Graph Theory

Dr Ian Cornelius

Hello

Hello (1)

Learning Outcomes

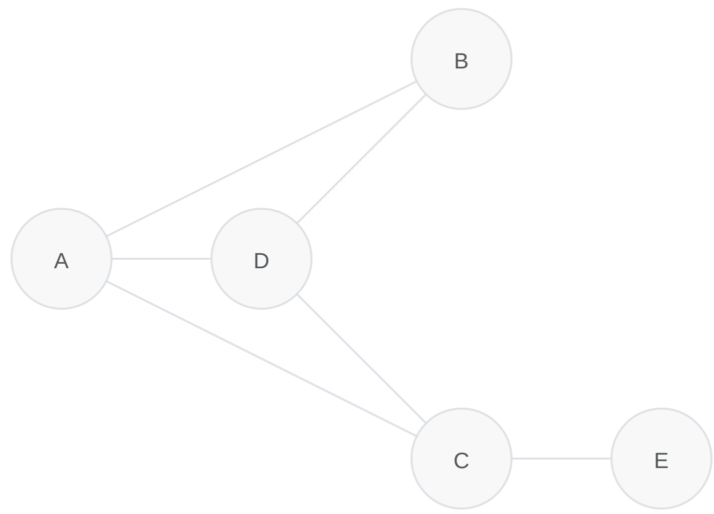
1. Understand the concept of graphs and their purpose as a data structure
2. Demonstrate and implement their knowledge of graphs

Graph Theory

Graph Theory (1)

What are Graphs?

- Graphs are the basis for a large amount of programming
- They are essentially a set of nodes with connections between them
- You may think of them as **Lists: The Next Generation**
  - imagine a linked list, where any element of data can have as many *next* elements and as many parents as needed



Graph Theory (2)

