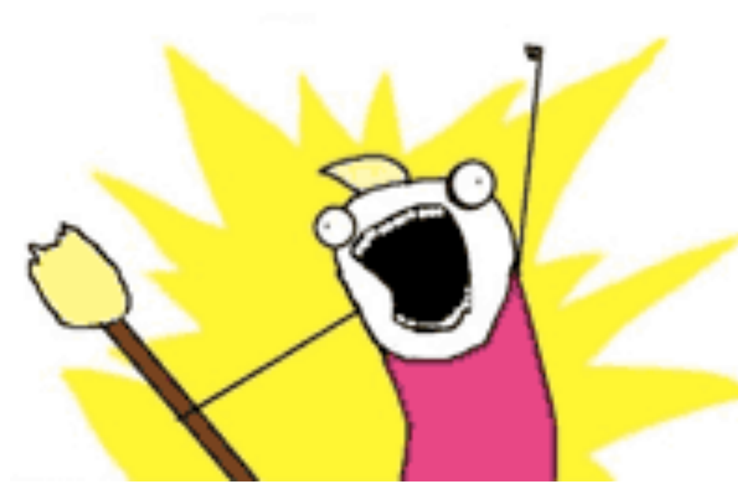
CSC263 Week 8

Announcements

Problem Set 4 is out!

Due Tuesday (Nov 17)



Other Announcements

➔Drop date Nov 8

➔Final exam schedule is posted

CSC263 exam Dec 11, 2-5pm

This week’s outline

➔Graphs

➔BFS

Graph

A really, really important ADT that is used to model **relationships** between objects.

Get that job at Google

Whenever someone gives you a problem, *think graphs*. They are the most fundamental and flexible way of representing any kind of a relationship, so it's about a 50-50 shot that any interesting design problem has a graph involved in it. Make absolutely sure you can't think of a way to solve it using graphs before moving on to other solution types. This tip is important!

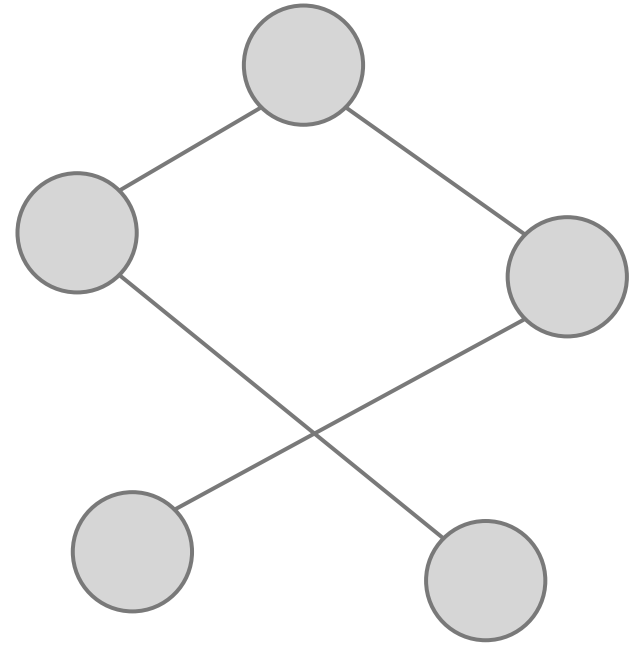
Reference: http://steve-yegge.blogspot.ca/2008/03/get-that-job-at-google.html

Things that can be modelled using graphs

- ➔Web
- ➔Facebook
- ➔Task scheduling
- ➔Maps & GPS
- ➔Compiler (garbage collection)
- ➔OCR (computer vision)
- ➔Database
- ➔Rubik's cube
- ➔…. (many many other things)

Definition

$$G = (V, E)$$

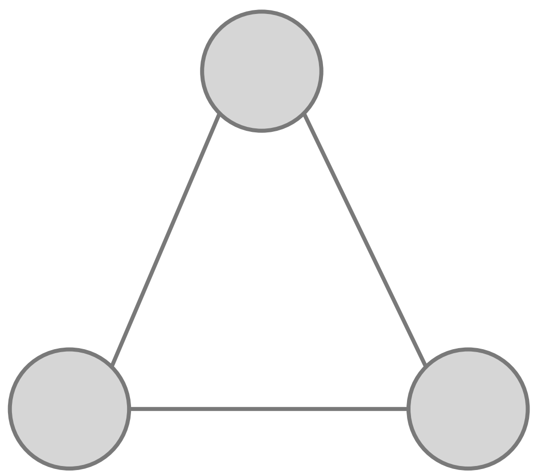


Set of **vertices**
e.g., {a, b, c}

Set of **edges**
e.g., { (a, b), (c, a) }

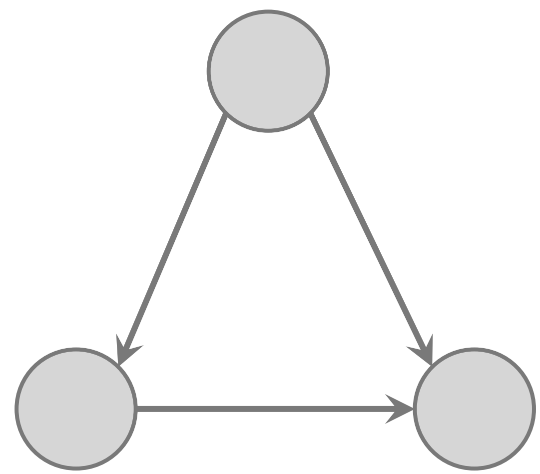
Flavours of graphs

each edge is an
unordered pair
(u, v) = (v, u)

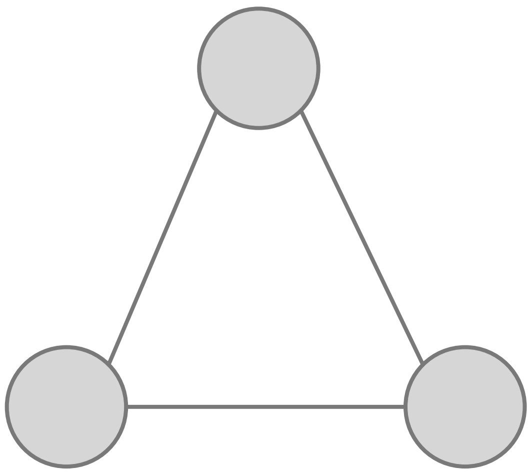


Undirected

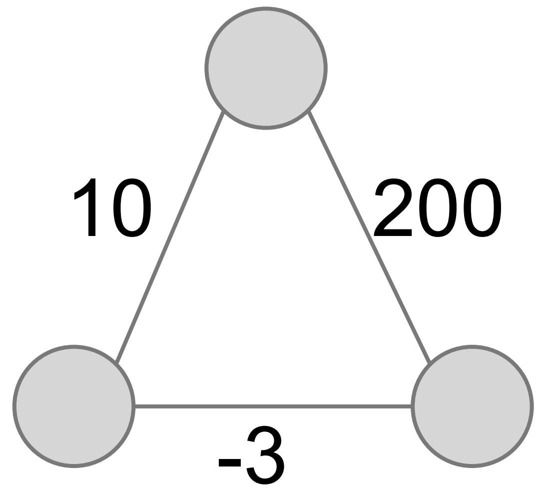
each edge is an
ordered pair
(u, v) ≠ (v, u)



