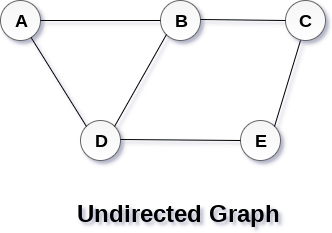
Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.

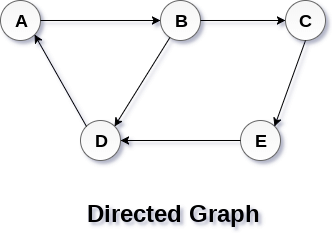


Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.

Simple Path

If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph representation

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

* **Sequential representation** (or, Adjacency matrix representation)
* **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

In this tutorial, we will discuss each one of them in detail.

Backward Skip 10sPlay VideoForward Skip 10s

Now, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry Aij in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Vi and Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

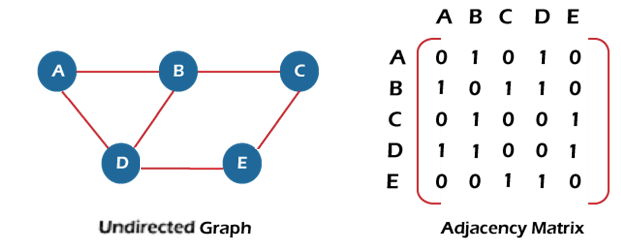
aij = 1 {if there is a path exists from Vi to Vj}

aij = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



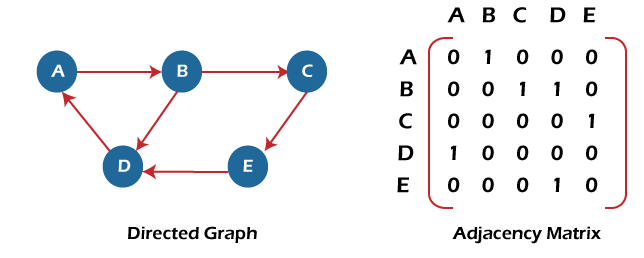
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

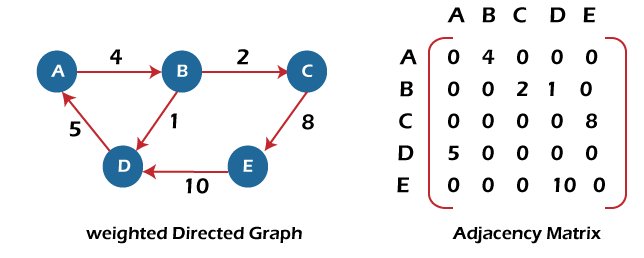
Consider the below-directed graph and try to construct the adjacency matrix of it.



In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

**Adjacency matrix for a weighted directed graph**

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

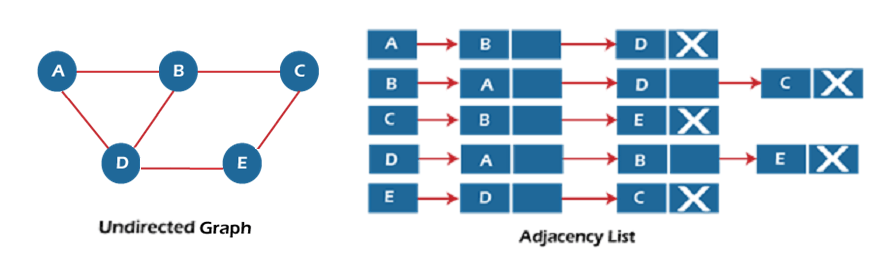
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

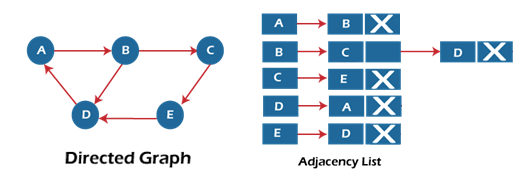


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

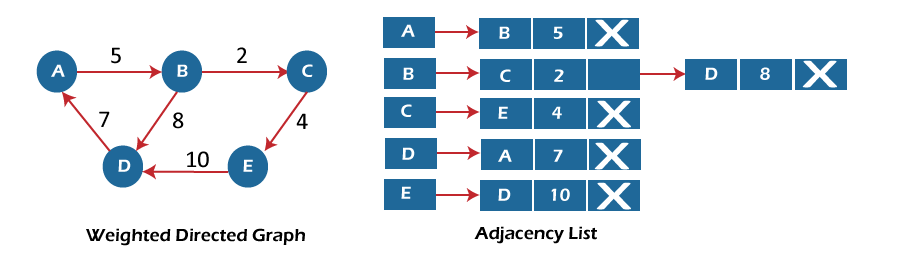
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

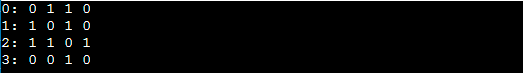
In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

Here, there are four vertices and five edges in the graph that are non-directed.

1. /\* Adjacency Matrix representation of an undirected graph in C \*/
3. #include <stdio.h>
4. #define V 4 /\* number of vertices in the graph \*/
6. /\* function to initialize the matrix to zero \*/
7. **void** init(**int** arr[][V]) {
8. **int** i, j;
9. **for** (i = 0; i < V; i++)
10. **for** (j = 0; j < V; j++)
11. arr[i][j] = 0;
12. }
14. /\* function to add edges to the graph \*/
15. **void** insertEdge(**int** arr[][V], **int** i, **int** j) {
16. arr[i][j] = 1;
17. arr[j][i] = 1;
18. }
20. /\* function to print the matrix elements \*/
21. **void** printAdjMatrix(**int** arr[][V]) {
22. **int** i, j;
23. **for** (i = 0; i < V; i++) {
24. printf("%d: ", i);
25. **for** (j = 0; j < V; j++) {
26. printf("%d ", arr[i][j]);
27. }
28. printf("\n");
29. }
30. }
32. **int** main() {
33. **int** adjMatrix[V][V];
35. init(adjMatrix);
36. insertEdge(adjMatrix, 0, 1);
37. insertEdge(adjMatrix, 0, 2);
38. insertEdge(adjMatrix, 1, 2);
39. insertEdge(adjMatrix, 2, 0);
40. insertEdge(adjMatrix, 2, 3);
42. printAdjMatrix(adjMatrix);
44. **return** 0;
45. }

**Output:**

After the execution of the above code, the output will be -



Implementation of adjacency list representation of Graph

Now, let's see the implementation of adjacency list representation of graph in C.

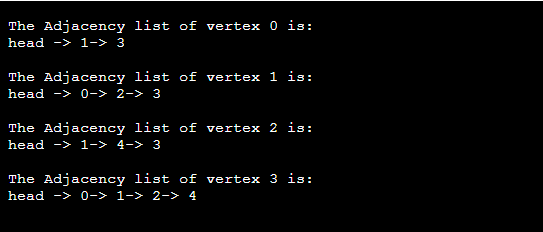
In this program, there is an adjacency list representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

1. /\* Adjacency list representation of a graph in C \*/
2. #include <stdio.h>
3. #include <stdlib.h>
5. /\* structure to represent a node of adjacency list \*/
6. **struct** AdjNode {
7. **int** dest;
8. **struct** AdjNode\* next;
9. };
11. /\* structure to represent an adjacency list \*/
12. **struct** AdjList {
13. **struct** AdjNode\* head;
14. };
16. /\* structure to represent the graph \*/
17. **struct** Graph {
18. **int** V; /\*number of vertices in the graph\*/
19. **struct** AdjList\* array;
20. };

