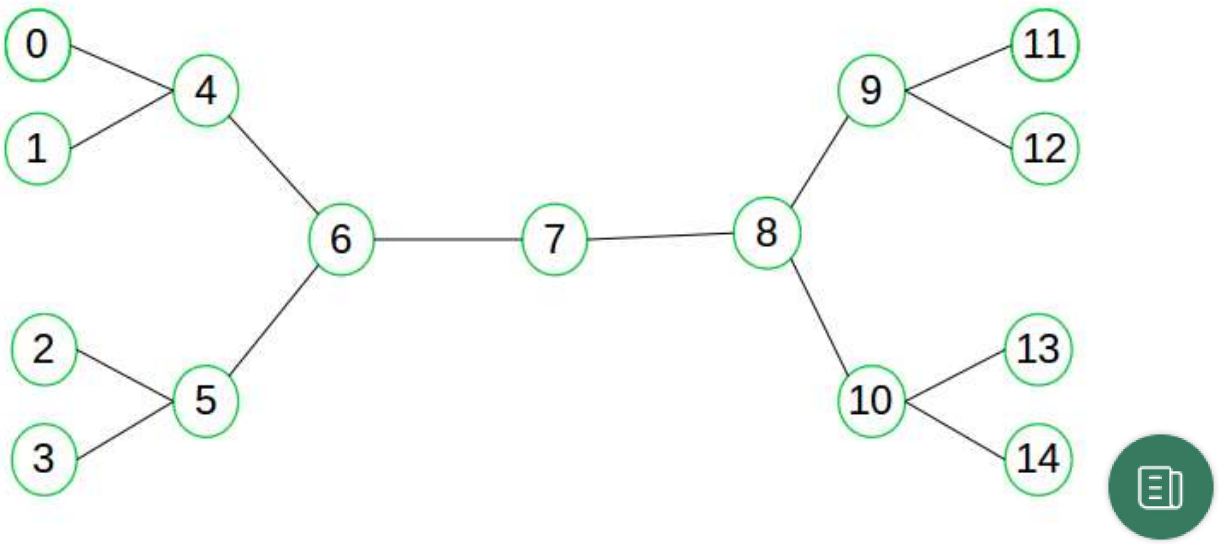
Bidirectional Search

Searching a graph is quite famous problem and have a lot of practical use. We have already discussed here how to search for a goal vertex starting from a source vertex using BFS. In normal graph search using BFS/DFS we begin our search in one direction usually from source vertex toward the goal vertex, **but what if we start search from both direction simultaneously.**
Bidirectional search is a graph search algorithm which find smallest path from source to goal vertex. It runs two simultaneous search –

1. Forward search from source/initial vertex toward goal vertex
2. Backward search from goal/target vertex toward source vertex

Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. **The search terminates when two graphs intersect.**
Just like A* algorithm, bidirectional search can be guided by a heuristic estimate of remaining distance from source to goal and vice versa for finding shortest path possible.
Consider following simple example-



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.

Why bidirectional approach?

Because in many cases it is faster, it dramatically reduce the amount of required exploration.
Suppose if branching factor of tree is **b** and distance of goal vertex from source is **d**, then the normal BFS/DFS searching complexity would be $O(b^d)$. On the other hand, if we execute two search operation then the complexity would be $O(b^{d/2})$ for each search and total complexity would be $O(b^{d/2} + b^{d/2})$ which is far less than $O(b^d)$.

When to use bidirectional approach?

We can consider bidirectional approach when-

1. Both initial and goal states are unique and completely defined.
2. The branching factor is exactly the same in both directions.

Performance measures

- Completeness : Bidirectional search is complete if BFS is used in both searches.
- Optimality : It is optimal if BFS is used for search and paths have uniform cost.

- Time and Space Complexity : Time and space complexity is $O(b^{d/2})$.

Below is very simple implementation representing the concept of bidirectional search using BFS. This implementation considers undirected paths without any weight.

