Graph Theory

Please read through the tutorial in the given link side by side of this tutorial.
HackerEarth Graph Theory Tutorial 1.

Ten everyday scenarios with an underlying application of graph theory:
1.Using your GPS or Google Maps/Yahoo Maps, to determine a route based on user settings (quickest route/shortest route) or finding the cheapest airfare between two destinations. The destinations are vertices and their connections are edges containing information such as distance or airfare. The software finds the critical path (optimal route) based on the user settings.
2.Connecting with friends via social media or a video going viral. Each user is a vertex, and when users connect they create an edge. Videos are known to be viral when they have reached a certain number of connections/views.
3.School Districts developing bus routes to pick up students to deliver to school. Each stop is a vertex and the route is an edge. A Hamiltonian path represents the efficiency of including every vertex in the route.
4.The working of traffic lights; turning green and timing between lights. The use of vertex coloring graphs to solve conflicts of time and space and identifying the chromatic number for the number of cycles needed.
5.Planning and processing the preparation of a meal. The use of PERT graphs to plan a course of action for projects.
6.Using Google to search for webpages. Pages on the internet are linked to each other by hyperlinks; each page is a vertex and the link between two pages is an edge. PageRank and Googlebot are used algorithms to aid the connectivity process.
7.Shopping on Amazon or movies on Netflix. Relationship graphs are used to make recommendations for future shopping or films.
8.City planning to put salt on the roads when ice develops. Euler paths or circuits are used to traverse the streets in the most efficient way.
9.Visiting a zoo, water park or theme park and wanting to see certain attractions or devise an efficient route to see all of the attractions. A Hamiltonian path or circuit contains every vertex in the graph.
10.Examination on the spread of viruses/diseases. Vertex-edge graphs provide a visual for the network connection of those affected by the virus.

Depth first search

DFS is one of the most fundamental Algorithms used in various fields of Computer science. DFS is a graph traversal algorithm which is used for searching, path finding in a graph etc. The depth first search is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph.

There are many use cases for depth-first search such as:
Topological sorting of directed graphs
Detecting a cycle
Finding connected components in a sparse graph
Finding biconnectivity in graphs.
Detection of Planar Graphs.

Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
Maze generation may use a randomized depth-first search.

It is frequently used in Artificial Intelligence Planning problems, Game Playing AI.

Algorithm
Depth first search is a recursive algorithm that uses the idea of backtracking. Basically, it involves exhaustive searching of all the nodes by going ahead - if it is possible, otherwise it will backtrack. By backtrack, here we mean that when we do not get any further node in the current path then we move back to the node,from where we can find the further nodes to traverse. In other words, we will continue visiting nodes as soon as we find an unvisited node on the current path and when current path is completely traversed we will select the next path.

DFS keeps walking down a path until it is forced to backtrack. It backtracks until it finds a new path to go down.

It can be implemented using Recursion or stack.
Prefer Recursion.
Please refer Hackerearth tutorial to the for C++ code.

Pseudo code:(Recursive implementation)

```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

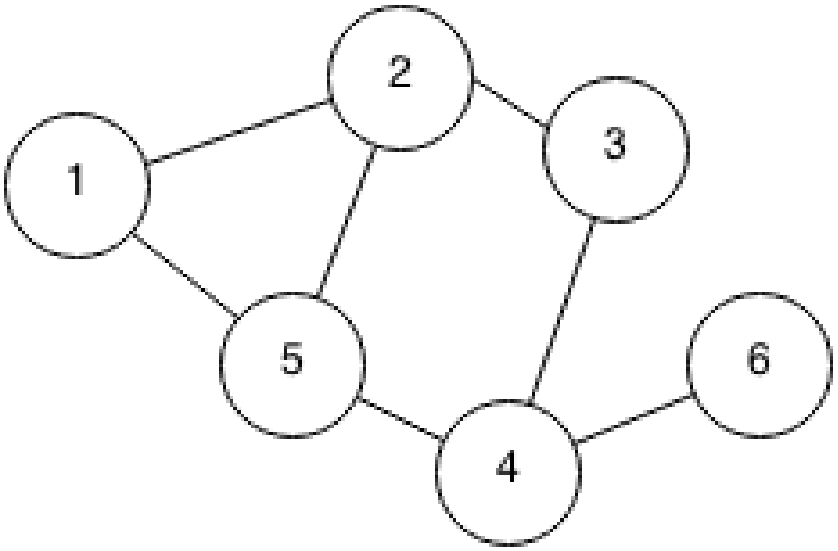
Imagine the following real life scenario,
Suppose a professor gives you an assignment and tells you to give the assignment to everyone and then collect the completed assignments from everyone(Tough job right?).
Assume the student community is an undirected connected graph.
The professor just called DFS(you).

You are Lazy and you decide that I will complete my assignment only after collecting all assignments from my friends.
By collect I mean first give him the assignment wait for him to complete and then collect it back.
Now You assign each of them to collect assignments from their friends and so on…

So you call DFS(v) for each v who is your friend (only if he hasn't already submitted).
The important catch is : **Before telling your other friend you wait for your current assigned friend to return his assignment and of all those from whom he collected.**
And your friends also thinks in the same way and hence pass on the DFS() call.

Each call to DFS(u) returns to its caller only when DFS(v) returns, where v are all nodes which are reachable from u and were not visited ever before u was visited.

Bazinga!! Recursion!



Let us assume you are node 1. Now
DFS(1) calls DFS(2)
DFS(2) calls DFS(3)
DFS(3) calls DFS(4)
DFS(4) calls DFS(6)
DFS(6) has no one else to call, He completes his assignment and submits it to 4.(returns to caller)
DFS(4) calls DFS(5)
DFS(6) has no one else to call, He completes his assignment and submits it to 4.(returns to caller)
DFS(4) has completed its job and now submits assignment to 3
3 Returns to 2
2 Returns to 1
Student 1, i.e you get everyone's assignment now and u submit them with your own assignment to the professor.
And DFS Saves your Day!

Now in the same example let us observe another Important property of DFS.
Let us store the start time and end time of each student.
Start time indicates when the DFS(u) was spawned(called and given the assignment by caller), end time indicates when the DFS(u) returned to its caller.
In our example, end time - start time will let us know how much time each student got to complete his assignment.

Pseudo code.