23. **struct** AdjNode\* newAdjNode(**int** dest)
24. {
25. **struct** AdjNode\* newNode = (**struct** AdjNode\*)malloc(**sizeof**(**struct** AdjNode));
26. newNode->dest = dest;
27. newNode->next = NULL;
28. **return** newNode;
29. }
31. **struct** Graph\* createGraph(**int** V)
32. {
33. **struct** Graph\* graph = (**struct** Graph\*)malloc(**sizeof**(**struct** Graph));
34. graph->V = V;
35. graph->array = (**struct** AdjList\*)malloc(V \* **sizeof**(**struct** AdjList));
37. /\* Initialize each adjacency list as empty by making head as NULL \*/
38. **int** i;
39. **for** (i = 0; i < V; ++i)
40. graph->array[i].head = NULL;
41. **return** graph;
42. }
44. /\* function to add an edge to an undirected graph \*/
45. **void** addEdge(**struct** Graph\* graph, **int** src, **int** dest)
46. {
47. /\* Add an edge from src to dest. The node is added at the beginning \*/
48. **struct** AdjNode\* check = NULL;
49. **struct** AdjNode\* newNode = newAdjNode(dest);
51. **if** (graph->array[src].head == NULL) {
52. newNode->next = graph->array[src].head;
53. graph->array[src].head = newNode;
54. }
55. **else** {
57. check = graph->array[src].head;
58. **while** (check->next != NULL) {
59. check = check->next;
60. }
61. // graph->array[src].head = newNode;
62. check->next = newNode;
63. }
65. /\* Since graph is undirected, add an edge from dest to src also \*/
66. newNode = newAdjNode(src);
67. **if** (graph->array[dest].head == NULL) {
68. newNode->next = graph->array[dest].head;
69. graph->array[dest].head = newNode;
70. }
71. **else** {
72. check = graph->array[dest].head;
73. **while** (check->next != NULL) {
74. check = check->next;
75. }
76. check->next = newNode;
77. }
78. }
79. /\* function to print the adjacency list representation of graph\*/
80. **void** print(**struct** Graph\* graph)
81. {
82. **int** v;
83. **for** (v = 0; v < graph->V; ++v) {
84. **struct** AdjNode\* pCrawl = graph->array[v].head;
85. printf("\n The Adjacency list of vertex %d is: \n head ", v);
86. **while** (pCrawl) {
87. printf("-> %d", pCrawl->dest);
88. pCrawl = pCrawl->next;
89. }
90. printf("\n");
91. }
92. }
94. **int** main()
95. {
97. **int** V = 4;
98. **struct** Graph\* g = createGraph(V);
99. addEdge(g, 0, 1);
100. addEdge(g, 0, 3);
101. addEdge(g, 1, 2);
102. addEdge(g, 1, 3);
103. addEdge(g, 2, 4);
104. addEdge(g, 2, 3);
105. addEdge(g, 3, 4);
106. print(g);
107. **return** 0;
108. }

**Output:**

In the output, we will see the adjacency list representation of all the vertices of the graph. After the execution of the above code, the output will be -



BFS algorithm

In this article, we will discuss the BFS algorithm in the data structure. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

The applications of breadth-first-algorithm are given as follows -

* BFS can be used to find the neighboring locations from a given source location.
* In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
* BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
* BFS is used to determine the shortest path and minimum spanning tree.
* BFS is also used in Cheney's technique to duplicate the garbage collection.
* It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -

**Step 1:** SET STATUS = 1 (ready state) for each node in G

**Step 2:** Enqueue the starting node A and set its STATUS = 2 (waiting state)

**Step 3:** Repeat Steps 4 and 5 until QUEUE is empty

**Step 4:** Dequeue a node N. Process it and set its STATUS = 3 (processed state).

**Step 5:** Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

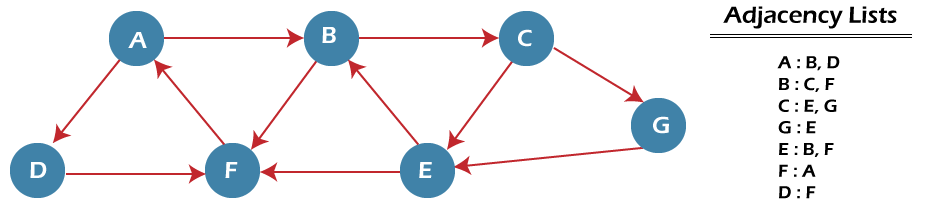
(waiting state)

[END OF LOOP]

**Step 6:** EXIT

Example of BFS algorithm

Now, let's understand the working of BFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Now, let's start examining the graph starting from Node A.

