Chapter 7 - Graphs

7.1 Introduction

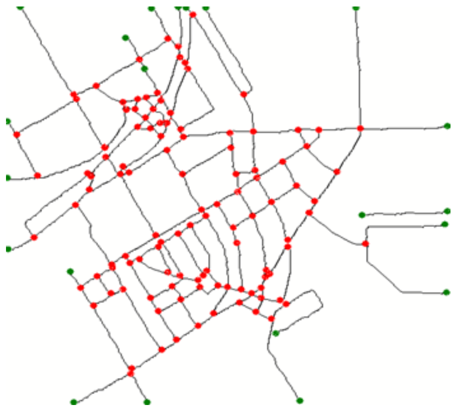


Figure 1: A graph for a road network[3]

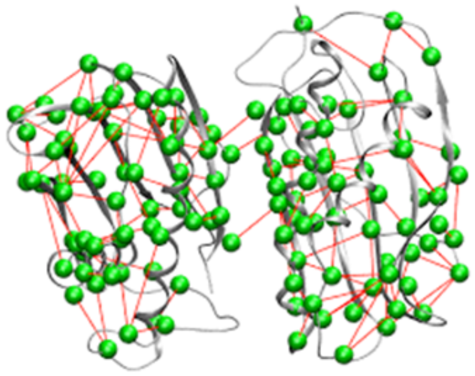
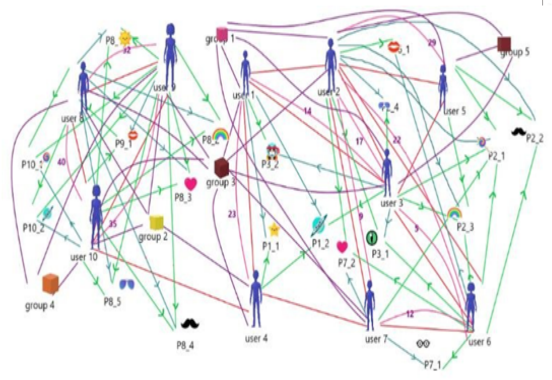


Figure 2: a graph showing a Protein Structure[4]



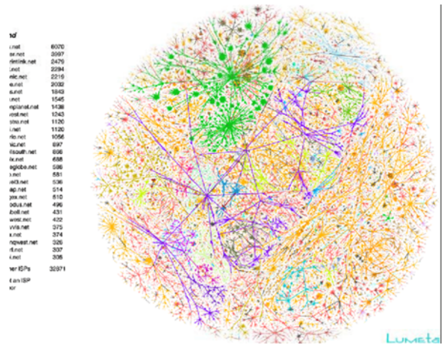


Figure 4: graph visualizing the internet[5]

Figure 3: A Facebook network[6]

So far, we have learned about data structures that can store linear sequences (e.g., arrays, lists, stack, queues) and data structures that can represent non-linear hierarchical relationships (i.e., different types of trees). What can be the most generic data structures that can represent arbitrary relationships among entities without any ordering restrictions? The answer is graphs. The graph is the most versatile data structure that can represent any relationship network among a collection of objects. For example, we can use graphs to represent road networks, the

[3] Image source: https://www.researchgate.net/figure/An-example-of-road-network-extraction-and-graph-representation_fig12_279263325
[4] Image source: https://benthamopen.com/contents/supplementary-material/TOBIOIJ-5-53_SD1.pdf
[5] Image source: https://www.researchgate.net/figure/2-A-graph-visualisation-of-the-topology-of-network-connections-of-the-core-of-the_fig2_239550496
[6] Image source: https://www.researchgate.net/publication/340347681_A_Study_on_Graph_Theory_Properties_of_On-line_Social_Networks

internet, structure of molecules and proteins, social networks, evolutionary relationships among species, geographic regions, and so on.

In fact, a single graph can capture multiple types of entities and relationships. Whenever we have a collection of entities and need to model their interactions and relations as a network, we can use graphs. Therefore, being able to store and use graphs in our programs is absolutely essential. Graphs are so important in computer science that graph theory and graph algorithms are considered core computer science courses and many books have been written on them. Even drawing a graph for visualization and human analysis is a vast topic taught as a separate course in many universities! Here we will only learn the basics related to graphs: the terminologies we have to understand, common graph data structure representations, and how to traverse a graph.

7.2 Graph Terminologies

A graph G is represented as a set of vertices (or nodes) V that represent the entities/objects/concepts and a set of edges (or links) E that represent relationship/interaction among those vertices. So, we typically write a graph G as G = (V, E). If a and b are two vertices in V and if there is an edge between them in the graph, we commonly write the edge as (a, b). When there is an edge (a,b) in E, we say vertex a and b are adjacent or neighbors. Note that the relationship a graph captures may be directional or undirected. In the former case we have a directed graph, in the latter case, an undirected graph. Below are examples of a directed and an undirected graph. We typically draw edges in an undirected graph as lines/curves and edges in a directed graph as arrows.

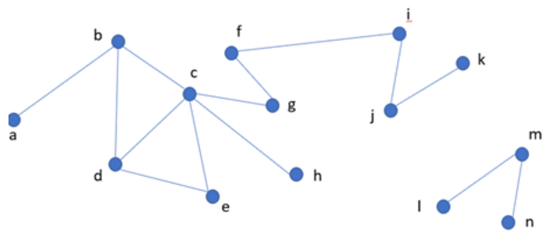


Figure 5: an undirected graph

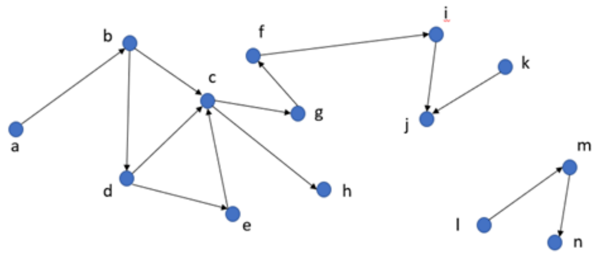


Figure 6: a directed graph

Notice that in a undirected graph having an edge (a,b) in E means the same thing as having the edge (b,a). However, in a directed graph (a,b) means an edge from a to b, while (b,a) means an edge from b to a. We can have none, either, or both in a directed graph. If we have both the edges, then we have to draw two arrows (one from a to b and another from b to a) when drawing the graph.

The **degree** of a vertex in an undirected graph is the number of edges the vertex has. For example, in Figure 5, degree of vertex d is 3. In a directed graph, a vertex has both in-degree and out-degree. The former represents the number of edges originating from other vertices and ending at that vertex, the latter represents the opposite. So, the in-degree of vertex d in Figure 6 is 1 and out-degree is 2. A directed edge (a,b) in E is an outgoing edge of a and an incoming edge of b.

A **path** between two vertices x and y in a graph is a sequence of edges of the form $(x = u_0, u_1),(u_1, u_2),(u_2,u_3),\ldots,(u_{n-1}, u_n=y)$. For example, the sequences of edges (a,b),(b,c),(c,e),(e,d) forms a path between vertex a and d in the graph of Figure 5. The length of the path is the number of edges in it. Note that, there may be zero, one, or more paths between any two vertices in a graph. When there is no path between two vertices u, v in a graph then the graph is called **disconnected**. Then we also say that u and v are unreachable from each other. The subset of vertices that are mutually reachable from one another along with the various edges form a **component** of a disconnected graph. Hence, there are two components in the graph of Figure 5.

Note that in the directed graph of Figure 6, there is no path (a,b),(b,c),(c,e),(e,d) as the edges are directional. However, vertex d is still reachable from a as we have the path (a,b),(b,d). Notice that, a is not reachable from d in this case, this is again because the edges are directional. The **distance** of a vertex v from another vertex u in a graph is the number of edges in the shortest path from u to v.

A **cycle** in a graph is a sequence of edges that starts and ends at the same vertex. For example in Figure 5, (b,d),(d,e),(e,c),(c,b) edge sequence form a cycle. In a directed graph, the cycles are also directional. Hence, there is no directed cycle in the graph of Figure 6.

In both directed and undirected cases, the vertices and/or the edges of the graph can have weights assigned to them. Then we call the graph a weighted directed/undirected graph. The weight assigned to a vertex is used to represent the importance of the entity/object/concept it represents. The weight assigned to an edge typically represents the cost or strength of the relationship among the vertices it connects. For example, in a road network graph, the vertices are road junctions and the edges are road segments. Then the weight of a junction can be the maximum traffic capacity at that junction and the weight of the edge between two junctions can be their distance, aka, the length of the road. Below are two examples of weighted graphs.

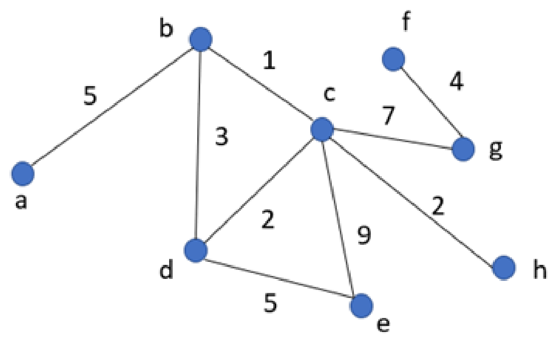


Figure 7: an edge weighted undirected graph

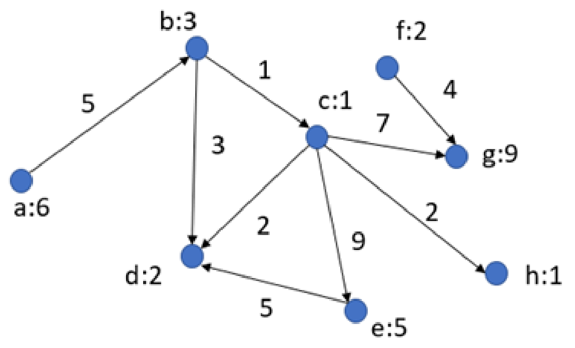


Figure 8: directed graph having both vertex and edge weights

We say a graph is sparse when the number of edges is too low compared to the number of vertices. We call it a dense graph when the number of edges is high. This distinction is important when we choose among various data structure representations of a graph.

