Searching in Graphs (cut points)

Arash Rafiey

10 November, 2015

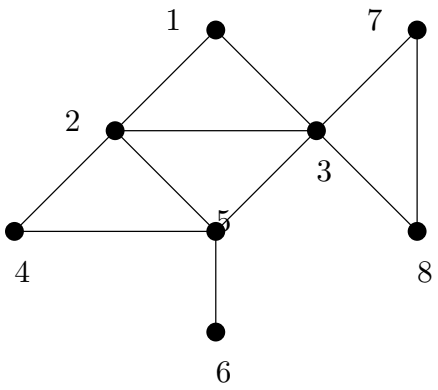
Breath First Search (BFS) in Graphs

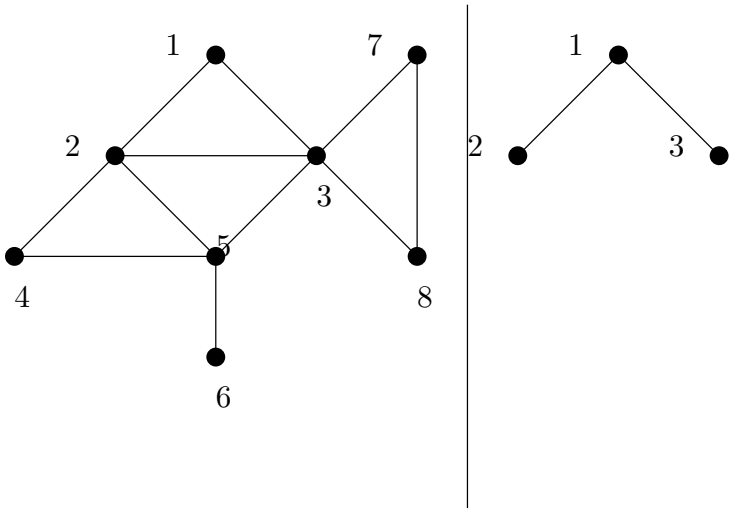
In BFS algorithm we visit the verices level by level. The BFS algorithm creates a tree with root s.

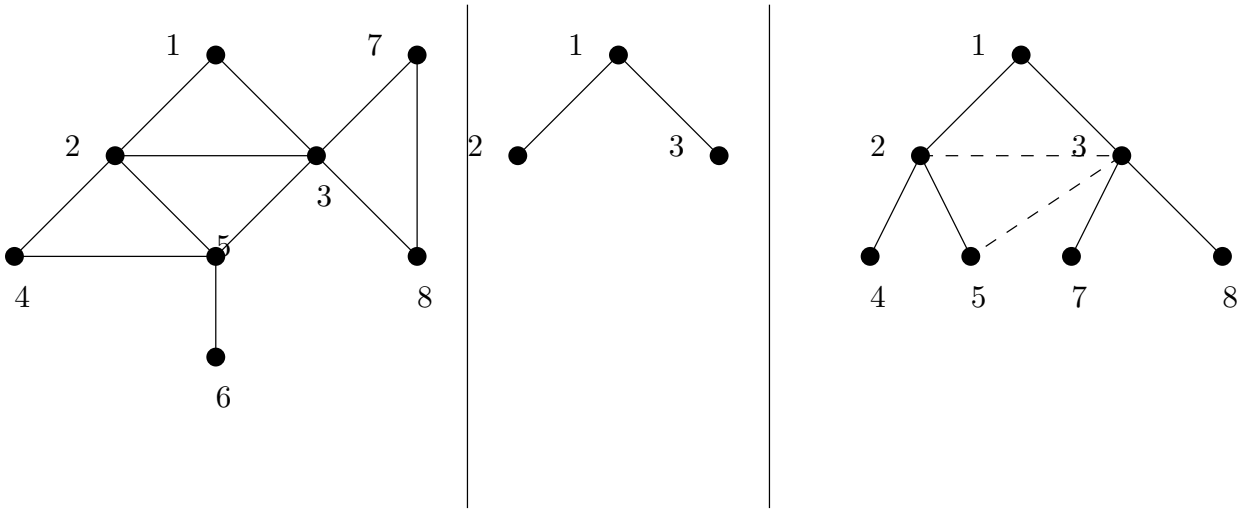
Once a node v is discovered by BFS algorithm we put an edge from v to all the nodes u that have not been considered. This way v is set to be the father of node u.