**Step 1** - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

**Step 2** - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

**Step 3** - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

**Step 4** - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

**Step 5** - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

**Step 5** - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

**Step 6** - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Complexity of BFS algorithm

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is **O(V+E)**, since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

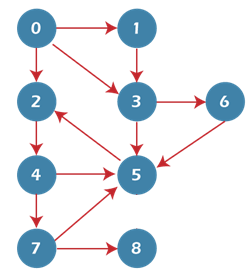
The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

Implementation of BFS algorithm

Now, let's see the implementation of BFS algorithm in java.

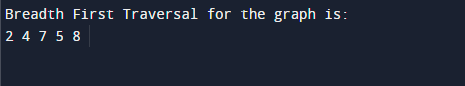
In this code, we are using the adjacency list to represent our graph. Implementing the Breadth-First Search algorithm in Java makes it much easier to deal with the adjacency list since we only have to travel through the list of nodes attached to each node once the node is dequeued from the head (or start) of the queue.

In this example, the graph that we are using to demonstrate the code is given as follows -



1. **import** java.io.\*;
2. **import** java.util.\*;
3. **public** **class** BFSTraversal
4. {
5. **private** **int** vertex;       /\* total number number of vertices in the graph \*/
6. **private** LinkedList<Integer> adj[];      /\* adjacency list \*/
7. **private** Queue<Integer> que;           /\* maintaining a queue \*/
8. BFSTraversal(**int** v)
9. {
10. vertex = v;
11. adj = **new** LinkedList[vertex];
12. **for** (**int** i=0; i<v; i++)
13. {
14. adj[i] = **new** LinkedList<>();
15. }
16. que = **new** LinkedList<Integer>();
17. }
18. **void** insertEdge(**int** v,**int** w)
19. {
20. adj[v].add(w);      /\* adding an edge to the adjacency list (edges are bidirectional in this example) \*/
21. }
22. **void** BFS(**int** n)
23. {
24. **boolean** nodes[] = **new** **boolean**[vertex];       /\* initialize boolean array for holding the data \*/
25. **int** a = 0;
26. nodes[n]=**true**;
27. que.add(n);       /\* root node is added to the top of the queue \*/
28. **while** (que.size() != 0)
29. {
30. n = que.poll();        /\* remove the top element of the queue \*/
31. System.out.print(n+" ");    /\* print the top element of the queue \*/
32. **for** (**int** i = 0; i < adj[n].size(); i++)  /\* iterate through the linked list and push all neighbors into queue \*/
33. {
34. a = adj[n].get(i);
35. **if** (!nodes[a])      /\* only insert nodes into queue if they have not been explored already \*/
36. {
37. nodes[a] = **true**;
38. que.add(a);
39. }
40. }
41. }
42. }
43. **public** **static** **void** main(String args[])
44. {
45. BFSTraversal graph = **new** BFSTraversal(10);
46. graph.insertEdge(0, 1);
47. graph.insertEdge(0, 2);
48. graph.insertEdge(0, 3);
49. graph.insertEdge(1, 3);
50. graph.insertEdge(2, 4);
51. graph.insertEdge(3, 5);
52. graph.insertEdge(3, 6);
53. graph.insertEdge(4, 7);
54. graph.insertEdge(4, 5);
55. graph.insertEdge(5, 2);
56. graph.insertEdge(6, 5);
57. graph.insertEdge(7, 5);
58. graph.insertEdge(7, 8);
59. System.out.println("Breadth First Traversal for the graph is:");
60. graph.BFS(2);
61. }
62. }

**Output**



DFS (Depth First Search) algorithm

In this article, we will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

* DFS algorithm can be used to implement the topological sorting.
* It can be used to find the paths between two vertices.
* It can also be used to detect cycles in the graph.
* DFS algorithm is also used for one solution puzzles.
* DFS is used to determine if a graph is bipartite or not.

