

---

# INTRO TO ALGORITHMS ENGINEERING

---

## Course Project

---

### Project 1 - Study of biconnectivity algorithms

#### Author

Priet Ukani

2022111039

Sumit Kumar

2022111012

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Objective</b>	<b>3</b>
<b>3</b>	<b>Tarjan Algorithm</b>	<b>3</b>
3.1	History . . . . .	3
3.2	Introduction . . . . .	3
3.3	Working of Tarjan's Algorithm . . . . .	4
3.4	Proof of Correctness . . . . .	4
3.5	Time Complexity Analysis . . . . .	5
3.6	Space Complexity Analysis . . . . .	6
3.7	Pseudo-code . . . . .	7
3.8	Code Explanation . . . . .	7
3.9	Example . . . . .	9
<b>4</b>	<b>Jen-Schmidt Algorithm</b>	<b>11</b>
4.1	History . . . . .	11
4.2	Introduction . . . . .	11
4.3	Working of Schmidt's Algorithm . . . . .	11
4.4	Proof of Correctness . . . . .	13
4.5	Time Complexity Analysis . . . . .	14
4.6	Space Complexity Analysis . . . . .	14
4.7	Pseudo-code . . . . .	14
4.8	Code Explanation . . . . .	16
<b>5</b>	<b>System Specifications</b>	<b>16</b>
<b>6</b>	<b>Code correctness</b>	<b>16</b>
<b>7</b>	<b>Experiments</b>	<b>17</b>
<b>8</b>	<b>Comparative Study</b>	<b>18</b>
<b>9</b>	<b>Conclusion</b>	<b>21</b>

# 1 Introduction

Biconnectivity is a fundamental concept in graph theory, which refers to the property of a graph where there exists at least two vertex-disjoint paths between any pair of vertices. It describes the connectivity of a graph when removing a single vertex does not disconnect the graph. It plays a crucial role in various applications, including network analysis, transportation planning, and circuit design. Two well-known algorithms for identifying biconnected components in a graph are Tarjan's algorithm and Schmidt's algorithm.

## 2 Objective

In this project, we will conduct a comprehensive study of the Tarjan and Schmidt algorithms, focusing on their practical differences in terms of runtime performance, behavior on sparse versus dense graphs, and their suitability for various graph topologies. We will implement both algorithms and compare their performance using a variety of test cases, including synthetic and real-world graph datasets. The goal of this study is to provide a deeper understanding of the strengths and weaknesses of these biconnectivity algorithms, which can inform the selection of the appropriate algorithm for specific applications and graph characteristics.

## 3 Tarjan Algorithm

### 3.1 History

Tarjan's algorithm for finding biconnected components was introduced by **Robert Tarjan** in 1972 in his paper titled "Depth-first search and linear graph algorithms". It is a fundamental algorithm in graph theory and has played a crucial role in the development of efficient algorithms for various graph problems.

Prior to Tarjan's work, the problem of identifying biconnected components in a graph was considered a difficult task, with no known efficient algorithms. The existing approaches at the time were either impractical or had significant limitations, such as requiring the graph to be planar or having specific properties.

Over the years, Tarjan's algorithm has been extensively studied, analyzed, and optimized by researchers in the field of graph theory and algorithms. It has become a fundamental building block in many graph-based applications, ranging from network analysis and computer networking to computational biology and software engineering.

Today, Tarjan's algorithm for finding biconnected components remains a cornerstone in the field of graph algorithms, and its principles continue to inspire new algorithmic developments and theoretical insights.

### 3.2 Introduction

Tarjan's algorithm revolutionized the field by providing a simple and elegant solution to the biconnectivity problem. It leverages the principles of depth-first search (DFS) and uses a clever technique of assigning low-link values to vertices, which capture the connectivity information needed to identify biconnected components.

The algorithm has a time complexity of  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph. This linear-time complexity made it highly efficient and practical for large-scale graphs, enabling its widespread adoption in various applications.