C++

```
// C++ program for Bidirectional BFS search
// to check path between two vertices
#include <bits/stdc++.h>
using namespace std;

// class representing undirected graph
// using adjacency list
class Graph
{
    //number of nodes in graph
    int V;

    // Adjacency list
    list<int> *adj;
public:
    Graph(int V);
    int isIntersecting(bool *s_visited, bool *t_visited);
    void addEdge(int u, int v);
    void printPath(int *s_parent, int *t_parent, int s,
                            int t, int intersectNode);
    void BFS(list<int> *queue, bool *visited, int *parent);
    int biDirSearch(int s, int t);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
};

// Method for adding undirected edge
void Graph::addEdge(int u, int v)
{
    this->adj[u].push_back(v);
    this->adj[v].push_back(u);
};

// Method for Breadth First Search
void Graph::BFS(list<int> *queue, bool *visited,
                                    int *parent)
{
    int current = queue->front();
    queue->pop_front();
```

```
        list<int>::iterator i;
        for (i=adj[current].begin();i != adj[current].end();i++)
        {
            // If adjacent vertex is not visited earlier
            // mark it visited by assigning true value
            if (!visited[*i])
            {
                // set current as parent of this vertex
                parent[*i] = current;

                // Mark this vertex visited
                visited[*i] = true;

                // Push to the end of queue
                queue->push_back(*i);
            }
        }
    };

    // check for intersecting vertex
    int Graph::isIntersecting(bool *s_visited, bool *t_visited)
    {
        int intersectNode = -1;
        for(int i=0;i<V;i++)
        {
            // if a vertex is visited by both front
            // and back BFS search return that node
            // else return -1
            if(s_visited[i] && t_visited[i])
                return i;
        }
        return -1;
    };

    // Print the path from source to target
    void Graph::printPath(int *s_parent, int *t_parent,
                          int s, int t, int intersectNode)
    {
        vector<int> path;
        path.push_back(intersectNode);
        int i = intersectNode;
        while (i != s)
        {
            path.push_back(s_parent[i]);
            i = s_parent[i];
        }
        reverse(path.begin(), path.end());
        i = intersectNode;
        while(i != t)
        {
            path.push_back(t_parent[i]);
            i = t_parent[i];
        }
```

```
        vector<int>::iterator it;
        cout<<"*****Path*****\n";
        for(it = path.begin();it != path.end();it++)
            cout<<*it<<" ";
        cout<<"\n";
    };

    // Method for bidirectional searching
    int Graph::biDirSearch(int s, int t)
    {
        // boolean array for BFS started from
        // source and target(front and backward BFS)
        // for keeping track on visited nodes
        bool s_visited[V], t_visited[V];

        // Keep track on parents of nodes
        // for front and backward search
        int s_parent[V], t_parent[V];

        // queue for front and backward search
        list<int> s_queue, t_queue;

        int intersectNode = -1;

        // necessary initialization
        for(int i=0; i<V; i++)
        {
            s_visited[i] = false;
            t_visited[i] = false;
        }

        s_queue.push_back(s);
        s_visited[s] = true;

        // parent of source is set to -1
        s_parent[s]=-1;

        t_queue.push_back(t);
        t_visited[t] = true;

        // parent of target is set to -1
        t_parent[t] = -1;

        while (!s_queue.empty() && !t_queue.empty())
        {
            // Do BFS from source and target vertices
            BFS(&s_queue, s_visited, s_parent);
            BFS(&t_queue, t_visited, t_parent);

            // check for intersecting vertex
            intersectNode = isIntersecting(s_visited, t_visited);

            // If intersecting vertex is found
            // that means there exist a path
```

```
            if(intersectNode != -1)
            {
                cout << "Path exist between " << s << " and "
                     << t << "\n";
                cout << "Intersection at: " << intersectNode << "\n";

                // print the path and exit the program
                printPath(s_parent, t_parent, s, t, intersectNode);
                exit(0);
            }
        }
        return -1;
    }

    // Driver code
    int main()
    {
        // no of vertices in graph
        int n=15;

        // source vertex
        int s=0;

        // target vertex
        int t=14;

        // create a graph given in above diagram
        Graph g(n);
        g.addEdge(0, 4);
        g.addEdge(1, 4);
        g.addEdge(2, 5);
        g.addEdge(3, 5);
        g.addEdge(4, 6);
        g.addEdge(5, 6);
        g.addEdge(6, 7);
        g.addEdge(7, 8);
        g.addEdge(8, 9);
        g.addEdge(8, 10);
        g.addEdge(9, 11);
        g.addEdge(9, 12);
        g.addEdge(10, 13);
        g.addEdge(10, 14);
        if (g.biDirSearch(s, t) == -1)
            cout << "Path don't exist between "
                 << s << " and " << t << "\n";

        return 0;
    }
```

