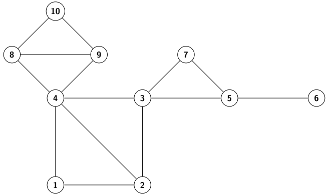
Finding the cut-vertices and biconnected components in an undirected graph

Ranveer
IIT Indore

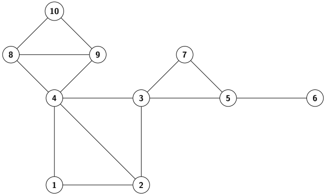
Connected graph

Let $G = (V, E)$ be an undirected graph, with vertex-set V and edge set E. Graph G is called *connected* when there is a path between every pair of vertices.



Cut-vertices

A *cut-vertex* in an undirected graph G is a vertex whose deletion creates more connected components than previously in the graph.



Vertices $3, 4, 5$ are the cut-vertices.

Finding Cut-vertices

Naive Approach: For every vertex $v \in G$: remove v from the corresponding component and check if the component remains connected (use Breadth-first search or Depth-First search). If the resulting subgraph is disconnected, add v to cut-vertex list.

Complexity: The time complexity of the above method is $O\big(|V| * (|V| + |E|)\big)$

Can we do better ?

Key observation

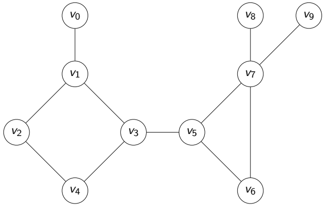
A vertex v in G is a cut-vertex if and only if there exist two other vertices x and y such that every path between x and y passes through v; in this case and only in this case the deletion of v from G destroy all paths between x and y.

This observation allow us to use depth-first search to find the cut-vertices of G in $O(|V| + |E|)$ operations.

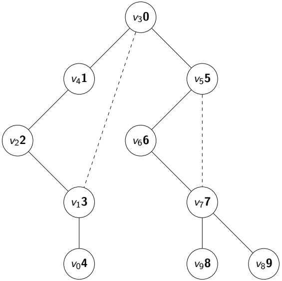
Depth-First Search (DFS)

1. Explore deeper in the graph whenever possible
2. Edges are explored out of the most recently discovered vertex v that still has unexplored edges
3. When all the incident edges of v have been explored, backtrack to the vertex from which v was discovered

Depth-first spanning tree



G



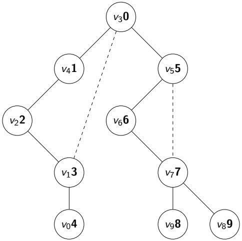
Depth-first spanning tree of G

In a Depth-first spanning tree, *depth first number (dfn)* of a vertex v, denoted by $dfn(v)$ is the discovery time of v (shown on the right side of the vertices in bold). Dashed edges are the back edges.

Vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
dfn	4	3	2	0	1	5	6	7	9	8

Cut-vertex

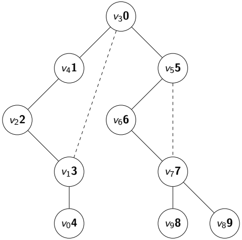
1. The root of a DFS-tree is a cut-vertex if and only if it has at least two children.
2. A nonroot vertex v of a DFS-tree is a cut-vertex of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.



Notation: Let T be a DFS-tree, $(v, u) \in T$ means v is the parent of u. B is the set of all the back edges.

low function

We define $low(v)$ as the smallest value of $dfn(x)$, where x is a vertex in DFS-Tree T that can be reached from v by following zero or more tree edges followed by at most one back edge.



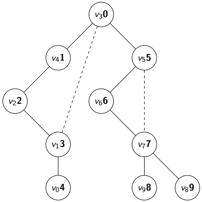
Vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

$$low(v) = min\Big(\{dfn(v)\} \cup \{low(x) | (v,x) \in T\} \cup \{dfn(x) | (v,x) \in B\}\Big).$$

Theorem for the cut-vertices

Theorem

Let $G = (V, E)$ be connected graph with a DFS-Tree T and with back edges B. Then $a \in V$ is a cut-vertex if and only if there exist vertices $v, w \in V$ such that v is a child of a in T, w is not a descendant of v in T and $low(v) \geq dfn(a)$.



Vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

Example: (v_3, v_4), $low(v_4) = dfn(v_3)$ so v_3 is a cut-vertex.
(v_6, v_7), $low(v_7) < dfn(v_6)$ so v_6 is not a cut-vertex.

Algorithmic steps for cut-vertices

1. During DFS, calculate the values of the *low* function for every vertex.
2. After we finish the recursive search from a child u of a vertex v, we update $low(v)$. Vertex v is a cut-vertex, disconnecting u, if $low(u) \geq dfn(v)$.
3. When encountering a back-edge (v, u) update $low(v)$ with the value of $dfn(u)$
4. If vertex v is the root of the DFS tree, check whether w is its second child

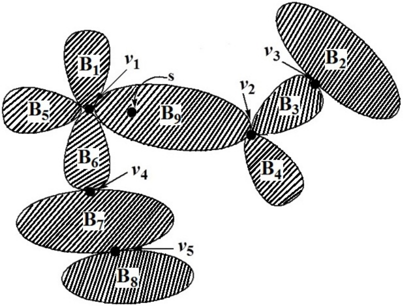
Blocks (or biconnected components or 2-connected components)

A *block* in a graph G is a maximally connected subgraph that has no cut-vertex.

How to find them ?

Finding biconnected components

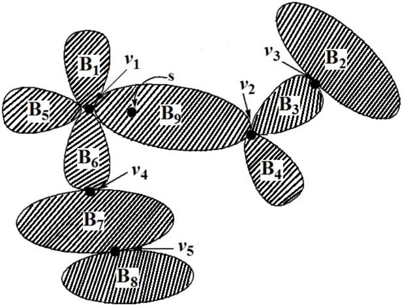
Recognizing cut-vertices, using DFS, we can determine the biconnected components by storing the edges on a stack as they are traversed. HOW?



$v_1, \ldots, v_5$ are the cut-vertices. $B_1, \ldots, B_9$ are the block.

Central Idea

Let us start DFS at vertex s in B_9. Next, suppose we wish to go into B_4 by passing through v_2. By DFS nature all the edges in B_4 must be traversed before we back up to v_2. Now if we leave B_4 and go into B_3 and then into B_2 through v_3. If we store the edges in a stack, by the time we pass through v_3 back into B_3, all the edges of B_2 will be on top of the stack, and forms a biconnected component. When they are removed, the edges on the top of the stack will be from B_3, and we will once again be traversing B_3.



Algorithmic steps for biconnected components

1. During DFS, use a stack to store visited edges (tree edges or back edges)
2. After we finish the recursive search from a child u of a vertex v, we check if v is a cut-vertex for u. If it is, we output all edges from the stack until (v, u). These edges form a biconnected component
3. When we return to the root of the DFS-tree, we have to output the edges even if the root is not a cut-vertex (graph may be biconnected).

Complexity: Since the algorithm is a depth-first search with a constant amount of extra work done as each edge is traversed, the time required is $O(|V| + |E|)$.

References

1. Hopcroft, J. and Tarjan, R., 1973. Algorithm 447: efficient algorithms for graph manipulation. Communications of the ACM, 16(6), pp.372-378.
2. Reingold, E.M., Nievergelt, J. and Deo, N., 1977. Combinatorial algorithms: theory and practice. Prentice Hall College Div.