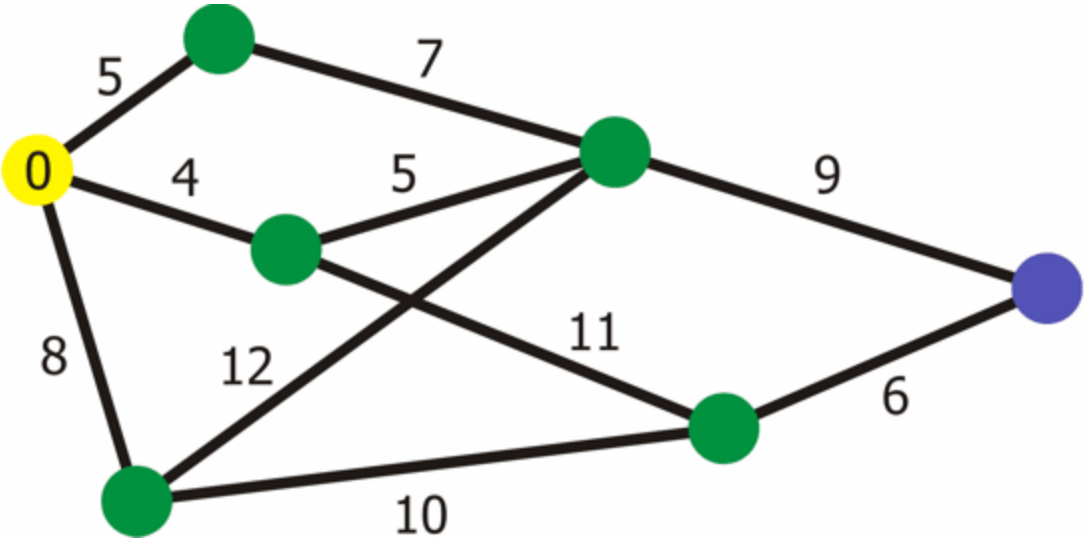
Hereford
SIXTH FORM COLLEGE

Dijkstra's Shortest Path Algorithm



Path Finding Algorithms

An important class of algorithms deals with this question:

If there are several ways to arrive at a goal and there are choices to be made along the way, what are the best set of choices and how do we find that optimal solution

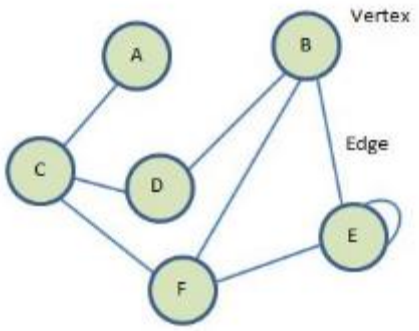
This is called path-finding and happens in many fields:

- Navigation
- Network traffic
- Speech recognition
- Image recognition
- Artificial Intelligence
- Robotics
- Gaming
- Military

Graphs

The problem needs to be modelled in such a way that a computer program can be applied to solve it.

This modelling is performed using a graph structure.



Each connected item or node in a graph is called a **vertex**

Each line connecting two vertices is called an **edge**

Each vertex has a **degree**, which describes how many connections that vertex has e.g. B is degree 3

