Introduction to Graphs

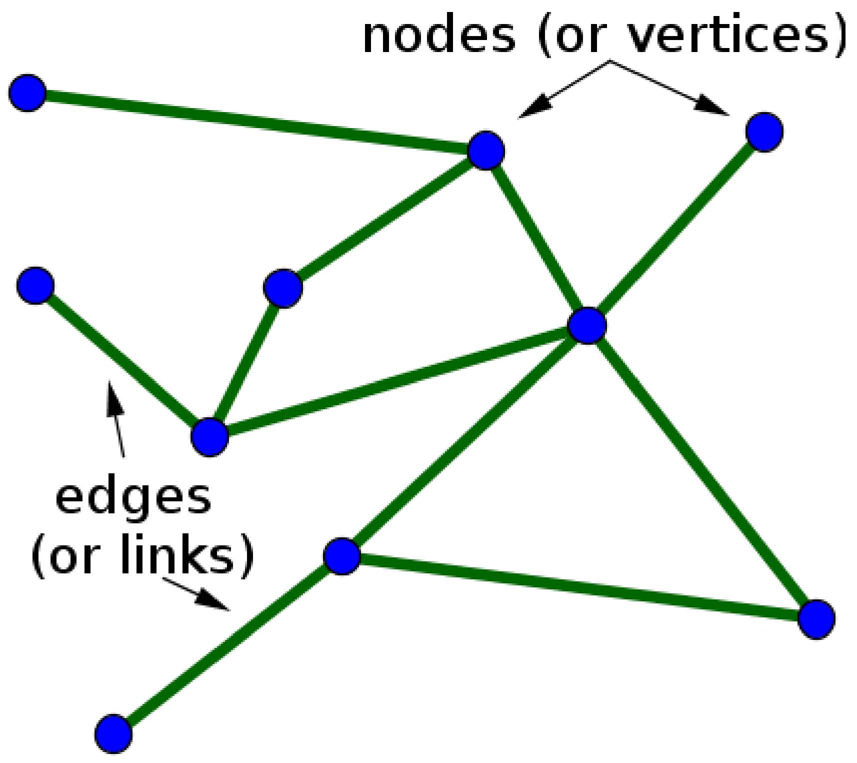
The objective of this short introduction is to provide a basic introduction about graphs and the commonly used algorithms used for traversing the graph, Breadth First Search (BFS) and Depth First Search (DFS).

What is a Graph?

Graphs are one of the most interesting data structures in computer science. Graphs and the trees are somewhat similar by their structure. In fact, tree is derived from the graph data structure. However there are two important differences between trees and graphs.

1. Unlike trees, in graphs, a node can have many "parents.".
2. The link between the nodes may have values or weights.

Graphs are good in modelling real world problems like representing cities which are connected by roads and finding the paths between cities, modelling air traffic controller system, etc. These kinds of problems are hard to represent using simple tree structures. The following example shows a very simple graph:



The edges can be directed edges which are shown by arrows; they can also be weighted edges in which some numbers are assigned to them. Hence, a graph can be a directed/undirected and weighted/un-weighted graph. In this, we will discuss undirected and un- weighted graphs.

Note: There is no actual hierarchy associated with graphs. Therefore the drawing above is just an arbitrary representation – it could be re-drawn any number of ways (as long as the nodes and edges remain consistent).

Examples of systems that could be modelled using graphs.

GRAPH	VERTICES	EDGES
Communication	telephones, computers	fiber optic cable
Circuits	gates, registers, processors	wires
Mechanical	joints	rods, beams, springs
Hydraulic	reservoirs, pumping stations	pipelines
Financial	stocks, currency	transactions
Transportation	street intersections, airports	highways, air routes
Scheduling	tasks	precedence constraints
Software systems	functions	function calls
Internet	web pages	hyperlinks
Games	board positions	legal moves
Social networks	people, actors, terrorists	friendships, movie casts, associations
Protein interaction networks	proteins	protein-protein interactions
Genetic regulatory networks	genes	regulatory interactions
Neural networks	neurons	synapses
Infectious disease	people	infections
Electrical power grid	transmission stations	cable
Chemical compounds	molecules	chemical bonds

Graph Representation in Programming Language

Every graph has two components, Nodes and Edges. Let’s see how these two components are implemented in a programming language like JAVA.

1. Nodes

Nodes are implemented by class, structures or as Link-List nodes. As an example in JAVA, we will represent node for the above graph as follows:

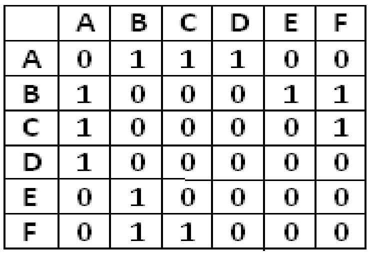
```
Class Node
{
   char label;
   public Node(char l)
   {
      this.label=l;
   }
}
//
```

2. Edges

Edges represent the connection between nodes. Below you are presented with two ways to represent edges.

Adjacency Matrix

It is a two dimensional array with Boolean flags. As an example, we can represent the edges for the above graph using the following adjacency matrix.



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	0	1
D	1	0	0	0	0	0
E	0	1	0	0	0	0
F	0	1	1	0	0	0

In the given graph, A is connected with B, C and D nodes, so adjacency matrix will have 1s in the ‘A’ row for the ‘B’, ‘C’ and ‘D’ column.

The advantages of representing the edges using adjacency matrix are:

1. Simplicity in implementation as you need a 2-dimensional array
2. Creating edges/removing edges is also easy as you need to update the Booleans

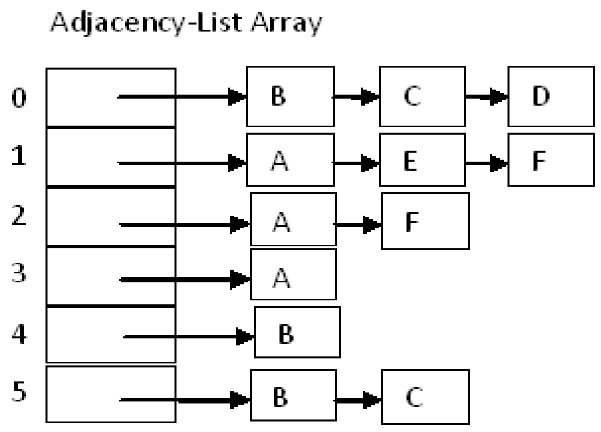
The drawbacks of using the adjacency matrix are:

1. Increased memory as you need to declare N*N matrix where N is the total number of nodes.
2. Redundancy of information, i.e. to represent an edge between A to B and B to A, it requires to set two Boolean flag in an adjacency matrix.

In JAVA, we can represent the adjacency matrix as a 2 dimensional array of integers/Booleans.

Adjacency List

It is an array of linked list nodes. In other words, it is like a list whose elements are a linked list. For the given graph example, the edges will be represented by the below adjacency list:

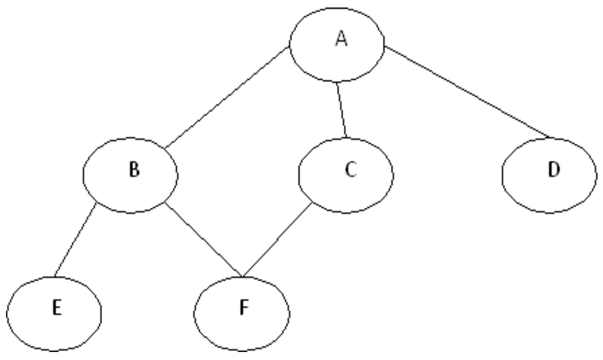


Graph Traversal

The breadth first search (BFS) and the depth first search (DFS) are the two main algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.

Depth First Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node[1]. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph:



As stated before, in DFS, nodes are visited by going as deep as possible from some designated starting node. Assuming we "mark" **A** as the start node, if we do the depth first traversal of the above graph and print the visited node, it will be "A B E F C D". DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

1. **Step 1**: Push the start node onto the Stack.
2. **Step 2**: Loop until stack is empty.
3. **Step 3**: Peek the node of the stack.
4. **Step 4**: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
5. **Step 5**: If the node does not have any unvisited child nodes, pop the node from the stack.

Based upon the above steps, the following Java code shows the implementation of the DFS algorithm:

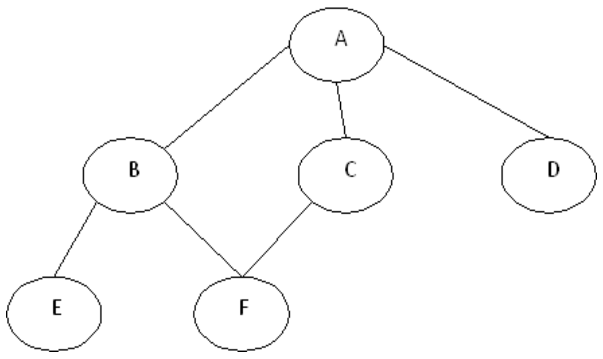
[1] The concept of a root node needs to be explained.
In a graph there is no hierarchy, so the idea of a root/start node is slightly different. Basically ***any*** node can be designated as a root/start node (it is entirely arbitrary).

```
//
public void dfs()
{
        //DFS uses Stack data structure
        Stack s=new Stack();
        s.push(this.startNode);
        startNode.visited=true;
        printNode(startNode);
        while(!s.isEmpty())
        {
                Node n=(Node)s.peek();
                Node child=getUnvisitedChildNode(n);
                if(child!=null)
                {
                        child.visited=true;
                        printNode(child);
                        s.push(child);
                }
                else
                {
                        s.pop();
                }
        }
        //Clear visited property of nodes
        clearNodes();
}

//
```

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the start node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph (again, assuming "A" is the start node and print the visited node as the output, it will print the following output. "A B C D E F". The BFS visits the nodes level by level.