### 3.3 Working of Tarjan's Algorithm

1. **Initialization:** The algorithm starts by assigning a unique number (usually called the discovery time) to each vertex during a depth-first search (DFS) traversal. Additionally, it maintains two values for each vertex:
  - **low:** The lowest discovery time reachable from the current vertex or any of its descendants through back-edges.
  - **parent:** The parent vertex in the DFS tree.
2. **DFS Traversal:** The algorithm performs a DFS traversal on the graph. For each vertex  $v$  encountered during the traversal, it does the following:
  - (a) Assign  $v$  a discovery time (say,  $\text{disc}[v]$ ).
  - (b) Initialize  $\text{low}[v]$  to  $\text{disc}[v]$ .
  - (c) For each unvisited neighbor  $u$  of  $v$ , recursively call the DFS on  $u$ , setting  $\text{parent}[u]$  to  $v$ . After the recursive call returns, update  $\text{low}[v]$  to be the minimum of  $\text{low}[v]$  and  $\text{low}[u]$ .
  - (d) For each visited neighbor  $u$  of  $v$  (except  $\text{parent}[v]$ ), update  $\text{low}[v]$  to be the minimum of  $\text{low}[v]$  and  $\text{disc}[u]$ . This step handles back-edges in the graph.
3. **Identifying Biconnected Components:** After the DFS traversal, the algorithm identifies biconnected components based on the following conditions:
  - If  $u$  is the root of the DFS tree and has at least two children, then  $u$  is an articulation point (cut-vertex), and each child and its descendants form a separate biconnected component.
  - If  $u$  is not the root and  $\text{low}[v] \geq \text{disc}[u]$  for some child  $v$  of  $u$ , then  $u$  is an articulation point, and each child  $v$  (for which  $\text{low}[v] \geq \text{disc}[u]$ ) and its descendants form a separate biconnected component.
  - If  $u$  is not an articulation point, then all vertices reachable from  $u$  through tree edges (and not already part of another biconnected component) form a single biconnected component.
4. **Output:** The algorithm outputs the biconnected components as sets of vertices that belong to the same biconnected component.

### 3.4 Proof of Correctness

We will prove the correctness of Tarjan's algorithm by showing that the biconnected components identified by the algorithm satisfy the following two properties:

1. Each biconnected component given by algorithm is indeed biconnected.

2. No two biconnected components share a vertex. (Each biconnected component is disjoint)

*Proof.* 1. **Each biconnected component given by algorithm is indeed biconnected:**

Consider a set of vertices  $S$  identified as a biconnected component by the algorithm. If  $S$  contains only a single vertex, then it is trivially biconnected. If  $S$  contains more than one vertex, we need to show that there exist at least two vertex-disjoint paths between every pair of vertices in  $S$ .

Let  $u$  and  $v$  be any two vertices in  $S$ . Since  $S$  is a biconnected component, there must be a path from  $u$  to  $v$  in the DFS tree. Let  $w$  be the lowest common ancestor of  $u$  and  $v$  in the DFS tree. By the algorithm's condition for identifying biconnected components, there must exist a back-edge  $(x, y)$  such that  $\text{low}[x] \leq \text{disc}[w]$ . This back-edge provides an alternate path from  $u$  to  $v$  that does not include  $w$ . Therefore, there are at least two vertex-disjoint paths between  $u$  and  $v$ .

2. **No two biconnected components share a vertex:**

Suppose there are two biconnected components  $S_1$  and  $S_2$  that share a vertex  $v$ . Let  $u_1$  and  $u_2$  be two vertices in  $S_1$  and  $S_2$ , respectively, such that  $u_1 \neq v \neq u_2$ .

Since  $S_1$  and  $S_2$  are biconnected components, there exist two vertex-disjoint paths between  $u_1$  and  $v$ , and two vertex-disjoint paths between  $u_2$  and  $v$ . These four paths together form a cycle containing  $u_1$ ,  $u_2$ , and  $v$ . However, the existence of this cycle contradicts the condition that  $S_1$  and  $S_2$  are separate biconnected components.

Therefore, no two biconnected components identified by the algorithm can share a vertex.

□

By satisfying the two properties above, we can conclude that Tarjan's algorithm correctly identifies all the biconnected components any graph.

### 3.5 Time Complexity Analysis

We will analyze the time complexity of Tarjan's algorithm for finding biconnected components in a graph with  $V$  vertices and  $E$  edges.

The algorithm consists of two main phases:

1. **Depth-First Search (DFS) Traversal:**

During the DFS traversal, each vertex is visited exactly once, and each edge is explored twice (once during the tree exploration and once during the back-edge exploration). Therefore, the time complexity of the DFS traversal is  $\mathcal{O}(V + E)$ .

2. **Identifying Biconnected Components:**

In this phase, the algorithm performs constant-time operations for each vertex and edge to update the low values and identify the biconnected components. These operations include:

- Updating  $\text{low}[v]$  for each unvisited neighbor  $u$  of  $v$ .
- Updating  $\text{low}[v]$  for each visited neighbor  $u$  of  $v$  (except  $\text{parent}[v]$ ).
- Identifying articulation points and biconnected components based on the low values.

Since these operations are performed for each edge, the time complexity of this phase is also  $\mathcal{O}(V + E)$ .

Therefore, the overall time complexity of Tarjan's algorithm for finding biconnected components is  $\mathcal{O}(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph. It is important to note that this time complexity analysis assumes that the graph is represented using an adjacency list data structure, which allows constant-time access to the neighbors of a vertex.

If an adjacency matrix representation is used instead, the time complexity would become  $\mathcal{O}(V^2 + E)$  due to the need to iterate over all vertices for each vertex's neighbors.

