CS2040C Tut 9

Graph Representation & Modelling
Graph Traversal
Practical Exam

Graph Representation

Adjacency Matrix

Adjacency List

Edge List

Graph Representation

Edge List

A list of edges of the entire graph.

Adjacency Matrix

2D matrix where matrix[x][y] stores information about edge $x \to y$ (or information that it does not exist).

Adjacency List

For each vertex, keep a list of edges it has.

Graph Representation: Edge List

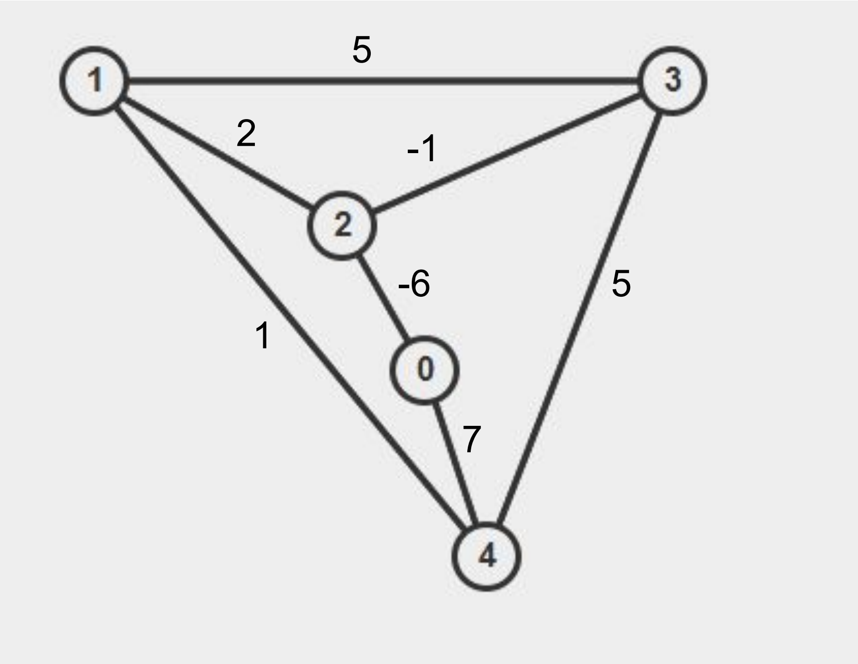


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

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

Graph Representation: Edge List

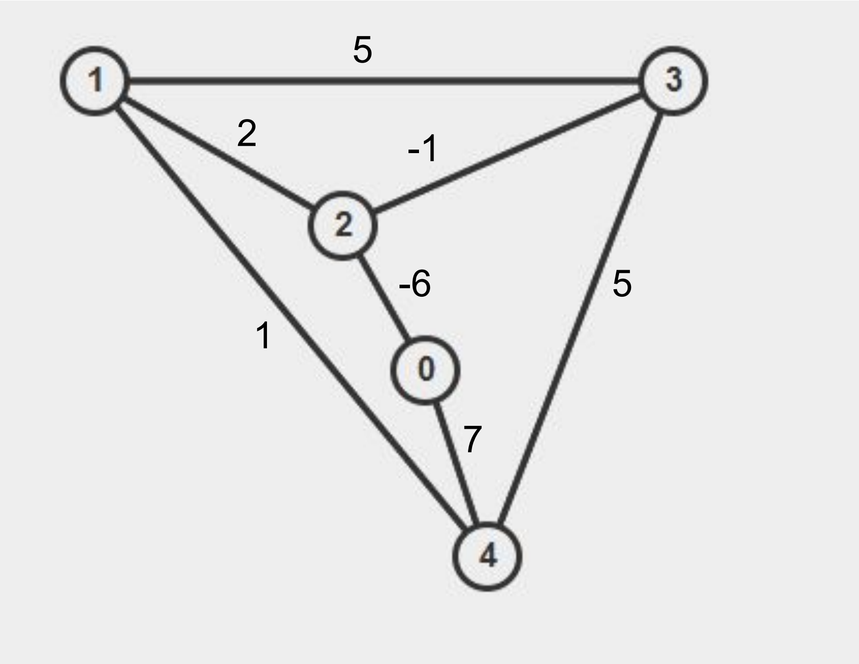


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

(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: Adjacency Matrix

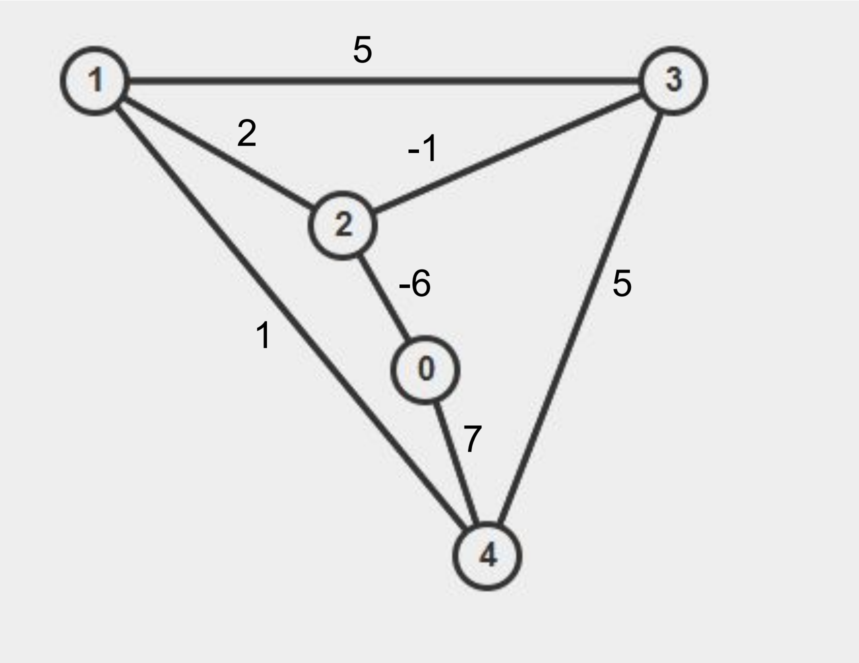


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

-	0	1	2	3	4
0	-				
1		-			
2			-		
3				-	
4					-

Graph Representation: Adjacency Matrix

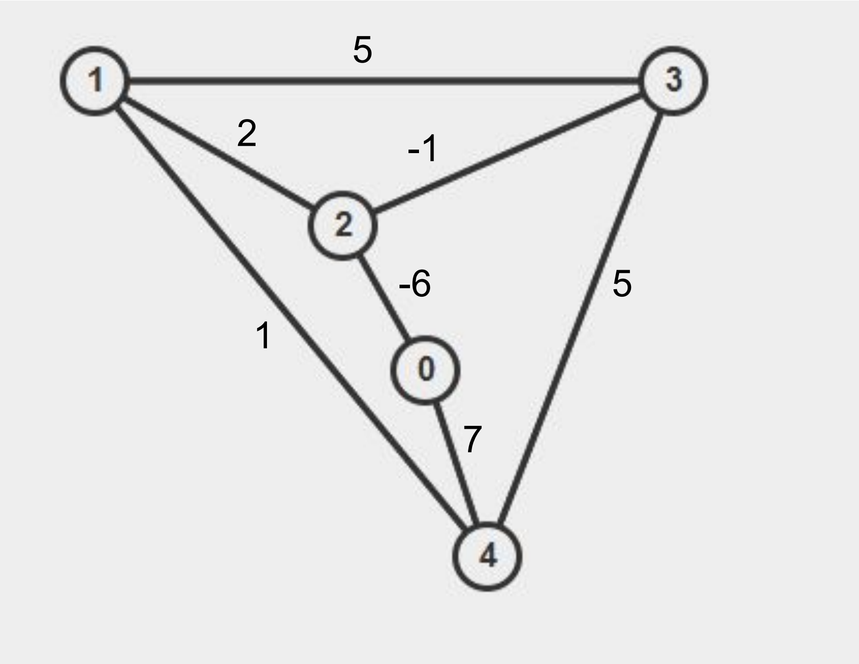


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

-	**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

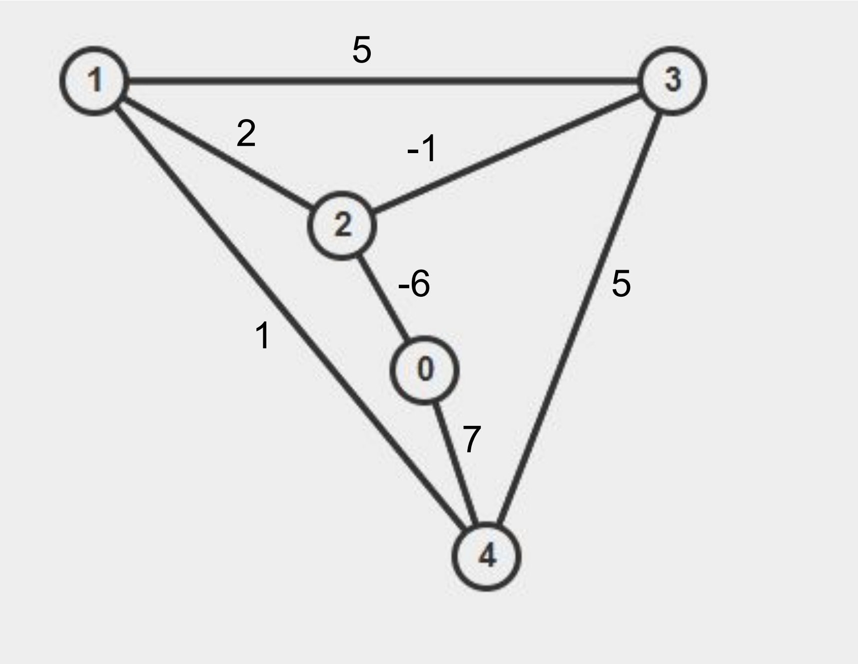


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

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

Symmetrical if bidirectional.