Java

```
// Java program for Bidirectional BFS search
// to check path between two vertices
```

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

// class representing undirected graph
// using adjacency list
class Graph {
    // number of nodes in graph
    private int V;

    // Adjacency list
    private LinkedList<Integer>[] adj;

    // Constructor
    @SuppressWarnings("unchecked") public Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++)
            adj[i] = new LinkedList<Integer>();
    }

    // Method for adding undirected edge
    public void addEdge(int u, int v)
    {
        adj[u].add(v);
        adj[v].add(u);
    }

    // Method for Breadth First Search
    public void bfs(Queue<Integer> queue, Boolean[] visited,
                    int[] parent)
    {
        int current = queue.poll();
        for (int i : adj[current]) {
            // If adjacent vertex is not visited earlier
            // mark it visited by assigning true value
            if (!visited[i]) {
                // set current as parent of this vertex
                parent[i] = current;

                // Mark this vertex visited
                visited[i] = true;

                // Push to the end of queue
                queue.add(i);
            }
        }
    }

    // check for intersecting vertex
    public int isIntersecting(Boolean[] s_visited,
                              Boolean[] t_visited)
    {
        for (int i = 0; i < V; i++) {
```

```
            // if a vertex is visited by both front
            // and back BFS search return that node
            // else return -1
            if (s_visited[i] && t_visited[i])
                return i;
        }
        return -1;
    }

    // Print the path from source to target
    public void printPath(int[] s_parent, int[] t_parent,
                          int s, int t, int intersectNode)
    {
        LinkedList<Integer> path
            = new LinkedList<Integer>();
        path.add(intersectNode);
        int i = intersectNode;
        while (i != s) {
            path.add(s_parent[i]);
            i = s_parent[i];
        }
        Collections.reverse(path);
        i = intersectNode;
        while (i != t) {
            path.add(t_parent[i]);
            i = t_parent[i];
        }

        System.out.println("*****Path*****");
        for (int it : path)
            System.out.print(it + " ");
        System.out.println();
    }

    // Method for bidirectional searching
    public int biDirSearch(int s, int t)
    {
        // Booleanean array for BFS started from
        // source and target(front and backward BFS)
        // for keeping track on visited nodes
        Boolean[] s_visited = new Boolean[V];
        Boolean[] t_visited = new Boolean[V];

        // Keep track on parents of nodes
        // for front and backward search
        int[] s_parent = new int[V];
        int[] t_parent = new int[V];

        // queue for front and backward search
        Queue<Integer> s_queue = new LinkedList<Integer>();
        Queue<Integer> t_queue = new LinkedList<Integer>();

        int intersectNode = -1;
```

```
        // necessary initialization
        for (int i = 0; i < V; i++) {
            s_visited[i] = false;
            t_visited[i] = false;
        }

        s_queue.add(s);
        s_visited[s] = true;

        // parent of source is set to -1
        s_parent[s] = -1;

        t_queue.add(t);
        t_visited[t] = true;

        // parent of target is set to -1
        t_parent[t] = -1;

        while (!s_queue.isEmpty() && !t_queue.isEmpty()) {
            // Do BFS from source and target vertices
            bfs(s_queue, s_visited, s_parent);
            bfs(t_queue, t_visited, t_parent);