There are many more definitions related to graphs. However, for our purpose, knowing up to this much is sufficient. Given an undirected, unweighted graph is the most common case, in the rest of this chapter we will use the term graph to mean an undirected, unweighted graph by default. When we will refer to the other cases, we mention them explicitly.

7.3 Graph Representations

Unlike a tree, a graph generally does not have a root vertex/node that we can use to recursively discover all other vertices. The reason for that is simple. As the graph may be disconnected, it may be impossible to explore the entire graph if we are given only a single starting vertex. Hence, in any typical graph representation we have all the vertices and edges given. The most common representations of a graph are the following:

1. Adjacency List Representation
2. Adjacency matrix representation, and
3. Incidence matrix representation

7.3.1 Adjacency List Representation

In this representation, the vertices of the graph are represented by the indices of an array. Each entry of the array is a linked list. The elements of the list contain indices of the vertices that are adjacent to the vertex owning the index.

Let us consider the graph of Figure 5, and assume vertices a to n are assigned successive increasing indices starting from 0. Then the adjacency list representation of the graph is as follows:

a	b	c	d	e	f	g	h	I	j	K	I	m	n
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0 2 3	1 3 4 6 7	1 2 4	2 3	6 8	2 5	2	5 9	8 10	9	12	11 13	12

Table 1: adjacency list representation of graph in Figure 5

It is quite easy to represent a directed graph using this format also. Everything remains the same, we just include the edge in the entry for the vertex where the edge starts from – not where it ends. Therefore, the adjacency list representation of the directed graph of Figure 6 is as follows:

a	b	c	d	e	f	g	h	I	j	K	I	m	n
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2 3	6 7	2 4	2	8	5		9		9	12	13	

Table 2: Adjacency list representation of the graph in Figure 6

When we have a weighted graph to represent then this plain array of list of numbers is not sufficient. Let us consider the most generic case, where both the vertices and the edges can have weights. We now have to construct an Edge class to hold all information about an edge and a separate array holding the node weights. Then the Edge class should look as follows:

```
Class Edge {
            int ep1
            int ep2
            Int weight
}
```

Here in the properties of the Edge class, ep1 and ep2 are the indices of the vertices that an edge connects. With this modification, we can represent the weighted directed graph of Figure 8 as follows.

Node Weight Array	a	b	c	d	e	f	g	h
	0	1	2	3	4	5	6	7

	6	3	1	2	5	2	9	1
Adjacency List	a	b	c	d	e	f	g	h
	0	1	2	3	4	5	6	7
	<0,1,5>	<1,2,1> <1,3,3>	<2,3,2> <2,4,9> <2,6,7> <2,7,2>		<4,3,5>	<5,6,4>		

Table 3: Adjacency list representation of the weighted directed graph of Figure 8

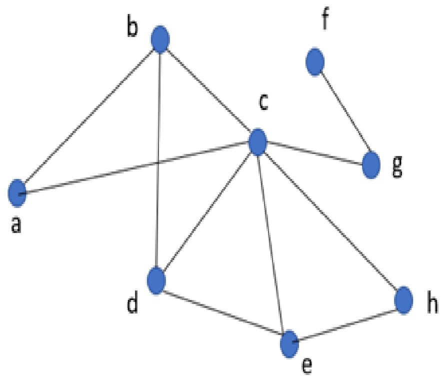
In the above, the tuple <x,y,z> stands for <ep1,ep2,weight> of an edge. Notice that in the directed weighted graph case, the edge class can record only the destination vertex of an edge; the source is not needed as it is the same as vertex owning the index of the array holding the list of edges. However, the way we defined the edge class, we can represent both directed and undirected weighted graphs using that class.

Finally, note that the vertices and edge can have other properties along with their weights in graph representations of real-world scenarios. In that case, we can define a Node class for the vertices and have an array of nodes and include more properties related to the edges in the Edge class. One can further, combine the two arrays by having a node class holding the list of edges along with a vertex's other properties.

Space Complexity: if a graph has N vertices and M edges, then the adjacency matrix representation needs an array of size N and the total number of nodes in various lists in different array indices will be total 2M (in case of undirected graph) or M (in case of a directed graph). Hence the space requirement for the adjacency list representation is proportional to N + M. In the asymptotic notation, we can write the space complexity as O(N+M).

7.3.2 Adjacency Matrix Representation

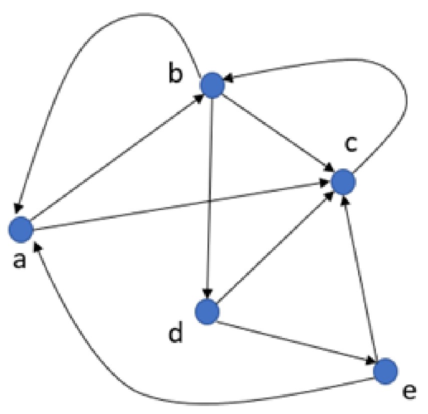
The adjacency matrix representation of a graph with N vertices uses an N×N matrix. As in the case of previous representation, vertices are assigned increasing indices starting from 0. The ith row and column of the matrix are for the ith vertex of the graph. The entry at ith row and jth column of the matrix is 1 if there is an edge between the corresponding vertices of the graph. Below is the drawing and adjacency matrix of a graph as an example.



	0	1	2	3	4	5	6	7
a 0	0	1	1	0	0	0	0	0
b 1	1	0	1	1	0	0	0	0
c 2	1	1	0	1	1	0	1	1
d 3	0	1	1	0	1	0	0	0
e 4	0	0	1	1	0	0	0	1
f 5	0	0	0	0	0	0	1	0
g 6	0	0	1	0	0	1	0	0
h 7	0	0	1	0	1	0	0	0

Table 4: an adjacency matrix representation of an undirected unweighted graph

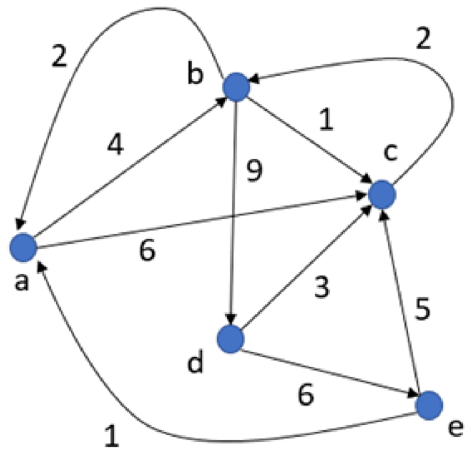
Notice that the matrix is symmetric. That is if we call the matrix A then each A[i][j] entry is the same as A[j][i] entry for all i,j < N. The reason for this is simple, as the edges are undirected, each edge occurs twice in the matrix. The adjacency matrix of a directed graph is not symmetric, unless of course when for each (a,b) edge of the graph there is also a (b,a) edge from vertex b to vertex a. Below is an adjacency matrix representation of a small directed graph.



	0	1	2	3	4
a 0	0	1	1	0	0
b 1	1	0	1	1	0
c 2	0	1	0	0	0
d 3	0	0	1	0	1
e 4	1	0	1	0	0

Table 5: an adjacency matrix representation of a directed graph

If a graph has only weight on its edges, then we can represent it using the adjacency matrix just as easily in both directed and undirected cases. If there is an edge (a,b) with weight w in the graph and the indices of a and b are i and j in the matrix respectively, then we simply write w in the <i,j> cell of the adjacency matrix, instead of writing 1. If the edge is undirected then we do the same in <j,i> cell also. Below is an adjacency representation when the earlier directed graph is edge-weighted (i.e., only the edges have weights).



	0	1	2	3	4
a 0	0	4	6	0	0
b 1	2	0	1	9	0
c 2	0	2	0	0	0
d 3	0	0	3	0	6
e 4	1	0	5	0	0

Table 6: adjacency matrix representation of a weighted directed graph

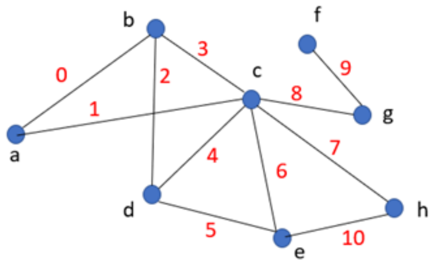
When both vertices and the edges of a graph have weights then we need a node weight array as in Table 3 to represents the weights of the vertices along with the adjacency matrix that captures the edge weights. Write the adjacency matrix representation of the graph of Figure 8 as an exercise of this case.

Space Complexity: for a graph with N vertices and M edges, the adjacency matrix representation will take N×N entries, that is, N^2 entries. Consequently, the asymptotic space complexity in this representation is $O(N^2)$. Apparently, the space cost of adjacency matrix representation is significantly higher than that of adjacency list representation. Then why we use this second representation? The answer is many computations are easier in the matrix representation than in the list representation. We trade space with time when using any representation.

7.3.3 Incidence Matrix Representation

The final commonly used representation for graphs is the incidence matrix representation. This representation is particularly useful and popularized by electrical circuit analysis where edges represent wires with resistance/inductance/capacitance and the vertices represents junctions where the edges connect and where voltage being applied. You would be surprised to know that this representation is more than 150 years old!

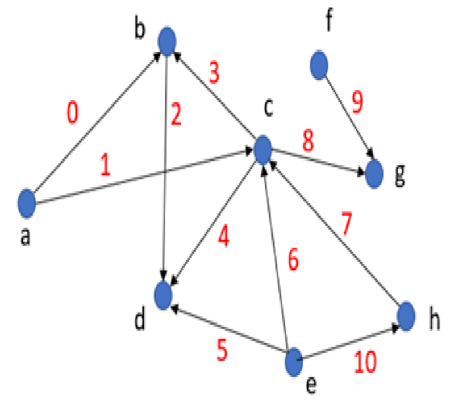
In the incidence matrix representation, the edges and vertices both are numbered. Thus, if there are M edges and N vertices in the graph, edges are numbered from 0 to M – 1 and vertices are from 0 to N – 1. Then a M×N matrix is used where the rows are the edges and the columns are the vertices. We write a 1 in the two columns of a row to indicate that these are the two vertices connected by the edge owning that row. The remaining entries of the row are filled with zeros. Below is an example (here the indices of the edges are shown in red color).



	a:0	b:1	c:2	d:3	e:4	f:5	g:6	h:7
0	1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0
2	0	1	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	1	1	0	0	0	0
5	0	0	0	1	1	0	0	0
6	0	0	1	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0
9	0	0	0	0	0	1	1	0
10	0	0	0	0	1	0	0	1

Table 7: The incidence matrix representation of an undirected graph

When we have a directed graph, we put a 1 in the source of the edge and a -1 in the destination (or vice versa). Below is an example.



	a:0	b:1	c:2	d:3	e:4	f:5	g:6	h:7
0	1	-1	0	0	0	0	0	0
1	1	0	-1	0	0	0	0	0
2	0	1	0	-1	0	0	0	0
3	0	-1	1	0	0	0	0	0
4	0	0	1	-1	0	0	0	0
5	0	0	0	-1	1	0	0	0
6	0	0	-1	0	1	0	0	0
7	0	0	-1	0	0	0	0	1
8	0	0	1	0	0	0	-1	0
9	0	0	0	0	0	1	-1	0
10	0	0	0	0	1	0	0	-1

Table 8: incidence matrix representation of a directed unweighted graph

You probably already understand how to represent an edge-weighted graph in this representation. Instead of writing 1 and -1, we have to put the positive and negative weight of the edge in the proper columns in a row.

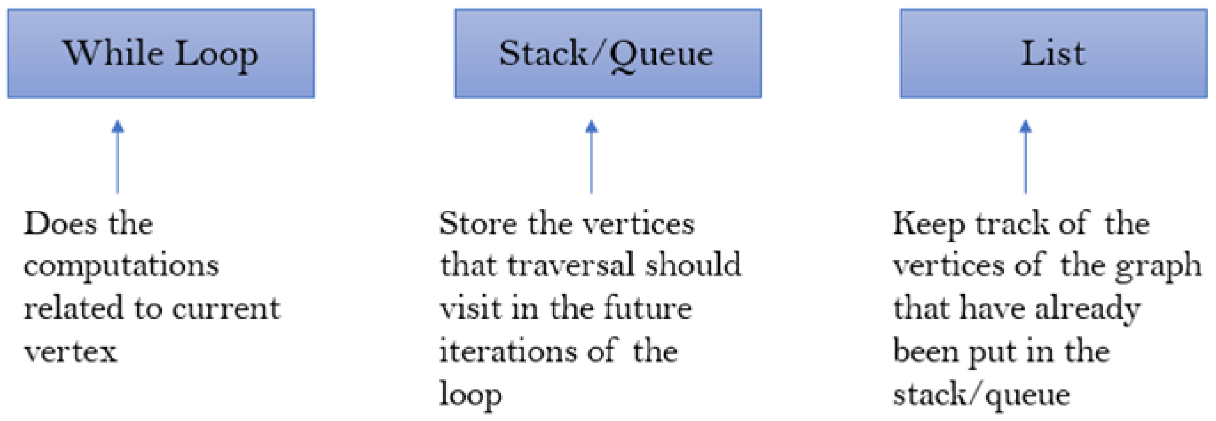
Space Complexity: for a graph with N vertices and M edges, the incidence matrix representation will require M×N entries. Consequently, the asymptotic space complexity in this representation is O(MN). When the underlying graph is sparse, the incidence matrix representation may be cheaper than the adjacency matrix representation or similar in cost. When the opposite is true, the adjacency matrix is better. Electrical circuits are typically sparse graphs, that is why, incidence matrix representation is quite popular in circuit analysis. However, there are algorithms that specifically need the incidence matrix representation for computation efficiency.

Practice Problems:

1. Given an unweighted graph in the adjacency matrix representation, calculates the number of edges the graph has.
2. Given a directed unweighted graph in the adjacency matrix representation, generate the adjacently list representation of the graph.
3. Solve the following problems for all three types of representation of the input graph.
    a. Given an undirected, unweighted graph as input, write a function to find the vertex with maximum degree and return the degree of that vertex.
    b. Given an undirected, edge-weighted graph as input, write a function to find the vertex whose sum of edge weights is maximum.
    c. Solve Problem #a and #b for directed, edge-weighted graph considering only outgoing edges.

7.4 Graph Traversals

To solve any interesting problem that involves graphs, one needs to know how to traverse a graph. When we traverse a graph, we start from an arbitrary vertex then follow its edges to reach its neighbor vertices, then follow their edges to their neighbors, and so on. Its sounds just like traversing a tree starting from a root node and following the children of a node in each step. Doesn't it? However, there are two important caveats. Since, unlike a tree, graph is not a hierarchical data structure, a recursive traversal can lead us back to an earlier vertex when the graph has cycles. When that happens, we will have an infinite recursion, that will keep rotating within the vertices of the first cycle it gets sucked into. The second concern is that, since a graph may be disconnected, a traversal starting from a particular vertex will only visits the vertices of the component where the starting vertex belongs to. For both these reasons, the straightforward recursive traversals that we used for trees do not apply for graphs. Rather, the recursive graph traversal is implemented using while loops and stack/queue and a list as supporting data structures to keep track of the progress of traversal. The following diagram explains the role of the different components of a graph traversal.


While Loop
Stack/Queue
List
Does the computations related to current vertex
Store the vertices that traversal should visit in the future iterations of the loop
Keep track of the vertices of the graph that have already been put in the stack/queue

The general approach is, we insert the starting vertex in the stack/queue and also in the list. Then we enter the while loop that will iterate until the stack/queue is empty. In each iteration of the loop, we will remove the top vertex from the stack/queue using the stack pop or queue dequeue method. We do the necessary computation using that vertex. Then we get the list of neighbors of that vertex and insert only those vertices in the stack/queue that are not in the list. As we insert a vertex in the stack/queue, we also insert that vertex in the list.

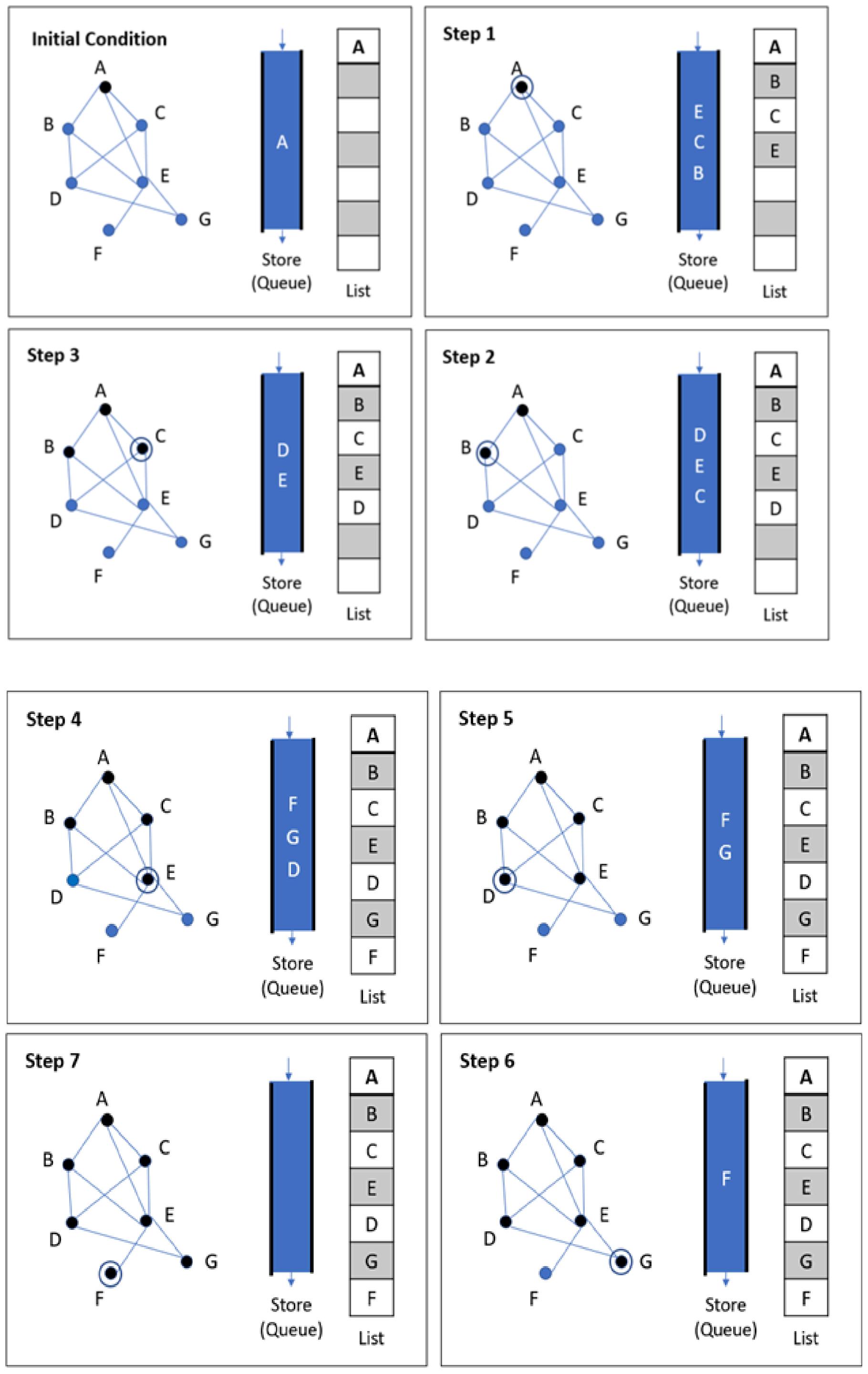
The above approach will visit all the vertices of the graph that are reachable from the starting vertex through some path. When dealing with a disconnected graph, we have to do one more thing. That is, after the while loop ends, we need to check if all vertices of the input graph are included in the list where we keep track of the visited vertices. If we find a vertex that is not already in the list, then we start the whole process of the previous paragraph by using that vertex as the starting point.

