Articulation Points (APs) Recitation #6

Zhexiong Liu
zhexiong@cs.pitt.edu

Department of Computer Science
University of Pittsburgh

October 16, 2020
Slides adapted from Nicholas Farnan
Recap graph concepts

A graph $G(V, E)$ is a structure that contains a set of objects in which pairs of the objects are in some “related” sense

- $V$ is called vertex or node, $E$ is called edge
Recap graph concepts

A connected graph is an undirected graph in which every pair of vertices in the graph is connected

- Connected graph (left) vs disconnected graph (right)
Recap graph concepts

A biconnected graph is an undirected graph in which every pair of vertices is connected by at least 2 distinct paths that have no common edges or vertices

- Biconnected (left) vs non-biconnected (right) graphs
Articulation points

An articulation point (AP) or a cut vertex is any vertex whose removal increases the number of connected components

- Vertex 5 and node D are articulation points
Articulation points

An articulation point (AP) or a cut vertex is any vertex whose removal increases the number of connected components.

- Vertex 5 and node D are articulation points.
Articulation points

- Any graph that has **one or more** articulation points (cut vertices) is not biconnected
- Articulation points represent **vulnerabilities** in a graph as an articulate point failure would split the graph into two or more components
Recap depth-first search

DFS is an algorithm to traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking.
Recap depth-first search

DFS is an algorithm to traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking

- Start at an arbitrary vertex B
Recap depth-first search

DFS is an algorithm to traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking

- Start at an arbitrary vertex B

![Graph Diagram]

A -- B -- C
    |    |
    D -- E -- G
    |    |
    F
Recap depth-first search

DFS is an algorithm to Traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking

- Start at an arbitrary vertex B BCE
Recap depth-first search

DFS is an algorithm to traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking

- Start at an arbitrary vertex B

BCEG
Recap depth-first search

DFS is an algorithm to traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking

- Start at an arbitrary vertex B

BCEGF
Recap depth-first search

DFS is an algorithm to traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking

- Start at an arbitrary vertex B

BCEGFA
Recap depth-first search

DFS is an algorithm to traverse graphs (tree), which starts at an arbitrary vertex (or root node) and explores (spans) vertices (nodes) as far as possible before backtracking

- Start at an arbitrary vertex B

BCEGFAD
Recap depth-first search

Using DFS to traverse a graph will generate a spanning tree (ST) through the traversal order BCEGFA
Recap depth-first search

A DFS algorithm has three modes to traverse tree nodes; usually uses pre-order mode to traverse a graph

- Pre-order
- In-order
- Post-order modes
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

```
A
```

```
B  C
 |
D  E
 |
H  I
```

```
F  G
```
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

AB
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

ABD
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

ABDH
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

ABDHI
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

ABDHIE
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

ABDHIEC
Recap depth-first search

A DFS that expands tree nodes in the order of **root-left-right** is called pre-order DFS

- Pre-order DFS
  
  ABDHIECF
Recap depth-first search

A DFS that expands tree nodes in the order of root-left-right is called pre-order DFS

- Pre-order DFS

ABDHIECFG
Recap depth-first search

A DFS that expands tree nodes in the order of left-root-right is called in-order DFS

- In-order DFS
  
  HDIBEAFCG
Recap depth-first search

A DFS that expands tree nodes in the order of left-right-root is called post-order DFS

- Post-order DFS

HIDEBFGCA
Back edges

An edge is called back edge if this edge connects node M to node N such that node M is explored (spanned) before N’s parent node

- Dashed lines are back edges; solid lines are ST edges
Back edges

An edge is called back edge if this edge connects node M to node N such that node M is explored (spanned) before N’s parent node

- Dashed lines are back edges; solid lines are ST edges
Back edge properties

A back edge could compose an alternative path for node N to reach its previously explored (expanded) nodes in case node N’s parent node is removed

- Guarantees the graph (tree) is still connected upon the removal of N’s parent node
Back edge properties

A back edge could compose an alternative path for node N to reach its previously explored (expanded) nodes in case node N’s parent node is removed

- Guarantees the graph (tree) is still connected upon the removal of N’s parent node
- If we remove the parent node 3 the graph is still connected
Back edge properties

A back edge could compose an alternative path for node N to reach its previously explored (expanded) nodes in case node N’s parent node is removed

- Guarantees the graph (tree) is still connected upon the removal of N’s parent node
- If we remove the parent node 3 the graph is still connected
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- Node A
- Node B
- Node C
- Node D
- Node E
- Node F

Edges:
- A to B
- A to D
- B to C
- C to E
- C to F

Articulation points likely include nodes A and C.
Discover articulation points

Leverage DFS to discover articulating points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Graph representation:

- A
- B
- C
- D
- E
- F

Connections:
- A to B
- A to C
- B to D
- C to E
- C to F
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph Diagram]
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A graph with nodes A, B, C, D, E, and F, with edges connecting A to B, B to C, C to D, C to E, and E to F.
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph Diagram](image-url)
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The nodes A, B, C, D, E, F form a graph. The articulation point is C, indicated by the dashed line connecting D to C.
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
A
/   \
B   D
|   |
C
/   \
E   F
```