Graphs

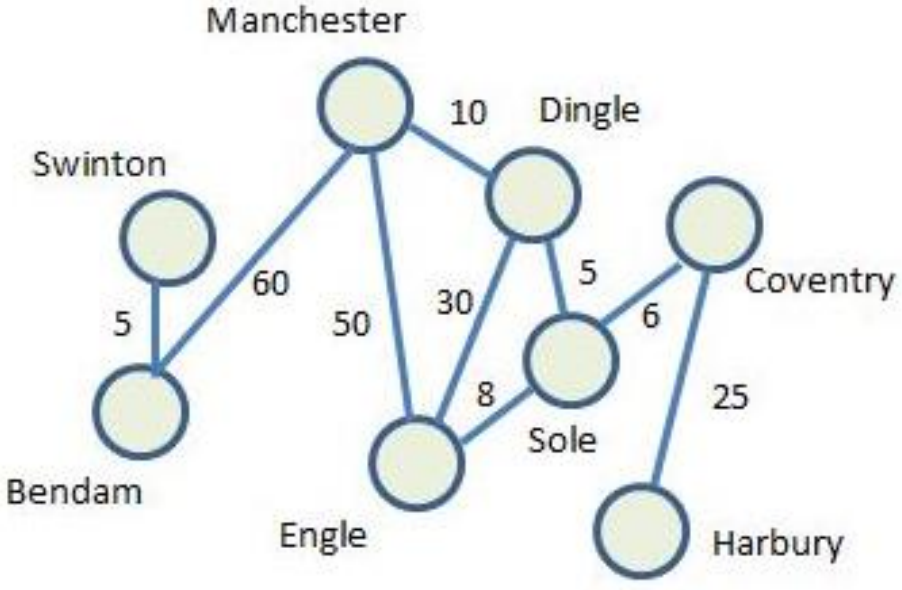
The graph represents an **abstraction** of the problem.

In order to use the graph with a path-finding algorithm, each vertex needs to represent something and each edge needs to have some measure associated with it – this is called the **cost**.

E.g. A graph could be used to model a road network.

Each vertex is a location and each edge is a road.

In this case the cost could be physical distance or travel time.



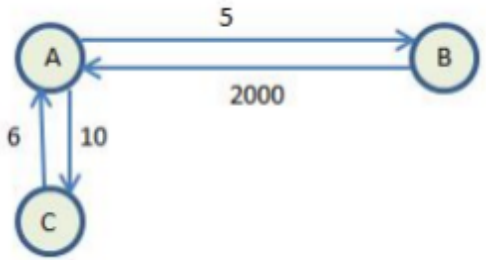
Graphs

Having a value associated with each edge means that a least-cost path can be calculated.

The nodes themselves could also have costs associated with them – this is called a **weighted graph**.

Each edge direction could be shown as arrows rather than lines, in which case it is a **directed weighted graph**.

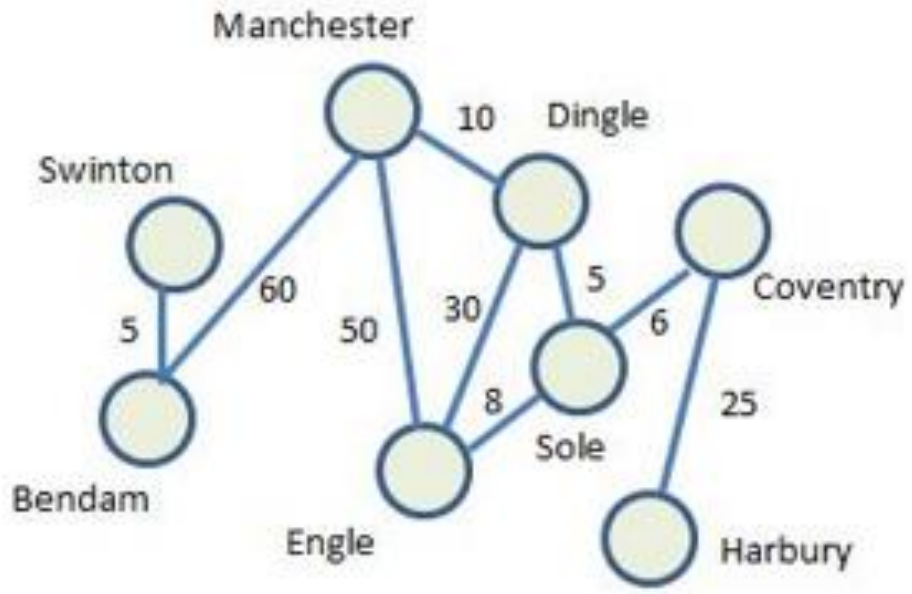
With a directed graph there could be two arrows indicating that the cost may depend on direction.



Modelling the Cost

Any non-trivial path needs to define three things:

- A start node
- A target node
- An intermediate node (if there isn't a direct edge between the start and the target)



Dijkstra's Algorithm

This algorithm is also called the **uniform cost search** algorithm as **no heuristic *h(n)*** is involved in finding the optimal path.

