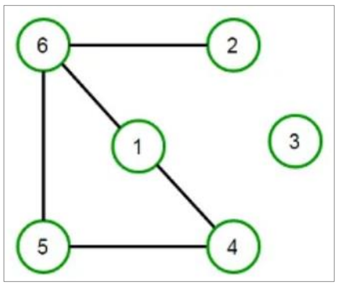
GRAPH DATA STRUCTURE AND ALGORITHMS - 1

Graph Terminology

What is a Graph?

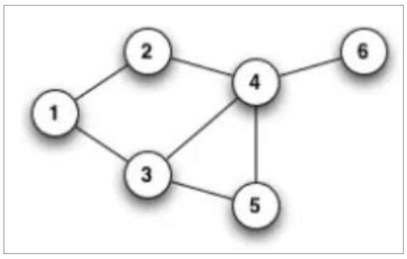
A graph G = (V, E) is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the **links** that connect the vertices are called edges.



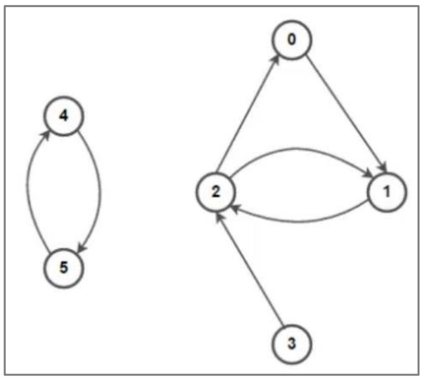
```
V = {1, 2, 3, 4, 5, 6}

E = {(1, 4), (1, 6), (2, 6), (4, 5), (5, 6)}
```

Types of Graphs

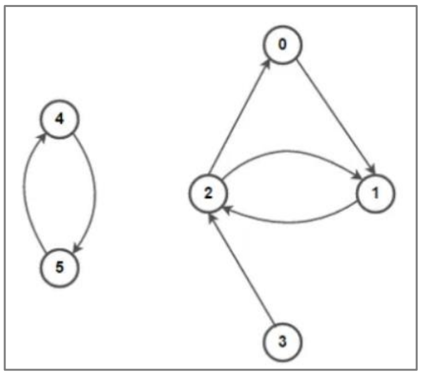
Undirected Graph: Edge (a, b) is identical with the edge (b, a).

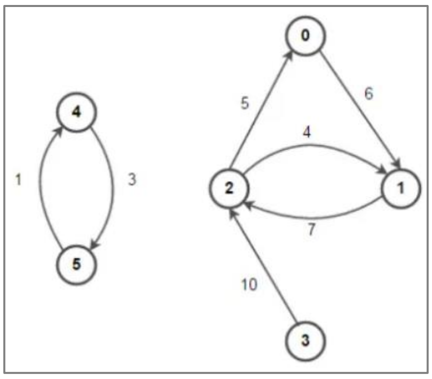


Directed Graph: Edges have directions. Edge (a, b) is not the same with the edge (b, a).

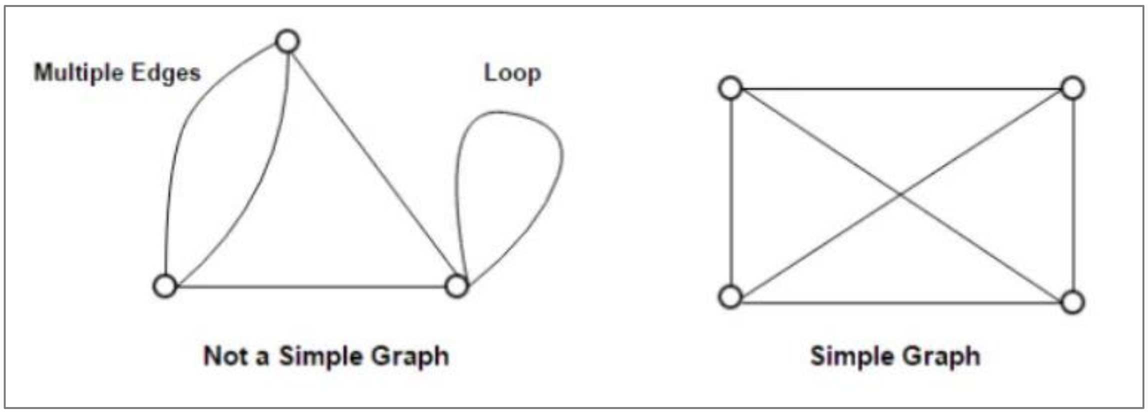


Weighted and Unweighted Graphs: In a weighted graph, each edge has a value.