            // check for intersecting vertex
            intersectNode
                = isIntersecting(s_visited, t_visited);

            // If intersecting vertex is found
            // that means there exist a path
            if (intersectNode != -1) {
                System.out.printf(
                    "Path exist between %d and %d\n", s, t);
                System.out.printf("Intersection at: %d\n",
                                  intersectNode);

                // print the path and exit the program
                printPath(s_parent, t_parent, s, t,
                          intersectNode);
                System.exit(0);
            }
        }
        return -1;
    }
}

public class GFG {
    // Driver code
    public static void main(String[] args)
    {
        // no of vertices in graph
        int n = 15;

        // source vertex
        int s = 0;
```

```
        // target vertex
        int t = 14;

        // create a graph given in above diagram
        Graph g = new Graph(n);
        g.addEdge(0, 4);
        g.addEdge(1, 4);
        g.addEdge(2, 5);
        g.addEdge(3, 5);
        g.addEdge(4, 6);
        g.addEdge(5, 6);
        g.addEdge(6, 7);
        g.addEdge(7, 8);
        g.addEdge(8, 9);
        g.addEdge(8, 10);
        g.addEdge(9, 11);
        g.addEdge(9, 12);
        g.addEdge(10, 13);
        g.addEdge(10, 14);
        if (g.biDirSearch(s, t) == -1)
            System.out.printf(
                "Path don't exist between %d and %d", s, t);
    }
}

// This code is contributed by cavi4762.
```

Python3

```
# Python3 program for Bidirectional BFS
# Search to check path between two vertices

# Class definition for node to
# be added to graph
class AdjacentNode:

    def __init__(self, vertex):

        self.vertex = vertex
        self.next = None

# BidirectionalSearch implementation
class BidirectionalSearch:

    def __init__(self, vertices):

        # Initialize vertices and
        # graph with vertices
        self.vertices = vertices
        self.graph = [None] * self.vertices

        # Initializing queue for forward
```

```
        # and backward search
        self.src_queue = list()
        self.dest_queue = list()

        # Initializing source and
        # destination visited nodes as False
        self.src_visited = [False] * self.vertices
        self.dest_visited = [False] * self.vertices

        # Initializing source and destination
        # parent nodes
        self.src_parent = [None] * self.vertices
        self.dest_parent = [None] * self.vertices

    # Function for adding undirected edge
    def add_edge(self, src, dest):

        # Add edges to graph

        # Add source to destination
        node = AdjacentNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node

        # Since graph is undirected add
        # destination to source
        node = AdjacentNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node

    # Function for Breadth First Search
    def bfs(self, direction = 'forward'):

        if direction == 'forward':

            # BFS in forward direction
            current = self.src_queue.pop(0)
            connected_node = self.graph[current]

            while connected_node:
                vertex = connected_node.vertex

                if not self.src_visited[vertex]:
                    self.src_queue.append(vertex)
                    self.src_visited[vertex] = True
                    self.src_parent[vertex] = current

                connected_node = connected_node.next
        else:

            # BFS in backward direction
            current = self.dest_queue.pop(0)
            connected_node = self.graph[current]
```

```
            while connected_node:
                vertex = connected_node.vertex

                if not self.dest_visited[vertex]:
                    self.dest_queue.append(vertex)
                    self.dest_visited[vertex] = True
                    self.dest_parent[vertex] = current

                connected_node = connected_node.next

    # Check for intersecting vertex
    def is_intersecting(self):

        # Returns intersecting node
        # if present else -1
        for i in range(self.vertices):
            if (self.src_visited[i] and
                self.dest_visited[i]):
                return i

        return -1

    # Print the path from source to target
    def print_path(self, intersecting_node,
                   src, dest):

        # Print final path from
        # source to destination
        path = list()
        path.append(intersecting_node)
        i = intersecting_node

        while i != src:
            path.append(self.src_parent[i])
            i = self.src_parent[i]

        path = path[::-1]
        i = intersecting_node

        while i != dest:
            path.append(self.dest_parent[i])
            i = self.dest_parent[i]

        print("*****Path*****")
        path = list(map(str, path))

        print(' '.join(path))

