Graph BFS

Graph traversals:

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.
During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS):

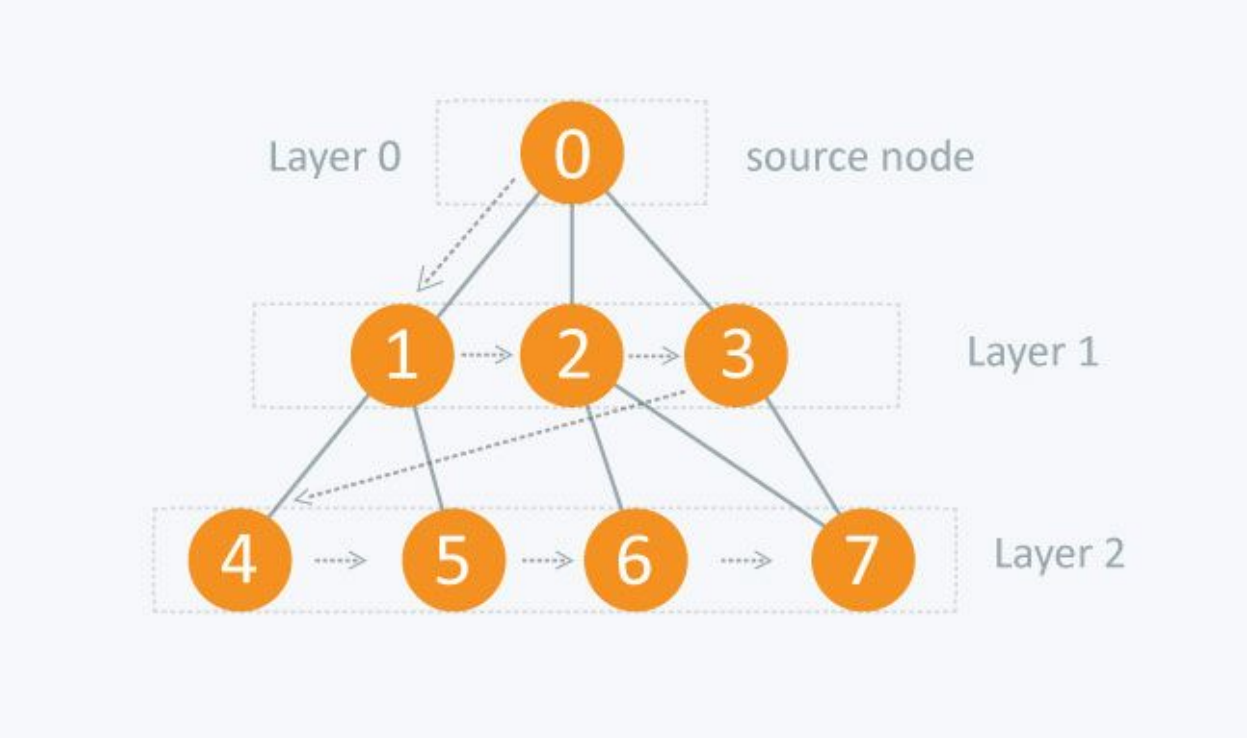
There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram:



The distance between the nodes in layer 1 is comparitively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

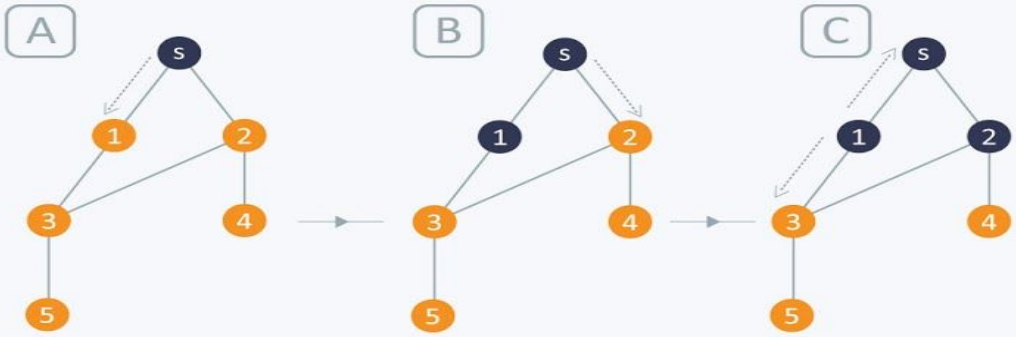
Traversing child nodes:

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

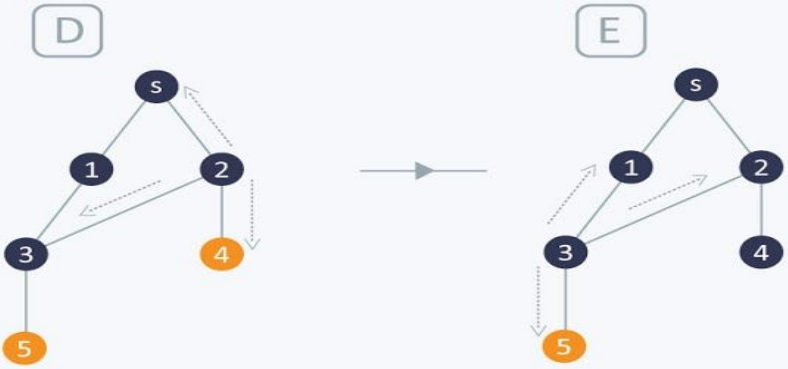
In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbours of the node will be visited