Graph Representation: Adjacency List

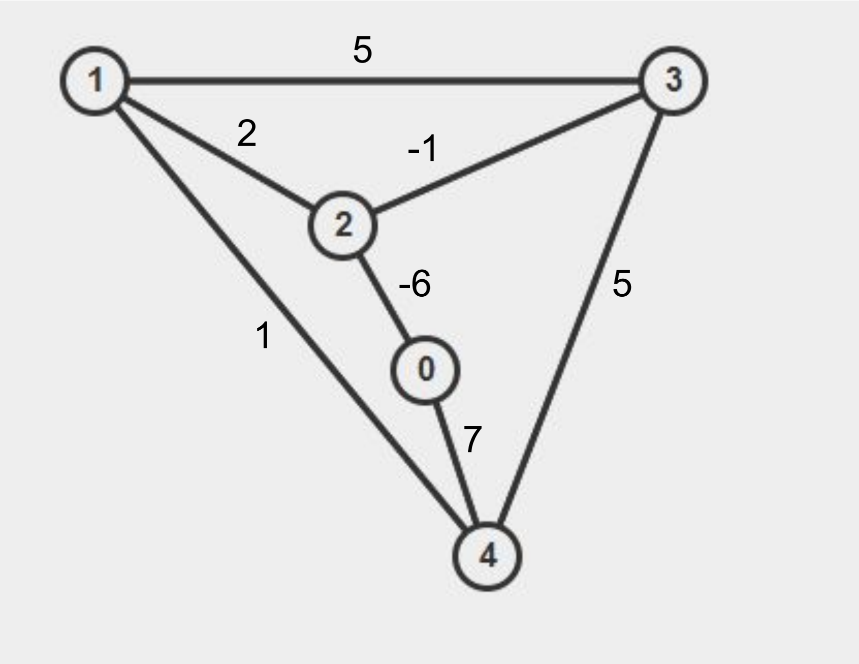


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

Vertex	List (Vertex, Value)
0	
1	
2	
3	
4	

Graph Representation: Adjacency List

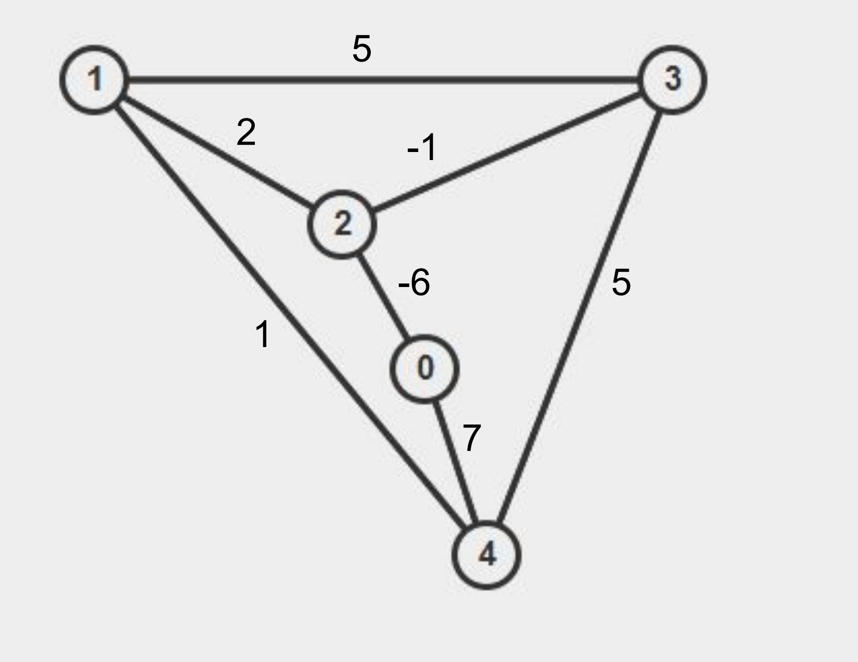


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

Vertex	List (Vertex, Value)
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

Representing a tree

What is the nicest way to store a tree? :O

How many parent does each vertex have?

What if all edges are from child → parent?

Graph Representation: Parent *array*

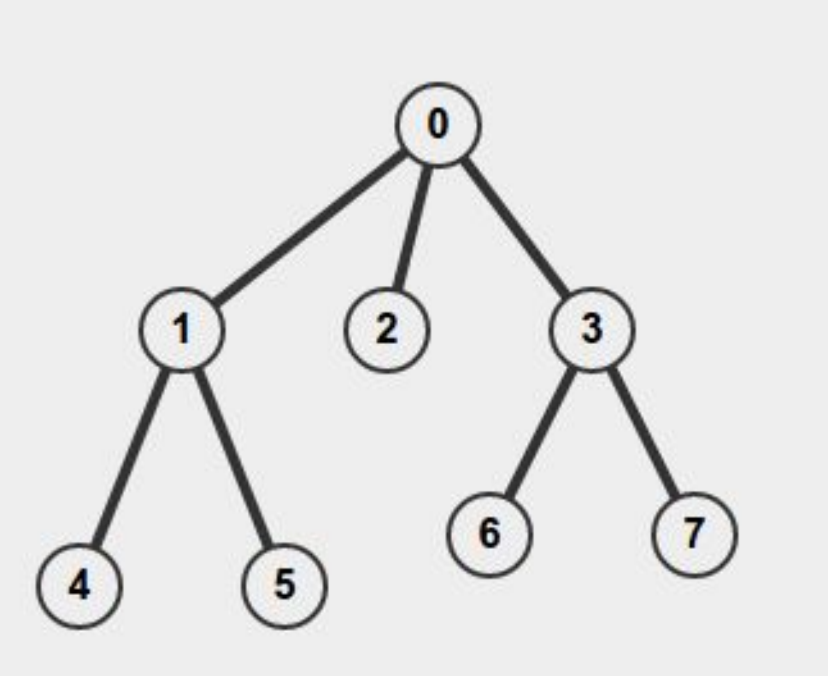


Image taken from NOI 2015 Dec Training, which was taken from **VisuAlgo** *very* long time ago.

Vertex	Parent (Vertex)
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

Yes. that's it.

Directed Acyclic Graph (DAG)

Definition

- **Directed**
  - Edges are **not** bidirectional.
- **Acyclic** $\rightarrow$ No cycles.
  - No self-loops.
- **Graph**

Directed Acyclic Graph (DAG)

Properties

After traversing an edge from vertex **u** → **v**,

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

Can you prove it? [By contradiction]

Q2

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

How to start?

Look at the **V-1** or **V+1** case,

see what the formula tells us.

Q2

V-1 vertices → **(V-1)(V-2)/2** directed edges

V vertices → **(V)(V-1)/2** directed edges.

V+1 vertices → **(V+1)(V)/2** directed edges.

From **(V-1)** → **V**: + **(V-1)** edges, +1 vertex.

From **V** → **(V+1)**: + **V** edges, +1 vertex.

Q2

Observation

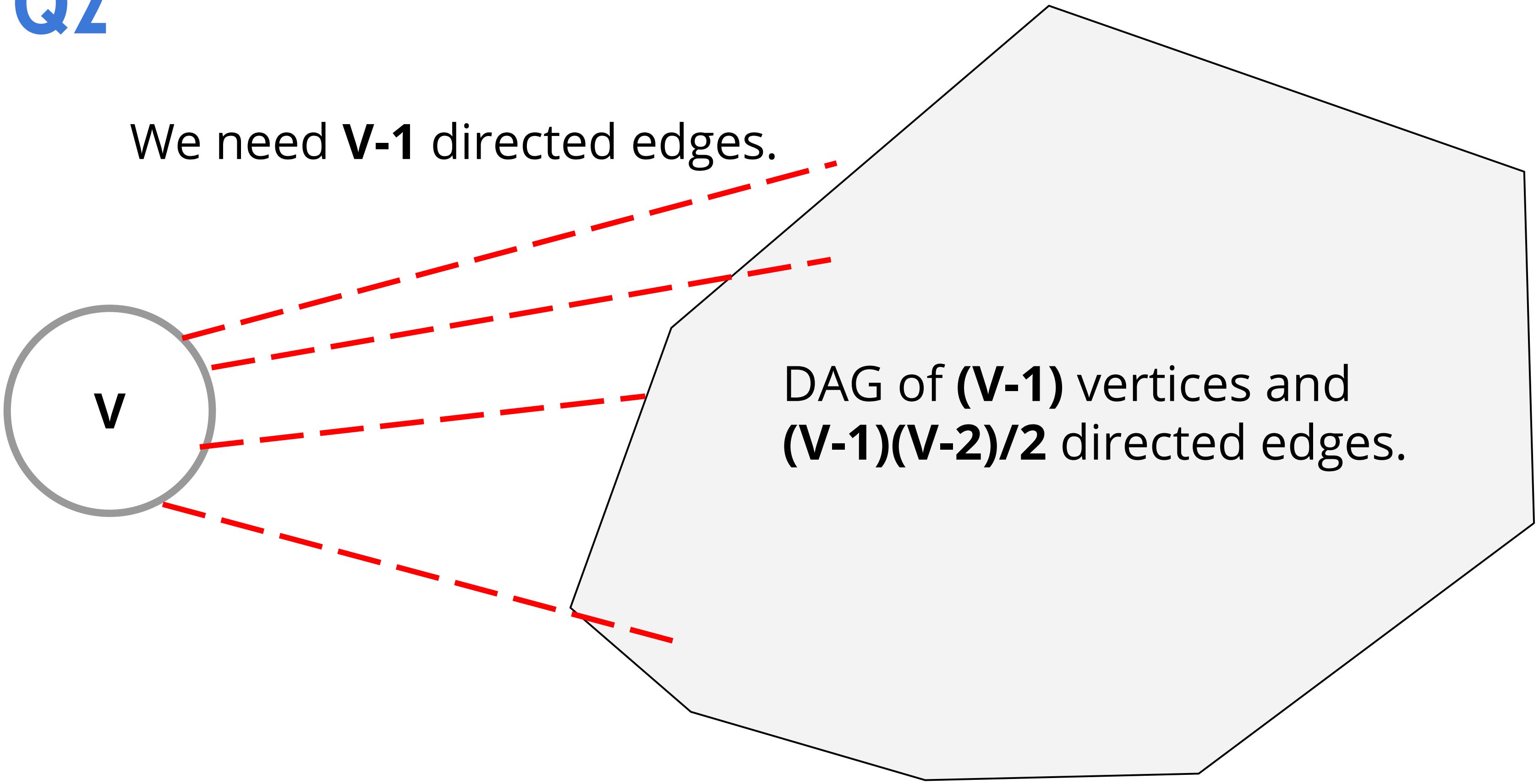
V-1 vertices → **(V-1)(V-2)/2** directed edges

