BridgeEdge.cpp

```cpp
// 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);
```

```cpp
            // 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);
}
int main() {
    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();
}
```