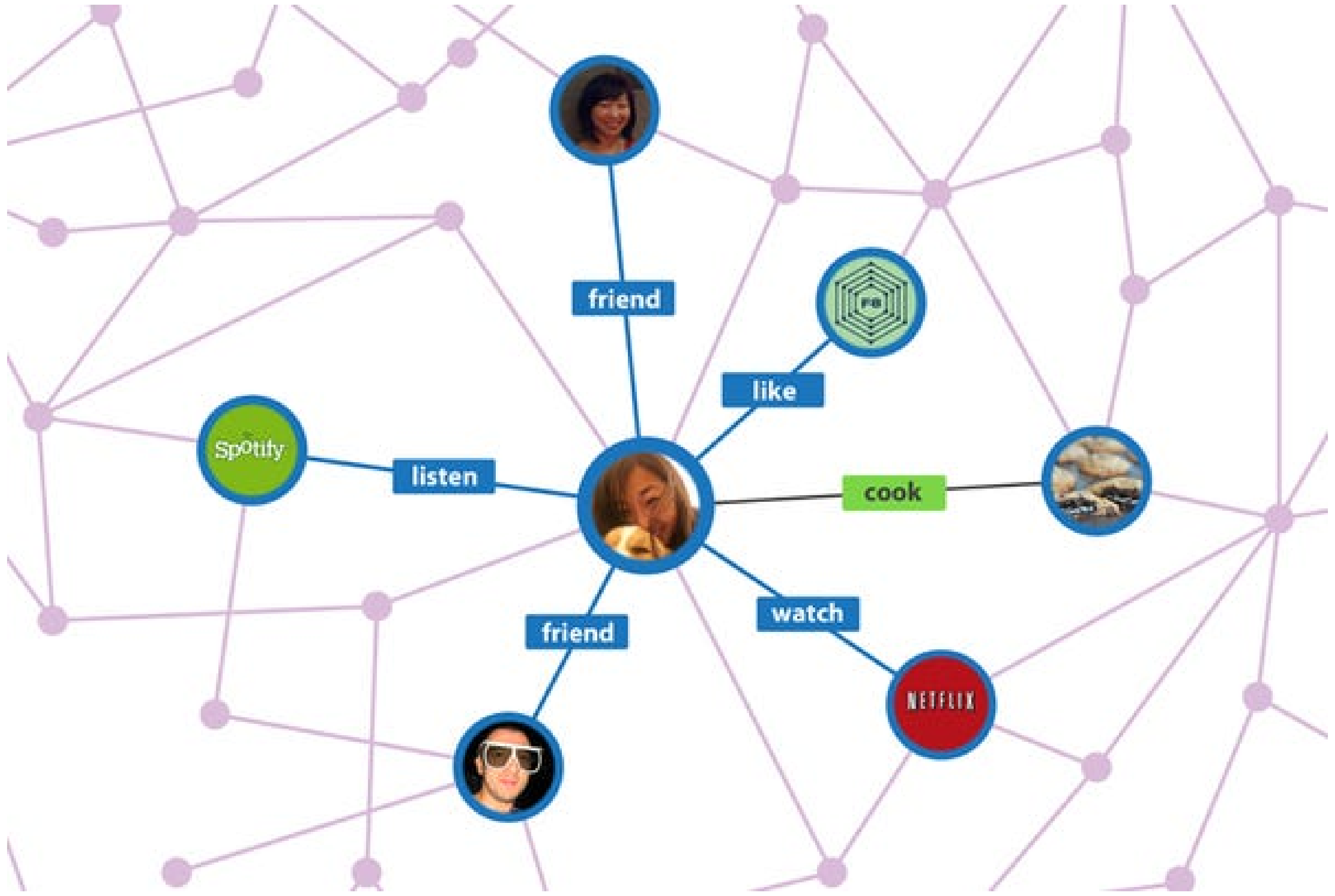
Use-cases for Graphs

- Graphs can be useful for:
  1. Map analysis, route finding, path planning
  2. Ranking search results
  3. Analysing related data such as social networks
  4. Compiler optimisation
  5. Constraint satisfaction
     - i.e. timetabling
  6. Physics simulations
     - i.e. games
  7. Social connections
  8. Decision-making
     - i.e. goal-oriented action planning and strategies

Graph Theory (3)

Facebook: The Social Graph

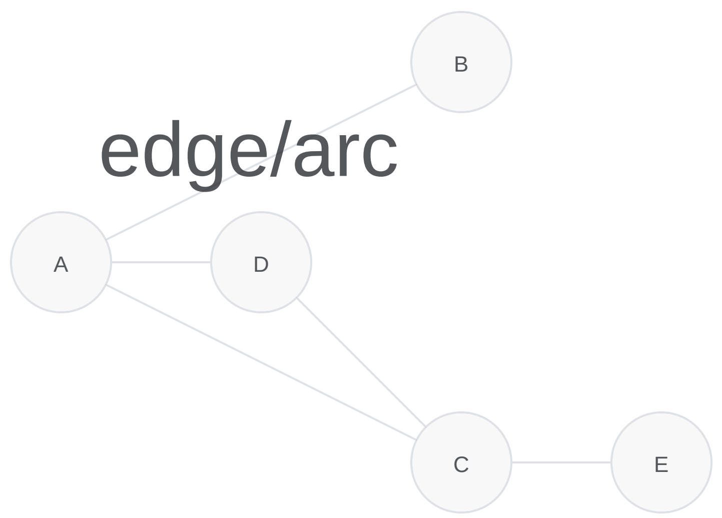
- Draws an edge between you and the people, places and things you interact with online
- Whenever you like something on Facebook, it becomes an edge
  - This edge is a connection between you and other people, places or things
- Your photos, events and pages are connected with other information



Graph Theory (4)

Formal Terminology of Graphs

- Graphs are a collection of nodes that have links between them
- There are some terminologies you need to remember:
  - *nodes* are called **vertices**
  - *links* (connections between nodes) are called **edges** (or sometimes **arcs**)
    - an edge is an **incident** if it connects to another vertex
  - connected vertices are called **adjacent** or **neighbours**
  - a vertices **degree** is a number of edges that incident on it



Types of Graphs

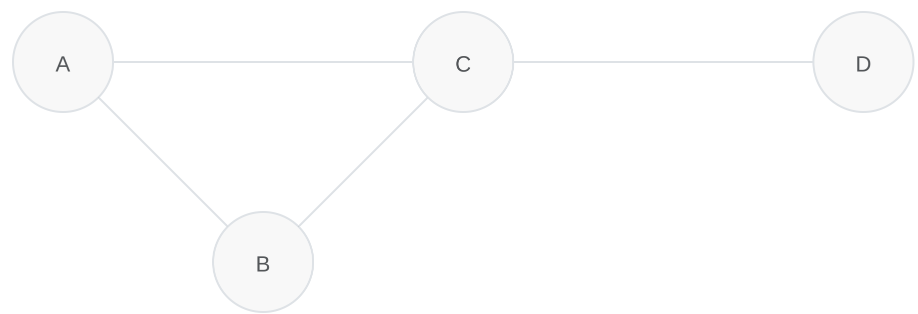
Types of Graphs (1)

- There are many types of graphs:
  1. Undirected
  2. Directed
  3. Vertex Labelled
  4. Cyclic
  5. Weighted
  6. Connected
  7. Disconnected
  8. Directed Acyclic Graph (DAG)

Types of Graphs (2)

Undirected Graphs

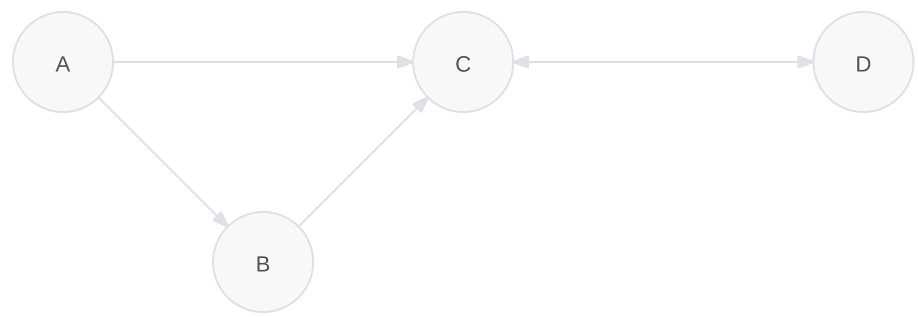
- Edges can be traversed in either direction
- The vertices can be imagined as junctions on a road network
  - the edges are a two-way road between junctions
  - i.e. you can travel from node A to node B and then travel back from node B to node A



Types of Graphs (3)

Directed Graphs

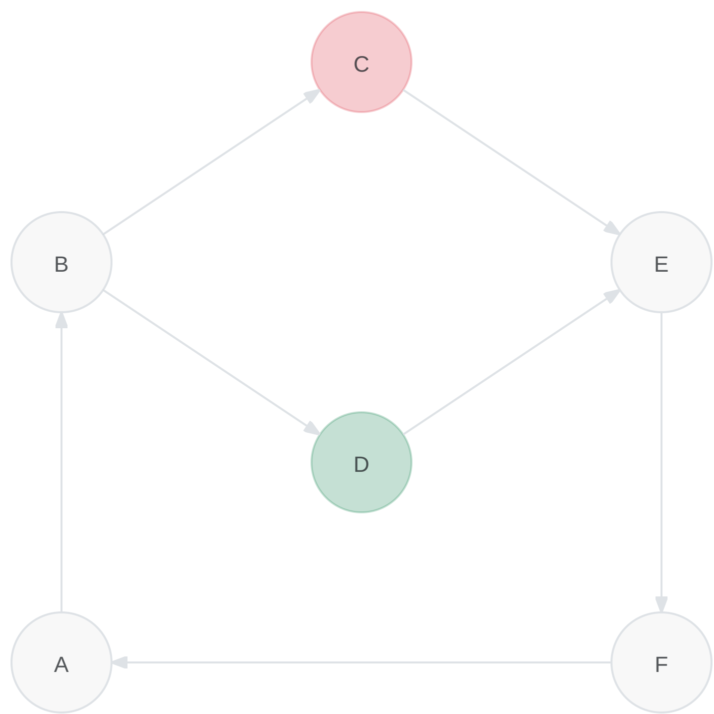
- Each edge is directional and does **not** imply the inverse
- The vertices can be imagined as junctions on a road network
  - the edges are a one-way roads between junctions
  - i.e. we can travel from node `A` to node `C` but we *cannot* travel back to node `A`



Types of Graphs (4)