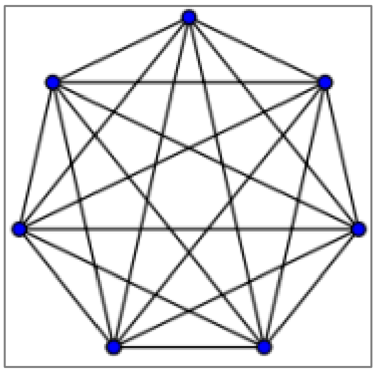




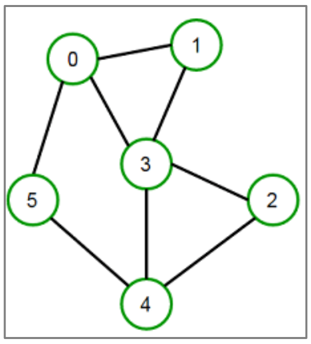
Simple Graph: There are no parallel edges (multiple edges) or loop (self-connected edge) in a sample graph. If a graph is not a simple graph it is called as **multigraph**.



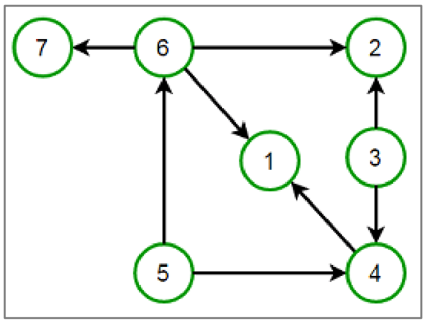
Complete Graph: In a complete graph, there is a direct connection between any two vertices. A simple complete graphs with N vertices has N*(N-1)/2 edges.



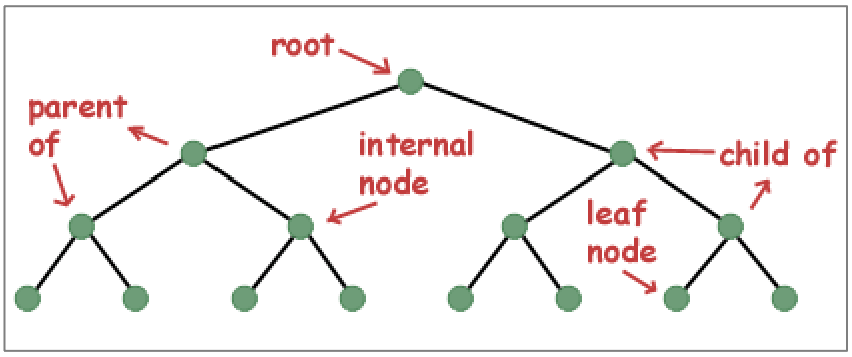
Connected Graph: There is a path between any two vertices in a connected graph. If a graph is not connected, it is called as **disconnected graph**.



Directed Acyclic Graph (DAG): It is a directed graph with no cycle.



Tree: It is an undirected graph with no cycle. Any tree with N nodes has N-1 edges.



Most Commonly Used Term in Graphs

- An **edge** is (together with vertices) one of the two basic units out of which graphs are constructed. Each edge has two vertices to which it is attached, called its **endpoints**.
- Two vertices are called **adjacent** if they are endpoints of the same edge.
- **Outgoing edges** of a vertex are directed edges that the vertex is the origin.
- **Incoming edges** of a vertex are directed edges that the vertex is the destination.
- The **degree** of a vertex in a graph is the number of edges incident to it.
- In a directed graph, **outdegree** of a vertex is the number of outgoing edges from it and **indegree** is the number of incoming edges.

- A vertex with indegree zero is called a source vertex, while a vertex with outdegree zero is called sink vertex.
- An isolated vertex is a vertex with degree zero; that is, a vertex that is not an endpoint of any edge.
- **Path** is a sequence of alternating vetches and edges such that each successive vertex is connected by the edge.
- **Cycle** is a path that starts and end at the same vertex.
- **Simple path** is a path with distinct vertices.

- A graph is **Strongly Connected** if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v.
- A directed graph is called **Weakly Connected** if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. The vertices in a weakly connected graph have either outdegree or indegree of at least 1.
- **Connected component** is the maximal connected sub-graph of a unconnected graph.
- A **bridge** is an edge whose removal would disconnect the graph.
- **Forest** is a graph without cycles.
- **Tree** is a connected graph with no cycles. If we remove all the cycles from DAG(Directed acyclic graph) it becomes tree and if we remove any edge in a tree it becomes forest.
- **Spanning tree** of an undirected graph is a subgraph that is a tree which includes all of the vertices of the graph.

Graph Representations

Graphs commonly represented in three different ways:

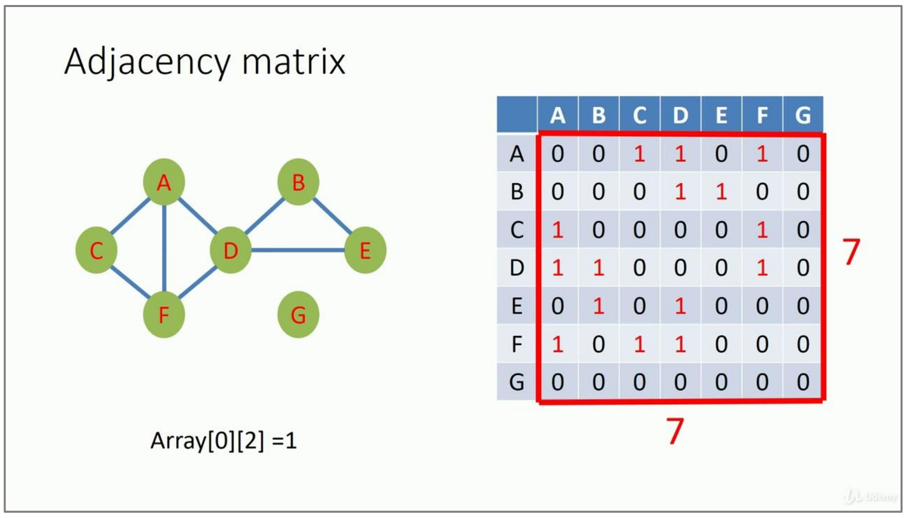
- Adjacency Matrix
- Adjacency List or Adjacency Set
- Edge List