Algorithmic Steps

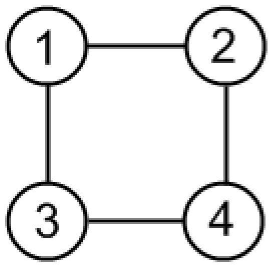
1. **Step 1**: Put the start node in the Queue.
2. **Step 2**: Loop until the queue is empty.
3. **Step 3**: Remove the node from the Queue.
4. **Step 4**: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Based upon the above steps, the following Java code shows the implementation of the BFS algorithm:

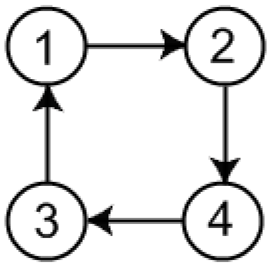
```java
//
public void bfs()
{
         //BFS uses Queue data structure
         Queue q=new LinkedList();
         q.add(this.startNode);
         printNode(this.startNode);
         startNode.visited=true;
         while(!q.isEmpty())
         {
                  Node n=(Node)q.remove();
                  Node child=null;
                  while((child=getUnvisitedChildNode(n))!=null)
                  {
                           child.visited=true;
                           printNode(child);
                           q.add(child);
                  }
         }
         //Clear visited property of nodes
         clearNodes();
}
//
```

Appendix: Some graph definitions

All graphs are divided into two big groups: directed and undirected graphs. The difference is that edges in directed graphs, called *arcs*, have a direction. Edge can be drawn as a line. If a graph is directed, each line has an arrow.



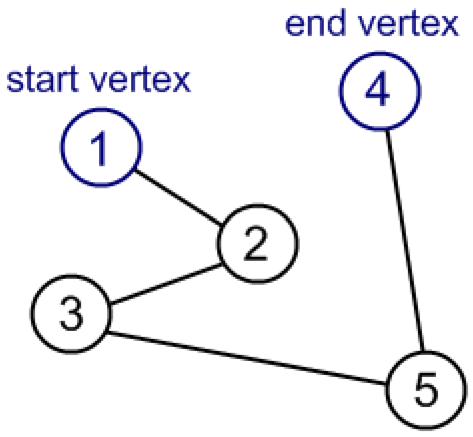
undirected graph



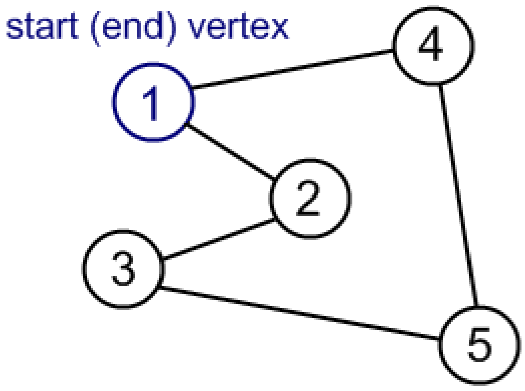
directed graph

Now, we present some basic graph definitions.

- Sequence of vertices, such that there is an edge from each vertex to the next in sequence, is called **path**. First vertex in the path is called the *start vertex*; the last vertex in the path is called the *end vertex*. If start and end vertices are the same, path is called **cycle**. Path is called *simple,* if it includes every vertex only once. Cycle is called *simple*, if it includes every vertex, except start (end) one, only once. Let's see examples of path and cycle.
- Note: we are not concerned with cycles in this module.

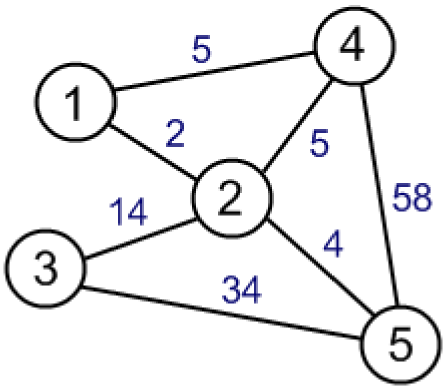


path (simple)



cycle (simple)

The last definition we give here is a weighted graph. Graph is called *weighted*, if every edge is associated with a real number, called edge weight. For instance, in the road network example, weight of each road may be its length or minimal time needed to drive along.



weighted graph