Lecture 6: Depth-First Search

Background

Graph Traversal Algorithms: Graph traversal algorithms visit the vertices of a graph, according to some strategy.

Example: The BFS is an example of a graph traversal algorithm that traverses each connected component separately. It traverses the vertices of each component in increasing order of the distances of the vertices from the ‘root’ of the component.
Can be thought of processing ‘wide’ and then ‘deep’.

DFS will process the vertices first deep and then wide. After processing a vertex it recursively processes *all* of its descendants.

DFS Algorithm

Graph is $G = (V, E)$. The algorithm works in discrete time steps. Each vertex v is given a "discovery" time $d[v]$ when it is first processed and a "finish" time, $f[v]$ when all of its descendants are finished.

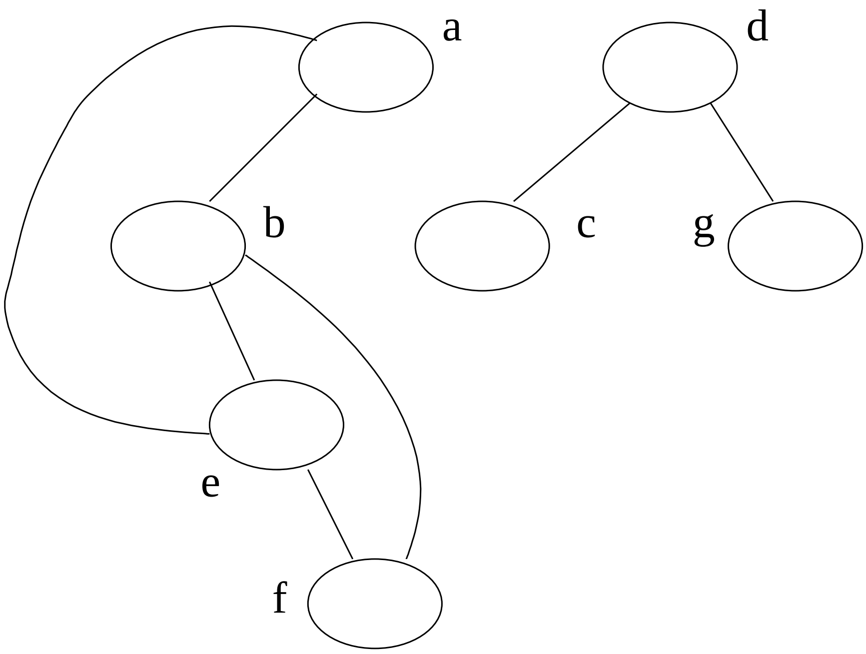
The output is a collection of trees. As well as $d[v]$ and $f[v]$, each node points to $\mathtt{pred}[v]$, its parent in the forest.

