**Find if there is a path between two vertices in a directed graph**

We can either use Breadth First Search (BFS) or Depth First Search (DFS) to find path between two vertices. Take the first vertex as source in BFS (or DFS), follow the standard BFS (or DFS). If we see the second vertex in our traversal, then return true. Else return false.

**Strongly Connected Graph:**

A directed graph is strongly connected if there is a path between any two pair of vertices.

**How To Check If A Graph Is Strongly Connected Or Not In Case Of Undirected Graph:**

It is easy for undirected graph, we can just do a BFS and DFS starting from any vertex. If BFS or DFS visits all vertices, then the given undirected graph is connected. This approach won’t work for a directed graph. For example, consider the following graph which is not strongly connected. If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.

**How To Check If A Graph Is Strongly Connected Or Not In Case Of Directed Graph:**

**A simple idea** is to use a all pair shortest path algorithm like Floyd Warshall or find Transitive Closure of graph. Time complexity of this method would be O(v3).

**Another idea** is doing BFS or DFS for all vertices. If any DFS, doesn’t visit all vertices, then graph is not strongly connected. This algorithm takes O(V\*(V+E)) time which can be same as transitive closure for a dense graph.

**Far Better idea:**

1) Initialize all vertices as not visited.

2) Do a DFS traversal of graph starting from any arbitrary vertex v. If DFS traversal doesn’t visit all vertices, then return false.

3) Reverse all arcs (or find transpose or reverse of graph)

4) Mark all vertices as not-visited in reversed graph.

5) Do a DFS traversal of reversed graph starting from same vertex v (Same as step 2). If DFS traversal doesn’t visit all vertices, then return false. Otherwise return true.

The idea is, if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 4, we check if all vertices can reach v (In reversed graph, if all vertices are reachable from v, then all vertices can reach v in original graph)

**Articulation Points In A graph:**

A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

How to find all articulation points in a given graph?

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

1) For every vertex v, do following

…..a) Remove v from graph

..…b) See if the graph remains connected (We can either use BFS or DFS)

…..c) Add v back to the graph

Time complexity of above method is O(V\*(V+E)) for a graph represented using adjacency list. Can we do better?

A O(V+E) algorithm to find all Articulation Points (APs)

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1) u is root of DFS tree and it has at least two children.

2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

// A C++ program to find articulation points in an undirected graph

#include<iostream>

#include <list>

#define NIL -1

using namespace std;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

void APUtil(int v, bool visited[], int disc[], int low[],

int parent[], bool ap[]);

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // function to add an edge to graph

void AP(); // prints articulation points

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

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

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

}

// A recursive function that find articulation points using DFS traversal

// u --> The vertex to be visited next

// visited[] --> keeps tract of visited vertices

// disc[] --> Stores discovery times of visited vertices

// parent[] --> Stores parent vertices in DFS tree

// ap[] --> Store articulation points

void Graph::APUtil(int u, bool visited[], int disc[],

int low[], int parent[], bool ap[])

{

// A static variable is used for simplicity, we can avoid use of static

// variable by passing a pointer.

static int time = 0;

// Count of children in DFS Tree

int children = 0;

// Mark the current node as visited

visited[u] = true;

// Initialize discovery time and low value

disc[u] = low[u] = ++time;

// Go through all vertices aadjacent to this

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

{

int v = \*i; // v is current adjacent of u

// If v is not visited yet, then make it a child of u

// in DFS tree and recur for it

if (!visited[v])

{

children++;

parent[v] = u;

APUtil(v, visited, disc, low, parent, ap);

// Check if the subtree rooted with v has a connection to

// one of the ancestors of u

low[u] = min(low[u], low[v]);

// u is an articulation point in following cases

// (1) u is root of DFS tree and has two or more chilren.

if (parent[u] == NIL && children > 1)

ap[u] = true;

// (2) If u is not root and low value of one of its child is more

// than discovery value of u.

if (parent[u] != NIL && low[v] >= disc[u])

ap[u] = true;

}

// Update low value of u for parent function calls.

else if (v != parent[u])

low[u] = min(low[u], disc[v]);

}

}

// The function to do DFS traversal. It uses recursive function APUtil()

void Graph::AP()

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

int \*disc = new int[V];

int \*low = new int[V];

int \*parent = new int[V];

bool \*ap = new bool[V]; // To store articulation points

// Initialize parent and visited, and ap(articulation point) arrays

for (int i = 0; i < V; i++)

{

parent[i] = NIL;

visited[i] = false;

ap[i] = false;

}

// Call the recursive helper function to find articulation points

// in DFS tree rooted with vertex 'i'

for (int i = 0; i < V; i++)

if (visited[i] == false)

