Find Biconnected Components in Undirected Graph

Name: Junjie Feng NetID: jf736 RUID: 165004173

1. Problem:

In graph theory, A connected graph is biconnected if the removal of any single vertex (and all edges incident on that vertex) can not disconnect the graph. More generally, the biconnected components of a graph are the maximal subsets of vertices such that the removal of a vertex from a particular component will not disconnect the component. A biconnected component (also known as a block or 2-connected component) is a maximal biconnected subgraph.

Specifically, a cut vertex (also known as articulation point or separating point) is any vertex whose removal increases the number of connected components. In other words, any two components are attached to each other at one of the cut vertices or have totally different set of vertices. Any connected graph decomposes into a tree of biconnected components called the block-cut tree of the graph.

How to determine the graph is biconnected or not? How to find all the biconnected components? This paper will introduce an algorithm to solve this problem.

2. Algorithm:

The idea of my algorithm comes from Tarjan's strongly connected components algorithm[4]. The problem of finding biconnected components in undirected graph just like an undirected-graph version of finding strong connected components in digraph.

Tarjan's algorithm takes a directed graph as input, and produces a partition of the graph's vertices into the graph's strongly connected components. Each vertex of the graph appears in exactly one of the strongly connected components.

For my problem, my algorithm takes an undirected graph as input, and produces a partition of graph's vertices into graph's biconnected components. However, the difference between my algorithm and Tarjan's algorithm is that the biconnected components may share the cut vertices. In other words, each cut vertex must occur more than one component.

The basic idea of my algorithm is this: a depth-first search starts with a random vertex and the subsequent depth-first search are recursively conducted on another random vertex which is connected to the current one and not visited yet. As the typical depth-first search does, the function visit each vertex exactly once. The biconnected component will be recovered as a certain part of the DFS search tree.

2.1 Core Function:

The core function of the algorithm is shown as follow:

```
void dfs_tarjan(int v){
        depth++;
        preorder[v] = depth;
        low[v] = depth;
        st.push(v);
        for(int i=0;i<graphs[v].size();i++){
                edge* e = graphs[v][i];
                int u = e->v1==v? e->v2:e->v1;
                if(!e->visited){
                        e->visited= true;
                        if(!preorder[u]){ // if the vertex u is not visited before
                                leaf[v] = false;
                                dfs_tarjan(u);
                                if(low[u]<low[v])  // update the low[v] if its child's low is smaller
                                        low[v]=low[u];
                                if(low[u]>=preorder[v]){ // this means vertex v is a cut vertex
                                        cutCnt[v]++;
                                        vector<int> tmp;
                                        tmp.push_back(v);
                                        tmp.push_back(st.top());
                                        while(st.top() != u){
                                                st.pop();
                                                tmp.push_back(st.top());
                                        }
                                        st.pop();
                                        components.push_back(tmp);
                                }
                        }
                        else if(preorder[u]<low[v])  // the back edge is found. update the low[v]
                                low[v] = preorder[u];
                }
        }
}
```

2.2 Data Structure:

(If you could not understand how the struct is used, you could temporarily ignore it, and read the next part which explain how the program runs.)

The "**edge**" is a struct which stores the two endpoints' value, edge's weight, and a boolean value to record the edge is visited or not. Each edge will only be considered once.

The "**graphs**" is a map structure, which is a adjacency-list graph representation. Each list is a vector of edges. Since the input graph is undirected, each edge will be added to two endpoints's lists respectively.

The “**st**" is a vertex stack, which tracks the visited vertices. The vertices are pushed into the stack in the order in which they are visited. When a cut vertex is found, the vertices which are pushed into the stack before the cut vertex will be popped out and constitute a biconnected component.

The “**depth**” is a integer value, which indicates the how many vertices have been visited. So it will increment by 1 when the dfs_tarjan function is called. It is initialized as 0.

The “**preorder**” is a vector of integers, which is used to track the preorder of each vertex. All the entries is initialized as 0. Once a dfs function calls on vertex v, low[v] set to depth.

The “**low**” is also a vector of integers, which is used to record the low value of each vertex. For each vertex v, the lowest “preorder” of neighbors of all descendant of v in the depth-first-search tree, called “low”. All the entries is initialized as 0. Once a dfs function calls on vertex v, low[v] set to depth.

The “**cutCnt**” is a vector of integer 0. when the vertex is determined as a cut vertex, its corresponding entry will increment by 1. After the traversing the whole graph, the program will check each entry of “cutCnt”. If a entry is larger than 0, the corresponding vertex is a cut vertex, otherwise, the corresponding vertex is not. However, there is one exception. For the first visited vertex, if its’ entry is larger than 1, it is a cut vertex; otherwise, it is not.

The “**components**” is a 2 dimension matrix, which stores the each biconnected component and each component stores as one row. Every time when a component is found, the vertices of it will be added into one row of “components” matrix.

2.3 Algorithm Detail & Explanation:

As the definition of biconnected component says, removal of any single vertex can not disconnect the graph. That also means, for any two vertices, there are at least two unique paths between them. When a vertex in one of these paths is removed, this path would disconnect, but there is still at least one path connects those two vertices. So, I could make a conclusion: a non-root vertex *v* is a cut vertex separating two biconnected components if and only if there is a child *u* of *v* such that low[*u]* ≥ preorder[*v]*. Because when such condition is satisfied, it indicates that v’s child vertex u does not have a path which connects to a more ancient ancestor vertex. Suppose we remove the current vertex from the graph, this child vertex and the child’s descendants would be disconnected.

Running the DFS algorithm on the graph would find the a DFS tree of the graph. All the edges in the origin graph can be divided into two categories after DFS. One is tree edges which are visited during the DFS recursion calls and shown in the tree. The other is back edges which are not visited during the DFS recursion calls because the endpoint of the back edges are definitely visited.

Following tree edges to traverse the graph, the “preorder” value of each vertex would increment 1 by 1. As the definition of “low” says: for each vertex v, the lowest “preorder” of neighbors of all

descendant of v in the depth-first-search tree, called “low”. So the “low” value would probably be changed when a back edge is found, because the back edge connects to an already visited vertex u and preorder[u] may smaller than low[v].

The root vertex must be handled separately: it is a cut vertex if and only if it has at least two children.

The algorithm roughly looks like DFS algorithm. It actually has the same visit order as DFS algorithm does. However, they shares no similarity in data structures. Now let us talk about the execution details of the algorithm.

The algorithm starts with a random vertex. First of all, the program sets the vertex’s “preorder” value and “low” value equals to the “depth”. The “depth” shows how many vertices has been visited already. For example, preorder[x] = 4, that means the vertex x is the 4th vertex be visited. The next instruction add the vertex at the tail of “st” stack. As a result, the “st” stack would contain all the vertices that has been visited.

Then the program would recursively visit the current vertex’s descendants. There are two conditions need to be met before recursively call. The first one is that the edge should not be visited before. This condition would limit access to the parent vertex. The second one is that the descendant vertex should not be visited before. I use “preorder” instead of “marked” in the typical DFS algorithm. Because the each vertex’s “preorder” value is initialized as 0, as long as the vertex has not been visited, its “preorder” value would be 0. Otherwise, it would be larger than 0.

Assume the current vertex is x, and it has a descendant vertex y which has already been visited. That means the edge which connects x and y is a back edge. The low[x] would be set equals to the preorder[y] if the preorder[y] is smaller than low[x].

After dfs_tarjan function call on vertex x is returned, its parent vertex z would set low[z] equals to low[x] if low[x] is smaller than low[z]. This instruction would recursively set the “low” value of all vertices from y to x, including z, to preorder[y] if they do not have any other back edge which connects to a more ancient ancestor vertex.

