CSC263 Winter 2020

Graphs

Lecture 7

Problem Set 1

Results are out

Problem 1

Why did marking take so long?

- Unexpected errors in programming submissions
- Test your code. Many of them had compile errors.
- Use the practice test given to match format
- Do **NOT** print anything. Just return the expected answer!
- Write your own tests and run them (remove them for submission)

Problem 1

Programming Question

Our solution code runs in **3 seconds** (**all** tests included)

- Timeout for automated testing **120 seconds**
- If your code times out, the complexity is almost surely incorrect.
- Bugs can be subtle 

Example (real submission)

equality check `heap == max_heap`

runs in time O(n), n is the size of the heap, not O(1)
resulting in your complexity being off by O(n)

Problem Set 1

Remarking

Remarking instructions will be posted on the discussion board shortly

Consider Remarking?

If the description of your code does
not meet the complexity requirements,
the programming part will not be awarded any points.

Please see the annotations on your pdf solutions
before submitting a remarking request
for the programming assignment.

Consider Remarking?

Website: One of the largest cases used to test your code, along with the solution

Our solution runs on this test in 0.6 seconds on my laptop.

`pretend_sorted_array_test.py` Profiles your code for debugging

Midterm

Outside of class! 6:00PM, Fri Mar 8, IB110
Midterm is 80 minutes

Coverage: everything up to and including week 6 material

Don't forget your T-card!

You can bring one 8.5x11 double-sided sheet

Midterm

Types of Questions

Multiple choice, true/false

Short answer questions (e.g. giving explanations)

Long answer (e.g. analysis, design)

Midterm

We will Test You on

- Understanding of data structures.
  What are they good for, how do they work, what are their properties?

- Problem-solving using appropriate data structures

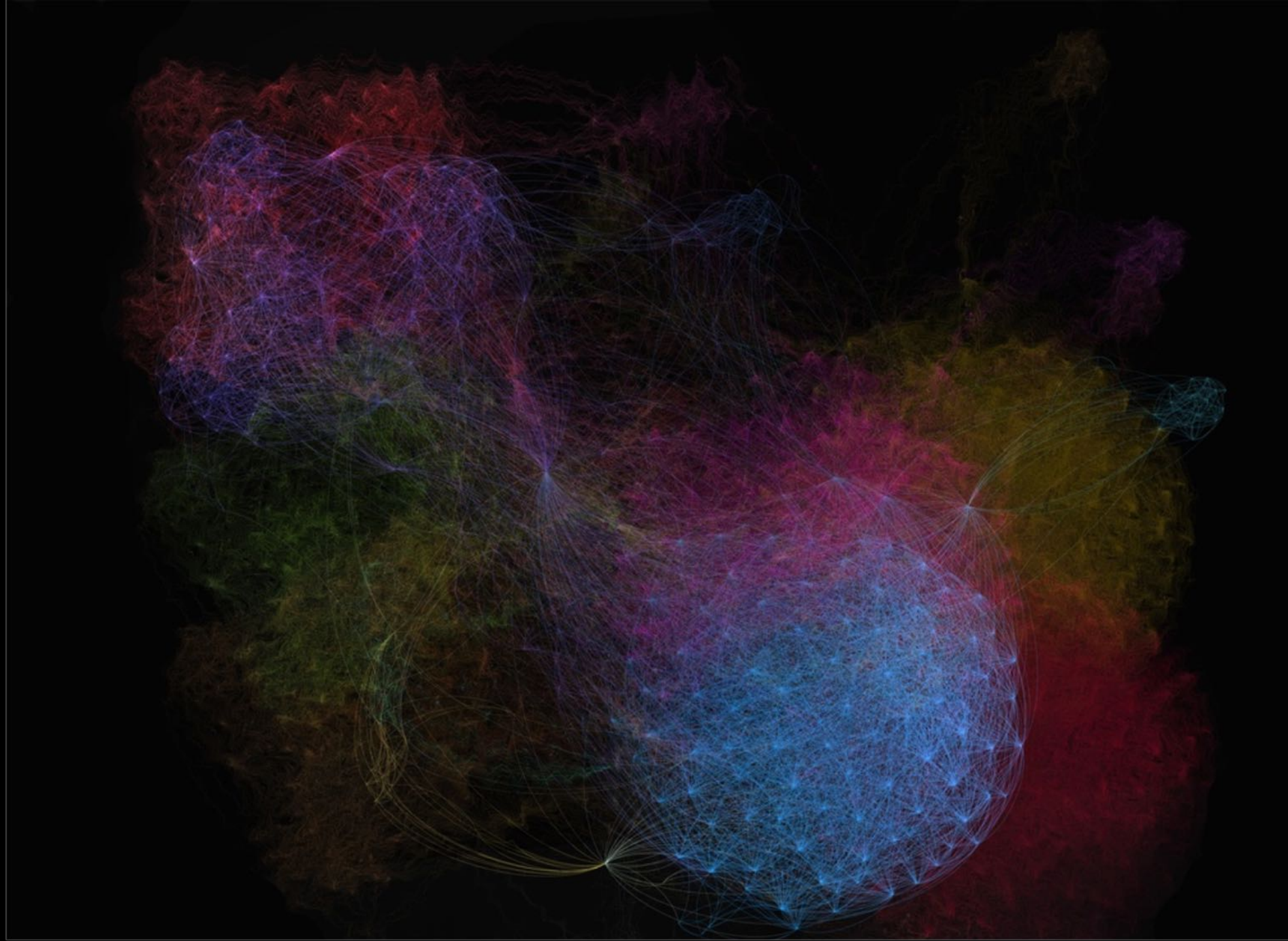
- Stuff from problem sets, tutorials, lectures

Midterm

How to study

- Be active! Solve problems (past tests/exams, course notes, textbook)
- Revisit problem sets and tutorials
- Read lecture slides to review
- Consult textbook or course notes when you want more information or to clear up confusion
- We'll do a mock midterm in the tutorial on Tue Mar 5

Graphs



What can be modelled using graphs

Web

Facebook

Task scheduling

Maps & GPS

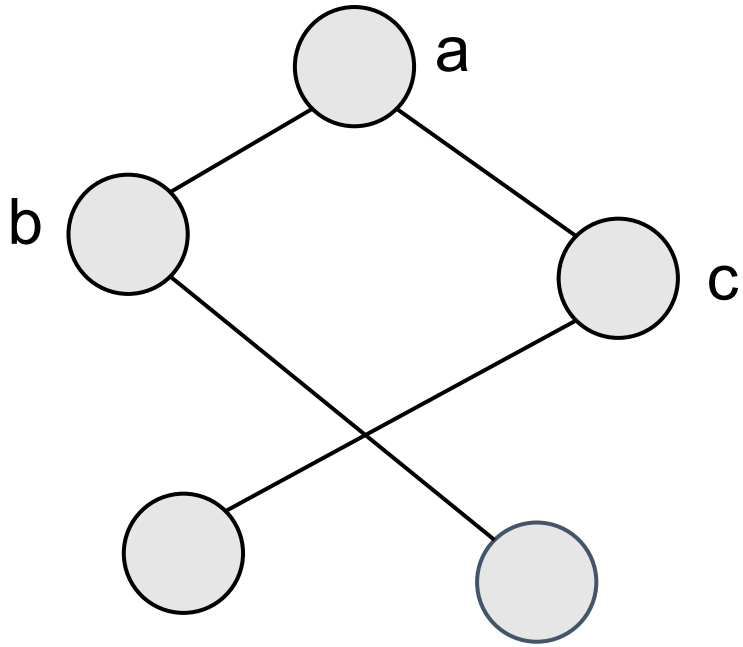
Compiler (garbage collection)

Database

Rubik's cube

….

Definition



G = (V, E)

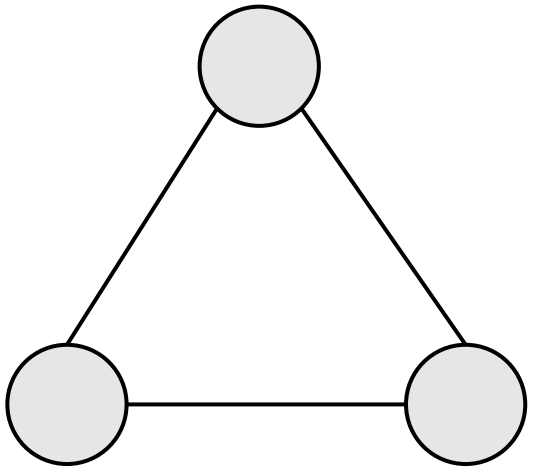
Set of **vertices**
e.g., V={a, b, c}

aka **nodes**

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

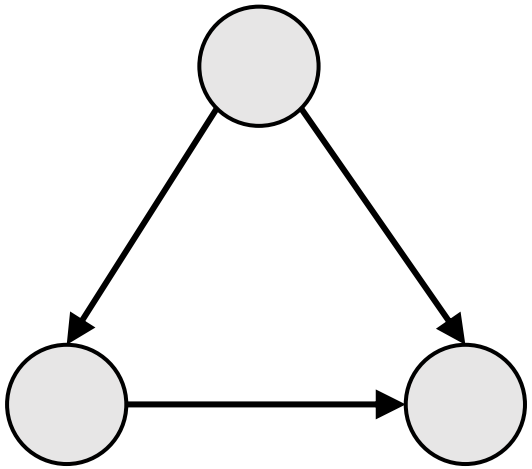
Types of Graphs

Undirected



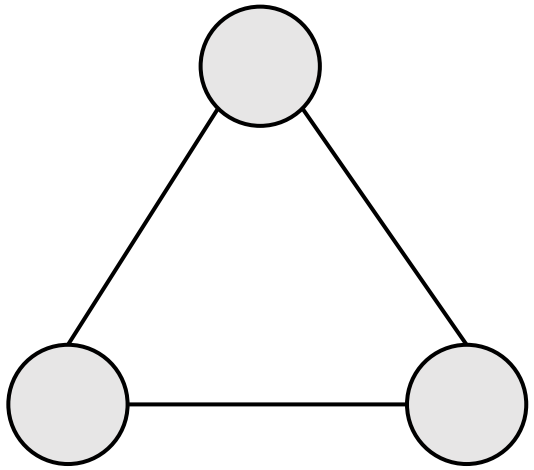
each edge is an
unordered pair
(u, v) = (v, u)

