



Week 8: Graphs/networks I

Algorithmic Data Science

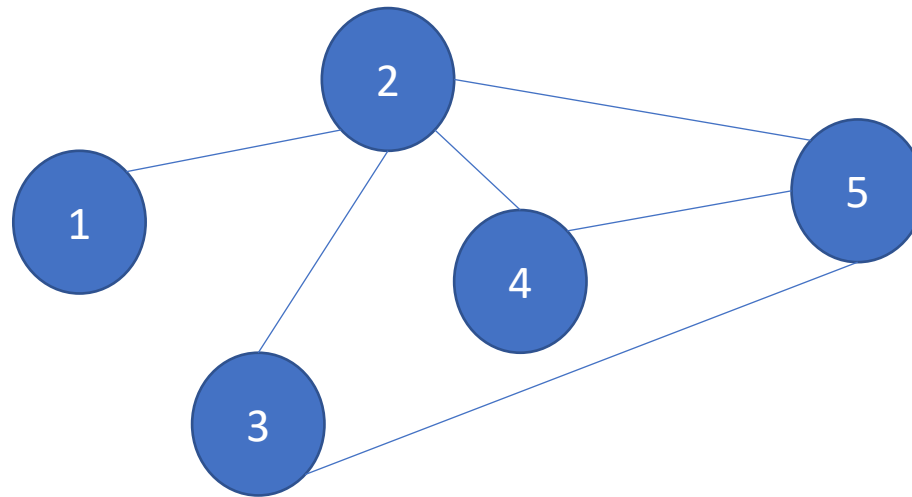
2021-22

Overview

- We have considered finding similarity for sets of objects
- The next topic for this week and next week is:
 - **Graphs / networks**
 - **Breadth-first and depth-first searching**
 - Dijkstra's algorithm for all pairs shortest paths
 - Mining social media networks
 - Clustering

Graphs / Networks

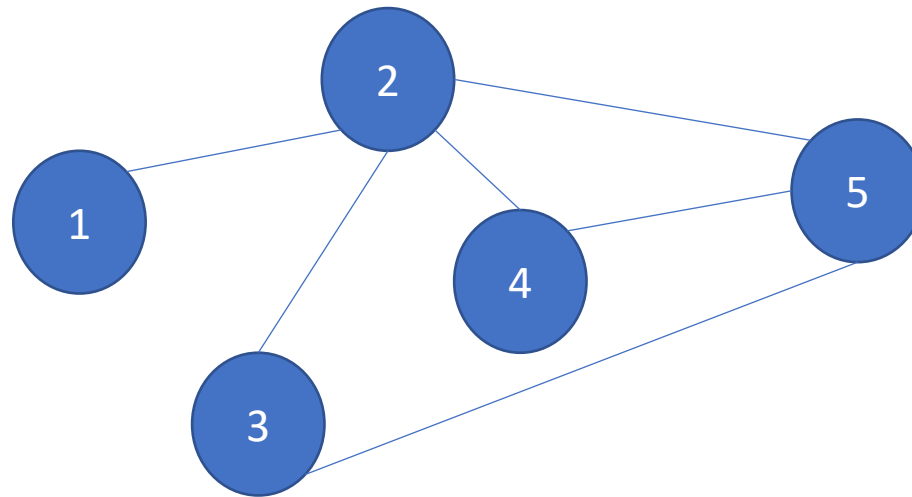
- Informally, a graph is a set of objects and a set of connections between them



- More formally,
 - the objects are referred to as **vertices** (or nodes).
 - the connections are referred to as **edges** (or arcs).
 - A graph G is the pair (V,E) where V is a finite set of vertices and E is a binary relation on V .