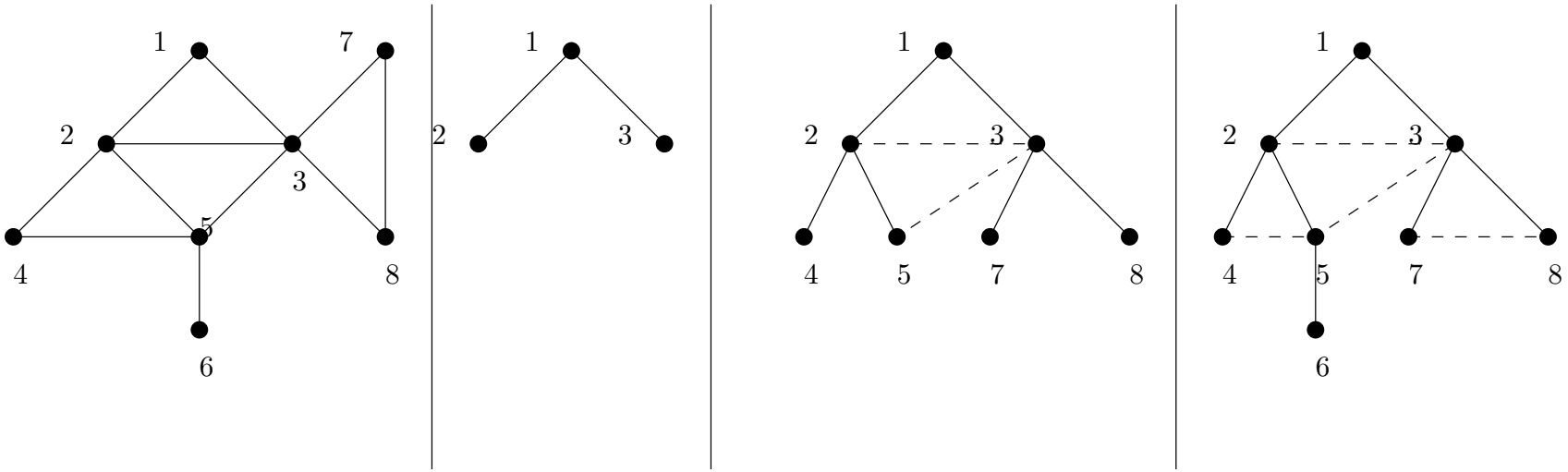
BFS (s)

1. Set Discover[s]=true and Discover[v]=false for all other v
2. Set $L[0] = \{s\}$
3. Set layer counter i=0
4. Set $T = \emptyset$
4. While $L[i]$ is not empty
5. Initialize empty set $L[i+1]$
6. For each node $u \in L[i]$
7. Consider each edge uv
8. If $Discover[v] =$ false then
9. Set $Discover[v] =$ true
10. Add edge uv to T
11. Add v to the list $L[i+1]$, and increase i by one


1
7
2
3
5
4
8
6


1
7
2
3
5
4
8
6
2
1
3


1
7
2
3
5
4
8
6
1
2
3
1
2
3
4
5
7
8


1
7
2
3
5
4
8
6
1
2
3
1
2
3
4
5
7
8
1
2
3
4
5
7
8
6

BFS running time

1) If we represent the graph G by adjacency matrix then the running time of BFS algorithm is $O(n^2)$, where n is the number of nodes.

2) If we represent the graph G by link lists then the running time of BFS algorithm is $O(m+n)$, where m is the number of edges and n is the number of nodes.

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Solution (Using BFS)
Start with node s and color it with red. Next color the neighbors of s by blue. Next color the neighbors of neighbors of s by red and so on.
If at the end there is an edge whose end points receive the same color G is not bipartite.