Adjacency Matrix

An adjacency matrix is a **VxV** binary matrix **A**. Element Ai,j is 1 if there is an edge from vertex i to vertex j else Ai,j is 0. It is symmetric for undirected graphs.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in Ai,j, the weight or cost of the edge will be stored.

Adjacency matrix provides **constant time access (O(1))** to determine if there is an edge between two nodes. **O(V)** time to access all incidents of a vertex. Space complexity of the adjacency matrix is **O(V^2).**



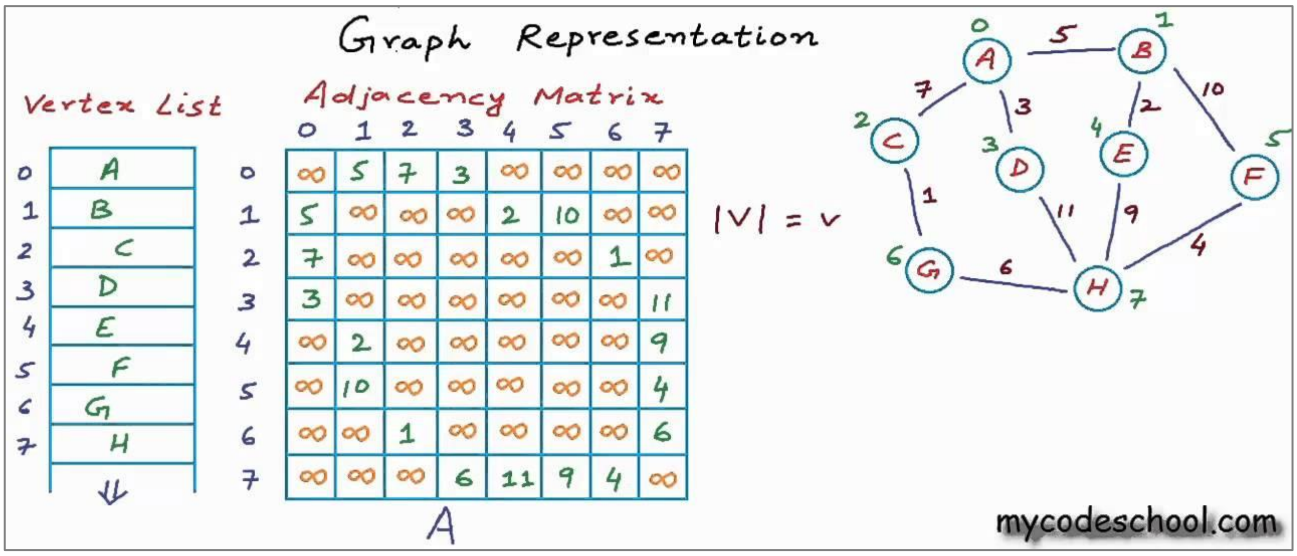
Adjacency matrix

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

7

7

Array[0][2] =1



Graph Representation

Vertex List

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

Adjacency Matrix

	0	1	2	3	4	5	6	7
0	∞	5	7	3	∞	∞	∞	∞
1	5	∞	∞	∞	2	10	∞	∞
2	7	∞	∞	∞	∞	∞	1	∞
3	3	∞	∞	∞	∞	∞	∞	11
4	∞	2	∞	∞	∞	∞	∞	9
5	∞	10	∞	∞	∞	∞	∞	4
6	∞	∞	1	∞	∞	∞	∞	6
7	∞	∞	∞	6	11	9	4	∞

A

$|V| = v$

Adjacency List

An adjacency list is an array A of separate lists. Each element of the array **A**$_i$ is a list, which contains all the vertices that are adjacent to vertex i.

The space complexity of adjacency list is **O(V + E)** because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful.