The cost expression of any node is simply:

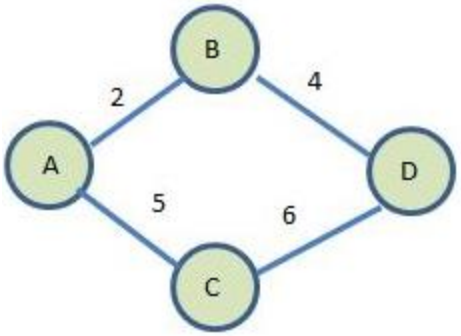
$$f(n) = g(n)$$

Where g(n) is the cost of getting from the start to node n.

When n is the target, then the path that offers the smallest f(n) is the shortest path to take.

Dijkstra's Algorithm

In the example below, where A is the start and D is the target node, each path is calculated in its entirety.



Path	F(d)
A, C, D	11
A, B, D	6

In this case the shortest path is A, B, D with a cost of 6.

Dijkstra's Algorithm

The algorithm works like this:

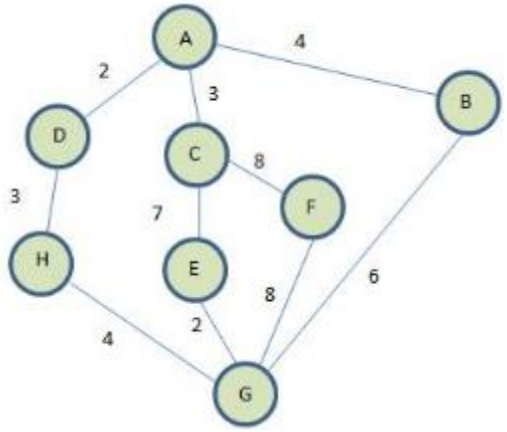
1. Assign a cost of 0 to the start node
2. Assign a cost of infinity to every other node
3. Create an unvisited list containing every node
4. From the start node, visit every directly connected neighbour.
5. If the cost of getting to that neighbour (the edge value) is less than the current stored cost, replace it with the lower cost (as we started with a cost of infinity, the neighbouring nodes of the start node will always be the edge cost).

Dijkstra's Algorithm

6. Record the last node that you went through to get to the current node. For the starting node's neighbours, this will be the start node initially.
7. If all its neighbours have been visited, remove the current node from the unvisited list.
8. From the nodes just visited, find the one with the least current cost
9. Visit it's neighbours that are still in the unvisited list
10. The cost of each node is sum of the edge value, plus the cost of the prior node
11. Repeat steps 5 – 9 until we reach the target and it is at the top of the unvisited list.

Dijkstra's Algorithm

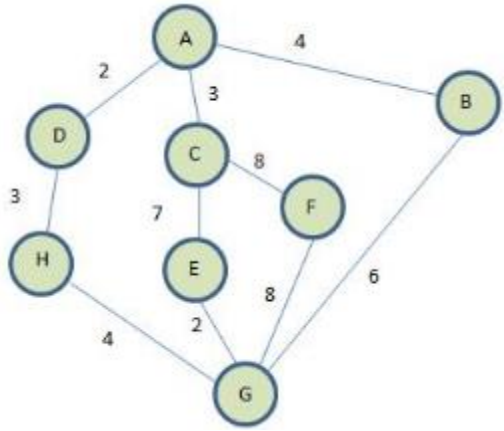
Worked example using this graph, trying to find the shortest path from A to G.



For this small graph we ***could*** use the brute force method i.e. work out every paths *f(G)* and determine the smallest (this was how Dijkstra's original algorithm worked)

This means that the big O of following every possible path is *O(n!)* - not good for large graphs!

Dijkstra's Algorithm



Let's use Dijkstra's method instead!