### 3.6 Space Complexity Analysis

In addition to the time complexity analysis, it is important to examine the space complexity of Tarjan's algorithm for finding biconnected components in a graph.

The primary data structures used by the algorithm are:

- An adjacency list representation of the graph, which requires  $\mathcal{O}(V + E)$  space, where  $V$  is the number of vertices and  $E$  is the number of edges.
- An array or vector to store the discovery time (or any unique number) for each vertex, requiring  $\mathcal{O}(V)$  space.
- An array or vector to store the low values for each vertex, requiring  $\mathcal{O}(V)$  space.
- An array or vector to store the parent values for each vertex, requiring  $\mathcal{O}(V)$  space.
- A stack or recursion stack for the depth-first search (DFS) traversal, which can take up to  $\mathcal{O}(V)$  space in the worst case when the graph is a single chain or path.

The space required for storing the biconnected components themselves depends on the output representation. If we store each biconnected component as a list or set of vertices, the additional space required would be proportional to the total number of vertices in all biconnected components, which is  $\mathcal{O}(V)$  in the worst case.

Therefore, the overall space complexity of Tarjan's algorithm for finding biconnected components is  $\mathcal{O}(V + E)$ , dominated by the space required for the graph representation and the auxiliary arrays or vectors used during the algorithm's execution.

It is worth noting that the space complexity can be reduced to  $\mathcal{O}(V)$  if the graph is represented using an adjacency matrix instead of an adjacency list, at the cost of increased time complexity for accessing neighbors. However, for sparse graphs with  $E \ll V^2$ , the adjacency list representation is more space-efficient.

### 3.7 Pseudo-code

---

**Algorithm 1** Tarjan's Algorithm for Biconnected Components

---

```

1: procedure BICONNECTEDCOMPONENTS( $G = (V, E)$ )
2:   time  $\leftarrow 0$ 
3:   for  $u \in V$  do
4:     disc[ $u$ ]  $\leftarrow \infty$  ▷ Initialize discovery times
5:     low[ $u$ ]  $\leftarrow \infty$ 
6:     parent[ $u$ ]  $\leftarrow \text{NIL}$ 
7:   end for
8:   for  $u \in V$  do
9:     if disc[ $u$ ] =  $\infty$  then
10:      DFS( $G, u$ )
11:    end if
12:  end for
13: end procedure
14: procedure DFS( $G, u$ )
15:   disc[ $u$ ]  $\leftarrow$  low[ $u$ ]  $\leftarrow$  time
16:   time  $\leftarrow$  time + 1
17:   children  $\leftarrow 0$ 
18:   for  $v \in \text{Adj}[u]$  do ▷ Adj[ $u$ ] is the set of adjacent vertices of  $u$ 
19:     if disc[ $v$ ] =  $\infty$  then
20:       children  $\leftarrow$  children + 1
21:       parent[ $v$ ]  $\leftarrow u$ 
22:       DFS( $G, v$ )
23:       low[ $u$ ]  $\leftarrow \min(\text{low}[u], \text{low}[v])$ 
24:     else if  $v \neq \text{parent}[u]$  then ▷ Back-edge
25:       low[ $u$ ]  $\leftarrow \min(\text{low}[u], \text{disc}[v])$ 
26:     end if
27:   end for
28:   if parent[ $u$ ] = NIL and children > 1 then
29:     OUTPUTBICONNECTEDCOMPONENT( $u$ )
30:   else if parent[ $u$ ]  $\neq$  NIL and low[ $v$ ]  $\geq$  disc[ $u$ ] for some  $v$  then
31:     OUTPUTBICONNECTEDCOMPONENT( $u$ )
32:   end if
33: end procedure
34: procedure OUTPUTBICONNECTEDCOMPONENT( $u$ )
35:   Output the biconnected component containing  $u$ 
36: end procedure

```

---

### 3.8 Code Explanation

Here we have written two codes for the algorithm for Tarjan. One code for just checking the biconnectedness of the graph and one for finding the total number of biconnected components and printing them. This is for effective comparison and analysis with the Jen-Schmidt

algorithm as Jen-Schmidt only finds the biconnectedness of the graph.

## Tarjan's Algorithm

### 1. Functionality:

- The algorithm uses depth-first search (DFS) to find the articulation points and bridges in the graph.
- It marks the edges forming the biconnected components.
- It determines if the graph is biconnected or not.

### 2. Code Explanation:

- Uses DFS with an additional array to keep track of the discovery time and low value for each vertex.
- It updates the low value based on the discovery time of adjacent vertices.
- When a back edge is found (low value of the adjacent vertex is greater than or equal to the discovery time of the current vertex), it signifies the presence of a cycle.
- The algorithm considers the special case of the root node separately.
- Finally, it checks if all nodes are visited or not to ensure the graph is connected.