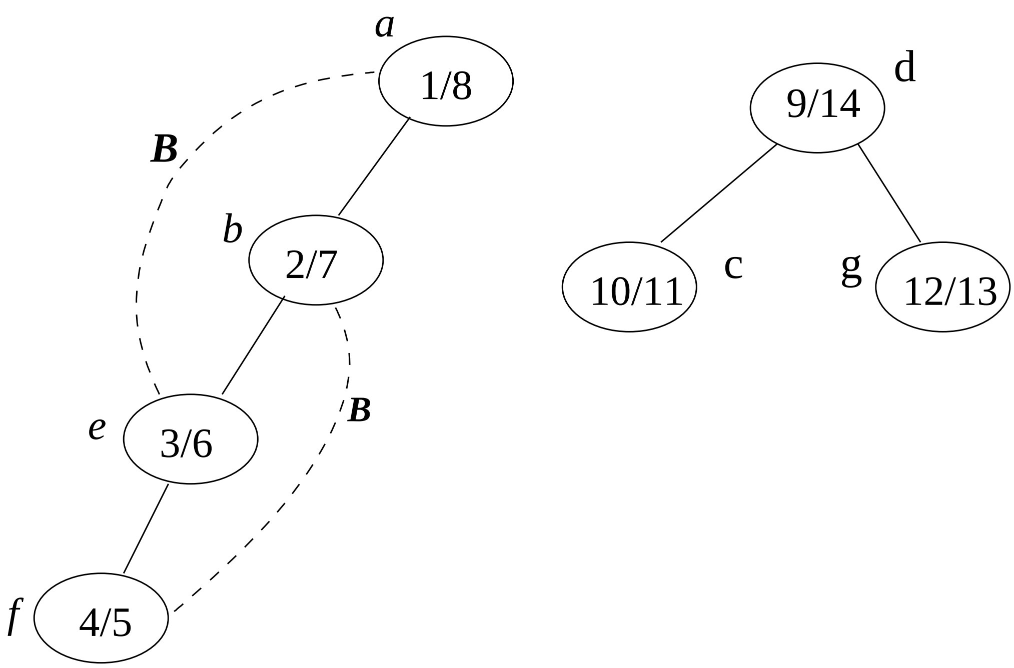
DFS Algorithm

```
DFS{G} {
 for each u in V do // Initialize
  color[u] = white;
  pred[u]  = NULL;
 time=0;
 for each u in V do
  // start a new tree
  if (color[u] == white) DFSVisit(u);
}

DFSVisit(u) {
 color[u] = gray; // u is discovered
 d[u] = ++time;   // u discovery time
 for each v in Adj(u) do
  // Visit undiscovered vertex
  if (color[v] == white) {
   pred[v] = u;
   DFSVisit(v);
  }
 color[u] = black; // u has finished
 f[u] = ++time;    // u finish time
}
```

DFS Example





What Does DFS Do

Given a digraph $G = (V, E)$, it traverses all vertices of G and

- constructs a forest (a collection of rooted trees), together with a set of source vertices (the roots); and

- outputs two arrays, $d[v]/f[v]$, the two time units.

Note: Forest is stored in pred[] array with pred$[v]$ pointing to parent of v in the forest. pred[] of a root node is *NULL.*

DFS Forest: DFS creates a forest $F = (V, E_f)$, a collection of rooted trees, where

$E_f = \{(pred[v], v)|$ where DFS calls are made$\}$

Idea of the DFS Algorithm

- In DFS, edges are explored out of the most recently discovered vertex v. Only edges to unexplored vertices are explored.
- When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.
- The process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then one of them is selected as a new source vertex, and the search is repeated from that source vertex.
- This process is repeated until all vertices are discovered.

The strategy of the DFS is to search "deeper" in the graph whenever possible.

Four Arrays for the DFS Algorithm

To record data gathered during traversal.

- $color[u]$, the color of each vertex visited: white means *undiscovered*, gray means *discovered* but not finished processing, and black means *finished* processing.

- $pred[u]$, the predecessor pointer, pointing back to the vertex that discovered u.

- $d[u]$, the *discovery time*, a counter indicating when vertex u is discovered.

- $f[u]$, the *finishing time*, a counter indicating when the processing of vertex u (and all its descendants) is finished.

Tree structure

DFS imposes a tree (a collection of trees, or *forest*) on the structure of the graph. For undirected graphs, the edges are classified as follows:

Tree edges: which are the edges *(pred[v], v) where DFS calls are made.*

Back edges: *which are the edges (u, v) where v is an ancestor of u in the tree.*

Time-stamp structure

There is also an important and useful structure to the time stamps.

- u is a descendant of v, if and only if $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$.

- u is an ancestor of v, if and only if $[d[u], f[u]]$ contains $[d[v], f[v]]$.

- u is unrelated to v, if and only if $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint intervals.

Time-stamp structure : Proof

The idea is to consider every case. We first consider $d[v] < d[u]$.

1. If $f[v] > d[u]$, then u is discovered when v is still not finished yet (marked gray). This implies u is descendant of v.

   Moreover, since u is discovered later than v, u should finish before v. Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$.

2. If $f[v] < d[u]$, obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint. It means that when u or v is discovered, the others are not marked gray. Hence neither vertex is a descendant of the other.

The argument for other case, where $d[v] > d[u]$, is similar.