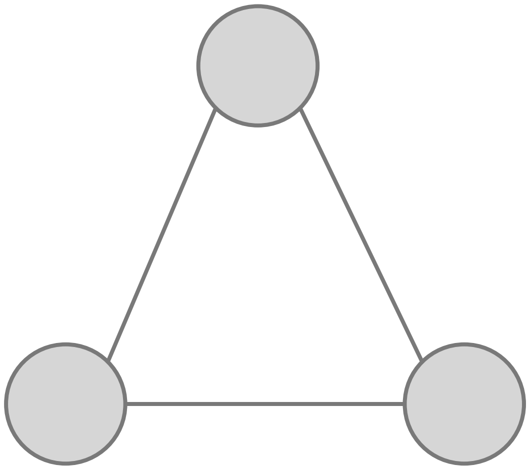
Directed



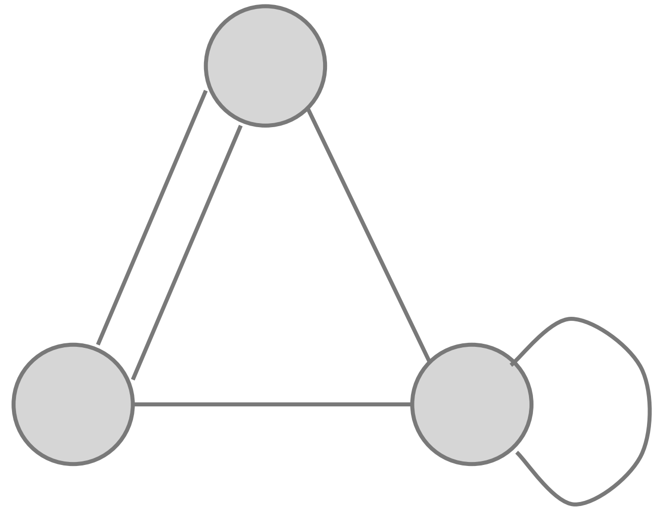
Unweighted



Weighted

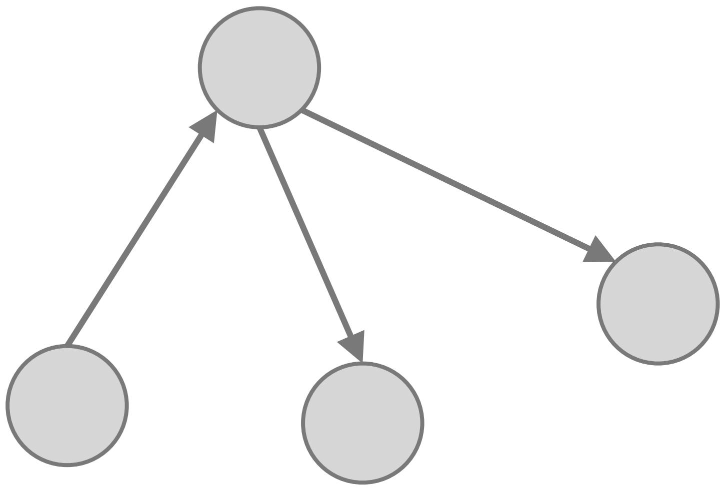


Simple

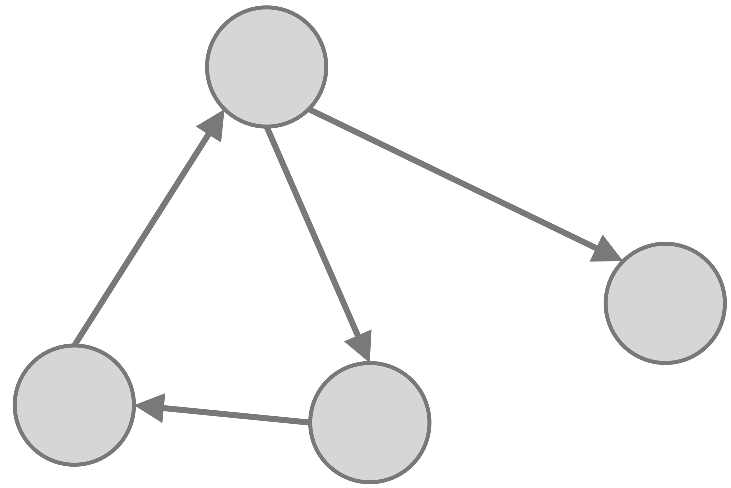


Non-simple

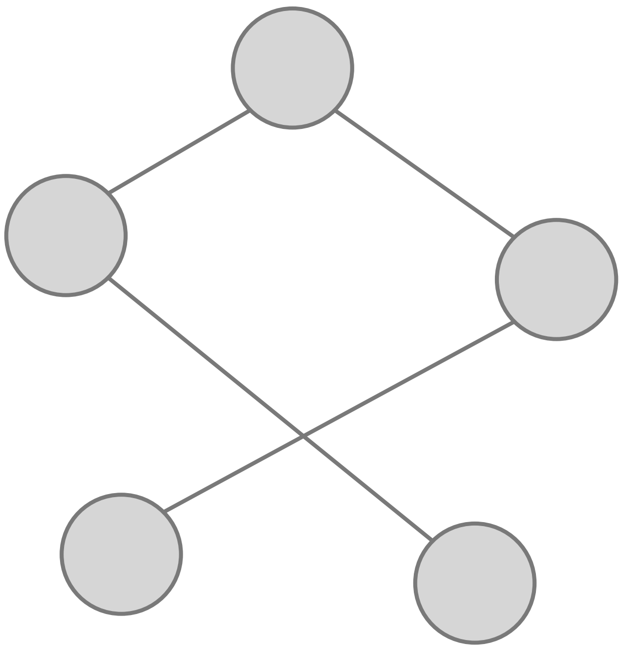
No multiple edge, no self-loop



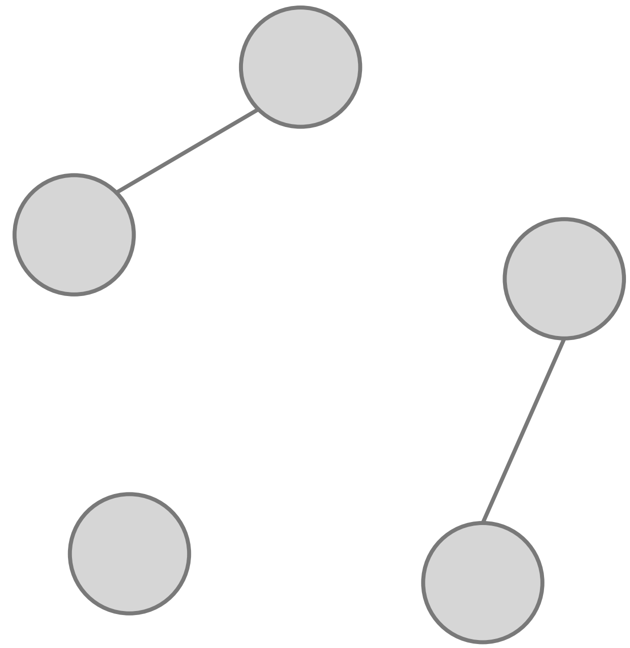
Acyclic



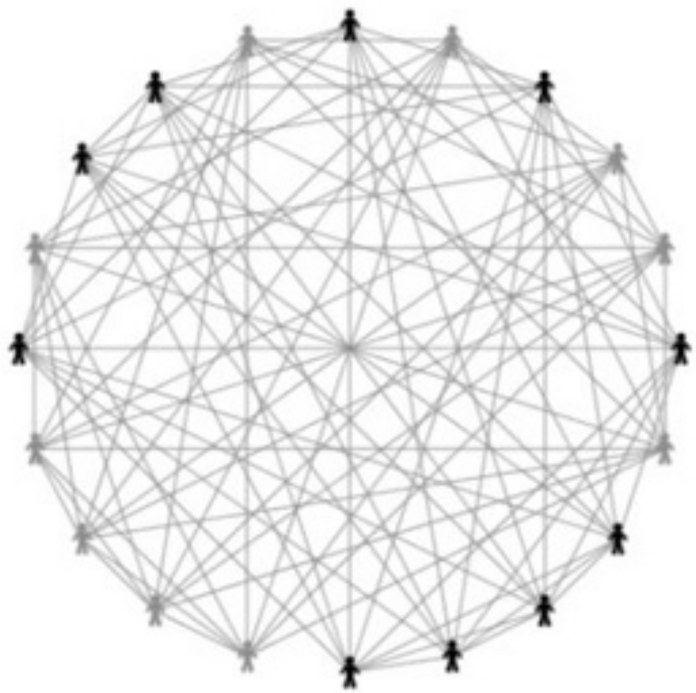
Cyclic



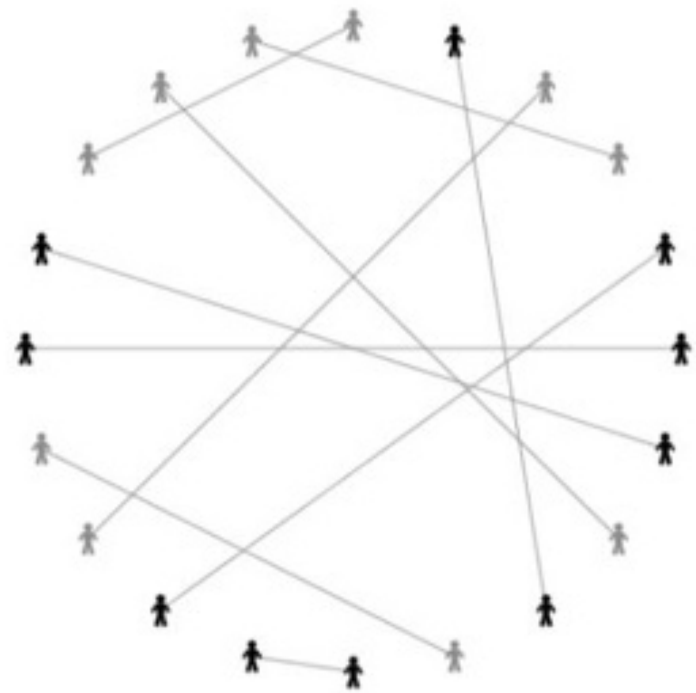
Connected



Disconnected

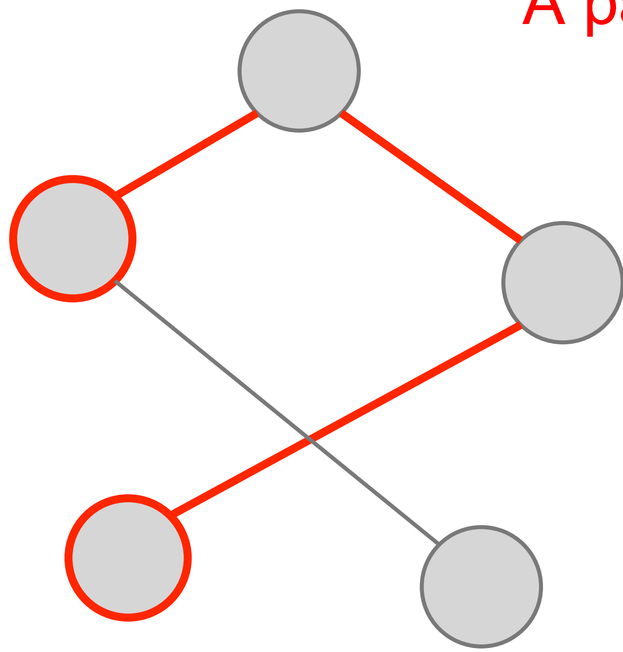


Dense



Sparse

Path



Length of path = number of edges

Read Appendix B.4 for more background on graphs.

Operations on a graph

- ➔**Add** a vertex; **remove** a vertex
- ➔**Add** an edge; **remove** an edge
- ➔Get **neighbours** (undirected graph)
  - ◆Neighbourhood(u): all $v \in V$ such that $(u, v) \in E$
- ➔Get **in**-neighbours / **out**-neighbours (directed graph)
- ➔**Traversal**: visit every vertex in the graph

Many other operations:

- ➔Traversal:
  - BFS (breadth first search)
  - DFS (depth first search)
- ➔Given s,t find a (minimum length) path from s to t
- ➔Given a connected graph G, output a spanning tree of G
- ➔Is G connected?

Data structures for the graph ADT

➔ Adjacency **matrix**

➔ Adjacency **list**

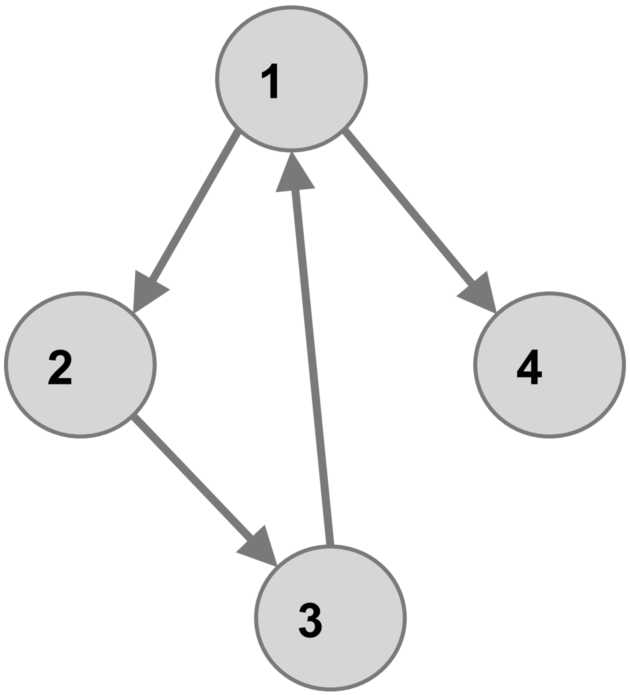
Adjacency matrix

A **|V|x|V|** matrix **A**

$$\text{Let } V = \{v_1, v_2, \ldots, v_n\}$$

$$A[i,j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency matrix



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

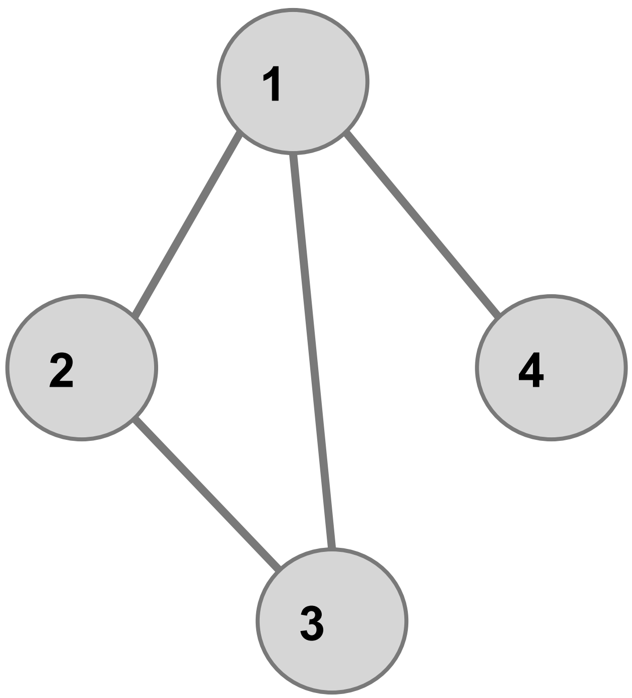
Adjacency matrix

How much space does it take?

$$|V|^2$$

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

Adjacency matrix (undirected graph)



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

The adjacency matrix of an **undirected** graph is _____**symmetric**________.

Adjacency matrix (undirected graph)

How much space does it take?