```
Int time=0;(Global)
  DFS-recursive(G, s):
     Start[s] = t;
     t++;
     mark s as visited
     for all neighbours w of s in Graph G:
        if w is not visited:
           DFS-recursive(G, w)
     End[s] = t;
```

Parenthesis Property

For every two vertices u and v , exactly one of the following conditions holds:

- The interval [Start[u],End[u]] , [Start[v],End[v]] are disjoint.
- One interval contains the other
    - Either start[u] < start[v] < end[v] < end[u]
    - Or start[v] < start[u] < end[u] < end[v]

Think about it!

Breadth first Search

It's a traversing algorithm, where we start traversing from selected node (source or starting node) and traverse the graph layerwise which means it explores the neighbour nodes (nodes which are directly connected to source node) and then move towards the next level neighbour nodes. As the name suggests, we move in breadth of the graph, i.e., we move horizontally first and visit all the nodes of the current layer and then we move to the next layer.

This is the Basis for Shortest Path Algorithm(Djikstra).
Applications:
1) Peer to Peer Networks: In peer to peer networks, Breadth First Search is used to find all neighbour nodes.
2) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.
3) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

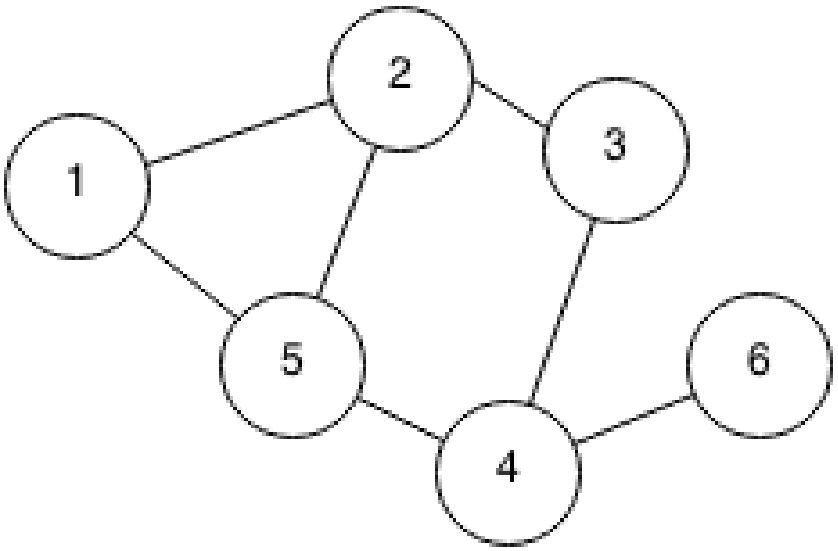
Please refer Hackerearth tutorial to the for C++ code.

```
BFS (G, s)                //where G is graph and s is source node.
 let Q be queue.
 Q.enqueue( s ) // inserting s in queue until all its neighbour vertices are marked.

 mark s as visited.
 while ( Q is not empty)
      // removing that vertex from queue,whose neighbour will be visited now.
      v  =  Q.dequeue( )

     //processing all the neighbours of v
```

```
for all neighbours w of v in Graph G
    if w is not visited
            Q.enqueue( w )          //stores w in Q to further visit its neighbour
            mark w as visited.
```



BFS(1)
Visiting order:
1 -> 2 -> 5 -> 3 -> 4 -> 6
It works following the rule that if node u is visited before node v then immediate neighbours of u are also visited before that of v.(FIFO)
We can see that since 2 was visited before 5, 3 was visited before 4.

Think about it!

I would recommend to try out your own graphs and visualize them to get a better understanding. Here Visualize.

These Algorithms are Easy but fundamental for any Computer science student!