Algorithm

**Step 1:** SET STATUS = 1 (ready state) for each node in G

**Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)

**Step 3:** Repeat Steps 4 and 5 until STACK is empty

**Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)

**Step 5:** Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

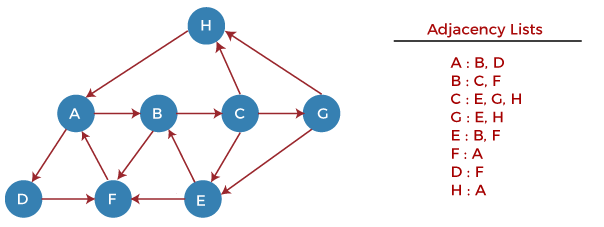
**Step 6:** EXIT

Pseudocode

1. DFS(G,v)   ( v is the vertex where the search starts )
2. Stack S := {};   ( start with an empty stack )
3. **for** each vertex u, set visited[u] := **false**;
4. push S, v;
5. **while** (S is not empty) **do**
6. u := pop S;
7. **if** (not visited[u]) then
8. visited[u] := **true**;
9. **for** each unvisited neighbour w of uu
10. push S, w;
11. end **if**
12. end **while**
13. END DFS()

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Now, let's start examining the graph starting from Node H.

**Step 1** - First, push H onto the stack.

1. STACK: H

**Step 2** - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

**Step 3** - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

**Step 4** - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

**Step 5** - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

**Step 6** - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

**Step 7** - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

**Step 8** - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

**Step 9** - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

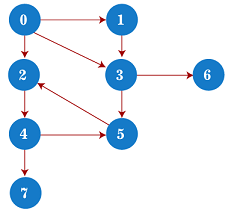
The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

Implementation of DFS algorithm

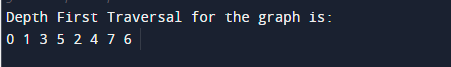
Now, let's see the implementation of DFS algorithm in Java.

In this example, the graph that we are using to demonstrate the code is given as follows -



1. /\*A sample java program to implement the DFS algorithm\*/
3. **import** java.util.\*;
5. **class** DFSTraversal {
6. **private** LinkedList<Integer> adj[]; /\*adjacency list representation\*/
7. **private** **boolean** visited[];
9. /\* Creation of the graph \*/
10. DFSTraversal(**int** V) /\*'V' is the number of vertices in the graph\*/
11. {
12. adj = **new** LinkedList[V];
13. visited = **new** **boolean**[V];
15. **for** (**int** i = 0; i < V; i++)
16. adj[i] = **new** LinkedList<Integer>();
17. }
19. /\* Adding an edge to the graph \*/
20. **void** insertEdge(**int** src, **int** dest) {
21. adj[src].add(dest);
22. }
24. **void** DFS(**int** vertex) {
25. visited[vertex] = **true**; /\*Mark the current node as visited\*/
26. System.out.print(vertex + " ");
28. Iterator<Integer> it = adj[vertex].listIterator();
29. **while** (it.hasNext()) {
30. **int** n = it.next();
31. **if** (!visited[n])
32. DFS(n);
33. }
34. }
36. **public** **static** **void** main(String args[]) {
37. DFSTraversal graph = **new** DFSTraversal(8);
39. graph.insertEdge(0, 1);
40. graph.insertEdge(0, 2);
41. graph.insertEdge(0, 3);
42. graph.insertEdge(1, 3);
43. graph.insertEdge(2, 4);
44. graph.insertEdge(3, 5);
45. graph.insertEdge(3, 6);
46. graph.insertEdge(4, 7);
47. graph.insertEdge(4, 5);
48. graph.insertEdge(5, 2);
50. System.out.println("Depth First Traversal for the graph is:");
51. graph.DFS(0);
52. }
53. }

**Output**



Spanning tree

In this article, we will discuss the spanning tree and the minimum spanning tree. But before moving directly towards the spanning tree, let's first see a brief description of the graph and its types.