    # Function for bidirectional searching
    def bidirectional_search(self, src, dest):

        # Add source to queue and mark
        # visited as True and add its
        # parent as -1
```

```
            self.src_queue.append(src)
            self.src_visited[src] = True
            self.src_parent[src] = -1

            # Add destination to queue and
            # mark visited as True and add
            # its parent as -1
            self.dest_queue.append(dest)
            self.dest_visited[dest] = True
            self.dest_parent[dest] = -1

            while self.src_queue and self.dest_queue:

                # BFS in forward direction from
                # Source Vertex
                self.bfs(direction = 'forward')

                # BFS in reverse direction
                # from Destination Vertex
                self.bfs(direction = 'backward')

                # Check for intersecting vertex
                intersecting_node = self.is_intersecting()

                # If intersecting vertex exists
                # then path from source to
                # destination exists
                if intersecting_node != -1:
                    print(f"Path exists between {src} and {dest}")
                    print(f"Intersection at : {intersecting_node}")
                    self.print_path(intersecting_node,
                                    src, dest)
                    exit(0)
            return -1

    # Driver code
    if __name__ == '__main__':

        # Number of Vertices in graph
        n = 15

        # Source Vertex
        src = 0

        # Destination Vertex
        dest = 14

        # Create a graph
        graph = BidirectionalSearch(n)
        graph.add_edge(0, 4)
        graph.add_edge(1, 4)
        graph.add_edge(2, 5)
        graph.add_edge(3, 5)
        graph.add_edge(4, 6)
```

```
    graph.add_edge(5, 6)
    graph.add_edge(6, 7)
    graph.add_edge(7, 8)
    graph.add_edge(8, 9)
    graph.add_edge(8, 10)
    graph.add_edge(9, 11)
    graph.add_edge(9, 12)
    graph.add_edge(10, 13)
    graph.add_edge(10, 14)

    out = graph.bidirectional_search(src, dest)

    if out == -1:
        print(f"Path does not exist between {src} and {dest}")

# This code is contributed by Nirjhari Jankar
```

C#

```
// C# program for Bidirectional BFS search
// to check path between two vertices

using System;
using System.Collections.Generic;

// class representing undirected graph
// using adjacency list
public class Graph {
    // number of nodes in graph
    private int V;

    // Adjacency list
    private List<int>[] adj;

    // Constructor
    public Graph(int v)
    {
        V = v;
        adj = new List<int>[ v ];
        for (int i = 0; i < v; i++)
            adj[i] = new List<int>();
    }

    // Method for adding undirected edge
    public void AddEdge(int u, int v)
    {
        adj[u].Add(v);
        adj[v].Add(u);
    }

    // Method for Breadth First Search
    public void BFS(Queue<int> queue, bool[] visited,
                    int[] parent)
```

```
{
    int current = queue.Dequeue();
    foreach(int i in adj[current])
    {
        // If adjacent vertex is not visited earlier
        // mark it visited by assigning true value
        if (!visited[i]) {
            // set current as parent of this vertex
            parent[i] = current;

            // Mark this vertex visited
            visited[i] = true;

            // Push to the end of queue
            queue.Enqueue(i);
        }
    }
}

// check for intersecting vertex
public int IsIntersecting(bool[] s_visited,
                          bool[] t_visited)
{
    for (int i = 0; i < V; i++) {
        // if a vertex is visited by both front
        // and back BFS search return that node
        // else return -1
        if (s_visited[i] && t_visited[i])
            return i;
    }
    return -1;
}

// Print the path from source to target
public void PrintPath(int[] s_parent, int[] t_parent,
                      int s, int t, int intersectNode)
{
    List<int> path = new List<int>();
    path.Add(intersectNode);
    int i = intersectNode;
    while (i != s) {
        path.Add(s_parent[i]);
        i = s_parent[i];
    }
    path.Reverse();
    i = intersectNode;
    while (i != t) {
        path.Add(t_parent[i]);
        i = t_parent[i];
    }