Undirected graphs

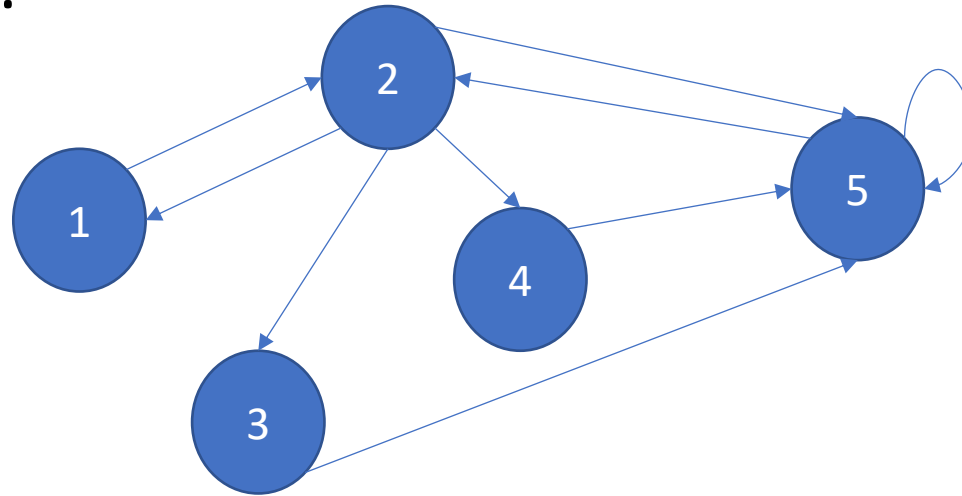
- If the relationship between the objects is symmetric then the graph is **undirected**.



- In this example, the vertices might represent people on a social networking site such as Facebook
- The edges represent friendships between them

Directed graphs

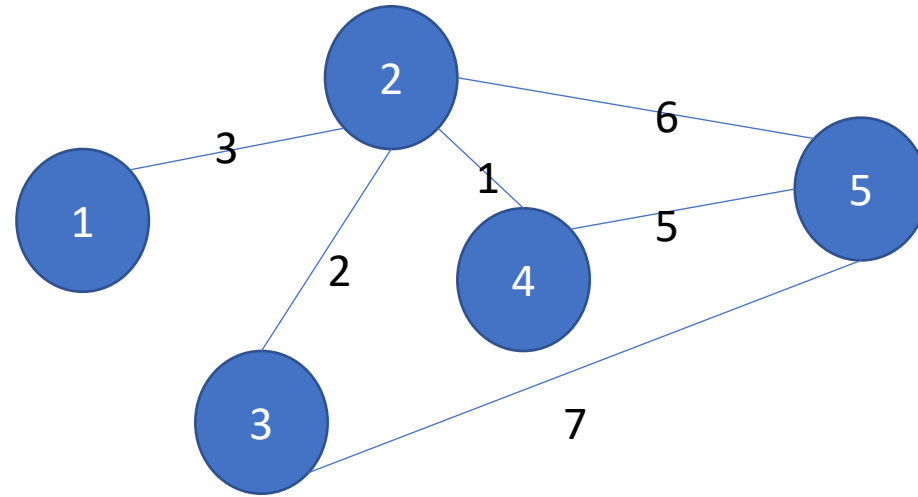
- If the relationship between the objects is asymmetric then the graph is **directed**.



- In this example, the vertices might represent pages on the Internet.
- The edges might represent links between them

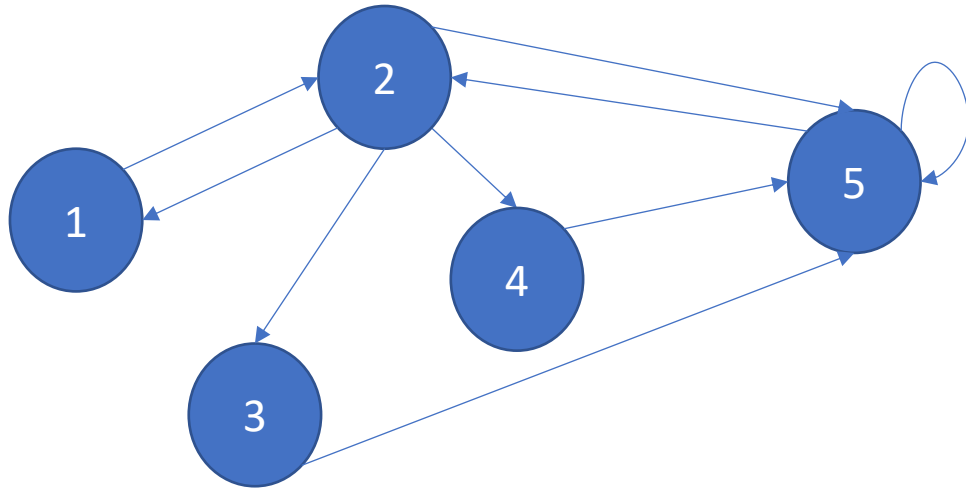
Weighted graphs

- Both directed and undirected graphs can be weighted i.e., have a weight associated with each of the edges.