Next step of the algorithm, which is the most important step and the typical DFS search does not have, is to check whether the current vertex is a cut vertex or not. As i mentioned before, the key fact is that a non-root vertex *v* is a cut vertex separating two biconnected components if and only if there is a child *u* of *v* such that low[*u]* ≥ preorder[*v]*. Assume one of the current vertex’s child vertex dfs_tarjan call returned. If the preorder[current_vertex] is larger than or equals to the low[child], the current vertex is a cut vertex.

Once a vertex is decided to be a cut vertex, its “cutCnt”’s corresponding entry will increment by 1. The vertices which are visited after the current one will be popped out from “st” stack and grouped as one biconnected component.

So far, we have found all the cut vertices and vertices that make up each component. The last part of the algorithm is to put each edge into one component which it should belong to.

In order to do this, the algorithm is shown as the following pseudocode. It re-run the DFS algorithm on the input graph. However, before recursively call DFS functions, it checks whether the vertex u belongs to the current biconnected component or not. If it is out of the range of current component, just ignore it and check the next vertex, otherwise, call the DFS function on it. For each component, the main function need to call one DFS function on a random vertex.

```
DFS(vertex v){
        marked[v] = true;
        for(each vertex u connects to v){
                if(edge (v, u) is unvisited){ // void visiting the parent vertex
                        if(vertex u belongs to this component){
                        // void visiting the vertex out of the current component
                                set the edge as visited & put this edge into this component
                                if(!marked[u])
                                        DFS(u);
        }
}
```

2.4 Time Complexity & Spatial Complexity & Why I Choose This Algorithm

Before the final decision of using this algorithm, I also think about the backtrack version algorithm and dynamic programming version algorithm. However, the backtracking algorithm take much more running time than this one. The dynamic programming algorithm is too complicated to implement and I failed to do so.

For my algorithm, it based on depth first search algorithm and Tarjan's strongly connected component algorithm. Both of these algorithms take O(V+E) time complexity.
My algorithm call dfs_tarjan function once on each node. The stack push and pop each edge once. Any other instructions would be done in constant time. So my algorithm has the same time complexity, O(E+V).

The largest data structure is the "graphs", which takes O(E+V) space. The other several data structures take O(V) or O(E) space. So the total spatial complexity is O(E+V).

In general, the algorithm takes linear (in number of edges and vertices) time complexity and spatial complexity.

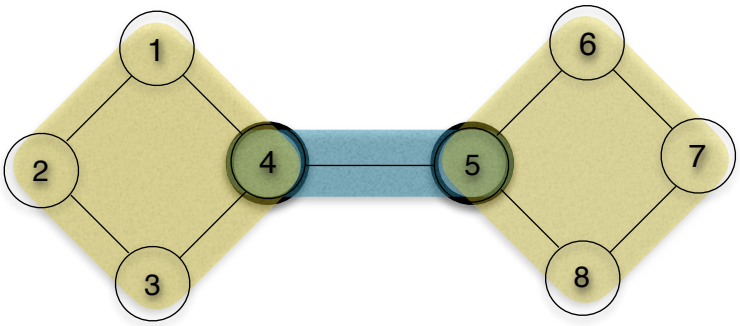
3. Experiment Result & Analysis

I generate several graphs, and each contains one or more biconnected components. I would show two of them to prove the correctness of my algorithm.

The first graph is pretty simple and easy understanding. This undirected graph is connected but not biconnected. It has 8 vertices and 9 edges. In theory, this graph has 3 biconnected components. The cut vertices are 4 and 5. I use different colors to indicate different

components, and draw circles around the cut vertices. As the screenshot of my program's output shows, my experimental result is consistent with theoretical result.