Directed



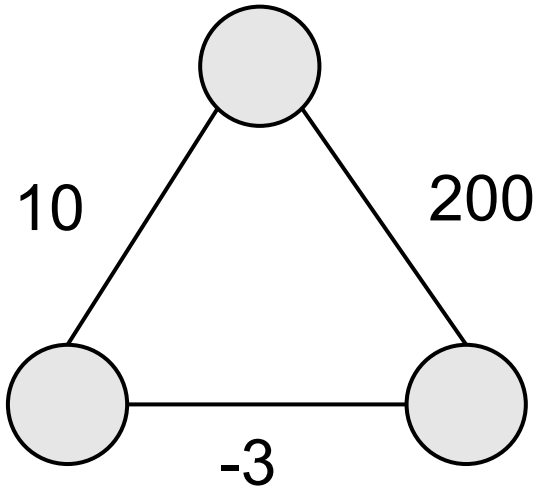
each edge is an
ordered pair
(u, v) ≠ (v, u)

Unweighted



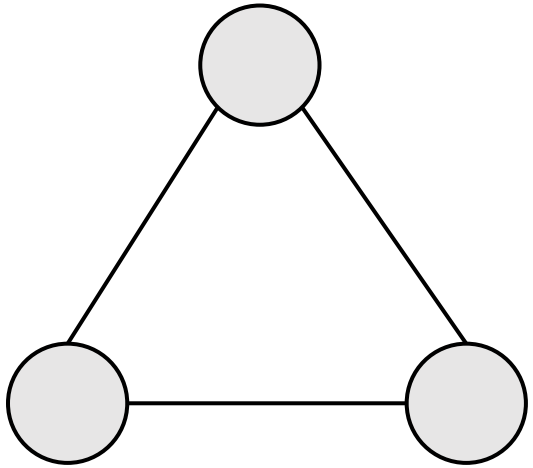
edges have no weights

Weighted



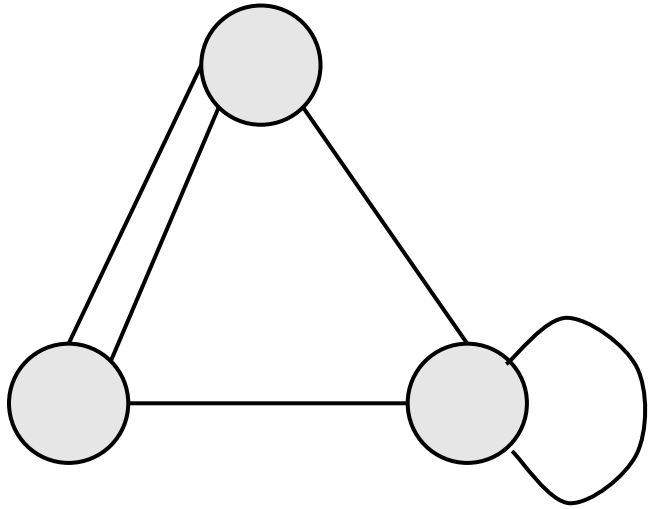
edges have weights

Simple

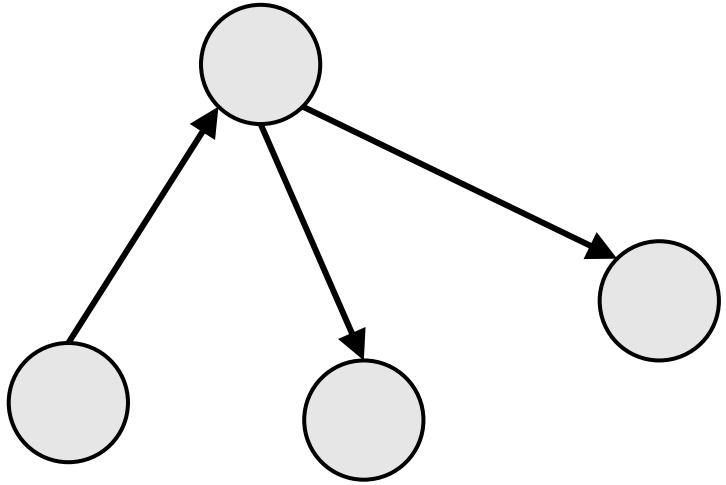


at most one edge between pairs of vertices, no self-loop

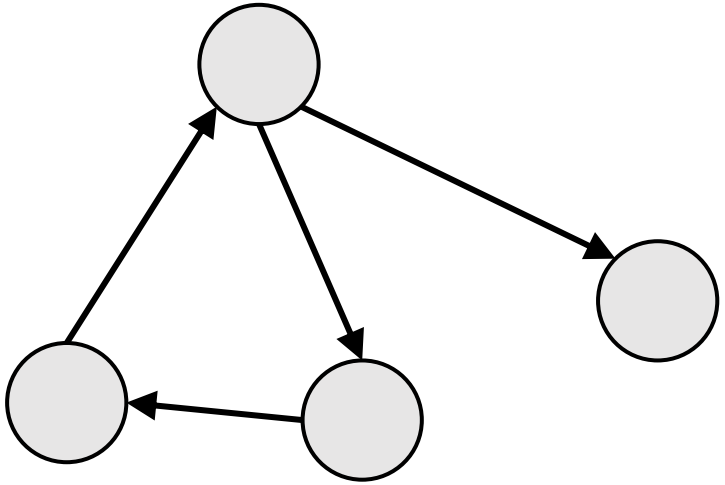
Non-simple



Acyclic

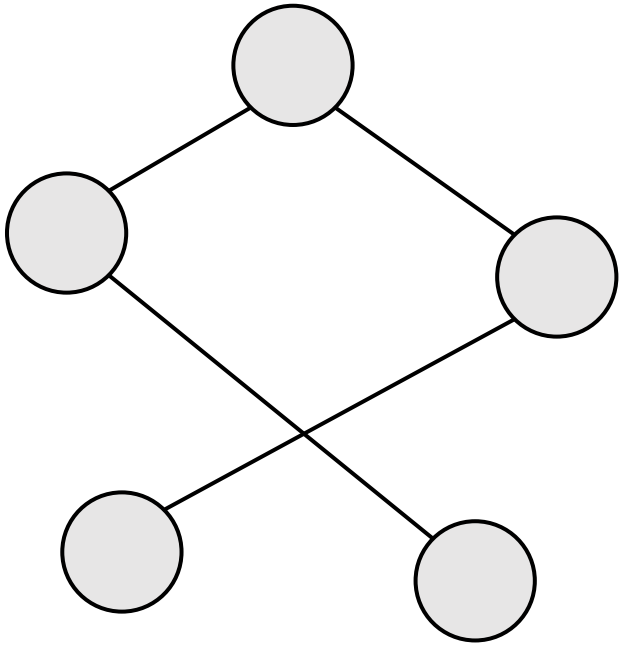


Cyclic



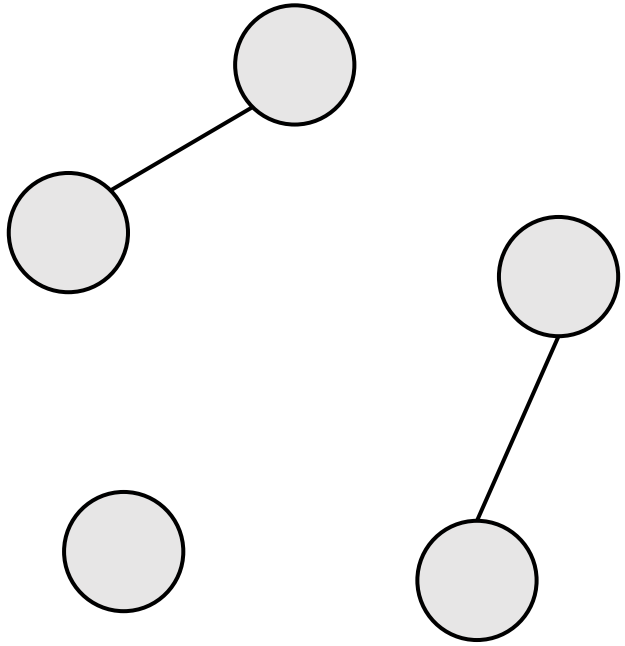
using each edge at most once, you cannot start at the some vertex and go back to it