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Solution (Using BFS)
Start with node s and color it with red. Next color the neighbors of s by blue. Next color the neighbors of neighbors of s by red and so on.
If at the end there is an edge whose end points receive the same color G is not bipartite.

This is essentially is the BFS algorithm. We color the nodes in L_0 by red and the nodes in L_1 by blue and the nodes in L_3 by red and so on.

Next we read each edge uv of G. If both u, v have the same color then G is not bipartite. Otherwise G is bipartite.

Lemma

Let G be a connected graph, and let $L_0, L_1, L_2, \ldots, L_k$ be the layers produced by BFS algorithm starting at node s.

(i) *There is no edge of G joining two nodes of the same layer. In this case G is bipartite and $L_0, L_2, \ldots, L_{2i}$ can be colored red and the nodes in odd layers can be colored blue.*

(ii) *There is an edge of G joining two nodes of the same layer. In this case G contains an odd cycle and G is not bipartite.*

Proof :

Suppose (i) happens. In this case the red nodes and blue nodes give a bipartition, and all the edges of G are between the red and blue nodes.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

2) Let i be the maximum index such that there is $z \in L(i)$ and z is in the intersection of P and Q, i.e. $z \in P \cap Q$ and $z \in L(i)$.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

2) Let i be the maximum index such that there is $z \in L(i)$ and z is in the intersection of P and Q, i.e. $z \in P \cap Q$ and $z \in L(i)$.

3) Portion of P, say P' from z to x has length $j - i$ and portion of Q, say Q' from z to y has length $j - i$.

4) By adding xy edges into P', Q' we get a cycle of length $(j - i) + 1 + (j - i)$ which is of odd length.

Depth-First Search (backtracking approach)

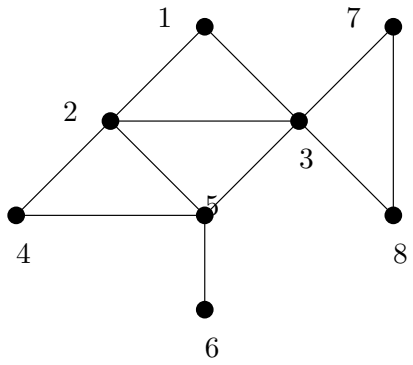
We don't visit the nodes level by level! As long as there is an unvisited node adjacent to the current visited node we continue! Once we are stuck, trace back and go to a different branch!

Depth-First Search (backtracking approach)

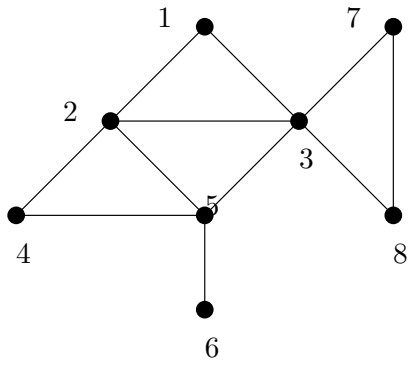
We don't visit the nodes level by level! As long as there is an unvisited node adjacent to the current visited node we continue! Once we are stuck, trace back and go to a different branch!

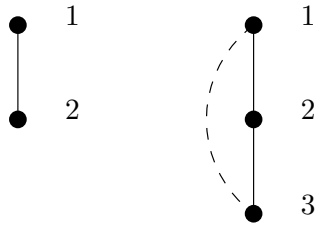
DFS (u)

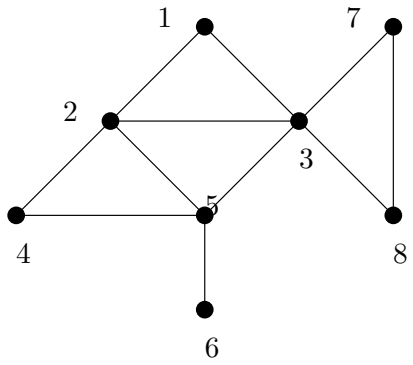
1. Mark u as Explored and add u to R
2. For every edge uv
3. If v is not Explored then call DFS (v)