1. **Initialise**

   - Set up the unvisited list:

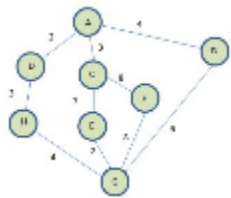
     {A, B, C, D, E, F, G, H}

   - Set up the initial cost of each node:

     A = 0

     B, C, D, E, F, G, H = ∞

   - Set up a came from list which is empty



Dijkstra's Algorithm

2. Iteration 1

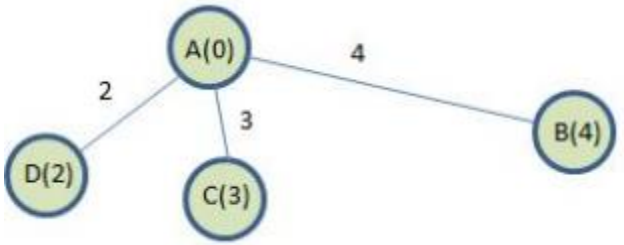
The initial data values are:

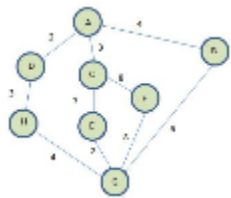
Vertex	Visited	Cost	From Vertex
A	No	0	N/A

The neighbours of A are visited and the relative costs calculated and ordered.

The *From Vertex* column is updated to the node immediately prior to the target node.

Vertex	Visited	Cost	From Vertex
A	No	0	N/A
D	No	2	A
C	No	3	A
B	No	4	A



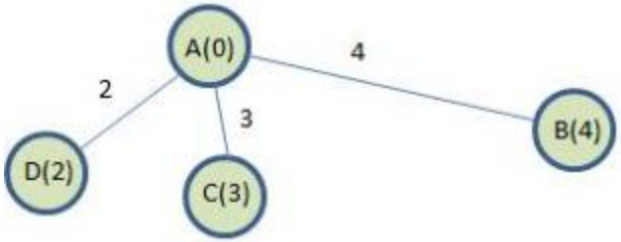


Dijkstra's Algorithm

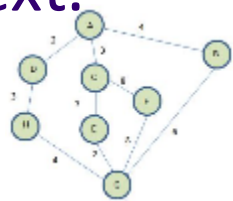
2. Iteration 1 cont.

All of A's neighbours have now been visited, so A is removed from the unvisited list (we'll mark it as visited in our data table)

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	No	2	A
C	No	3	A
B	No	4	A



The costs of the unvisited neighbours are studied and the node with the shortest path (D) is considered next.



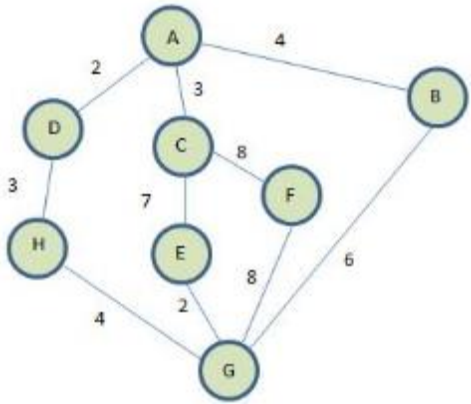
Dijkstra's Algorithm

3. Iteration 2

The unvisited neighbours of D are visited and the relative costs calculated.

The calculation includes the cost to D (2) and then the cost of D to H (3)

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	No	2	A
C	No	3	A
B	No	4	A
H	No	2 + 3 = 5	D

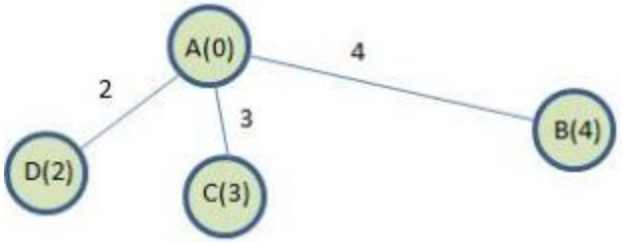


Dijkstra's Algorithm

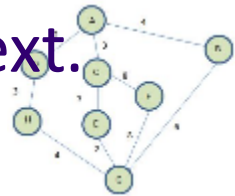
3. Iteration 2 cont.

All of D's neighbours have now been visited, so D is removed from the unvisited list (we'll mark it as visited in our data table)

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	No	3	A
B	No	4	A
H	No	5	D



The costs of the unvisited neighbours are studied and the node with the shortest path (C) is considered next.