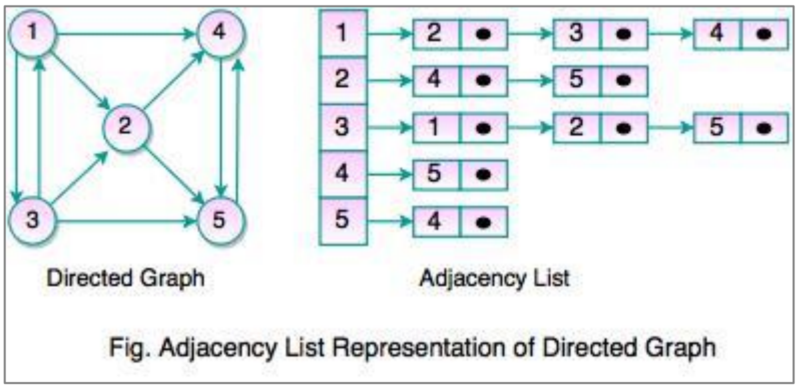
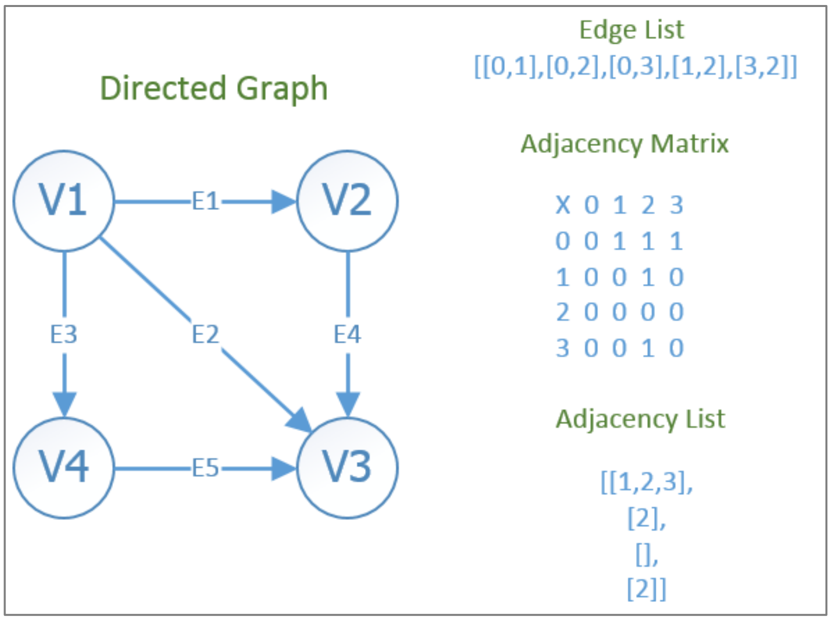


Fig. Adjacency List Representation of Directed Graph



Sample Problem 1: Adjacent vertices

Given graph G (V, E) and a source node S, find the vertices in the graph that are directly connected to the source.

Code 1: Solution with adjacency matrix

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
	//# of vertices, edges and the source node
	int N = 6, M = 9, S = 3;
	vector<pair<int, int>> edges = { {0,3}, {3,3}, {1,2}, {4,5}, {2,2}, {3,1}, {1,3}, {0,2},
	                                  {1,0} };

	//Declare and initialize the adjacency matrix
	vector<vector<int>> adjMat(N, vector<int>(N, 0));

	//Set the connections in the adjMat
	for (auto e : edges)
		adjMat[e.first][e.second] = adjMat[e.second][e.first] = 1;

	//print result
	for (int i = 0; i < N; i++)
		if (adjMat[S][i] == 1 && i != S)
			cout << i << " ";
}
```

Output

```
0 1
```

Code 1: Solution with adjacency set

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

int main()
{
	//# of vertices, edges and the source node
	int N = 6, M = 9, S = 3;
	vector<pair<int, int>> edges = { {0,3}, {3,3}, {1,2}, {4,5}, {2,2}, {3,1}, {1,3}, {0,2},
	                                 {1,0} };

	//Declare and initialize the adjacency set
	vector<set<int>> adjSet(N);

	//Set the connections in the adjList
	for (auto e : edges)
	{
		adjSet [e.first].insert(e.second);
		adjSet [e.second].insert(e.first);
	}

	//print result
	for (auto x : adjSet [S])
		if (x != S)
			cout << x << " ";
}
```

Output

```
0 1
```

Graph Traversal (Graph Search)

Graph traversal refers to the process of visiting each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Depth-first Search (**DFS**) and Breadth-first Search (**BFS**) are main traversal methods.

Graph traversal may require that some vertices be visited more than once, since it is not necessarily known before transitioning to a vertex that it has already been explored. As graphs become dense, this redundancy becomes more prevalent, causing computation time to increase; as graphs become more sparse, the opposite holds true. Thus, it is usually necessary to remember which vertices have already been explored by the algorithm, so that vertices are revisited as infrequently as possible (or in the worst case, to prevent the traversal from continuing indefinitely).

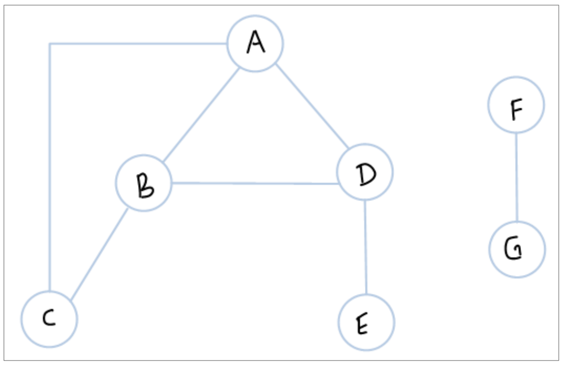
If each vertex in a graph is to be traversed by DFS or BFS, then the algorithm must be called at least once for each connected component of the graph. This is easily accomplished by iterating through all the vertices of the graph, performing the algorithm on each vertex that is still unvisited when examined.