Vertex Labelled

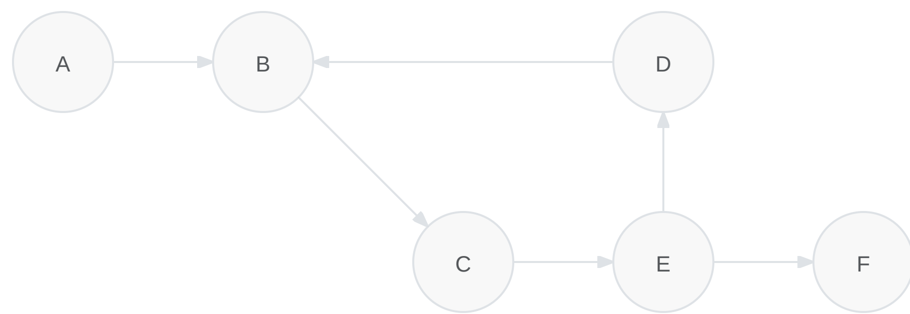
- The data used to identify each node is not the only information that is important about that node
  - i.e. it may also have a `colour` assigned to it that affects the algorithm decision or choice
- The nodes can be imagined as roundabouts or slip-roads on a road network
  - i.e. a red node is a heavily congested roundabout and should be bypassed



Types of Graphs (5)

Cyclic

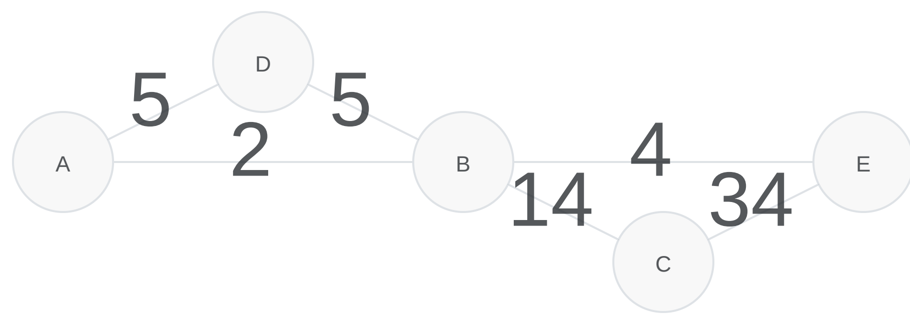
- Consists of at least one *cycle*
  - i.e. there is a path that exists from a single node that can lead back to itself
- Imagine our road network again; it is a roundabout where there is a vertex for each junction



Types of Graphs (6)

Weighted

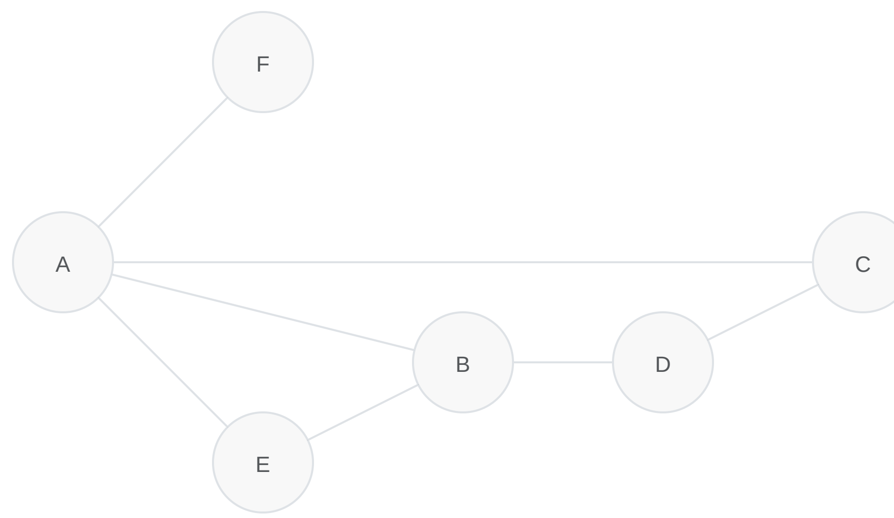
- Parameters along edges or at nodes are interval data and can be summed and/or compared
- This is similar to vertex labelled but more versatile
- Thinking about the road network example, the weight of our graph edges could be according to the speed limit
  - it could mean the algorithm would favour faster roads over a slower road in route planning



