Extraction of blocks by identifying split vertices & conversion of circuit matrix to diagonal matrix

Maharshi Roy, Rajat Kumar Sahu, Amrit Daimary

ism2014006@iiita.ac.in
ihm2014501@iiita.ac.in
irm2014001@iiita.ac.in

Abstract—*The objective of this paper is to identify split vertices in a graph, called articulation points, and break down the graph into multiple subgraphs such that each one is a biconnected component or block. Thereafter, a circuit matrix is developed from those individual blocks and then the matrix is converted into a diagonal matrix form by identifying a suitable rearrangement of the edges which form the columns based on the blocks information.*

Keywords — *Articulation points, Cut-vertices, Biconnected Components, Circuit Matrix, Diagonal Matrix*

MOTIVATION

Finding articulation points or cut-vertices has been a very interesting topic in research areas, due to its vast applications in real-life scenarios involving connectivity. Articulation Points are the weak vertices of a graph, which if removed, can lead to the whole graph getting disconnected. This has practical applications in routing where we need to reinforce those nodes which upon failure may lead to a disconnectivity in a network. Also, it is found to be of great importance in power generation network, where we might need to place alternate transformers for those ones which upon failure can lead to no power availability for some parts of the grid. Thus, the concepts discussed here basically aims to find out the weak points/vertices in the graph, and divide the graph into blocks, such that each block is a biconnected component. (i.e. has no cut-vertices, each vertex upon removal doesn't split it further).

METHODOLOGY AND APPROACH

We will discuss two methods for identifying biconnected components, of which the second method is a better single-pass DFS method with $O(V+E)$ complexity. The graphs considered are undirected. i.e.

$$v \rightarrow u \Leftrightarrow u \rightarrow v$$

and we store the graph as an adjacency list, so as to avoid space wastage in case of sparse graphs.

Naive Method (Recursive Multiple Pass Method)

In the naive method, we check each vertex recursively for an articulation point. If we find a vertex as an articulation point, we separate the individual components, and recursively call the method for those components. The process of subdivision continues until we reach a stage where each component is biconnected and no articulation point remains inside each component.

Algorithm:-

```
ans = φ
function find_blocks(block B) {
       isblock=True
       for each vertex v of B {
            B_v = B - v
            if  (is_connected(B_v)) {
                 continue
             }
            else {
                 isblock=False
                DC = get_components (B_v )
                for each dc in DC {
                     find_blocks(dc)
                }
             }
        }

       if (isblock==True) {
          ans = ans ∪ B
       }
}
```

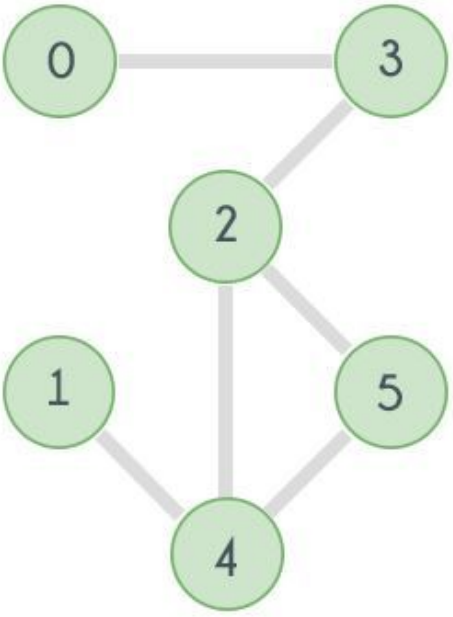
As explained in the algorithm, we receive a component on which we check it's connectivity by removing each vertex on by one. If for some vertex, the component splits up, we recursively call the same function for each individual component, and keep on doing so.The algorithm terminated when we are unable to find any split vertex and thus *isblock* boolean remains true. In such a case, we append that component to the solution set denoted as *ans*.

Single Pass DFS Method

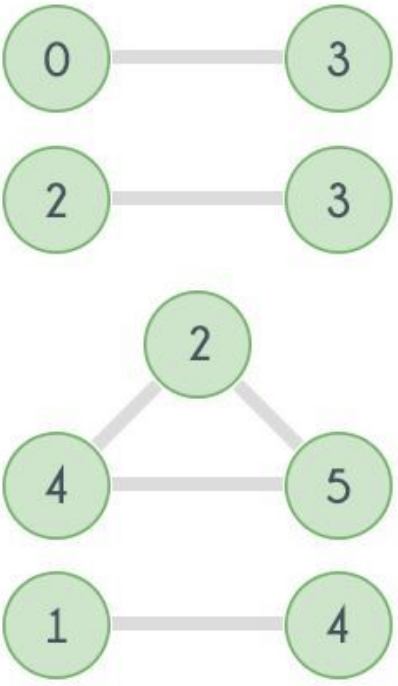
Unlike the previous Naive approach, this method finds out the articulation points and their corresponding biconnected components in a single pass. Since we represent our graph as an adjacency list, thus we expect the time complexity of this method as $O(V+E)$. The algorithm for the process is described below:-

Algorithm:-

```
time=0
stack st
function dfs (u)
{
    disc[u]=low[u]=(time++)
    visited[u]=true
    child=0
    for i=1 to sizeof (adj[u]) {
         v=adj[u][v]
         if (!visited[v]) {
              parent[v]=u
              child=child+1
              st.push((u,v))
              dfs(v)
              if (parent[u]!=NULL and
                  low[v]>=disc[u]) or
                 (parent[u]==NULL and child>1)) {
                        AP[u]=True
                        empty stack and get a
                        biconnected component
               }
               low[u]=min(low[u],low[v])
         }
         else if (parent[u]!=v and
                   disc[v]<low[u]) {
                   low[u]=disc[v]
                   s.push((u,v))
         }
     }
}
```



Original graph



Biconnected Components Present

Generating a Circuit Matrix

A circuit matrix represents the biconnected components in a more formal manner. The mathematical definition of a circuit matrix is as follows:-

$$M_{ij} = \begin{cases} 1, \text{ if edge } e_j \text{ is in circuit } C_j \\ 0, \text{ otherwise} \end{cases}$$

Thus, we categorically mark the biconnected components obtained C_1, C_2, C_3,.....,C_n. and form a matrix as shown below:-

	e_1	e_2	e_3	e_4	e_5
C_1		1	1		1
C_2	1		1	1	

Indicating that

$C_1 : \{e_2, e_3, e_5\}$
$C_2 : \{e_1, e_3, e_4\}$

are the edges corresponding to the respective circuits.

Conversion to Diagonal Matrix

With a proper rearrangement of the edges in the columns, we can have a diagonal matrix instead of an arbitrary column matrix as shown below:-

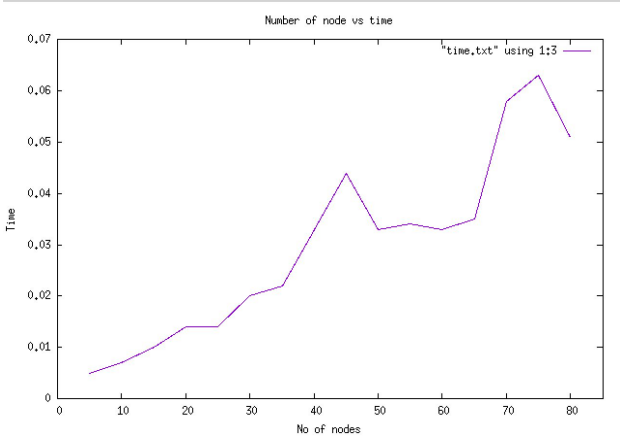
	e_2	e_5	e_3	e_4	e_1
C_1	1	1	1		
C_2			1	1	1

Algorithm:-

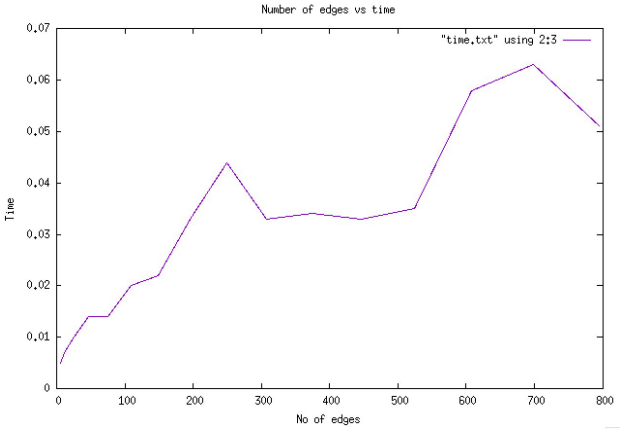
1) Accumulate for each circuit the edges present in it.
2) Rearrange the columns for the first circuit suitably, and mark all those entries corresponding to that circuit and the edges
3) Repeat 1-2 until no circuit remains

IMPLEMENTATION AND RESULTS

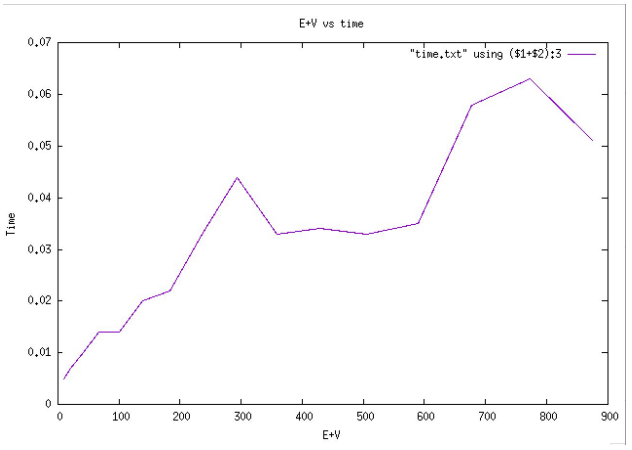
All the programs were run on multiple graphs to get an accurate description and analysis of time-complexity for each algorithm and the various sub-steps involved. The computation was performed in a standard home desktop pc with specifications: (2.00 GHz, 8GB RAM). The time complexity graphs for the algorithms are described below:-



No of Nodes vs Time



No of Edges vs Time



Edges + Vertices vs Time

CONCLUSION

We thus observe that the ***Single Pass DFS*** method performs well in identification of blocks or biconnected components. After identification of these circuits, we develop a circuit matrix, and convert it to a diagonal matrix. This helps us to easily locate the circuits and the corresponding edges, saving processing time.

REFERENCES

[1] "*Hackerearth Biconnected Components Tutorial*"
Link

[2] "*Graph Theory*", Keijo Ruohonen.
Link