    Console.WriteLine("*****Path*****");
    foreach(int it in path) Console.Write(it + " ");
    Console.WriteLine();
```

```
    }

    // Method for bidirectional searching
    public int BiDirSearch(int s, int t)
    {
        // boolean array for BFS started from
        // source and target(front and backward BFS)
        // for keeping track on visited nodes
        bool[] s_visited = new bool[V];
        bool[] t_visited = new bool[V];

        // Keep track on parents of nodes
        // for front and backward search
        int[] s_parent = new int[V];
        int[] t_parent = new int[V];

        // queue for front and backward search
        Queue<int> s_queue = new Queue<int>();
        Queue<int> t_queue = new Queue<int>();

        int intersectNode = -1;

        // necessary initialization
        for (int i = 0; i < V; i++) {
            s_visited[i] = false;
            t_visited[i] = false;
        }

        s_queue.Enqueue(s);
        s_visited[s] = true;

        // parent of source is set to -1
        s_parent[s] = -1;

        t_queue.Enqueue(t);
        t_visited[t] = true;

        // parent of target is set to -1
        t_parent[t] = -1;

        while (s_queue.Count > 0 && t_queue.Count > 0) {
            // Do BFS from source and target vertices
            BFS(s_queue, s_visited, s_parent);
            BFS(t_queue, t_visited, t_parent);

            // check for intersecting vertex
            intersectNode
                = IsIntersecting(s_visited, t_visited);

            // If intersecting vertex is found
            // that means there exist a path
            if (intersectNode != -1) {
                Console.WriteLine(
                    "Path exist between {0} and {1}", s, t);
```

```
                Console.WriteLine("Intersection at: {0}",
                                  intersectNode);

                // print the path and exit the program
                PrintPath(s_parent, t_parent, s, t,
                          intersectNode);
                Environment.Exit(0);
            }
        }
        return -1;
    }
}

public class GFG {
    // Driver code
    static void Main(string[] args)
    {
        // no of vertices in graph
        int n = 15;

        // source vertex
        int s = 0;

        // target vertex
        int t = 14;

        // create a graph given in above diagram
        Graph g = new Graph(n);
        g.AddEdge(0, 4);
        g.AddEdge(1, 4);
        g.AddEdge(2, 5);
        g.AddEdge(3, 5);
        g.AddEdge(4, 6);
        g.AddEdge(5, 6);
        g.AddEdge(6, 7);
        g.AddEdge(7, 8);
        g.AddEdge(8, 9);
        g.AddEdge(8, 10);
        g.AddEdge(9, 11);
        g.AddEdge(9, 12);
        g.AddEdge(10, 13);
        g.AddEdge(10, 14);
        if (g.BiDirSearch(s, t) == -1)
            Console.WriteLine(
                "Path don't exist between {0} and {1}", s,
                t);
    }
}

// This code is contributed by cavi4762.
```

Javascript

```
class Graph {
    // Constructor to initialize the graph with V vertices.
    constructor(V) {
        this.V = V; // Number of vertices
        // Adjacency list representation
        this.adj = new Array(V).fill().map(() => []);
    }

    // Function to add an edge between vertices u and v.
    addEdge(u, v) {
        this.adj[u].push(v);
        this.adj[v].push(u);
    }

    // Breadth-First Search (BFS) starting from a given source node.
    BFS(queue, visited, parent) {
        const current = queue.shift();
        for (const neighbor of this.adj[current]) {
            if (!visited[neighbor]) {
                parent[neighbor] = current;
                visited[neighbor] = true;
                queue.push(neighbor);
            }
        }
    }

    // Check if there is an intersection between two BFS searches.
    isIntersecting(s_visited, t_visited) {
        for (let i = 0; i < this.V; i++) {
            if (s_visited[i] && t_visited[i]) {
                return i; // Return the intersecting node
            }
        }
        return -1; // No intersection found
    }