Graph

A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -

* **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
* **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
* **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

Now, let's move towards the topic spanning tree.

What is a spanning tree?

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have **nn-2** number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be **55-2 = 125.**

Applications of the spanning tree

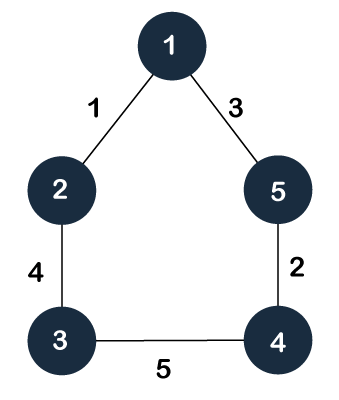
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

* Cluster Analysis
* Civil network planning
* Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

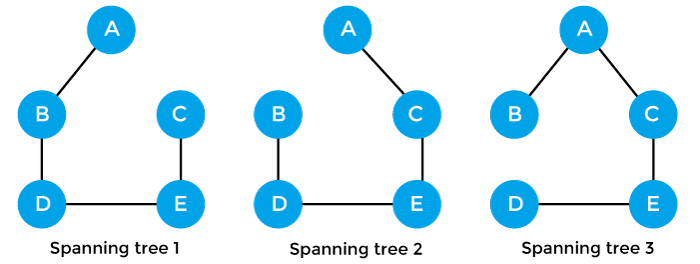
Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

* There can be more than one spanning tree of a connected graph G.
* A spanning tree does not have any cycles or loop.
* A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
* A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
* There can be a maximum **nn-2** number of spanning trees that can be created from a complete graph.
* A spanning tree has **n-1** edges, where 'n' is the number of nodes.
* If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

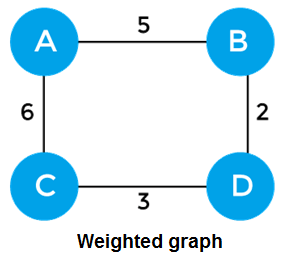
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Minimum Spanning tree

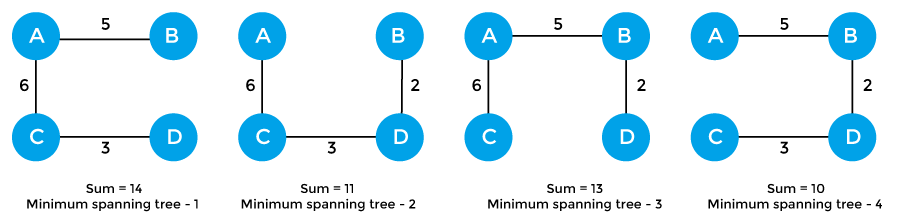
A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

Example of minimum spanning tree

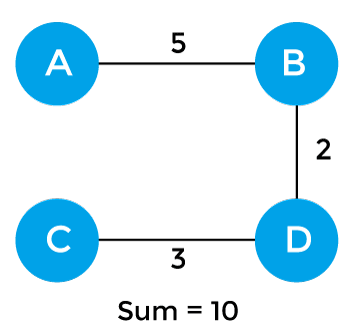
Let's understand the minimum spanning tree with the help of an example.



The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

* Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
* It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

* Prim's Algorithm
* Kruskal's Algorithm

Let's see a brief description of both of the algorithms listed above.

**Prim's algorithm -** It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

To learn more about the prim's algorithm, you can click the below link - <https://www.javatpoint.com/prim-algorithm>

**Kruskal's algorithm -** This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.