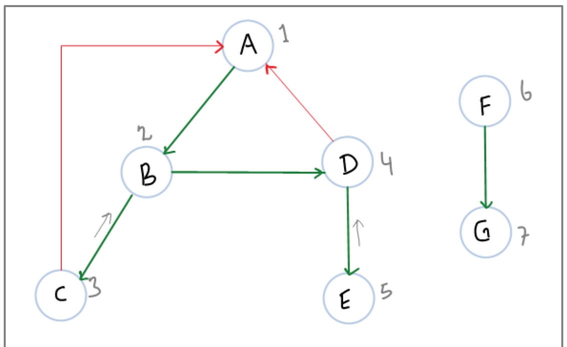
DFS

The DFS algorithm is a **recursive algorithm** that uses the idea of **backtracking**. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Time complexity is O(V + E).

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

An Undirected Graph and its DFS Tree





DFS tree edges (green) and back edges (red)

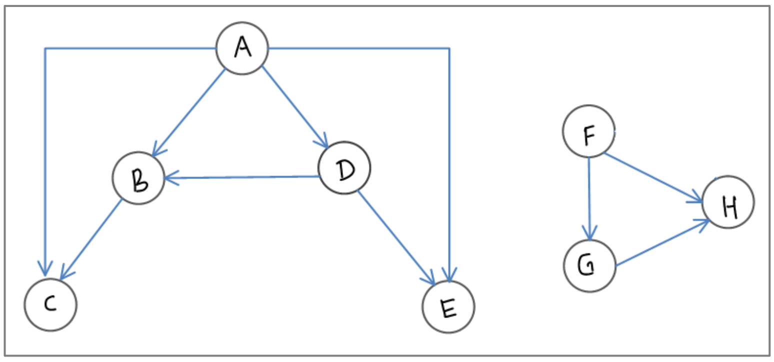
```
// DFS Undirected Graph.cpp

#include <iostream>
#include <vector>

using namespace std;

int N = 7, M = 7;
vector<pair<int, int>> edgeList = { {0,1}, {0,3}, {1,3}, {2,0}, {3,4}, {5,6}, {1,2} };
vector<vector<int>> adjList;
vector<bool>visited;
//-------------------------------------
// Construct the adjacency list from the edge list.
void createAdjList()
{
	adjList.resize(N);
	for (auto edge : edgeList)
	{
		adjList[edge.first].push_back(edge.second);
		adjList[edge.second].push_back(edge.first);
	}
}
//-------------------------------------
//Print node u, set it as visited and
//explorer all unvisited neighbours of u.
```

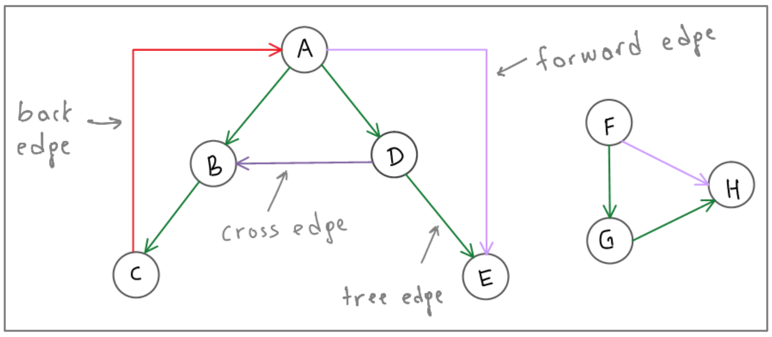
```
void dfsUtil(int u)
{
        cout << u << " ";
        visited[u] = true;
        for (auto v : adjList[u])
                if (!visited[v])
                        dfsUtil(v);
}
//-------------------------------------
//Start a new DFS search for each component
void DFS()
{
        visited.resize(N, false);
        for (int i = 0; i < N; i++)
                if (!visited[i])
                        dfsUtil(i);
}
//-------------------------------------
int main()
{
        createAdjList();
        DFS();
}
```

Output

```
0 1 2 3 4 5 6
```

A Directed Graph and its DFS Tree





```
// DFS Directed Graph.cpp

#include <iostream>
#include <vector>

using namespace std;

int N = 8, M = 10;
vector<pair<int, int>> edgeList = { {0,1}, {0,3}, {1,2}, {2,0}, {3,1}, {3,4}, {0,4}, {5,6},
{5,7}, {6,7} };
vector<vector<int>> adjList;
vector<bool>visited;
```

```
//---------------------------------------
// Construct the adjacency list from the edge list.
void createAdjList()
{
	adjList.resize(N);
	for (auto edge : edgeList)
	{
		adjList[edge.first].push_back(edge.second);
	}
}
//---------------------------------------
//Print node u, set it as visited and
//explorer all unvisited neighbours of u.
void dfsUtil(int u)
{
	cout << u << " ";
	visited[u] = true;
	for (auto v : adjList[u])
		if (!visited[v])
			dfsUtil(v);
}
//---------------------------------------
//Start a new DFS search for each component
void DFS()
{
	visited.resize(N, false);
	for (int i = 0; i < N; i++)
		if (!visited[i])
			dfsUtil(i);
}
//---------------------------------------
int main()
{
	createAdjList();
	DFS();
}
```