Types of Graphs (7)

Connected

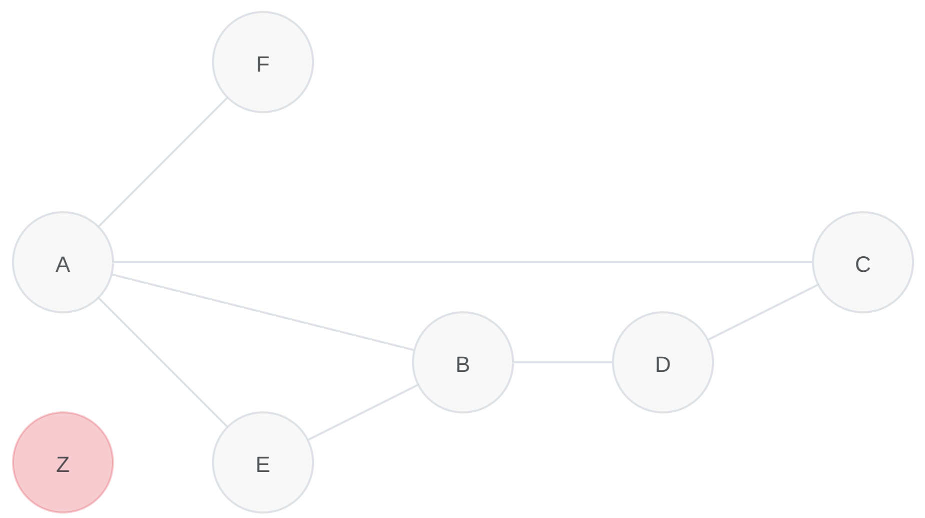
- There is an edge between every pair of nodes in a graph



Types of Graphs (8)

Disconnected

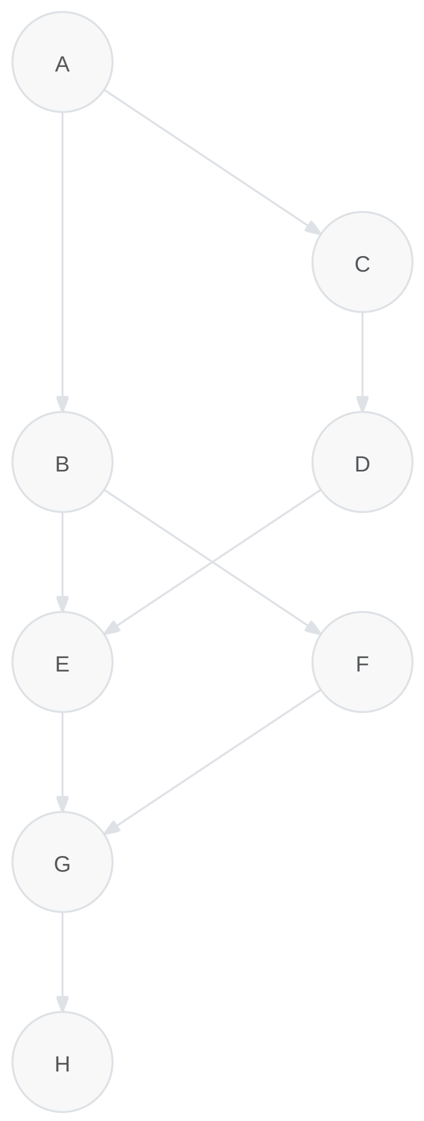
- There is a node that does **not** have any connection to a node in the graph
  - the red node makes our graph *disconnected*



Types of Graphs (9)

Directed Acyclic Graph (DAG)

- Links in these graphs have a direction and there are no cycles
- It consists of vertices and edges, where each edge is directed from one vertex to another
  - they follow the directions of the other nodes and never form a closed loop
- It Can be visualised like a river system heading out to sea
  - it may fork and join at parts, but it always does so going downstream



Degrees, In-degrees and Out-degrees

Degrees, In-degrees and Out-degrees (1)

