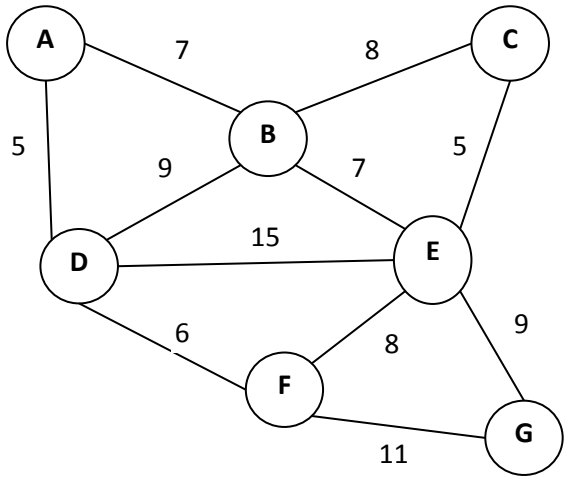
Introduction to Graph Algorithms

Consider the following weighted undirected (or bidirectional) graph. It could represent a number of different problems such as length or cost of optical fibre between network points, road distances between villages.



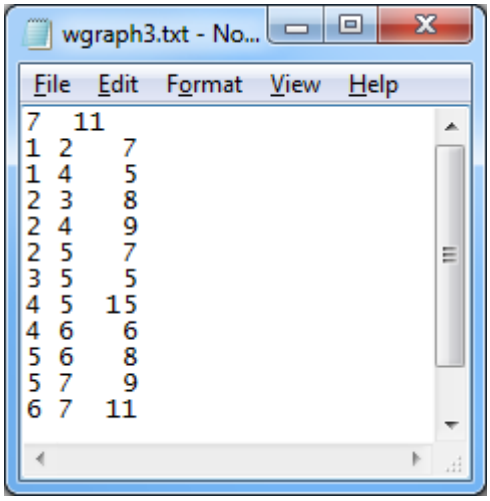
In computer science there are many useful algorithms that operate on a graph data structure such as:

- depth first search
- breadth first search
- shortest path tree
- minimum spanning tree
- Hamiltonian cycle
- Eulerian cycle

Before any of these are implemented we first need to consider how a graph such as the one above might be stored or represented in computer memory. Also it makes sense to save it on secondary storage as a text file such as:

wGraph3.txt

7	11	
1	2	7
1	4	5
2	3	8
2	4	9
2	5	7
3	5	5
4	5	15
4	6	6
5	6	8
5	7	9
6	7	11



Here the first row tells us there are 7 vertices and 11 edges. Second row says there is an edge from A to B with weight 7 and vice versa. Third row: edge from A to D weight 5 etc.

The simples way to store this graph in primary memory is to use a 2-D array called an adjacency matrix. We say “adjacency” because it records which vertex is next to or connected to any other vertex.

Adjacency Matrix

If we use the graph vertices as array indices and call the matrix **adj[,]** (or **adj[ ][ ]** if using C++ or Java style 2-D arrays) then:

> adj[u, v] represents the weight of the edge between vertices u and v. For example adj[A, B] or adj[1, 2] = 7 and adj[2, 1] = 7. If there is no edge we say adj[u, v] = 0.

		A	B	C	D	E	F	G
	0	1	2	3	4	5	6	7
	0							
A	1		7		5			
B	2	7		8	9	7		
C	3		8			5		
D	4	5	9			15	6	
E	5		7	5	15		8	9
F	6				6	8		11
G	7					9	11	

Here we have not shown all the matrix values. The complete matrix would be:

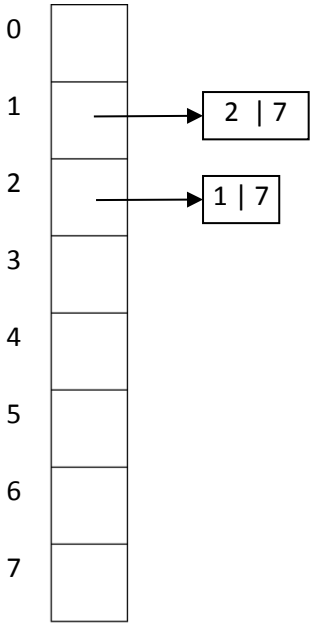
		A	B	C	D	E	F	G
	0	1	2	3	4	5	6	7
	0							
A	1	0	7	0	5	0	0	0
B	2	7	0	8	9	7	0	0
C	3	0	8	0	0	5	0	0
D	4	5	9	0	0	15	6	0
E	5	0	7	5	15	0	8	9
F	6	0	0	0	6	8	0	11
G	7	0	0	0	0	9	11	0

