**Introduction Of Graph Data Structure**

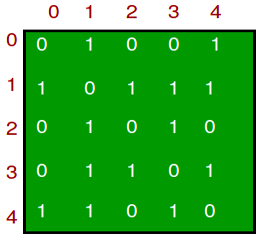
A graph is a data structure that consists of the following two components:   
**1. A finite set of vertices also called as nodes.**  
**2.** **A finite set of ordered pair of the form (u, v) called as edge**. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.  
Following is an example of an undirected graph with 5 vertices. 



The following two are the most commonly used representations of a graph.   
**1.** **Adjacency Matrix**   
**2.** **Adjacency List**   
There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

**Adjacency Matrix:**   
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w. 

The adjacency matrix for the above example graph is:



***Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).  
*Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.**   
  
**Adjacency List:**   
An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

Lightbox

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/AdjacencyMatrix.java>

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/AdjacenyListGraph.java>

1. **Breadth First Search or BFS for a Graph**

[Breadth-First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.



<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/BFS.java>

1. **Depth First Search or DFS for a Graph**

[Depth First Traversal (or Search)](http://en.wikipedia.org/wiki/Depth-first_search) for a graph is similar to [Depth First Traversal of a tree.](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) The only catch here is, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array.

**Algorithm:**   
Create a recursive function that takes the index of the node and a visited array.

1. Mark the current node as visited and print the node.
2. Traverse all the adjacent unvisited nodes and call the recursive function with the index of the adjacent node.

**Implementation:**

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/DFS.java>

**Applications of Depth First Search**

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.

**1) Detecting cycle in a graph**  
A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

**2) Path Finding**   
We can specialize the DFS algorithm to find a path between two given vertices u and z.   
i) Call DFS(G, u) with u as the start vertex.   
ii) Use a stack S to keep track of the path between the start vertex and the current vertex.   
iii) As soon as destination vertex z is encountered, return the path as the   
contents of the stack

**3)**[**Topological Sorting**](https://www.geeksforgeeks.org/topological-sorting/)   
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

**4) To test if a graph is**[**bipartite**](http://en.wikipedia.org/wiki/Bipartite_graph)   
We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn’t link two vertices of the same color. The first vertex in any connected component can be red or black!

**5) Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

**6) 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.)

**Applications of Breadth First Traversal**

**1) Shortest Path and Minimum Spanning Tree for unweighted graph** In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

**2) Peer to Peer Networks.** In Peer to Peer Networks like [BitTorrent](https://www.geeksforgeeks.org/how-bittorrent-works/), Breadth First Search is used to find all neighbor nodes.

**3) Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

**4) Social Networking Websites:**In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

**5) GPS Navigation systems:** Breadth First Search is used to find all neighbouring locations.

**6) Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

**7) In Garbage Collection:** Breadth First Search is used in copying garbage collection using [Cheney’s algorithm](http://en.wikipedia.org/wiki/Cheney%27s_algorithm). Breadth First Search is preferred over Depth First Search because of better locality of reference:

**8)**[**Cycle detection in undirected graph:**](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/) In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use [BFS to detect cycle in a directed graph](https://www.geeksforgeeks.org/detect-cycle-in-a-directed-graph-using-bfs/) also,

**9)** [**Ford–Fulkerson algorithm**](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).

**10)**[**To test if a graph is Bipartite**](https://www.geeksforgeeks.org/bipartite-graph/) We can either use Breadth First or Depth First Traversal.

**11) Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

**12) Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Many algorithms like [Prim’s Minimum Spanning Tree](https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/) and [Dijkstra’s Single Source Shortest Path](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/) use structure similar to Breadth First Search.

There can be many more applications as Breadth First Search is one of the core algorithms for Graphs.

