in V.

You can assign one of two colours for each node such that all edges have different colour on two side

Properties:

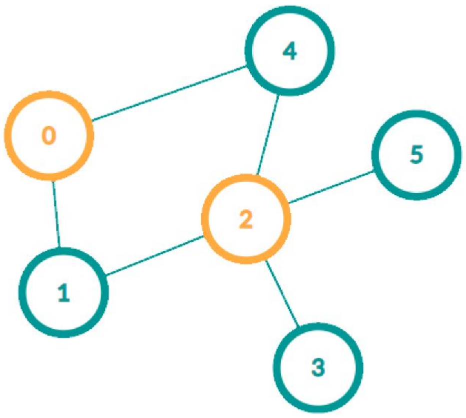
- Does not contain odd cycles



# Bipartite Graph

Algorithm to check if the graph is bipartite / color the graph into a bicolor graph

```
Perform DFS/BFS on any node
     Assign 0 and 1 alternately to every node according to the depth
     If a node is assigned 0, assign 1 to its neighbors, and vice versa
If any edge has same number on two sides, the graph is not bipartite
```



# Practice Problems

HKOJ 01035 Patrol Area

HKOJ 01067 Maze

HKOJ M1311 Dokodemo Door

HKOJ T022 Bomber Man

HKOJ M0911 Theseus and the Minotaur

HKOJ 30422 Knights in FEN

CF 1033A King Escape

CF 893C Rumor

<dfs or equivalent / graph tag in codeforces>

演算法筆記

Graph Theory by Teb’s Lab