Connected

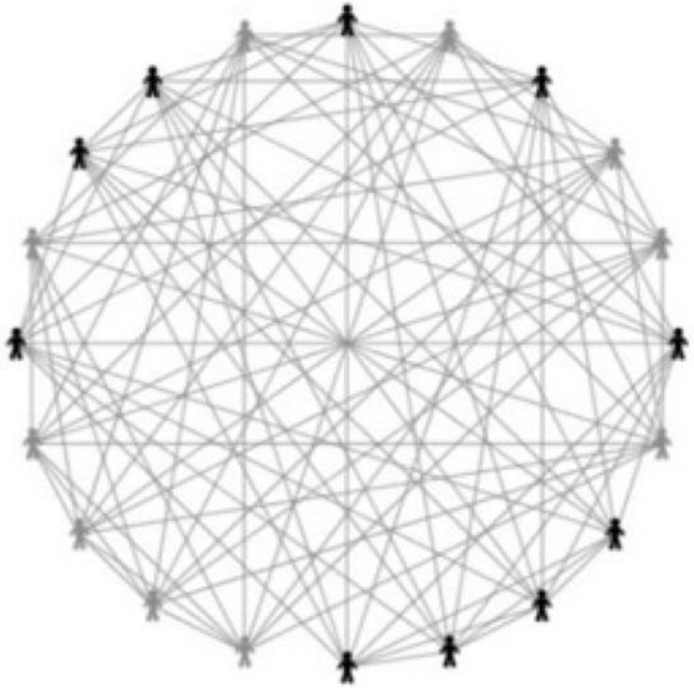


can get from any vertex
to any other vertex

Disconnected

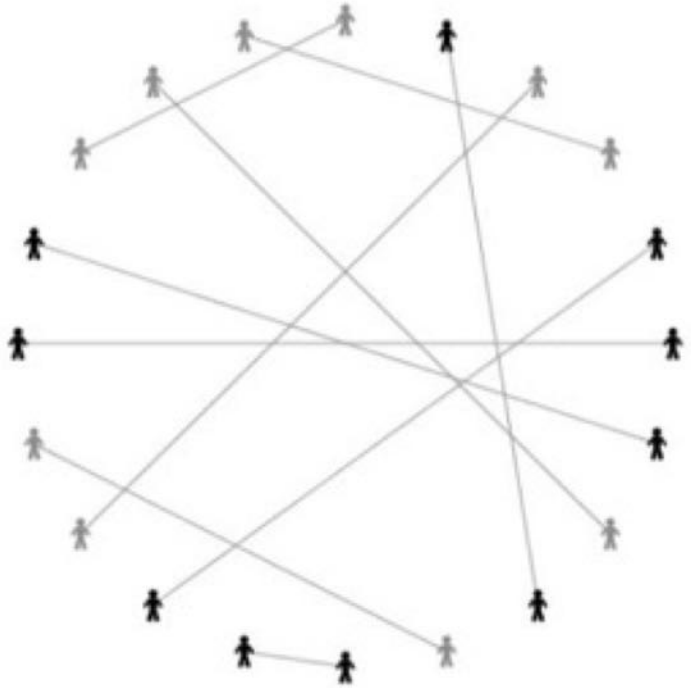


Dense



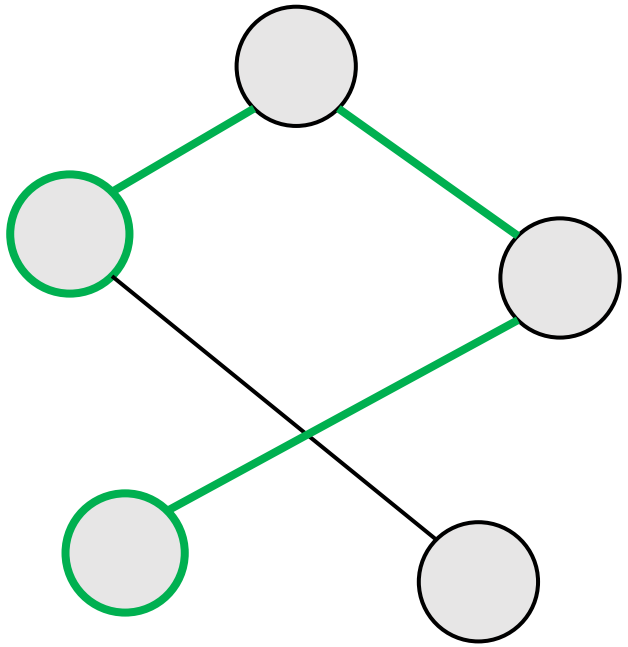
most edges are present

Sparse



Path in a Graph

A path of length 3



Read CLRS Appendix B.4 for more background on graphs.

A **path** begins at some start node
and follows 0 or more edges to some end node

Length of path = **number of edges**

Graph Operations

- **Add** a vertex; **remove** a vertex
- **Add** an edge; **remove** an edge
- Determine whether an edge (i,j) is **present**
- Get **neighbours**
  - For vertex u, return all v ∈ V such that (u, v) ∈ E (undirected graph)
  - Get **in**-neighbours / **out**-neighbours (directed graph)
- **Traversal**: visit all vertices in the graph

Data Structures for Graph ADT

Adjacency **Matrix**

Adjacency **List**

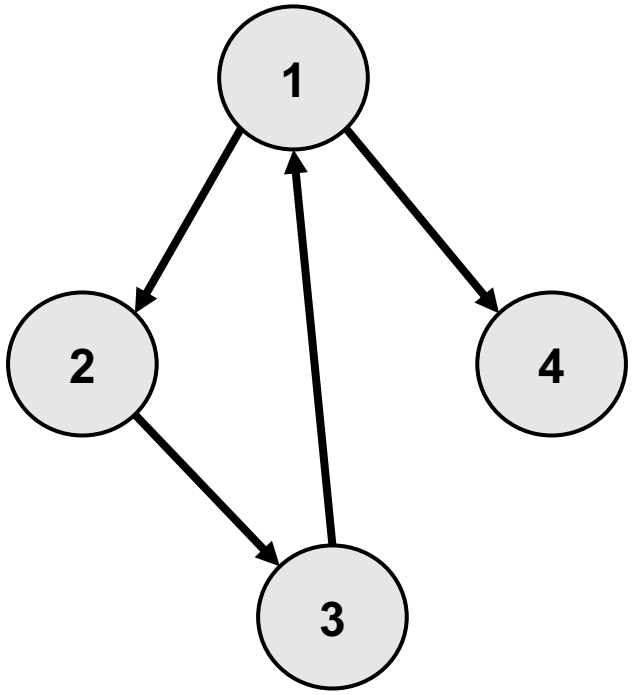
Adjacency Matrix

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

Adjacency matrix $\boldsymbol{A}$ is a $|V| \times |V|$ array

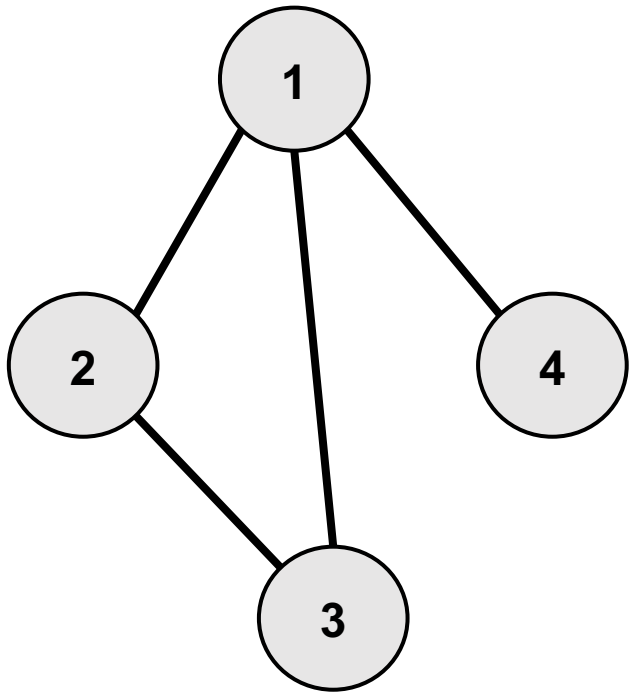
$$A[i,j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency Matrix (directed graph)



	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: Space Complexity

How much space is required for the adjacency matrix?

- $|V|$ rows, each with $|V|$ columns
- Total: $|V|^2$

So, space is $\Theta(|V|^2)$ for both directed and undirected graphs.

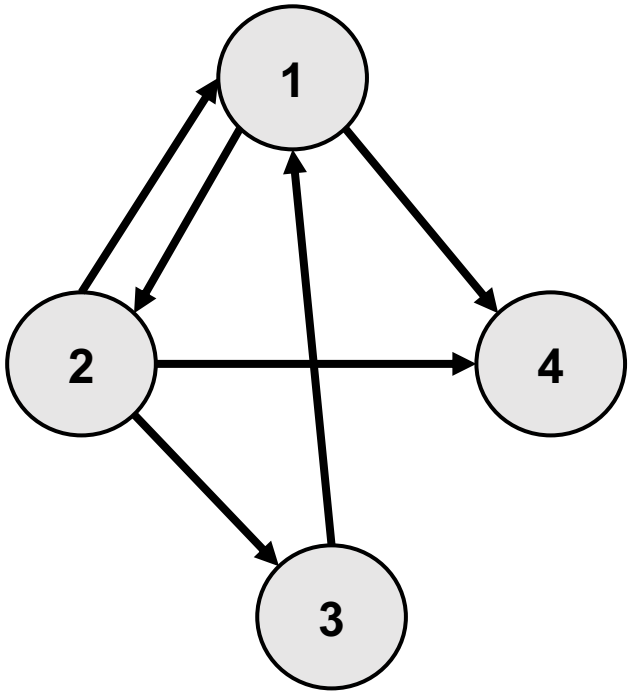
Adjacency List

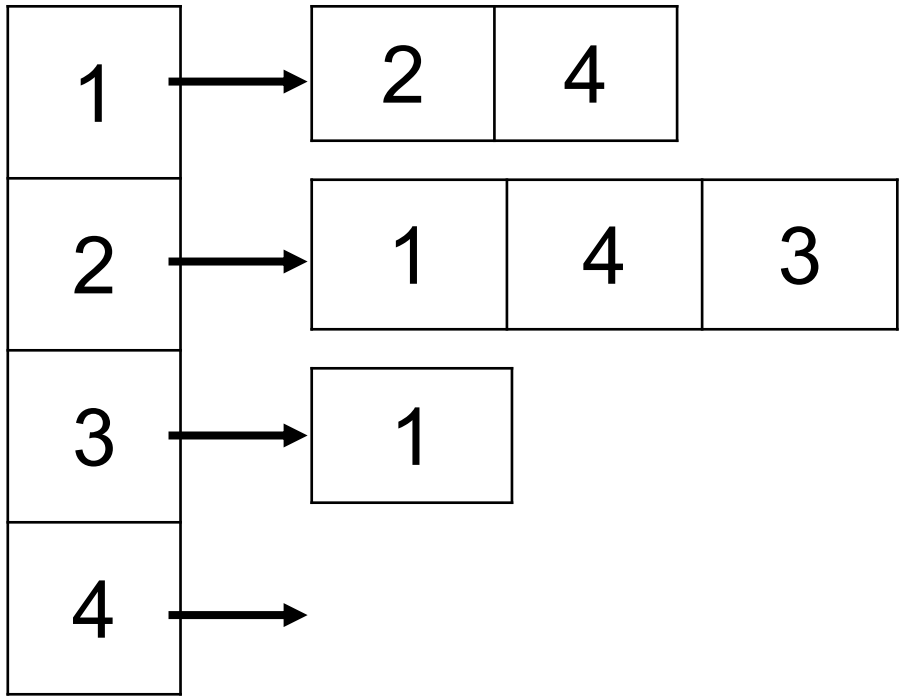
Adjacency List

In an adjacency list, each vertex stores a list of vertices

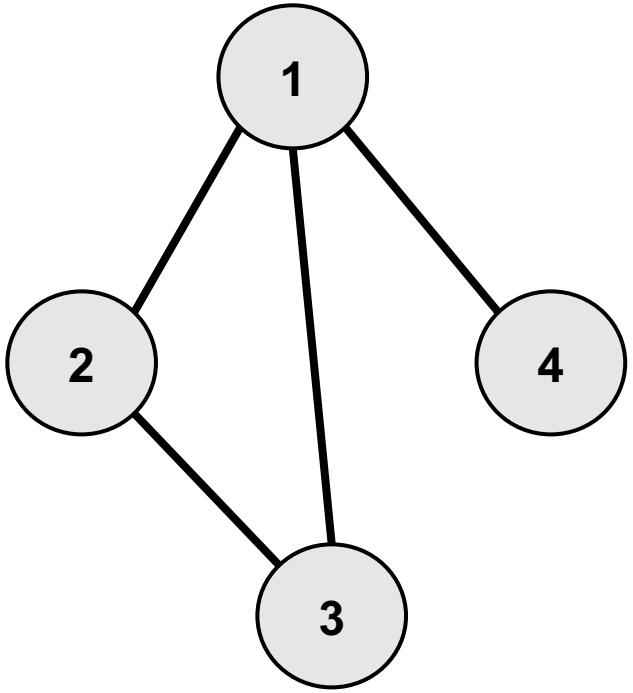
The list for vertex v_i stores all vertices v_j such that $(v_i, v_j) \in E$