### 3. Time Complexity:

- Tarjan's algorithm typically runs in  $O(V + E)$  time, where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.

### 4. Space Complexity:

- The space complexity is  $O(V)$  for maintaining the visited, discovery, and low arrays, where  $V$  is the number of vertices.

## Tarjan's Extended Algorithm

### 1. Functionality:

- This algorithm extends Tarjan's algorithm to find and count the number of biconnected components in the graph.
- It handles cases where there are single nodes forming their own biconnected components.

### 2. Code Explanation:

- Similar to Tarjan's algorithm, it uses DFS, but it extends the functionality to handle biconnected components explicitly.
- It uses a stack to keep track of edges forming the biconnected components.
- When a biconnected component is found, it increments the count and marks the nodes in the component as visited.



- After DFS, it counts the unvisited nodes, considering them as single-node biconnected components.

### 3. Time Complexity:

- The time complexity remains the same as Tarjan's algorithm, i.e.,  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges.

### 4. Space Complexity:

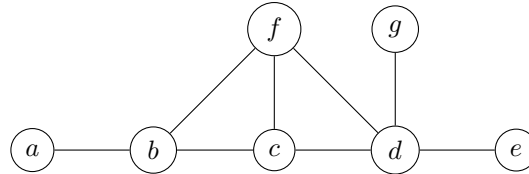
- The space complexity is also  $O(V)$  for maintaining the visited array and the stack, where  $V$  is the number of vertices.

## Comparative Analysis

- Both algorithms use DFS and have similar time and space complexities.
- Tarjan's Extended Algorithm extends the functionality of Tarjan's algorithm to explicitly count the number of biconnected components or print them.
- Tarjan's Extended Algorithm is more suitable when you need to count or explicitly handle the biconnected components in the graph.

## 3.9 Example

Consider the following undirected graph  $G$ :



We will apply Tarjan's algorithm to find the biconnectedness of this graph.

### Tarjan's Algorithm

#### 1. Initialization:

- We have the following vertices:  $a, b, c, d, e, f$ , and  $g$ .
- The edges are:  $(a, b), (b, c), (c, d), (d, e), (b, f), (c, f), (d, f)$ , and  $(d, g)$ .

#### 2. Adjacency List Representation:

- After converting the given graph into an adjacency list representation:

```

a -> b
b -> a, c, f
c -> b, d, f
d -> c, e, f, g
e -> d
f -> b, c, d
g -> d

```

### 3. DFS Traversal:

- We start DFS from node  $a$ .
- DFS explores the graph depth-first, marking nodes as visited and updating the low and discovery values.

**Node  $a$ :**

- It's the starting node.
- It visits node  $b$ .

**Node  $b$ :**

- It visits nodes  $a$ ,  $c$ , and  $f$ .
- Since  $a$  was already visited, it doesn't update the low or discovery values.
- It explores  $c$ .

**Node  $c$ :**

- It visits nodes  $b$ ,  $d$ , and  $f$ .
- $b$  was already visited.
- It explores  $d$ .

**Node  $d$ :**

- It visits nodes  $c$ ,  $e$ ,  $f$ , and  $g$ .
- $c$  was already visited.
- It explores  $e$ .

**Node  $e$ :**

- It visits node  $d$ .
- It doesn't update the low or discovery values.

**Node  $f$ :**

- It visits nodes  $b$ ,  $c$ , and  $d$ .
- All these nodes were already visited.

**Node  $g$ :**

- It visits node  $d$ .
- It doesn't update the low or discovery values.

### 4. Results:

- After DFS traversal, all nodes are visited, indicating that this passes the check for biconnectedness.
- The algorithm also computes the low and discovery values for each node.
- Based on this low and discovery values, we figure out that the given graph is not Biconnected.

### 5. Output:

- The program prints the elapsed time taken to perform the computation.
- It may also print "No" to indicate the given graph is not biconnected.

## 4 Jen-Schmidt Algorithm

### 4.1 History

This algorithm was published in 2012 by **Jens M. Schmidt** in his paper A Simple Test on 2-Vertex and 2-Edge-Connectivity. This algorithm's primary aim is to find the ear decomposition of a graph which also helps in testing 2-edge and 2-vertex connectivity of the graph. There is not much information about the author Jens M. Schmidt on the internet. Currently, he is the Chair of Algorithms and Complexity at University of Rostock, Germany. His research areas include Algorithmic Graph Theory, Connectivity in Networks, and Simplification and Certification of Algorithms.

### 4.2 Introduction

Testing a graph on 2-vertex-connectivity and on 2-edge-connectivity are fundamental algorithmic graph problems. Tarjan presented the first linear-time algorithms for these problems. Since then, many linear-time algorithms have been given that compute structures which inherently characterize either the 2- or 2-edge-connectivity of a graph. The aim of this algorithm is a self-contained exposition of an even simpler linear-time algorithm that tests both the 2-vertex and 2-edge-connectivity of a graph.