- **Recap**: Degrees count the number of edges connected to a node
- Three types of *degrees* for a graph:
  1. degree
  2. in-degree
  3. out-degree

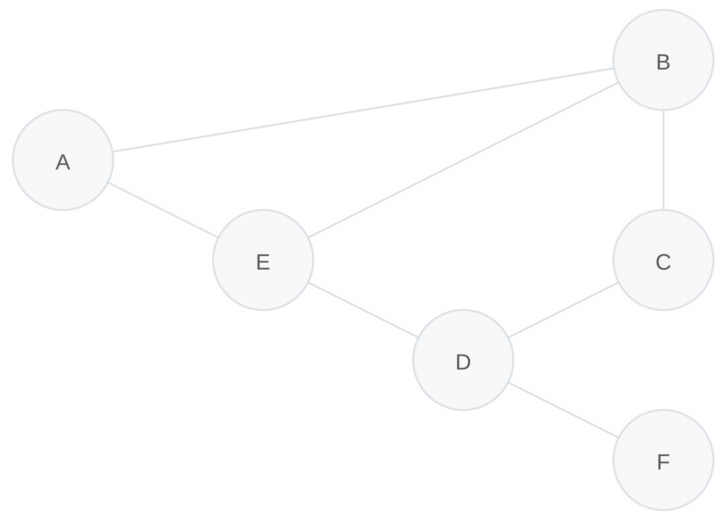
Degrees, In-degrees and Out-degrees (2)

Undirected Graphs

- Concerned with only counting the total number of connections for a node
  - in this instance, the **degree**

Vertex	Degree
A	
B	
C	
D	
E	
F	

Populate



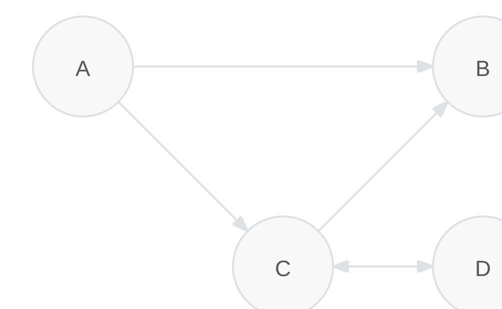
Degrees, In-degrees and Out-degrees (3)

Directed Graphs

- Concerned with counting:
  - the total number of connections, known as the **degree**
  - the number of incoming connections, known as the **in-degree**
  - the number of outgoing connections, known as the **out-degree**

Vertex	Degree	In-Degree	Out-Degree
A			
B			
C			
D			

Populate

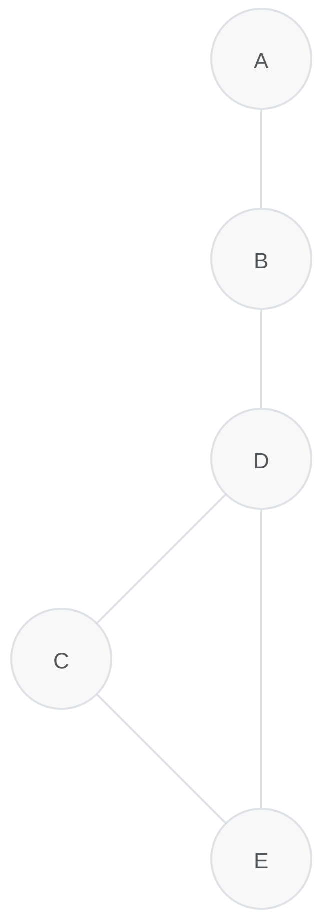


Representation of Graphs

Representation of Graphs (1)

Mathematical Notation

- The elements of a graph can be represented in different methods:
  - vertices with integers (or any unique value)
  - edges as a pair of vertices, i.e. `(1, 0)`
- A graph `G` will consist of a set of vertices `V` and a set of edges `E`
  - represented as `G = (V, E)`
- `n` and `m` can be used to represent the number of vertices and edges
  - think `n` for node if you find it difficult to remember which way around these go
- `G = (V, E)`, where,
  - `V = [A, B, C, D, E]`
  - `E = [(A,B), (B,D), (C,D), (C,E), (D,E)]`
- Final form:
  - $G = ([A, B, C, D, E], [(A, B), (B, D), (C, D), (C, E), (D, E)])$