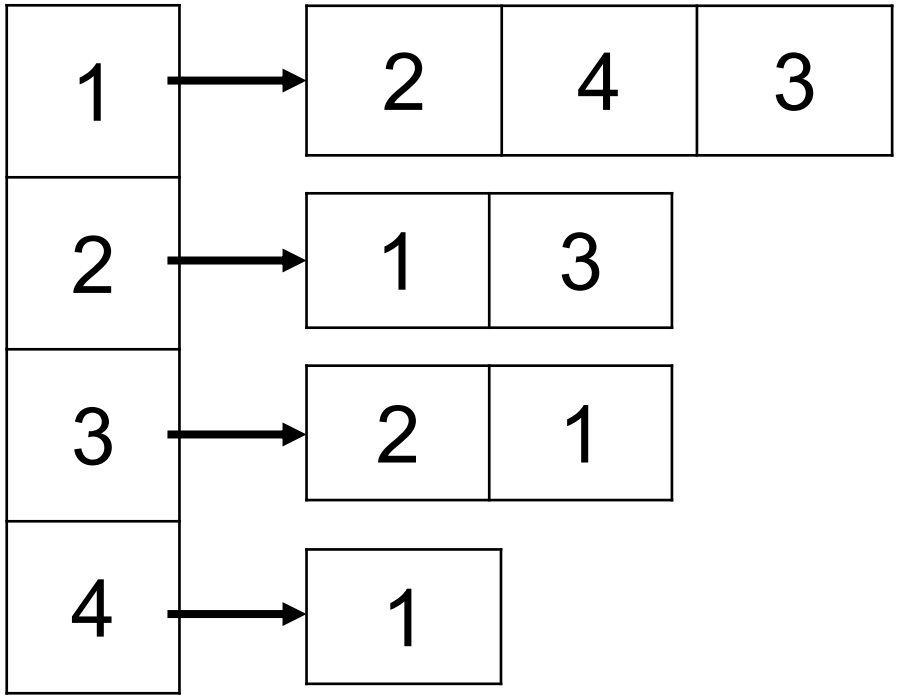
Adjacency List (directed graph)





Adjacency List (undirected graph)





Adjacency List: Space Complexity

How much space is required for the adjacency list?

- We store each node, so that is |V|
- We also have one entry for each edge
  - Directed graph: |E|
  - Undirected graph: 2|E|

So, space is **Θ(|V|+|E|)** for both directed and undirected graphs.

Matrix vs List

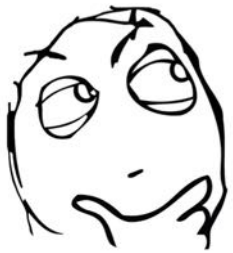
In term of space complexity

- adjacency matrix is **$\Theta(|V|^2)$**
- adjacency list is **$\Theta(|V|+|E|)$**

Which one is more space-efficient?

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

Matrix vs List



Anything that **Adjacency Matrix** does better than **Adjacency 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.

7 min

Graph Traversals

BFS and DFS

Example

Social Network

Get the names of everyone

who is connected

to a particular person.

Graph Traversals

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

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

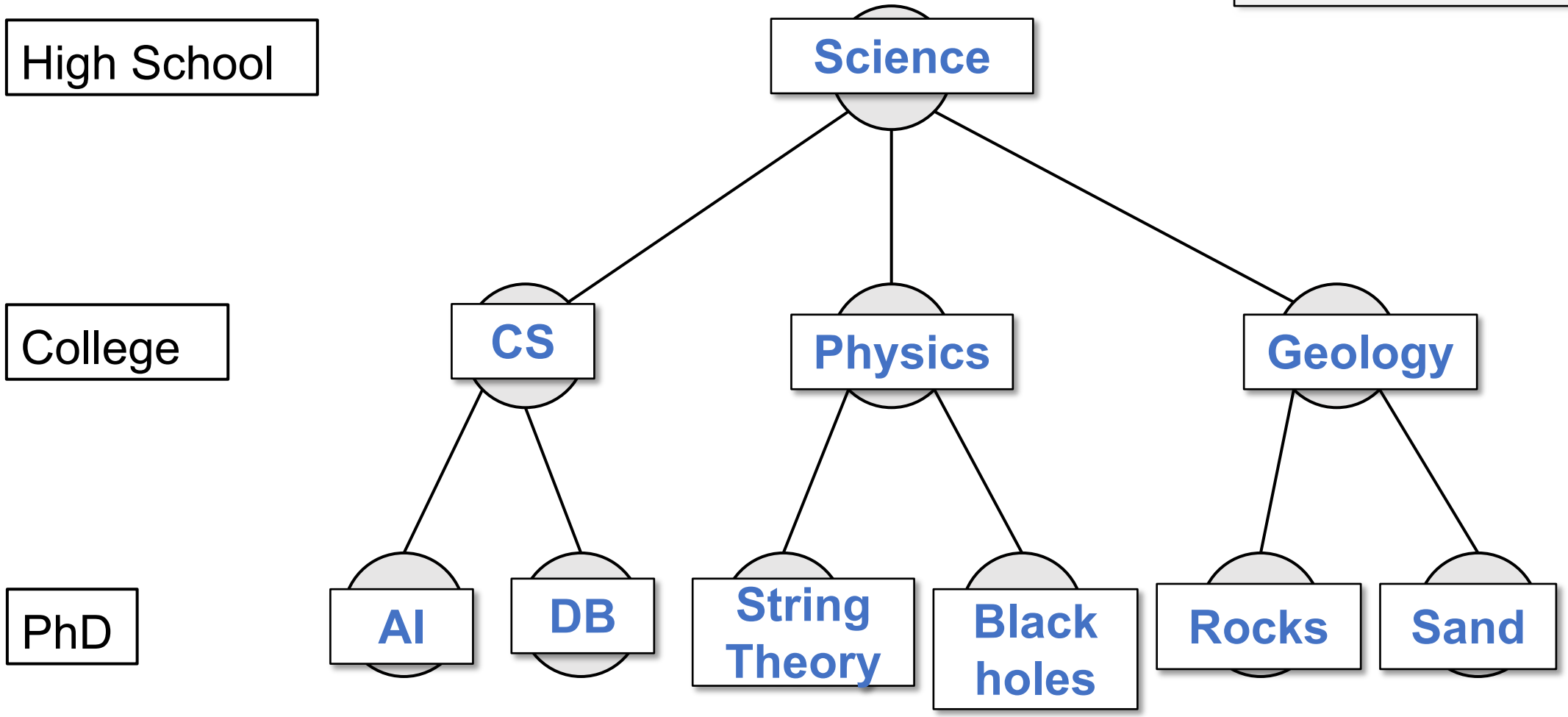
- **Breadth** First Search (**BFS**)
- **Depth** First Search (**DFS**)

Intuitions of BFS and DFS

Consider a special graph - a **tree**

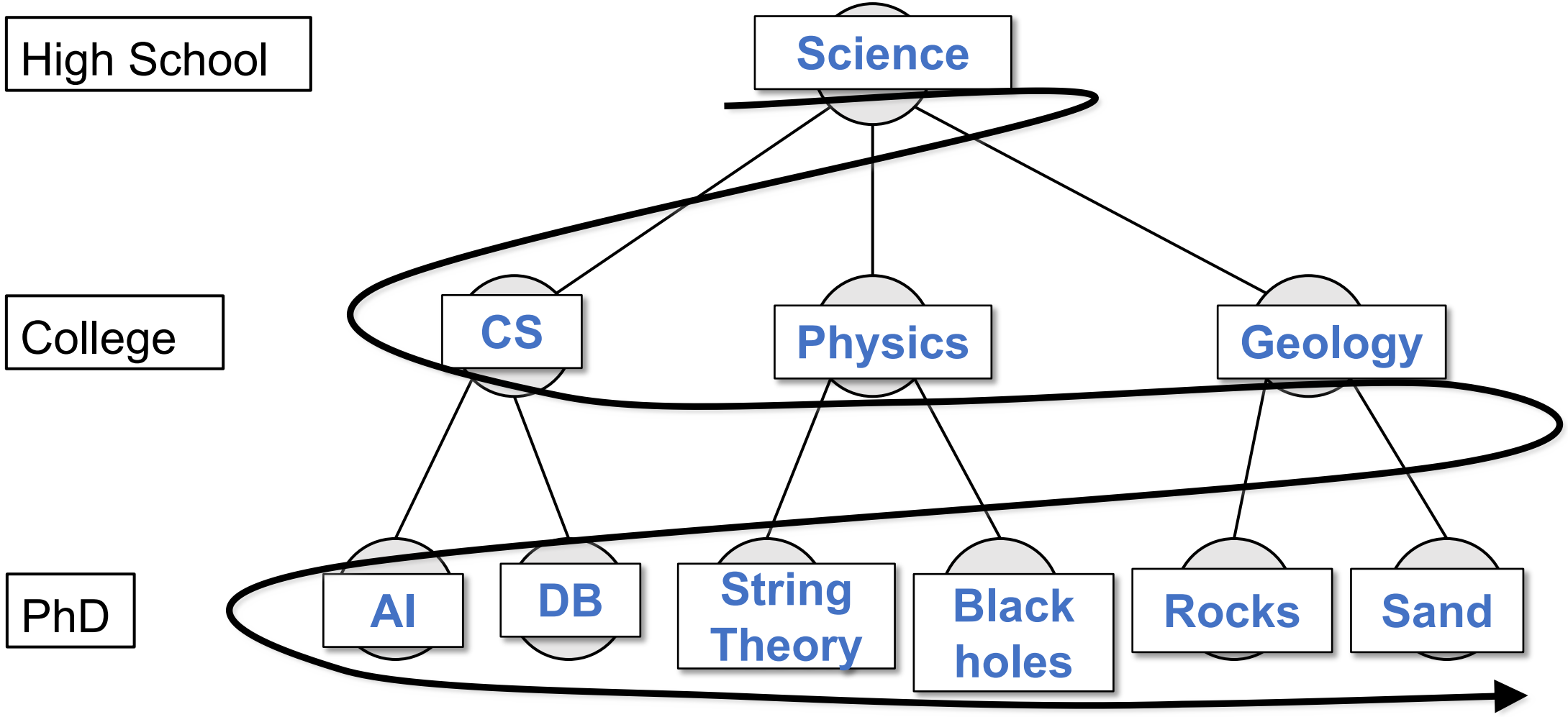
“The knowledge learning tree”

Traversing this graph means learning all these subjects.