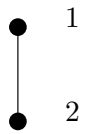

1
7
2
3
5
4
8
6

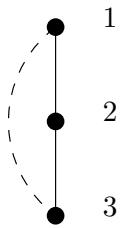

1
2

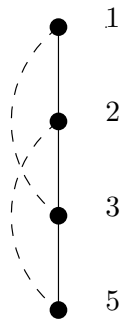

1
7
2
3
5
4
8
6

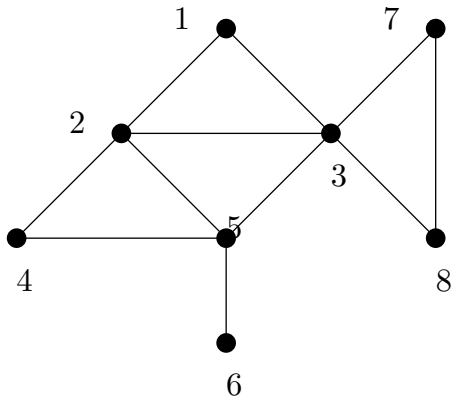

1
2
1
2
3

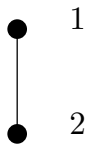

1
7
2
3
5
4
8
6

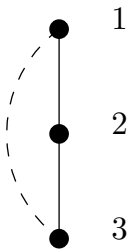

1
2

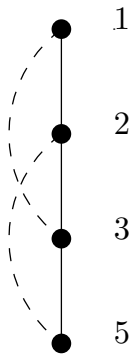

1
2
3

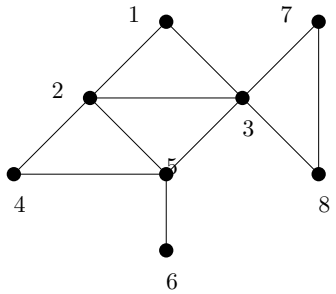

1
2
3
5

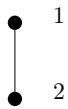

1
7
2
3
5
4
8
6

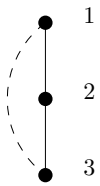

1
2

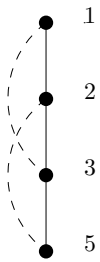

1
2
3

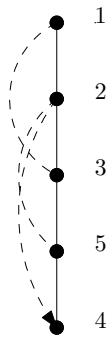

1
2
3
5

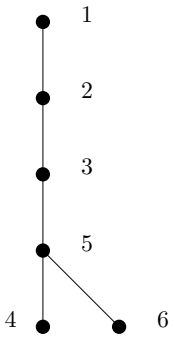










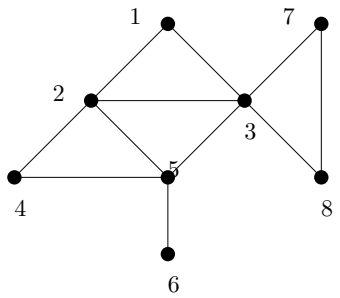


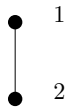
1

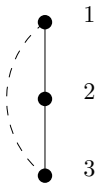
2

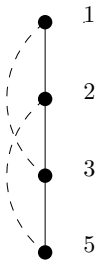
3

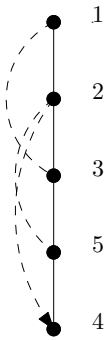
5

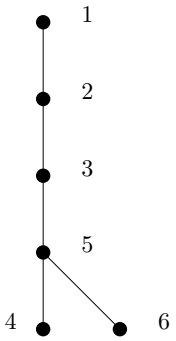

1
7
2
3
5
4
8
6

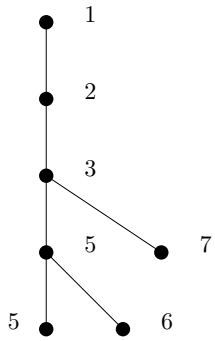

1
2

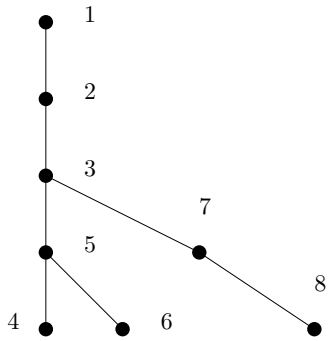

1
2
3


1
2
3
5


1
2
3
5
4


1
2
3
5
4
6


1
2
3
5
7
5
6


1
2
3
7
5
8
4
6

DFS running time

1) If we represent the graph G by adjacency matrix then the running time of BFS algorithm is $O(n^2)$, where n is the number of nodes.