Below is a pseudo-code of the graph traversal process.

```
Algorithm Traverse-Graph(G)
    visited := new List
    store := new Queue/Stack
    V := G.vertices
    start := V[0]
    while (start != nill) {
    store.insert(start)
    visited.insert(start)
    while(store.isEmpty() == FALSE) {
            u := store.removeNext()
            // do computation related to u
            edges := u.edges
            foreach (e in edges) {
                        if (e.ep1 == u) {
                                v := e.ep2
                } else {
                        v := e.ep1
                        }
                        If (visited.contains(v) == FALSE) {
                                visited.append(v)
                                store.insert(v)
                        }
            }
        }
        start = nill
        foreach (v in V) {
            If (visited.contains(v) == FALSE) {
                start := v
                break
            }
        }
}
```

7.4.2 Breadth First and Depth First Traversals

Interestingly, we have two very different graph exploration patterns depending on whether we used a stack or a queue in the aforementioned traversal algorithms. Let us examine the behavior of the graph traversal for a small graph and a queue as the store of vertices to be visited next, as in Figure 9. For simplicities sake, we use an example of a connected graph.



Notice that in this given a queue is a first come first out data structure, the item that is oldest in the queue comes out when we invoke the remove operation in the traversal algorithm.

As a result, if we start the traversal from vertex A then the next three vertices that are visited are the three neighbors B,C, and E of A.

We can explain the traversal progression as follows: starting from A, all vertices that are at distance 1 from A is visited next (B, C, and E). Then vertices that are at distance 2 from A are visited (D and G). Finally, the vertices that are at distance 3 from A are visited (F).

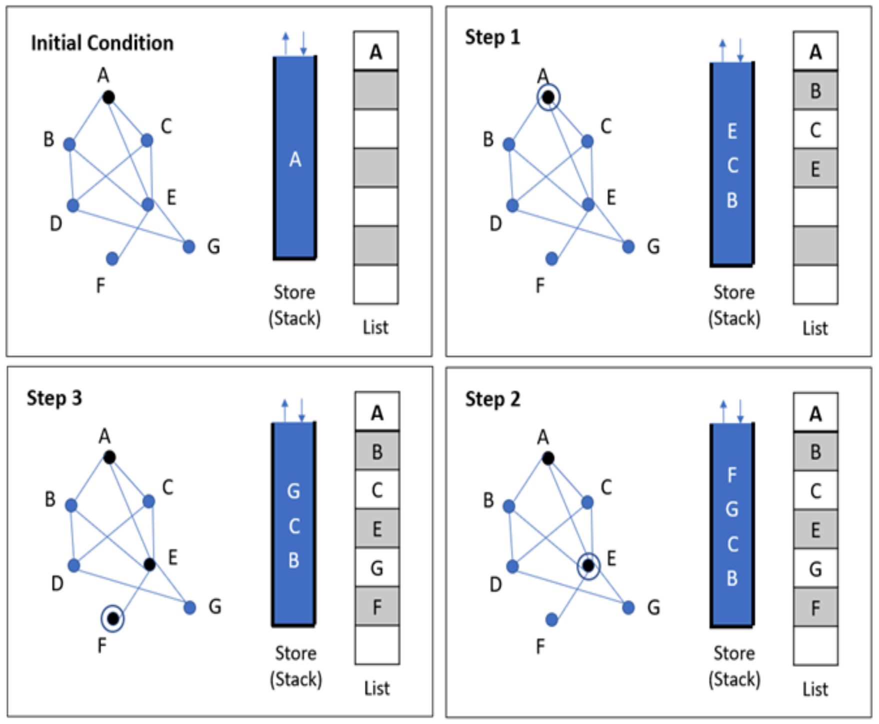
As if we start from the center A and expanding our search within the breadth of successively increasing distances.