#### Notation:

1. A cut vertex is a vertex in a connected graph that disconnects the graph upon deletion.
2. A bridge is an edge in a connected graph that disconnects the graph upon deletion.
3. A graph is 2-connected if it is connected and contains at least 3 vertices, but no cut vertex.
4. A graph is 2-edge-connected if it is connected and contains at least 2 vertices, but no bridge.

### 4.3 Working of Schmidt's Algorithm

We will decompose the input graph into a set of paths and cycles, each of which will be called a chain. Some easy-to-check properties on these chains will then characterize both the 2-vertex and 2-edge-connectivity of the graph.

Let  $G = (V, E)$  be the input graph and assume for convenience that  $G$  is simple and that  $|V| \geq 3$ .

1. We first perform a depth-first search on  $G$ . This implicitly checks  $G$  on being connected. If  $G$  is connected, we get a DFS-tree  $T$  that is rooted on a vertex  $r$ ; otherwise, we stop, as  $G$  is neither 2- nor 2-edge-connected.
2. The DFS assigns a depth-first index (DFI) to every vertex. We assume that all tree edges (i. e., edges in  $T$ ) are oriented towards  $r$  and all backedges (i. e., edges that are in  $G$  but not in  $T$ ) are oriented away from  $r$ . Thus, every backedge lies in exactly one directed cycle  $C(e)$ . Let every vertex be marked as unvisited.

3. We now decompose  $G$  into chains by applying the following procedure for each vertex  $v$  in ascending DFI-order:
  - (a) For every backedge  $e$  that starts at  $v$ , we traverse  $C(e)$ , beginning with  $v$ , and stop at the first vertex that is marked as visited. During such a traversal, every traversed vertex is marked as visited.
  - (b) We call this path or cycle a **chain** and identify it with the list of vertices and edges in the order in which they were visited. The  $i^{th}$  chain found by this procedure is referred to as  $C_i$ .
  - (c) The chain  $C_1$ , if exists, is a cycle, as every vertex is unvisited at the beginning (note  $C_1$  does not have to contain  $r$ ). There are  $|E| - |V| + 1$  chains, as every of the  $|E| - |V| + 1$  backedges creates exactly one chain.
  - (d) We call the set  $C = C_1, \dots, C_{|E|-|V|+1}$  a **chain decomposition**.

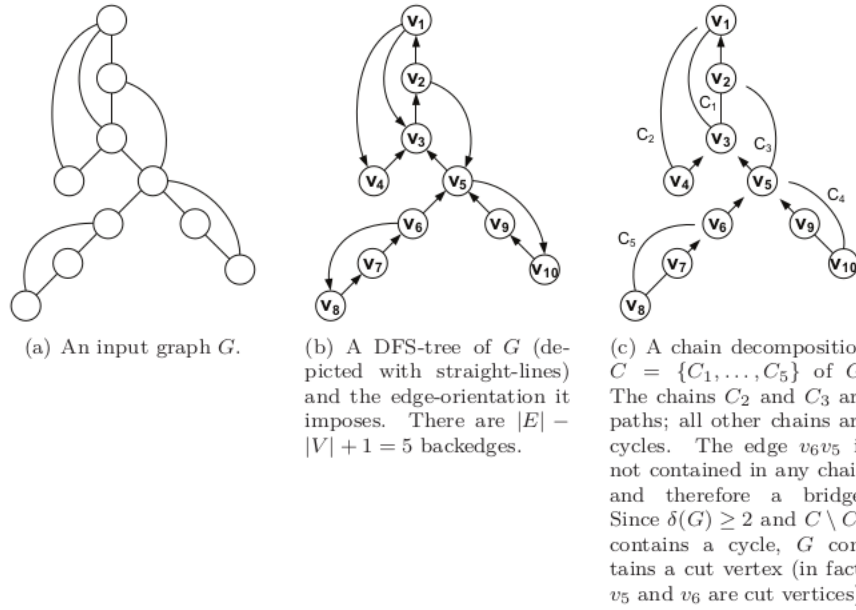


Figure 1: A graph  $G$ , its DFS-tree and a chain decomposition of  $G$ .

This concludes the algorithmic part.

Now, we check the conditions on  $C$  that characterize 2-vertex and 2-edge connectivity.

1. Let  $C$  be a chain decomposition of a simple connected graph  $G$ . Then  $G$  is 2-edge-connected if and only if the chains in  $C$  partition  $E$ .
2. Let  $C$  be a chain decomposition of a simple 2-edge-connected graph  $G$ . Then  $G$  is 2-connected if and only if  $C_1$  is the only cycle in  $C$ .
3. Let  $C$  be a chain decomposition of a simple connected graph  $G$ . Then  $G$  is 2-connected if and only if minimum degree of  $G \geq 2$  and  $C_1$  is the only cycle in  $C$ .

