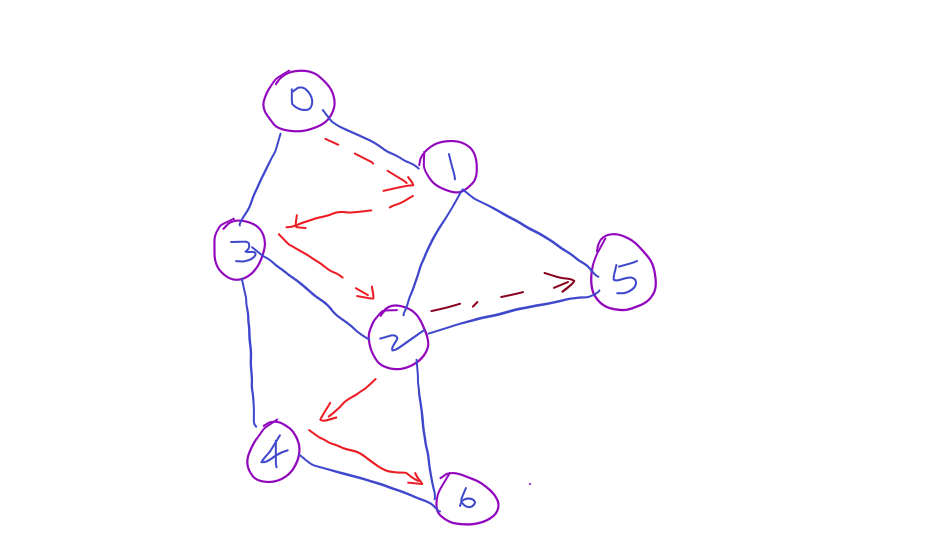
**AA LAB ASSIGNMENT | BFS**

**TITLE:**  Analysis, Proof of Analysis and Implementation of **Breadth First Search**

**MECHANISM**

In BFS, the algorithm starts at the root vertex (or any arbitrary vertex) and explores the graph by visiting all the adjacent vertices/nodes first, before moving on to the vertices/nodes that are at a greater depth (i.e., at the next level) from the starting vertex. BFS uses a queue data structure to keep track of the vertices/nodes to be visited, and it ensures that vertices/nodes at the same level are visited before moving to the next level.



**ALGORITHM / PSEUDOCODE**

**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

**IMPLEMENTATION**

*#include* <bits/stdc++.h>

using namespace std;

class Graph

{

    int V;

    vector<list<int>> adj;

public:

    Graph(int *V*);

    void addEdge(int *v*, int *w*);

    void BFS(int *s*);

};

Graph::Graph(int *V*)

{

    this->V = *V*;

    adj.resize(*V*);

}

void Graph::addEdge(int *v*, int *w*)

{

    adj[*v*].push\_back(*w*);

}

void Graph::BFS(int *s*)

{

    vector<bool> visited;

    visited.resize(V, false);

    list<int> queue;

    visited[*s*] = true;

    queue.push\_back(*s*);

*while* (!queue.empty())

    {

*s* = queue.front();

        cout << *s* << " ";

        queue.pop\_front();

*for* (auto adjacent : adj[*s*])

        {

*if* (!visited[adjacent])

            {

                visited[adjacent] = true;

                queue.push\_back(adjacent);

            }

        }

    }

}

int main()

{

    Graph g(4);

    g.addEdge(0, 1);

    g.addEdge(0, 3);

    g.addEdge(1, 0);

    g.addEdge(1, 2);

    g.addEdge(2, 1);

    g.addEdge(2, 3);

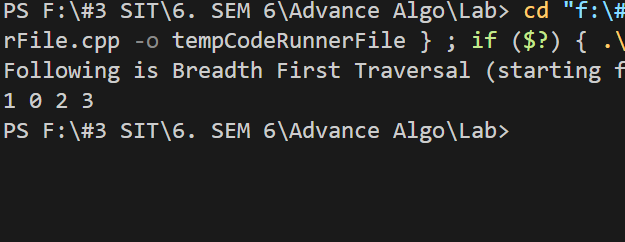
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal (starting from vertex 1) \n";

    g.BFS(1);

*return* 0;

}



**T(n) ANALYSIS WITH PROOF**

Time Complexity - O(V + E)

**SPACE COMPLEXITY ANALYSIS**

O(V)

**ADVANTAGES / DISADVANTAGES**

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| BFS can be used to find the shortest path between two vertices in an unweighted graph. Since BFS explores the graph in breadth-first order, it guarantees that the shortest path will be found when the destination vertex is reached. | BFS uses a queue data structure to keep track of vertices to be visited, which can require a large amount of memory, especially for large graphs with many vertices. The space complexity of BFS is O(V), where V is the number of vertices in the graph. |
| BFS is a complete algorithm, meaning it will always find a solution if one exists. If there is a path between the source and destination vertices, BFS will find it. | BFS is primarily designed for unweighted graphs, where all the edges have the same weight. It is not directly applicable to finding the shortest path in weighted graphs without modifications, as it does not take edge weights into account. |
| BFS visits vertices at the same level before moving to the next level. This property makes BFS suitable for certain problems where level-by-level exploration is desired, such as finding all the connected components of a graph. | BFS may visit some vertices multiple times, leading to redundant exploration and increased computation, especially in graphs with cycles or disconnected components. |

**REAL LIFE APPLICATIONS:**

1. Web Crawling: BFS can be used to crawl and index the web by visiting websites and following links in a depth-first order.
2. Finding Connected Components: BFS can be used to find connected components in a graph, which can be useful in social network analysis and community detection.
3. Solving Puzzles: BFS can be used to solve puzzles such as the maze problem, the n-queens problem, and Sudoku.
4. Topological Sorting: BFS can be used to perform a topological sort of a directed acyclic graph, which is useful in scheduling tasks and building compilers.
5. Pathfinding: BFS can be used to find a path between two nodes in a graph, which is useful in navigation systems and route planning.
6. Detecting Cycles: BFS can be used to detect cycles in a graph, which is useful in detecting deadlocks in operating systems and databases.