Running Time Analysis of DFS

```
DFS{G} {
 for each u in V do // 2n
  color[u] = white;
  pred[u]  = NULL;
 time=0;      // 1
 for each u in V    // n
  if (color[u] == white) DFSVisit(u);
}// Sum: 3n + 1

DFSVisit(u) {     // 1
 color[u] = gray; // 1
 d[u] = ++time;   // 2
 // out-degree of u
 for each v in Adj(u) do
  if (color[v] == white) {
   pred[v] = u;
   DFSVisit(v);   // 2
  }
 color[u] = black;// 1
 f[u] = ++time;   // 2
}// Sum : T_u <= 7 + 2 out-degree(u)
```

Running Time Analysis of DFS – Continued

The total running time is

$$(3n+1) + \sum_{u \in V} T_u.$$

We have

$$\sum_{u \in V} T_u \le \sum_{u \in V} (7 + 2\,\text{outdeg}(u)) = 7n + 2e$$

and

$$\sum_{u \in V} T_u \ge \sum_{u \in V} (7 + \text{outdeg}(u)) = 7n + e.$$

Hence

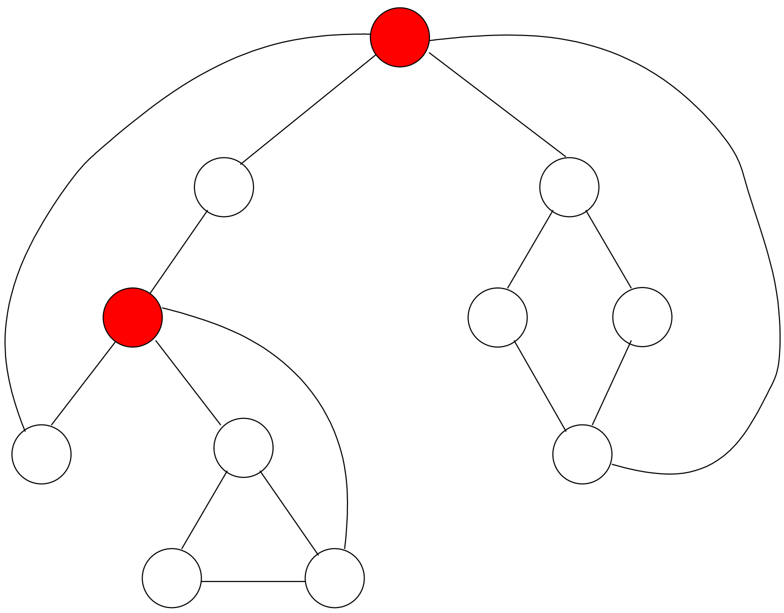
$$T(n,e) \le 10n + 2e + 1 = O(n+e),$$

$$T(n,e) \ge 10n + e + 1 = \Omega(n+e).$$

Therefore $T(n,e) = \Theta(n+e)$.

An Application of DFS : Articulation points

Definition : Let $G = (V, E)$ be a connected undirected graph. An articulation point (or cut vertex) of G is a vertex whose removal disconnects G.



Given a connected graph G, how to find all articulation points?

Articulation points: Easy solution

The easiest solution is to remove a vertex (and its corresponding edges) one by one from G and test whether the resulting graph is still connected or not (say by DFS). The running time is $O(V * (V + E))$.

A more elegant algorithm always starts at simple observations. Suppose we run DFS on G, we get a DFS tree. If the root has two or more children, it is an articulation point. Moreover, a leaf is not an articulation point.

Some parts of the tree have edges that 'climbs' to the upper part of the tree , while other does not have this edge.