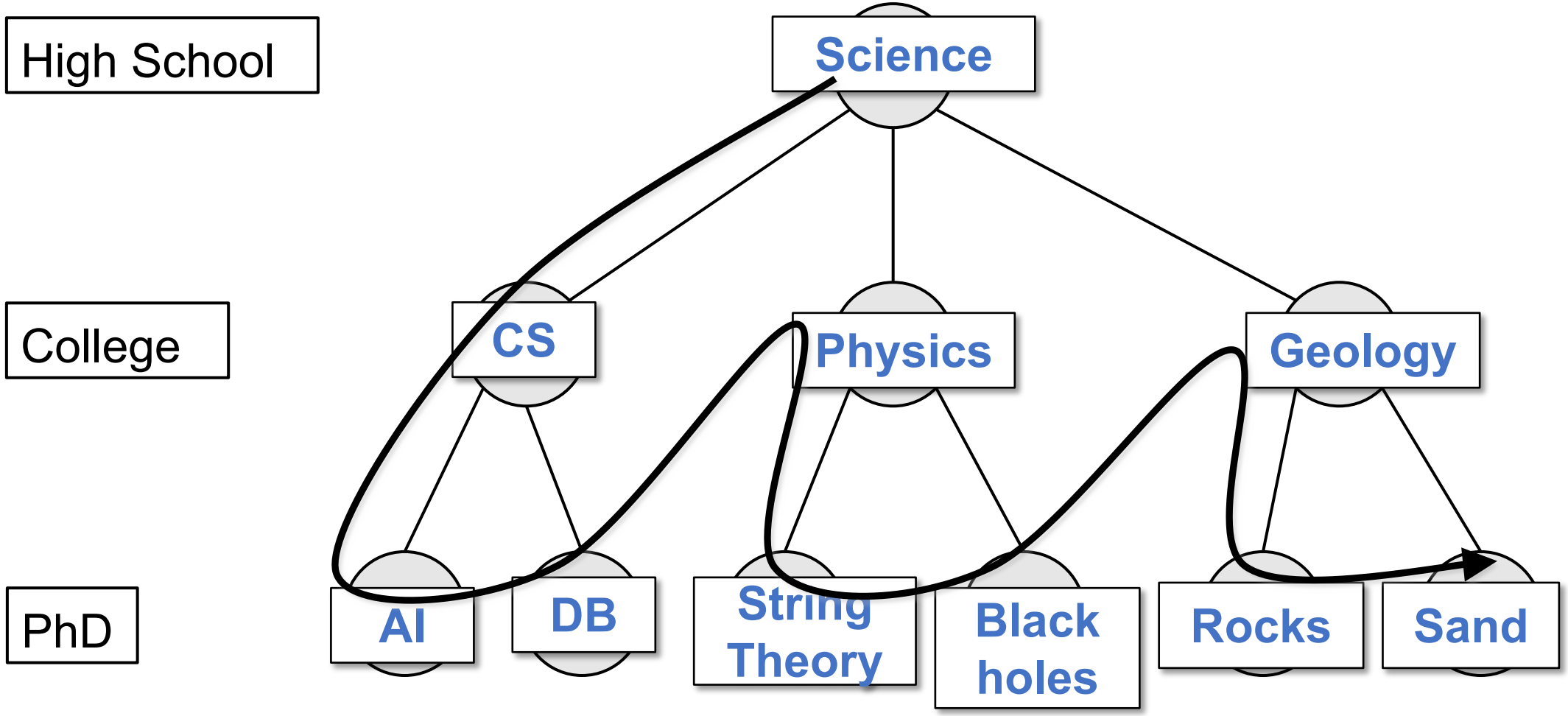
The Breadth-First ways of learning these subjects

➔ 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.



Special Case: BFS in a Tree

Review CSC148

BFS in a tree (starting from root) is a **level-by-level** (NOT preorder!) traversal.

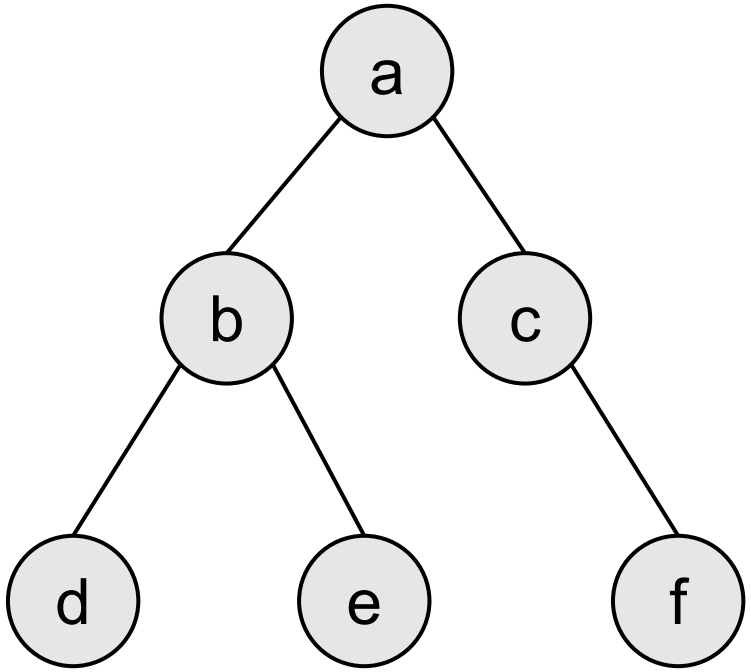
What ADT did we use for implementing the **level-by-level** traversal?

Queue!

Special Case: BFS in a Tree

Output:

```
a
b
c
d
e
f
```



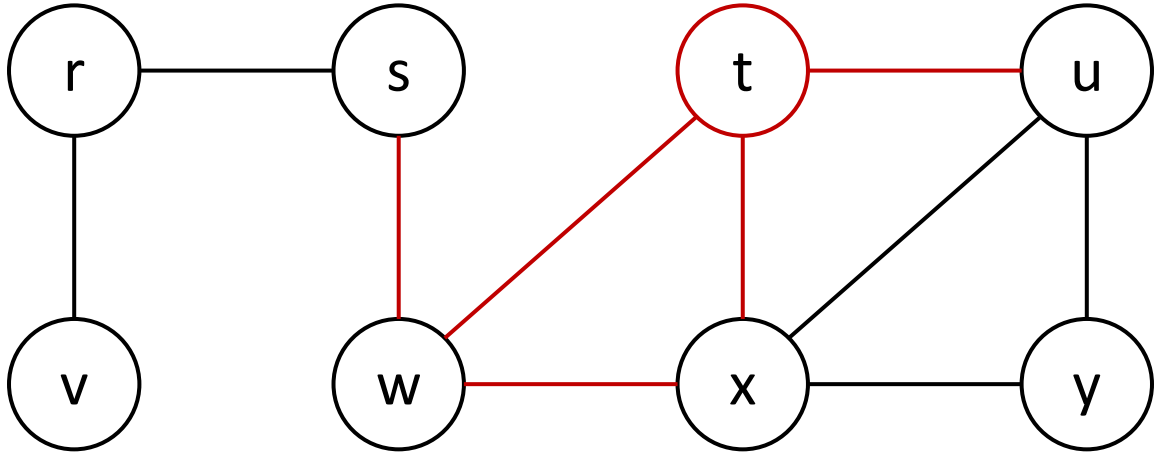

```
NOT_QUITE_BFS(root):
  Q ← Queue()
  Enqueue(Q, root)
  while Q not empty:
    x ← Dequeue(Q)
    print x
    for each child c of x:
      Enqueue(Q, c)
```

Queue:

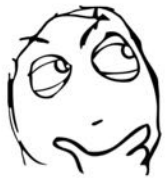
a	b	c	d	e	f
DQ	DQ	DQ	DQ	DQ	DQ

EMPTY!

BFS in a Graph




```
NOT_QUITE_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)
```



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

It would want to visit some vertex **twice** (e.g., ***x***), which must be **avoided**!

BFS in General

How can we avoid visiting a vertex multiple times?

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

➔ **White**: “unvisited”
➔ **Gray**: “encountered”
➔ **Black**: “explored”

➔ Initially all vertices start off as **white**
➔ Colour a vertex **gray** the **first** time visiting it
➔ Colour a vertex **black** when **all** its **neighbours** have been encountered
➔ Don’t visit **gray** or **black** vertices
➔ In the end, all vertices are **black** (sort-of)

BFS in General

We are going to remember two more things about each node.

- **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

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

BFS Pseudocode

```
BFS(G=(V, E), s):
 1    for all v in V:          # Initialize vertices
 2        colour[v] ← white
 3        d[v] ← ∞
 4        pi[v] ← NIL
 5    Q ← Queue()
 6    colour[s] ← gray         # start BFS by encountering the source vertex
 7    d[s] ← 0                 # distance from s to s is 0
 8    Enqueue(Q, s)
```

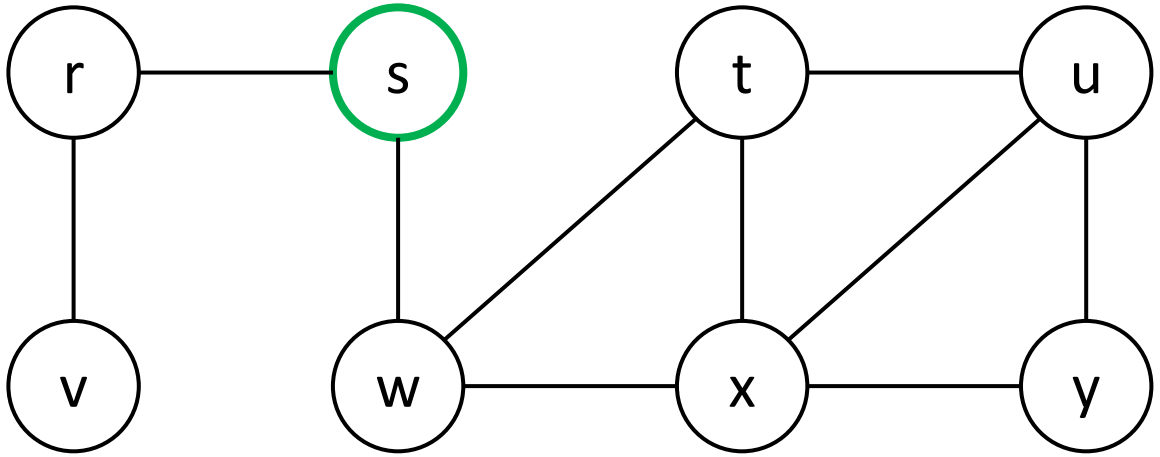
The blue lines are the same as NOT_QUITE_BFS

BFS Pseudocode

```
BFS(G=(V, E), s):
 1    for all v in V:
 2        colour[v] ← white
 3        d[v] ← ∞                  # Initialize vertices
 4        pi[v] ← NIL
 5    Q ← Queue()
 6    colour[s] ← gray              # start BFS by encountering the source vertex
 7    d[s] ← 0                      # distance from s to s is 0
 8    Enqueue(Q, s)
 9    while Q not empty:
10        u ← Dequeue(Q)
11        for each neighbour v of u:
12            if colour[v] = white  # only visit unvisited vertices
13                colour[v] ← gray
14                d[v] ← d[u] + 1   # v is at 1 more distance than u
15                pi[v] ← u         # v is introduced as u's neighbour
16                Enqueue(Q, v)
17        colour[u] ← black         # all neighbours of u have been
                                    # encountered, therefore u is explored
```

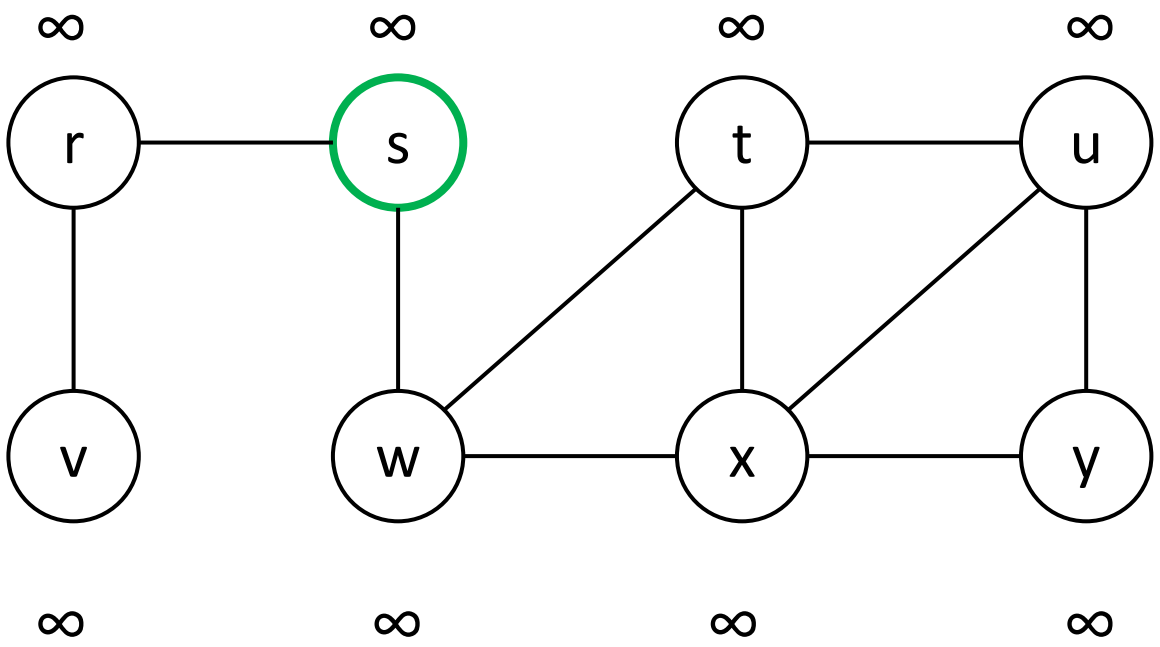
The blue lines are the same as NOT_QUITE_BFS

Example



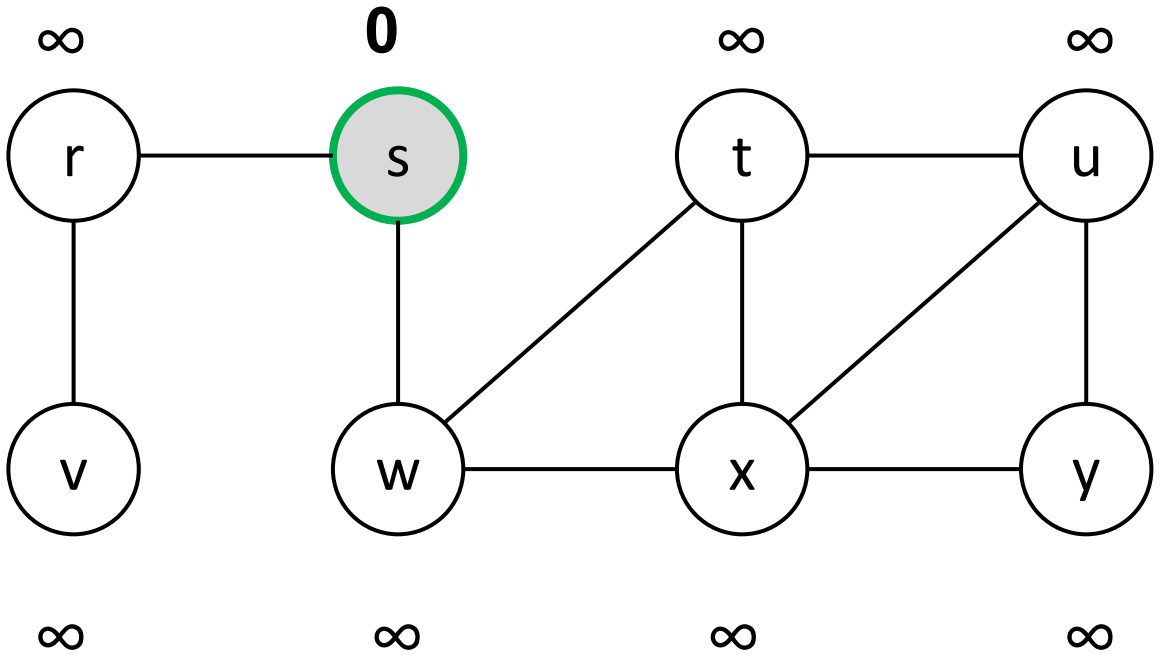
BFS(G, s)

After Initialization



All vertices are **white** with **d = ∞**

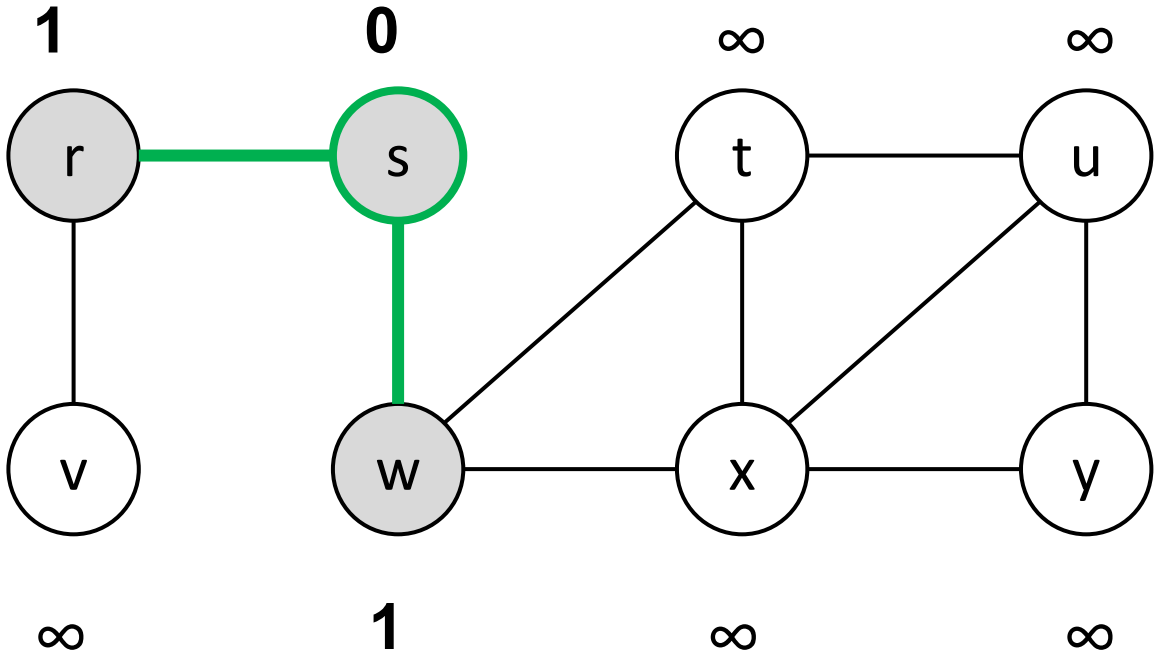
Start by “encountering” the source



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

Queue: s

Dequeue, explore neighbours

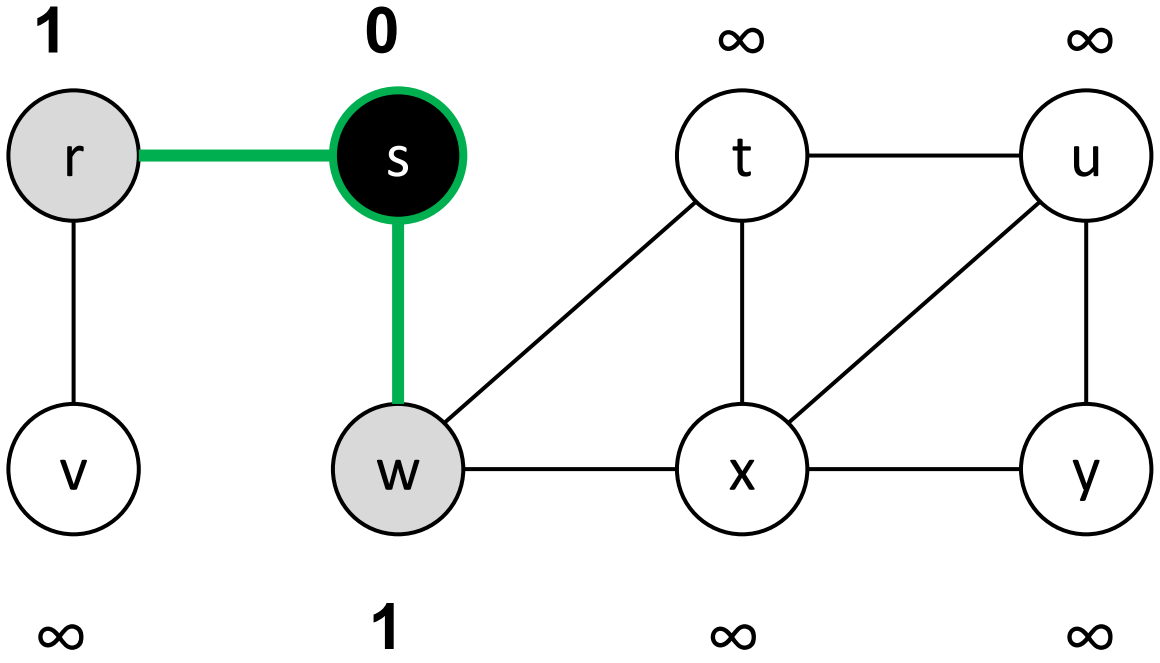


The green edge indicates the **pi[v]** that was stored

DQ

Queue: | s | r | w |

Colour black after exploring all neighbours

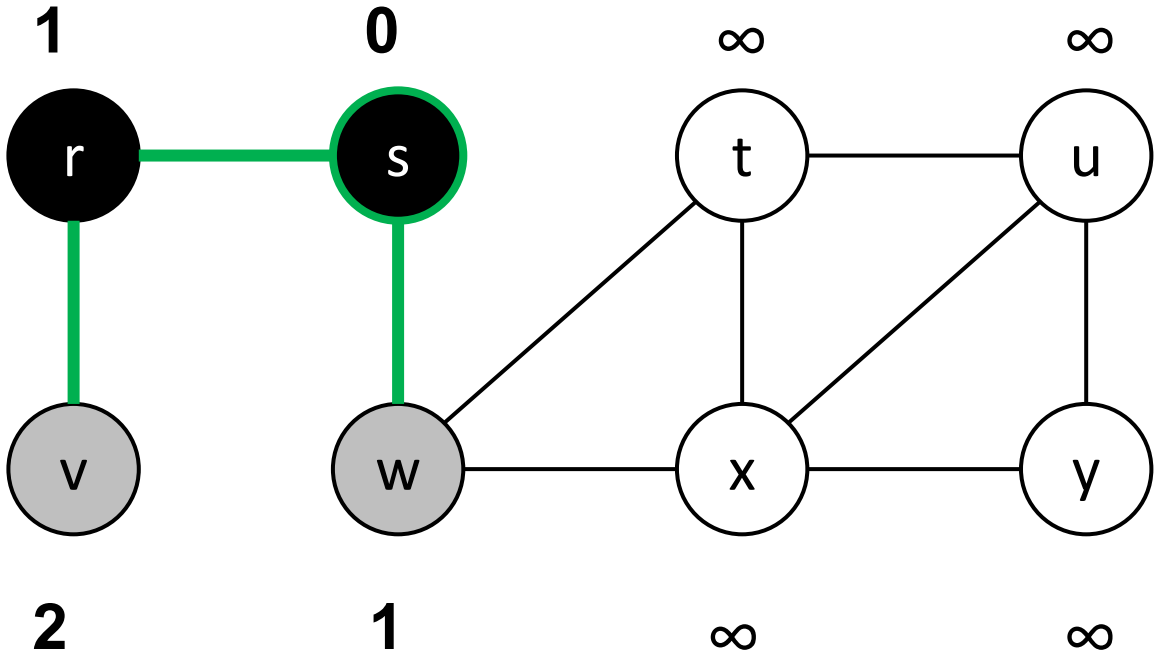


DQ

Queue:

s	r	w

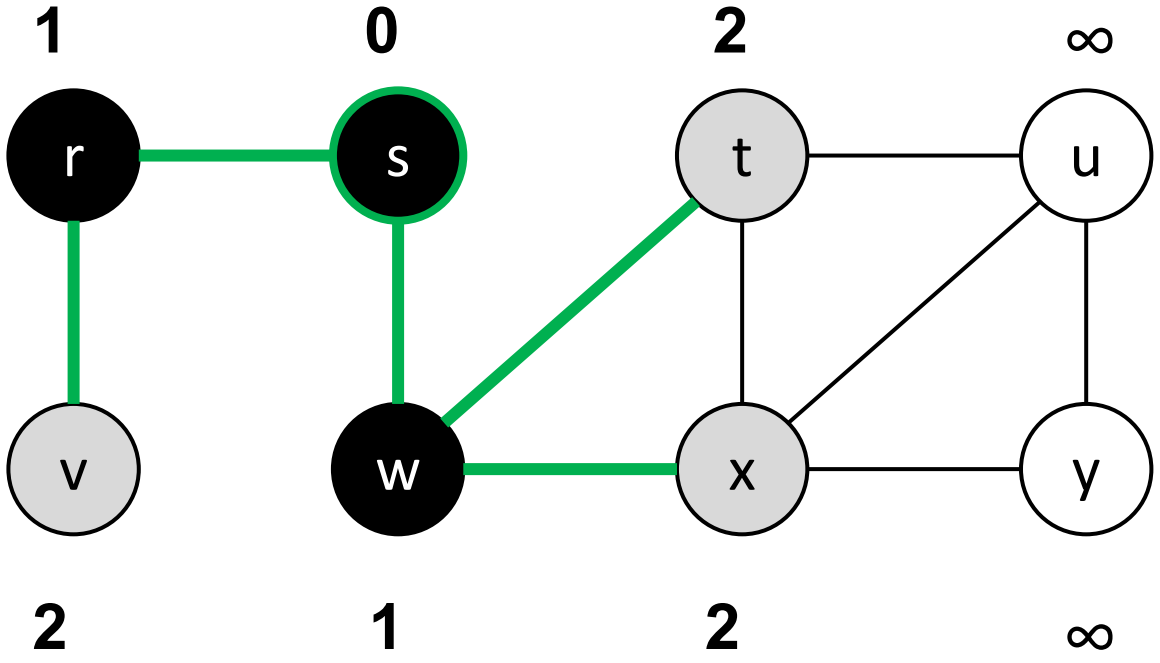
Dequeue, explore neighbours (2)



DQ DQ

Queue: | s | r | w | v |

Dequeue, explore neighbours (3)



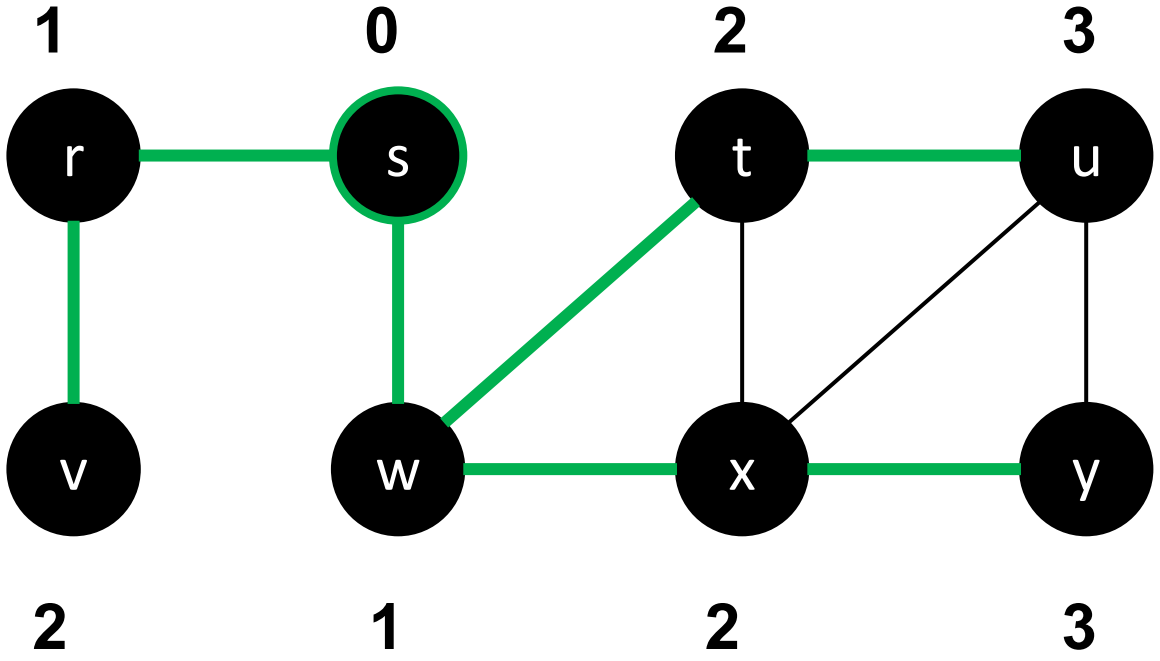
DQ DQ DQ

Queue:

s	r	w	v	t	x

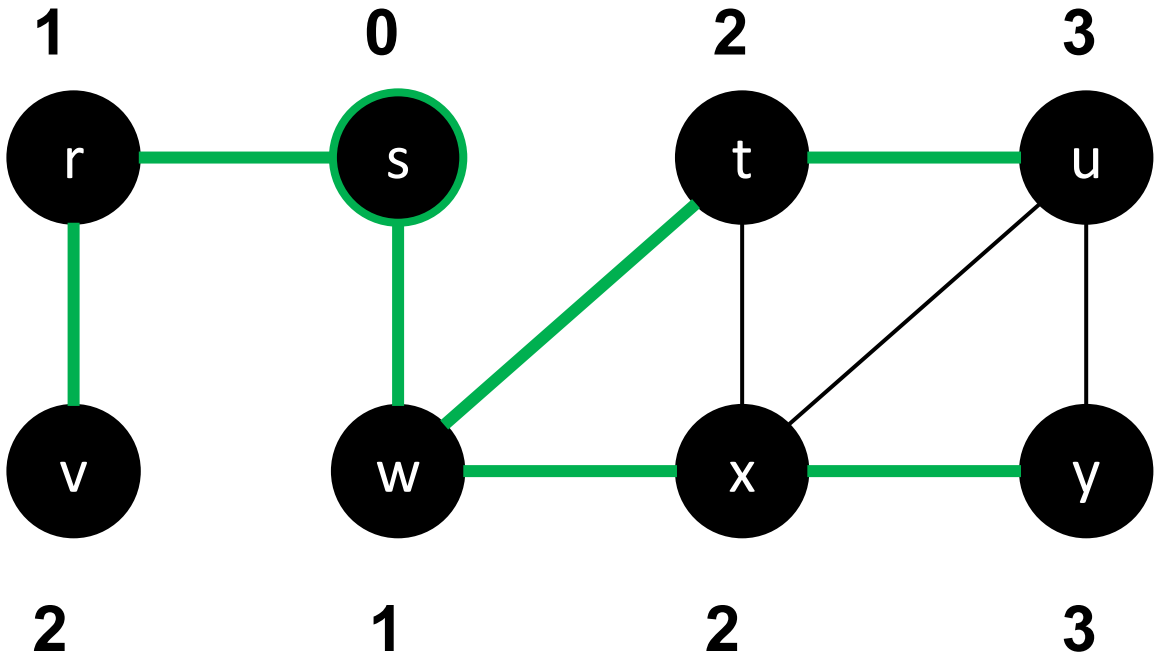
after a few more steps...

BFS complete

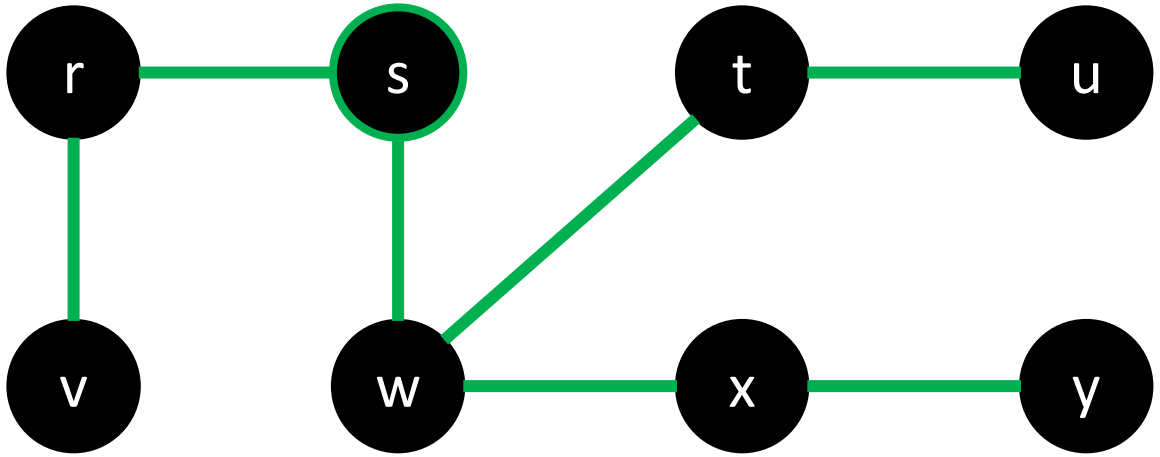


	DQ	DQ	DQ	DQ	DQ	DQ	DQ	DQ
Queue:	s	r	w	v	t	x	u	y

What do we get after doing BFS?



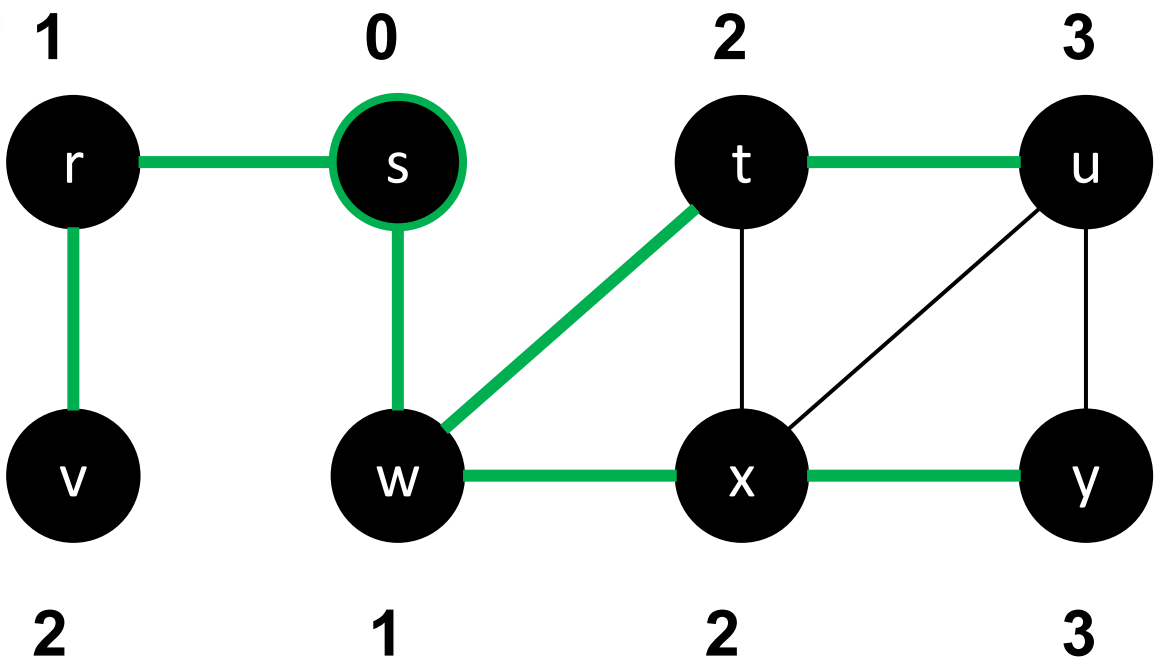
First of all, we get to visit **every** vertex exactly **once**.



Green edges give us a **BFS tree**.

A **tree** that connects all vertices, if the graph is **connected**.

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



The **d[v]** value indicates the vertex's **distance** from the source vertex.

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

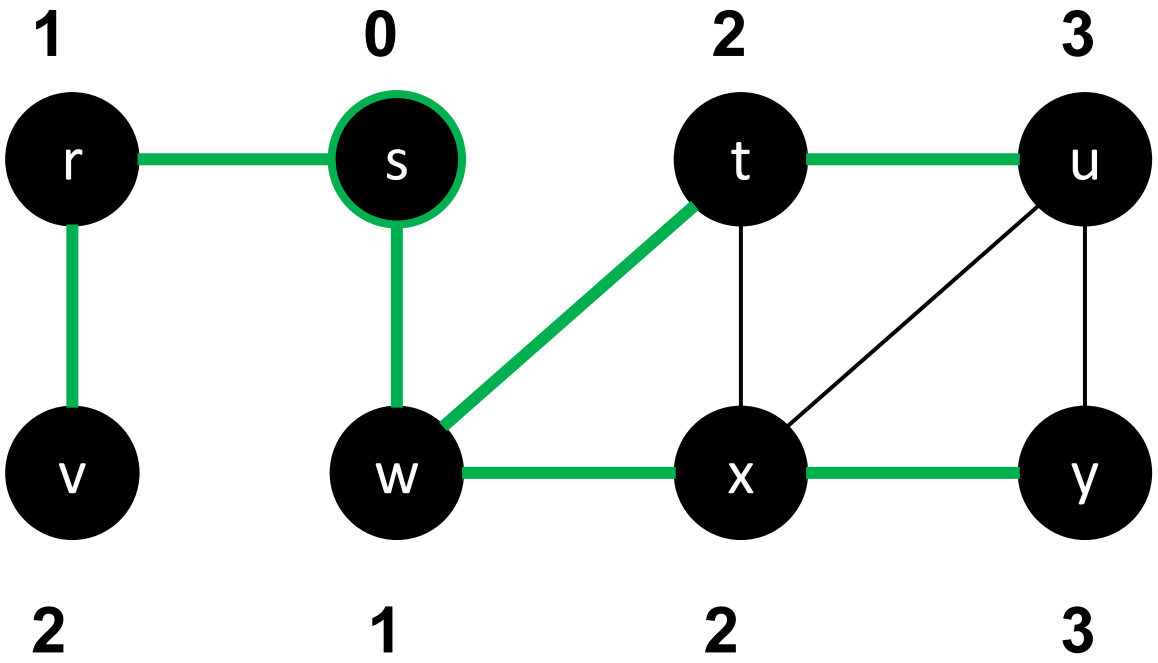
How about finding a **shortest path** itself? Follow the green edges, **pi[v]** comes in handy for this.

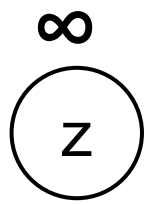
Shortest path from u to s:

u → pi[u] → pi[pi[u]] → pi[pi[pi[u]]] → … → s

What if G is disconnected?

Example Application: Garbage Collection

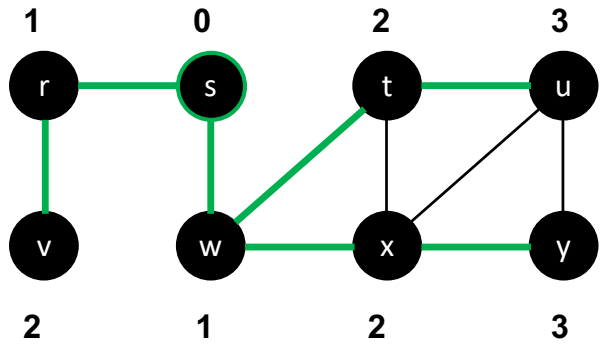




After BFS(s), z is of white colour and d[v] = ∞

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

Runtime Analysis



Using **adjacency list**

- Visit each vertex once
  - Enqueue, Dequeue, change colours, assign d[v], …, constant work per vertex
  - Total: **O(|V|)**
- For each vertex, check all its neighbours
  - Each edge is checked **once** (directed graph) or **twice** (undirected graph)
  - Total: **O(|E|)**

Total runtime: O(|V|+|E|)

Summary of BFS

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

DFS

Next week