```plaintext
A
/   \
B   D
|   |
C
/   \
E   F
```

```plaintext
A
/   \
B   D
|   |
C
/   \
E   F
```
Discover articulation points

Leverage DFS to discover articulate points in a graph by maintaining pre-ordered & lowest-ordered arrays

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

D B A
C
E F

A B C D
E F

A B C D
E F

A B C D
E F

A B C D
E F

A B C D
E F
Discover articulation points

If a node has multiple reachable nodes that have lower node orders, take Low with the lowest node order

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Discover articulation points

If a node has multiple reachable nodes that have lower node orders, take Low with the lowest node order

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Discover articulation points

If a node has multiple reachable nodes that have lower node orders, take Low with \textbf{the lowest node order}

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Discover articulation points

If a node has multiple reachable nodes that have lower node orders, take Low with the lowest node order

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

Given distinct vertex u and vertex v in the graph, where u is the parent and v is the child (v is discovered by u)

If u==root
  u is an articulation point
  if u has more than one child

If u!=root
  u is an articulation point
  if Low(u) >= Pre(v)

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

If $u == \text{root}$
   u is an articulation point
if u has more than one child
   - A is not an articulation point

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

If $u == \text{root}$
- $u$ is an articulation point
  - if $u$ has more than one child
    - A is not an articulation point

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

If \( u \neq \text{root} \)
- \( u \) is an articulation point
- If \( \text{Low}(v) \geq \text{Pre}(u) \)
  - If \((u,v) = (B,C)\), \( u \) is parent, \( v \) is child
  - since \( \text{Low}(C) < \text{Pre}(B) \), \( B \) is not AP

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

If \( u \neq \text{root} \)

- \( u \) is an articulation point
- if \( \text{Low}(v) \geq \text{Pre}(u) \)
  - If \((u,v) = (B,C)\), \( u \) is parent, \( v \) is child
    - since \( \text{Low}(C) < \text{Pre}(B) \), \( B \) is not AP

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

If \( u!\text{=}\text{root} \)

\( u \) is an articulation point

if \( \text{Low}(v) \geq \text{Pre}(u) \)

- If \((u,v)\text{=}\text{(C,E)}, u \text{ is parent, v is child}\)

\( \text{since Low(E) \geq Pre(C)}, C \text{ is AP} \)

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

If \( u \neq \text{root} \)
  u is an articulation point
if \( \text{Low}(v) \geq \text{Pre}(u) \)
  - If \( (u,v)=(C,E) \), u is parent, v is child
    since \( \text{Low}(E) \geq \text{Pre}(C) \), C is AP

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Algorithm of discovering AP

If $u \neq \text{root}$
   
   $u$ is an articulation point

   if $\text{Low}(v) \geq \text{Pre}(u)$

   - If $(u,v) = (C,E)$, $u$ is parent, $v$ is child
     
     since $\text{Low}(E) < \text{Pre}(C)$, $C$ is not AP
Algorithm of discovering AP

If \( u \neq \text{root} \)
  \( u \) is an articulation point
if \( \text{Low}(v) \geq \text{Pre}(u) \)
  - If \((u,v) = (C,E)\), \( u \) is parent, \( v \) is child
  since \( \text{Low}(E) < \text{Pre}(C) \), \( C \) is not AP
Algorithm of maintaining Pre & Low

Pre(v) = pre-order traversal order

Low(v) = order of the lowest node reached by vertex v through a path that contains zero/more spanning tree edges and at most one back edge

- Case 1: single node v, where v is node A
  Low(v) = Pre(v)

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm of maintaining Pre & Low

Pre(v) = pre-order traversal order

Low(v) = order of the lowest node reached by vertex v through a path that contains zero/more spanning tree edges and at most one back edge

- Case 2: edge v-w is a spanning tree edge where v is parent E, w is child F

Low(v) = min[Low(v), Low(w)]

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm of maintaining Pre & Low

Pre(v) = pre-order traversal order

Low(v) = order of the lowest node reached by vertex v through a path that contains zero/more spanning tree edges and at most one back edge

- Case 3: edge v-w is a back edge where v is parent F, w is child A or C

Low(v) = min[Low(v), Pre(w)]

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm implementation

```java
// private int[] low;
// private int[] pre;
// private int cnt;
// private boolean[] articulation;

public Biconnected(Graph G) {
    low = new int[G.V()];
    pre = new int[G.V()];
    articulation = new boolean[G.V()];
    // initial low
    for (int v = 0; v < G.V(); v++)
        low[v] = -1;
    // initial pre
    for (int v = 0; v < G.V(); v++)
        pre[v] = -1;
    // run dfs algorithm
    for (int v = 0; v < G.V(); v++)
        if (pre[v] == -1)
            dfs(G, v, v);
}

private void dfs(Graph G, int u, int v) {
    int children = 0;
    pre[v] = cnt++;
    // pre-order node number
    low[v] = pre[v];
    // case1: single node
    for (int w : G.adj(v)) {
        if (pre[w] == -1) {
            children++;
            dfs(G, v, w);
            // case2: spanning tree edges
            low[v] = Math.min(low[v], low[w]);
            // check articulation points
            if (low[w] >= pre[v] && u != v)
                articulation[v] = true;
        } else if (w != u)
            // case3: back edges
            low[v] = Math.min(low[v], pre[w]);
    }
    // root has more than one children
    if (u == v && children > 1)
        articulation[v] = true;
}
```
What is the time complexity

- Brute-force algorithm of discovering articulate points in $O(V^*(E+V))$, but the discovery can be done in $O(E+V)$, which is the same as DFS time complexity