V vertices → **(V)(V-1)/2** directed edges.

Assuming we have a way to draw for **V-1**, we can add additional 1 vertex just and add **(V-1)** more directed edges and maintain DAG property.

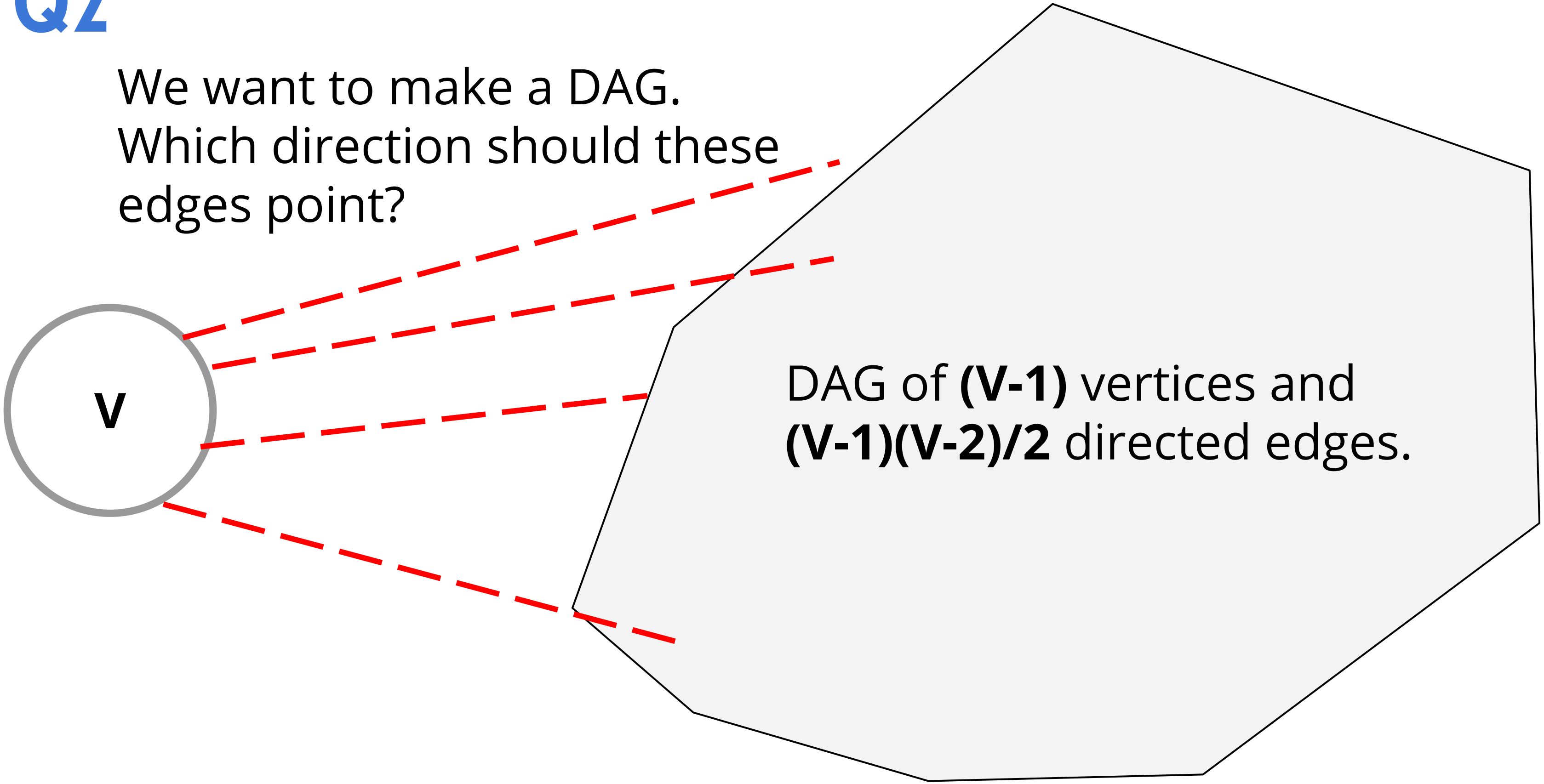
Q2

We need **V-1** directed edges.



Q2

We want to make a DAG.
Which direction should these
edges point?



Q2

Construction

We can now start with a graph with only 1 vertex and 0 edges.

Note that it is a DAG.

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

Q2

Construction

Label the first vertex as 1.

For each vertex **x** from **2** to **V**,

- Draw a *directed* edge from vertex **x** to those vertices < **x**.

Q2

On the edge

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

Q2

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*.

Bipartite Graph

<u>**Bi**</u>-<u>**part**</u>ite

Bipartite Graph

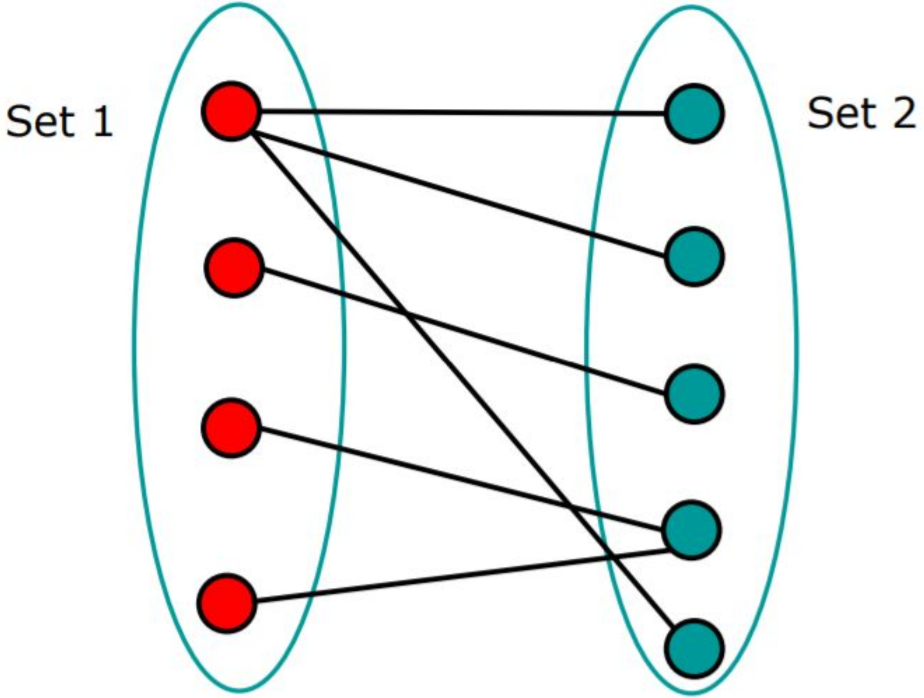
Definition

- A graph which vertices can be partitioned into 2 disjoint sets such that …
- There are **no** edges between vertices in the same set.

Bipartite Graph

Definition

- No edges between set 1 and set 1.
- No edges between set 2 and set 2.



Q3

Draw a bipartite graph with $\mathbf{V}$ vertices and $\mathbf{V^2/4}$ undirected edges.

Q3

Bipartite graph must have 2 sets of vertices, and only edges in between.

Lets say we have **X** vertices in the first set (left).

We will have **(V-X)** vertices on the right.

The max number of edges is then **X(V-X)**.

Q3

If … we construct a graph with too many edges, we can just omit them in the final answer :).

So lets try to maximize the number of edges!