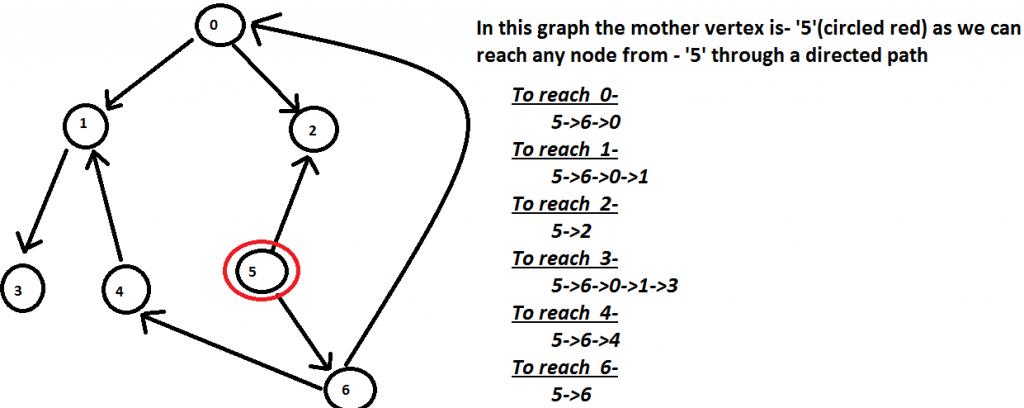
1. **Find a Mother Vertex in a Graph**

**What is a Mother Vertex?**   
A mother vertex in a graph G = (V, E) is a vertex v such that all other vertices in G can be reached by a path from v.

Example :

Input : Below Graph

Output : 5



**How to find mother vertex?**

* ***Case 1:- Undirected Connected Graph :***In this case, all the vertices are mother vertices as we can reach to all the other nodes in the graph.
* ***Case 2:- Undirected/Directed Disconnected Graph*** : In this case, there is no mother vertices as we cannot reach to all the other nodes in the graph.
* ***Case 3:- Directed Connected Graph*** : In this case, we have to find a vertex -v in the graph such that we can reach to all the other nodes in the graph through a directed path.

**Algorithm:**

1. Do DFS traversal for all the non-visited nodes, and get the last node assume it is v

If there is any mother vertex then v must be the one or one of them.

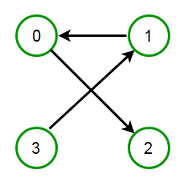
1. Mark all the nodes to non-visited
2. Do the DFS traversal considering source node as v
3. And then check if all nodes are visited or not…if it is visited then return v otherwise -1.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/FindMotherVertex.java>

1. **Transitive Closure of a Graph using DFS**

Given a directed graph, find out if a vertex v is reachable from another vertex u for all vertex pairs (u, v) in the given graph. Here reachable means that there is a path from vertex u to v. The reach-ability matrix is called transitive closure of a graph.

For example, consider the following directed graph:



**Its connectivity matrix C is**  
   
1   0   1   0  
1   1   1   0  
0   0   1   0  
1   1   1   1

The value of tc[i][j] is 1 only if a directed path exists from vertex i to vertex j. Note that all diagonal elements in the connectivity matrix are 1 since a path exists from every vertex to itself.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/TransitiveClosure.java>

1. **Count the number of nodes at given level in a tree using BFS.**

Given a tree represented as an undirected graph. Count the number of nodes at a given level l. It may be assumed that vertex 0 is the root of the tree.

Input : 7

0 1

0 2

1 3

1 4

1 5

2 6

2

Output : 4

In this code, while visiting each node, the level of that node is set with an increment in the level of its parent node i.e., level[child] = level[parent] + 1. This is how the level of each node is determined. The root node lies at level zero in the tree.

**Explanation :**

0 Level 0

/ \

1 2 Level 1

/ |\ |

3 4 5 6 Level 2

**Algorithm:**

**1) do the BFS traversal, level of root node is always 0**

**2) find the level of all the child nodes, level of child node is level[child] = level[parent] + 1**

**3) count the nodes at given level**

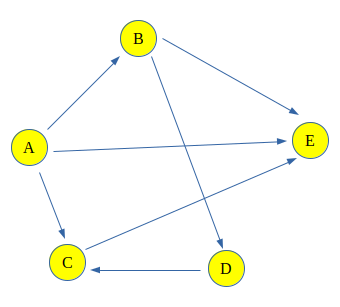
<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/CountNodesOfBinaryTreeAtGivenLevel.java>

1. **Count all possible paths between two vertices**

Count the total number of ways or paths that exist between two vertices in a directed graph. These paths don’t contain a cycle, the simple enough reason is that a cycle contains an infinite number of paths and hence they create a problem.

**Examples:**

**For the following Graph:**



**Input:** Count paths between A and E

**Output :** Total paths between A and E are 4

**Explanation:** The 4 paths between A and E are:

A -> E

A -> B -> E

A -> C -> E

A -> B -> D -> C -> E

**Algorithm:**

1. Create a recursive function that takes source node, destination node and count. Keep a global or a static variable count to store the count
2. If the current nodes is the destination increase the count.
3. Else for all the adjacent nodes, i.e. nodes that are accessible from the current node, call the recursive function with the index of adjacent node and the destination and store the returned value to the count.
4. Print the Count.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/CountPossiblePathBetweenTwoVertex.java>

1. **Count number of trees in a forest**

Given n nodes of a forest (collection of trees), find the number of trees in the forest.  
**Examples :**

Input : edges[] = {0, 1}, {0, 2}, {3, 4}

Output : 2

Explanation : There are 2 trees

0 3

/ \ \

1 2 4

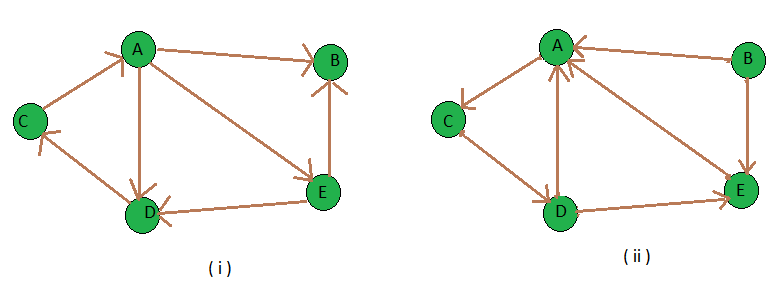
**Algorithm:**   
1. Apply DFS on every non visited node.   
2. Increment count by one if every connected node is visited from one source.   
3. Count will give the number of trees in forest.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/CountNoOfTreesInForest.java>

1. **Transpose graph**

[Transpose](https://en.wikipedia.org/wiki/transposeGraph) of a directed graph G is another directed graph on the same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in G. That is, if G contains an edge (u, v) then the converse/transpose/reverse of G contains an edge (v, u) and vice versa.  
Given a [graph (represented as adjacency list)](https://www.geeksforgeeks.org/graph-and-its-representations/), we need to find another graph which is the transpose of the given graph.

**Example:**



Input : figure (i) is the input graph.

Output : figure (ii) is the transpose graph of the given graph.

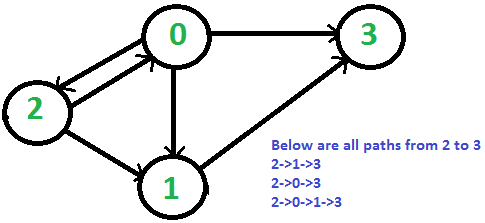
We traverse the adjacency list and as we find a vertex v in the adjacency list of vertex u which indicates an edge from u to v in main graph, we just add an edge from v to u in the transpose graph.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/Transpose.java>

1. **Print all paths from a given source to a destination using BFS**

Given a directed graph, a source vertex ‘src’ and a destination vertex ‘dst’, print all paths from given ‘src’ to ‘dst’.

Consider the following directed graph. Let the src be 2 and dst be 3. There are 3 different paths from 2 to 3.



**Algorithm :**

1) create a queue of type list which will store path

2) initialise the queue with first path starting from src

3) Now run a loop till queue is not empty

4) get the front most path from queue

a) check if the last node of this path is destination

if true then print the path

run a loop for all the vertices connected to the current vertex i.e. last node extracted from path

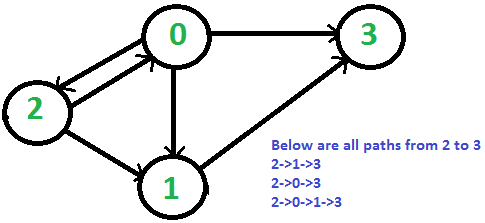
if the vertex is not there in current path

a) create a new path from earlier path and append this vertex

b) insert this new path to queue

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/PrintAllPathsFromSourceToDestinationUsingBFS.java>

**10) Print all paths from a given source to a destination using DFS**

Given a directed graph, a source vertex ‘s’ and a destination vertex ‘d’, print all paths from given ‘s’ to ‘d’.   
Consider the following directed graph. Let the s be 2 and d be 3. There are 3 different paths from 2 to 3.  
 

**Algorithm.**

1. Create a queue of list which stores paths
2. Add the source into path.
3. Now call the recursive method with param src, dest, visited, path
4. In the recursive method check, If the destination vertex is reached, print contents of path[].
5. Mark the source node as visited
6. Get all the adj node of the source and if it is not visited then add into the path and do the recursive call.
7. And after recursive call remove the current node from the path
8. And in the last mark the visited node as false.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/PrintAllPathsFromSourceToDestinationUsingDFS.java>

**11) Water Jug problem using BFS**

You are given a m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.

(X, Y) corresponds to a state where X refers to amount of water in Jug1 and Y refers to amount of water in Jug2   
Determine the path from initial state (xi, yi) to final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

**Examples:**

Input : 4 3 2

Output : {(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2)}

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/introduction/WaterJugProblemUsingBFS.java>