- In this example, the vertices might represent physical locations
- The edges represent roads between the locations.
- The weights represent the distance or cost of travelling along those roads

More terminology

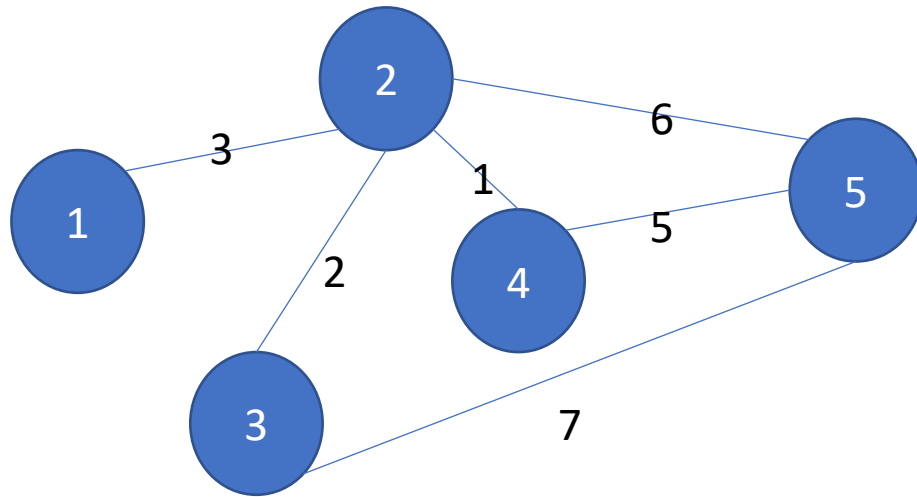


- If (u,v) is an edge in the graph, we say v is **adjacent** to u .
- The **out-degree** of a vertex is the number of edges leaving it
- The **in-degree** of a vertex is the number of edges entering it

- A **path** from a vertex v_0 to a vertex v_k is a sequence of vertices $\langle v_0, v_1, \dots, v_k \rangle$ such that $(v_{i-1}, v_i) \in E$ for $i=1, 2, \dots, k$
- The **length** of a path is the number of edges in the path.
- If there is a path from u to v then we say that v is **reachable** from u
- A **cycle** is a path from a vertex to itself.
- An undirected graph is **connected** if every pair of vertices is connected by a path.
- A directed graph is **strongly connected** if every two vertices are reachable from each other

Is the graph shown strongly connected?

Representing graphs



Adjacency list representation:

An array, list or dictionary of lists which record, for each vertex, its adjacent vertices (with weights)

{1:[(2,3)],
2: [(1,3),(3,2),(5,6),(4,1)],
3:[(2,2),(5,7)],
4:[(5,5),(2,1)],
5:[(3,7),(2,6),(4,5)]

memory requirements = $O(V+E)$

Adjacency (or characteristic) matrix representation:

A matrix where each rows and columns refers to vertices and the value of m_{ij} is the weight on the edge from vertex i to vertex j (0 or null if no edge)

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

memory requirements = $O(V^2)$

Which representation is better?

Searching graphs

Example: A web crawler starts at a given web page, scrapes all of the text from the page, identifies all of the hyperlinks and then visits all of the pages linked to by that page and repeats the process. It stops when it has visited all of pages within a given distance, i.e., number of links, of the original page. How can we be sure that the crawler will visit all of the required pages and terminate?

This is a graph problem. Vertices are webpages. Edges are hyperlinks. Edges are directed but unweighted.

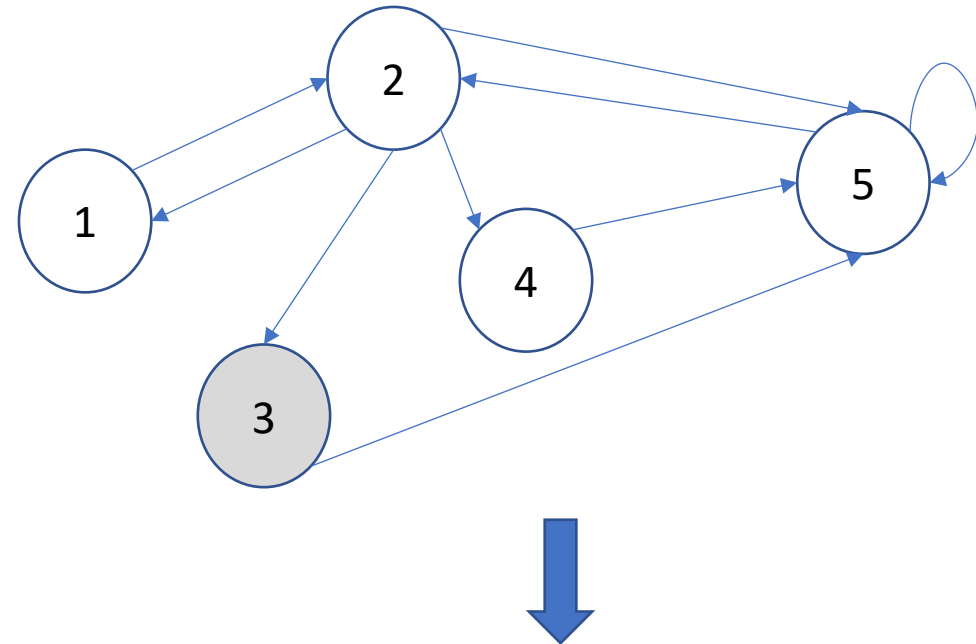
Breadth-first search

- Given a graph $G = (V, E)$ and a **source** vertex s , breadth-first search systematically explores the edges of G to discover every vertex that is reachable from s
- It computes the distance (fewest number of edges) from s to all such reachable vertices
- It produces a **breadth-first tree** with root s that contains all such reachable vertices
- It is so-named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

Breadth-first search: initialisation

```
BFS( $G=(V,E),s$ ):  
  for each vertex  $u \in V - \{s\}$ :  
    colour[u]=white  
     $d[u] = \infty$   
     $\pi[u] = \text{Null}$   
  colour[s] = grey  
   $d[s] = 0$   
   $\pi[s] = \text{Null}$   
   $Q=[s]$ 
```

$Q=[3]$



Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

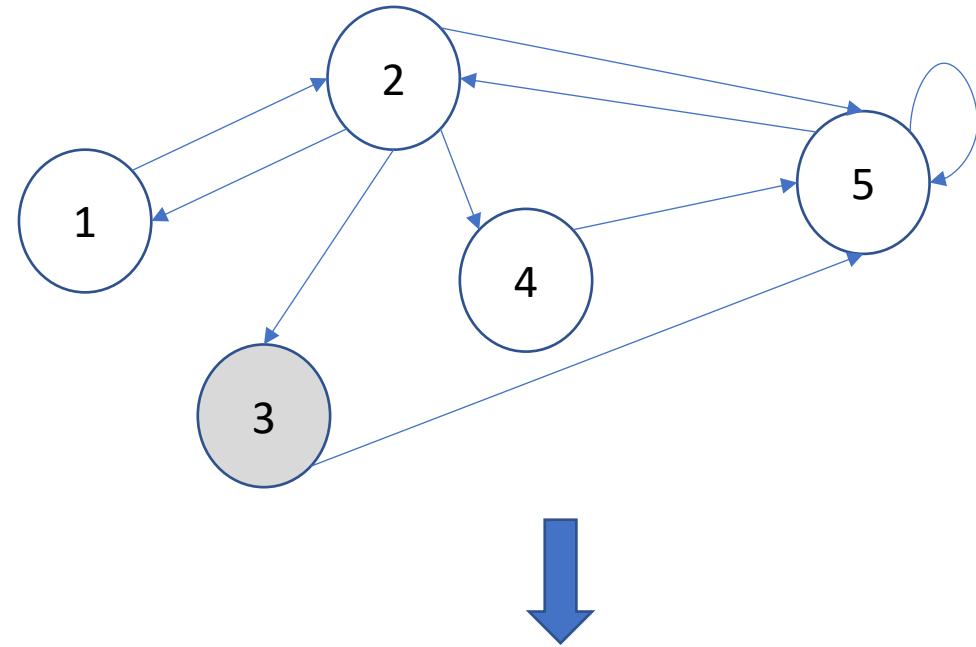
colour=
{1:white,
2:white,
3:grey,
4:white,
5:white}

d =
{1: ∞ ,
2: ∞ ,
3:0,
4: ∞ ,
5: ∞ }

π =
{1:Null,
2:Null,
3:Null,
4:Null,
5:Null}

Breadth-first search: iteration

```
while notempty(Q):  
    u = head(Q)  
    for each v in Adj(u):  
        if colour[v] == white:  
            colour[v] = grey  
            d[v]=d[u]+1  
             $\pi[v]=u$   
            Q.append(v)  
    Q.remove(u)  
    colour[u]=black
```



Q=[3]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

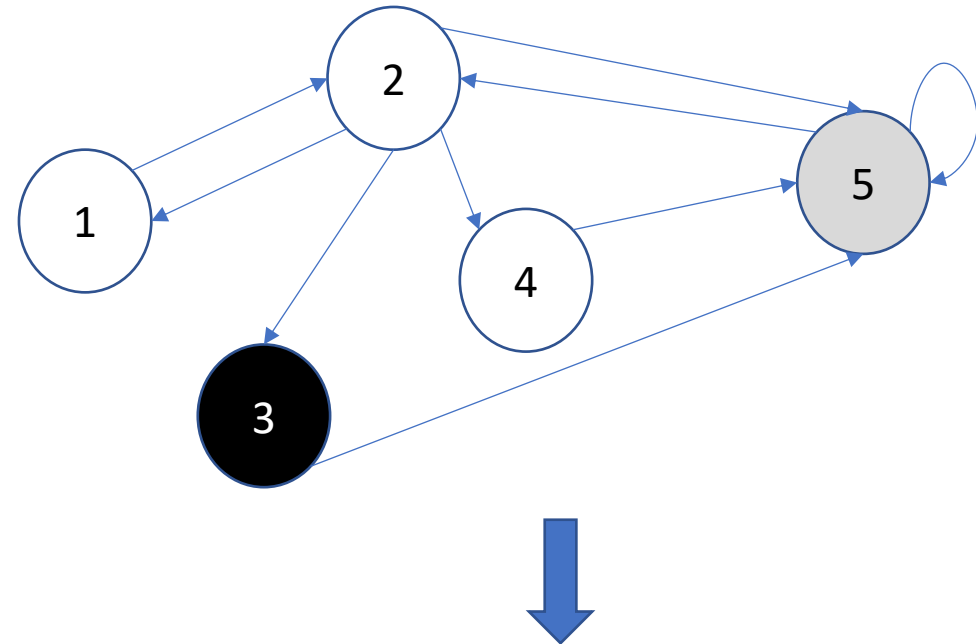
colour=
{1:white,
2:white,
3:grey,
4:white,
5:white}

d =
{1: ∞ ,
2: ∞ ,
3:0,
4: ∞ ,
5: ∞ }

π =
{1:Null,
2:Null,
3:Null,
4:Null,
5:Null}

Breadth-first search: iteration

```
while notempty(Q):  
    u = head(Q)  
    for each v in Adj(u):  
        if colour[v] == white:  
            colour[v] = grey  
            d[v]=d[u]+1  
             $\pi[v]=u$   
            Q.append(v)  
    Q.remove(u)  
    colour[u]=black
```



Q=[5]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

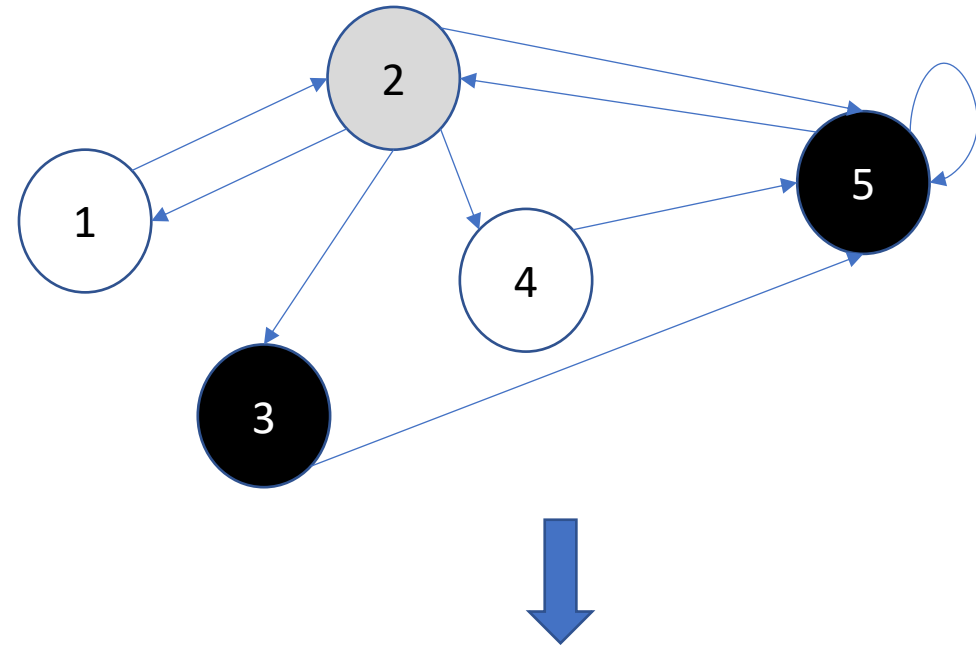
colour=
{1:white,
2:white,
3:black,
4:white,
5:grey}

d =
{1: ∞ ,
2: ∞ ,
3:0,
4: ∞ ,
5:1}

π =
{1:Null,
2:Null,
3:Null,
4:Null,
5:3}

Breadth-first search: iteration

```
while notempty(Q):  
    u = head(Q)  
    for each v in Adj(u):  
        if colour[v] == white:  
            colour[v] = grey  
            d[v]=d[u]+1  
             $\pi[v]=u$   
            Q.append(v)  
    Q.remove(u)  
    colour[u]=black
```



Q=[2]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

colour=
{1:white,
2:white,
3:black,
4:white,
5:black}

d =
{1: ∞ ,
2:2,
3:0,
4: ∞ ,
5:1}

π =
{1:Null,
2:5,
3:Null,
4:Null,
5:3}

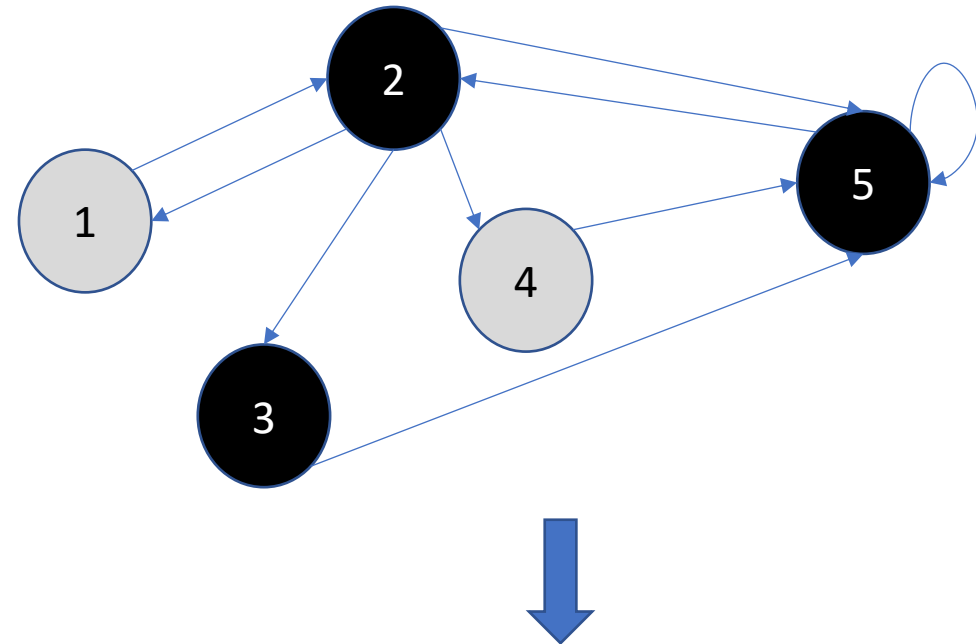
Breadth-first search: iteration

```
while notempty(Q):  
    u = head(Q)  
    for each v in Adj(u):  
        if colour[v] == white:  
            colour[v] = grey  
            d[v]=d[u]+1  
             $\pi[v]=u$   
            Q.append(v)  
    Q.remove(u)  
    colour[u]=black
```

Q=[1,4]

In our example we might want to check $d[v]$ before appending v to Q i.e.,:

if $d[v] < \text{threshold}$: Q.append(v)



Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

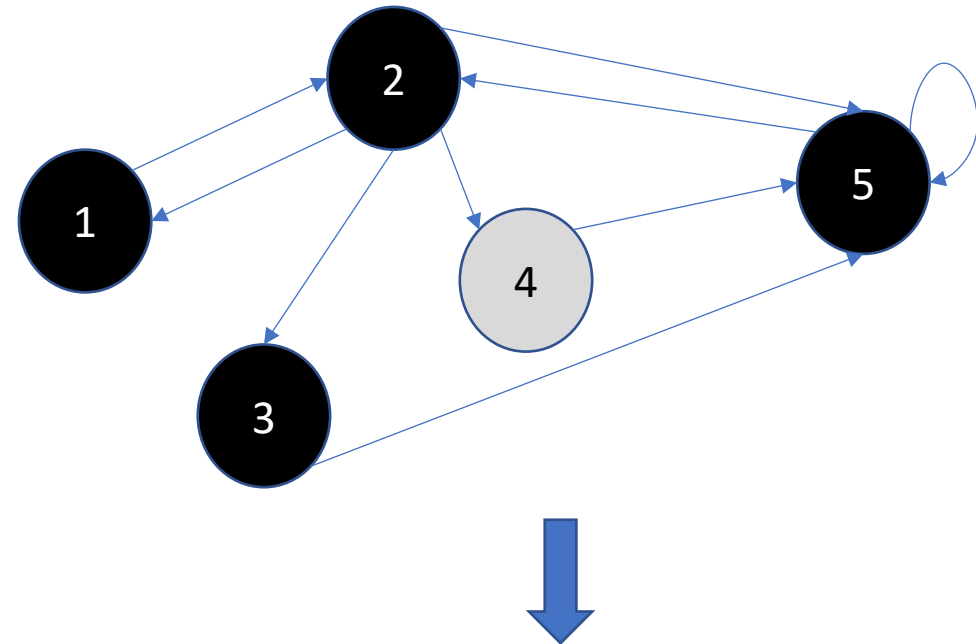
colour=
{1:grey,
2:black,
3:black,
4:grey,
5:black}

d =
{1:3,
2:2,
3:0,
4:3,
5:1}

π =
{1:2,
2:5,
3:Null,
4:2,
5:3}

Breadth-first search: iteration

```
while notempty(Q):  
    u = head(Q)  
    for each v in Adj(u):  
        if colour[v] == white:  
            colour[v] = grey  
            d[v]=d[u]+1  
             $\pi[v]=u$   
            Q.append(v)  
    Q.remove(u)  
    colour[u]=black
```



Q=[4]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

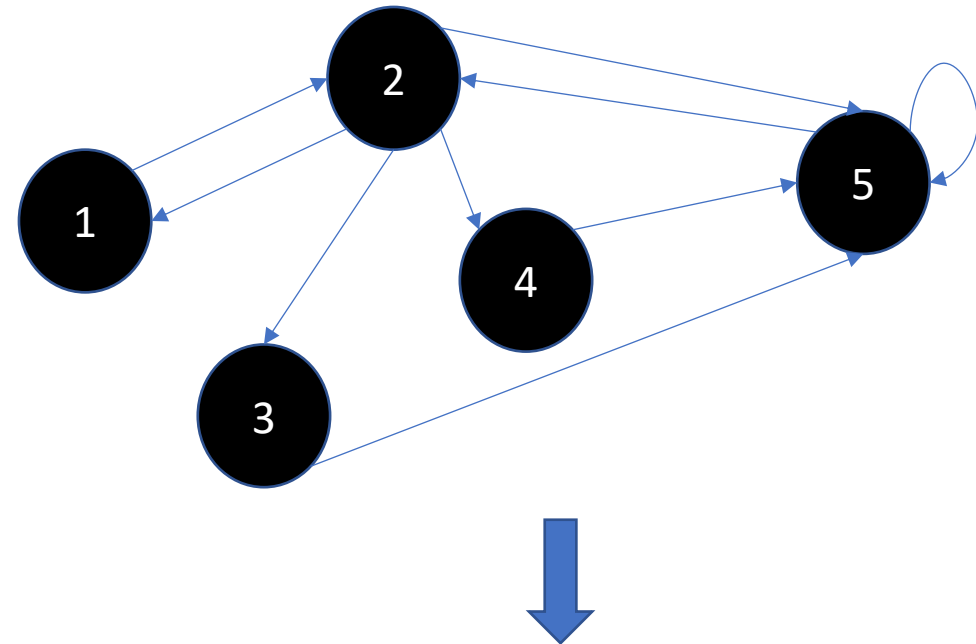
colour=
{1:black,
2:black,
3:black,
4:grey,
5:black}

d =
{1:3,
2:2,
3:0,
4:3,
5:1}

π =
{1:2,
2:5,
3:Null,
4:2,
5:3}

Breadth-first search: iteration

```
while notempty(Q):  
    u = head(Q)  
    for each v in Adj(u):  
        if colour[v] == white:  
            colour[v] = grey  
            d[v]=d[u]+1  
             $\pi[v]=u$   
            Q.append(v)  
    Q.remove(u)  
    colour[u]=black
```



Q=[4]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

colour=
{1:black,
2:black,
3:black,
4:black,
5:black}

d =
{1:3,
2:2,
3:0,
4:3,
5:1}

π =
{1:2,
2:5,
3:Null,
4:2,
5:3}

Breadth-first search: output

- d contains the shortest distance of each vertex from s
- π contains the predecessor of v on the shortest path from s to v
- It can be used to reconstruct the shortest paths / the breadth-first search tree if required
- Other functions (such as scraping the webpage and identifying links) can be called at the time that a vertex is discovered (i.e., it is coloured grey).

Breadth-first search: analysis

- Initialisation is $O(V)$
- The iteration happens at most V times
- In each iteration, the adjacency list for that vertex is considered
- Each edge is considered at most once throughout the whole iterative procedure
- So iteration is $O(E)$
- running time of $\text{BFS}(G,s) = O(V+E)$

Depth-first search

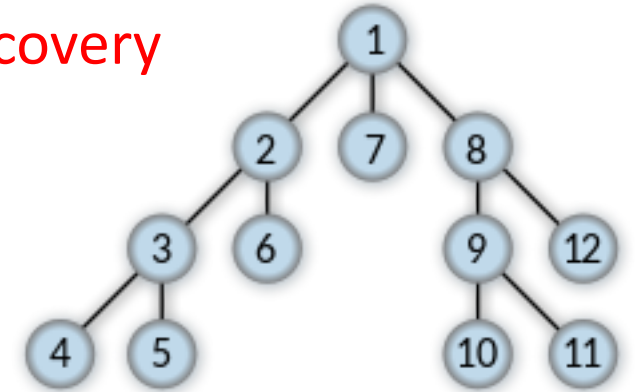
- Alternative to breadth-first search
- Searches *deeper* in the graph whenever possible
- This can be done using a recursive call to itself

```
DFS-visit(u):  
  colour[u] = grey  
  for each v in Adj[u]:  
    if colour[v] == white:  
      d[v]=d[u]+1  
       $\pi[v]=u$   
      DFS-visit[v]  
  colour[u] = black
```

- Running time = $O(V+E)$

This recursive call means that edges from v will be explored before the remainder of the edges of u are explored.

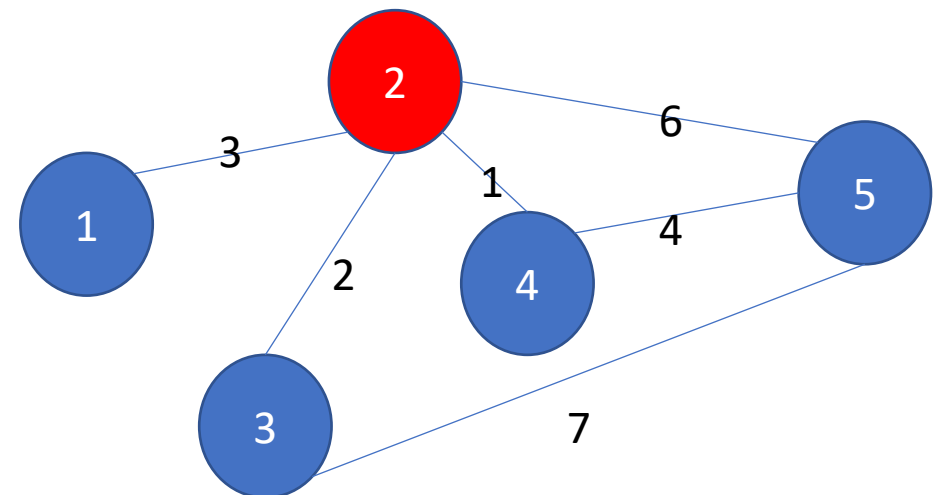
Example order of discovery



Wikipedia

Dijkstra's algorithm

- Solves the shortest paths problem in a weighted graph
- Modification of BFS
- Visit vertices in order of increasing distance from the source.
- As each vertex u is visited, check to see if the distance of each vertex v which is adjacent to u would be shorter if the path went via u
- We will explore this in the lab.



Overview

- We have considered finding similarity for sets of objects
- Current topic:
 - **Graphs / networks**
 - Breadth-first and depth-first searching **DONE**
 - Dijkstra's algorithm for all pairs shortest paths **INTRODUCED- EXAMPLE NEXT WEEK**
 - Mining social media networks **NEXT WEEK**
 - Clustering **NEXT WEEK**
- Also next week: PageRank algorithm