Representation of Graphs (2)

Programming Methodology

- When it comes to representing a graph in programming, there are two ways of implementing a graph:
  1. Adjacency Matrices
  2. Adjacency Lists

Representation of Graphs (3)

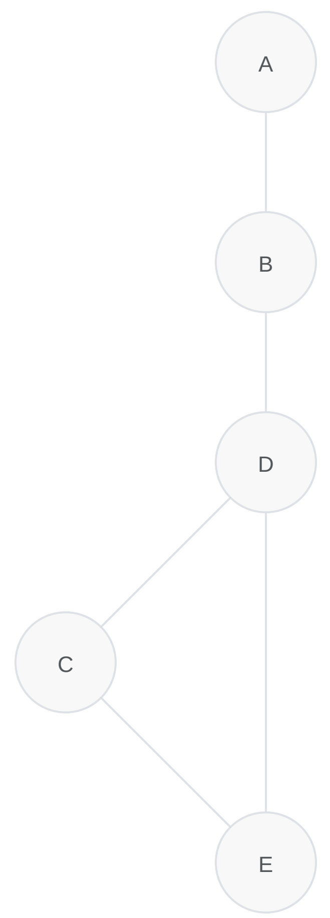
Adjacency Matrices

- A two-dimensional matrix of boolean values
  - the value of a cell `(i, j)` is true if the vertices are connected
- Adjacency matrices are symmetrical along the diagonal for undirected graphs
- However, for directed graphs a connection `(1, 2)` does not imply a connection between `(2, 1)`
- Adjacency matrix requires $O(n^2)$ space, where n is the number of vertices
- Code Representation:

```
aGraph = [
  [False, True, False, False, False],
  [True, False, False, True, False],
  [False, False, False, True, True],
  [False, True, True, False, True],
  [False, False, True, True, False]
]
```

- Tabular Representation:

	A	B	C	D	E
A		T			



Representation of Graphs (4)

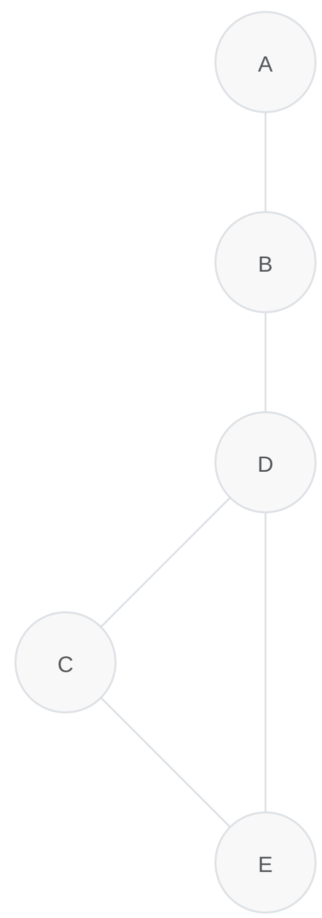
Adjacency List

- Each vertex contains a list of vertices that it is connected to the other
  - often stored as a dictionary in Python
- Adjacency lists requires up to $O(n+m)$ space
  - where n is the number of nodes in our graph and m is the number of edges
- Code Representation:

```
aGraph = {
  "A": ['B'],
  "B": ['A', 'D'],
  "C": ['D', 'E'],
  "D": ['B', 'C', 'E'],
  "E": ['C', 'D']
}
```

- Tabular Representation:

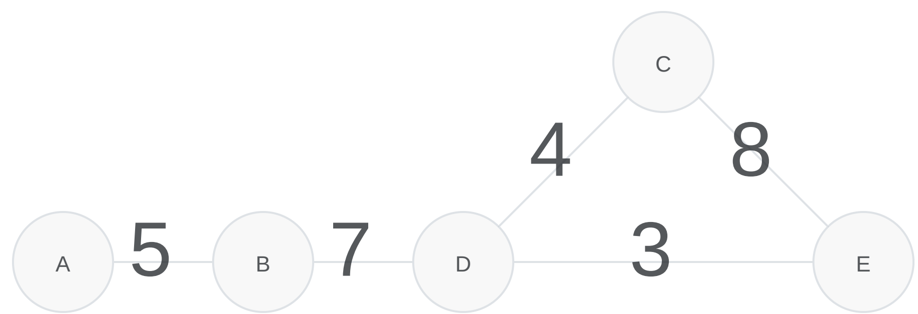
Vertex	Adjacency List
A	B
B	A, D
C	E, D



Representation of Graphs (5)