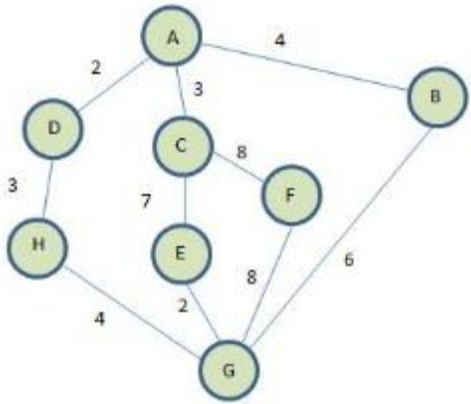
Dijkstra's Algorithm

4. Iteration 3

The unvisited neighbours of C are visited and the relative costs calculated.

The calculation includes the cost to C (3) as well as the next edge

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	No	3	A
B	No	4	A
H	No	5	D
E	No	3 + 7 = 10	C
F	No	3 + 8 =11	C



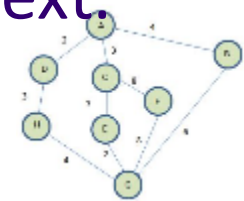
Dijkstra's Algorithm

4. Iteration 3 cont.

All of C's neighbours have now been visited, so C is removed from the unvisited list (we'll mark it as visited in our data table)

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	Yes	3	A
B	No	4	A
H	No	5	D
E	No	10	C
F	No	11	C

The costs of the unvisited neighbours are studied and the node with the shortest path (B) is considered next.



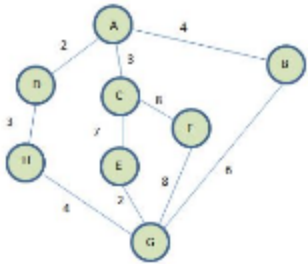
Dijkstra's Algorithm

5. Iteration 4

The unvisited neighbours of B are visited and the relative costs calculated.

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	Yes	3	A
B	No	4	A
H	No	5	D
E	No	10	C
G	No	4 + 6 = 10	B
F	No	11	C

The calculation includes the cost to B (4) as well as the next edge



Dijkstra's Algorithm

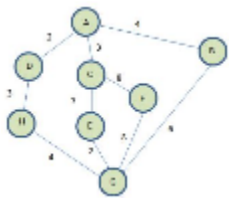
5. Iteration 4 cont.

All of B's neighbours have now been visited, so B is removed from the unvisited list (we'll mark it as visited in our data table)

Note that we have found the target, but as it isn't the next shortest path, we keep going – there could be a shorter route than the current one

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	Yes	3	A
B	Yes	4	A
H	No	5	D
E	No	10	C
G	No	10	B
F	No	11	C

The costs of the unvisited neighbours are studied and the node with the shortest path (H) is considered.



Dijkstra's Algorithm

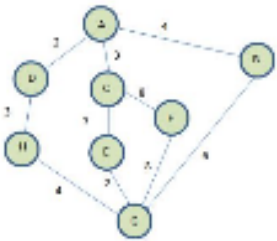
6. Iteration 5

The unvisited neighbours of H are visited and the relative costs calculated.

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	Yes	3	A
B	Yes	4	A
H	No	5	D
G	No	9	H
E	No	10	C
F	No	11	C

The calculation includes the cost to H (5) as well as the next edge

As the cost of H to G is **less** than getting to G via B, the **cost** and **from vertex** are updated



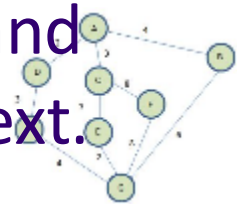
Dijkstra's Algorithm

6. Iteration 5 cont.

All of H's neighbours have now been visited, so H is removed from the unvisited list (we'll mark it as visited in our data table)

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	Yes	3	A
B	Yes	4	A
H	Yes	5	D
G	No	9	H
E	No	10	C
F	No	11	C

The costs of the unvisited neighbours are studied and the node with the shortest path (G) is considered next.