Output

```
0 1 2 3 4 5 6 7
```

Some Applications of DFS

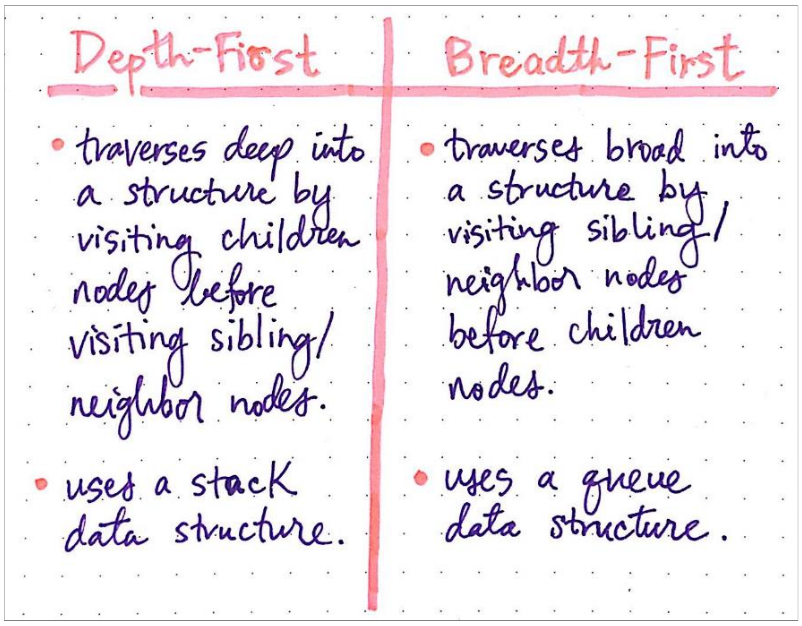
- Cycle detection
- Testing if a graph is bipartite
- Finding all the paths between two nodes
- Finding connected components of a graph
- Finding a spanning tree (MST) of an unweighted graph
- Topological sorting of an directed acyclic graph
- Finding depth/height of each node in a tree
- Calculating size of each subtree in a tree
- *Finding Strongly Connected Components of a di-graph*
- *Finding articulation points (cut vertices)*
- *Finding bridges (cut edges)*
- *Finding augmenting path in a flow network*

(Italic topics are usually asked in Platinum division of the USACO contest)

BFS

Breadth-first search (**BFS**) is an algorithm for traversing or searching graph data structures. It starts at the some arbitrary node of a graph and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. Level by level traversal in O(V + E) time.

It uses the opposite strategy as depth-first search, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.

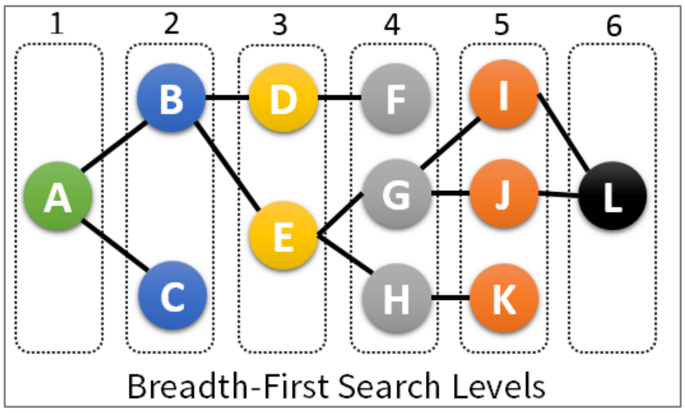


Depth-First	Breadth-First
• traverses deep into a structure by visiting children nodes before visiting sibling/neighbor nodes.	• traverses broad into a structure by visiting sibling/neighbor nodes before children nodes.
• uses a stack data structure.	• uses a queue data structure.

BFS Algorithm

1. Add a node/vertex from the graph to a queue of nodes to be “visited”.
2. Visit the topmost node in the queue, and mark it as such.
3. If that node has any neighbors, check to see if they have been “visited” or not.
4. Add any neighboring nodes that still need to be “visited” to the queue.
5. Remove the node we’ve visited from the queue.

Program terminates when there is no more nodes in the queue.



```
// BFS2.cpp : Queue implementation

#include <iostream>
#include <queue>
#include <vector>

using namespace std;

int N = 12, M = 12;
int startNode = 0;
vector<pair<int, int>> edgeList = {
{0,1},{1,3},{3,5},{1,4},{4,6},{6,8},{8,11},{0,2},{4,7},{7,10},{6,9},{9,11} };
vector<vector<int>> adjList;
//---------------------------------------------
void createAdjList()
{
        adjList.resize(N);
        for (auto edge:edgeList)
        {
                adjList[edge.first].push_back(edge.second);
                adjList[edge.second].push_back(edge.first);
        }
}
//---------------------------------------------
//print BFS traversal of the graph
void BFS()
{
        //None of the nodes has been visited yet
        vector<bool> visited(N, false);

        //start the traversal from the start node
        queue<int> q;
        q.push(startNode);
        visited[startNode] = true;

        //Keep going BFS until visiting all nodes
        while (!q.empty())
        {
                int u = q.front();    //current node
                q.pop();

                cout << u << " ";

                //Push all unvisited neighbours of current node into the queue
                for (int v : adjList[u])
                        if (!visited[v])
                        {
                                visited[v] = true;
                                q.push(v);
                        }
        }
}
//---------------------------------------------
int main()
{
        createAdjList();
        BFS();
}
```

Output

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Some Applications of BFS

- Cycle detection
- Testing if a graph is bipartite

- Finding connected components of a graph
- Finding the **shortest path** in an unweighted graph
- Finding a spanning tree (MST) of an unweighted graph
- Finding level of each node in a tree
- *Finding augmenting path in a flow network*

Sample Program 1: Count Cycles in a Di-graph with DFS

Use a Depth First Search (DFS) traversal algorithm to detect cycles in a directed graph.

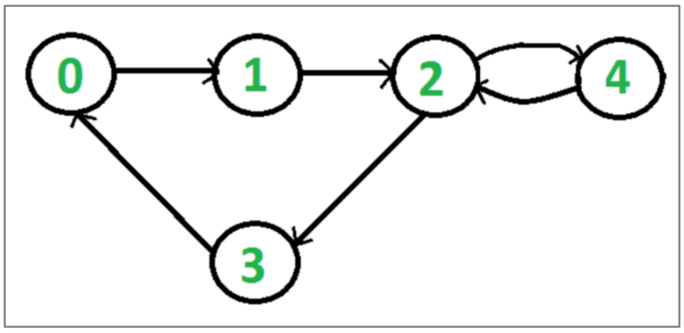
If there is any self-loop in any node, it will be considered as a cycle, otherwise, when the child node has another edge to connect its parent it will also a cycle.

Algorithm

Use three different sets to assign nodes to perform the DFS traversal: **White**, **Grey** and the **Black**.

Initially, all nodes will be stored inside the white set. When one new node is traversed, it will be stored in the gray set and removed from the white one. And after completing backtracking, when that task for that node is completed, it will be swapped from gray to black set.

Any back edge between any two gray nodes denotes a cycle. There are two cycles in the following graph.



Code

```
// Cycle Dedection DFS.cpp :
#include <iostream>
#include <vector>

using namespace std;

int N = 5, M = 6, res = 0;
vector<pair<int, int>> edgeList = { {0,1}, {3,0},{1,2},{2,4},{4,2}, {2,3} };
vector<vector<int>> adjList;
vector<bool> white, gray, black;
//----------------------------------------
void constructAdjList()
{
	adjList.resize(N);
	for (auto e : edgeList)
		adjList[e.first].push_back(e.second);
}
//----------------------------------------
void dfsUtil(int u)
{
        //u becomes gray
        white[u] = false;
        gray[u] = true;

        for (int v : adjList[u])
                if (gray[v]) //cycle dedected
                        res++;
                else if (white[v])
                        dfsUtil(v);

        //u becomes black
        gray[u] = false;
        //black[u] = true;
}
//----------------------------------------
void go()
{
        white.resize(N, true);       //white: not processed yet,
        gray.resize(N, false);       //gray: is being processed
        //black.resize(N, false);    //black: has been processed with its all decendents

        for (int i = 0; i < N; i++)
                if (white[i])
                        dfsUtil(i);
}
//----------------------------------------
int main()
{
        constructAdjList();
        go();
        cout << "There are " << res << " cycles." << endl;
}
```