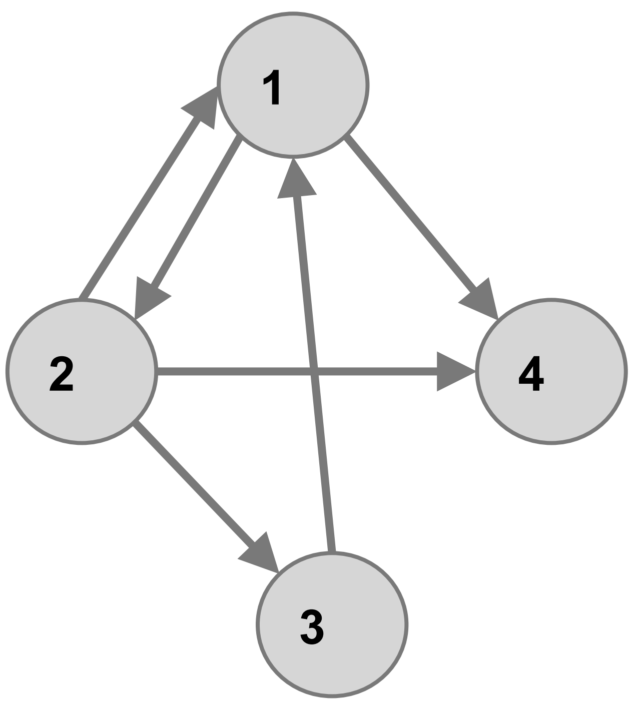
$$|V|^2$$

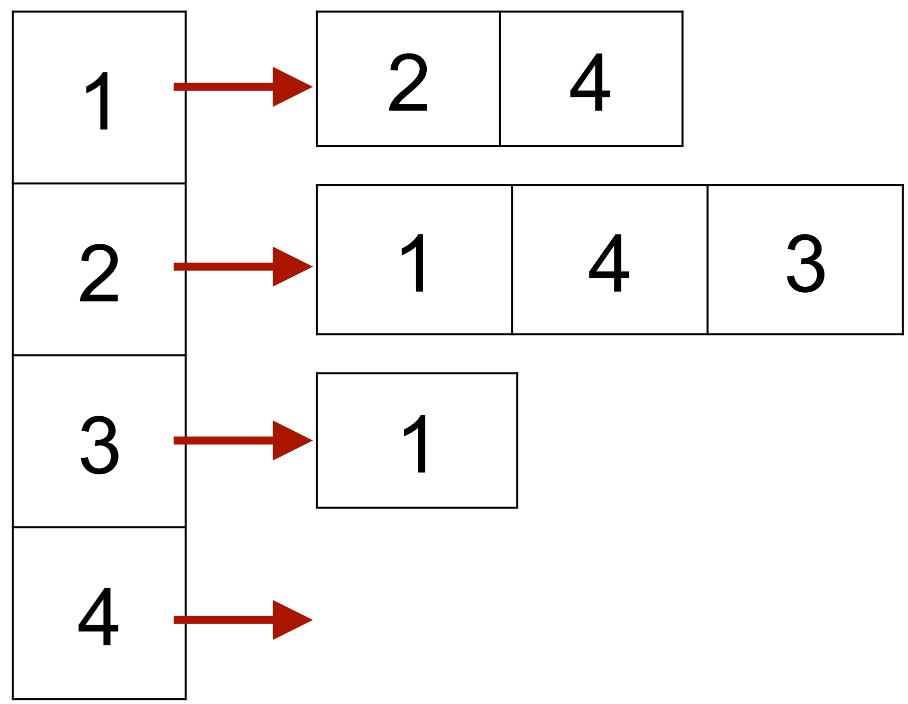
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	0
4	1	0	0	0

Adjacency list

Adjacency list (directed graph)

Each vertex v_i stores a list **A[i]** of v_j that satisfies $(v_i , v_j) \in E$

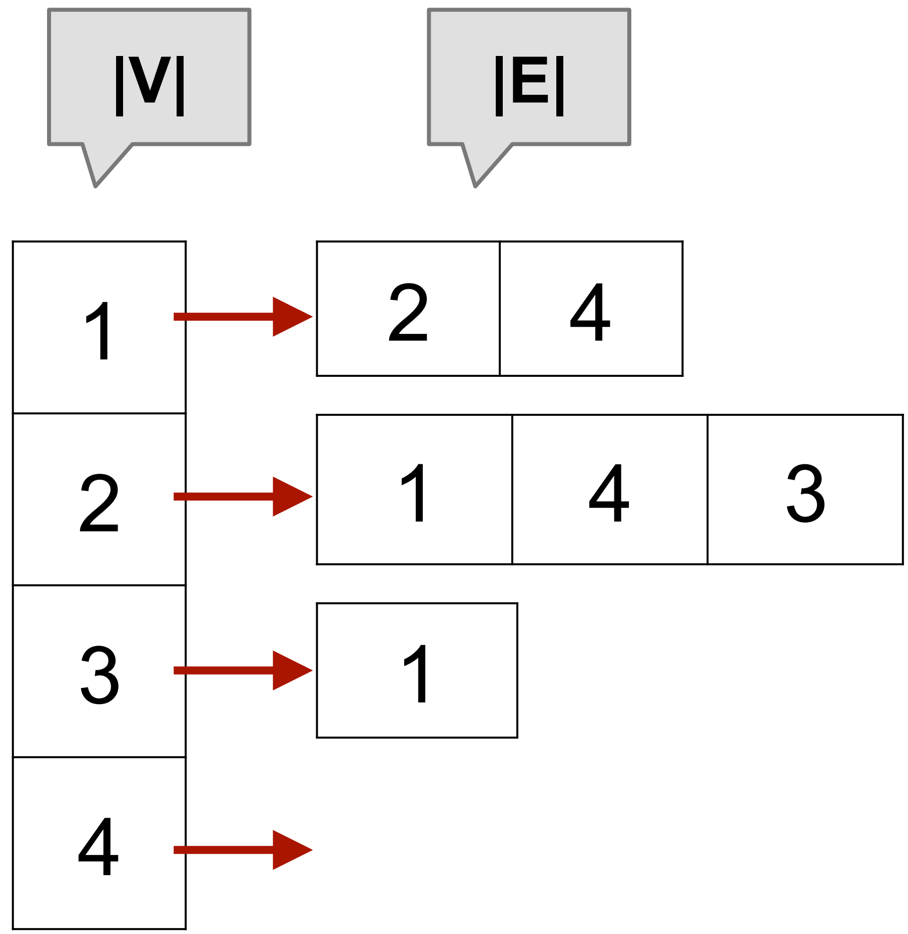




Adjacency list (directed graph)

How much space does it take?

|V|+|E|

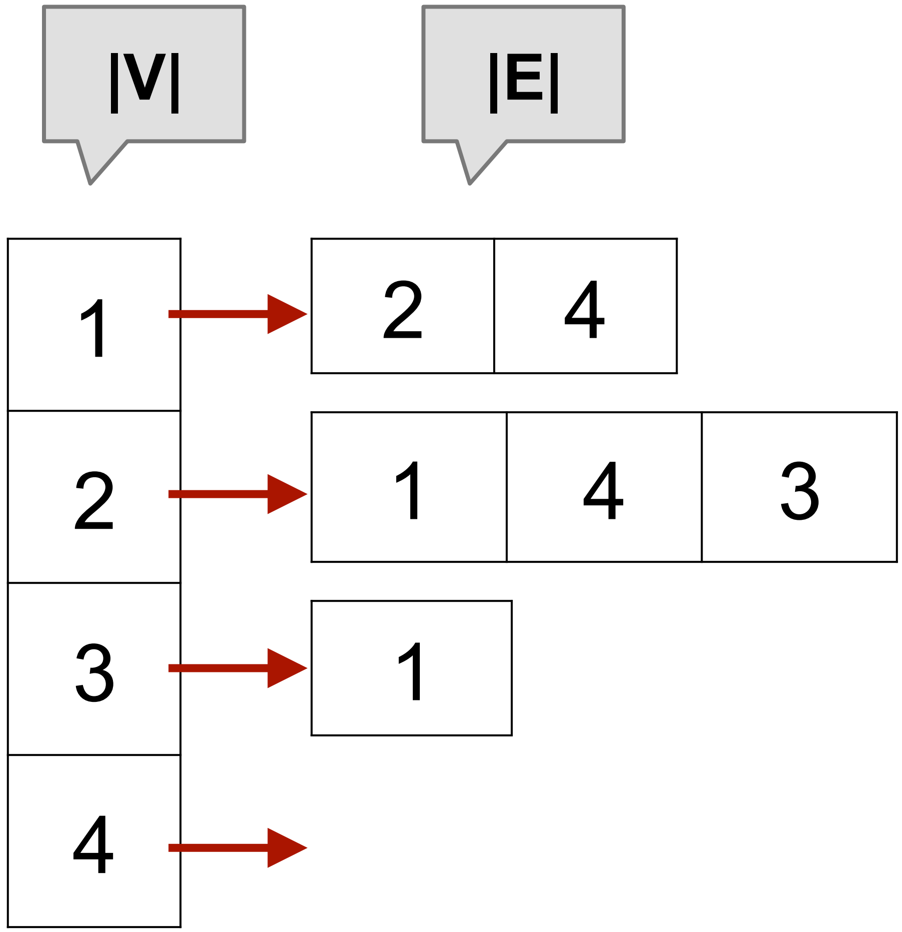


Adjacency list (directed graph)

How much space does it take?

|V|+|E|

This assumes we can store the name of a vertex in one cell of the linked list. In terms of bits, the size would be more like |V| + |E| (log |V|)

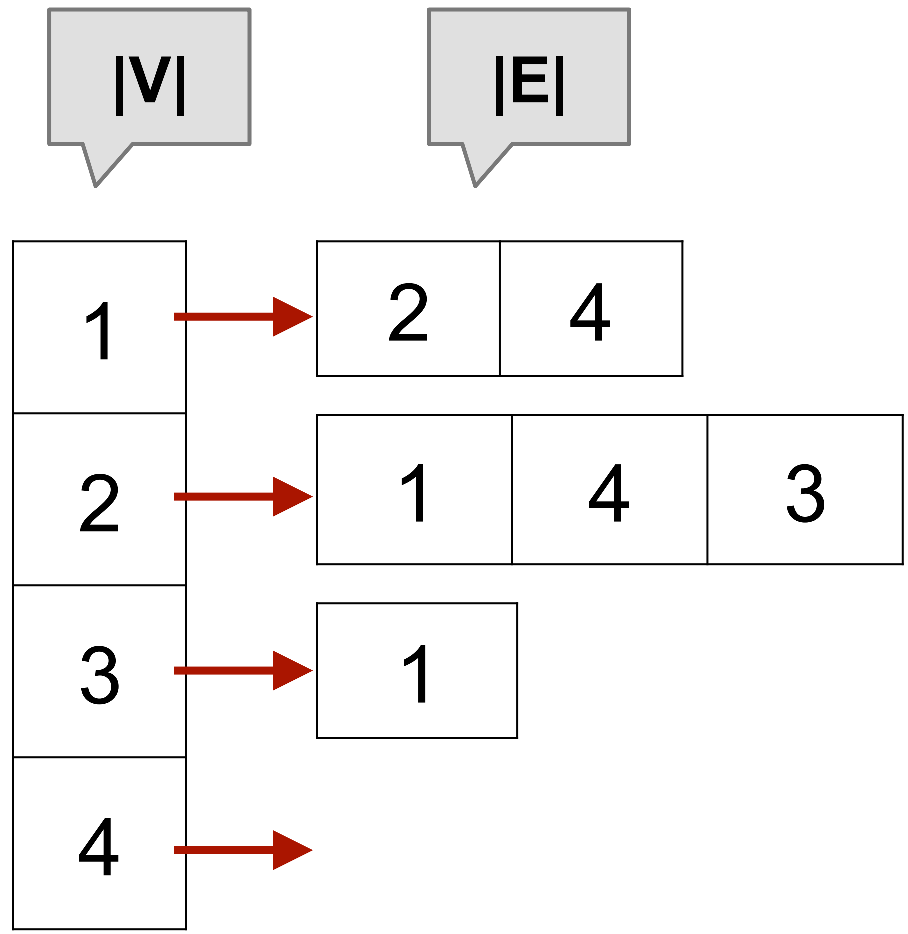


Adjacency list (directed graph)

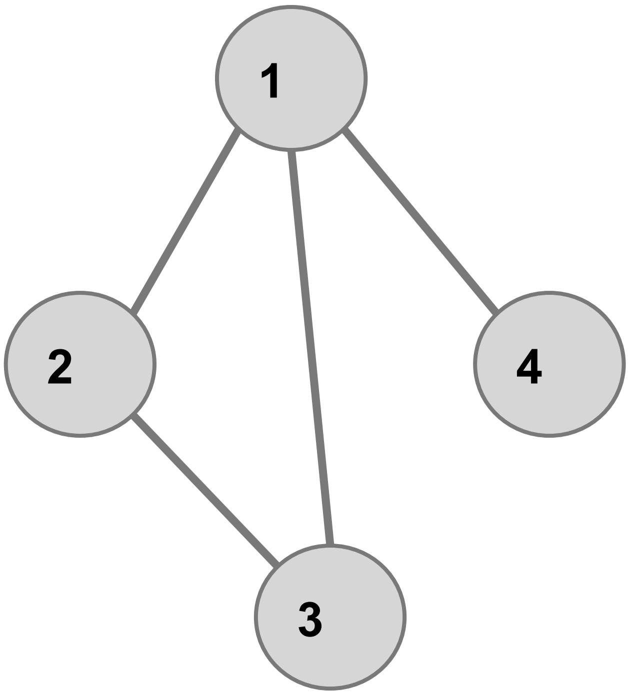
How much space does it take?

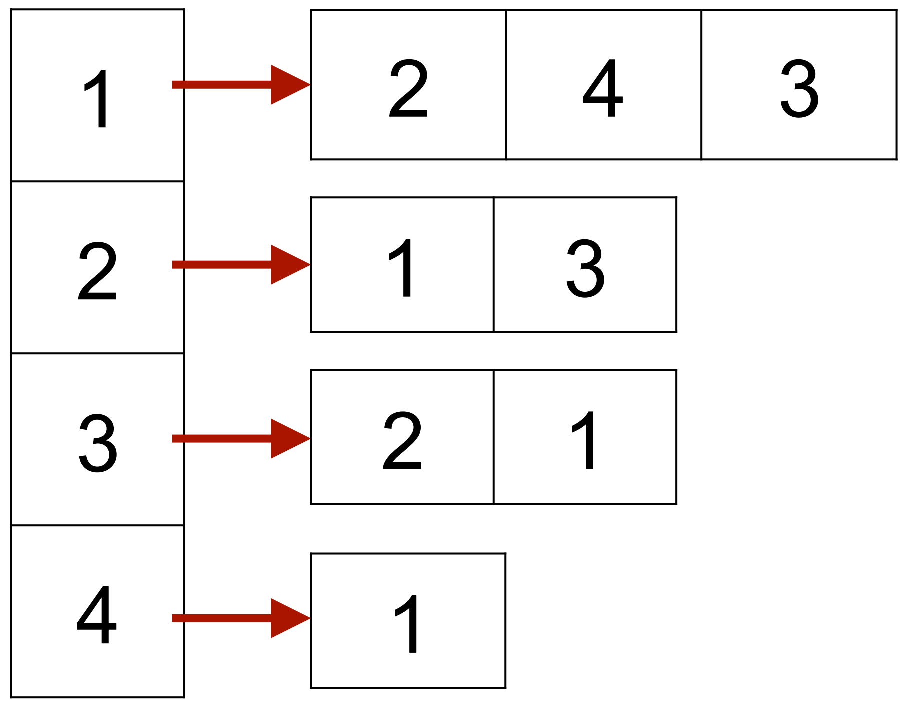
|V|+|E|

One often ignores these lower order factors of log n. (Recall for hashing, we assume h(x) in constant time.)



Adjacency list (undirected graph)

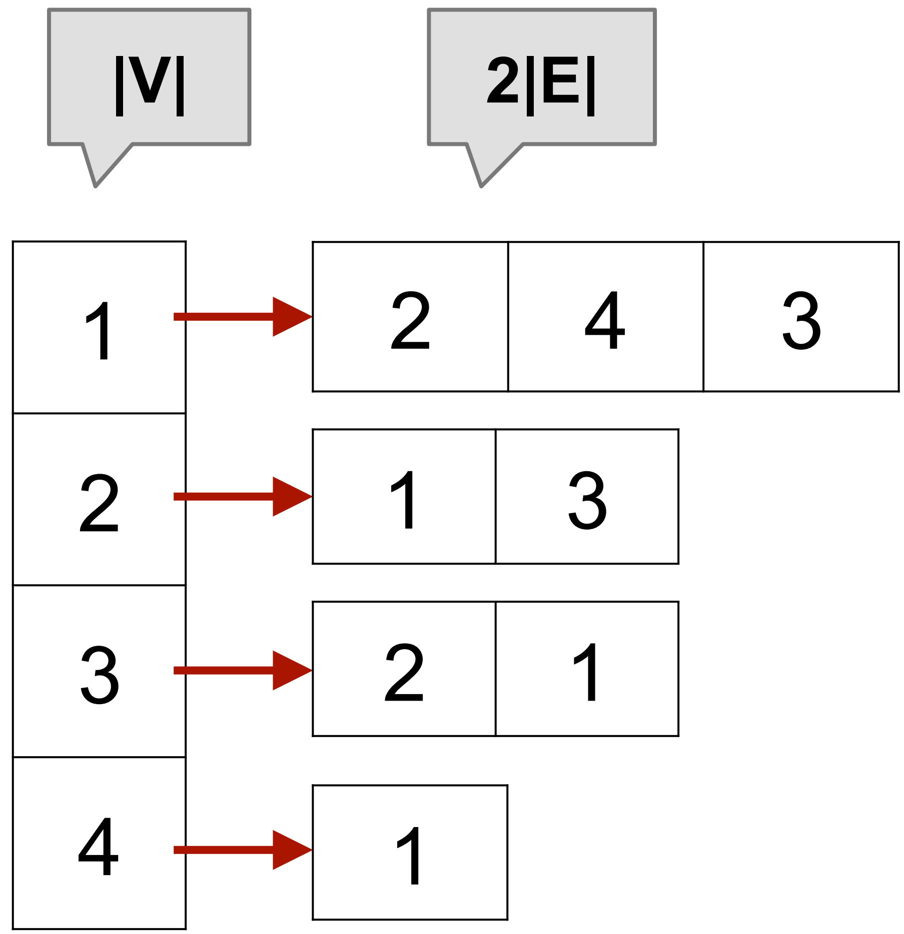




Adjacency list (undirected graph)

How much space does it take?

|V|+2|E|



Matrix *versus* List

In term of space complexity

→adjacency matrix is $\boldsymbol{\Theta(|V|^2)}$

→adjacency list is $\boldsymbol{\Theta(|V|+|E|)}$

Which one is more space-efficient?

Adjacency list, if $\boldsymbol{|E| \ll |V|^2}$, i.e., the graph is not very **dense**.

Matrix *versus* List

Anything that **Matrix** does better than **List**?

Check whether edge **(v_i , v_j)** is in **E**

➔ **Matrix**: just check if A[i, j] = 1, **O(1)**

➔ **List**: go through list A[i] see if j is in there, **O(length of list)**

Takeaway

Adjacency **matrix** or adjacency **list**?

Choose the more appropriate one depending on the problem.

Graph Traversals

BFS and DFS

They are twins!

Graph traversals

Visiting **every** vertex **once**, starting from a given vertex.

The visits can follow different **orders**, we will learn about the following two ways

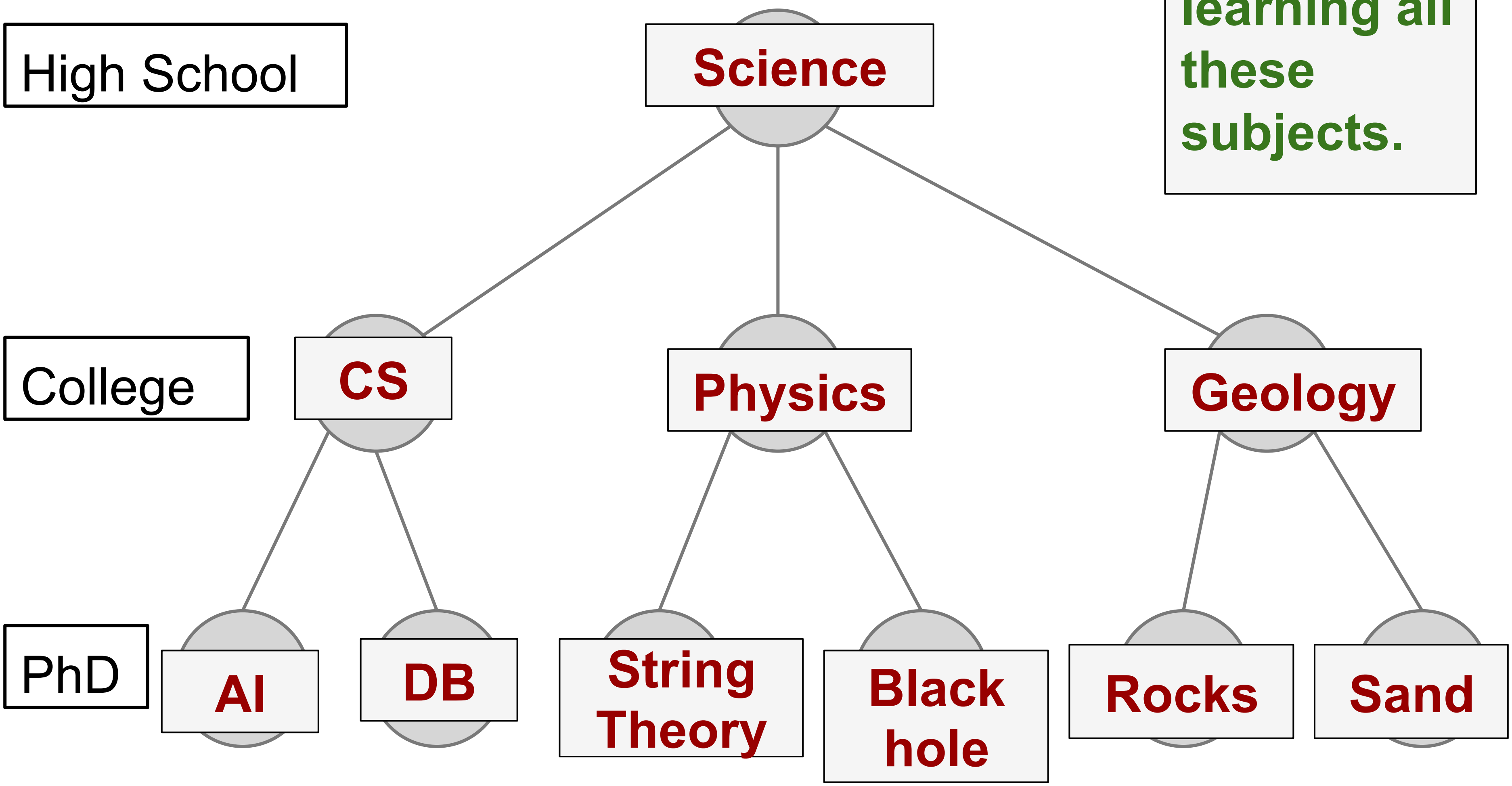
➔**Breadth** First Search (**BFS**)
➔**Depth** First Search (**DFS**)