Articulation points: Three observations

These lead to the following ideas to identify articulation points:

1. The root of the DFS tree is an articulation if it has two or more children.

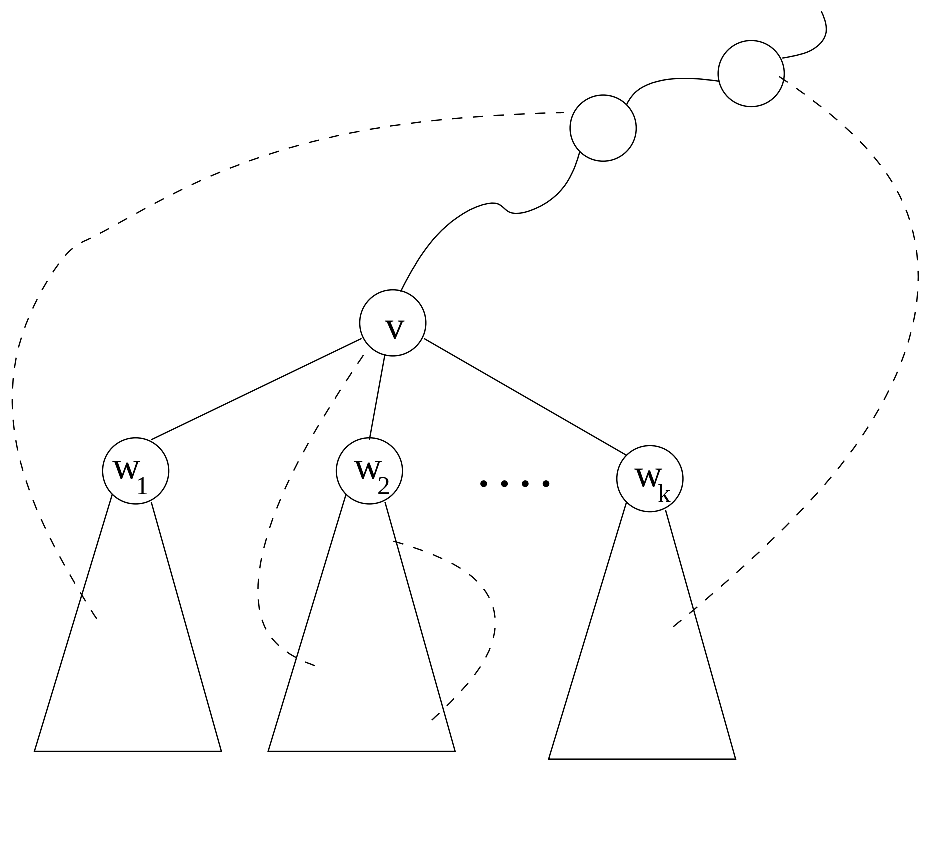
2. A leaf of a DFS tree is not an articulation point.

3. Any other internal vertex v in the DFS tree, if it has one or more subtrees rooted at a child of v that does NOT have an edge which climbs 'higher' than v, then v is an articulation point.

Articulation points: How to climb up

Observation 1 and 2 can be handled easily. How to make use of observation 3?

Observe that for an undirected graph, the DFS tree can only has tree edges or back edges. A subtree can only climb to the upper part of the tree by a back edge, and a vertex can only climb up to its ancestor.



Articulation points: Tackle observation 3

We make use of the *discovery time* in the DFS tree to define 'low' and 'high'. Observe that if we follow a path from an ancestor (high) to a descendant (low), the *discovery time* is in *increasing order*.

If there is a subtree rooted at a children of v which does not have a back edge connecting to a SMALLER discovery time than $d[v]$, then v is an articulation point.

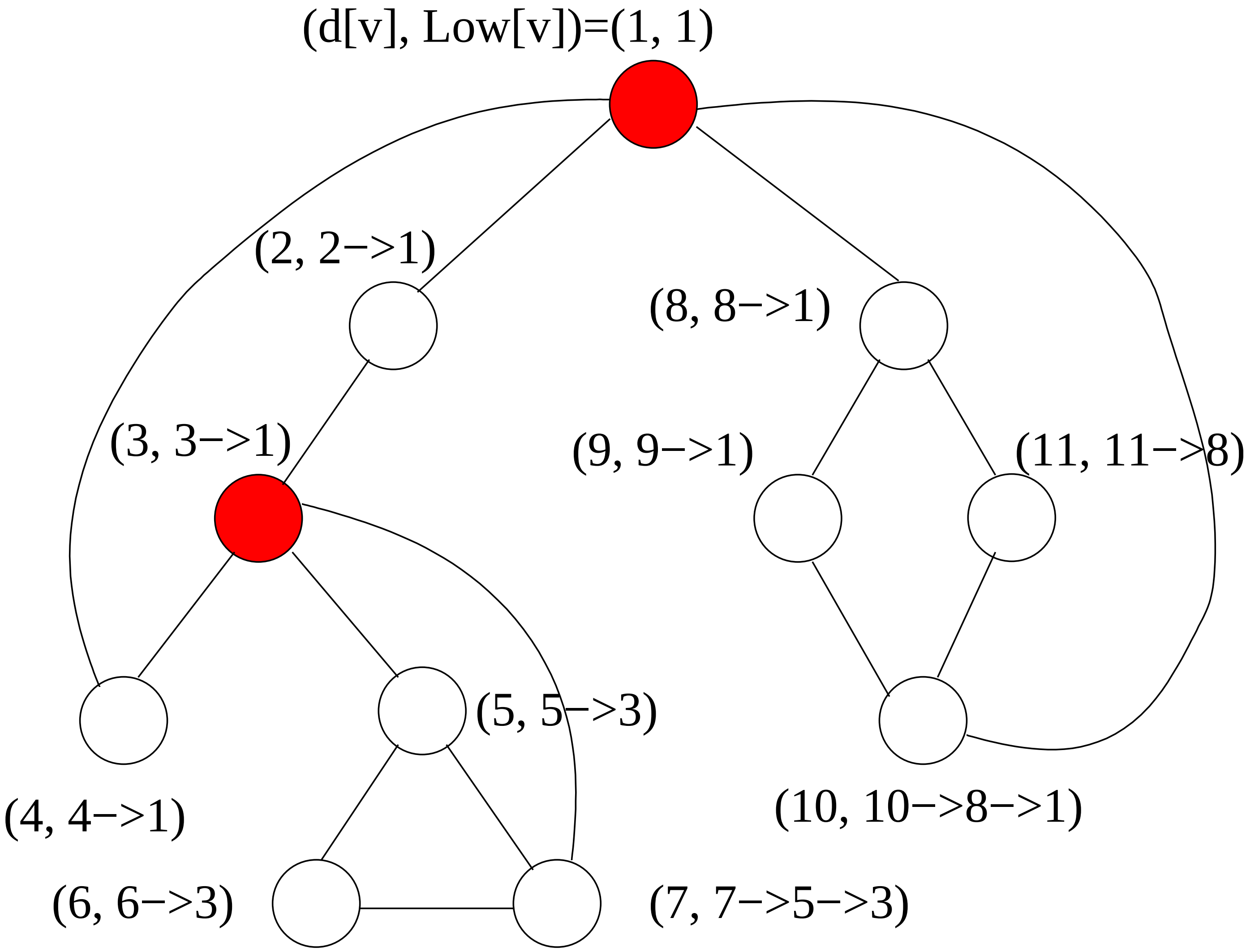
How do we know a subtree has a back edge climbing to an upper part of the tree ? We make use of *recursion*.

In the following algorithm, we define $Low[v]$ be the smallest value of a subtree rooted at v to which it can climb up by a back edge.

```
ArtPt(v) {
 color[v] = gray;
 // v initially can only climb up to itself
 Low[v] = d[v] = ++time;
 for all w in Adj(v) do {
  if (color[w] == white) {
   pred[w] = v;
   ArtPt(w);
   // When ArtPt(w) is completed, Low[w] stores the
   // lowest value it can climb up for a subtree
   // rooted at w.
   // Recall v is the parent of w.
   if (pred [v] == NULL) {
    // v has no predecessor , so v is the root.
    // apply observation 1.
    if ('w' is v's second child) output v;
   }
   else if (Low[w] >= d[v]) output v;
   // subtree rooted at w can't climb higher than v
   // apply observation 3.

   // update Low[v] if a children subtree can
   // climb higher
   Low[v] = min(Low[v], Low[w]);
  }
  else if (w != pred[v]) { // (v, w) is a back edge
   // update Low[v] if a back edge climbs higher
   Low[v] = min(Low[v], d[w]);
  }
 }
 color[v] = black;
}
```

Articulation points: Example



An Application of DFS : Biconnected components

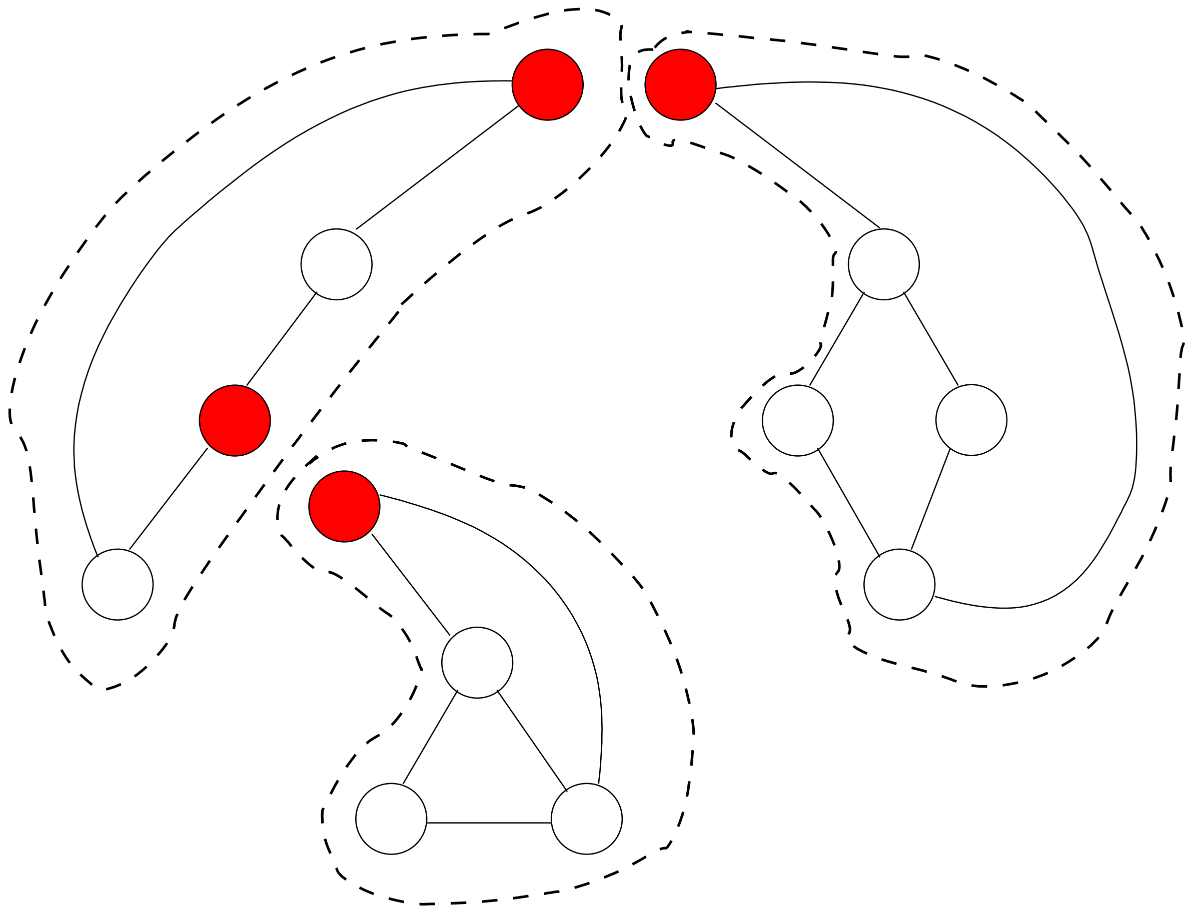
A **biconnected graph** $G = (V, E)$ is a connected graph which has no articulation points.

To disconnect a biconnected graph, we must remove at least two vertices.

A **biconnected component** of a graph G is maximal biconnected subgraph (i.e., it is not contained in any larger biconnected subgraph) of G.

The problem is how to identify all biconnected components of G?

An Application of DFS : Biconnected components



An Application of DFS : Biconnected components

Key observations:

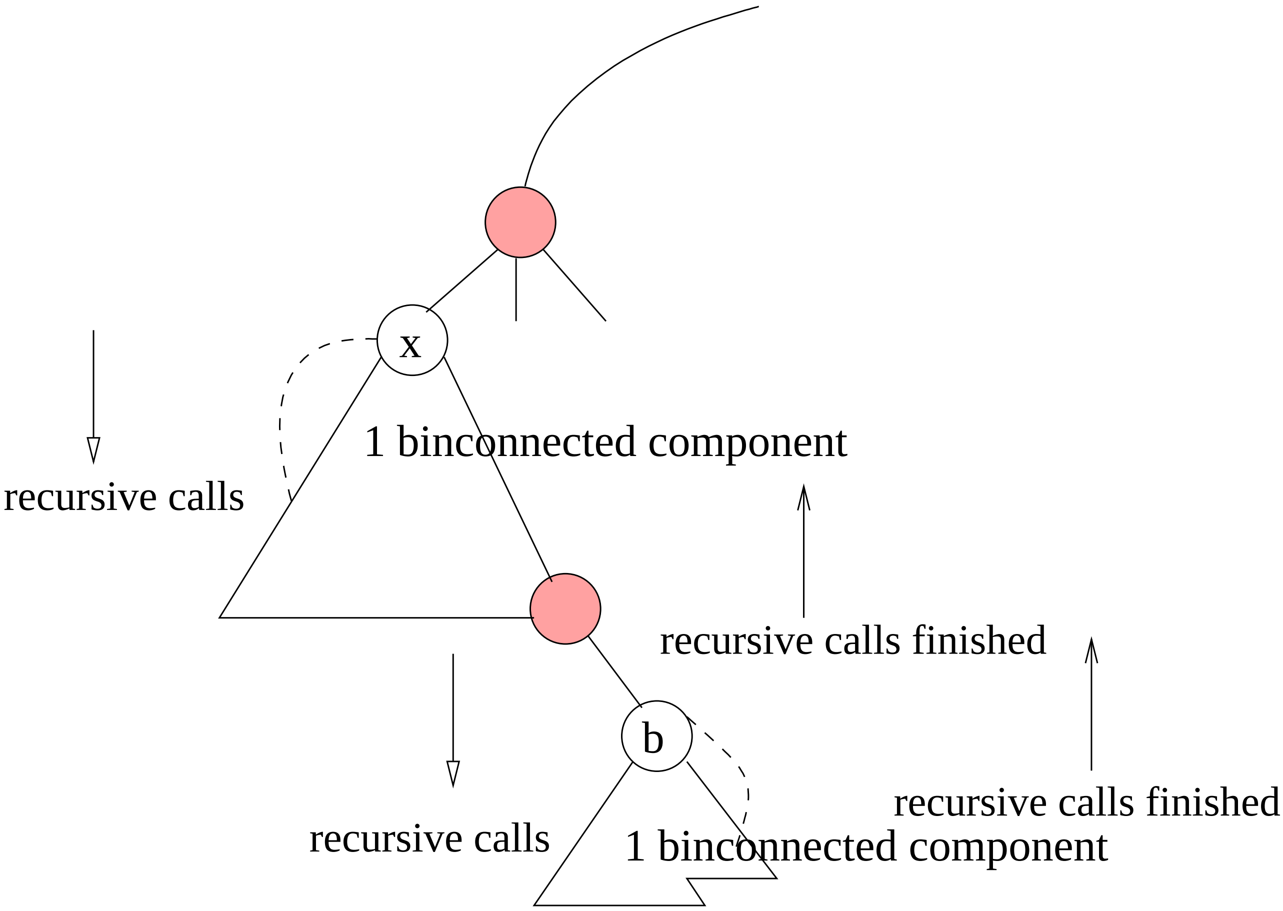
1. Two different biconnected components should not have any common edges (but they can have common vertex).

2. That common vertex linking two (or more) biconnected components must be an articulation point of G.

3. That is, the articulation points of G 'seperate' the biconnected components of G. If G has no articulation point, G is biconnected.

An Application of DFS : Biconnected components

It now boils down to find all the articulation points of G and check how they seperate the biconnected components.

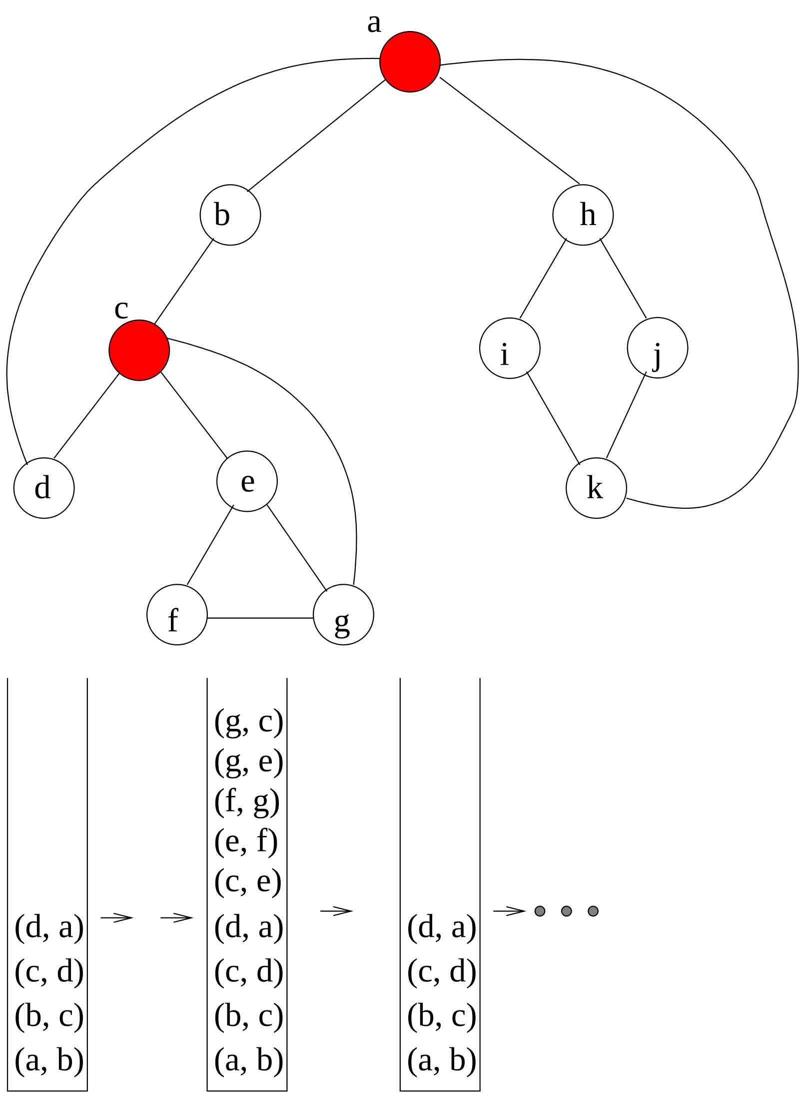


Again, we make use of the recursive call structure.

An Application of DFS : Biconnected components

Recall that DFS is a recursive algorithm, we make use of a *stack* to trace back the recursive calls. When we process an edge (u, x) (either by a recursive call on vertex x from vertex u, or (u, x) is back edge), we push that edge to a stack. Later, if we identify u as an articulation point (where the subtree rooted at x can't climb higher than u), then all the edges from the top of the stack down to (u, x) are the edges of one biconnected component. (Observe how a stack is used to trace the recursive calls). So we pop edges out of the stack until (u, x) (also pop (u, x)), those edges belong to a biconnected component.

An Application of DFS : Biconnected components



’c’ is idenfied as an articulation point, as ’e’ can’t climb higher; pop the stack until (c, e), those edges are in the same biconnected component