in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

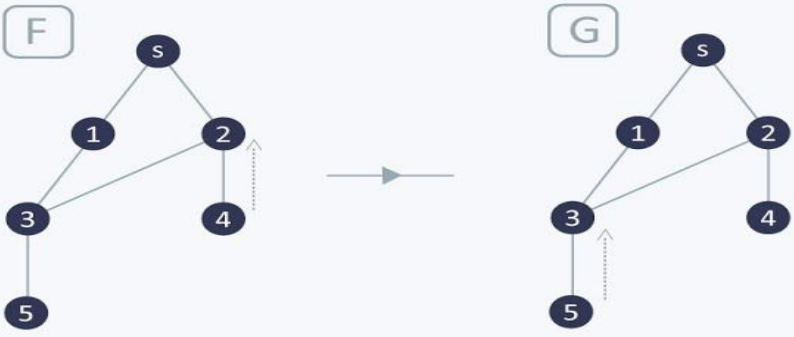


Here s is already marked, so it will be ignored



Here s and 3 are already marked, so they will be ignored

Here 1 & 2 are already marked so they will be ignored



Here 2 is already marked, so it will be ignored

Here 3 is already marked, so it will be ignored

The traversing will start from the source node and push *s* in queue. *s* will be marked as 'visited'.

First iteration

- s will be popped from the queue
- Neighbors of s i.e. 1 and 2 will be traversed
- 1 and 2, which have not been traversed earlier, are traversed. They will be:
    - Pushed in the queue
    - 1 and 2 will be marked as visited

Second iteration

- 1 is popped from the queue
- Neighbors of 1 i.e. s and 3 are traversed
- s is ignored because it is marked as 'visited'
- 3, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Third iteration

- 2 is popped from the queue
- Neighbors of 2 i.e. s, 3, and 4 are traversed
- 3 and s are ignored because they are marked as 'visited'
- 4, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Fourth iteration

- 3 is popped from the queue
- Neighbors of 3 i.e. 1, 2, and 5 are traversed
- 1 and 2 are ignored because they are marked as 'visited'
- 5, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Fifth iteration

- 4 will be popped from the queue
- Neighbors of 4 i.e. 2 is traversed
- 2 is ignored because it is already marked as 'visited'

Sixth iteration

- 5 is popped from the queue
- Neighbors of 5 i.e. 3 is traversed
- 3 is ignored because it is already marked as 'visited'

The queue is empty and it comes out of the loop. All the nodes have been traversed by using BFS.

As in this diagram, start from the source node, to find the distance between the source node and node 1. If you do not follow the BFS algorithm, you can go from the source node to node 2 and then to node 1. This approach will calculate the distance between the source node and node 1 as 2, whereas, the minimum distance is actually 1. The minimum distance can be calculated correctly by using the BFS algorithm.

Complexity:

The **time complexity** of BFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.

Applications:

1. How to determine the level of each node in the given tree?
As you know in BFS, you traverse level wise. You can also use BFS to determine the level of each node.

Implementation code of "BFS" in "C":

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue {
    int items[SIZE];
    int front;
    int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node
{
    int vertex;
    struct node* next;
};

struct node* createNode(int);
```

```
struct Graph
{
    int numVertices;
    struct node** adjLists;
    int* visited;
};

struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void bfs(struct Graph* graph, int startVertex);

int main()
{
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);

    return 0;
}

void bfs(struct Graph* graph, int startVertex) {

    struct queue* q = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while(!isEmpty(q)){
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

       struct node* temp = graph->adjLists[currentVertex];
```

```
        while(temp) {
            int adjVertex = temp->vertex;

            if(graph->visited[adjVertex] == 0){
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}


struct node* createNode(int v)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}


struct Graph* createGraph(int vertices)
{
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));


    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}
```

```c
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

struct queue* createQueue() {
    struct queue* q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}


int isEmpty(struct queue* q) {
    if(q->rear == -1)
        return 1;
    else
        return 0;
}

void enqueue(struct queue* q, int value){
    if(q->rear == SIZE-1)
        printf("\nQueue is Full!!");
    else {
        if(q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

int dequeue(struct queue* q){
    int item;
```

```
    if(isEmpty(q)){
        printf("Queue is empty");
        item = -1;
    }
    else{
        item = q->items[q->front];
        q->front++;
        if(q->front > q->rear){
            printf("Resetting queue");
            q->front = q->rear = -1;
        }
    }
    return item;
}

void printQueue(struct queue *q) {
    int i = q->front;

    if(isEmpty(q)) {
        printf("Queue is empty");
    } else {
        printf("\nQueue contains \n");
        for(i = q->front; i < q->rear + 1; i++) {
                printf("%d ", q->items[i]);
        }
    }
}
```

Implementation code of BFS in "Java":

```
import java.io.*;
import java.util.*;

class Graph
{
    private int numVertices;
    private LinkedList<Integer> adjLists[];
    private boolean visited[];

    Graph(int v)
    {
        numVertices = v;
        visited = new boolean[numVertices];
        adjLists = new LinkedList[numVertices];
        for (int i=0; i i = adjLists[currVertex].listIterator();
            while (i.hasNext())
            {
                int adjVertex = i.next();
                if (!visited[adjVertex])
                {
                    visited[adjVertex] = true;
                    queue.add(adjVertex);
                }
            }
        }
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Breadth First Traversal "+
                           "(starting from vertex 2)");
```

```
        g.BFS(2);
    }
}
```