#### 4.4 Proof of Correctness

Above conclusions from the chain decomposition can be directly concluded from the following lemmas:

---

**Lemma 1:** Let  $C$  be a chain decomposition of a simple connected graph  $G$ . An edge  $e$  in  $G$  is a bridge if and only if  $e$  is not contained in any chain in  $C$ .

**Proof:** Let  $e$  be a bridge and assume to the contrary that  $e$  is contained in a chain whose first edge (i. e., whose backedge) is  $b$ . The bridge  $e$  is not contained in any cycle of  $G$ , as otherwise the end points of  $e$  would still be connected when deleting  $e$ , contradicting that  $e$  is a bridge. This contradicts the fact that  $e$  is contained in the cycle  $C(b)$ .

Let  $e$  be an edge that is not contained in any chain in  $C$ . Let  $T$  be the DFS-tree that was used for computing  $C$  and let  $x$  be the end point of  $e$  that is farthest away from the root  $r$  of  $T$ , in particular  $x \neq r$ . Then  $e$  is a tree-edge, as otherwise  $e$  would be contained in a chain. For the same reason, there is no backedge with exactly one end point in  $T(x)$ . Deleting  $e$  therefore disconnects all vertices in  $T(x)$  from  $r$ . Hence,  $e$  is a bridge.

---

**Lemma 2:** Let  $C$  be a chain decomposition of a simple connected graph  $G$  with minimum degree  $\geq 2$ . A vertex  $v$  in  $G$  is a cut vertex if and only if  $v$  is incident to a bridge or  $v$  is the first vertex of a cycle in  $C \setminus C_1$ .

**Proof:** Let  $v$  be a cut vertex in  $G$ ; we may assume that  $v$  is not incident to a bridge. Let  $X$  and  $Y$  be connected components of  $G \setminus v$ . Then  $X$  and  $Y$  have to contain at least two neighbors of  $v$  in  $G$ , respectively. Let  $X^{+v}$  and  $Y^{+v}$  denote the subgraphs of  $G$  that are induced by  $X \cup v$  and  $Y \cup v$ , respectively. Both  $X^{+v}$  and  $Y^{+v}$  contain a cycle through  $v$ , as both  $X$  and  $Y$  are connected. It follows that  $C_1$  exists; assume w.l.o.g. that  $C_1 \notin X^{+v}$ . Then there is at least one backedge in  $X^{+v}$  that starts at  $v$ . When the first such backedge is traversed in the chain decomposition, every vertex in  $X$  is still unvisited. The traversal therefore closes a cycle that starts at  $v$  and is different from  $C_1$ , as  $C_1 \notin X^{+v}$ .

If  $v$  is incident to a bridge, minimum degree of  $G \geq 2$  implies that  $v$  is a cut vertex. Let  $v$  be the first vertex of a cycle  $C_i \neq C_1$  in  $C$ . If  $v$  is the root  $r$  of the DFS-tree  $T$  that was used for computing  $C$ , both cycles  $C_1$  and  $C_i$  end at  $v$ . Thus,  $v$  has at least two children in  $T$  and  $v$  must be a cut vertex. Otherwise  $v \neq r$ ; let  $w - v$  be the last edge in  $C_i$ . Then no backedge starts at a vertex with smaller DFI than  $v$  and ends at a vertex in  $T(w)$ , as otherwise  $v - w$  would not be contained in  $C_i$ . Thus, deleting  $v$  separates  $r$  from all vertices in  $T(w)$  and  $v$  is a cut vertex.

---

## 4.5 Time Complexity Analysis

- DFS Traversal: The time complexity of a DFS traversal is  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.
- Finding Chains: For each backedge, the algorithm traverses the chain, which could potentially visit every vertex and edge in the graph once. Since there are  $|E| - |V| + 1$  backedges and each edge/vertex only gets visited atmost once, the total time complexity for finding chains is  $O(E + V)$ .

Overall, the time complexity of the algorithm is dominated by the DFS traversal and finding chains, so it is  $O(V + E)$ .

## 4.6 Space Complexity Analysis

- Adjacency List: The space required to store the graph's adjacency list is  $O(V + E)$ .
- Visited Array: A boolean array of size  $V$  is used to keep track of visited vertices during DFS, requiring  $O(V)$  space.
- Edge List: The DFS traversal may create an edge list of size at most  $E$ , as there can be at most  $E$  edges in the DFS tree.
- Chains: The space required to store the chains depends on the number of vertices and edges in the graph. In the worst case, if the graph is a tree, the space required could be  $O(V)$  for each chain.

Therefore, the overall space complexity is  $O(V + E)$ , dominated by the adjacency list and the space required for the DFS traversal and storing chains.

## 4.7 Pseudo-code

---

**Algorithm 2** DFS(int *node*, int *par*)

---

```
1: Add node to euler
2: tin[node]  $\leftarrow$  length of euler
3: vis[node]  $\leftarrow$  true
4: for each k in g[node] do
5:   if  $\neg$ vis[k] then
6:     Add (node, 0) to dir[k]
7:     Call DFS(k, node)
8:   else if vis[k] and k  $\neq$  par and tin[node] > tin[k] then
9:     Add (node, 1) to dir[k]
10:  end if
11: end for
```

---

---

**Algorithm 3** ChainDecomposition(*euler*, *dir*)

---

```
1: Initialize empty array chain_decomposition
2: Initialize integer total  $\leftarrow 0$ 
3: for each node in euler do
4:   for each (v, isTree) in dir[node] do
5:     if  $\neg isTree$  then
6:       Initialize empty array chain
7:       Add node to chain
8:       v  $\leftarrow$  next node in chain from dir
9:       while v  $\neq$  node do
10:        if vis[v] then
11:          Break
12:        end if
13:        Add v to chain
14:        vis[v]  $\leftarrow$  true
15:        v  $\leftarrow$  next node in chain from dir[v]
16:      end while
17:      if v  $\neq$  node then
18:        Add v to chain
19:      end if
20:      Add chain to chain_decomposition
21:      total  $\leftarrow$  total + length of chain - 1
22:    end if
23:  end for
24: end for
25: if total  $\neq m$  then
26:   return null
27: end if
28: for i = 1 to length of chain_decomposition do
29:   if first element of chain_decomposition[i]  $\neq$  last element of chain_decomposition[i]
   then
30:    return null
31:   end if
32: end for
33: return chain_decomposition
```

---

## 4.8 Code Explanation

First we run a dfs to construct a dfs tree of the graph in which tree edges are directed towards the root and non tree edges are directed away from the root. If not all nodes are visited in the dfs, then we return No.

Then we run chain-decomposition on the graph which starts with the earliest non-tree edge in the graph. We obtain all the chains and check how many of those are cycles. If the number of cycles is  $> 1$ , then we return No. If any of the edges don't get visited or traversed during the chain decomposition, then we return No. Else we return Yes.

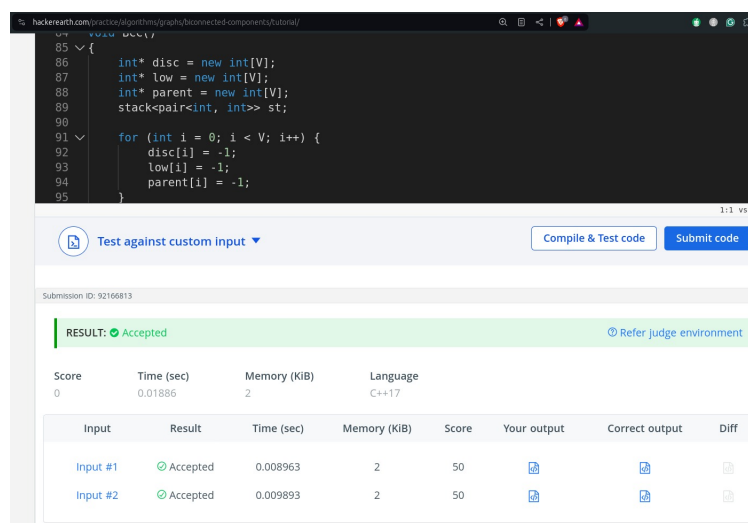
## 5 System Specifications

All the computations were done on one of our laptops with the specifications mentioned below. (The Runtimes are dependent on the specifications of the system)

- CPU : AMD Ryzen 9 5900HX (16 Core CPU)
- RAM : 16GB
- Cache : 3 level
  - L0 Cache : 512KB
  - L1 Cache : 4MB
  - L2 Cache : 16MB
- GPU : Nvidia GeForce RTX 3060 6GB Graphics

## 6 Code correctness

To test if the codes written by us is correct or not, we submitted the Tarjan's algorithm on a Hackerearth problem and got accepted there.



The screenshot shows a Hackerearth submission interface. At the top, there's a code editor with C++ code for Tarjan's algorithm. Below the code editor, there are buttons for "Test against custom input", "Compile & Test code", and "Submit code". The submission ID is 92166813. The result is "Accepted". Below the result, there's a table showing the score, time, memory, and language for the submission. The score is 0, time is 0.01886 seconds, memory is 2 KIB, and the language is C++17. Below this, there's a table showing the results for two test cases, both of which were accepted.