APUtil(i, visited, disc, low, parent, ap);

// Now ap[] contains articulation points, print them

for (int i = 0; i < V; i++)

if (ap[i] == true)

cout << i << " ";

}

// Driver program to test above function

int main()

{

// Create graphs given in above diagrams

cout << "\nArticulation points in first graph \n";

Graph g1(5);

g1.addEdge(1, 0);

g1.addEdge(0, 2);

g1.addEdge(2, 1);

g1.addEdge(0, 3);

g1.addEdge(3, 4);

g1.AP();

cout << "\nArticulation points in second graph \n";

Graph g2(4);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.addEdge(2, 3);

g2.AP();

cout << "\nArticulation points in third graph \n";

Graph g3(7);

g3.addEdge(0, 1);

g3.addEdge(1, 2);

g3.addEdge(2, 0);

g3.addEdge(1, 3);

g3.addEdge(1, 4);

g3.addEdge(1, 6);

g3.addEdge(3, 5);

g3.addEdge(4, 5);

g3.AP();

return 0;

}

**Bi-connected Graph:**A connected graph is Biconnected if it is connected and doesn’t have any Articulation Point. We mainly need to check two things in a graph.

1) The graph is connected.

2) There is not articulation point in graph.

We start from any vertex and do DFS traversal. In DFS traversal, we check if there is any articulation point. If we don’t find any articulation point, then the graph is Biconnected. Finally, we need to check whether all vertices were reachable in DFS or not. If all vertices were not reachable, then the graph is not even connected.

**Bridges in a graph**

An edge in an undirected connected graph is a bridge iff removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components.

Like Articulation Points,bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

// A C++ program to find bridges in a given undirected graph

#include<iostream>

#include <list>

#define NIL -1

using namespace std;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

void bridgeUtil(int v, bool visited[], int disc[], int low[],

int parent[]);

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // to add an edge to graph

void bridge(); // prints all bridges

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

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

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

}

// A recursive function that finds and prints bridges using

// DFS traversal

// u --> The vertex to be visited next

// visited[] --> keeps tract of visited vertices

// disc[] --> Stores discovery times of visited vertices

// parent[] --> Stores parent vertices in DFS tree

void Graph::bridgeUtil(int u, bool visited[], int disc[],

int low[], int parent[])

{

// A static variable is used for simplicity, we can

// avoid use of static variable by passing a pointer.

static int time = 0;

// Mark the current node as visited

visited[u] = true;

// Initialize discovery time and low value

disc[u] = low[u] = ++time;

// Go through all vertices aadjacent to this

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

{

int v = \*i; // v is current adjacent of u

// If v is not visited yet, then recur for it

if (!visited[v])

{

parent[v] = u;

bridgeUtil(v, visited, disc, low, parent);

// Check if the subtree rooted with v has a

// connection to one of the ancestors of u

low[u] = min(low[u], low[v]);

// If the lowest vertex reachable from subtree

// under v is below u in DFS tree, then u-v

// is a bridge

if (low[v] > disc[u])

cout << u <<" " << v << endl;

}

// Update low value of u for parent function calls.

else if (v != parent[u])

low[u] = min(low[u], disc[v]);

}

}

// DFS based function to find all bridges. It uses recursive

// function bridgeUtil()

void Graph::bridge()

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

int \*disc = new int[V];

int \*low = new int[V];

int \*parent = new int[V];

// Initialize parent and visited arrays

for (int i = 0; i < V; i++)

{

parent[i] = NIL;

visited[i] = false;

}

// Call the recursive helper function to find Bridges

// in DFS tree rooted with vertex 'i'

for (int i = 0; i < V; i++)

if (visited[i] == false)

bridgeUtil(i, visited, disc, low, parent);

}

// Driver program to test above function

int main()

{

// Create graphs given in above diagrams

cout << "\nBridges in first graph \n";

Graph g1(5);

g1.addEdge(1, 0);

g1.addEdge(0, 2);

g1.addEdge(2, 1);

g1.addEdge(0, 3);

g1.addEdge(3, 4);

g1.bridge();

cout << "\nBridges in second graph \n";

Graph g2(4);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.addEdge(2, 3);

g2.bridge();

cout << "\nBridges in third graph \n";

Graph g3(7);

g3.addEdge(0, 1);

g3.addEdge(1, 2);

g3.addEdge(2, 0);

g3.addEdge(1, 3);

g3.addEdge(1, 4);

g3.addEdge(1, 6);

g3.addEdge(3, 5);

g3.addEdge(4, 5);

g3.bridge();

return 0;

}