# 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

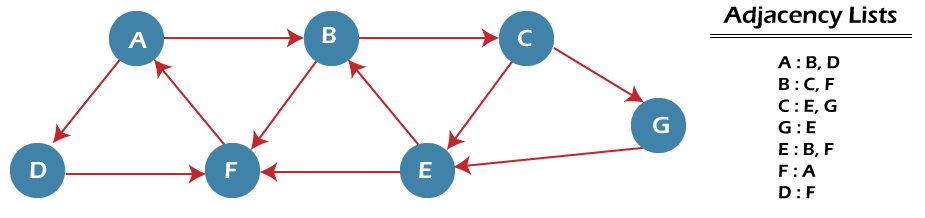
(waiting state)

[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.

# 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]

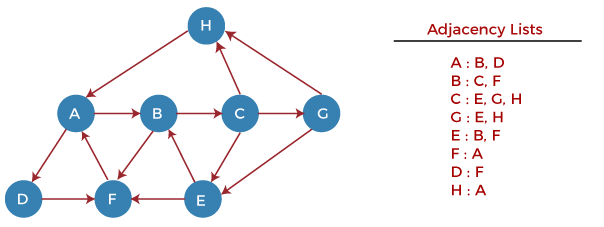
**Step 6:** EXIT

### 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 u
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.

ADVERTISEMENT

ADVERTISEMENT

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: