Set 8: Decomposition of Graphs (Chpt 3 of DPV)

February 24, 2018

Problem 1: [3.7]
A bipartite graph is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V_1$, then there is no edge between u and v).
(a) Give a linear-time algorithm to determine whether an undirected graph is bipartite.
(b) There are many other ways to formulate this property. For instance, an undirected graph is bipartite if and only if it can be colored with just two colors. Prove the following formulation: an undirected graph is bipartite if and only if it contains no cycles of odd length.
(c) At most how many colors are needed to color in an undirected graph with exactly one odd length cycle?
Answer:
(a) For simplicity, we will assume that the graph is connected. If not, we can check for all the connected components using the same algorithm.
We can check whether a graph is bipartite or note using the previsit array $pre[\cdot]$ which is obtained from DFS. While exploration of v in V, if we detect an edge (v, u) which leads to an already visited vertex, we compare the parity of previsit values. That is, we compare $pre[v]$ and $pre[u]$. If they are of same parity, it means there exists an odd length cycle in the graph. In fact, we can get a path from u to v along the DFS tree, and by appending the edge (v, u) we obtain a cycle. And we can easily see that the parity of $pre[\cdot]$ changes for every edge in the path of tree. So if the parity of $pre[v]$ and $pre[u]$ are the same, the length of that cycle is odd.
(b)
- If G is bipartite and the vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set. Now suppose that G contains an odd cycle $C = v_0e_1v_1e_2...e_{2k+1}v_0$. Without loss of generality, let v_0 be a vertex in V_1. Then v_1 must be a vertex in V_2. Similarly, e_{2n+1} is preceded by a vertex in V_1 and proceeded by a vertex in V_2 for all $n \in N$. But e_{2k+1} is proceeded by v_0, which is a vertex in V_1. This contradicts to the assumption : $V_1 \cap V_2 = \emptyset$.
- Now, let us assume that there are not any odd-length cycles in the graph G. Then as in (a), we can partition the vertices according to the parity of $pre[\cdot]$. Because there are not odd-length cycles, vertices with odd $pre[]$ values can only be connected to the vertices with even $pre[]$. So if we set $V_1 = \{v \in V | pre[v]$ is odd$\}, V_2 = \{v \in V | pre[v]$ is even$\}$, then $V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset$. And there are no edges between vertices in the same set which shows that G is bipartite.
(c) Let the odd-length cycle be $C = v_0e_1v_1e_2...v_{2k}e_{2k+1}v_0$. If we remove the edge e_{2k+1}, then the remaining graph does not have any odd-length cycle. So every vertex can be colored with two colors, say Red and Blue. Now we repair the edge e_{2k+1}, then v_0 and v_{2k} became connected and they are of same colors. Now we change the color of v_{2k} into another color, say Green. Then it will not affect the coloring of other vertices and no adjacent vertices are colored with same color. So three colors are enough.

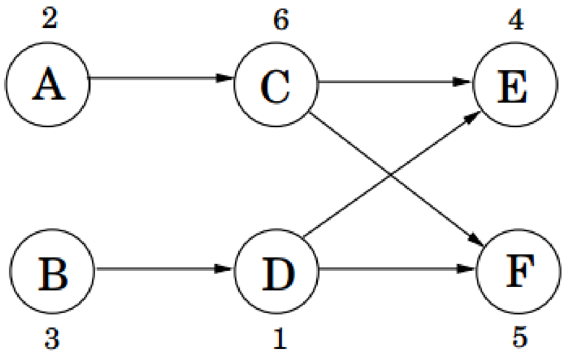
Problem 2: [3.15]
The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a linear-time algorithm. (a) Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time.
(b) Suppose it now turns out that the mayors original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.
Answer:
(a) Let us formulate the problem graph-theoretically. If we regard the intersections in the city as vertices and the streets as edges, we obtain a directed graph. The mayor's claim is equivalent that all vertices are connected in the sense of direct graph. This is equivalent to that the whole graph is strongly-connected. We saw that strongly-connected components can be obtained in linear time. So if no vertex is left after linearization and removal of the first strongly-connected component(which starts from a vertex with maximum $post[]$ value), the mayor's claim is right.
(b) In the graph-theoretical view, this is equivalent to that there are no connections between strongly connected components. So in this case, the graph can contain several strongly-connected components which are independent from others. This weaker claim also can be checked in linear time. We just add one additional step into the second step of strongly-connected component exploration algorithm. In the original algorithm, we run the undirected connected components algorithm on G and log the connected vertices(say V_C). But we additionally run the undirected connected components algorithm for the same vertex on G^R and log the connected vertices(say V_C^R). If the two subset are coincident, i.e. $V_C = V_C^R$, it's ok. But if they don't coincide, the claim is false. This additional work will increase the whole time linear to the size of graph.

Problem 3: [3.25]

You are given a directed graph in which each node $u \in V$ has an associated price p_u which is a positive integer. Define the array *cost* as follows: for each $u \in V$,

$$cost[u] = \text{price of the cheapest node reachable from } u \text{ (including } u \text{ itself).}$$

For instance, in the graph below (with prices shown for each vertex), the *cost* values of the nodes A, B, C, D, E, F are $2, 1, 4, 1, 4, 5$, respectively



Your goal is to design an algorithm that fills in the entire *cost* array (i.e., for all vertices). (a) Give a linear-time algorithm that works for directed acyclic graphs. (Hint: Handle the vertices in a particular order.) (b) Extend this to a linear-time algorithm that works for all directed graphs. (Hint: Recall the two-tiered structure of directed graphs.)

Answer:

(a) First of all, we can easily suppose that this problem would be related to Dynamical Programming method, because the problem can be divided into subproblems. So in the case of the above figure, $cost[A] = min\{cost[C], p_A\}$. Now we need to traverse in a 'good' order so that the vertices are traversed from 'sink' to 'source'. From the fact that the graph is a DAG, we can sort the vertices in the topological order. The final algorithm would like as follows.

```
begin calculate cost at $G : DAG$
    $\{v_1, \cdots, v_n\}$=topological_sort($G$);
    $cost[v] = p_v$ for all $v \in G$;
    for $k = n$ to 1 do
        foreach $v_j$ where $(v_k, v_j) \in E$ do
            $cost[v_k] = min\{cost[v_j], cost[v_k]\}$;
        end
    end
end
```

It is well-known that topological sort for DAG has linear time complexity and for the double for clause, it traverses every vertex and edge only once, so it also has linear time complexity. So the overall time complexity is linear, $O(V + E)$.

(b) In the case of general directed graphs, we first decompose the graph into strongly connected components and then make use of the above algorithm for the meta-graph (see Fig 3-9 in the textbook).

First we note that all vertices in the same strongly connected have the same *cost*, and the *cost* would be the minimum *price* of the vertices in that component. So we first decompose the graph into the strongly connected components and set the *cost* values of all vertices in the same component with the minimum *price* in that component. Now we make a meta-graph where every strongly-connected component becomes a single vertex. Then the resulting meta-graph becomes a DAG and then we can apply the algorithm proposed in (a).

The time complexity is clearly linear in the size of the graph.

```
begin calculate cost at $G : DirectedGraph$
    $\{V_1, \cdots, V_m\}$=strongly_connected_components($G$);
    $cost[v] = min\{p_u | u \in V_k\}$ for all $k = 1 \cdots m$ and $v \in V_k$;
    $G^M = meta\_graph(G)$;
    calculate cost at $G^M$;
end
```

Problem 4: [3.29]
Let S be a finite set. A binary relation on S is simply a collection R of ordered pairs $(x, y) \in S \times S$. For instance, S might be a set of people, and each such pair $(x, y) \in R$ might mean x knows y. An equivalence relation is a binary relation which satisfies three properties:
- Reflexivity: $(x, x) \in R$ for all $x \in S$.
- Symmetry: if $(x, y) \in R$ then $(y, x) \in R$.
- Transitivity: if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$.
For instance, the binary relation has the same birthday as is an equivalence relation, whereas is the father of is not, since it violates all three properties. Show that an equivalence relation partitions set S into disjoint groups $S_1, S_2, \cdots, S_k$ (in other words, $S = S_1 \cup S_2 \cup \cdots \cup S_k$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$) such that:
- Any two members of a group are related, that is, $(x, y) \in R$ for any $(x, y) \in S_i$, for any i.
- Members of different groups are not related, that is, for all $i \neq j$, for all $x \in S_i$ and $y \in S_j$, we have $(x, y) \neq R$.
Answer:
Let us make a graph model for this problem. We make a set of vertices, say V, corresponding to the elements of S. Then make the edges corresponding to the binary relation.
So if $(x, y) \in R$ then we make an edge $(u, v) \in E$ where u, v are vertices corresponding to x, y respectively. From the symmetry property, this graph (V, E) can be regarded to be undirected. And from the reflexivity, we can think every vertex is connected to itself. Now we run DFS on this undirected graph. We can start from any vertex and the vertices of the resulting DFS tree are grouped into a set. Repeating this process until we have no vertices left, then the whole graph can be divided into several components $C_1, \cdots, C_m$. And there are no connections between different connected components clearly.
Furthermore, from the transitivity all vertices in the same connected components are connected directly. So $C_1, \cdots, C_m$ become connected components where all vertices are connected directly. Now we can get the partition of original set S correspondingly.

Problem 5: [3.30]

On page 102, we defined the binary relation connected on the set of vertices of a directed graph. Show that this is an equivalence relation (see Exercise 3.29), and conclude that it partitions the vertices into disjoint strongly connected components.

Answer:

We define the binary relation as $R = \{(u, v)|u \text{ is connected to } v\}$. Here the term 'connected' means that there is a path from u to v and also from v to u. - Reflexivity : We can think every vertex is connected to itself. It's clear. - Symmetry : It's also clear from the definition. - Transitivity: If u is connected to v and v is connected to w, we can make a path from v to w by appending the paths. It's also possible to make a path from w to u. So it's transitive.

Now consider the partition by equivalence relation. Let $C_1 = \{u|(u, v_0) \in R\}$ be a equivalence class. Then it coincides with the strongly connected component which includes v_0.

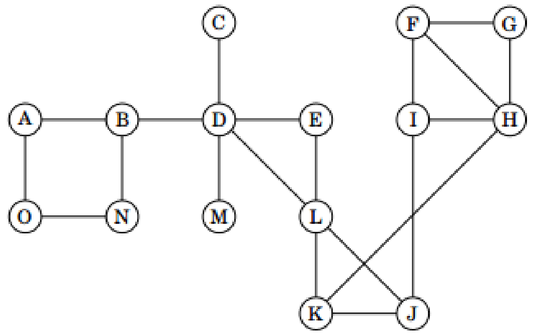
Problem 5: [3.31]

Biconnected components Let $G = (V, E)$ be an undirected graph. For any two edges $e, e' \in E$, we'll say $e \approx e'$ if either $e = e'$ or there is a (simple) cycle containing both e and e'.

(a) Show that $\approx$ is an equivalence relation (recall Exercise 3.29) on the edges.

The equivalence classes into which this relation partitions the edges are called the *biconnected components* of G. A *bridge* is an edge which is in a biconnected component all by itself. A separating vertex is a vertex whose removal disconnects the graph.

(b) Partition the edges of the graph below into biconnected components, and identify the bridges and separating vertices.



Not only do biconnected components partition the edges of the graph, they also almost partition the vertices in the following sense.

(c) Associate with each biconnected component all the vertices that are endpoints of its edges. Show that the vertices corresponding to two different biconnected components are either disjoint or intersect in a single separating vertex.
(d) Collapse each biconnected component into a single meta-node, and retain individual nodes for each separating vertex. (So there are edges between each component-node and its separating vertices.) Show that the resulting graph is a tree.

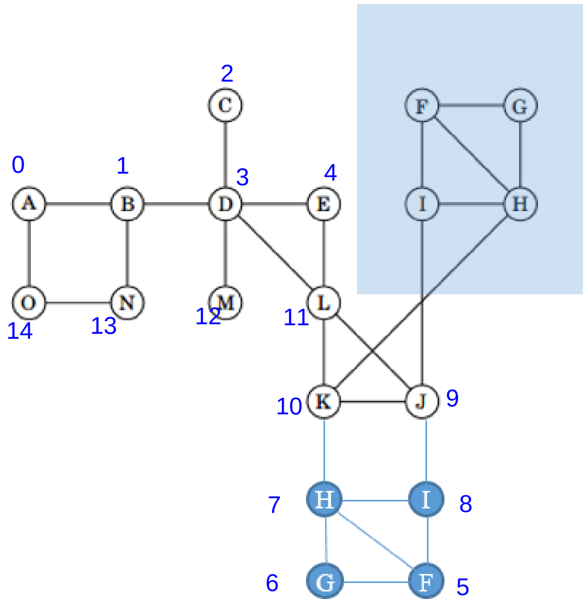
DFS can be used to identify the biconnected components, bridges, and separating vertices of a graph in linear time.

(e) Show that the root of the DFS tree is a separating vertex if and only if it has more than one child in the tree.
(f) Show that a non-root vertex v of the DFS tree is a separating vertex if and only if it has a child v_0 none of whose descendants (including itself) has a backedge to a proper ancestor of v.
(g) For each vertex u define:

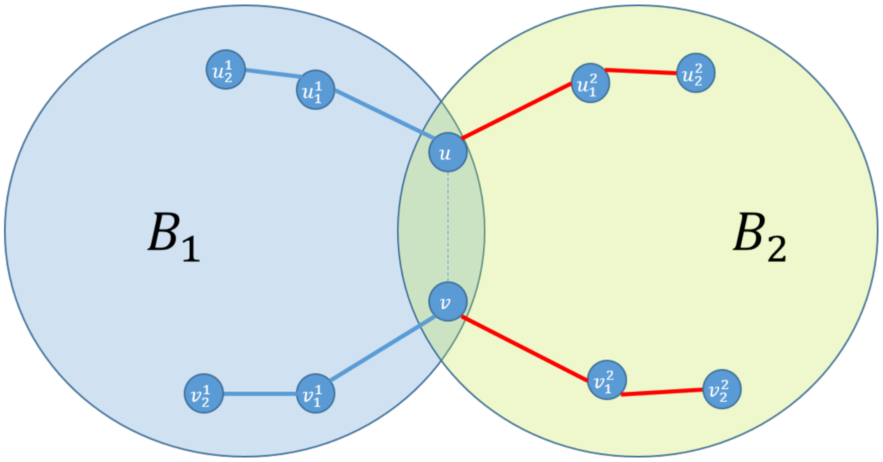
$$low(u) = min\{pre(u), pre(w)\}$$

where (v, w) is a backedge for some descendant v of u. Show that the entire array of low values can be computed in linear time.
(h) Show how to compute all separating vertices, bridges, and biconnected components of a graph in linear time. (Hint: Use low to identify separating vertices, and run another DFS with an extra stack of edges to remove biconnected components one at a time.)
Answer:
(a)
- Reflexivity : From the definition of the binary relation $\approx$, it's clear.
- Symmetry : If there is a cycle containing both e and e', itself is a cycle containing both e' and e.
- Transitivity: Let us assume $e_0 \approx e_1, e_1 \approx e_2$. Let $C_1 = e_0 e_{01} e_{02} \cdots e_{0k} e_1$ be a simple cycle containing e_0 and

e_1, and $C_2 = e_1 e_{11} e_{12} \cdots e_{1j} e_2$ be a cycle containing e_1, e_2. Now the 'sum' of these two cycles $C = C_1 \cup C_2$ is also a cycle and it contains both u and w. So this relation is transitive.
(b) We can rearrange the vertices topologically for simplicity as follows.

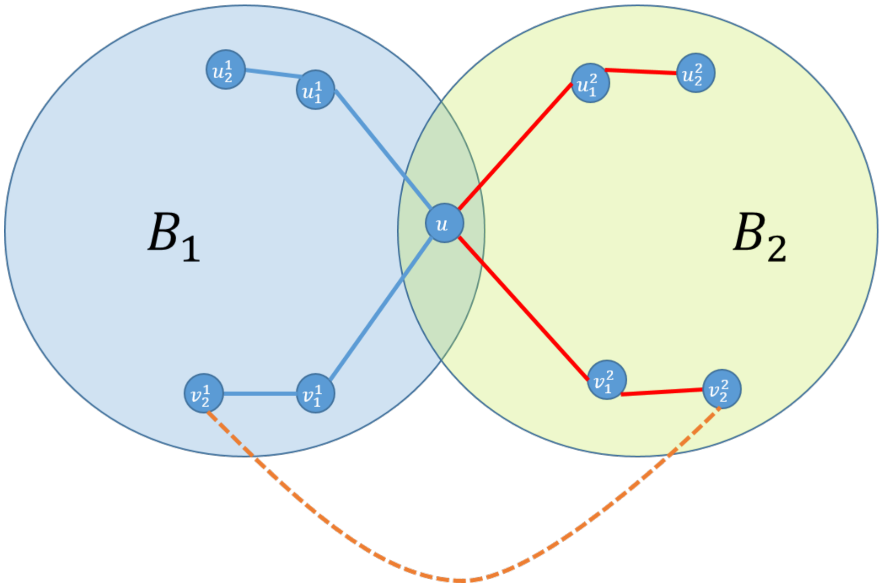


Now it's easier to find biconnected components.
$B_1 = \{AB, BN, NO, OA\}, B_2 = \{BD\}, B_3 = \{CD\}, B_4 = \{DM\}$,
$B_5 = \{DE, EL, LD\}, B_6 = \{LK, KJ, JL, KH, JI, IH, IF, FH, HG, GF\}$
And the bridges are
$B_2 = \{BD\}, B_3 = \{CD\}, B_4 = \{DM\}$.
The separating vertices are $\{B, D, L\}$.
(c)
- Let the two different biconnected components be B_1, B_2 and their corresponding vertex sets be V_1, V_2. First let us assume V_1 and V_2 intersect in more than two separating vertices, say u, v. Then there exist $u_1, v_1 \in V_1, u_2, v_2 \in V_2$ such that $(u, u_1) \in B_1, (u, u_2) \in B_2, (v, v_1) \in B_1, (v, v_2) \in B_2$(See the figure).

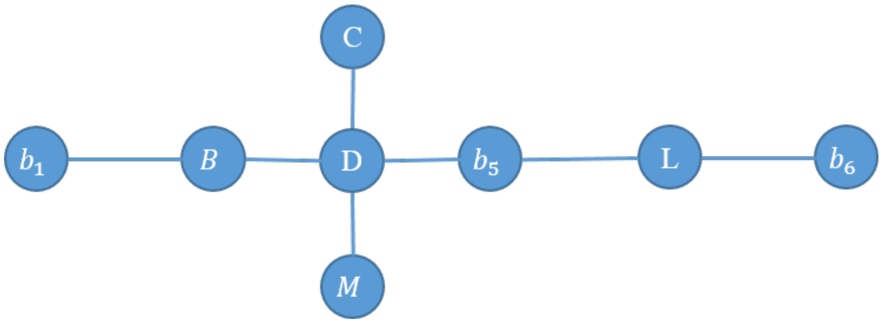


Because B_1 is a biconnected component, there exists a cycle which contains both (u, u_1) and (v, v_1). Let the cycle be like $uu_1^1u_2^1 \cdots v_2^1v_1^1v \cdots u$. Likewise there exists a cycle $uu_1^2u_2^2 \cdots v_2^2v_1^2v \cdots u$ in B_2.
Now we can easily make a cycle like like $uu_1^1u_2^1 \cdots v_2^1v_1^1vv_1^2v_2^2 \cdots u_2^2u_1^2u$. This contradicts to the fact that B_1, B_2 are different biconnected components.

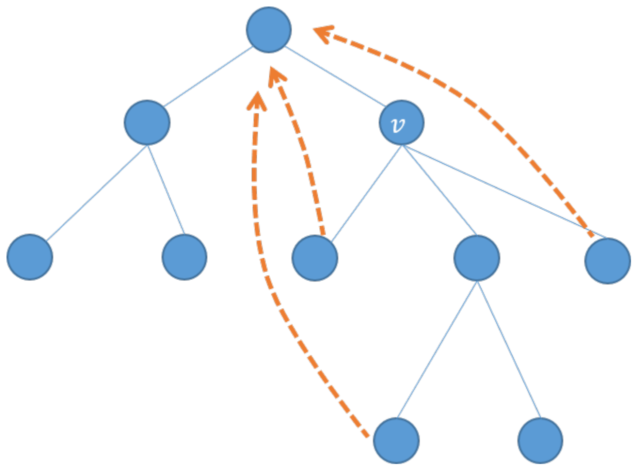
- Now let us assume V_1 and V_2 intersect in a vertex u. Then it should be a separating vertex. If not, after removing u there should still exist a connection between V_1 and V_2. It means that we can make a cycle in $V_1 \cup V_2$ using that connection and u. (see the figure below) This is a same contradiction again.



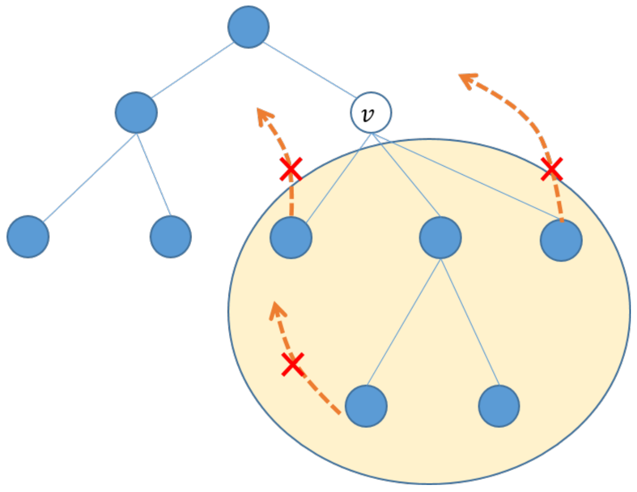
So if the two different biconnected components intersect, the intersection is a separating vertex.
(d) The resulting graph after collapsing biconnected components into a 'meta-node' is a tree.
In the figure, b_1 stands for 3 vertices A, O, N which belong to the component B_1 and note that B is not merged because it's a separate vertex. Likewise $b5$ stands for a vertex E in the original graph (component B_5) and we didn't merge D, L in this component because they are separate vertices.



- If the original graph is connected, the connection is reserved clearly.
- The resulting graph does not contain a cycle in it. If it had one, we can easily deduce that all the original vertices int the nodes in that cycle make a 'bigger' biconnected component in the original graph.
(e)
- If a root r is a separate vertex, the graph will become not connected after removing it. But if it has only a child, removing it does not affect to the connectivity of the graph. (Actually a root with only one child can be seen as a 'leaf' in some sense.) So it should have more than 2 children in the tree.
- Let us assume that a root r has two children u, v in the DFS tree. If we remove the root r, there's no way to connect these two vertices. So it's a separate vertex.
(f)
- Let us assume a non-root vertex v of the DFS tree is a separating vertex. If all of its children have descendants that have a backedge to an ancestor of v, removing v does not affect the connectivity of the graph. (see the figure below)



- Inversely if there exists a child u of v such that none of the descendants of u has a backedge, after removing v, we have no way to connect u and its descendants to the parent of v. So v is a separate vertex.