2) If we represent the graph G by link lists then the running time of BFS algorithm is $O(m+n)$, where m is the number of edges and n is the number of nodes.

Biconnected Components

Let $G = (V, E)$ be a loop-free connected undirected graph. A vertex v in G is called an *articulation point* if $\kappa(G - v) > \kappa(G)$. $G - v$ has more connected components than v.

A loop-free connected undirected graph with no articulation points is called *biconnected* .

Lemma

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a DFS spanning tree for G. If $ab \in E$ but $ab \notin E'$, then a is either an ancestor or a descendant of b in the tree.

Lemma

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a DFS spanning tree for G. If $ab \in E$ but $ab \notin E'$, then a is either an ancestor or a descendant of b in the tree.

Lemma

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a DFS spanning tree for G. If r is a root of T, then r is an articulation point of G if and only if r has at least two children in T.

Lemma

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a DFS spanning tree for G. If $ab \in E$ but $ab \notin E'$, then a is either an ancestor or a descendant of b in the tree.

Lemma

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a DFS spanning tree for G. If r is a root of T, then r is an articulation point of G if and only if r has at least two children in T.

Lemma

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a DFS spanning tree for G. Let r be a root of T, and let $v \in V$, $v \neq r$. Then v is an articulation point of G if and only if there exists a child c of v with no back edge from a vertex in T_c (the subtree rooted at c) to an ancestor of v.

Finding Articulation Points

1) We traverse the graph in DFS (preorder) manner.

2) For vertex x of G define $dfi(x)$ to be the index of x in DFS (time we visit x). If y is a descendant of x then $dfi(x) < dfi(y)$.

Finding Articulation Points

1) We traverse the graph in DFS (preorder) manner.

2) For vertex x of G define $dfi(x)$ to be the index of x in DFS (time we visit x). If y is a descendant of x then $dfi(x) < dfi(y)$.

3) Define $low(x) =$
$\min\{dfi(y) |\ y \textit{ is adjacent in } G \textit{ to either } x \textit{ or a descendant of } x\}$

Let z be the parent of x in T. Then :

1) $low(x) = dfi(z)$: In this case T_x contains no vertex that is adjacent to an ancestor of z (by back edge).
Hence z is an articulation point of G.
If T_x contains no articulation points, then T_x together with edge zx is a biconnected component of G.
Remove T_x and the edge xz and apply on the remaining subtree of T.

Let z be the parent of x in T. Then :

1) $low(x) = dfi(z)$: In this case T_x contains no vertex that is adjacent to an ancestor of z (by back edge).
Hence z is an articulation point of G.
If T_x contains no articulation points, then T_x together with edge zx is a biconnected component of G.
Remove T_x and the edge xz and apply on the remaining subtree of T.

2) If $low(x) < dfi(z)$: there is a descendant of z in T_x that is joined (by a back edge in G) to an ancestor of z.

Algorithm for Articulation Points

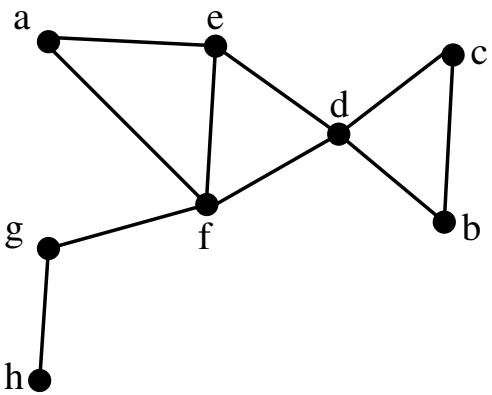
1) Find a DFS ordering $x_1, x_2, \ldots, x_n$ of the vertices of G.