Recap: Weighted Graphs

- It is sometimes useful to store a number with each edge
  - this will change the way the 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 sture edges as pairs, including the connection and the weight
  - For example, a tuple could easily be represented using a simple struct with two variables
    - `int neighbour` and `float weighting`
    - i.e. `(0, 5.0)`



Representation of Graphs (6)

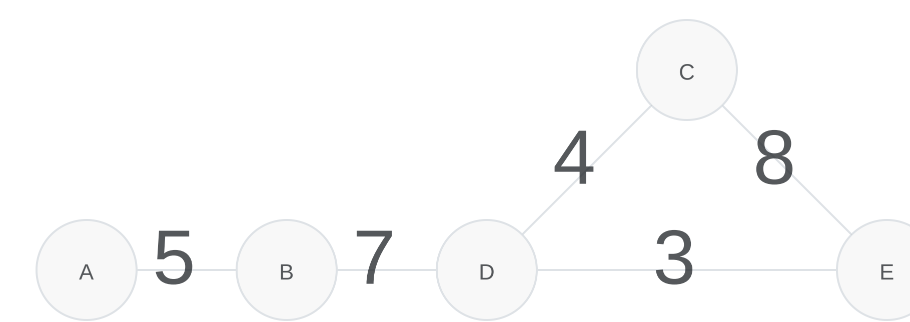
Adjacency Matrix — Weighted

- For a weighted adjacency matrix, boolean values are replaced with numbers
  - i.e. the cost of traversing from one node to another
- **Note**: That if your weightings are positive floating point values
  - may benefit you by making the `False` values a negative number, i.e. `-1`, rather than checking for `== 0.0f`
- Code Representation:

```
aGraph = [
  [-1, 5, -1, -1, -1],
  [5, -1, -1, 7, -1],
  [-1, -1, -1, 4, 8],
  [-1, 7, 4, -1, 3],
  [-1, -1, 8, 3, -1]
]
```

- Tabular Representation:

	A	B	C	D	E
A		5			
B	5			7	
C				4	8



Representation of Graphs (7)

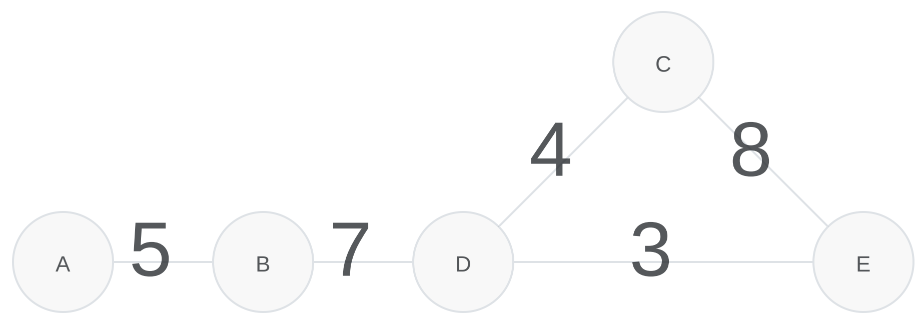
Adjacency List — Weighted

- For a weighted adjacency lists, connected nodes have an associated weight provided next to them
  - i.e. a number provided in brackets
- Code Representation:

```
aGraph = {
  "A": [('B',5)],
  "B": [('A',5), ('D',7)],
  "C": [('D',4), ('E',8)],
  "D": [('B',7), ('C',4), ('E',3)],
  "E": [('C',8), ('D',3)]
}
```

- Tabular Representation:

Node	Adjacency List
A	B(5)
B	A(5), D(7)
C	E(4), D(8)
D	B(7), C(4), E(3)
E	C(8), D(3)



Goodbye

Goodbye (1)

Questions and Support

- Questions? Post them on the **Community Page** on Aula
- Additional Support? Visit the Module Support Page
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk