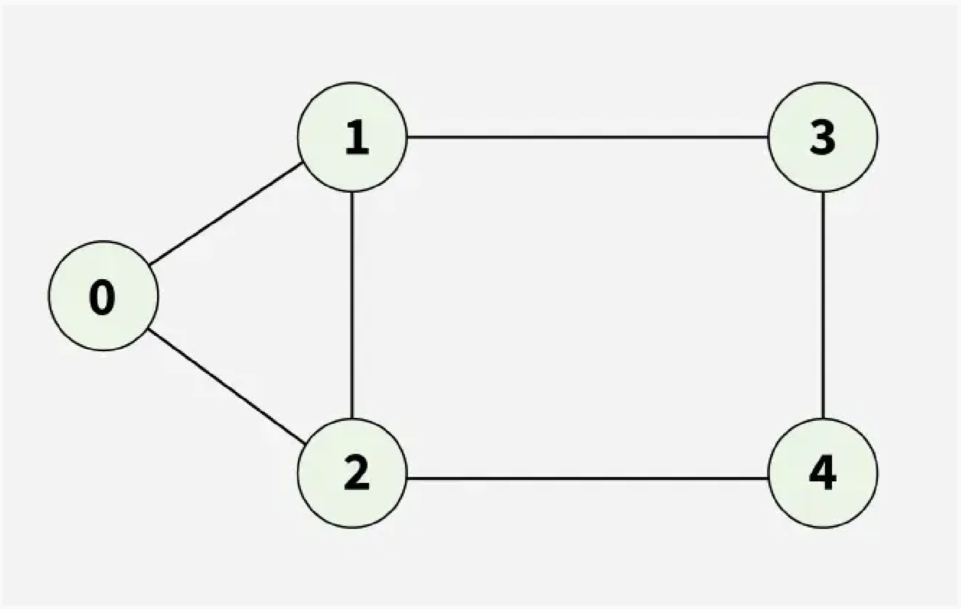
Breadth First Search or BFS for a Graph

Last Updated : 21 Apr, 2025

Given a **undirected graph** represented by an adjacency list **`adj`**, where each `adj[i]` represents the list of vertices connected to vertex `i`. Perform a **Breadth First Search (BFS)** traversal starting from vertex `0`, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.

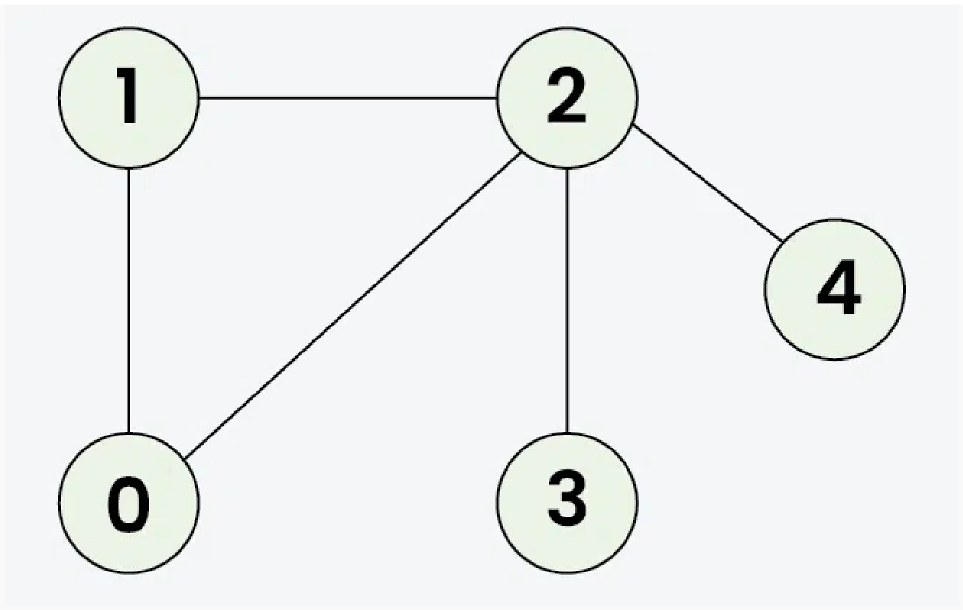
Examples:

Input: *adj[][] = [[1,2], [0,2,3], [0,1,4], [1,4], [2,3]]*



Output: *[0, 1, 2, 3, 4]*
Explanation: *Starting from 0, the BFS traversal will follow these steps:*
Visit 0 → Output: [0]
Visit 1 (first neighbor of 0) → Output: [0, 1]
Visit 2 (next neighbor of 0) → Output: [0, 1, 2]
Visit 3 (next neighbor of 1) → Output: [0, 1, 2, 3]
Visit 4 (neighbor of 2) → Final Output: [0, 1, 2, 3, 4]

Input: *adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]*



Output: *[0, 1, 2, 3, 4]*
Explanation: *Starting from 0, the BFS traversal proceeds as follows:*
Visit 0 → Output: [0]
Visit 1 (the first neighbor of 0) → Output: [0, 1]
Visit 2 (the next neighbor of 0) → Output: [0, 1, 2]
Visit 3 (the first neighbor of 2 that hasn't been visited yet) → Output: [0, 1, 2, 3]
Visit 4 (the next neighbor of 2) → Final Output: [0, 1, 2, 3, 4]

Table of Content

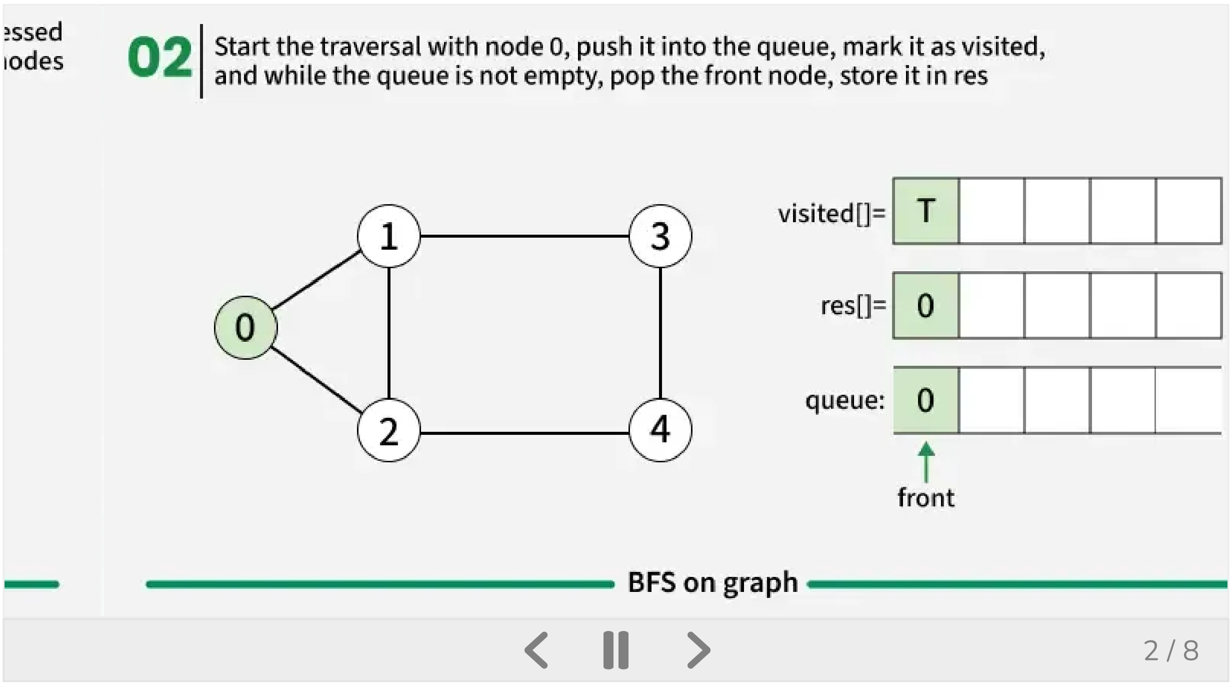
What is Breadth First Search?

Breadth First Search (BFS) *is a fundamental* ***graph traversal algorithm.*** *It begins with a node, then first traverses all its adjacent nodes. Once all adjacent are visited, then their adjacent are traversed.*

- *BFS is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level.*
- *Popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS.*
- *BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.*

BFS from a Given Source

*The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to the Breadth-First Traversal of a tree. Like tree, we begin with the given source (in tree, we begin with root) and traverse vertices level by level using a queue data structure. The only catch here is that, 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.*



Start the traversal with node 0, push it into the queue, mark it as visited, and while the queue is not empty, pop the front node, store it in res

visited[]=	T				
res[]=	0				
queue:	0				

front

Follow the below given approach:

1. **Initialization:** Enqueue the given source vertex into a queue and mark it as visited.
2. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
    - Enqueue the neighbor into the queue.
    - Mark the neighbor as visited.

3. **Termination:** Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

C++ Java Python C# JavaScript

```cpp
#include<bits/stdc++.h>
using namespace std;

// BFS from given source s
vector<int> bfs(vector<vector<int>>& adj)  {
    int V = adj.size();

    int s = 0; // source node
    // create an array to store the traversal
    vector<int> res;

    // Create a queue for BFS
    queue<int> q;

    // Initially mark all the vertices as not visited
    vector<bool> visited(V, false);

    // Mark source node as visited and enqueue it
    visited[s] = true;
    q.push(s);

    // Iterate over the queue
    while (!q.empty()) {

        // Dequeue a vertex from queue and store it
        int curr = q.front();
        q.pop();
        res.push_back(curr);

        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (int x : adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }

    return res;
}

int main()  {

    vector<vector<int>> adj = {{1,2}, {0,2,3}, {0,4}, {1,4},
{2,3}};
    vector<int> ans = bfs(adj);
    for(auto i:ans) {
        cout<<i<<" ";
    }
    return 0;
}
```

Output

```
0 1 2 3 4
```

BFS of the Disconnected Graph

The above implementation takes a source as an input and prints only those vertices that are reachable from the source and would not print all vertices in case of disconnected graph. Let us see the algorithm that prints all vertices without any source and the graph maybe disconnected.

The algorithm is simple, instead of calling BFS for a single vertex, we call the above implemented BFS for all not yet visited vertices one by one.

C++ Java Python C# JavaScript

```cpp
#include<bits/stdc++.h>
using namespace std;

// BFS from given source s
void bfs(vector<vector<int>>& adj, int s,
        vector<bool>& visited, vector<int> &res) {

    // Create a queue for BFS
    queue<int> q;

    // Mark source node as visited and enqueue it
    visited[s] = true;
    q.push(s);

    // Iterate over the queue
    while (!q.empty()) {

        // Dequeue a vertex and store it
        int curr = q.front();
        q.pop();
        res.push_back(curr);

        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (int x : adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }
}

// Perform BFS for the entire graph which maybe
// disconnected
vector<int> bfsDisconnected(vector<vector<int>>& adj) {
    int V = adj.size();

    // create an array to store the traversal
    vector<int> res;

    // Initially mark all the vertices as not visited
    vector<bool> visited(V, false);

    // perform BFS for each node
    for (int i = 0; i < adj.size(); ++i) {
        if (!visited[i]) {
            bfs(adj, i, visited, res);
        }
    }

    return res;
}

int main()  {

    vector<vector<int>> adj = { {1, 2}, {0}, {0},
                                       {4}, {3, 5}, {4}};
    vector<int> ans = bfsDisconnected(adj);
    for(auto i:ans) {
        cout<<i<<" ";
    }
    return 0;
}
```

Output

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

Complexity Analysis of Breadth-First Search (BFS) Algorithm

> ***Time Complexity: O(V + E),*** *BFS explores all the vertices and edges in the graph. In the worst case, it visits every vertex and edge once. Therefore, the*

time complexity of BFS is O(V + E), where V and E are the number of vertices and edges in the given graph.

Auxiliary Space: O(V), *BFS uses a queue to keep track of the vertices that need to be visited. In the worst case, the queue can contain all the vertices in the graph. Therefore, the space complexity of BFS is O(V).*

Applications of BFS in Graphs

BFS has various applications in graph theory and computer science, including:

- **Shortest Path Finding:** BFS can be used to find the shortest path between two nodes in an unweighted graph. By keeping track of the parent of each node during the traversal, the shortest path can be reconstructed.
- **Cycle Detection:** BFS can be used to detect cycles in a graph. If a node is visited twice during the traversal, it indicates the presence of a cycle.
- **Connected Components:** BFS can be used to identify connected components in a graph. Each connected component is a set of nodes that can be reached from each other.
- **Topological Sorting:** BFS can be used to perform topological sorting on a directed acyclic graph (DAG). Topological sorting arranges the nodes in a linear order such that for any edge (u, v), u appears before v in the order.
- **Level Order Traversal of Binary Trees:** BFS can be used to perform a level order traversal of a binary tree. This traversal visits all nodes at the same level before moving to the next level.
- **Network Routing:** BFS can be used to find the shortest path between two nodes in a network, making it useful for routing data packets in network protocols.

Problems on BFS for a Graph

- Find the level of a given node in an Undirected Graph
- Minimize maximum adjacent difference in a path from top-left to bottom-right
- Minimum jump to the same value or adjacent to reach the end of an Array

- Maximum coin in minimum time by skipping K obstacles along the path in Matrix
- Check if all nodes of the Undirected Graph can be visited from the given Node
- Minimum time to visit all nodes of a given Graph at least once
- Minimize moves to the next greater element to reach the end of the Array
- Shortest path by removing K walls
- Minimum time required to infect all the nodes of the Binary tree
- Check if destination of given Matrix is reachable with required values of cells

FAQs on Breadth First Search (BFS) for a Graph

Question 1: What is BFS and how does it work?

Answer: *BFS is a graph traversal algorithm that systematically explores a graph by visiting all the vertices at a given level before moving on to the next level. It starts from a starting vertex, enqueues it into a queue, and marks it as visited. Then, it dequeues a vertex from the queue, visits it, and enqueues all its unvisited neighbors into the queue. This process continues until the queue is empty.*

Question 2: What are the applications of BFS?

Answer: *BFS has various applications, including finding the shortest path in an unweighted graph, detecting cycles in a graph, topologically sorting a directed acyclic graph (DAG), finding connected components in a graph, and solving puzzles like mazes and Sudoku.*

Question 3: What is the time complexity of BFS?

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

Question 4: What is the space complexity of BFS?

Answer: *The space complexity of BFS is O(V), as it uses a queue to keep track of the vertices that need to be visited.*

Question 5: What are the advantages of using BFS?

Answer: *BFS is simple to implement and efficient for finding the shortest path in an unweighted graph. It also guarantees that all the vertices in the graph are visited.*

Related Articles:

- Recent Articles on BFS
- Depth First Traversal
- Applications of Breadth First Traversal
- Applications of Depth First Search
- Time and Space Complexity of Breadth First Search (BFS)

Breadth First Search

Visit Course ↗

Comment More info Advertise with us

Next Article

C Program for Breadth First Search or BFS for a Graph

Similar Reads

Breadth First Search or BFS for a Graph

Given a undirected graph represented by an adjacency list adj, where each adj[i] represents the list of vertices connected to vertex i. Perform a Breadth First Search (BFS) traversal starting from vertex 0,...

15+ min read

BFS in different language

BFS for Disconnected Graph

In the previous post, BFS only with a particular vertex is performed i.e. it is assumed that all vertices are reachable from the starting vertex. But in the case of a disconnected graph or any vertex that is...

14 min read

Applications, Advantages and Disadvantages of Breadth First Search (BFS)

We have earlier discussed Breadth First Traversal Algorithm for Graphs. Here in this article, we will see the applications, advantages, and disadvantages of the Breadth First Search. Applications of Breadth...

4 min read

Breadth First Traversal (BFS) on a 2D array

Given a matrix of size M x N consisting of integers, the task is to print the matrix elements using Breadth-First Search traversal. Examples: Input: grid[][] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15,...

9 min read

0-1 BFS (Shortest Path in a Binary Weight Graph)

Given a graph where every edge has weight as either 0 or 1. A source vertex is also given in the graph. Find the shortest path from the source vertex to every other vertex.Â For Example:Â Input : Source...

9 min read

Variations of BFS implementations

Easy problems on BFS

Intermediate problems on BFS

Hard Problems on BFS