(g) We give the pseudocode to get the entire array of low values.
$low[u]$ means means the lowest visit time that can be found during DFS starting from the vertex u so if we find some w which has already been visited, we have to take into account its visit time so that $low(u)$ is updated. The following pseudocode shows this clearly.
The time complexity is the same as DFS, so linear.

```
Function explore (u);
clock = clock+1
pre[u] = clock
low[u] = clock
visited[u] = true
foreach w in Adj(u) do
    if visited[w]==false then
        explore(w)
        low[u]=min(low[u], low[w])
    else
        low[u]=min(low[u], pre[w])
    end
end
clock = clock+1
post[u] = clock
```

(h)

- Separating vertex

We discussed about the separating vertex in (e), and using the additional *low* values we can decide if a vertex is a separating one or not during DFS. For a root of a DFS, if it has 2 or more children, it's a separating vertex. For a non-root vertex u of a DFS, we check after every exploration of its child v if $low[u] >= pre[u]$, and if we find such a child, the vertex u is a separating vertex.

- Bridge

```
Function explore (u);
// detection of separate vertex
clock = clock+1
pre[u] = clock
low[u] = clock
child[u] = 0
visited[u] = true
foreach w in Adj(u) do
    child[u] = child[u] + 1
    if visited[w]==false then
        explore(w)
        low[u]=min(low[u], low[w])
        if low[v]≥pre[u] then
            OUTPUT("{0} is a separate vertex", u)
        end
    end
    else if v != parent[u] AND pre[w] < pre[u] then
        low[u]=min(low[u], pre[w])
    end
end
if parent[u]==NULL then //u is a root
    if child[u]>1 then
        OUTPUT("{0} is a separate vertex", u)
    end
end
clock = clock+1
post[u] = clock
```

If a vertex u has a back edge pointing to it, then no edge below u in the DFS tree can be a bridge. The reason is that each back edge gives us a cycle, and no edge that is a member of a cycle can be a bridge. So during the DFS, if u is a parent of v, and no ancestor of v has a back edge pointing to it, then (u, v) is a bridge. This can be checked comparing $low[v]$ and $pre[u]$. If $low[v] > pre[u]$, (u, v) is a bridge.

- Biconnected components

As we have seen in (d), the separating points separate the biconnected components of a graph. If the graph has no separating points, this graph is biconnected in a whole. So during DFS, we use a stack to store visited edges. After we finish the exploration for a child v of a vertex u, we check if u is an separating point. If it is, we output all edges from the stack. These edges form a biconnected component. And after finishing the exploration of a whole graph, we output the every remaining edge in the stack as a component.

We give the pseudocode for all of these works below.

It can be easily seen that the time complexity of this algorithm is linear, i.e. $O(V + E)$.

```
Function explore ($u$);
// detection of separate vertex, bridge, biconnected components
clock = clock+1
pre[u] = clock
low[u] = clock
child[u] = 0
visited[u] = true
foreach v in Adj(u) do
    child[u] = child[u] + 1
    if visited[v]==false then
        parent[v] = u
        stack.push(pair(u,v)) // this stack is for enumerating biconnected components
        explore(v)
        low[u] = min(low[u], low[v])
        if low[v]≥pre[u] then
            OUTPUT("{0} is a separate vertex", u)
            // output all the edges in the subtree starting from v
            OUTPUT("Biconneced component found:")
            while true do
                edge = stack.top();
                if edge.first == u AND edge.second == v then
                    break;
                end
                OUTPUT("{0}, {1}", edge.first, edge.second)
                stack.pop();
            end
        end
        if low[v]>pre[u] then
            OUTPUT("{0}, {1} is a bridge", u, v)
        end
    end
    else if v != parent[u] AND pre[v] < pre[u] then
        low[u]=min(low[u], pre[v])
    end
end
if parent[u]==NULL then //u is a root
    if child[u]>1 then
        OUTPUT("{0} is a separate vertex", u)
    end
    while stack.empty()==False do
        edge = stack.top();
        OUTPUT("{0}, {1}", edge.first, edge.second)
        stack.pop();
    end
end
clock = clock+1
post[u] = clock
```