Maximize: $\mathbf{X(V\text{-}X)}$

$$= VX - X^2$$

Q3

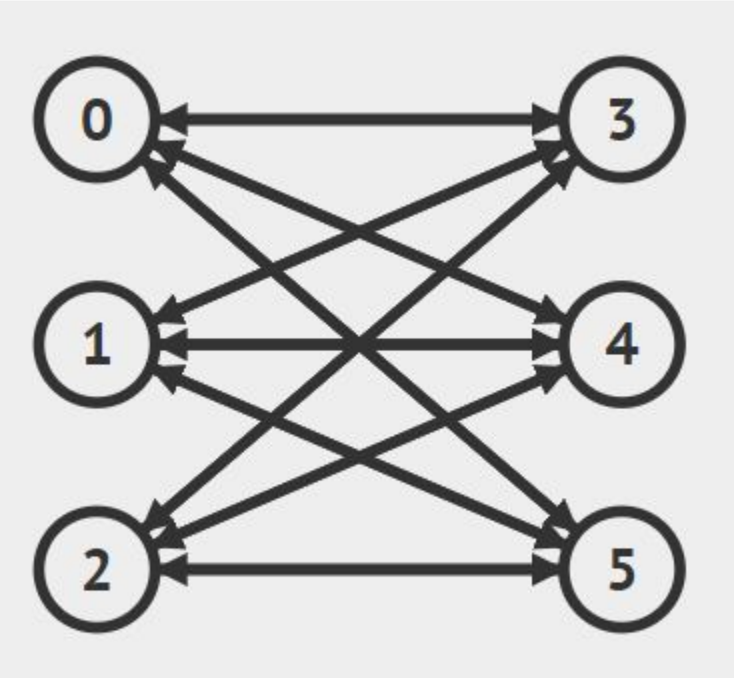
Maximize: $\mathbf{X(V\text{-}X)}$

$$
\begin{aligned}
&= VX - X^2 \\
&= -(X^2 - VX) \\
&= -(\ (X - V/2)^2 - V^2/4) \\
&= -\ (X - V/2)^2 + V/4
\end{aligned}
$$

Maximized when **X = V/2** :)

Q3

V/2 **V/2**



Edges

$= (V/2)*(V/2)$

$= V^2/4$

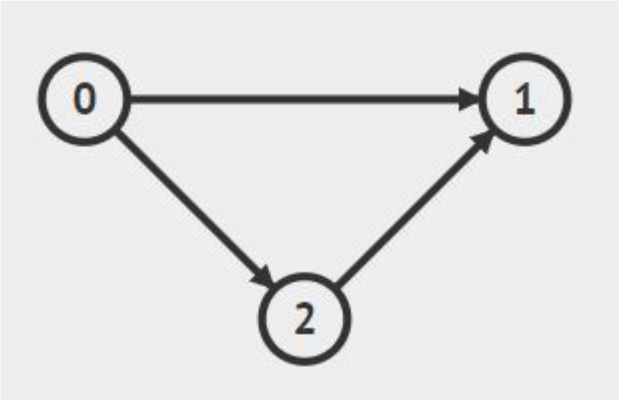
Are all DAGs a valid Bipartite graph?

If True, proof it.

If False, give an counter-example.

Are all DAGs a valid Bipartite graph?

False, counter example:



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.

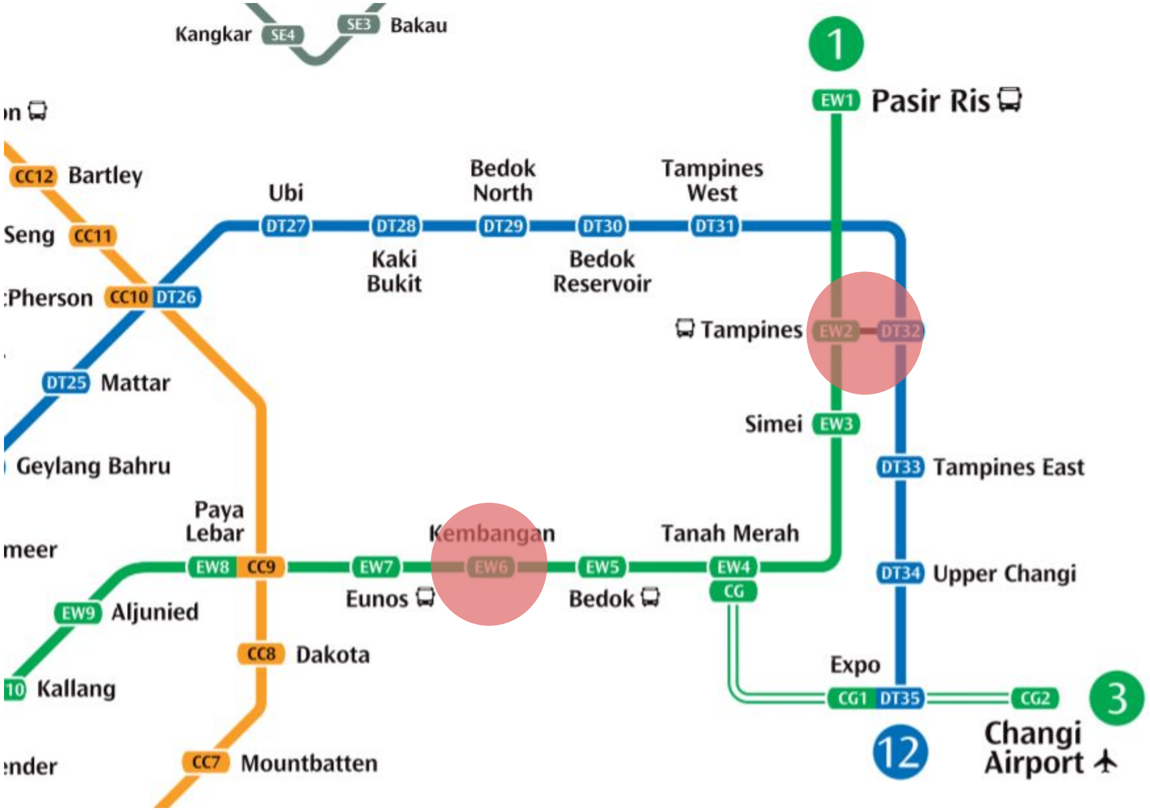
Given 2 stations, **A** and **B**:

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

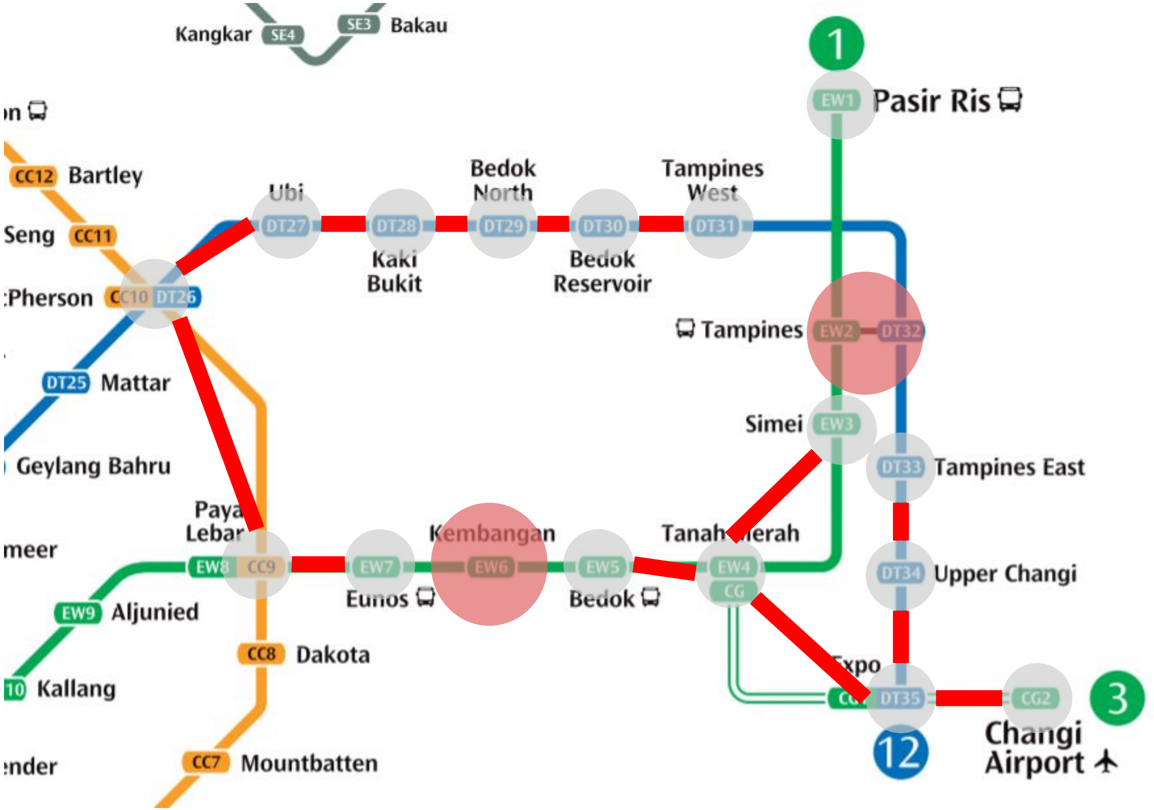
Edges: Draw an edge between 2 stations if they are adjacent in a MRT line