This is why this traversal process is called **Breadth First Search** or **BFS**.

Figure 9: Breadth First Graph Traversal of a Graph

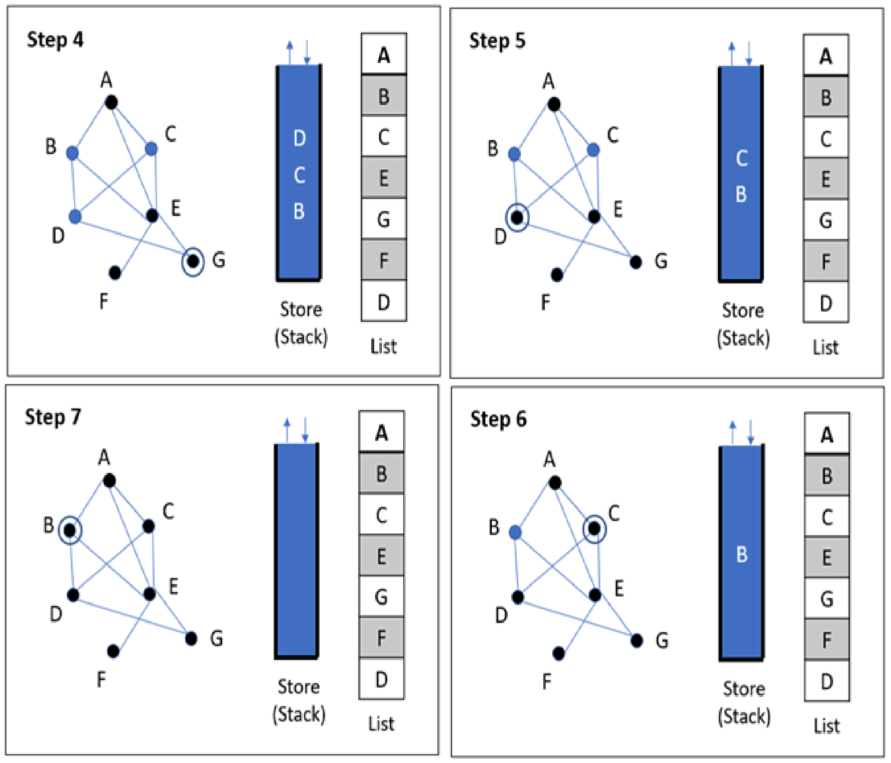
The algorithm for graph traversal visits each vertex only once and when visiting it, it has to evaluate all its edges. For an undirected graph, each edge is evaluated twice as the edge is considered from the both endpoints. Hence, if a graph has n vertices and m edges, the traversal itself takes n + 2m steps. However, in each step, checking whether the other endpoint of an edge is in the visited list take n steps in the worst case. So the running time of the traversal algorithm is $O(n + 2m + n^2) = O(m + n^2)$. However, if we use a Boolean array instead of the visited list for tracking the already considered vertices, then checking if a vertex needs to be put in the store takes a constant time. Then running time of traversal becomes $O(n + m)$.

If we rather use a stack for the store then the exploration pattern would be as shown in Figure 10.



Notice that starting from A, the traversal algorithm inserts the neighbors of A in the store in the order B, C, E as in the BFS case. However, given the store is a stack, which is a last come first serve data structure, the vertex that comes out from the store in the next iteration is E.

Progressing from E, the neighbors G and F are inserted in the store. F comes out from the store in the next iteration. Since F has no neighbor that are not in the visited list, the next iteration pops G from the store. Which puts D in the store.



So, we see that the traversal process is as such that, in each iteration, it goes one distance farther from the starting vertex following some path. When it cannot move any farther, the traversal backtracks to the previous step and try another path from there.

This is why this traversal process is called **Depth First Search** or **DFS**.

Figure 10: Depth First Traversal of a Graph