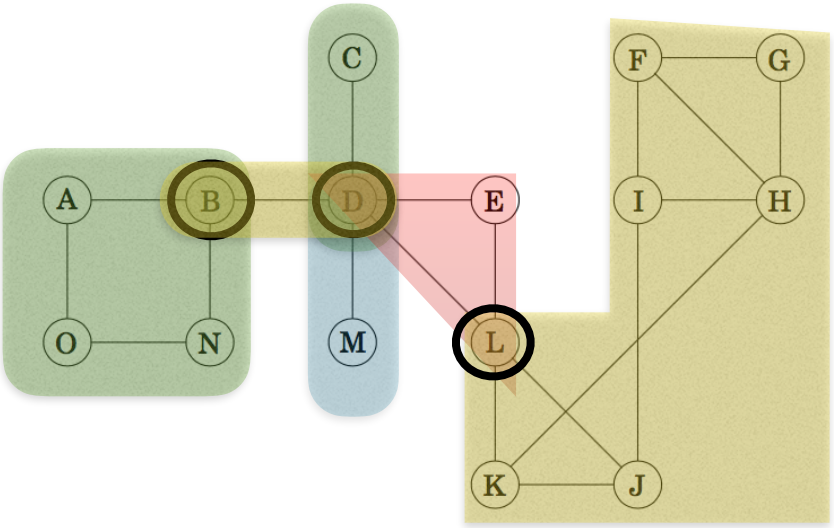
When the biconnected graph only has one edge in it, this component also called bridge. My algorithm could also find out these special components.




```
Cut Vertices:
4
5

Vertex Biconnected Components:
{  5  8  7  6  }
{  4  5  }
{  1  4  3  2  }

Edge Biconnected Components:
{  (5,6)  (6,7)  (7,8)  (5,8)  }
{  (4,5)  }
{  (1,2)  (2,3)  (3,4)  (1,4)  }

Bridges:
(4,5)
```

The second graph comes from the book called "Algorithm"[2], page 107th. This undirected graph is also connected but not biconnected. It has 15 vertices and 20 edges. In theory, this graph has 6 biconnected components. The cut vertices are B, D, L. I use different colors to indicate different components, and draw circles around the cut vertices. As the screenshot shows, my experimental result is consistent with theoretical result.

I upload all my code and testing inputs on Github. If you are interested in my code, you could download and run it from this link: https://github.com/Junjie-Feng/BiconnectedComponent . You can also use your graph as test file, and my program can deal with both character and integer vertex value.




```
Cut Vertices:
B
D
L

Vertex Biconnected Components:
{  D  C  }
{  L  G  F  I  H  K  J  }
{  D  L  E  }
{  D  M  }
{  B  D  }
{  A  O  N  B  }

Edge Biconnected Components:
{  (C,D)  }
{  (L,J)  (K,J)  (L,K)  (K,H)  (I,H)  (J,I)  (I,F)  (F,G)  (G,H)  (F,H)  }
{  (D,E)  (E,L)  (L,D)  }
{  (D,M)  }
{  (B,D)  }
{  (A,B)  (A,O)  (N,O)  (B,N)  (N,O)  }

Bridges:
(C,D)
(D,M)
(B,D)
```

4. Discusion

Actually, the bridge is an interesting component.

First, let us recall the conditional statement I used to check whether a vertex is cut vertex or not: low[u] >= preorder[v], u is one of the descendant vertex of v. However only when the edge (u,v) is a bridge, low[u] would be larger than preorder. Otherwise, when low[u] equals to preorder[v], u and v would be within a component with more than 1 edge (or more than 2 vertices).

Second, unlike any other edges, the removal of a bridge edge would disconnect the graph, just like the removal of the cut vertex does.

Because of these two reasons, I also record the bridges while finding the biconnected components.

5. Reference:

[1] https://en.wikipedia.org/wiki/Biconnected_component
[2] Book: “Algorithms”. S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, July 18, 2006

[3] http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/biconnected_components.html
[4] https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
[5] http://blog.csdn.net/fuyukai/article/details/51303292
[6] https://www.byvoid.com/zhs/blog/biconnect
[7] Book: “Algorithms, fourth edition”, Robert Sedgewick and Kevin Wayne, Princeton University