and both stations are still functional

Algorithm: Check if 2 vertices are connected

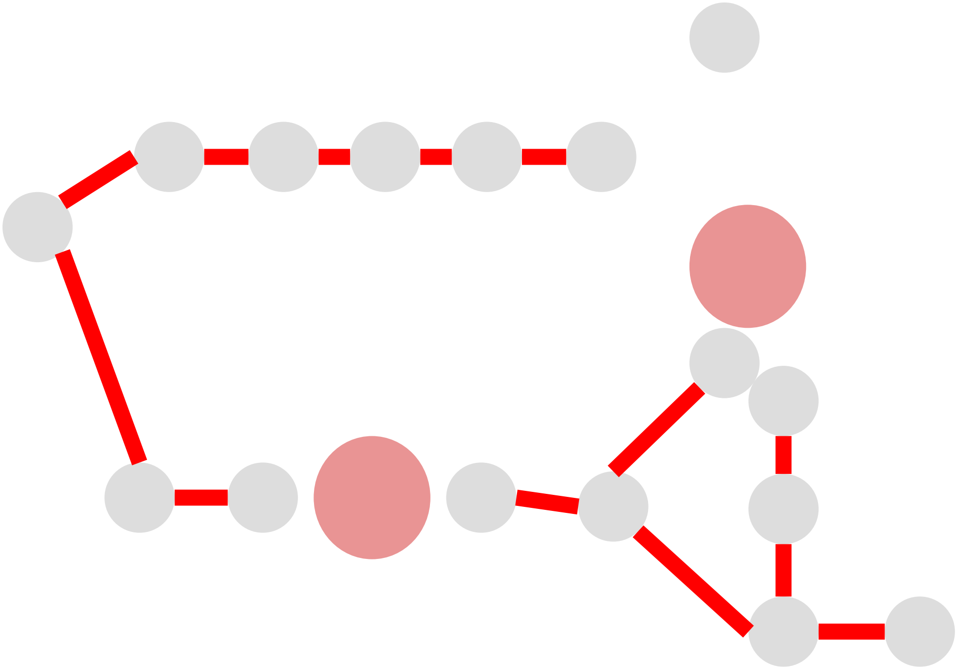
Real Life Applications

MRT network



Real Life Applications

MRT network

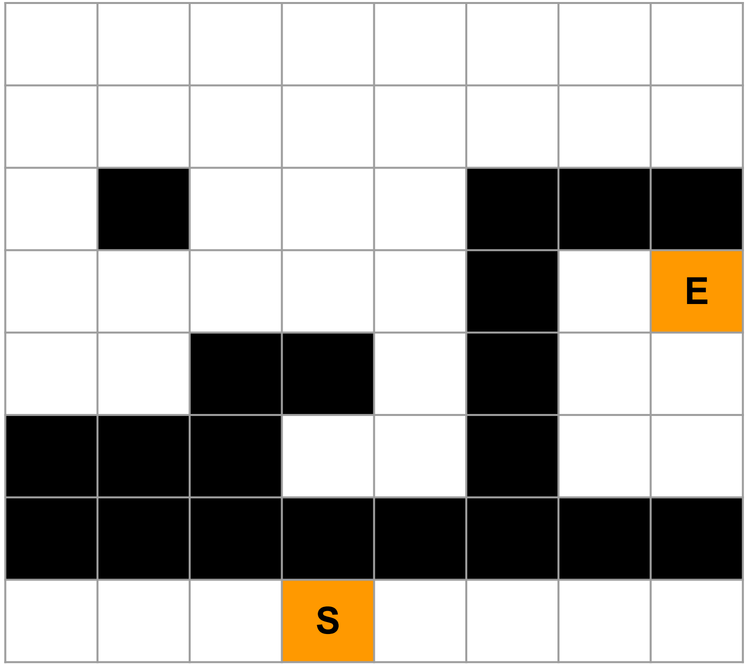


Real Life Applications

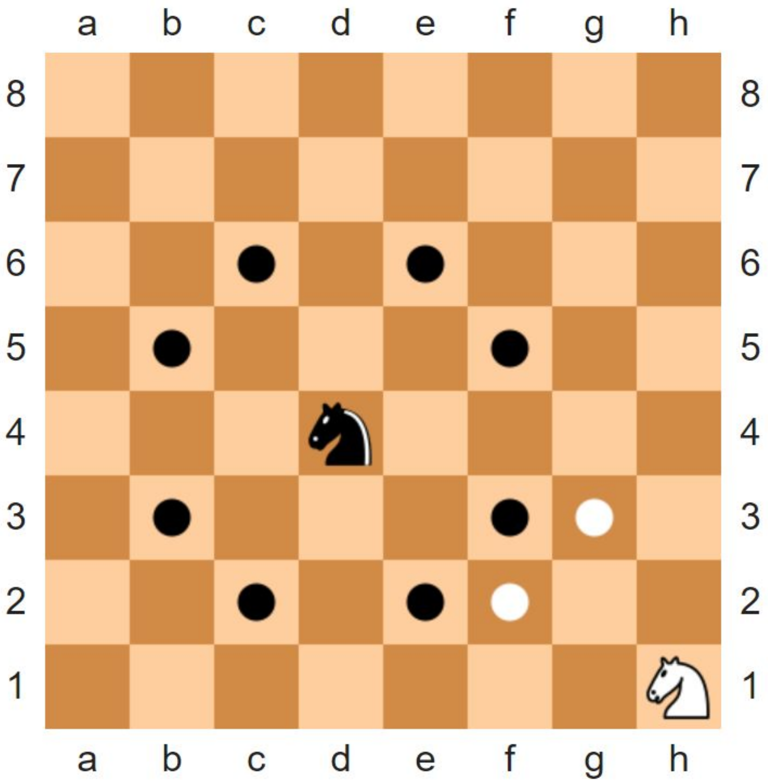
Chessboard

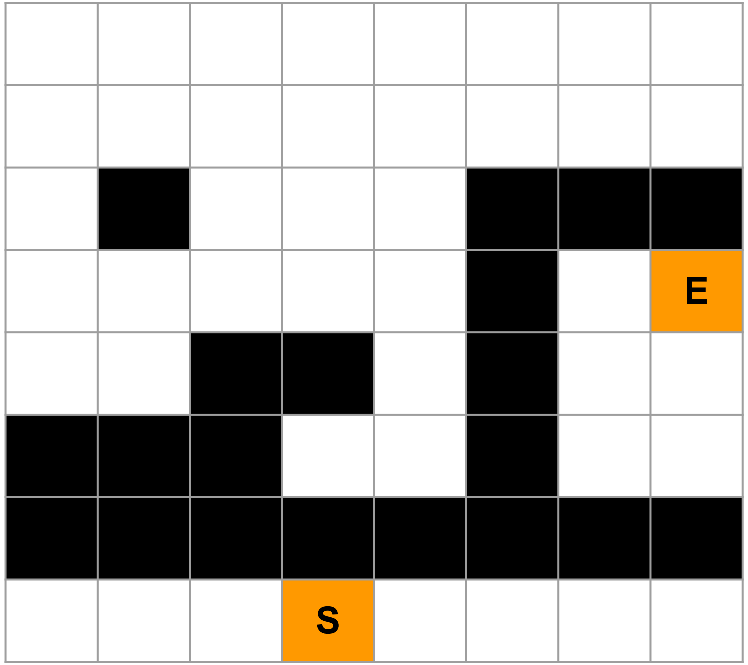
I have a chessboard with *some blocked* cells.

You have a knight at **S**, can you get to **E**?



Real Life Applications





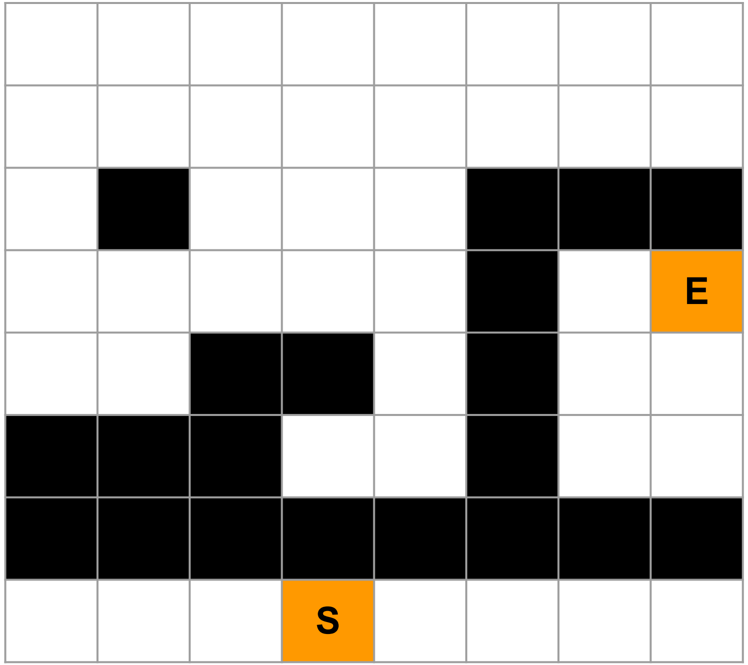
Real Life Applications

Chessboard

Vertex:

Cells of the chessboard.

Denoted by (row, col)

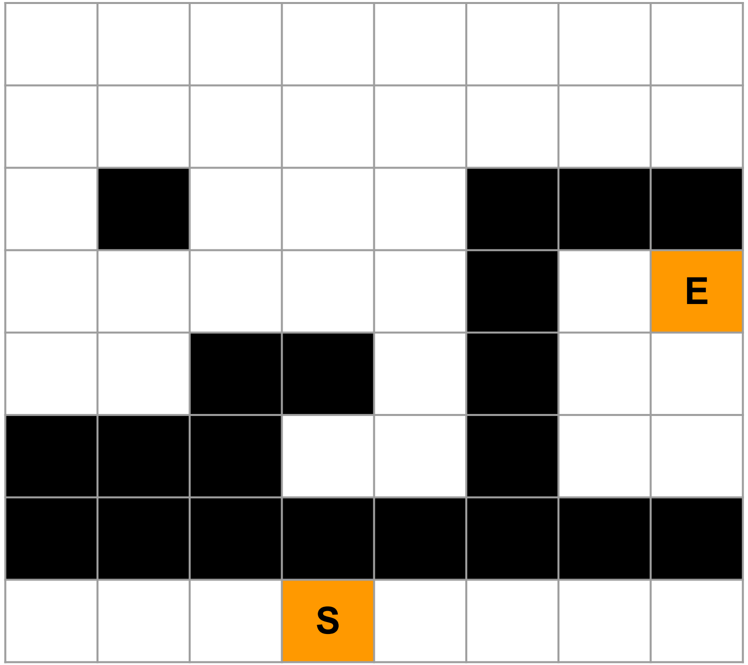


Real Life Applications

Chessboard

Edge:

Draw an edge representing the 8 moves of the knight.