You can see that for a sparse graph most of the 2-D array would be occupied by 0's. So an adjacency matrix is extremely inefficient spacewise for a sparse graph, especially as V (number of vertices) increases. So for sparse graphs a linked list approach would be a more efficient use of memory as linked list nodes would only be allocated for edges that exist.

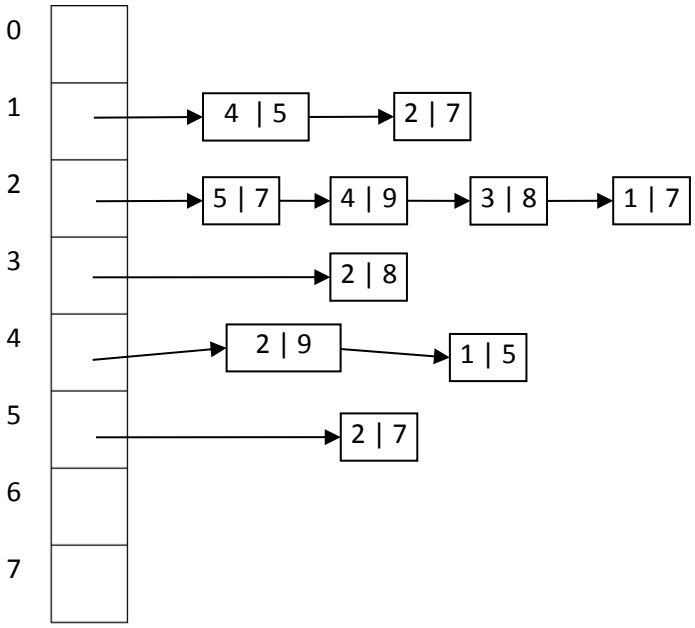
Adjacency Lists

In this approach every vertex has a linked list associated with it which only records those vertices connected to it with an edge. The edge weights are also recorded. So in a bidirectional graph, each edge is represented by two linked list nodes.

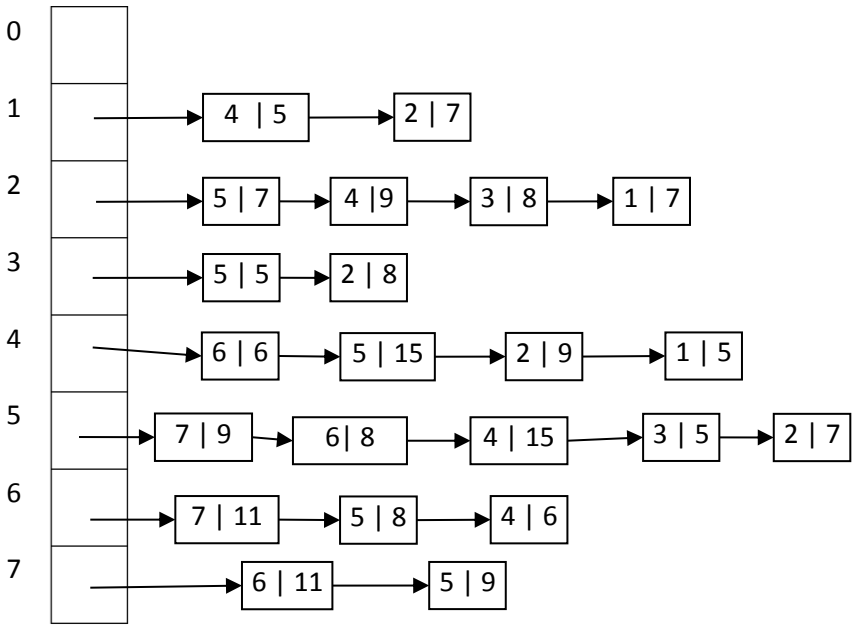
For example a graph constructor using adjacency lists on reading in the first edge from the above text file would have created an array of linked lists with the following logical structure:



And after reading 5 edges from the file:



On reading in all 11 edges:



Graph Search or Traversal

A graph is a complex data structure and can be viewed as a collection on which it may be desirable to do a traversal and process each vertex or edge in some way (similar to the way a **for** loop can be used to traverse an array). Three very useful traversal algorithms are:

- depth first, may use a stack or recursion
- breadth first, uses a queue
- best first, uses a priority queue or heap

Depth First Traversal

For a connected graph (no isolated nodes), the following algorithm will visit each vertex in a depth first fashion. If the graph is disconnected, it will only traverse 1 component of the graph. It is easily implemented using recursion. Alternatively a stack may be used for iterative implementation.

```
Graph :: DF( Vertex s)
Begin
    id = 0
    for v = 1 to V
        visited[v] = 0
    dfVisit(Null, s)
End

Graph :: dfVisit( Vertex prev, Vertex v)
Begin
   visited[v] = ++id
   print "Visited vertex ", v, " along edge ", prev, "—", v

   for each vertex u ∈ adj(v)
        if  not visited[u]
            dfVisit(u)
End
```

Exercise

For the example graph above, show the contents of the stack for each stage of DFS, assuming it uses iteration rather than recursion. Consider how the algorithm could be changed so that it uses a while loop rather than recursion.

Breadth First Traversal

```
Graph :: BF( Vertex s)
Begin
    Queue q
    id = 0
    for v = 1 to V
        visited[v] = 0

    q.enQueue(s)
     while (not q.isEmpty())
          v = q.deQueue()
          if (not visited[v])
              visited[v] = ++id
              for each vertex u ∈ adj(v)
                  if  (not visited[u])
                      q.enQueue(u)
              end for
          end if
     end while
End
```

Breadth First Traversal (BF) will give a shorted path spanning tree on an unweighted graph. In an unweighted graph, you can consider all edges to have an equal weight of 1. BF visits all the vertices on the same level before proceeding to a "deeper" level, so all vertices newly visited on a level will be the same distance from the starting vertex. Further, since BF proceeds level by level with distances of 0, 1, 2 etc., it will reach a vertex via the shortest path to that vertex.

If there was another path which was shorter to a vertex, then the vertex would also belong to another previous level closer the starting vertex; which of course is a contradiction since any level is exhausted before a deeper one is examined.

Best First Traversal

This approach attaches a value or priority to each vertex and for each iteration of the while loop, it 'visits' the vertex with the highest priority. The natural auxiliary data structure here is a heap. This method can be used in Minimum Spanning Tree and Shortest Path algorithms.

Iterative version of Depth First Traversal

This is nearly identical to breadth first traversal except a stack is used instead of a queue.

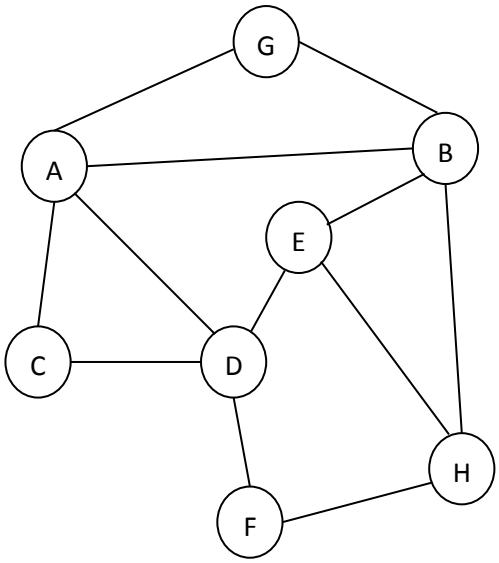
```
Graph :: DF_iter( Vertex s)
Begin
    Stack s
    id = 0
    for v = 1 to V
        visited[v] = 0

    s.push(s)
     while (not s.isEmpty())
          v = s.pop()
          if (not visited[v])
              visited[v] = ++id
              for each vertex u ∈ adj(v)
                  if  (not visited[u])
                      s.push(u)

     end while
End
```

Exercise

While showing the contents of a stack or queue, do a paper based DFS and BFS of the following graph.



Possible Solution

Blue – order of BFS
Red – order of DFS