Dijkstra's Algorithm

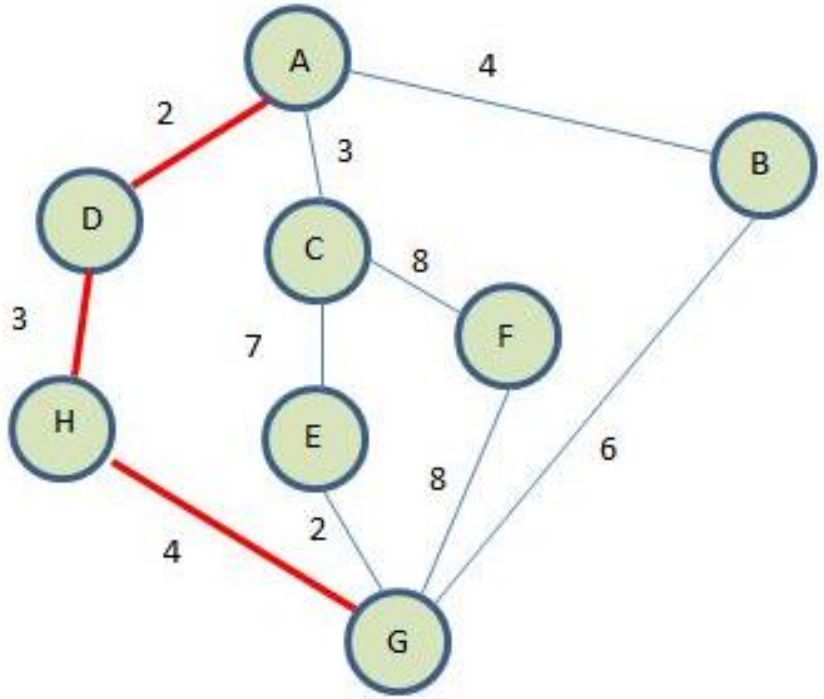
G is our target and is at the **top** of the unvisited priority queue, so we are done!

We don't bother looking at the paths through E & F, as we would in a brute force method, as their costs are higher.

Now use the *from* vertex data to walk the best path:

G ⇒H ⇒D ⇒A = 9

Vertex	Visited	Cost	From Vertex
A	Yes	0	N/A
D	Yes	2	A
C	Yes	3	A
B	Yes	4	A
H	Yes	5	D
G	Yes	9	H
E	No	10	C
F	No	11	C



Dijkstra's Performance

Using the **priority queue structure** (a modification of Dijkstra's algorithm made in 1984), where the next unvisited node has the smallest cost, the performance is improved from brute force (*O(n!)).*