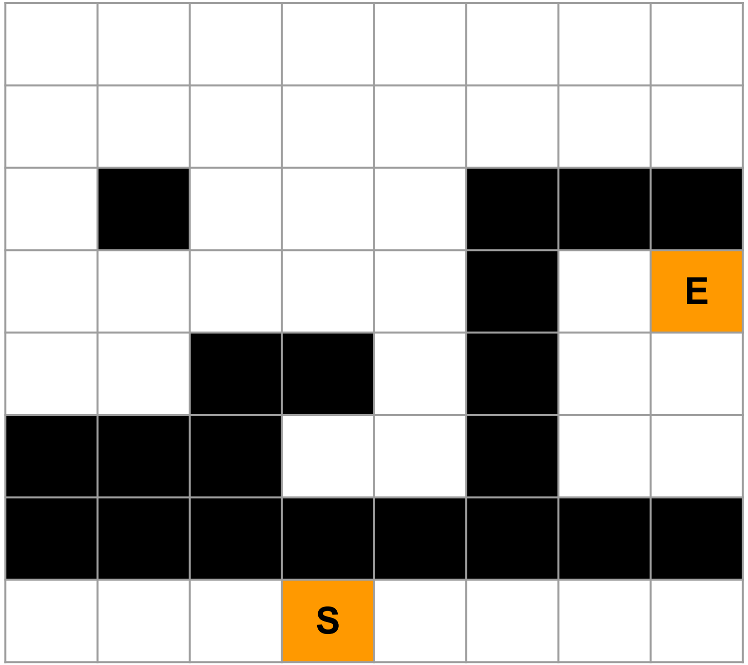


Real Life Applications

Chessboard

Blocked Cells:

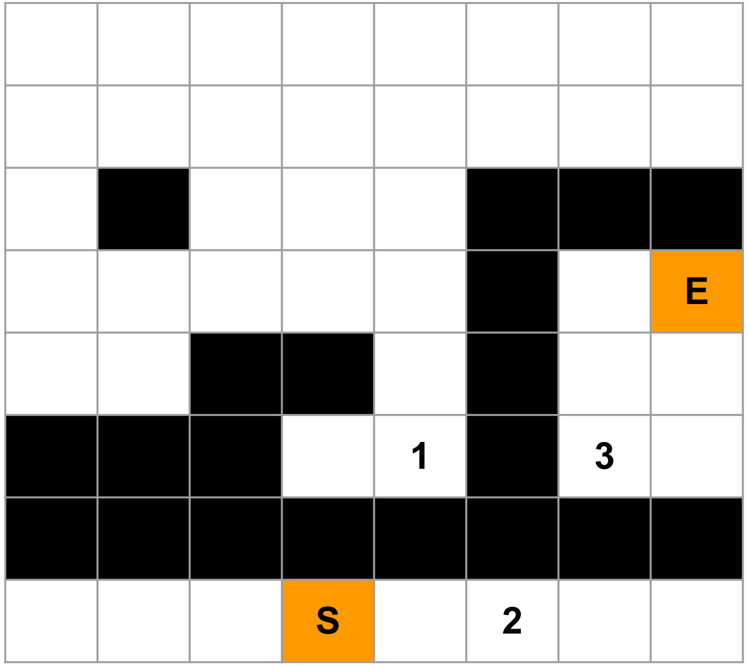
Don’t draw the edge if it touches any blocked cells.



Real Life Applications

Chessboard

Anybody found the solution? :P



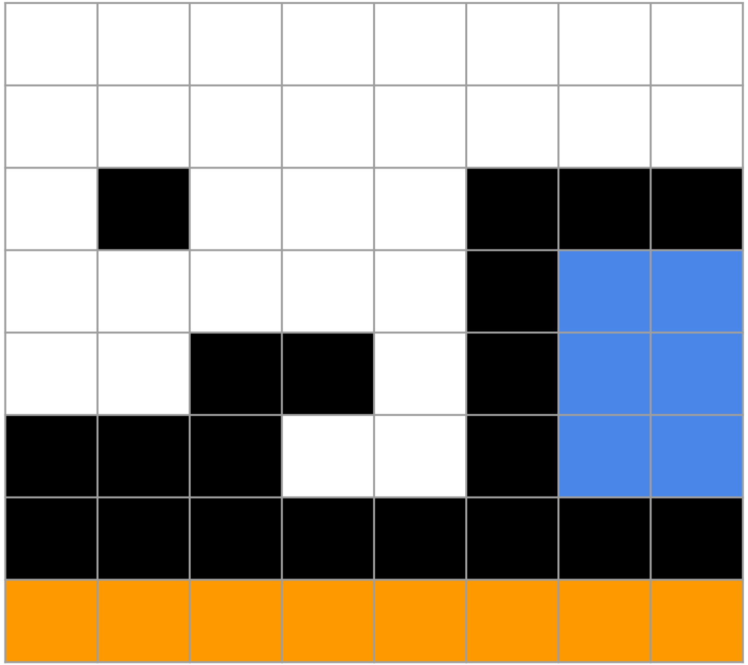
Real Life Applications

Flood Fill

Paint's "fill bucket" tool

Will 'flood' all adjacent cells with the same colour.

(4 directions)



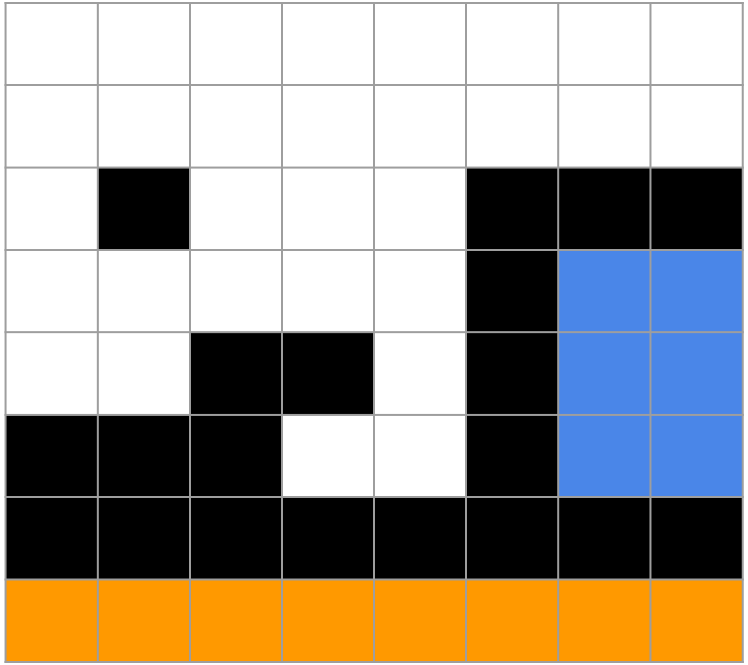
Real Life Applications

Flood Fill

How many different colours can you have?

(excluding black)

3



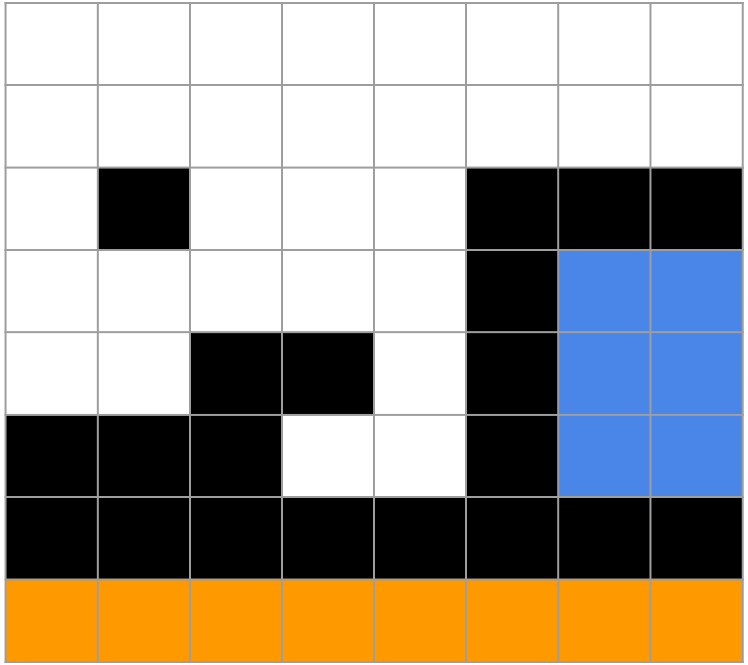
Real Life Applications

Flood Fill

How to calculate?

Find *connected component* (cc).

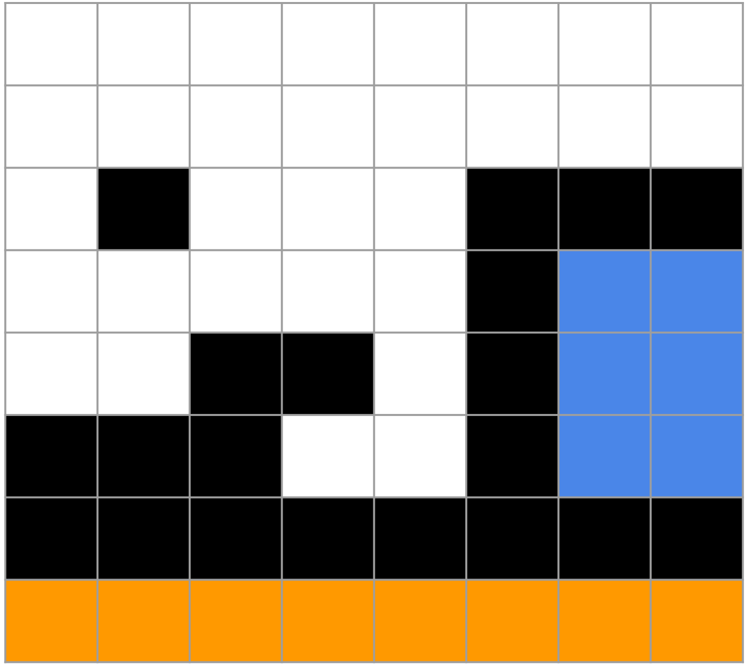
Each *cc* must share the same colour.