Intuitions of BFS and DFS

Consider a special graph -- a **tree**

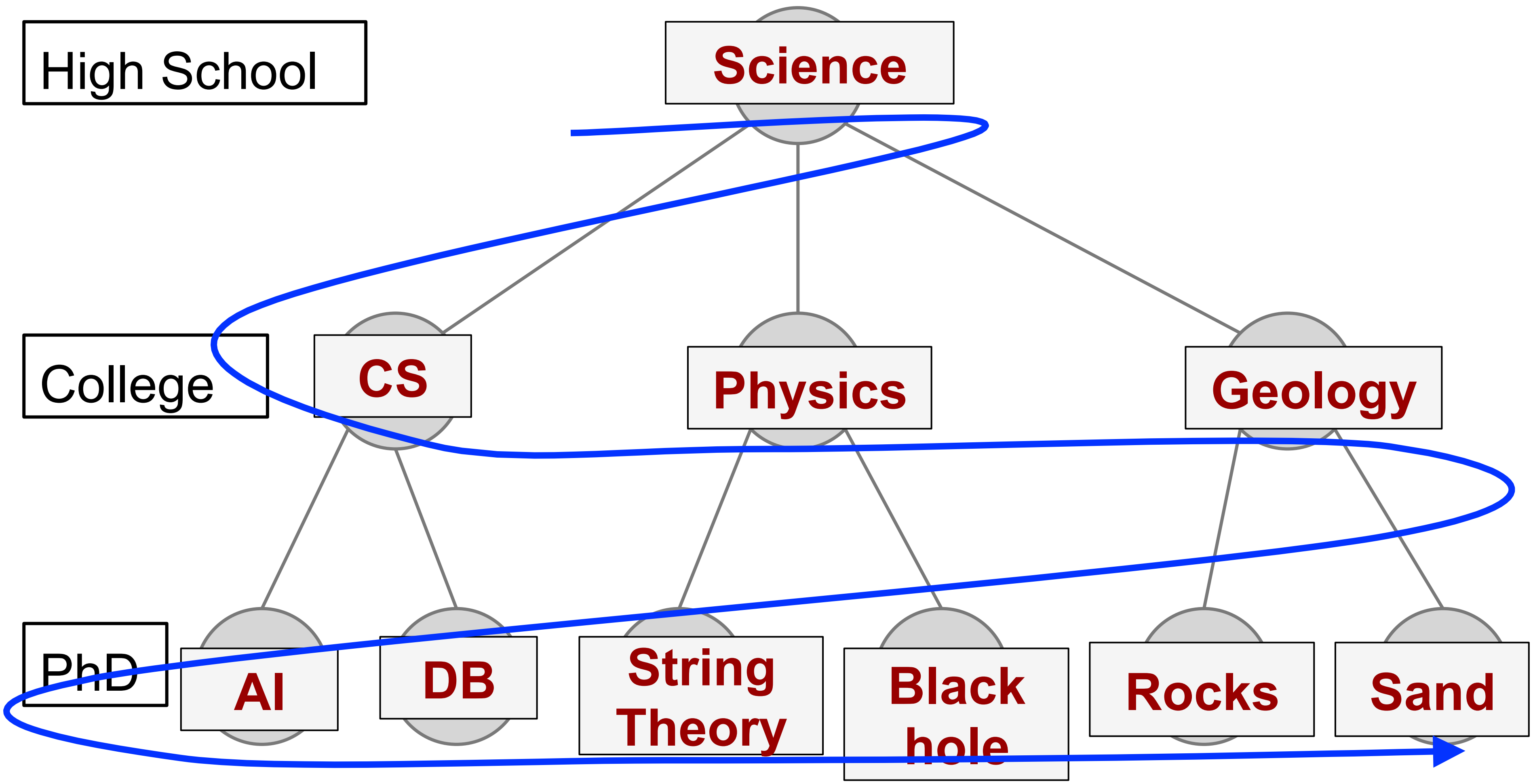
"The knowledge learning tree"

Traversing this graph means learning all these subjects.



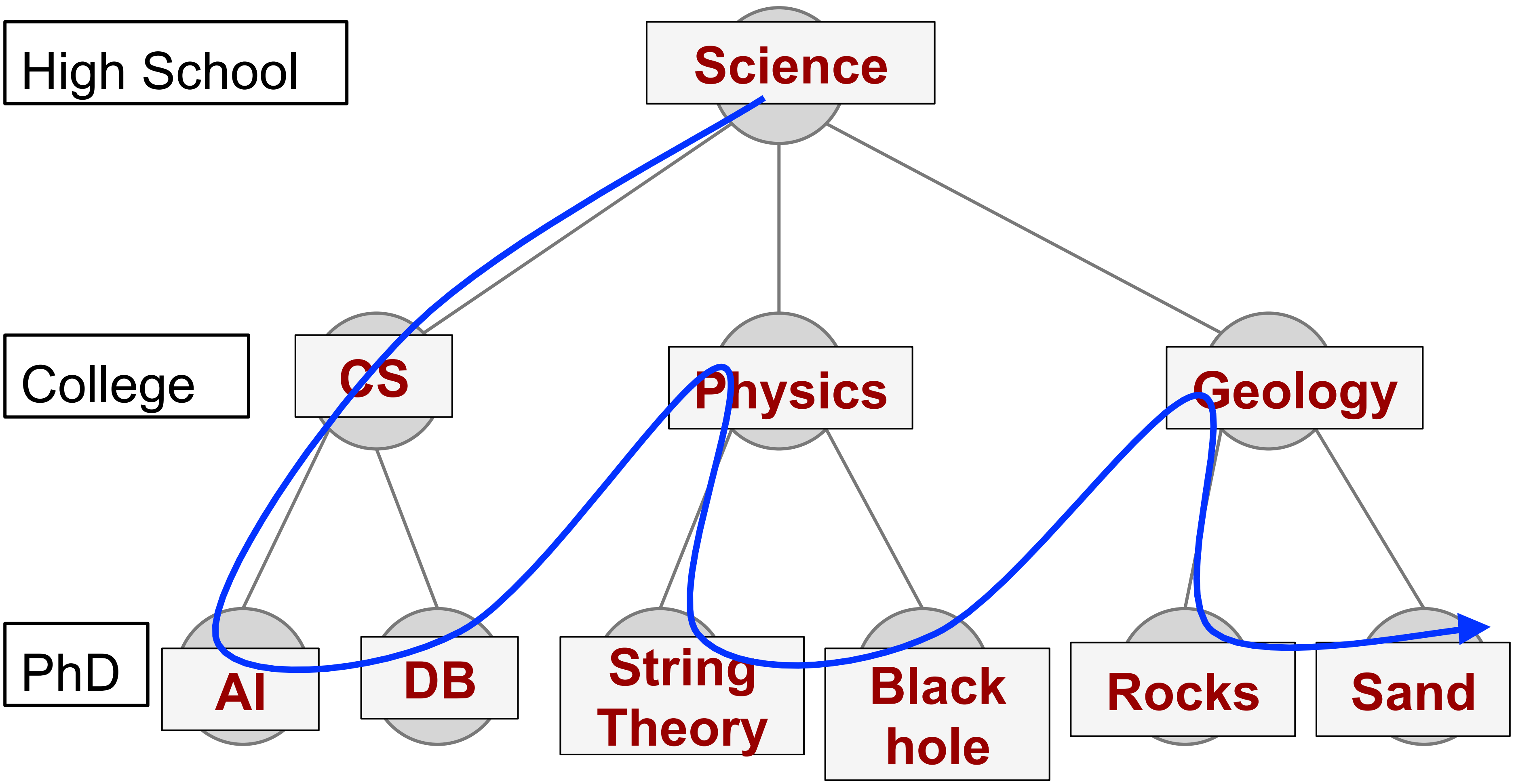
The Breadth-First ways of learning these subjects

➔ Level by level, finish high school, then all subjects at College level, then finish all subjects in PhD level.

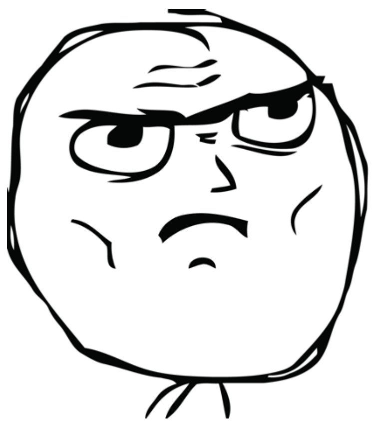


The Depth-First way of learning these subjects

➔ Go towards PhD whenever possible; only start learning physics after finishing everything in CS.



Now let's seriously start studying BFS



Special case: BFS in a tree

Review CSC148:

BFS in a tree (starting from root) is a

level-by-level ________________ traversal.

(NOT preorder!)

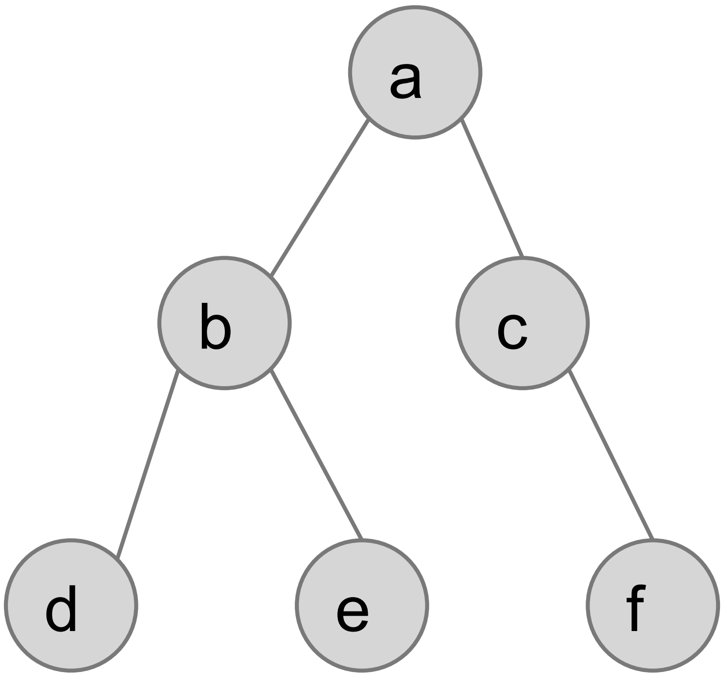
What ADT did we use for implementing the **level-by-level** traversal?

Queue!

Special case: BFS in a tree

Output:

a
b
c
d
e
f



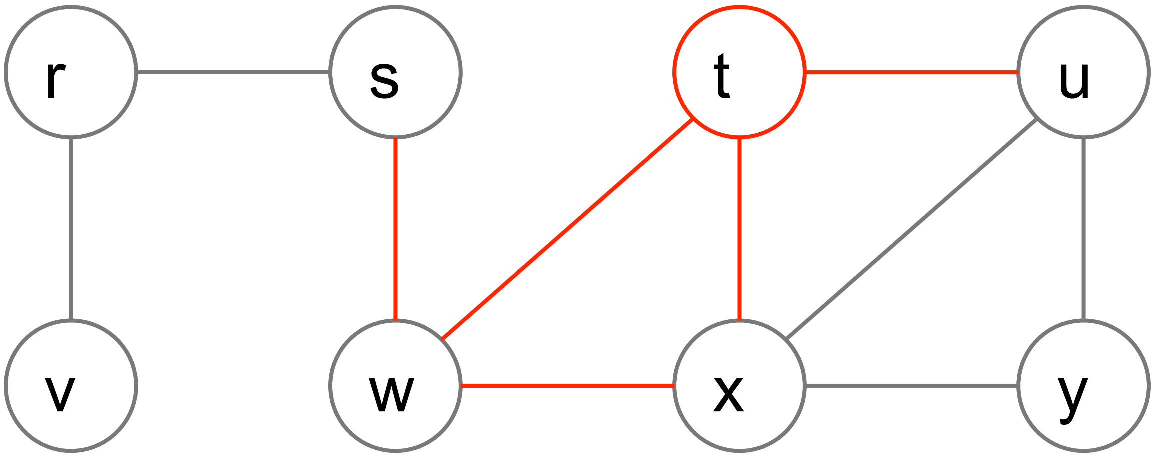

```
NOT_YET_BFS(root):
  Q ← Queue()
  Enqueue(Q, root)
  while Q not empty:
    x ← Dequeue(Q)
    print x
    for each child c of x:
      Enqueue(Q, c)
```

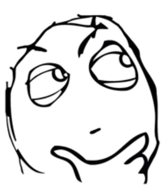
Queue:

a	b	c	d	e	f
DQ	DQ	DQ	DQ	DQ	DQ

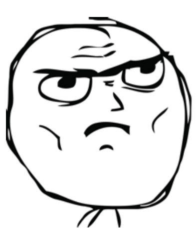
EMPTY!

The real deal: BFS in a Graph





If we just run **NOT_YET_BFS(t)** on the above graph. What problem would we have?