Output

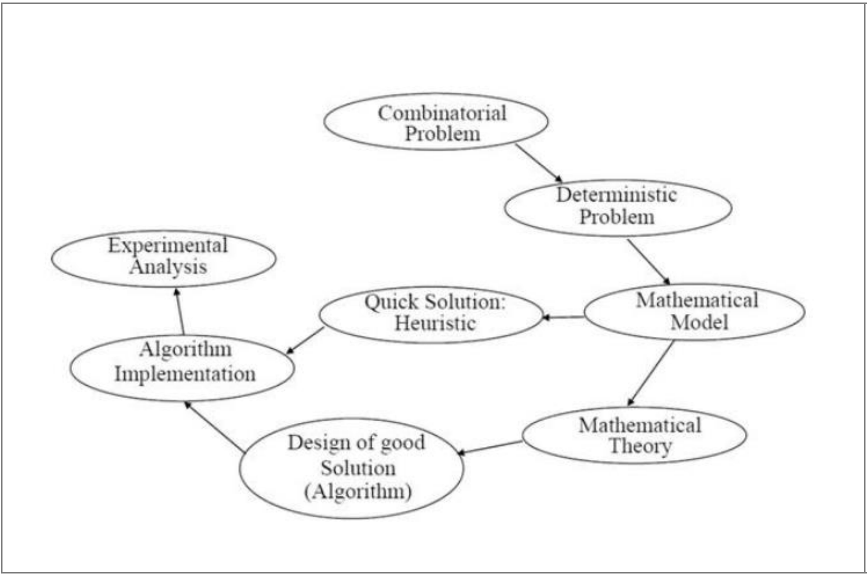
```
There are 2 cycles.
```

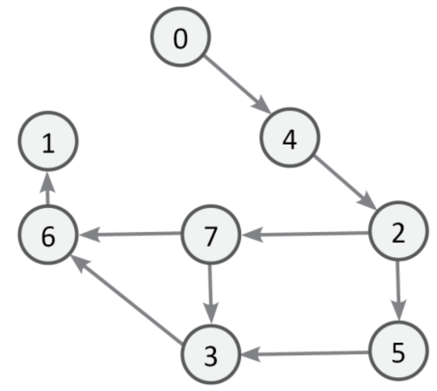
Sample Problem 2: Scheduling Problem (Topological Sort)

Given a set of jobs to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs such that they are all completed while still respecting the constraints?

Consider a college student planning a course schedule; under the constraint that certain courses are prerequisite for certain other courses, as in the example below.

Find an ordering of courses that student should take to finish all courses. There may be multiple correct orders; you just need to print one of them.





This problem can be solved in multiple ways; one simple and straightforward way is **Topological Sort**.

A **topological sort** or topological ordering of a directed acyclic graph (**DAG**) is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

Kahn's algorithm for Topological Sorting – O(V + E)

Step-1: Compute in-degree (number of incoming edges) for each of the vertex.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

- Decrease in-degree by 1 for all its neighboring nodes.
- If in-degree of neighboring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.

```cpp
// Top Sort.cpp : Khan's Algorithm
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

int N = 8, M = 9;
vector<pair<int, int>> edgeList = { {0,4},{4,2},{7,6},{6,1},{2,5},{5,3},{2,7},{7,3}, {3,6} };
vector<vector<int>> adjList;
vector<int> inDegrees;
//------------------------------------------
void constructAdjList()
{
	adjList.resize(N);
	inDegrees.resize(N, 0);
	for (auto e : edgeList)
	{
		adjList[e.first].push_back(e.second);
		inDegrees[e.second]++;
	}
}
//------------------------------------------
```

```
vector<int> topSort()
{
	queue<int> q;
	//Push starting nodes into the queue
	for (int i = 0; i < N; i++)
		if (inDegrees[i] == 0)
			q.push(i);

	vector<int> res;

	while (!q.empty())
	{
		int u = q.front();    //the current node
		q.pop();

		//Add u to the result vector
		res.push_back(u);

		//Decrease in-degree by 1 for all neighbors of u
		for (int v : adjList[u])
		{
			inDegrees[v]--;
			//If in-degree of neighboring nodes is reduced to zero, then add it to the 
queue.
			if (inDegrees[v] == 0)
				q.push(v);
		}
	}

	return res;
}
//----------------------------------------
int main()
{
	constructAdjList();
	vector<int> topSortVec = topSort();
	for (int x : topSortVec)
		cout << x << " ";
}
```

Output

```
0 4 2 5 7 3 6 1
```

Practice Problems

1. Perform a complete DFS traversal of a directed graph and count number of DFS tree edges, back edges, forward edges and cross edges.
2. Print a topological sort of a DAC with DFS.
3. Make cycle detection in a directed graph using a modified Top Sort algorithm.
4. Enumerate all topological sorts in a DAC (backtracking).
5. Determine if a graph is bipartite. A graph is bipartite graph if and only if it is 2-colorable.
6. Find all bridges (cut edges) in a graph in O(E* (E + V)) time.
7. Find all articulation points (cut vertices) in a graph in O(N*(V + E)) time.
8. Find all shortest distances from a source vertex to all other vertices in an unweighted graph.
9. Print one of the shortest paths between two vertices in an unweighted graph.
10. Find size of each subtree in a tree.

Useful Links and References

https://www.tutorialspoint.com/graph_theory/graph_theory_introduction.htm
https://www.dailysmarty.com/posts/graph-types
https://medium.com/basecs/a-gentle-introduction-to-graph-theory-77969829ead8
https://www.hackerearth.com/practice/algorithms/graphs/graph-representation/tutorial/
https://www.hackerearth.com/practice/algorithms/graphs/graph-representation/tutorial/
https://www.thecrazyprogrammer.com/2017/08/difference-between-tree-and-graph.html
https://www.tutorialride.com/data-structures/graphs-in-data-structure.htm
http://brianvanderplaats.com/cheat-sheets/Graph-Data-Structure-Cheat-Sheet.html
https://www.youtube.com/watch?v=d43cSRJg8YU
https://www.wikizeroo.org/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvR3JhcGhfdHJhdmVyc2Fs
https://medium.com/basecs/going-broad-in-a-graph-bfs-traversal-959bd1a09255
http://alexvolov.com/2015/02/breadth-first-search-bfs/
https://www.tutorialspoint.com/Detect-Cycle-in-a-Directed-Graph
https://www.geeksforgeeks.org/detect-cycle-direct-graph-using-colors/
https://xiaokangstudynotes.com/2017/01/17/course-schedule-and-topological-sort/
https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/