Real Life Applications

Flood Fill

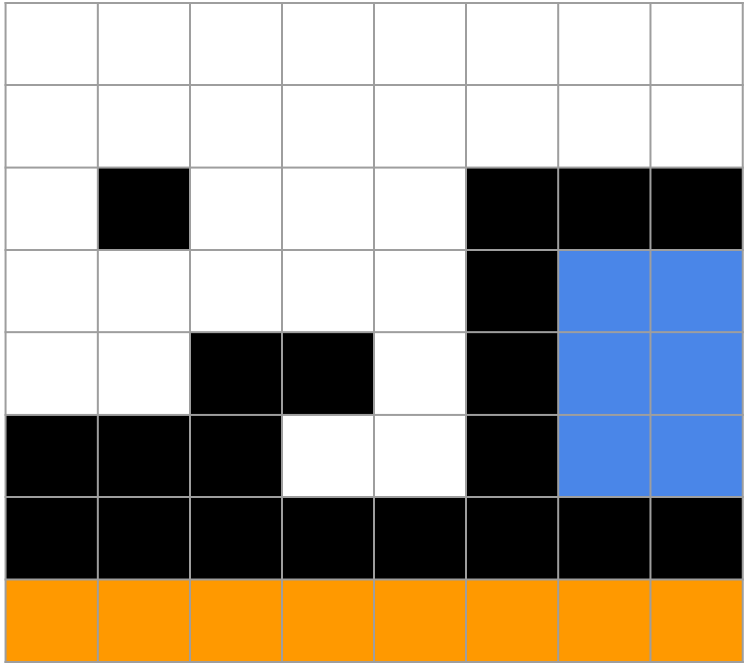
If we transform into a graph, each *cc* is essentially a *disjoint* graph.



Real Life Applications

Flood Fill

Do we have to *'store'* the vertices and edges in a AL/AM/EL?

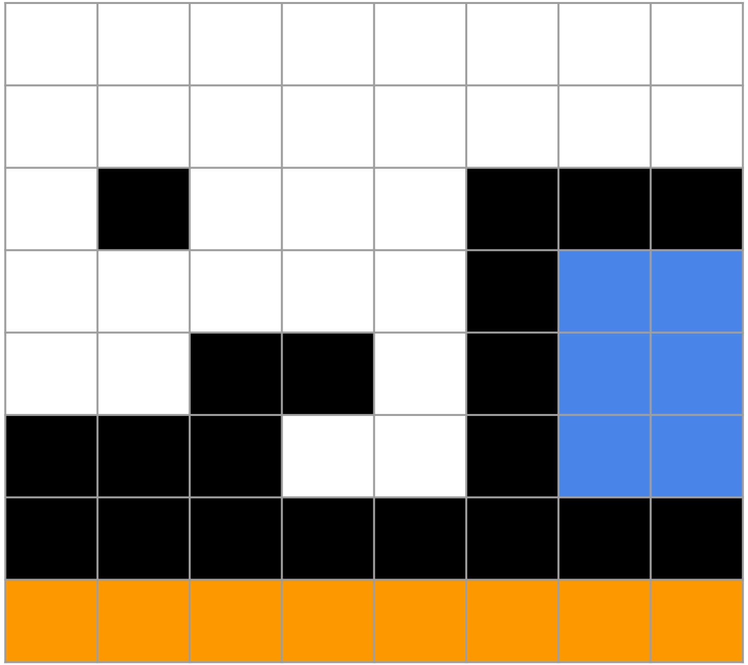


Real Life Applications

Flood Fill

Do we have to *'store'* the vertices and edges in a AL/AM/EL?

We *can*... but do we need to?

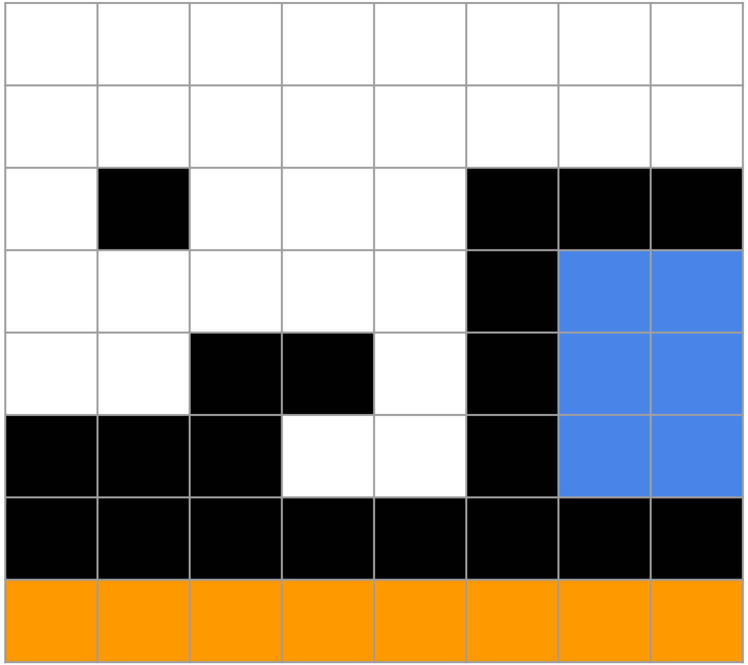


Real Life Applications

Flood Fill

Do we have to *'store'* the vertices and edges in a AL/AM/EL?

No.



Graph Traversal

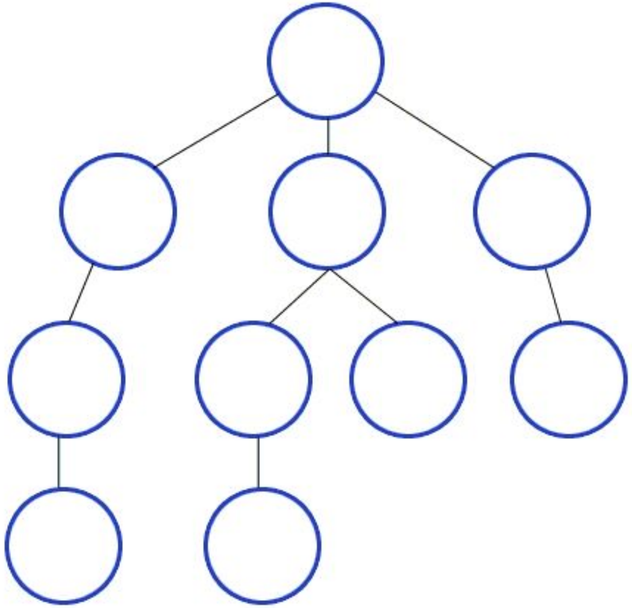
Dinner **F**irst; **S**leep

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

Depth First Search (DFS)

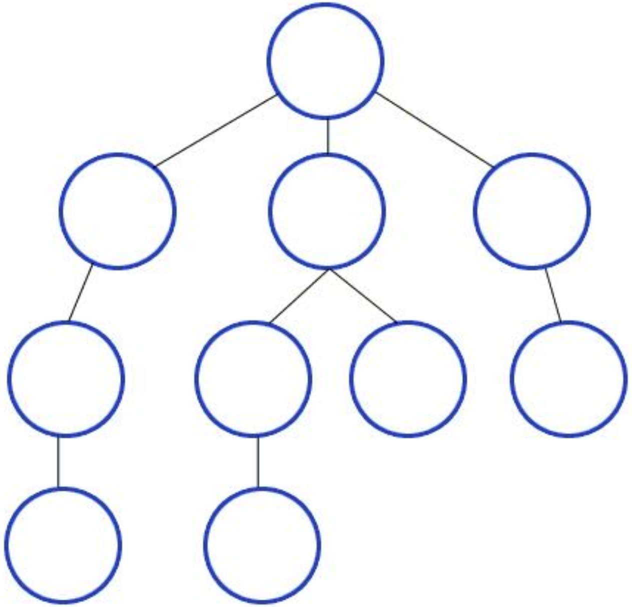
Likes to go *deeper*.

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



Depth First Search

```
void dfs(int vertex_id) {
     cout << vertex_id << endl;
     for (auto &it: adjList[vertex_id]){
          dfs(it); // infinite recursion!
     }
}
```



Depth First Search

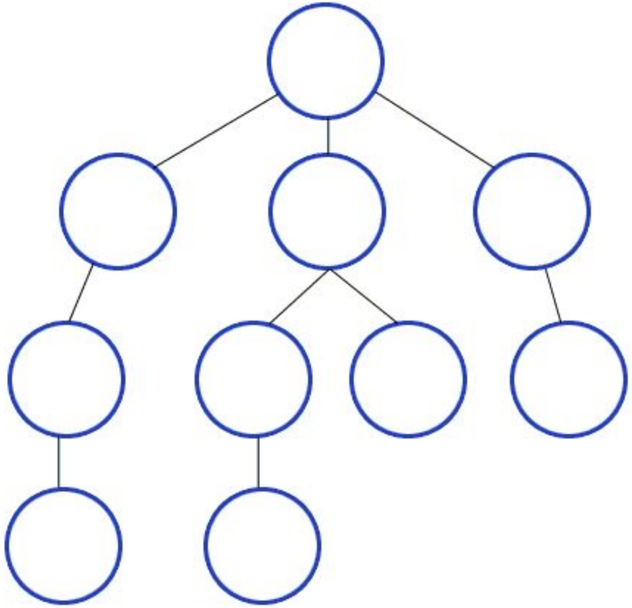
```
void dfs(int vertex_id) {
    if (visited[vertex_id]) return;
    visited[vertex_id] = true;
    cout << vertex_id << endl;
    for (int i = 0; i < adjList[vertex_id].size(); i++) {
        //recurse to neighbours
        dfs(adjList[vertex_id][i]);
    }
}
```

Depth First Search

Optional Challenge

Implement DFS
without recursion.