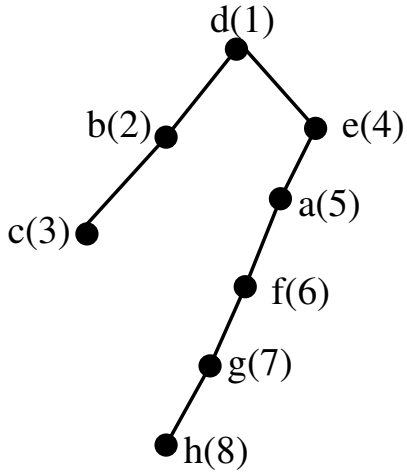
2) Start from x_n and continue to $x_{n-1}, x_{n-2}, \ldots, x_1$ and determine $low(x_j)$ as follows :

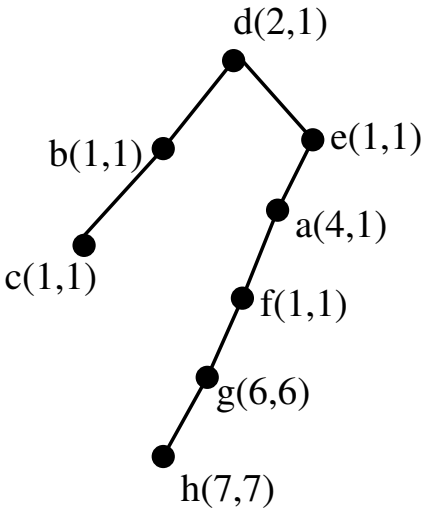
- a) $low'(x_j) = \min\{dfi(z) |\ z \ \ is \ \ adjacent \ \ in \ \ G \ \ to \ \ x_j\}$
- b) If $c_1, c_2, \ldots, c_m$ are children of x_j, then

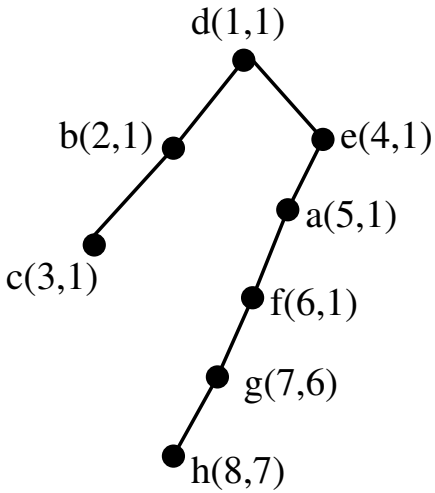
$$low(x_j) = \min\{low'(x_j), low'(c_1), low'(c_2), \ldots, low'(c_m)\}$$

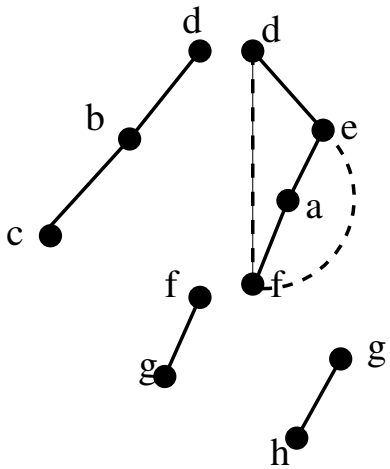
3) Let w_j be the parent of x_j in T. If $low(x_j) = dfi(w_j)$ then w_j is an articulation point of G, unless w_j is the root of T and w_j has no child in T other than x_j.
Moreover, in either situation the subtree rooted at x_j together with the edge $w_j x_j$ is part of a biconnected component of G.


a
e
c
d
g
f
b
h


d(1)
b(2)
e(4)
c(3)
a(5)
f(6)
g(7)
h(8)


d(2,1)
b(1,1)
e(1,1)
a(4,1)
c(1,1)
f(1,1)
g(6,6)
h(7,7)


d(1,1)
b(2,1)
e(4,1)
a(5,1)
c(3,1)
f(6,1)
g(7,6)
h(8,7)


d
d
b
e
a
c
f
f
g
g
h