    // Print the path from source 's' to target 't' through the intersection node
    printPath(s_parent, t_parent, s, t, intersectNode) {
        const path = [];
        path.push(intersectNode);
        let i = intersectNode;
        while (i !== s) {
            path.push(s_parent[i]);
            i = s_parent[i];
        }
        path.reverse();
        i = intersectNode;
        while (i !== t) {
            path.push(t_parent[i]);
            i = t_parent[i];
        }
        console.log("*****Path*****");
        console.log(path.join(' '));
```

```
    }

    // Bidirectional search to find the shortest path between 's' and 't'.
    biDirSearch(s, t) {
        const s_visited = new Array(this.V).fill(false);
        const t_visited = new Array(this.V).fill(false);
        const s_parent = new Array(this.V).fill(-1);
        const t_parent = new Array(this.V).fill(-1);
        const s_queue = [];
        const t_queue = [];
        let intersectNode = -1;

        // Start BFS from the source node 's'.
        s_queue.push(s);
        s_visited[s] = true;
        s_parent[s] = -1;

        // Start BFS from the target node 't'.
        t_queue.push(t);
        t_visited[t] = true;
        t_parent[t] = -1;

        // Continue BFS until an intersection is found or both searches are exhau
        while (s_queue.length > 0 && t_queue.length > 0) {
            this.BFS(s_queue, s_visited, s_parent);
            this.BFS(t_queue, t_visited, t_parent);
            intersectNode = this.isIntersecting(s_visited, t_visited);

            if (intersectNode !== -1) {
                console.log(`Path exists between ${s} and ${t}`);
                console.log(`Intersection at: ${intersectNode}`);
                this.printPath(s_parent, t_parent, s, t, intersectNode);
                return;
            }
        }
        console.log(`Path does not exist between ${s} and ${t}`);
    }
}

// Driver code
const n = 15;
const s = 0;
const t = 14;
const g = new Graph(n);

// Adding edges to the graph.
g.addEdge(0, 4);
g.addEdge(1, 4);
g.addEdge(2, 5);
g.addEdge(3, 5);
g.addEdge(4, 6);
g.addEdge(5, 6);
g.addEdge(6, 7);
g.addEdge(7, 8);
```

```
g.addEdge(8, 9);
g.addEdge(8, 10);
g.addEdge(9, 11);
g.addEdge(9, 12);
g.addEdge(10, 13);
g.addEdge(10, 14);

// Perform bidirectional search to find the path from 's' to 't'.
g.biDirSearch(s, t);
```


Output:

```
Path exist between 0 and 14
Intersection at: 7
*****Path*****
0 4 6 7 8 10 14
```

References

- https://en.wikipedia.org/wiki/Bidirectional_search

34

Previous

Best First Search (Informed Search)

Next

Assign Mice to Holes

Similar Reads

Print adjacency list of a Bidirectional Graph

Difference between the shortest and second shortest path in an Unweighted Bidirectional Graph

Shortest distance between given nodes in a bidirectional weighted graph by removing any K edges

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

Meta Binary Search | One-Sided Binary Search

Anagram Substring Search (Or Search for all permutations) | Set 2

Is Sentinel Linear Search better than normal Linear Search?

Search N elements in an unbalanced Binary Search Tree in O(N * logM) time

Binary Search Tree vs Ternary Search Tree

Interpolation search vs Binary search

Complete Tutorials

Learn Algorithms with Javascript | DSA using JavaScript Tutorial

DSA Crash Course | Revision Checklist with Interview Guide

Learn Data Structures and Algorithms | DSA Tutorial

Mathematical and Geometric Algorithms - Data Structure and Algorithm Tutorials

Learn Data Structures with Javascript | DSA using JavaScript Tutorial

A Atul Kumar

Article Tags : Algorithms , DSA , Graph , Searching

Practice Tags : Algorithms, Graph, Searching

Additional Information

Random Number Generator

Random Password Generator