It would want to visit some vertices more than once (e.g., ***x***)

```
NOT_YET_BFS(root):
  Q ← Queue()
  Enqueue(Q, root)
  while Q not empty:
    x ← Dequeue(Q)
    print x
    for each neighbr c of x:
      Enqueue(Q, c)
```

How avoid visiting a vertex twice

Remember you visited it by **labelling** it using **colours**.

➔**White**: “unvisited”

➔**Gray**: “encountered”

➔**Black**: “explored”

- ➔ Initially all vertices are **white**
- ➔ Colour a vertex **gray** the **first** time visiting it
- ➔ Colour a vertex **black** when **all** its **neighbours** have been encountered
- ➔ Avoid visiting **gray** or **black** vertices
- ➔ In the end, all vertices are **black** (sort-of)

Some other values we want to remember during the traversal...

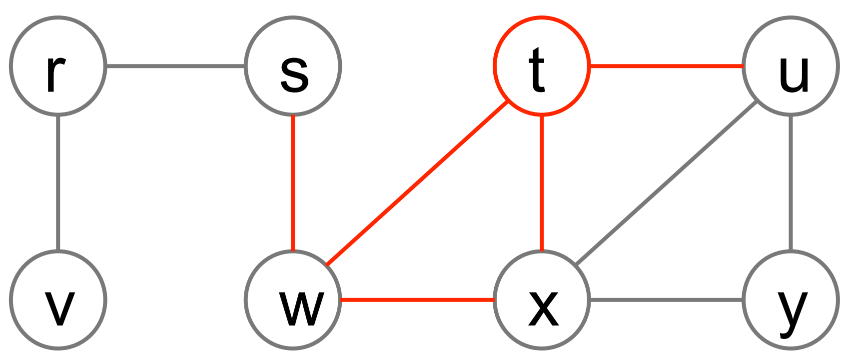
➔**pi[v]**: the vertex from which v is encountered

- "I was introduced as **whose** neighbour?"

➔**d[v]**: the distance value

- the distance from **v** to the source vertex of the BFS

This **d[v]** is going to be **really** useful!

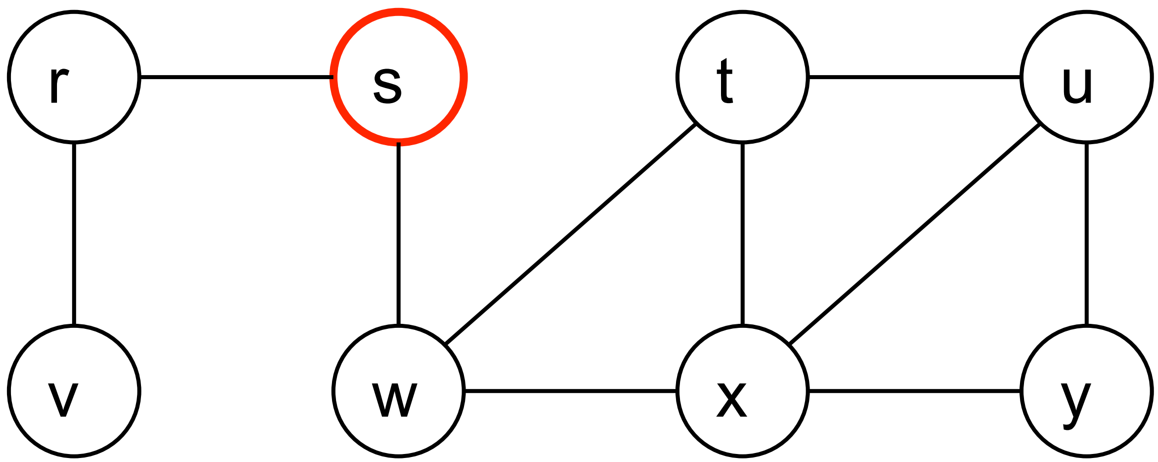


Pseudocode: the real BFS

```
BFS(G=(V, E), s):
    for all v in V:
        colour[v] ← white
        d[v] ← ∞                          # Initialize vertices
        pi[v] ← NIL
    Q ← Queue()
    colour[s] ← gray                      # source s is encountered
    d[s] ← 0                              # distance from s to s is 0
    Enqueue(Q, s)
    while Q not empty:                    # only visit unvisited vertices
        u ← Dequeue(Q)
        for each neighbour v of u:
            if colour[v] = white
                colour[v] ← gray
                d[v] ← d[u] + 1           # v is “1-level” farther from s than u
                pi[v] ← u                 # v is introduced as u’s neighbour
                Enqueue(Q, v)
        colour[u] ← black                 # all neighbours of u have been
                                          encountered, therefore u is explored
```

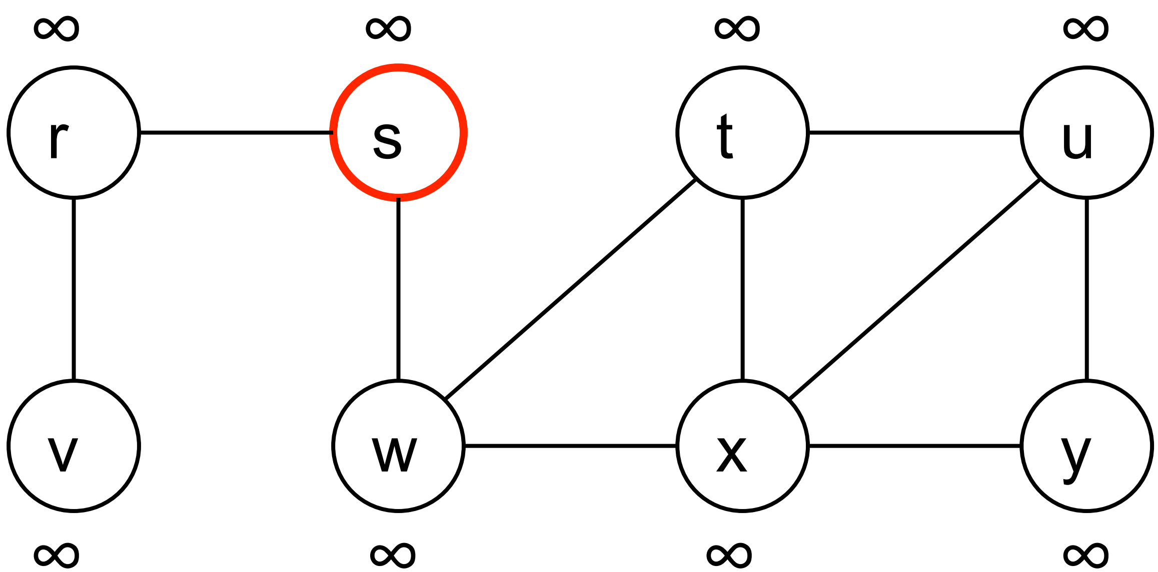
The blue lines are the same as NOT_YET_BFS

Let's run an example!



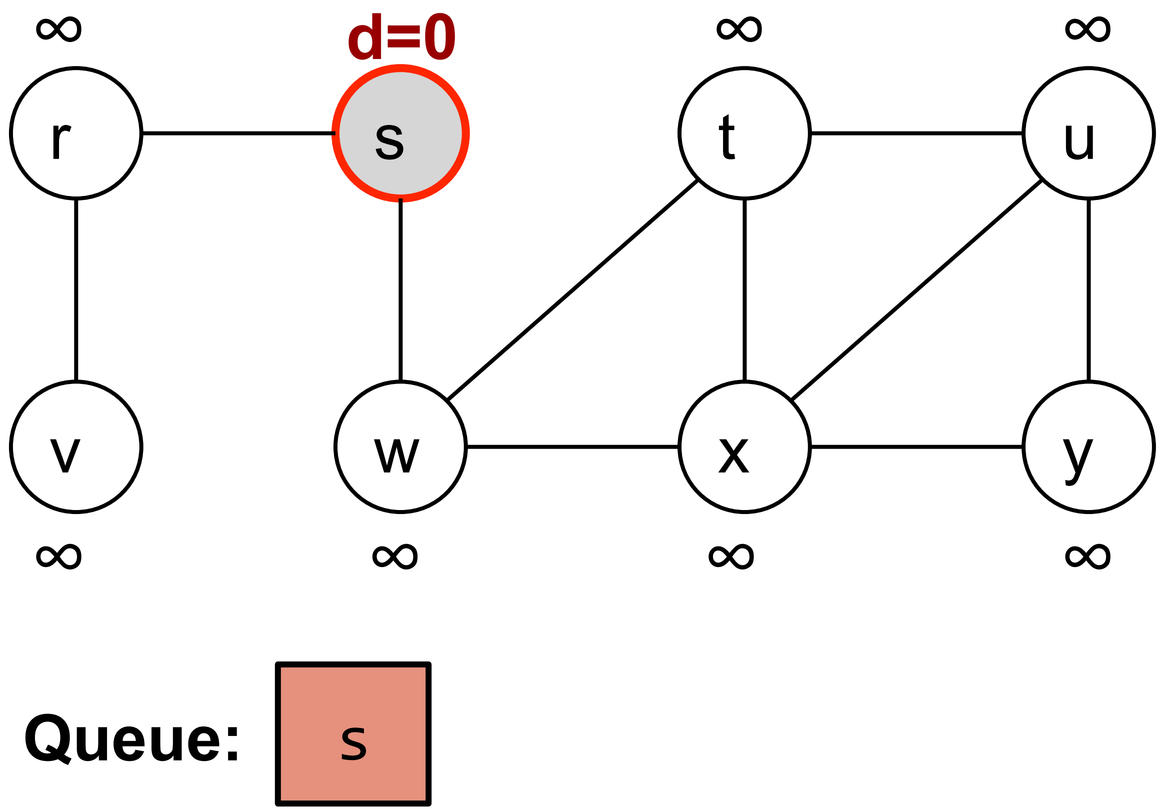
BFS(G, s)

After initialization



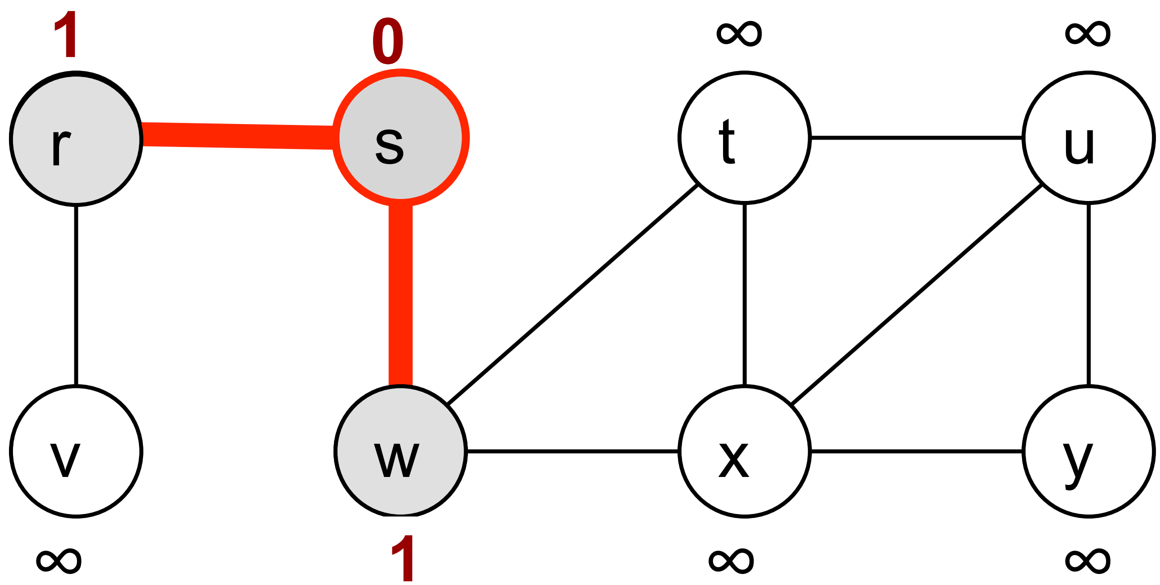
All vertices are **white** and have **d =** ∞

Start by "encountering" the source



Colour the source **gray** and set its d = 0, and Enqueue it

Dequeue, explore neighbours

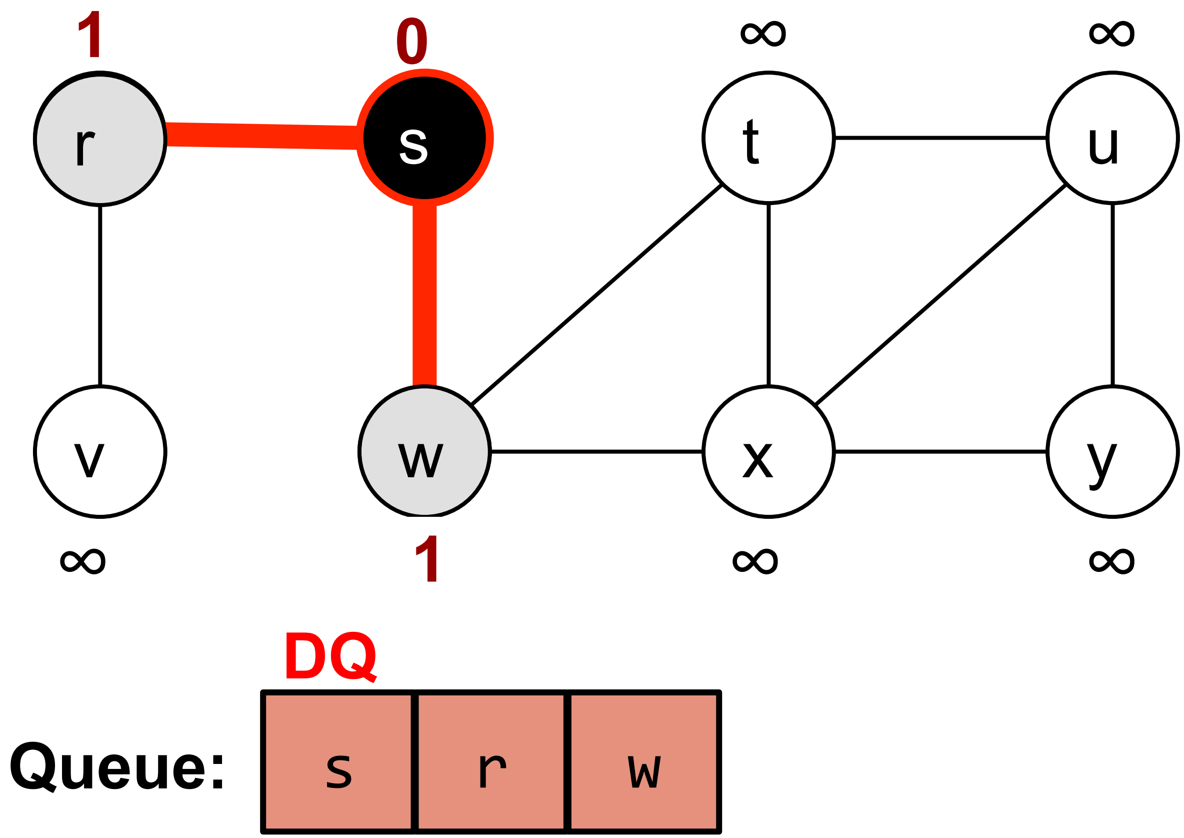


DQ

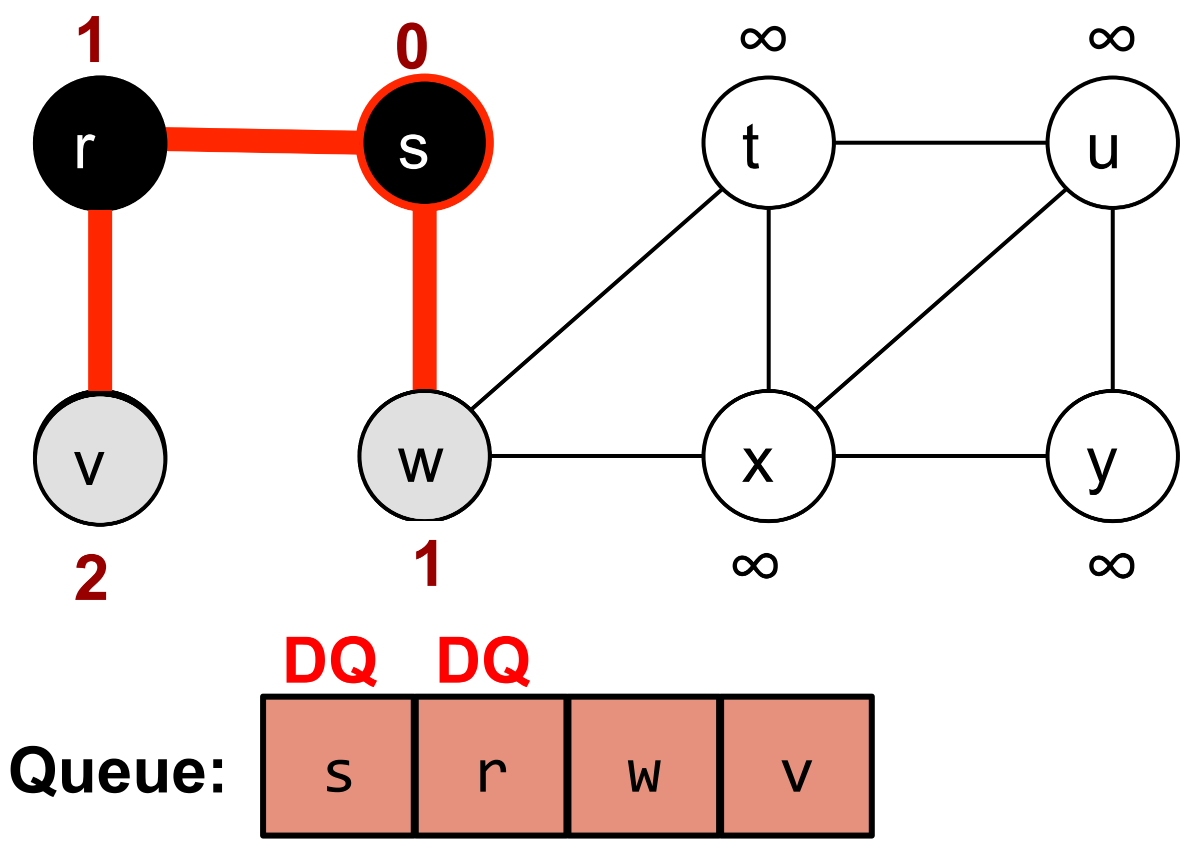
Queue: | s | r | w |

The red edge indicates the **pi[v]** that got remembered

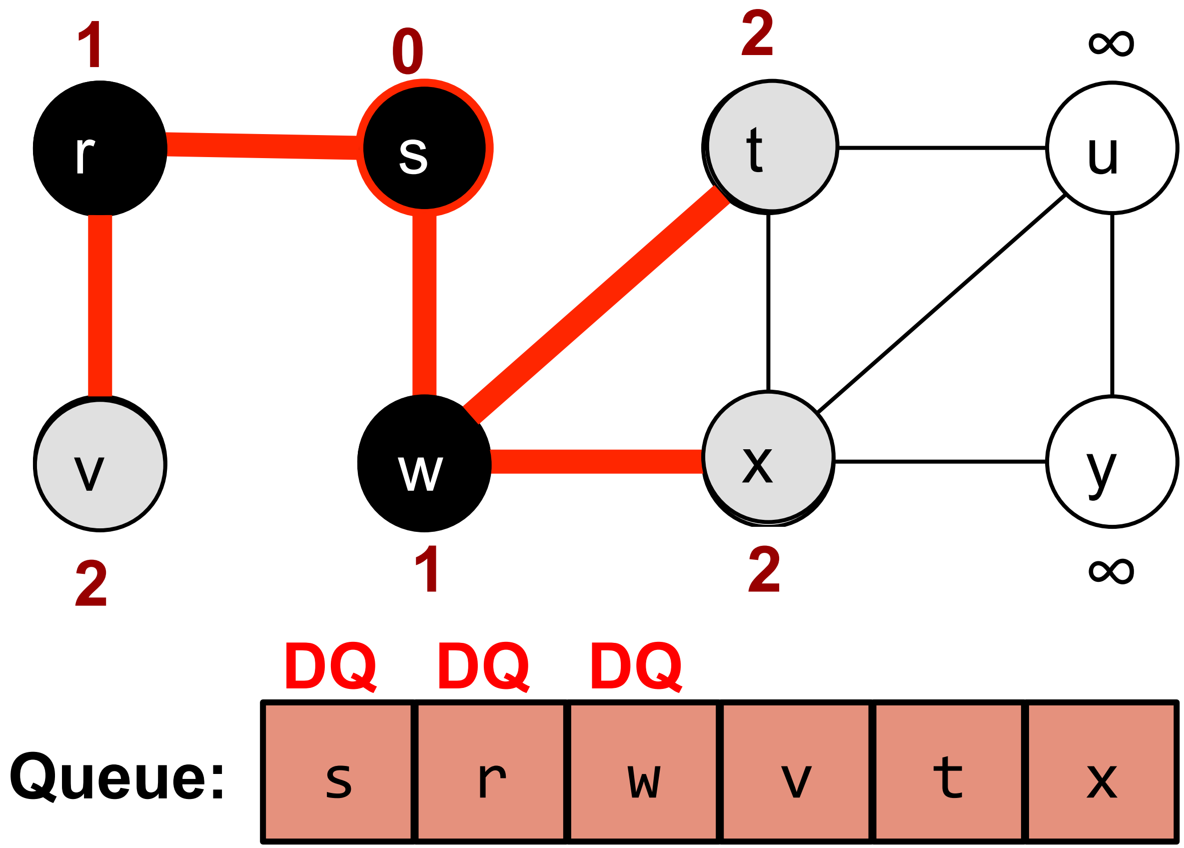
Colour black after exploring all neighbours



Dequeue, explore neighbours (2)

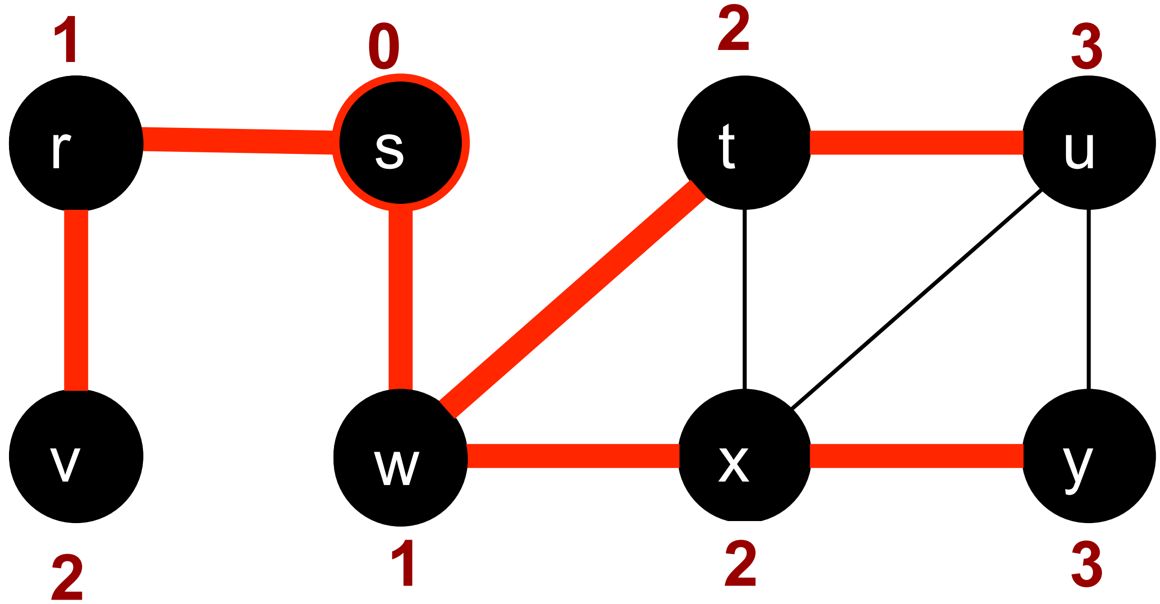


Dequeue, explore neighbours (3)



