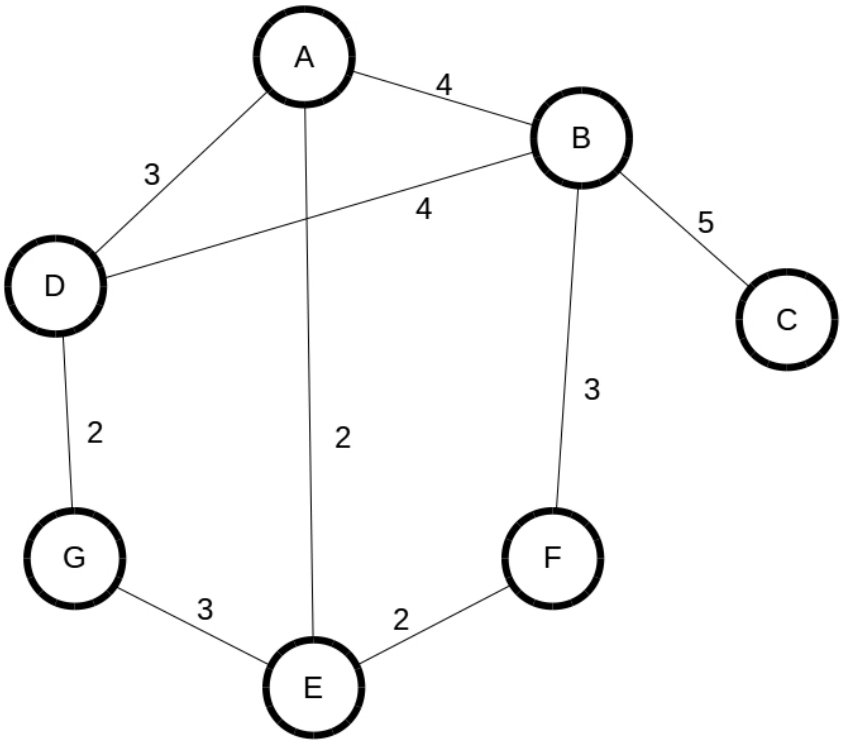
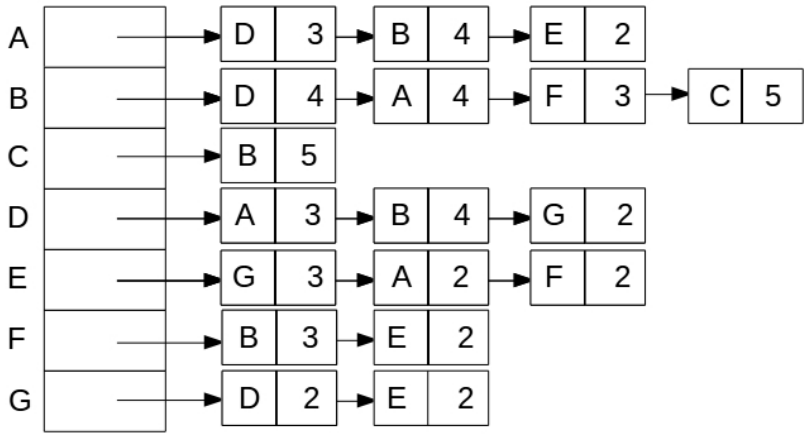
Depth-First Search



Here is the same graph as before, with weights added to the edges. We can modify the adjacency matrix and adjacency list representations to incorporate this information.

	A	B	C	D	E	F	G
A		4		3	2		
B	4		5	4		3	
C		5					
D	3	4					2
E	2					2	3
F		3			2		
G				2	3		



We have already discussed breadth-first search – now we look at depth-first search.

The idea of DFS (Depth-First Search) is to explore the graph by always taking a step from the most recently seen vertex – when we can't go to a new vertex, we back up and go forward from the most recent vertex that still has any unvisited neighbours.

Using the graph shown above, if we start at vertex E the candidates for the next vertex are E's neighbours G, A and F – we choose one of them - this choice is random or arbitrary – suppose we choose A. From A, the unvisited neighbours are D and B. Suppose we choose D. D's unvisited neighbours are B and G … and so on.

The DFS algorithm is very simple to express:

```
def DFS(v):
     mark v "visited"
     for each neighbour y of v:
          if y is not visited:
               DFS(y)
```

This expression of the algorithm gains elegance by using recursion to manage all the "back up and go forward" operations. Unfortunately, we know that recursive algorithms involve more overhead and are often slower in execution than iterative versions of the same algorithm.

As with many recursive algorithms we can replace the recursion in DFS with a loop, just by implementing a stack.

```
def DFS(v):
     create a stack S
     S.push(v)
     while not S.isEmpty():
          x = S.pop()
          if x is not visited:
               process x          # as in BFS this can represent
                                  # different operations
               for each neighbour y of x:
                    if y is not visited:
                         S.push(y)
               mark x "visited"
               # we can also process x after its neighbours,
               # rather than before its neighbours
```

You should confirm that this executes a proper Depth-First Search of a graph.

It's important to note that when we examine all the neighbours of a vertex x, there is no pre-set order in which we look at them. So in the graph shown above, we could list the neighbours of B as {A, D, C, F} or {C,A,F,D} etc. The DFS trees that we get from different orderings of the neighbours can be very very different. DFS trees starting from a particular vertex do not show the same "same set of vertices at level i" property that BFS trees starting from a particular vertex all share.

If you compare the BFS and DFS algorithms you will see that the only significant difference between them is the data structure – a queue for BFS and a stack for DFS. Since the basic operations on queues and stacks are all in O(1), this means that the over-all complexity is exactly the same for the two algorithms. This is a great illustration of the idea that knowing the properties of data structures can greatly simplify the analysis of algorithms.

It's also a great illustration of the power of choosing the right data structure – it's kind of magical that by changing the queue to a stack, the algorithm changes from doing a breadth-first search to a depth-first search.

We should note that both BFS and DFS can be used to determine if a graph is connected. BFS is also useful because (as we know) in an unweighted graph it finds the shortest paths from the start vertex to all other vertices. It would be really nice if we could claim that DFS finds

the longest paths from the start vertex to all other vertices … well we could claim that, but we would be wrong. DFS cannot guarantee that it will find either shortest or longest paths.

It turns out that DFS is the perfect graph-searching method for some really interesting problems … but they are outside the scope of this course. You will just have to live in suspense for a bit longer!