Tutorial 08 — Graph DS & Traversal

CS2040C Semester 2 2018/2019

By Jin Zhe, adapted from slides by Ranald, AY1819 S2 Tutor

Graph Representation

Adjacency Matrix
Adjacency List
Edge List

Graph Representation

Edge List

A list of edges in the entire graph.

Adjacency Matrix

2D Array where `adj_mat[x][y]` stores information about edge $u \rightarrow v$ (or information that it does not exist).

Adjacency List

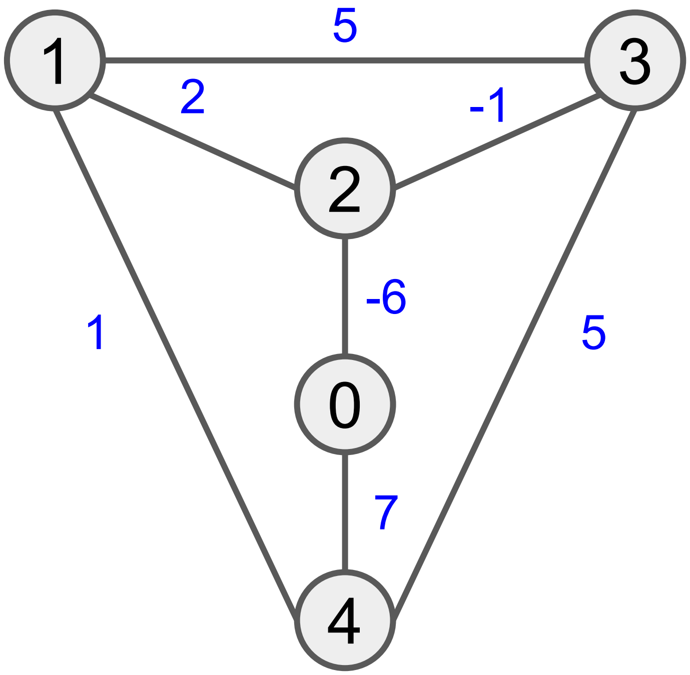
For each vertex, keep a list of vertices it has outgoing edges to

Point to consider

For each of the graph representations, think of the following questions for a graph with V vertices and E edges:

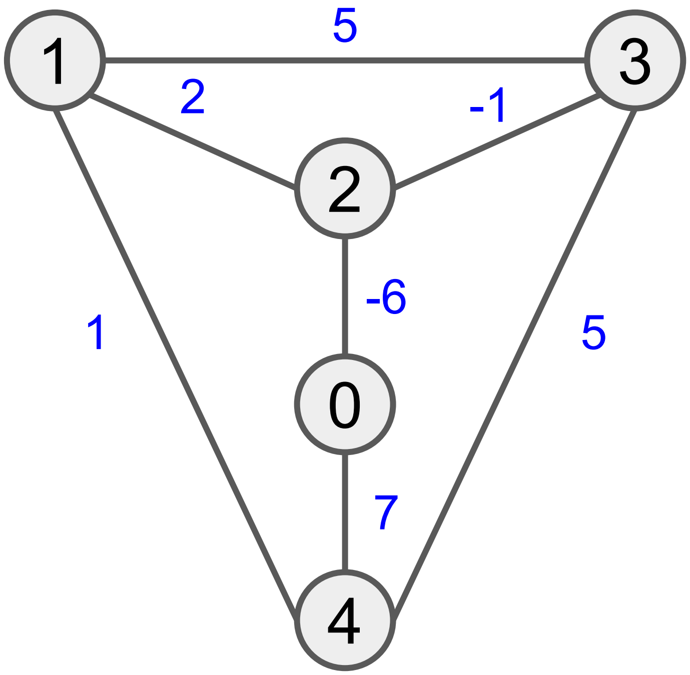
- What's the space complexity?
- What's the time complexity?
  - To very if a edge exists between u and v
  - To retrieve a list of all the neighbours of a vertex u

Graph Representation—Edge List



(Vertex **u**, Vertex **v**, Edge weight **w**)

Graph Representation—Edge List



(Vertex **u**, Vertex **v**, Edge weight **w**)
`(0, 2,-6)`
`(0, 4, 7)`
`(1, 2, 2)`
`(1, 3, 5)`
`(1, 4, 1)`
`(2, 3,-1)`
`(3, 4, 5)`

Graph Representation—Edge List

- Space complexity: ___
- Time complexity
  - Verify (u, v) is an edge: ___
  - Get neighbours of u: ___

(Vertex **u**, Vertex **v**, Edge weight **w**)
`(0, 2,-6)`
`(0, 4, 7)`
`(1, 2, 2)`
`(1, 3, 5)`
`(1, 4, 1)`
`(2, 3,-1)`
`(3, 4, 5)`

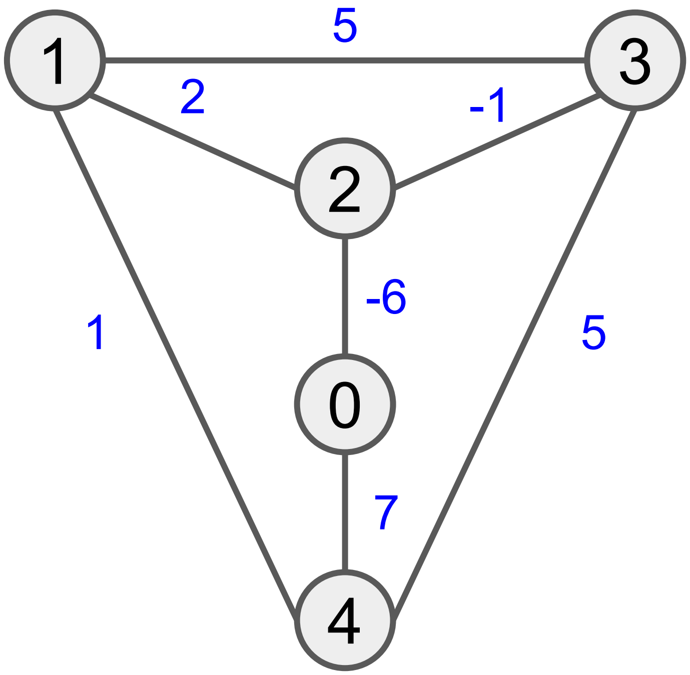
Graph Representation — Edge List

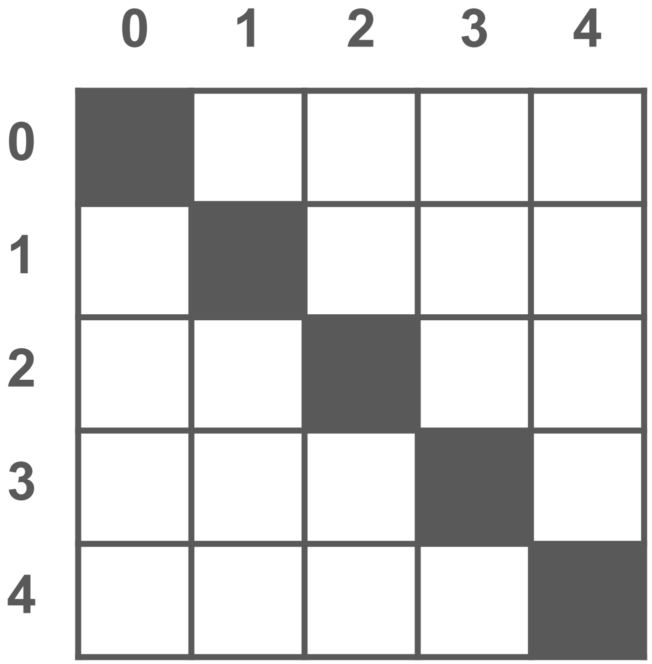
- Space complexity: $O(E)$
- Time complexity
  - Verify (u, v) is an edge: $O(E)$
  - Get neighbours of u: $O(E)$

(Vertex **u**, Vertex **v**, Edge weight **w**)
`(0, 2,-6)`
`(0, 4, 7)`
`(1, 2, 2)`
`(1, 3, 5)`
`(1, 4, 1)`
`(2, 3,-1)`
`(3, 4, 5)`

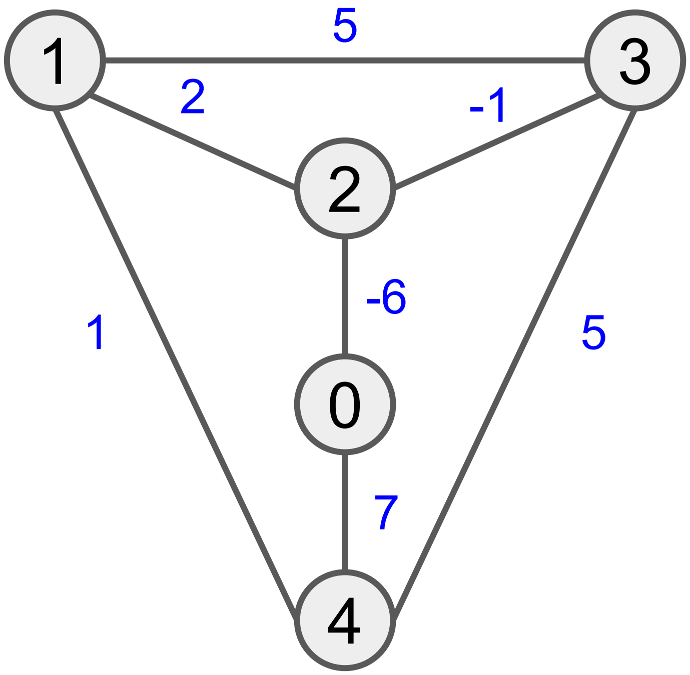
E

Graph Representation — Adjacency Matrix



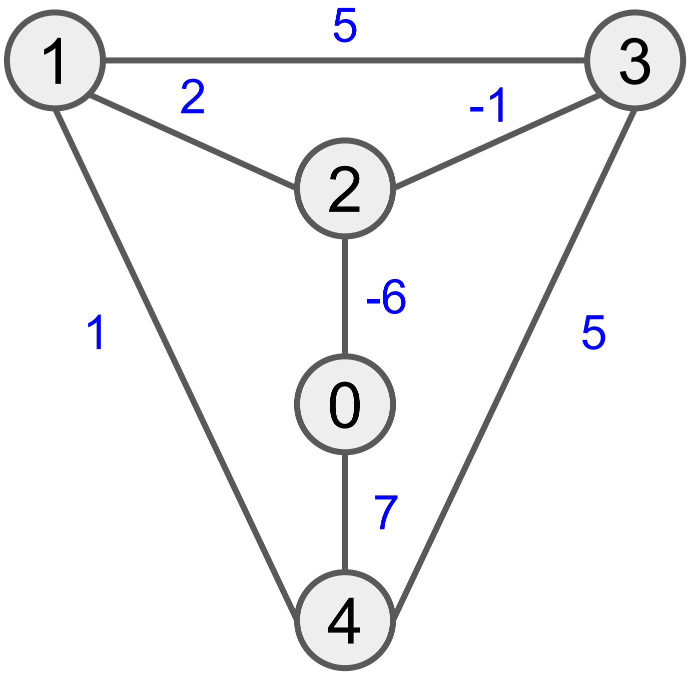


Graph Representation－Adjacency Matrix



	0	1	2	3	4
0			-6		7
1			2	5	1
2	-6	2		-1	
3		5	-1		5
4	7	1		5	

Graph Representation—Adjacency Matrix



	0	1	2	3	4
0			-6		7
1			2	5	1
2	-6	2		-1	
3		5	-1		5
4	7	1		5	

Symmetry along diagonal
for bidirectional graphs!

Graph Representation — Adjacency Matrix

- Space complexity: ___
- Time complexity
  - Verify (u, v) is an edge: ___
  - Get neighbours of u: ___

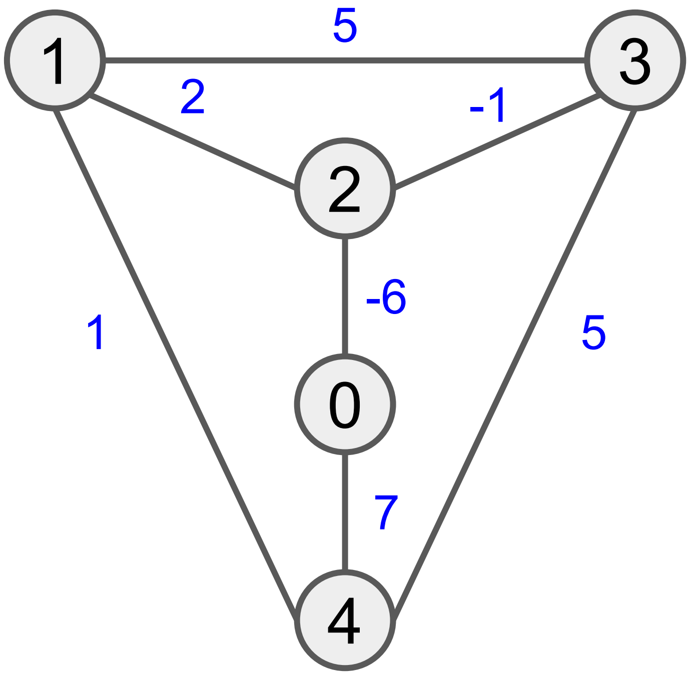
	0	1	2	3	4
0			-6		7
1			2	5	1
2	-6	2		-1	
3		5	-1		5
4	7	1		5	

Graph Representation — Adjacency Matrix

- Space complexity: $O(V^2)$
- Time complexity
  - Verify (u, v) is an edge: $O(1)$
  - Get neighbours of u: $O(V)$

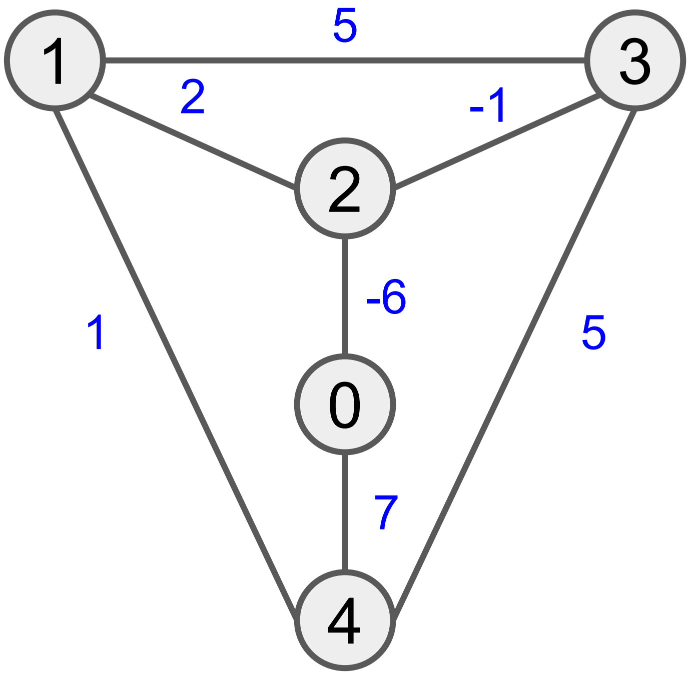
	0	1	2	3	4
0			-6		7
1			2	5	1
2	-6	2		-1	
3		5	-1		5
4	7	1		5	

Graph Representation—Adjacency List



Vertex **u**	List(vertex **v**, weight **w**)
0	
1	
2	
3	
4	

Graph Representation—Adjacency List



Vertex **u**	List(vertex **v**, weight **w**)
`0`	`(2, -6), (4, 7)`
`1`	`(2, 2), (3, 5), (4, 1)`
`2`	`(0, -6), (1, 2), (3, -1)`
`3`	`(1, 5), (2, -1), (4, 5)`
`4`	`(0, 7), (1, 1), (3, 5)`

Graph Representation — Adjacency List

- Space complexity: ___
- Time complexity
  - Verify (u, v) is an edge: ___
  - Get neighbours of u: ___

Vertex **u**	List(vertex **v**, weight **w**)
`0`	`(2, -6), (4, 7)`
`1`	`(2, 2), (3, 5), (4, 1)`
`2`	`(0, -6), (1, 2), (3, -1)`
`3`	`(1, 5), (2, -1), (4, 5)`
`4`	`(0, 7), (1, 1), (3, 5)`

Graph Representation — Adjacency List

- Space complexity: $O(V+E)$
- Time complexity
  - Verify (u, v) is an edge: $O(V)$
  - Get neighbours of u: $O(1)$

Vertex **u**	List(vertex **v**, weight **w**)
`0`	`(2, -6), (4, 7)`
`1`	`(2, 2), (3, 5), (4, 1)`
`2`	`(0, -6), (1, 2), (3, -1)`
`3`	`(1, 5), (2, -1), (4, 5)`
`4`	`(0, 7), (1, 1), (3, 5)`

V

E if directed
$2E$ if undirected

Graph Representation—Parent Array

Representing a tree

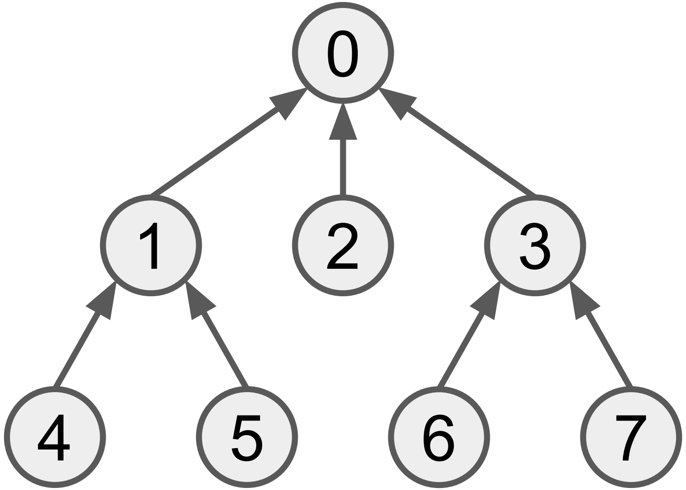
What is the most concise way to capture all the information regarding a tree? :O

Realize that a tree entails hierarchy!

How many parent does each vertex have?

What if all edges are from child → parent?

Graph Representation — Parent Array



Vertex	Parent
0	
1	0
2	0
3	0
4	1
5	1
6	3
7	3

DAG

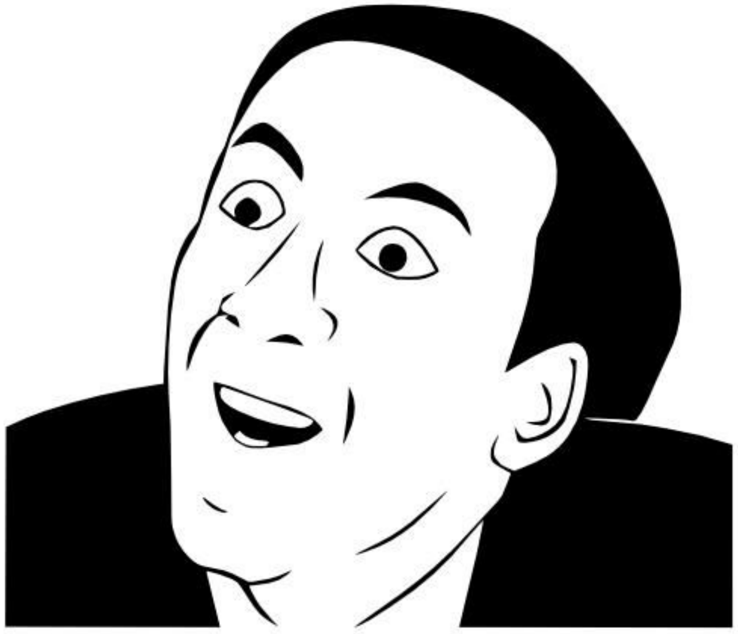
Directed **A**cyclic **G**raph

Directed Acyclic Graph (DAG)

Definition

- Directed
- Acyclic
- Graph

Yup that's it!



Directed Acyclic Graph (DAG)

Definition

- Directed: Edges are **not** bidirectional
- Acyclic: No cycles and no self-loops
- Graph

Directed Acyclic Graph (DAG)

Properties

After traversing an edge from vertex $u \rightarrow v$,

You can *never* reach vertex u again through any series of directed edges.

Can you prove it? [By contradiction]

Question 2

Problem statement

Draw a DAG with V vertices and $E = V(V-1)/2$ directed edges.

Observations

How to start?

Here’s an idea: Observe how the graph property when we change the number of vertices by 1.

So let’s consider the cases when V is $k-1$, k and $k+1$

Observations

$k-1$ vertices:	$(k-1)(k-2)/2$	edges
k vertices:	$(k)(k-1)/2$	edges
$k+1$ vertices:	$(k+1)(k)/2$	edges

Observations

When we increased V from $(k-1)$ to k:

- E increased by $(k-1)$

When we increased V from k to $(k+1)$:

- E increased by k

Observations

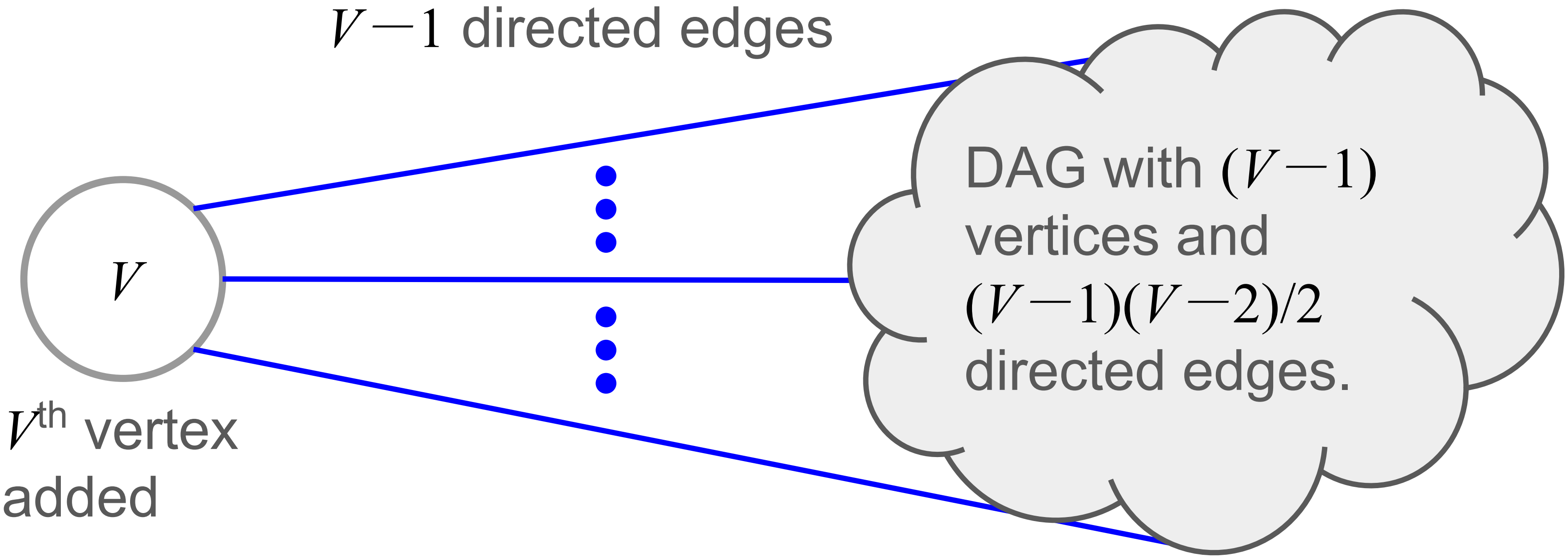
$V - 1$ vertices: $(V-1)(V-2)/2$ edges

V vertices: $(V)(V-1)/2$ edges

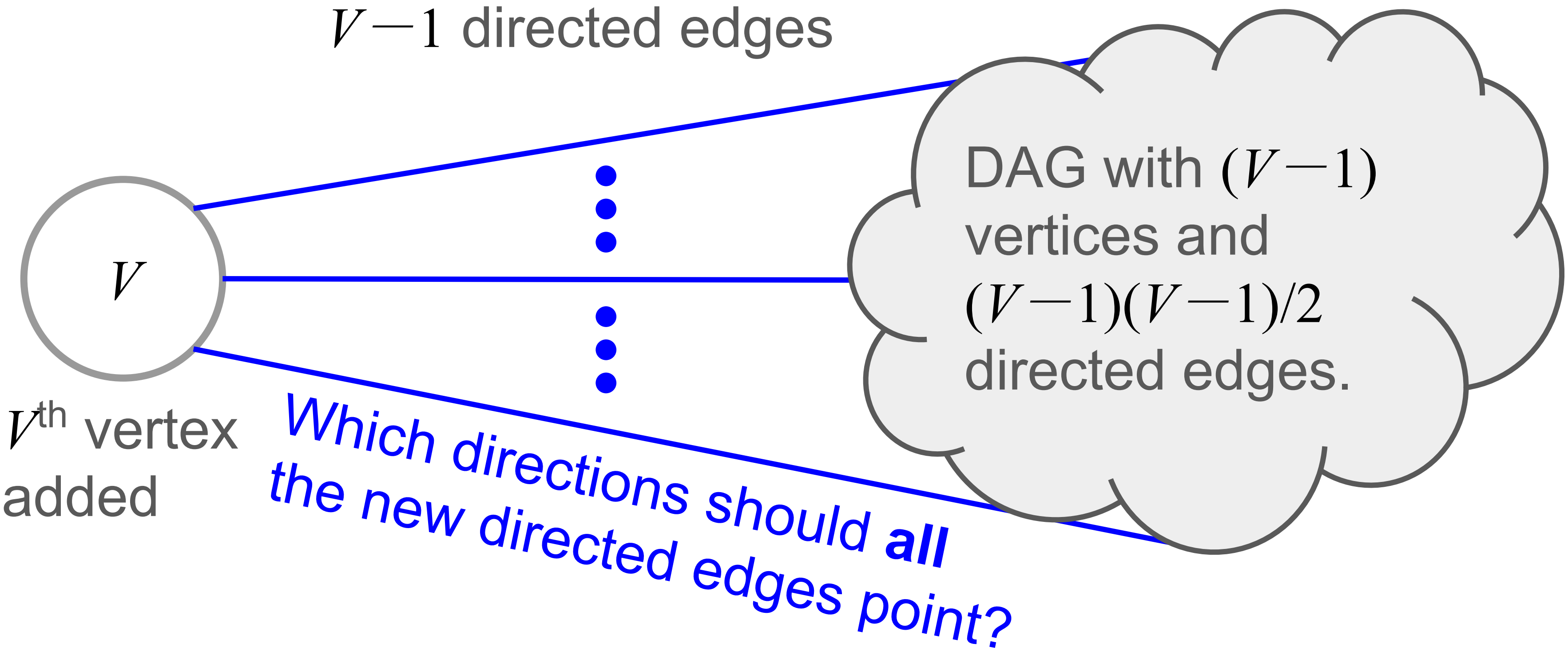
Assuming we can draw a graph with $V-1$ vertices, we can construct a solution by:

- Adding 1 vertex and $V-1$ directed edges to it
- While maintaining DAG property

Observations



Observations



Construction

Let's start by drawing a graph with only 1 vertex and 0 edges

Note that it is a DAG.

It has $(V)(V-1)/2 = 0$ directed edges.

Construction

We can now inductively construct a solution with the following algorithm:

We label the first vertex as 1.

For each vertex u from 2 to V,

- Draw a *directed* edge from vertex u to vertices $< u$

Test yourself!

On the edge

Can we have more than $V(V-1)/2$ directed edges for a DAG with V vertices?

Test yourself!

On the edge

Can we have more than $V(V-1)/2$ directed edges for a DAG with V vertices?

No!

With $V(V-1)/2$ directed edges, there is already exactly 1 edge between every pair of vertices. Adding any more will form a *bi-directional edge*.

Hold up a sec!

Did you realize that **if the edges were undirected**, you are essentially drawing a *complete graph*?

This should be obvious if you knew the *handshaking lemma* from CS1231! More on this in the next question.

Question 3

Problem statement

Prove that a **complete** ***simple graph*** of V vertices has

$$^{V}C_{2} = V(V-1)/2 \qquad \textbf{edges}$$

This is frequently used as the bound of a graph algorithm's time complexity on simple graphs!

Definitions

Simple graph

- Undirected
- Unweighted
- No self-loops, no multi-edges

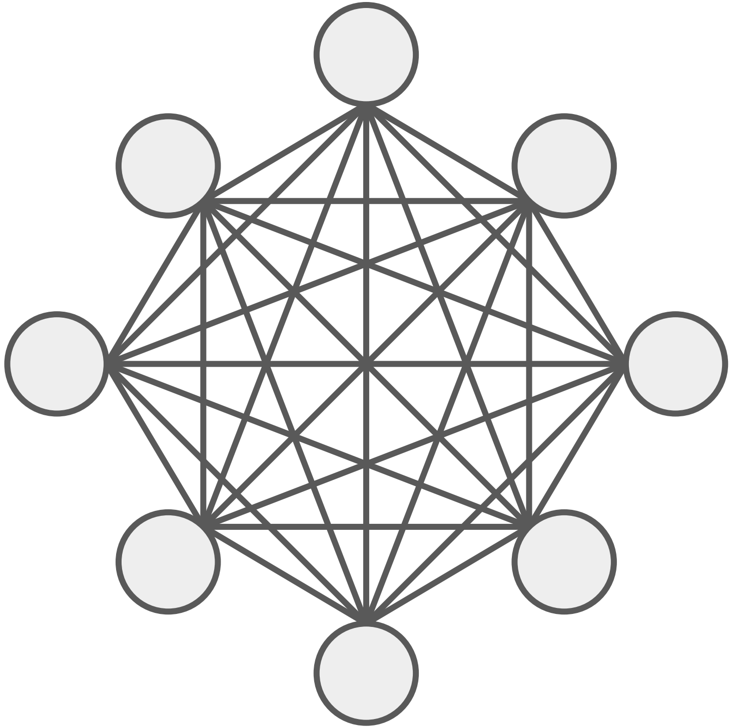
Complete graph

- There is an edge between every pair of distinct vertices. i.e. Not possible to add any more edges to graph

Question 3: Combinatorics solution

A complete graph essentially has every possible combination of edges between any 2 vertices.

Counting them is thus the same as the counting the total number of ways to choose 2 vertices out of V vertices. Therefore ${}^{V}C_{2}$



Complete undirected graph with 8 vertices

Question 3: Mathematical induction solution

We can prove by mathematical induction analogous to the construction approach outlined in the previous question

Question 3: Handshaking lemma solution

- We shall model vertices as people and edges as handshakes.
- Imagine a room with V people and every person shakes hands with everyone else
- The total number of **handshakes received** is $V(V-1)$
- However, whenever there is a handshake, two hands are being shaken!
- So to count the distinct pairs of hands that shook with each other (i.e. edges), we need to divide the total number of handshakes by 2
- Thus, we have $V(V-1)/2$

$Q.E.D$

Question 3: Direct counting method

Label the vertices from 1 to V.

Since its a complete graph, each vertex has edges to every other vertex.

We maintain a cumulative sum initially set to 0. This will be the number of edges at the end.

Question 3: Direct counting method

- First add $V - 1$ (number of V's edges) to sum
- Then remove vertex V **and all its $V - 1$ edges** to avoid double-counting later
- Remaining vertices are 1 to $V - 1$, with each having $V - 2$ edges.
- Repeat this process with vertex $V - 1$ and so on
- Eventually we will reach the last 2 vertices and removing one of them counts in the final edge

Question 3: Direct counting method

By repeatedly performing removal,

We get number of edges

$$\begin{aligned} \text{sum} &= (V-1)+(V-2)+\ldots+1 \\ &= (V-1)/2 \times ((V-1)+1) \qquad \text{from sum of AP} \\ &= V(V-1)/2 \\ &= {}^{V}C_{2} \end{aligned}$$

General Properties of Graphs

What is the **min/max** number of edges for a *connected, simple, undirected* graph of V vertices?

Min: ______________

Max: ______________

General Properties of Graphs

What is the **min/max** number of edges for a *connected, simple, undirected* graph of V vertices?

Min: $V-1$ Case: **tree**

Max: $V(V-1)/2$ or ${}^{V}C_{2}$ Case: **complete** graph

Recall flyingsafely?

General Properties of Graphs

What is the **max** number of edges for a **directed graph** of V vertices?

With no multi-edges, no self-loops. But cycles are permitted

I omit using the word *connected* as it is vague for *directed* graphs.

(Those interested, ask during consultations..)

General Properties of Graphs

What is the **max** number of edges for a *directed* graph of V vertices?

With no multi-edges, no self-loops.

Max: $\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \times 2 = \_\_\_\_\_\_\_\_\_\_\_\_\_\_$

Hint: Basically for each undirected edge, break it into 2 directed edges

General Properties of Graphs

What is the **max** number of edges for a *directed* graph of V vertices?

With no multi-edges, no self-loops.

Max: ${}^{V}C_{2} \times 2 = {}^{V}P_{2}$

Graph Modelling

Graph Modelling

The process of constructing a graph from other sources of information.

Vertex:

- What is represented by a vertex

Edges:

- What is represented by an edge
- When do you draw an edge

Real Life Applications－MRT network

K of the stations are affected by train faults.

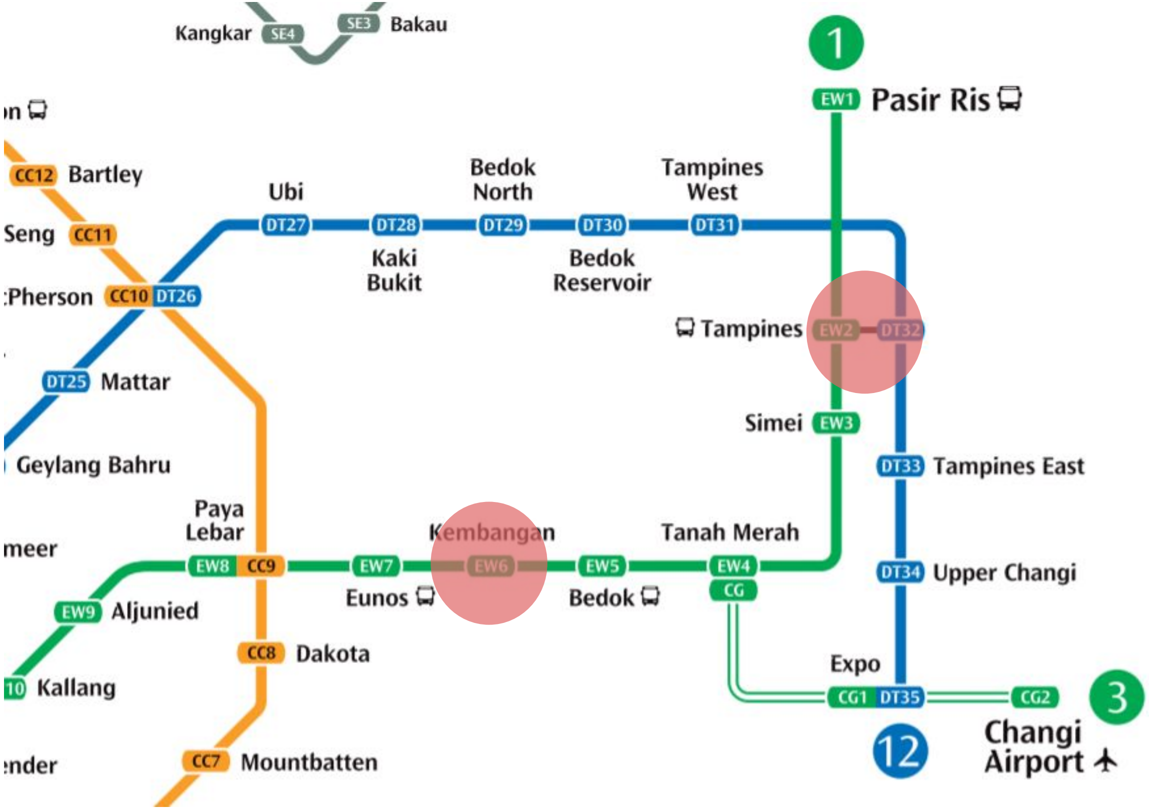
No MRT services to/from these stations.

Given 2 stations, **A** and **B**:

I start from station **A**, can I *still* reach station **B**?

Real Life Applications—MRT network

fault



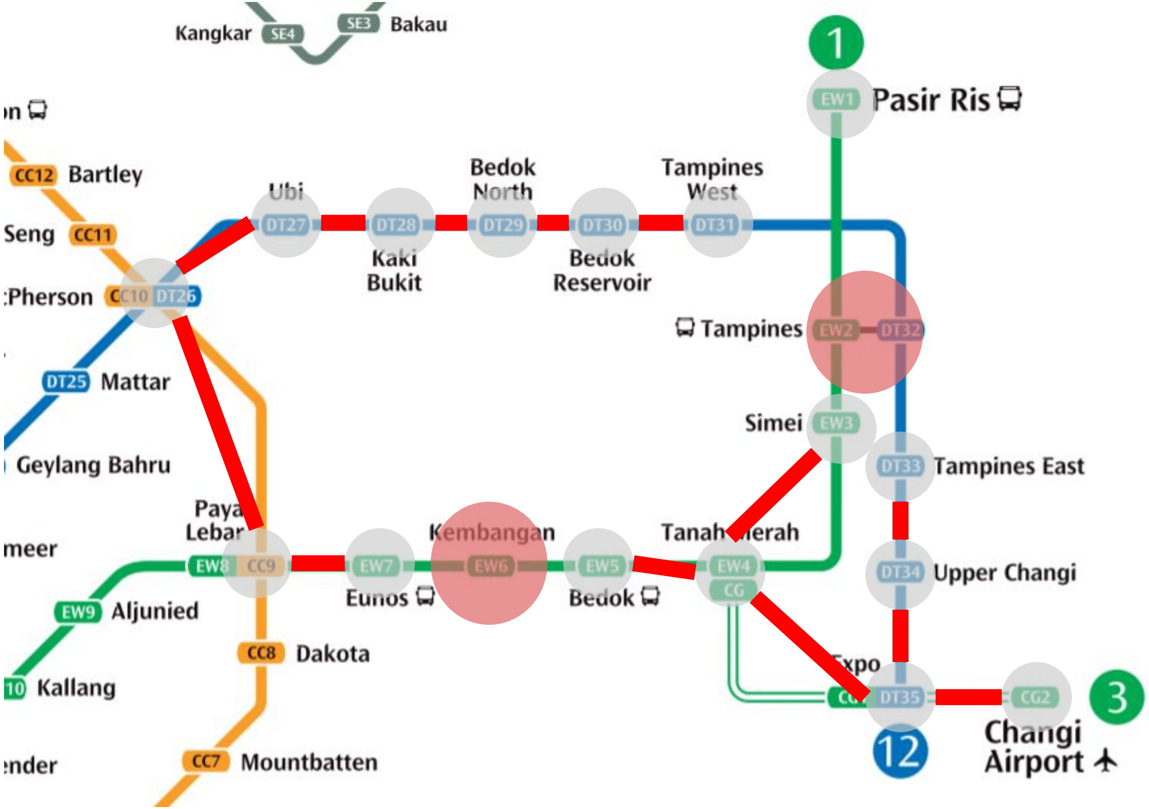
Real Life Applications — MRT network

Vertex: MRT stations

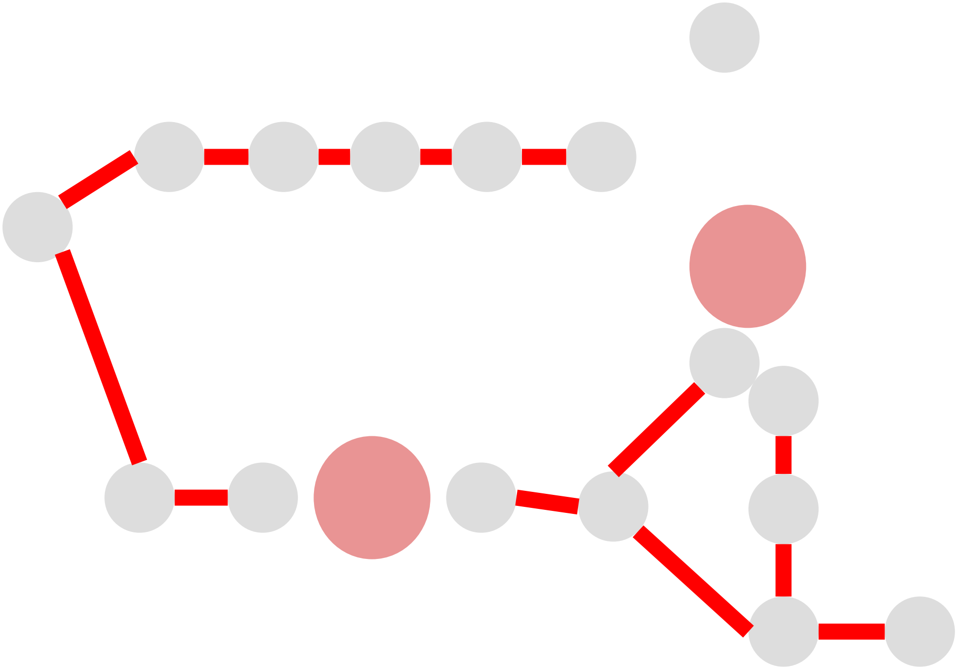
Edges: Draw an edge between 2 stations if they are adjacent along an MRT line **and** both stations are still operating

Algorithm: Check if 2 vertices are connected

Real Life Applications — MRT network



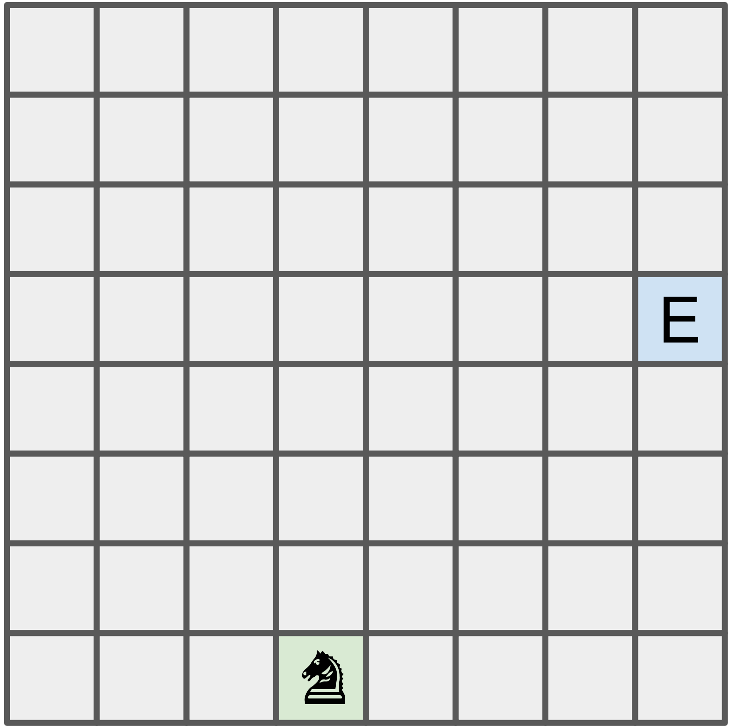
Real Life Applications—MRT network



Real Life Applications—Chessboard

We have a standard 8x8 chessboard and a knight ♞.

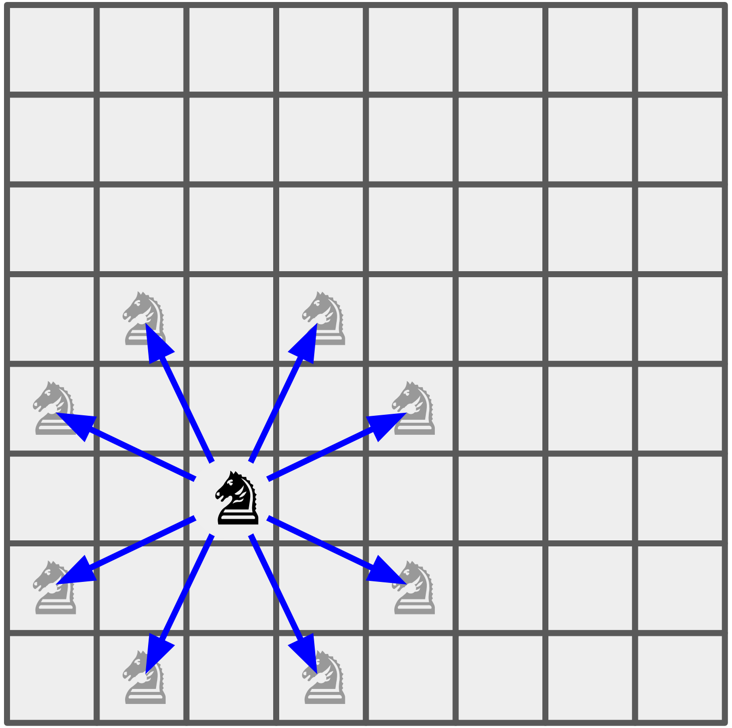
Can our ♞ reach E on this empty chessboard?



Real Life Applications—Chessboard

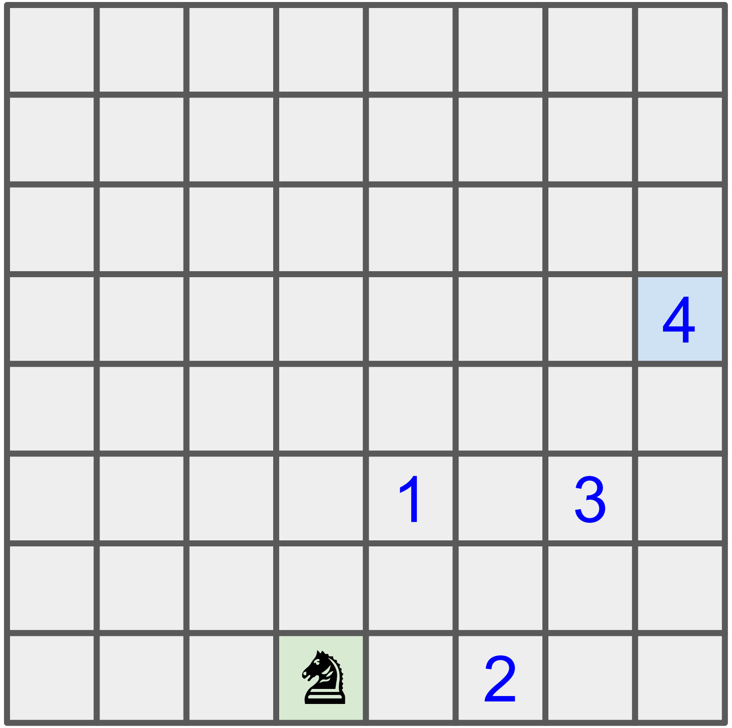
A knight moves in an L-shaped manner. So there are a maximum of 8 valid positions it can move to at any one step.

It cannot move to a valid position if any cell along its L-shaped path is blocked.



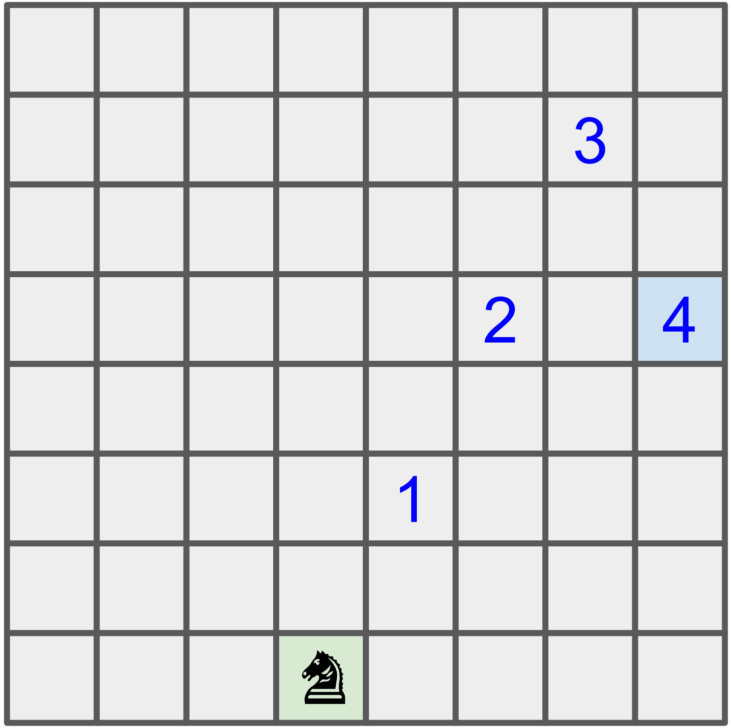
Real Life Applications — Chessboard

So yes our knight can reach cell E via these following steps numbered from 1 to 4!



Real Life Applications — Chessboard

There are many other possible solutions...



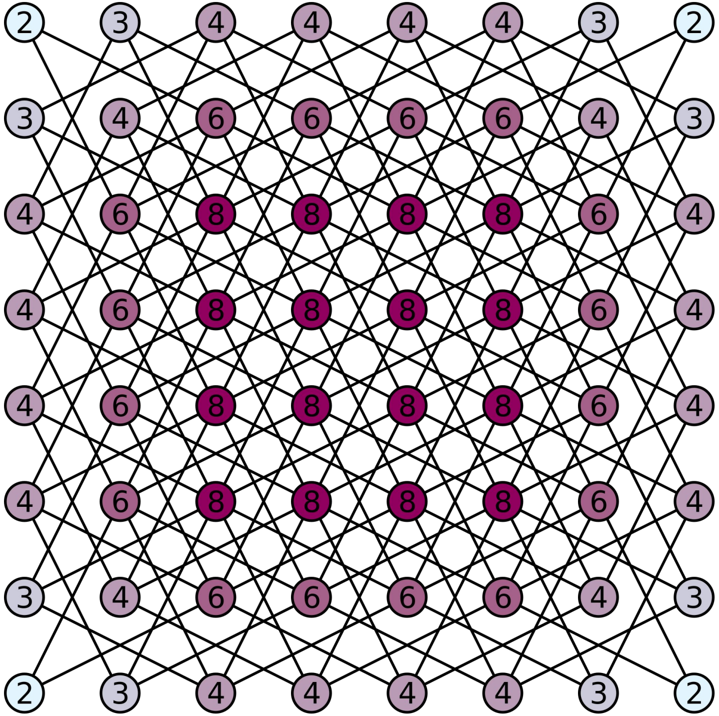
Real Life Applications—Chessboard

Here's a question

Unrestricted by the number of steps taken, can a knight eventually reach every cell on an empty chess board?

Real Life Applications — Chessboard

It turns out that the answer is yes! This is a well-studied problem known as the Knight's tour. Just FYI



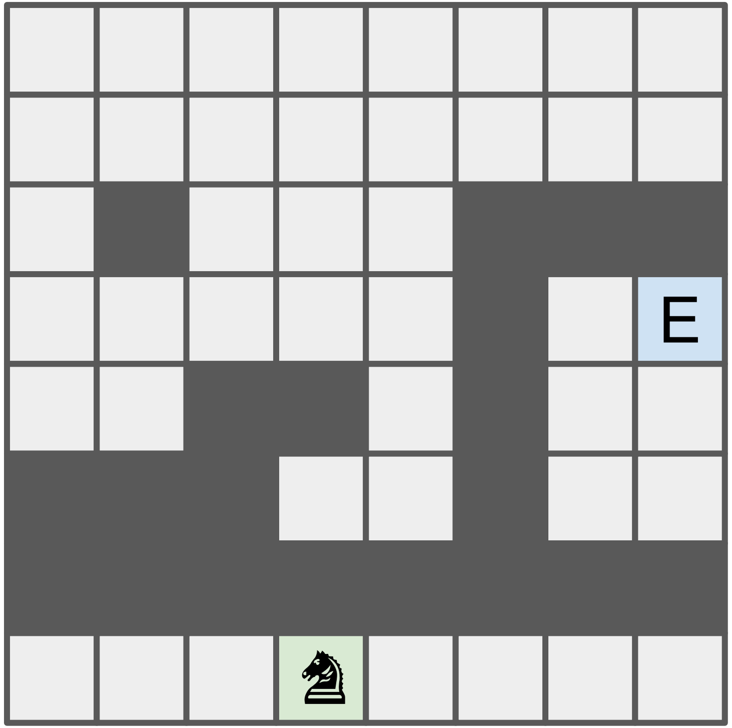
From Wikipedia

Real Life Applications — Chessboard

Now we *block* some cells and shade them on the chessboard.

Can our our knight still get to cell **E** without crossing any blocked cells along the way?

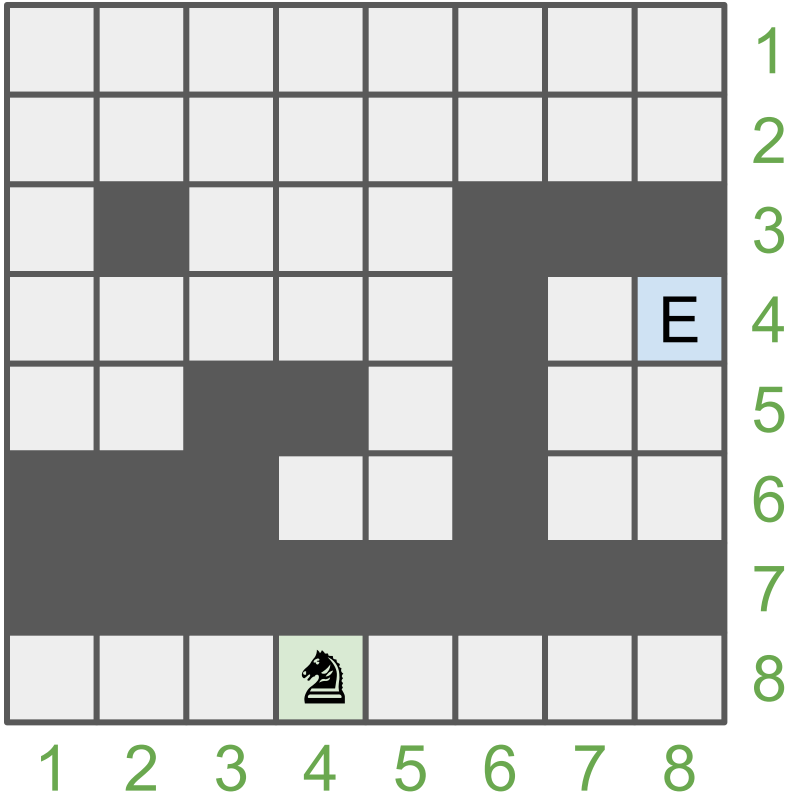
How can we model this?



Real Life Applications — Chessboard

Vertex:

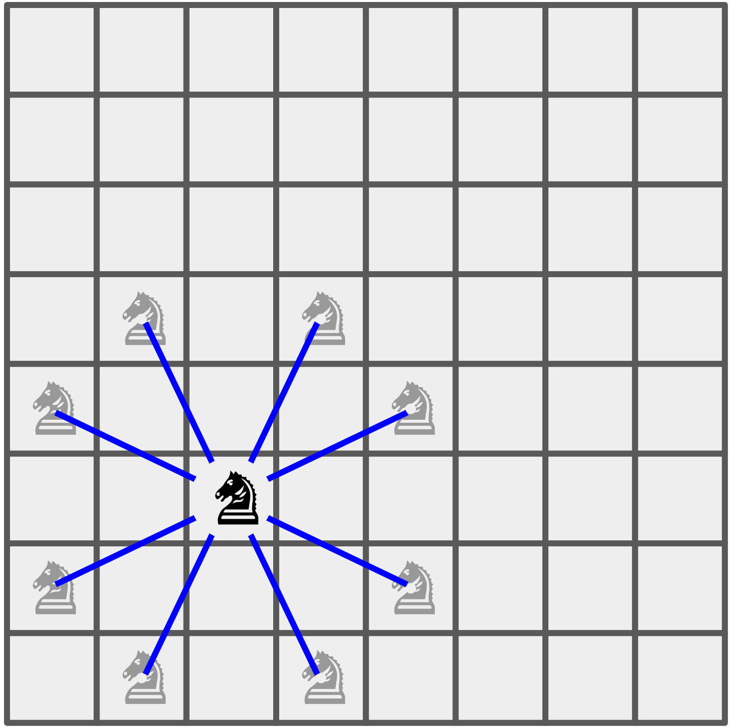
- Cells of the chessboard since the knight can possible reach every one of them
- Denoted by (row, col)



Real Life Applications — Chessboard

Edge:

- For every vertex, draw an undirected edge to 8 of its destination vertices, each representing a valid move
- We don't draw if a blocked cell exists along the L-shaped path

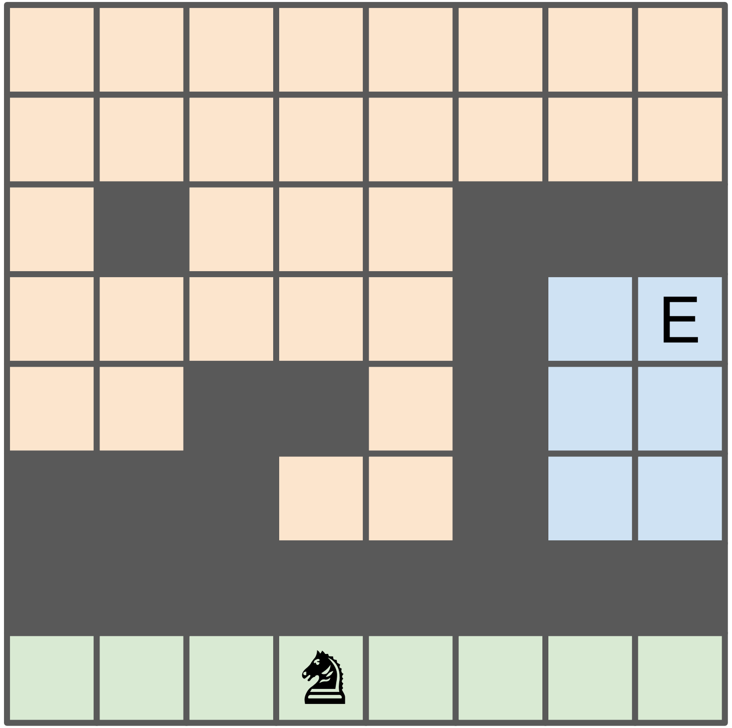


Real Life Applications — Chessboard

Flood Fill

We shall flood fill all empty cells adjacent to each other (top, down, left, right) with the same colour.

Which algorithm(s) can we modify to achieve this?

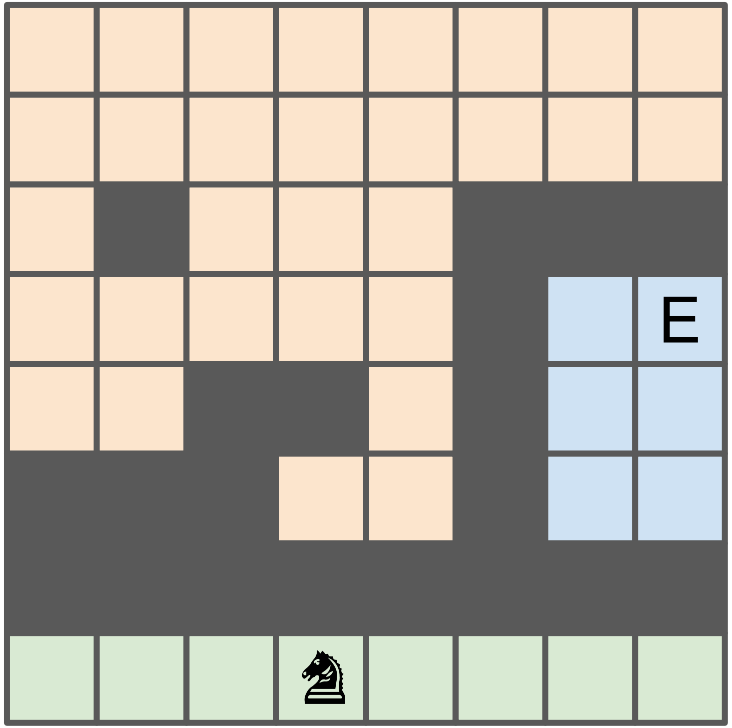


Real Life Applications—Chessboard

Flood Fill

In this case, how many different colours can you have excluding shaded cells?

3



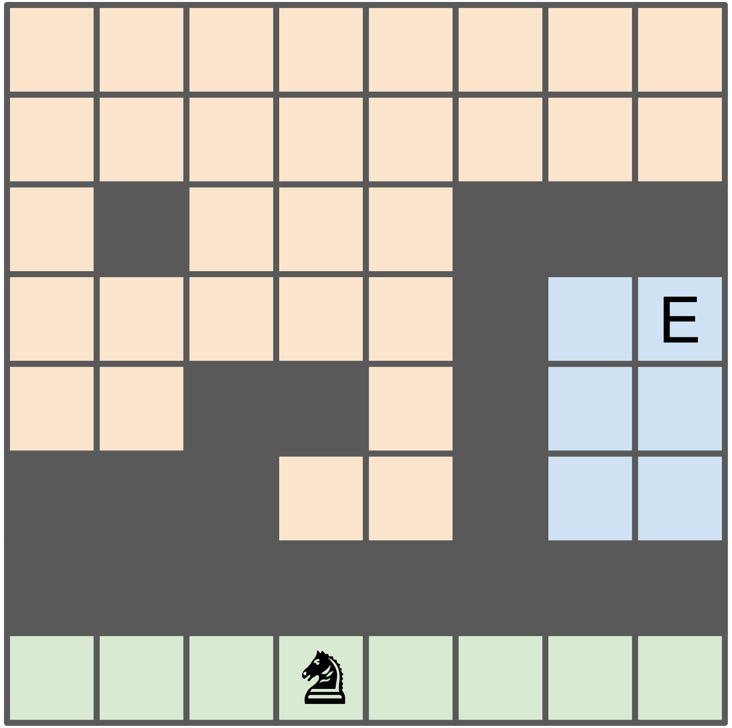
Real Life Applications — Chessboard

Flood Fill

How to calculate?

Find *connected components* (CC).

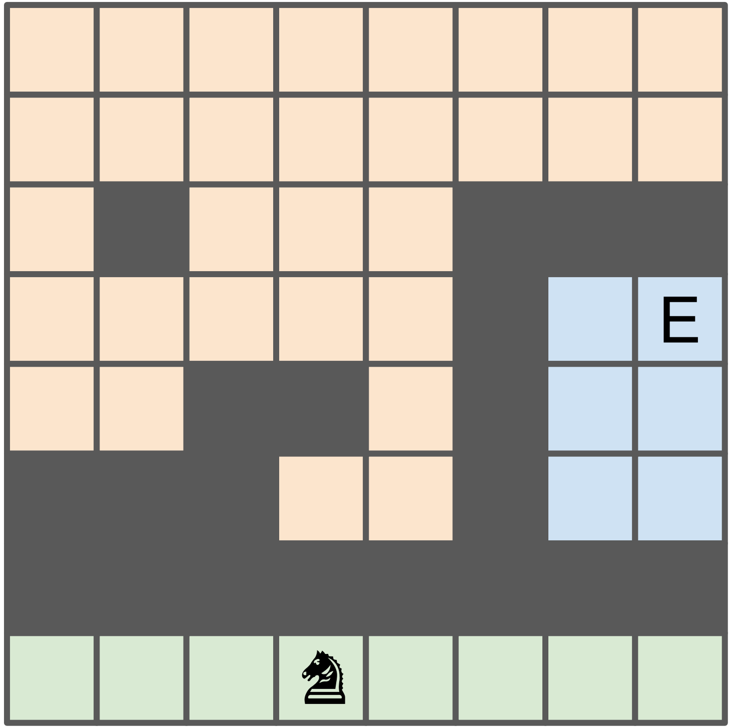
Each CC must share the same colour.



Real Life Applications－Chessboard

Flood Fill

If we transform into a graph, it will be a disjoint graph consisting of 3 separate CCs.



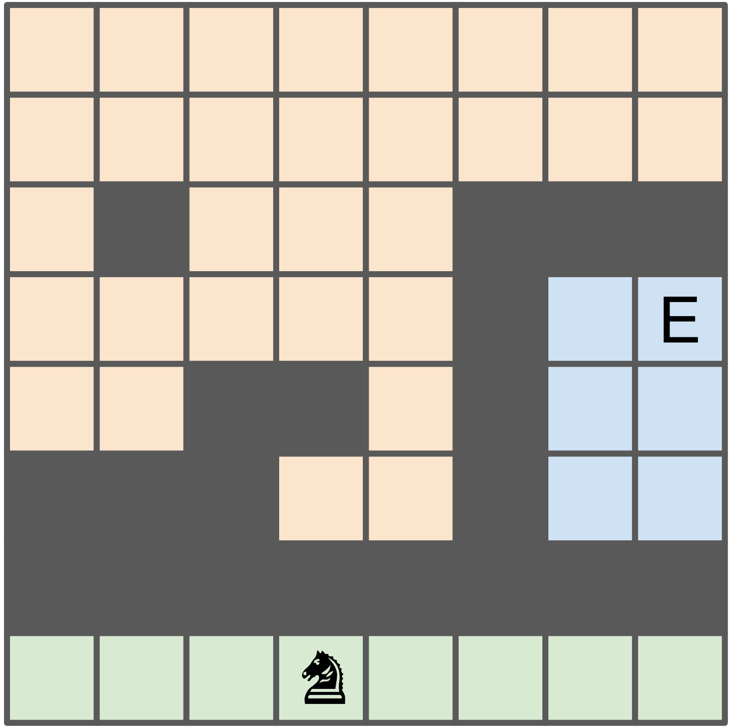
Real Life Applications—Chessboard

Flood Fill

Must we “encode” the graph in a AL/AM/EL?

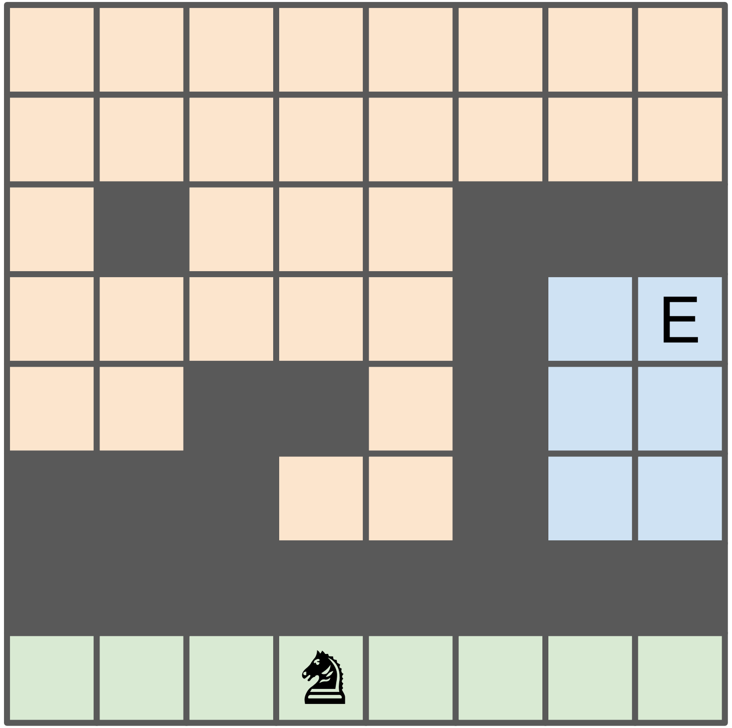
We can… but do we need to?

No! We can just work with this 2D array.



Real Life Applications — Chessboard

Our knight thus cannot reach cell E simply because they exists in separate CCs!



Graph Traversal

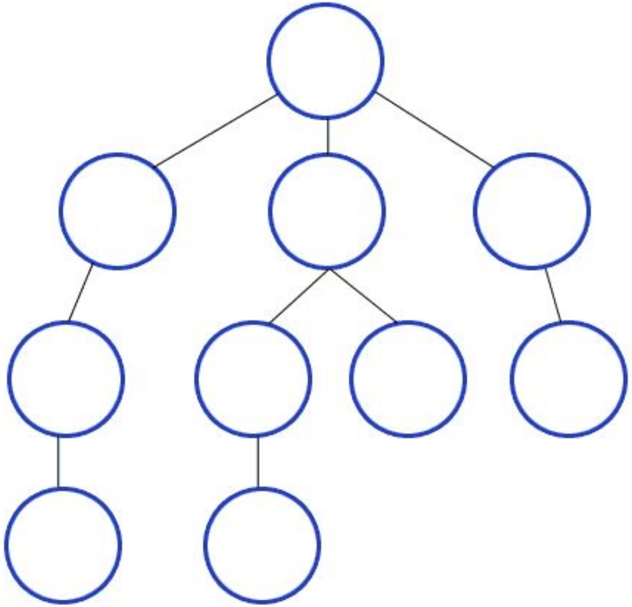
Dinner **F**irst; **S**leep

Breakfast **F**irst; **S**leep

Depth First Search (DFS)

Likes to go *deeper*!

It will only backtrack if there is no other way to continue deeper.



Note: This is an animated image and so it will not animate in pdf

Depth First Search — Pre-order Traversal

Recall that previously we introduced DFS for **binary trees** via the following pre-order traversal algorithm?

```
void dfs(vertex v) {
    cout << v.id << endl;
    if (v.left)  dfs(v.left);
    if (v.right) dfs(v.right);
}
```

How can we generalize this for **graphs**? i.e. When vertices are no longer restricted to having just 2 neighbours

Depth First Search — Pre-order Traversal

You may be tempted to just update it to the following. This is a classic mistake which will lead to infinite recursion! Why?

```
void dfs(int vert_id) {
    cout << vert_id << endl;
    for (int & nb_id : adjList[vert_id]){
        dfs(nb_id);
    }
}
```

Recurse down every neighbour

Depth First Search — Pre-order Traversal

We should only recurse further on unvisited vertices! Just ignore those which have been visited before!

```
void dfs(int vert_id) {
    if (visited[vert_id]) return;
    visited[vert_id] = true;
    cout << vert_id << endl;
    for (int & nb_id : adjList[vert_id]) {
        dfs(nb_id);
    }
}
```

Just fix by adding these 2 new lines!

Challenge yourself!

Implement DFS *without* recursion.

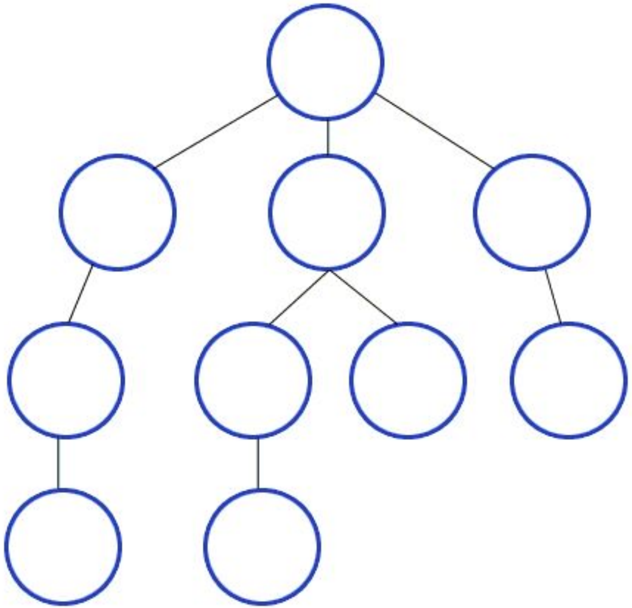
It is ok to not try this!

Hint: What DS have you learnt that exhibits recursive property?

Breadth First Search (BFS)—Level-order Traversal

Likes to proceed *radially*!

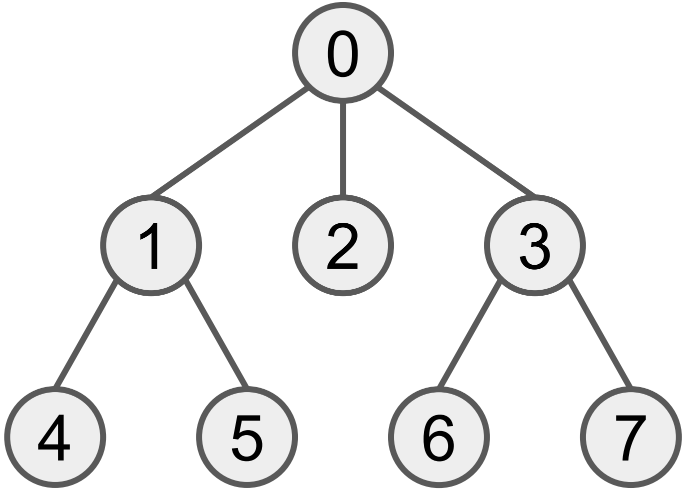
It will go deeper **only when it has cleared the current level/radius**.



Note: This is an animated image and so it will not animate in pdf

Breadth First Search—Level Order Traversal

```
queue<int> q;
visited[src_id] = true;
q.push(src_id);
while (!q.empty()) {
    int v_id = q.front(); q.pop();
    cout << v_id << endl; // operate on v_id
    for (int & nb_id: adjList[v_id]) {
        if (visited[nb_id]) continue;
        q.push(nb_id);
        visited[nb_id] = true;
    }
}
```

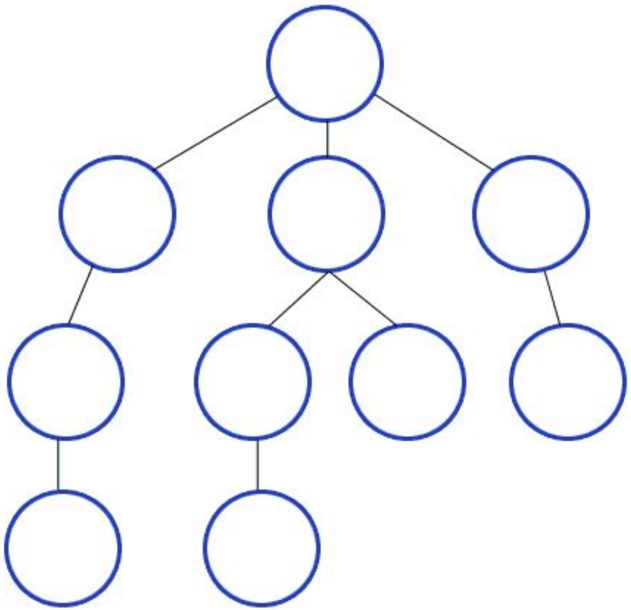


Output:

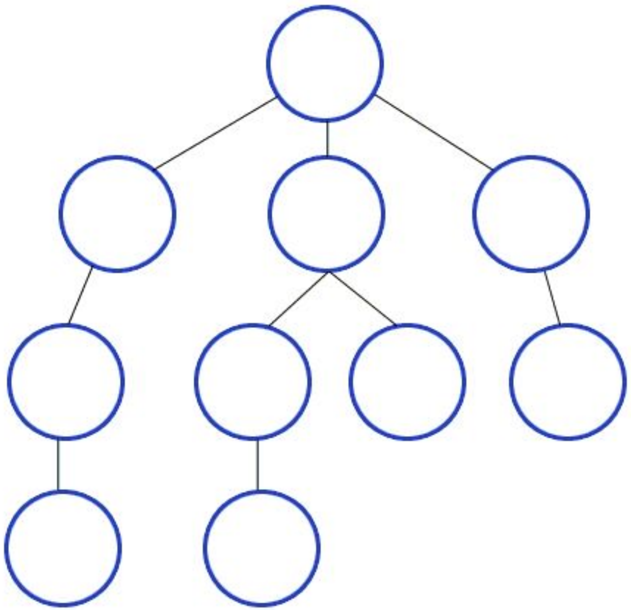
```
0, 1, 2, 3, 4, 5, 6, 7
```

Comparison

BFS



DFS



Note: These are animated images and so they will not animate in pdf

Practical Exam

STL Containers

Container	Vector		Stack Queue		Deque		List		Priority Queue		Set		Map	
Insert	`push_back`	$O(1)$	`push`	$O(1)$	`push_back` `push_front`	$O(1)$	`push_back` `push_front`	$O(1)$	`push`	$O(\log N)$	`insert`	$O(\log N)$	`[ ] operator`	$O(\log N)$
Delete	`pop_back`	$O(1)$	`pop`	$O(1)$	`pop_front` `pop_back`	$O(1)$	`pop_front` `pop_back`	$O(1)$	`pop`	$O(\log N)$	`erase`	$O(\log N)$	`erase`	$O(\log N)$
Random Access	`[ ] operator`	$O(1)$	NIL		`[ ] operator`	$O(1)$	`Loop Through`	$O(N)$	NIL		`find`	$O(\log N)$	`[ ] operator`	$O(\log N)$
Access	`front` `back`	$O(1)$	`s.top` `q.front`	$O(1)$	`front` `back`	$O(1)$	`front` `back`	$O(1)$	`top`	$O(1)$	NIL (Use iterators)		NIL (Use iterators)	
Sorted	No (Use STL sort)		No		No (Use STL sort)		No (Use List.sort)		**Yes**		**Yes**		**Yes**	
Binary Search	`lower_bound` `upper_bound`	$O(\log N)$	NIL		`lower_bound` `upper_bound`	$O(\log N)$	NIL		NIL		`lower_bound` `upper_bound`	$O(\log N)$	`lower_bound` `upper_bound`	$O(\log N)$
Unique	No		No		No		No		No		Yes (Use Multiset for non-unique keys)		Unique **keys** Non-unique values (Use Multimap for non-unique keys)	
Iterators	**Yes**		No		**Yes**		**Yes**		No		**Yes**		**Yes**	

Iterators behave like pointers (Credits: NOI 2015 Dec Training Team)

```
vector<int> v;
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << endl;

set<int> s;
for (set<int>::iterator it = s.begin(); it != s.end(); ++it)
    cout << *it << endl;

map<string, int> m;
for (map<string, int>::iterator it = m.begin(); it != m.end(); ++it)
    cout << "Key: " << it->first << ", value: " << it->second << endl;
```

C++11 auto (Credits: NOI 2015 Dec Training Team)

```
vector<int> v;
for (auto it = v.begin(); it != v.end(); ++it)
    cout << *it << endl;

set<int> s;
for (auto it = s.begin(); it != s.end(); ++it)
    cout << *it << endl;

map<string, int> m;
for (auto it = m.begin(); it != m.end(); ++it)
    cout << "Key: " << it->first << ", value: " << it->second << endl;
```

C++11 range-based loops (Credits: NOI 2015 Dec Training Team)

```
vector<int> v;
for (int it : v)                         // Pass by copy
    cout << it << endl;

set<int> s;
for (int it : s)                         // Pass by copy
    cout << it << endl;

map<string, int> m;
for (pair<string, int> it : m)           // Pass by copy
    cout << "Key: " << it.first << ", value: " << it.second << endl;
```

C++11 range-based loops (Credits: NOI 2015 Dec Training Team)

```
vector<int> v;
for (auto it : v)                    // Pass by copy
    cout << it << endl;

set<int> s;
for (auto it : s)                    // Pass by copy
    cout << it << endl;

map<string, int> m;
for (auto it : m)                    // Pass by copy
    cout << "Key: " << it.first << ", value: " << it.second << endl;
```

C++11 range-based loops (Credits: NOI 2015 Dec Training Team)

```
vector<int> v;
for (int &it : v)
    cout << it << endl;

set<int> s;
for (const int &it : s)
    cout << it << endl;

map<string, int> m;
for (const pair<string, int> &it : m)
    cout << "Key: " << it.first << ", value: " << it.second << endl;
```

C++11 range-based loops (Credits: NOI 2015 Dec Training Team)

```
vector<int> v;
for (auto &it : v)
    cout << it << endl;

set<int> s;
for (auto &it : s)
    cout << it << endl;

map<string, int> m;
for (auto &it : m)
    cout << "Key: " << it.first << ", value: " << it.second << endl;
```

Binary Search in STL containers (Credits: RI Oct 2016 Training Team)

```
vector<int> v; // v = [2, 5, 7, 9, 10]
cout << *lower_bound(v.begin(), v.end(), 7) << endl; //prints 7
cout << *upper_bound(v.begin(), v.end(), 7) << endl; //prints 9

set<int> s; // s = {2, 5, 7, 9, 10}
set<int>::iterator it = s.lower_bound(7); //*it = 7
it = s.upper_bound(7); //*it = 9

map<string, int> m; // m = {"Hello" = 5, "Kitty" = 2, "World" = 17}
map<string, int>::iterator it2 = m.lower_bound("Hello");
it2 = m.upper_bound("Hello");
```

Summary of STL Iterators

`Iterator, it`	`vector<value>::iterator`		`deque<value>::iterator`		`list<value>::iterator`		`set<key>::iterator`		`map<key, value>::iterator`	
Value of `*it`	value		value		value		key		pair(key, value)	
Insert	insert	**O(N)**	insert	**O(N)**	insert	**O(1)**	NIL		NIL	
Delete	erase	**O(N)**	erase	**O(N)**	erase	**O(1)**	erase	**O(logN)**	erase	**O(logN)**
Update	*it = new value	**O(1)**	*it = new value	**O(1)**	*it = new value	**O(1)**	NIL (delete & insert instead)		it->second = new value	**O(logN)**
Traversal	**O(1) per it++**		**O(1) per it++**		**O(1) per it++**		**O(logN) per it++**		**O(logN) per it++**	

C++ typedef

```
#include <bits/stdc++.h>
using namespace std;
pair<long long, long long> f (long long x) {
    return make_pair(x-1, x+1);
}
```

C++ typedef

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
pair<ll, ll> f (ll x) {
    return {x-1, x+1};
}
```

C++ typedef

```
#include <bits/stdc++.h>
using namespace std;

vector<pair<int, int>> v;
map<pair<int, int>, pair<int, int>> m;

vector<pair<int, int>>::iterator itv = v.begin();
map<pair<int, int>, pair<int, int>>::iterator itm = m.begin();
```

C++ typedef

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pi;
vector<pi> v;
map<pi, pi> m;

vector<pi>::iterator itv = v.begin();
map<pi, pi>::iterator itm = m.begin();
```

C++ typedef

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pi;
vector<pi> v;
map<pi, pi> m;

auto itv = v.begin();
auto itm = m.begin();
```

C++ typedef

```
typedef pair<int, int> pi;
typedef pair<pi, pi> pipi;

pi a(3, 5);
pi b = make_pair(7, 0);
pi c = pi(7, 3);
pipi ab = make_pair(a, b);
```

C++ typedef

```
#include <bits/stdc++.h>
using namespace std;
typedef tuple<int, string, int, string> isis;

isis a = make_tuple(0, "a", 1, "b");
isis b = isis(0, "a", 1, "b");

vector<isis> v;
set<isis> s; //tuple and pairs have default comparators
// warning on typedef: May NOT be good for Software Engineering
```

Max/Min/Arithmetic

```cpp
int a = 3, b = 7;

int x = min(a, b);
int y = max(a, b);

x += 2;          // x = x + 2
b -= a;          // b = b - a
y *= x;          // y = y * x
y %= 4;          // y = y % 4
a /= 2;          // a = a / 2
```

Input / Output

- How to input an entire line.
  - And how to input space separated variables from an inputted line
- How to input strings
- How to output space separated variables on a single line

Implementation/Debugging Tips

- *'Binary Search'* your code (if Runtime Error)
  - Terminate it after running half of your code.
  - Commenting out suspected problematic parts.
- Replace the segment with 'non-optimized' version, see if it results in the same output
- Compile regularly
- **D**on't **R**epeat **Y**ourself (DRY principle)

Sit-in/Practical Exam Tips

- **<u>Plan</u>** what you want to code
    - Don’t dive into the code immediately
- Partition the task into subtasks
    - Handle them one by one
    - Modular Programming

```
/* Deduplicate the array using unordered_set */

/* Sort the array */

/* Find maximum of ... */
```

Sit-in/Practical Exam Tips

- Clarify when in doubt
- Be suspicious of any *weird* limits
  - Remember *long long*?
- More <u>**practice**</u> → code faster
  - Range based for-loops
  - STL Data Structures

Sit-in/Practical Exam Tips

When stuck (first half):

1. **Don't panic**
2. Rethink the problem from another angle
   a. Each vertex can become more vertices?
   b. Restrict direction of edge? Flip direction?
   c. Not a graph question?
3. Data structure**s** are your friend :)

Sit-in/Practical Exam Tips

When stuck (second half):

1. **Don't panic (that much)**
2. Damage control
   a. "Fastest-to-code" implementation
   b. Handle **general case** first, abandon corner cases
   c. Try small cases
   d. Make the code "look" similar to what you think it is **:X** (aka try to scam the marker...)