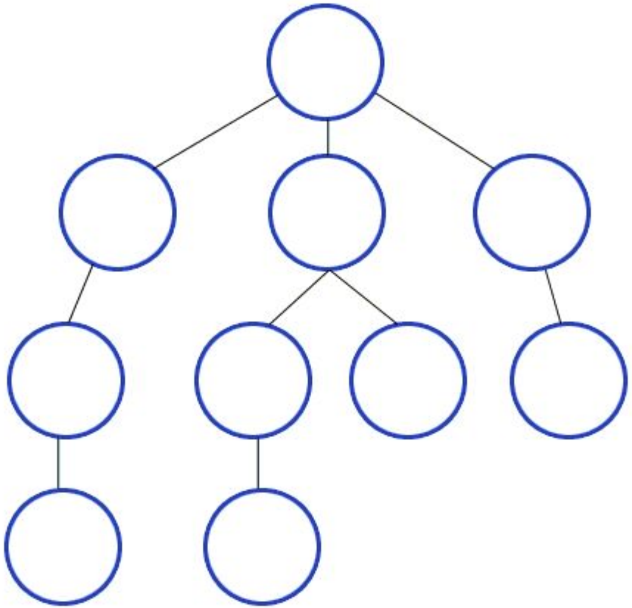
[It is ok to not try this]



Breadth First Search (BFS)

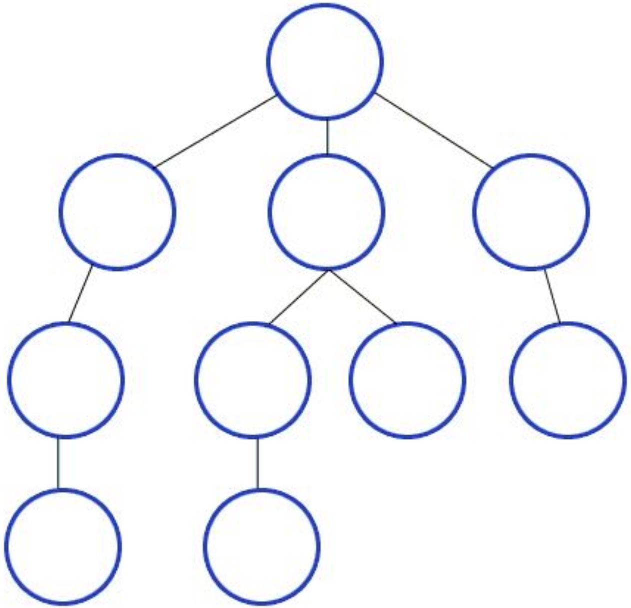
Likes to go *sideways*.

It will only go deeper when there is nothing of the same depth.

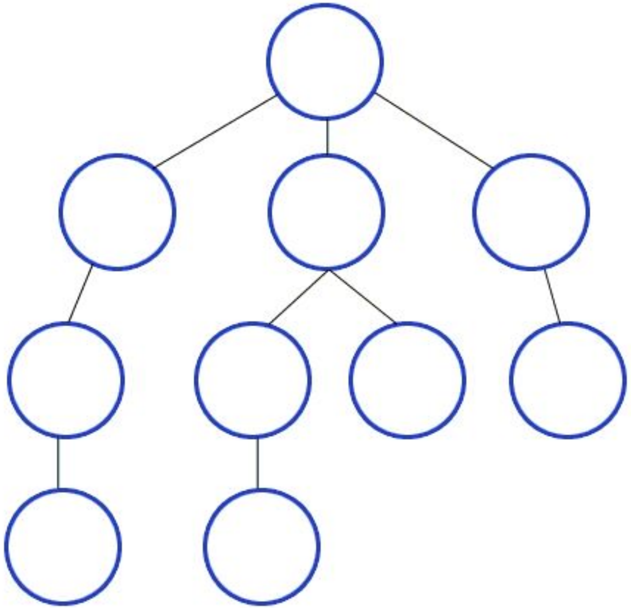


Comparison

BFS

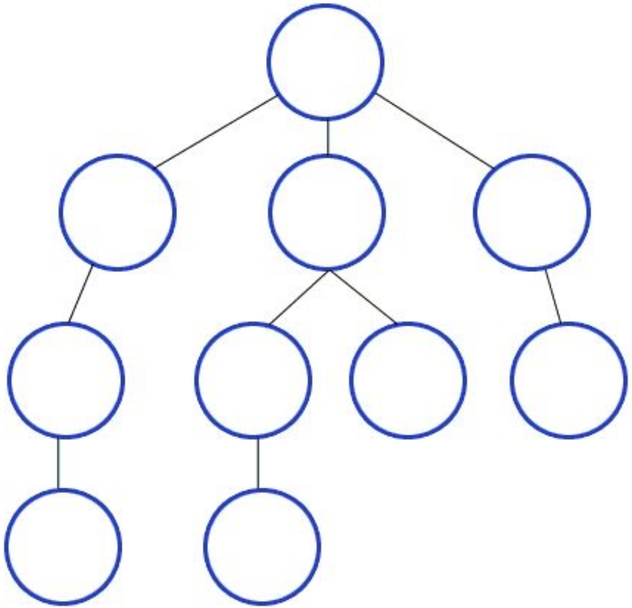


DFS



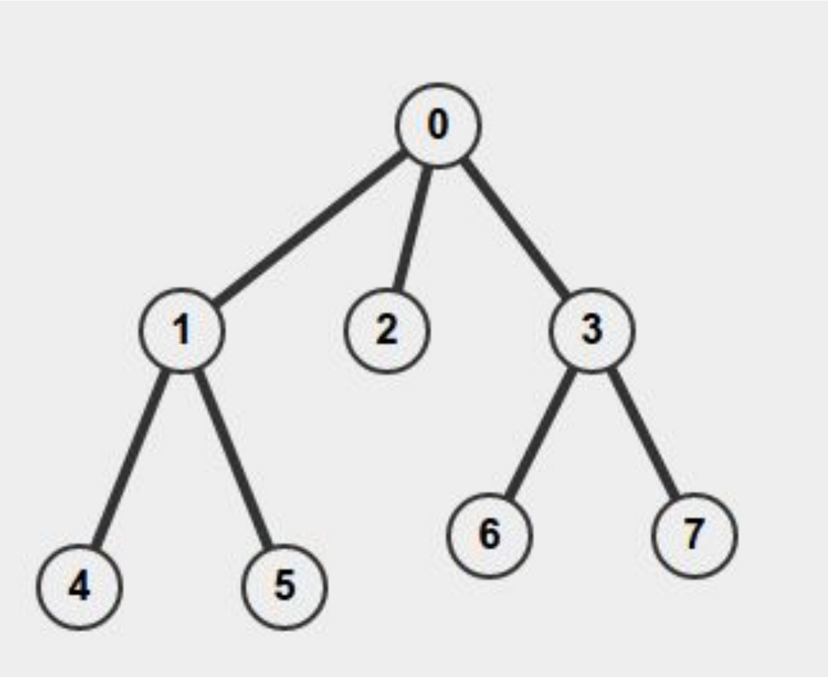
Breadth First Search

```
queue<int> q;
visited[s] = 1;
q.push(s);
while (!q.empty()) {
     int v = q.front(); q.pop();
     // visit v
     for (auto &it: adjList[v]) {
          if (visited[it]) continue;
          q.push(it);
          visited[it] = 1;
     }
}
```



Breadth First Search: Level Order

```
queue<int> q;
visited[s] = 1;
q.push(s);
While (!q.empty()) {
     int v = q.front(); q.pop();
     cout << v << endl;
     for (auto &it: adjList[v]) {
          if (visited[it]) continue;
          q.push(it);
          visited[it] = 1;
     }
}
```



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

Practical Exam

Scope

- Data Structures
    - Linear Data Structure
    - Non-linear Data Structure
- Sorting/Searching

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(logN)**	insert	**O(logN)**	[ ] operator	**O(logN)**
Delete	pop_back	**O(1)**	pop	**O(1)**	pop_front pop_back	**O(1)**	pop_front pop_back	**O(1)**	pop	**O(logN)**	erase	**O(logN)**	erase	**O(logN)**
Random Access	[ ] operator	**O(1)**	NIL		[ ] operator	**O(1)**	Loop Through	**O(N)**	NIL		find	**O(logN)**	[ ] operator	**O(logN)**
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(logN)**	NIL		lower_bound upper_bound	**O(logN)**	NIL		NIL		lower_bound upper_bound	**O(logN)**	lower_bound upper_bound	**O(logN)**
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

```
#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

```
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 <u>line</u>.
  - 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

- **Plan** 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 *unsigned long long* from PS0?
- More **practice** → code faster
  - Range based for-loops
  - STL Data Structures
  - If you run out of problems… …

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 structures 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...)