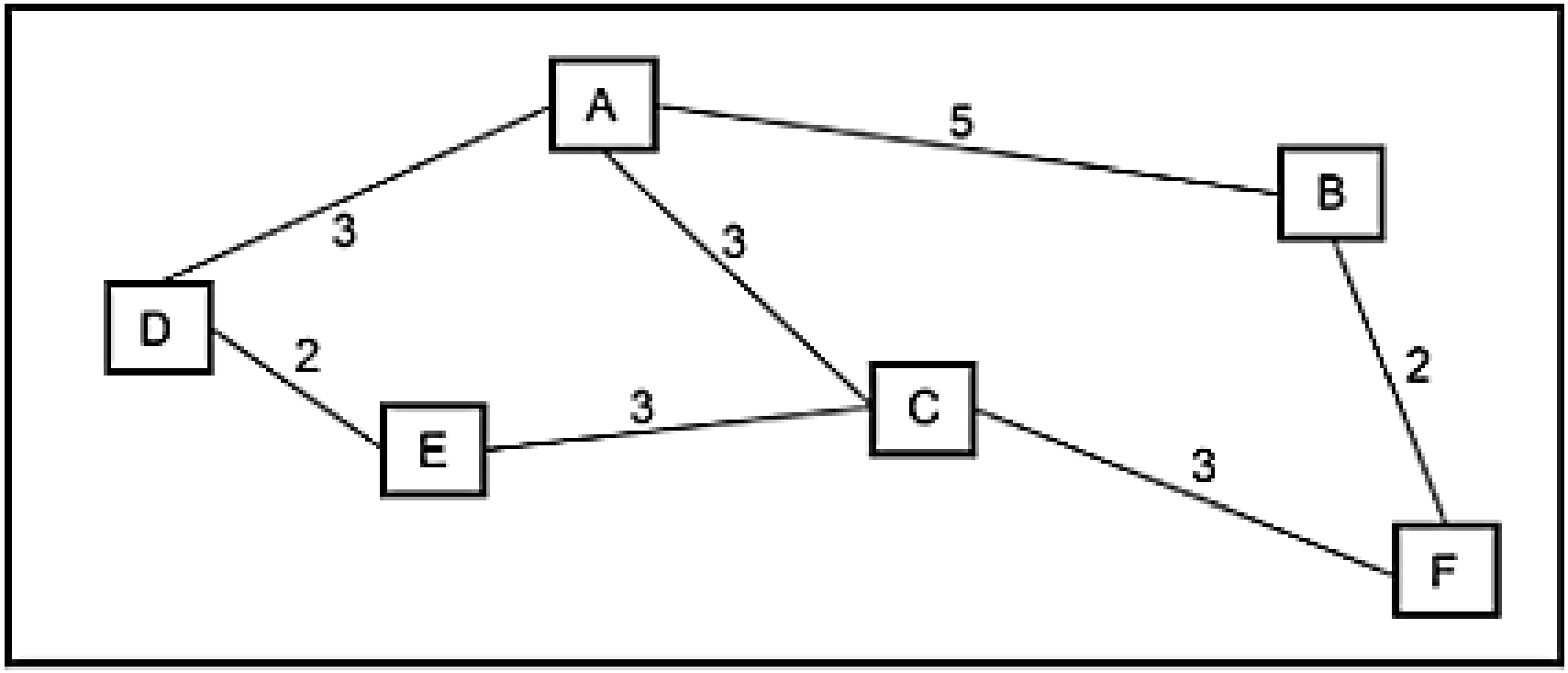
The worst case is now:

$$O(E + V \log V)$$

Where E is the number of edges and V is the number of vertices.

This makes it suitable for large graphs with 100's of nodes.

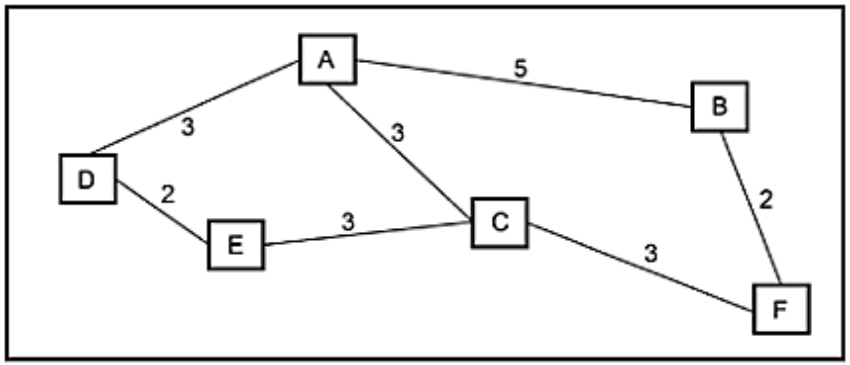
Over to you!



On your whiteboard, show how Dijkstra's algorithm would find the shortest path from A to F

Node	Visited	Cost	From Vertex
A	No	0	N/A

Worked example



Node	Visited	Cost	From Vertex
A	~~No~~ Yes	0	N/A
D	~~No~~ Yes	~~∞~~ 3	A
C	~~No~~ Yes	~~∞~~ 3	A
B	~~No~~ Yes	~~∞~~ 5	A
E	~~No~~ Yes	~~∞~~ 3+2 =5	D
F	No	~~∞~~ 3+3 =6 ~~5+2=7~~	C ~~B~~

E has no unvisited neighbours, so simply mark as visited

The route through B is longer than through C, so don't update the path

The shortest path to F is: A, C, F with a cost of 6