Coventry University

Graphs

Dr. Diana Hintea

Lecturer in Computer Science

diana.hintea@coventry.ac.uk

210CT Week 7

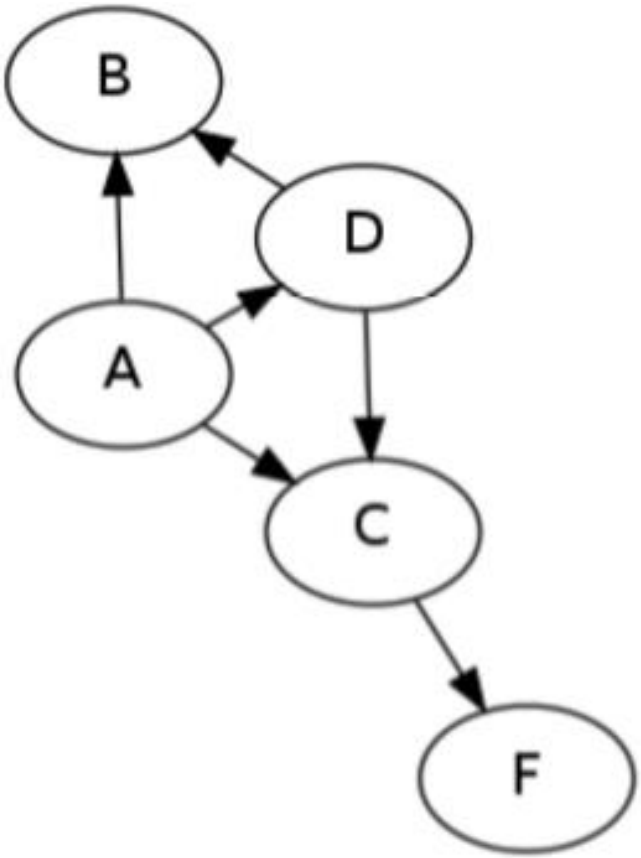
Learning Outcomes

1. Graphs Intro
2. Representation and Implementation
3. Graph Traversal (DFS)
4. Graph Traversal (BFS)
5. Dijkstra's Algorithm

Graphs

What are graphs?

- Set of nodes and connections between them.



Graphs

What are they used for?

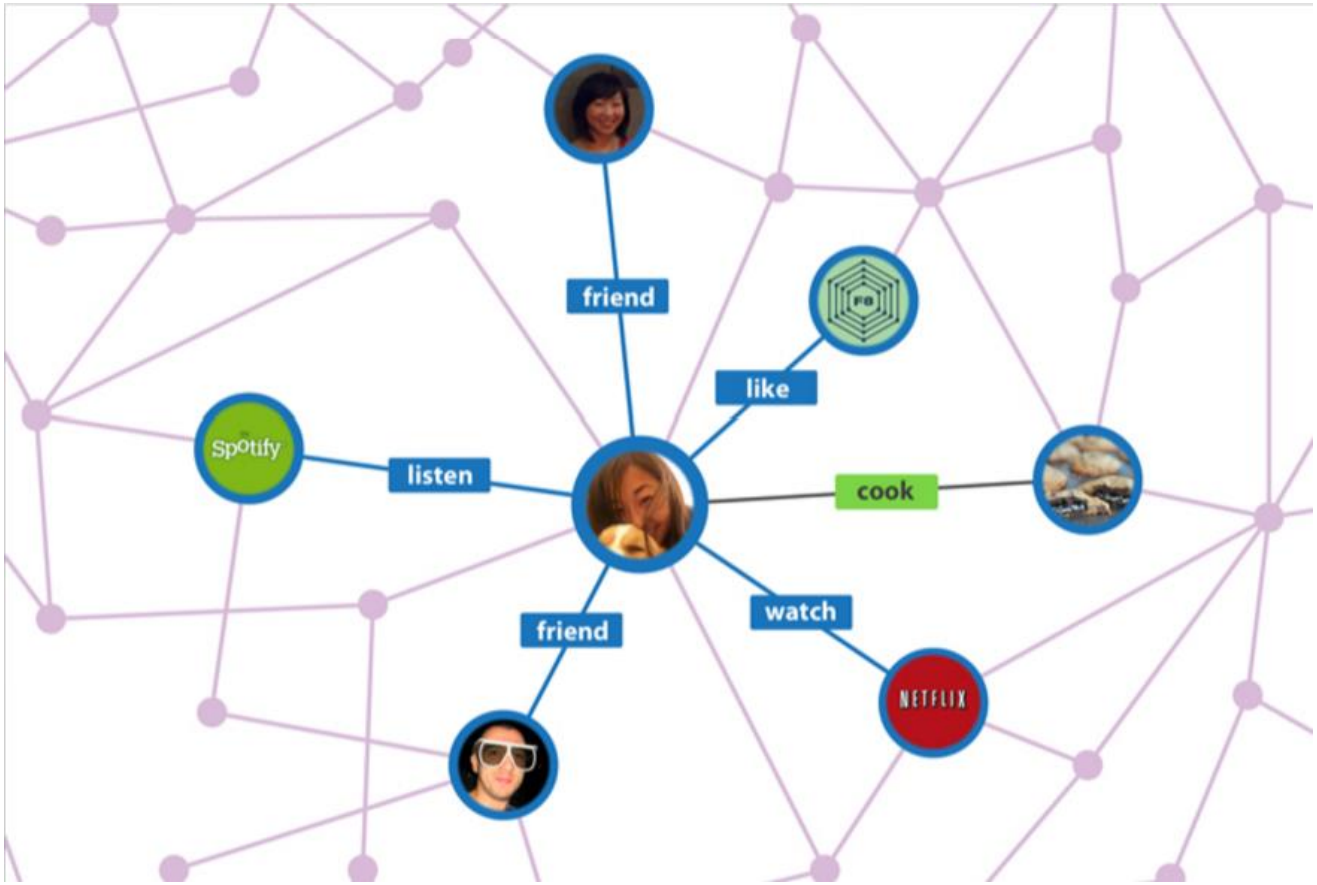
Graphs

What are they used for?

- Map analysis, route finding, path planning
- Ranking search results
- Analysing related data, such as social networks, proteins, etc.
- Compiler optimisation
- Constraint satisfaction (timetabling)
- Physics simulations (actual physics)
- Physics simulations (games)
- Social connections: Facebook uses graphs for this

Graphs

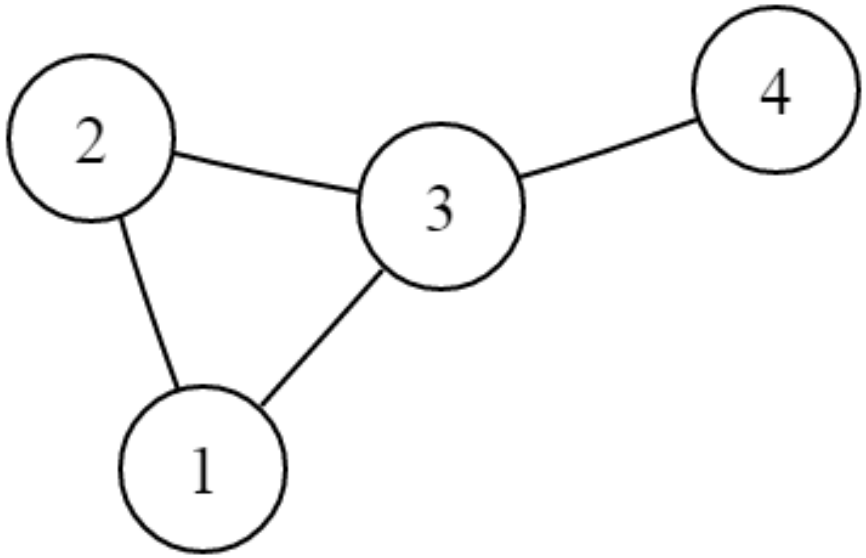
Facebook!



Graphs

Types of graphs

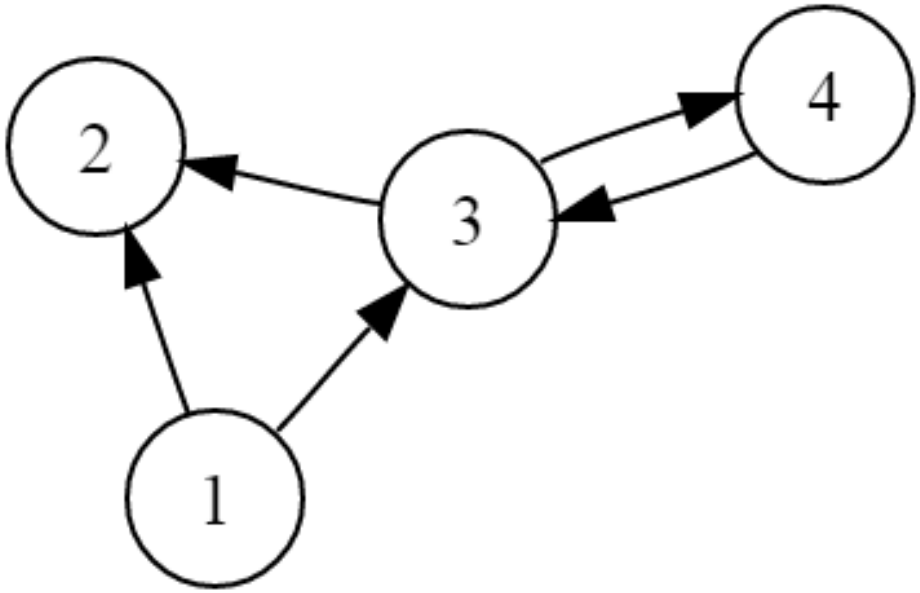
- **Undirected graph**: A link is both ways.



Graphs

Types of graphs

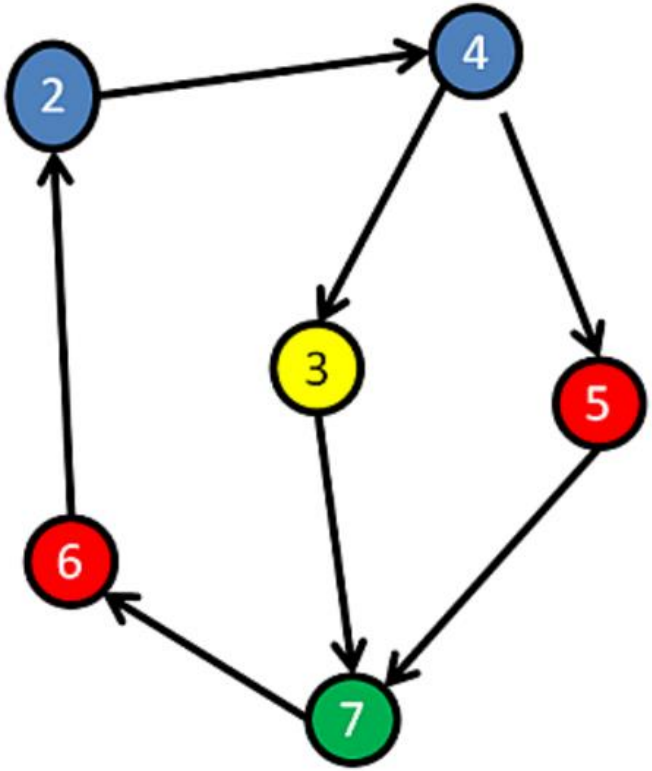
- **Directed graph**: Each link is directional and does not imply the inverse.



Graphs

Types of graphs

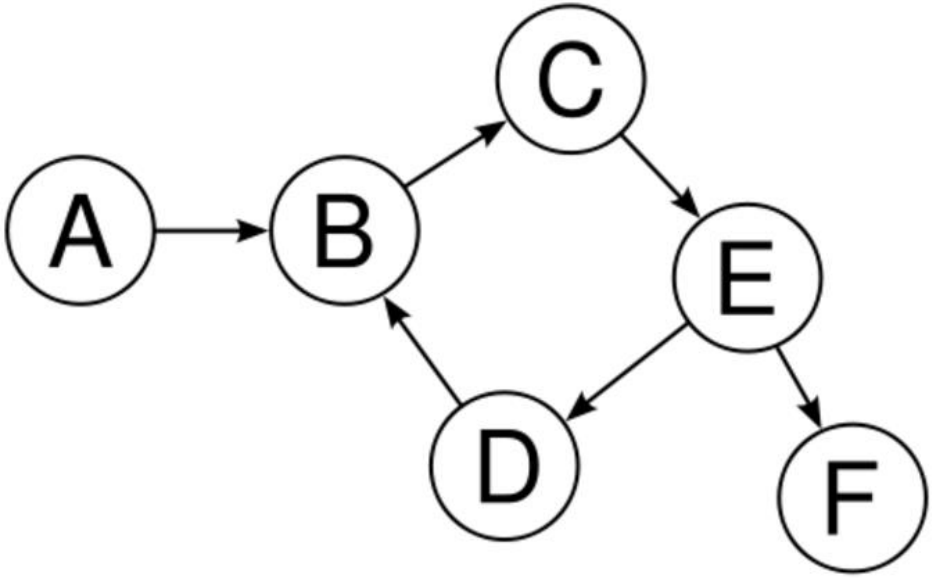
- Vertex labelled graph: The data used to identify each node is not the only data that is important about that node.
  - For example, it might also have a "colour" value that affects algorithms.



Graphs

Types of graphs

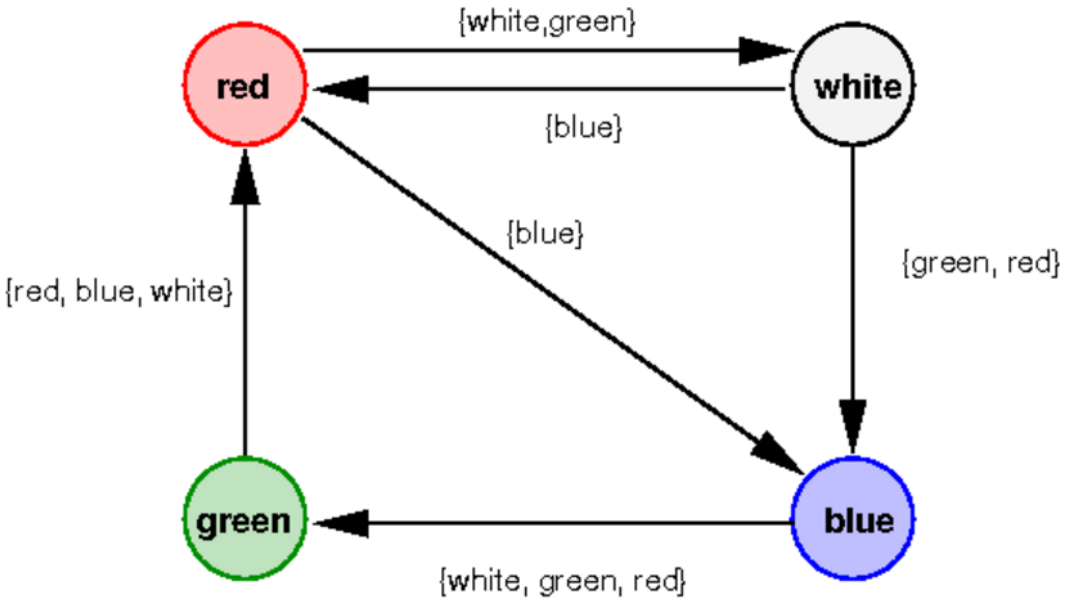
- Cyclic graphs: Has at least one "cycle". That is, a path from a node that leads back to itself.



Graphs

Types of graphs

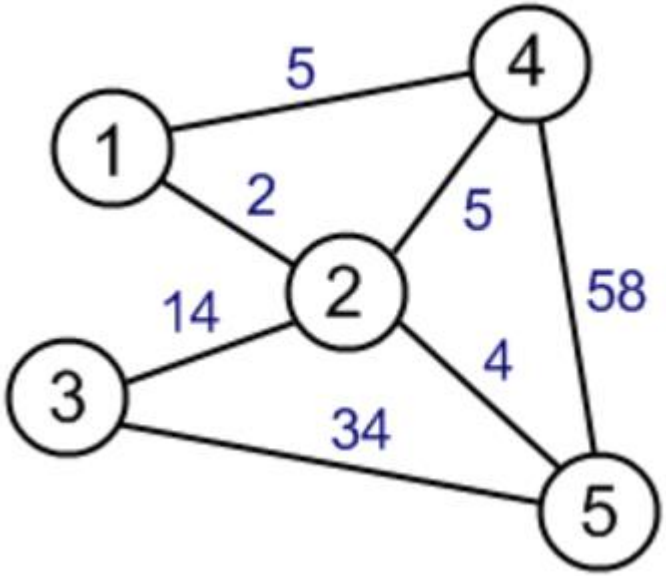
- Edge labelled graphs:
  - Links have data values that might affect algorithms.
  - Imagine in a graph of cities in which the links represent roads, the length of the road might be recorded as the edge data.



Graphs

Types of graph

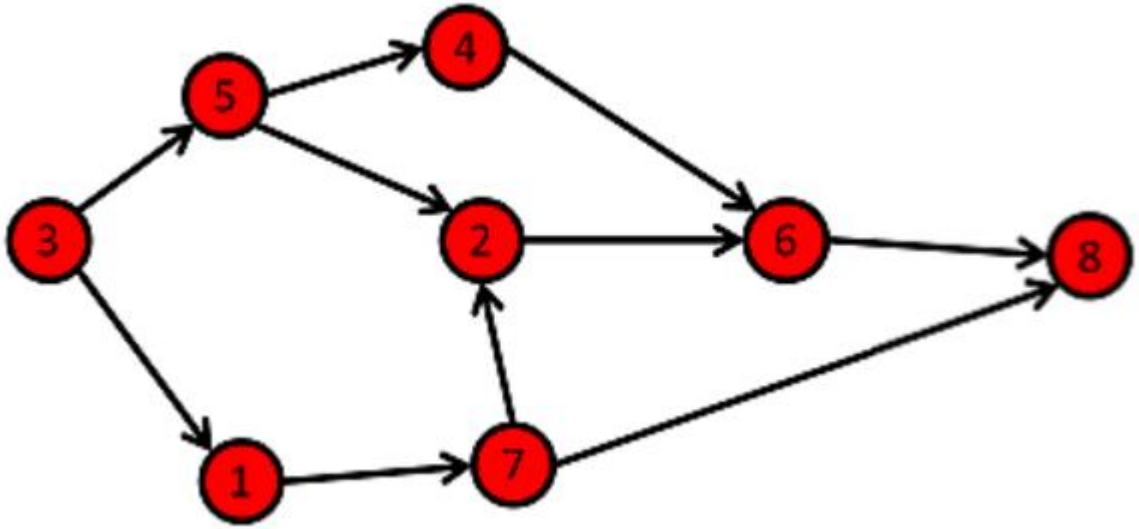
- **Weighted graph**: The parameters along edges or at nodes are interval data and can be summed and compared.



Graphs

Types of graphs

- Directed acyclic graph: Links have direction.
  - No cycles exist.
  - Used a lot. Called a "DAG" for short. Possibly humorous to Australians.

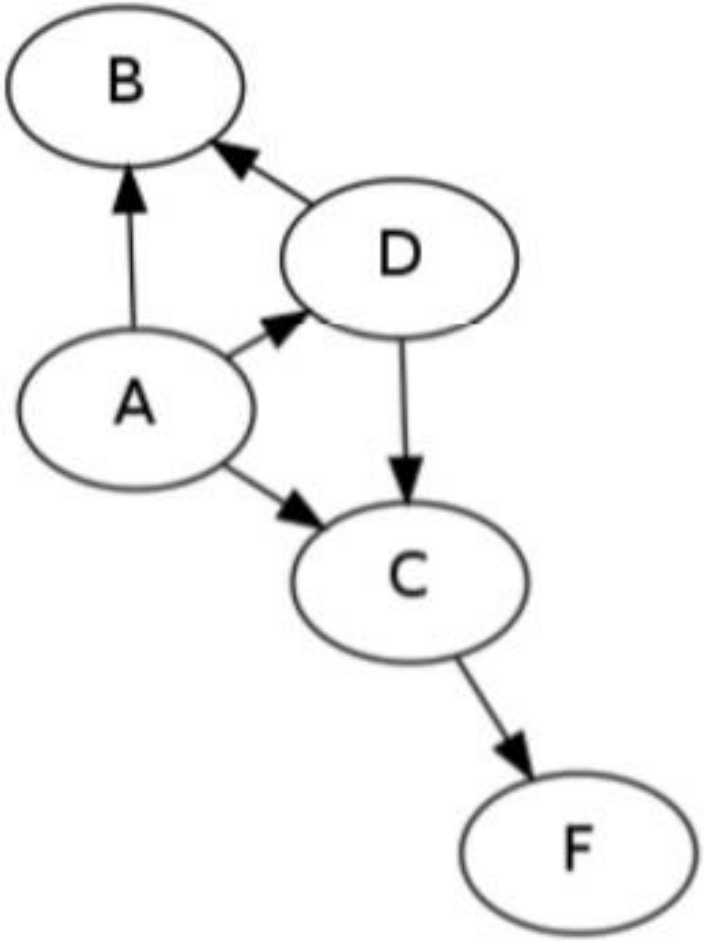


Graphs

Terminology

- The nodes are called **vertices**.
- The links are called **edges** (sometimes **arcs**).
- An edge is **incident** to/on a vertex if it connects it to another.
- Connected vertices are called **adjacent** or **neighbours**.
- A vertex's **degree** is the number of edges incident on it.

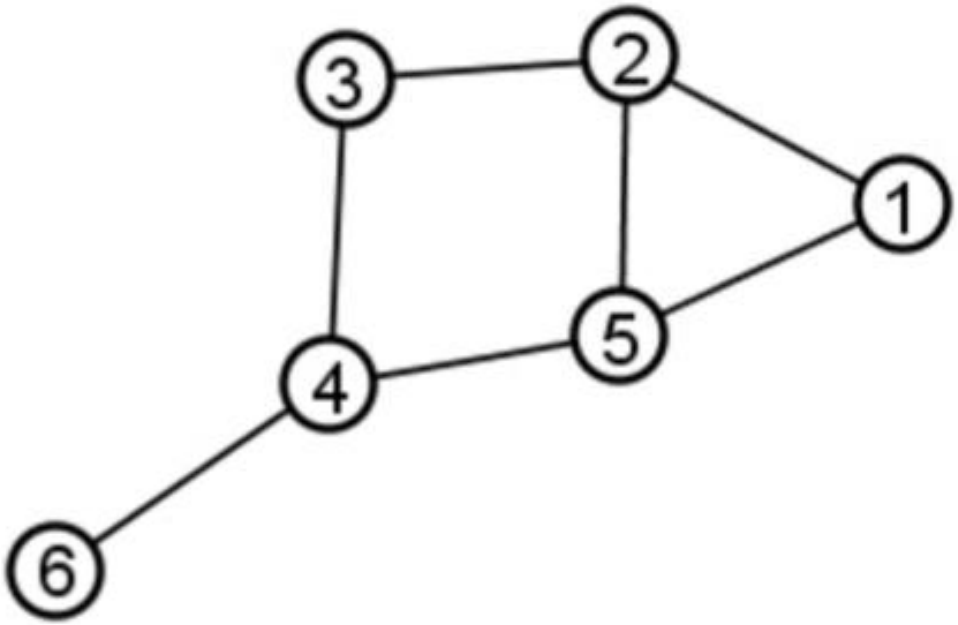
Graphs



Graphs

Vertex Degree (Undirected Graph)

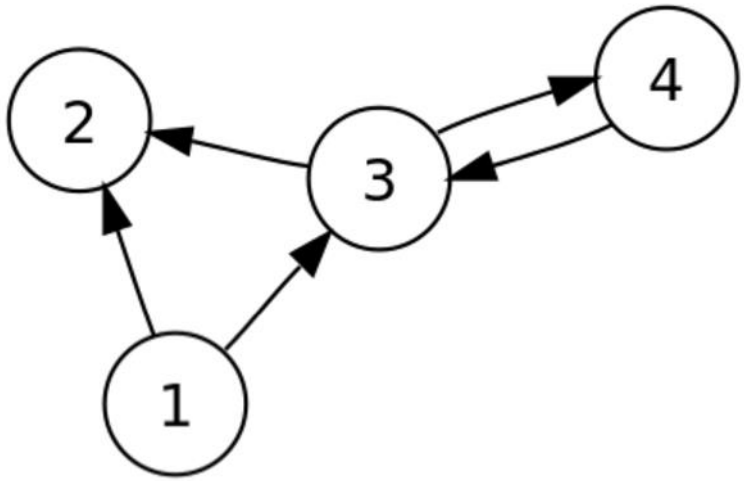
Vertex	Degree
1	2
2	3
3	2
4	3
5	3
6	1



Graphs

Vertex Degree (Directed Graph)

Vertex	Degree	Indegree	Outdegree
1	2	0	2
2	2	2	0
3	3	2	2
4	1	1	1



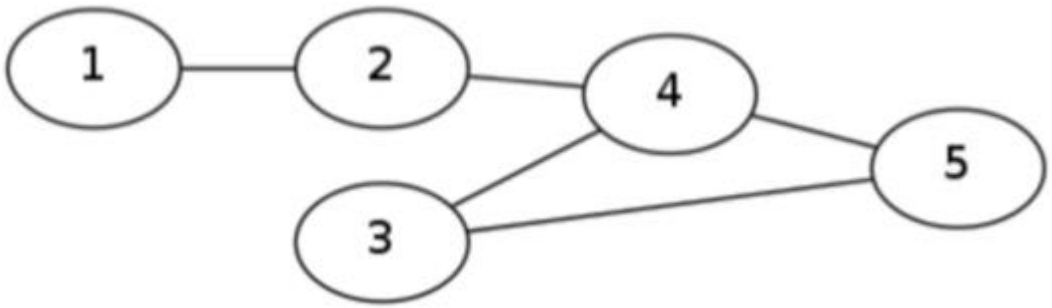
Representing graphs

- We can represent vertices with integers (actually, any unique value, would do for most of the times).
- Edges are pairs of vertices, (a,b), connecting a and b.
- A graph, G, has a set of vertices, V, and a set of edges, E.
- We say G=(V,E).
- We use n and m to represent the numbers of vertices and edges.
- Think n for node if you find it hard to remember which way around these are.

Representation and Implementation

Visualisation

- A graph G=(V,E)
- Where:
  - V={1,2,3,4,5}
  - E={{1,2},{2,4},{3,4},{3,5},{4,5}}

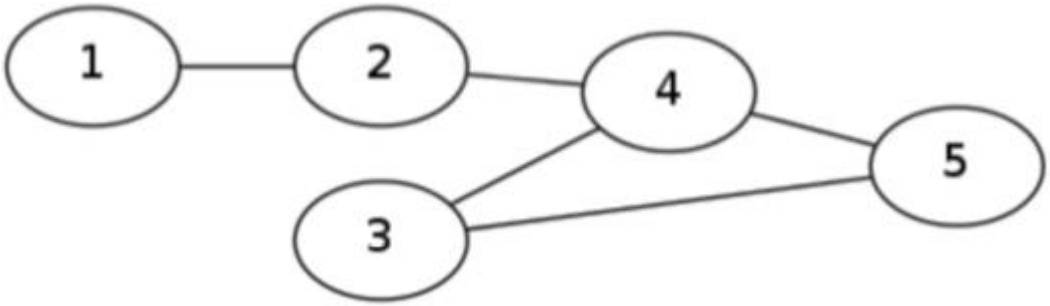


Implementing Graphs

- There are two ways of implementing a graph when programming.
- Adjacency Matrix:
  - A two dimensional matrix of boolean values where the value of a cell i, j is true if vertices i and j are connected.
- Adjacency List:
  - Each vertex contains a list of vertices that it is connected to.

Representation and Implementation

Adjacency Matrix

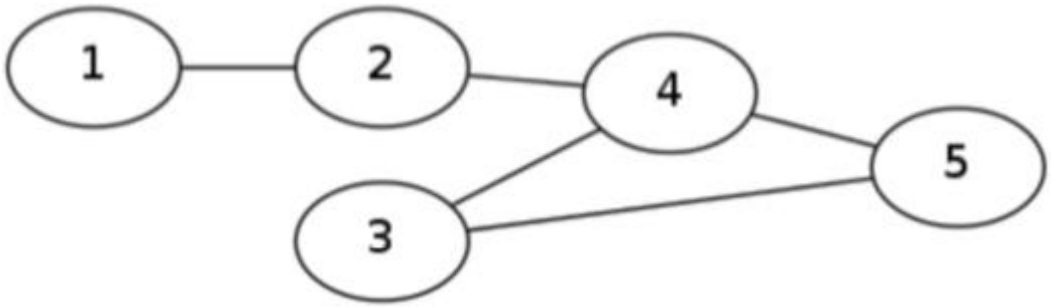


	1	2	3	4	5
1		T			
2	T			T	
3				T	T
4		T	T		T
5			T	T	

- Note reflection around i=j, due to undirectedness.
- Implied False in empty cells.

Representation and Implementation

Adjacency List



Node	Adjacency List
1	2
2	1,4
3	4,5
4	2,3,5
5	3,4

Space complexity

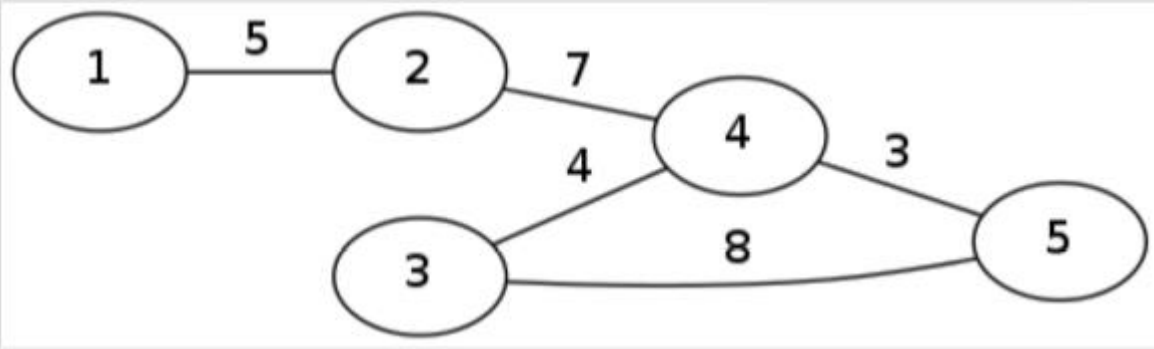
- So far, the data structures we have looked at have all taken about the same amount of memory space $O(n)$, where n is the number of elements.
- The adjacency matrix requires $O(n^2)$ space, where n is the number of vertices.
- Adjacency lists require $O(m)$ space where m is the number of edges.
- This method uses less space (**generally** - what if every node is connected to every other?) and is faster to search for common operations.

Weighted Graphs

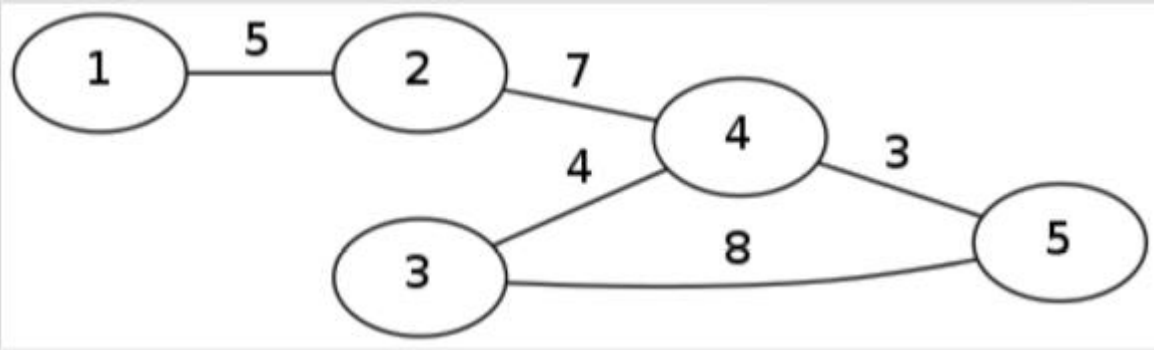
- Sometimes it's useful to store a number with each edge.
- This changes the way graphs are represented.
- The adjacency matrix is now numerical instead of boolean.
- Unconnected nodes can be given a default value, such as infinity for shortest path finding.
- The adjacency list must store the edges as pairs including the connection and the weight.

Representation and Implementation

Weighted Graph - Example



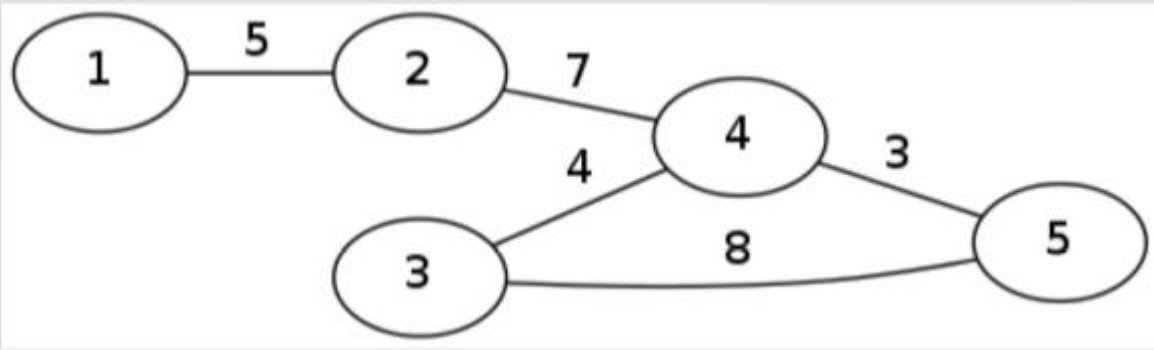
Representation and Implementation



Adjacency Matrix

	1	2	3	4	5
1		5			
2	5			7	
3				4	8
4		7	4		3
5			8	3	

Representation and Implementation



Adjacency List

Node	Adjacency List
1	2(5)
2	1(5), 4(7)
3	4(4), 5(8)
4	2(7), 3(4), 5(3)
5	3(8), 4(3)

Pre-Homework

- C++ people: create a simple object type (doesn't matter what) and then make an array of it. Try adding items to the array.
  Typically, we set all values to NULL and then we can check to see if each element of an array has been used by looking for NULL.
- Python people: create a simple object type (doesn't matter what) and then make a list. How do you add and remove new objects from the list? How do you know how many items are in the list? How can you find a specific item from the list?

Homework

1. Write the pseudocode for an unweighted graph data structure. You either use an adjacency matrix or an adjacency list approach. Also write a function to add a new node and a function to add an edge.
2. Implement the graph you have designed in the programming language of your choice. You may use your own linked list from previous labs, the STL LL, or use a simple fixed size array (10 elements would be fine).
3. What changes would you need to make to include weighted vertices?

Homework

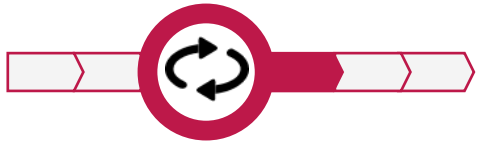
```
CLASS VERTEX
        label←Ø
        edges←[]

v←new VERTEX
v.label←24
v.edges.add(3)
```

Advanced Homework

- You must first complete the standard homework for this task. Look at the example code given previously for trees. You will find functions that output in the "dot" language of graphviz. This was used in lectures to show graphical representations of trees. Graphviz can also be used to display graphs. The examples given in this lecture are generated this way. Write a function that exports a graph as a dot file, or displays dot language output that can be redirected to a file.

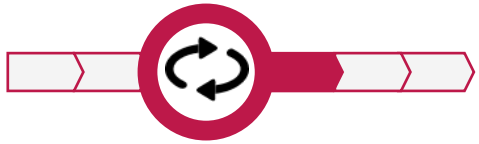
BREAK!!!

Graph Traversal (DFS)



Just a nice video

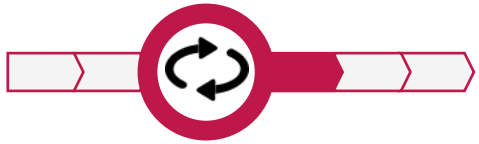
Graph
Traversal (DFS)



Graph Traversal

- Storing data in a graph is fairly easy.
- They don't really become useful until we can search for data or find information about the interconnections.
- Algorithms that interrogate the graph by following the edges are called **traversal algorithms**.

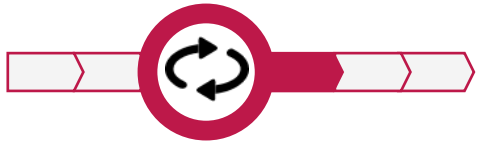
Graph
Traversal (DFS)



Depth-First Search (DFS)

- Visit all nodes beginning with v in graph G.
- Sometimes called Depth First Traversal (DFT).
- depthFirstSearch(G,v)
  - Not searching for v, even though this is what it reads like…

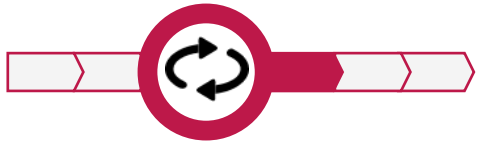
Graph
Traversal (DFS)



Depth-First Search (DFS)

- Traverses a graph by visiting each vertex in turn.
- Finds the first edge for the current vertex.
- Follows it to reach the next unvisited vertex.
- Marks the vertex as visited.
- Repeats until all vertices are marked as visited.

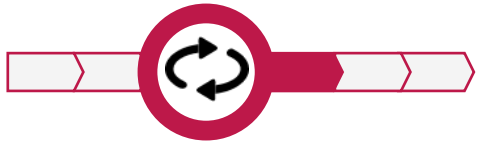
Graph
Traversal (DFS)



DFS Illustration

- https://www.cs.usfca.edu/~galles/visualization/DFS.html
- http://visualgo.net/dfsbfs.html
- https://www.youtube.com/watch?v=4AsEBQSjNUE
- https://www.youtube.com/watch?v=zLZhSSXAwxI

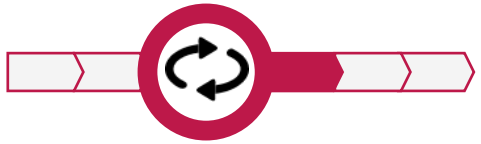
Graph Traversal (DFS)



DFS Pseudocode

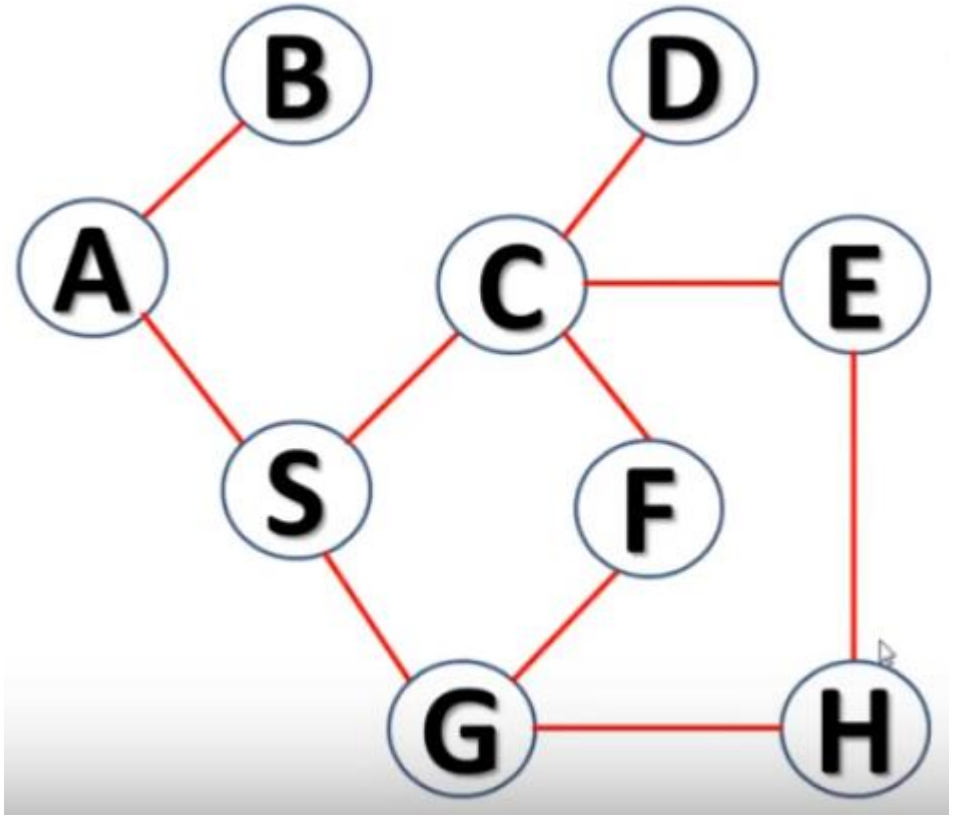
```
DEPTH-FIRST-SEARCH (G, v)
        S←new Stack()
        visited←[]
        S.push(v)
        while  S is not empty
                u←S.pop()
                if u is not in visited
                        visited.append(u)
                        for all edges, e, from u
                                S.push(e.to)
        return  visited
```

Graph
Traversal (DFS)

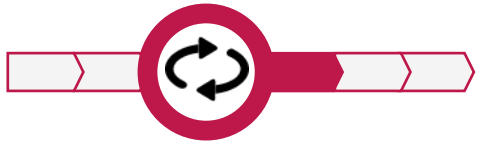


DFS Undirected Graph

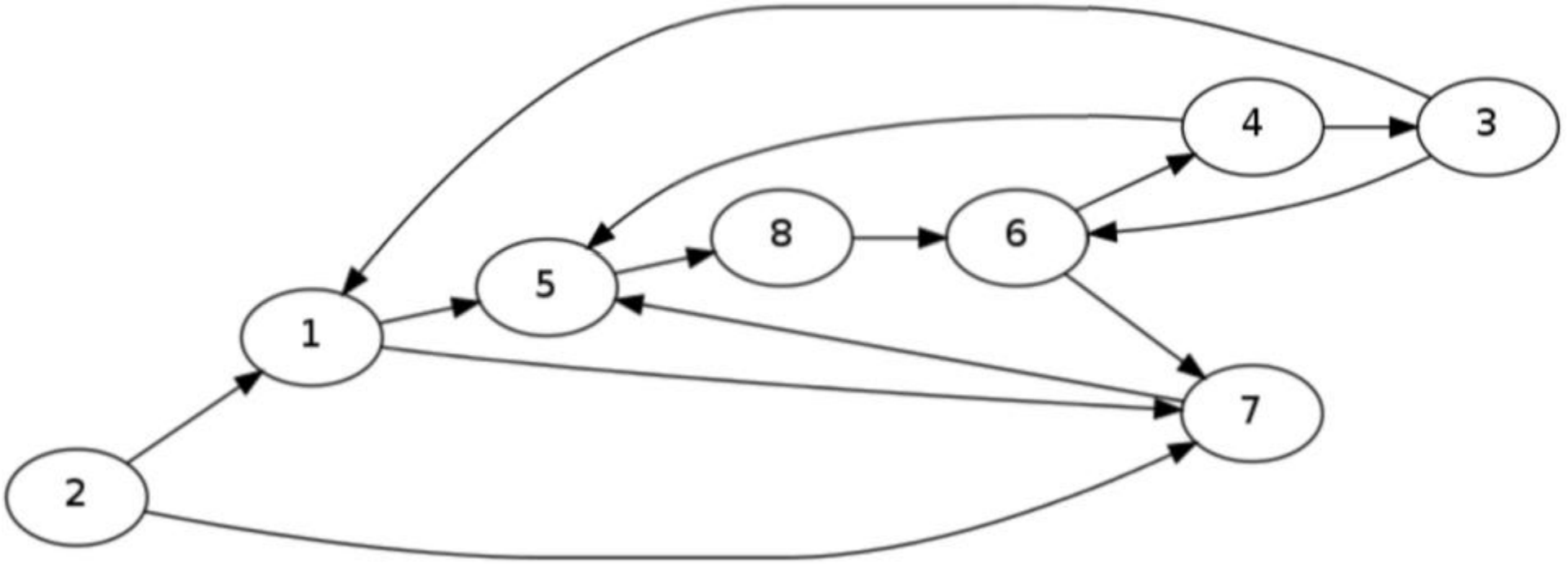
Output from our example graph is [A,B,S,C,D,E,H,G,F] when beginning with vertex A. What other order could it be?



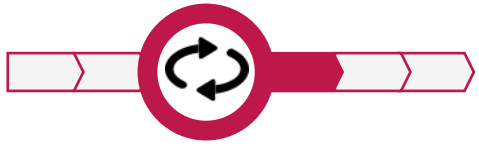
Graph Traversal (DFS)



DFS Directed Graph



Graph
Traversal (DFS)



DFS Pseudocode

- Output from the example graph is [1, 7, 5, 8, 6, 4, 3] when beginning with vertex 1. 2 is never visited because no edges go to it.
- Could have been different, but the edge ordering from each node is arbitrary.
- What other order could it be?

Graph
Traversal (BFS)

Breadth-First Search (BFS)

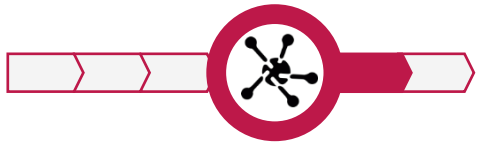
- Traverses the graph by searching every edge from the current vertex.
- Sometimes called Breadth First Traversal (BFT).
- Only moves to the next vertex in the graph when all edges have been explored.
- The algorithm is the same as DFS except that a Queue is used to store the list of vertices to visit.
- Will find the shortest path.

Graph
Traversal (BFS)

BFS Illustration

- http://visualgo.net/dfsbfs.html
- https://www.cs.usfca.edu/~galles/visualization/BFS.html
- http://joseph-harrington.com/2012/02/breadth-first-search-visual/
- https://www.youtube.com/watch?v=zLZhSSXAwxI

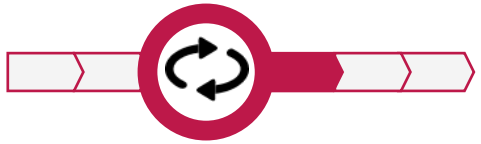
Graph
Traversal (BFS)



BFS Psedocode

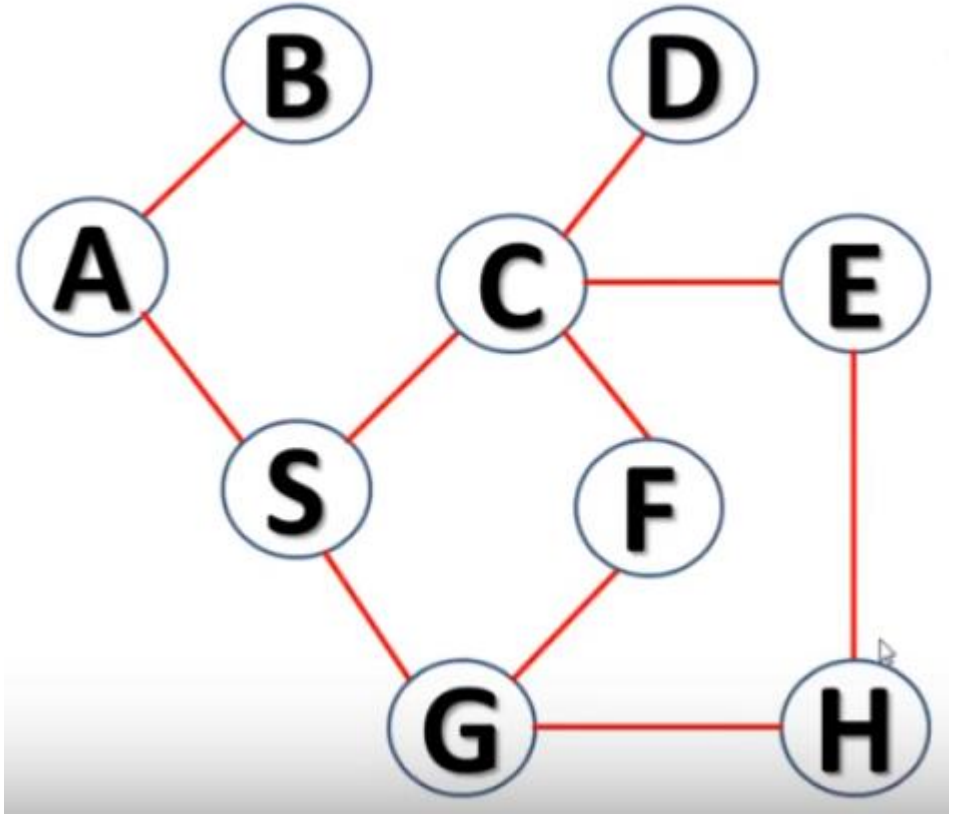
```
BREADTH-FIRST-SEARCH(G,v)
        Q←new Queue()
        visited←[]
        Q.enqueue(v)
        while  Q is not empty
                u←q.dequeue()
                if u is not in visited
                        visited.append(u)
                for all edges, e, from u
                        Q.enqueue(e.to)
        return  visited
```

Graph
Traversal (BFS)



BFS Undirected Graph

Output from our example graph is [A,B,S,C,G,D,E,F,H] when beginning with vertex A. What other order could it be?



Dijkstra's Algorithm

- Finds shortest path in a **weighted** graph.
- We start by setting a starting value for each vertex, 0 for the current one and infinity for all others.
- We also have a set for visited nodes and a set of tentative weights for unvisited nodes.
- Then we scan the neighbours of the source node and update the tentative distance to each of its neighbour nodes so that it is equal to the weight of the edge from the source to the neighbour the unvisited node with the minimum tentative weight now has its shortest path calculated.

Dijkstra's Algorithm

- Having found the shortest path to the node with minimum tentative weight, we then add it to the scanned set and then update the tentative weights of all the neighbours as follows:

```
for all scanned nodes u of G
    for all unscanned neighbours v of u
        v.tw ← min(v.tw, u.tw + u[v].w)
```

- v.tw is the tentative weight of vertex v
- u[v] is the edge from u to v
- x.w is the weight of edge x

Dijkstra's Algorithm

Dijkstra's Algorithm

- http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html
- http://visualgo.net/sssp.html
- https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html
- https://www.youtube.com/watch?v=UG7VmPWkJmA

```
DIJKSTRA(G,s,d)  //graph, source, destination
    v ← s //v is always our currently scanned node
    for all n in G.vertices
        n.tw ← ∞
    s.tw ← 0  //Source is no distance from itself
    visited ← []
    while v≠d
        for all vertices, u, adjacent to v
            if v.tw+v[u].w < u.tw
                u.tw ← v.tw+v[u].w
                u.pre ← v          //Store the return path
        visited.append(v)
        min ← ∞
        for all nodes n ∈ V
            if n ∉ visited ∧ n.tw < min
                v ← n
                min ← n.tw
```

Actual homework

- For the graph you have already implemented, implement either depth first or breadth first search. You may use the stack or queues from a library or your own implementation.

Alternative homework

- Adapt your graph to support weighted connections.
- Implement Dijkstra's algorithm.