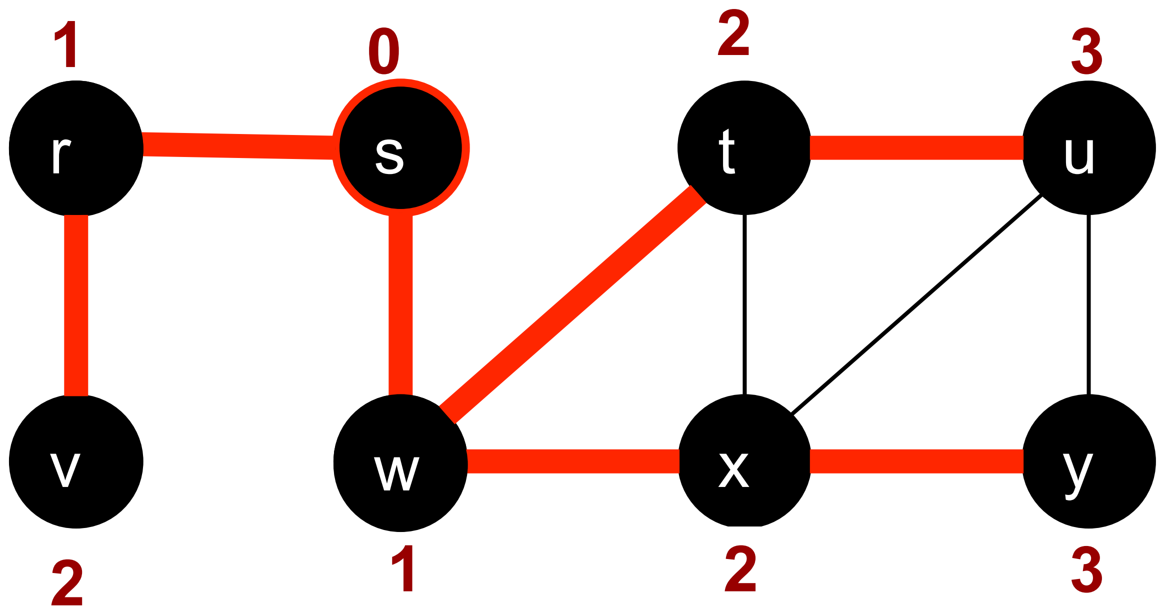
after a few more steps...

BFS done!



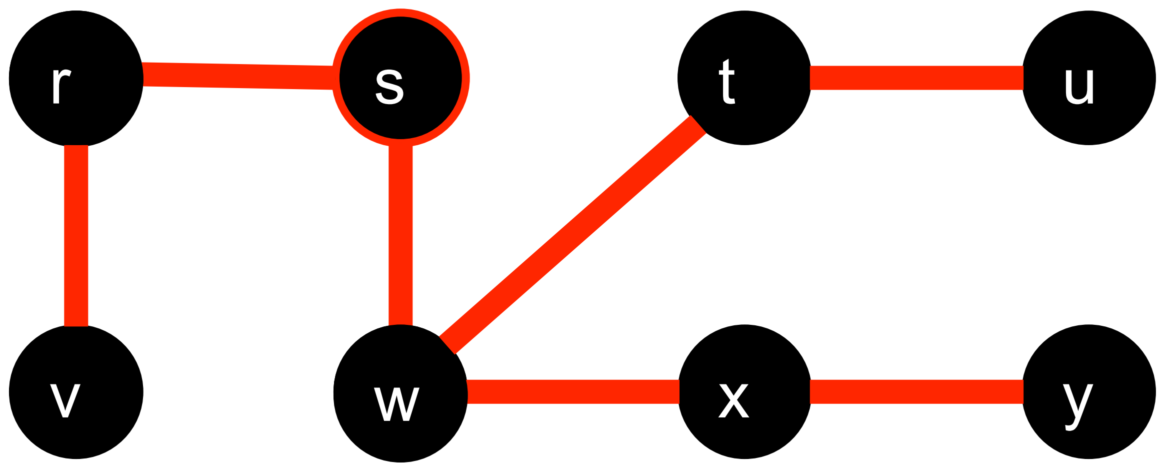
	DQ	DQ	DQ	DQ	DQ	DQ	DQ	DQ
Queue:	s	r	w	v	t	x	u	y

What do we get after doing all this?



First of all, we get to visit **every** vertex **once**.

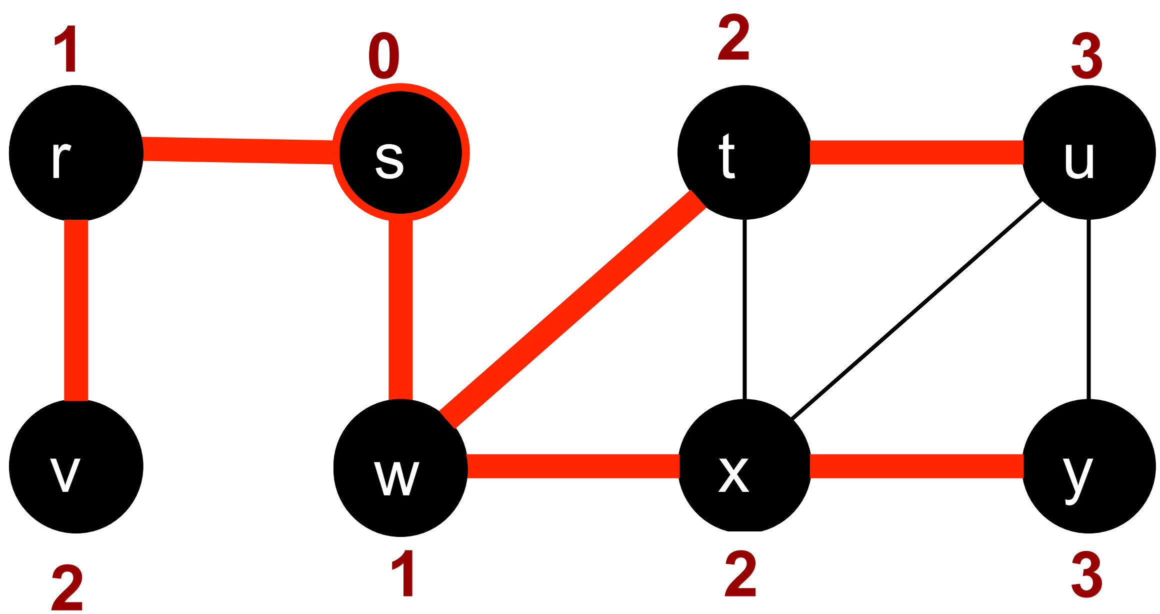
Did you know? The official name of the red edges are called **"tree edges"**.



This is called the **BFS-tree**, it's a **tree** that connects all vertices, if the graph is **connected**.

These **d[v]** values, we said they were going to be really useful.

Short path from u to s:
u → pi[u] → pi[pi[u]] → pi[pi[pi[u]]] → … → s

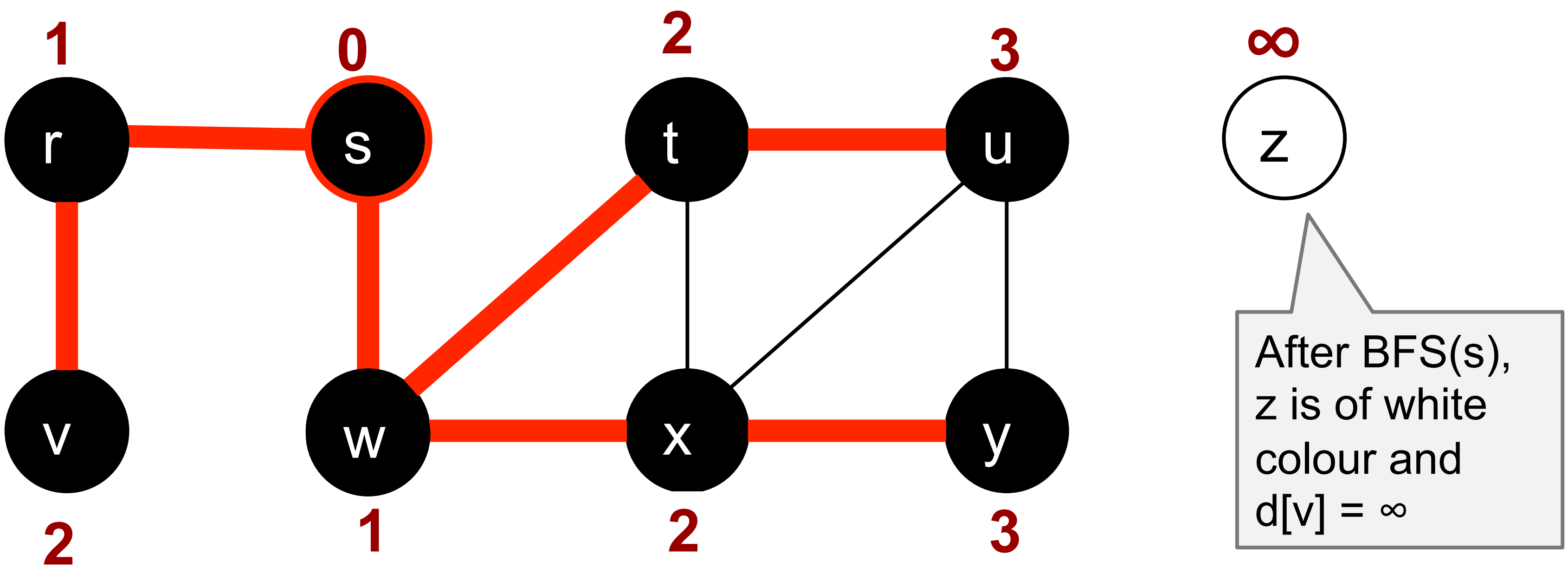


The value indicates the vertex's **distance** from the source vertex.

Actually more than that, it's the **shortest-path distance**, we can prove it.

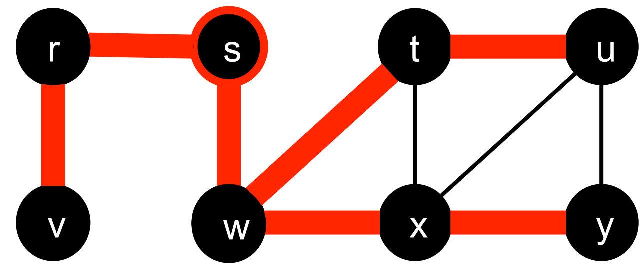
How about finding **short path** itself?
Follow the red edges, **pi[v]** comes in handy for this.

What if G is disconnected?



The infinite distance value of **z** indicates that it is **unreachable** from the source vertex.

Runtime analysis!



The total amount of work (use **adjacency list**):

- ➔ Visit each vertex once
  - ◆ Enqueue, Dequeue, change colours, assign d[v], …, constant work per vertex
  - ◆ in total: **O(|V|)**
- ➔ At each vertex, check all its neighbours (all its **incident edges**)
  - ◆ Each edge is checked **twice** (by the two end vertices)
  - ◆ in total: **O(|E|)**

Total runtime:

O(|V|+|E|)

Summary of BFS

- ➔Explores **breadth** rather than **depth**
- ➔Useful for getting **single-source shortest paths** on **unweighted** graphs
- ➔Useful for testing reachability
- ➔Runtime **O(|V|+|E|)** with adjacency **list** (with adjacency **matrix** it'll be different)

Next week

DFS BFS