Input	Result	Time (sec)	Memory (KIB)	Score	Your output	Correct output	Diff
Input #1	Accepted	0.008963	2	50			
Input #2	Accepted	0.009893	2	50			



Now, to check Jen-Schmidt algorithm, we ran both Jen-schmidt algorithm and Tarjan algorithm against a generator and checked if both are having the same output or not. For doing this we used a tool called BugFinder and we got no output difference.

```
Run 48.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
Perfect Run. No error encountered.

Run 49.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
Perfect Run. No error encountered.

Run 50.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
Perfect Run. No error encountered.

Run 51.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
Perfect Run. No error encountered.

Run 52.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
Perfect Run. No error encountered.

Run 53.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
Perfect Run. No error encountered.

Run 54.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
Perfect Run. No error encountered.

Run 55.....
[./home/sumit_kk10/Desktop/Code/generator.exe]
[./home/sumit_kk10/Desktop/Code/jen-schmidt.exe]
[./home/sumit_kk10/Desktop/Code/tarjan.exe]
```

After checking for correctness of the code, we removed the print statements(as the print statements can have a significant impact on the runtimes) and just printed how much time it takes for the algorithm to run.

## 7 Experiments

Now for the experimentation, we are printing how much time each code takes as output of the code.

We have made a custom generator which generates a graph given  $n$  and  $m$ . Now, we have made a script which for different values of  $n$  [10, 100, 1000, 10000, 100000, 1000000] and 20 values of  $m$  from  $[0, (n * (n - 1))/2]$ , plots the graph.

For  $n = 100000, 1000000$ , we have limited the number of edges from  $[0, 10000]$ . So in this we are handling and checking for sparse graphs.

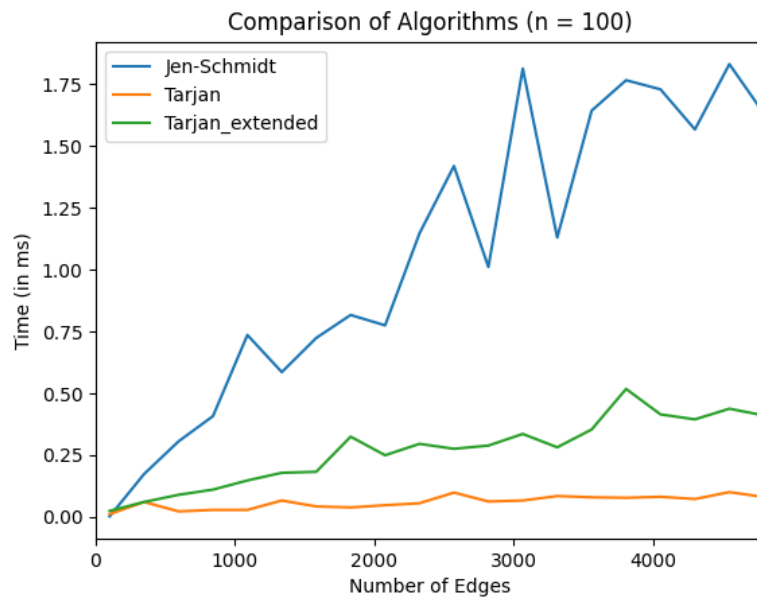
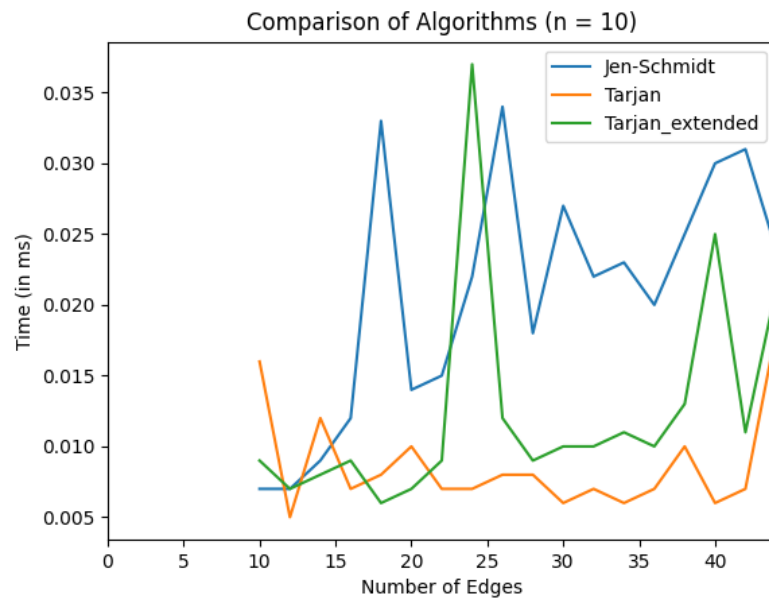
For  $n = 10, 100, 1000, 10000$ , we are keeping different values of  $m$  between  $[0, (n * (n - 1)) / 2]$ , which accounts for very dense graphs too.

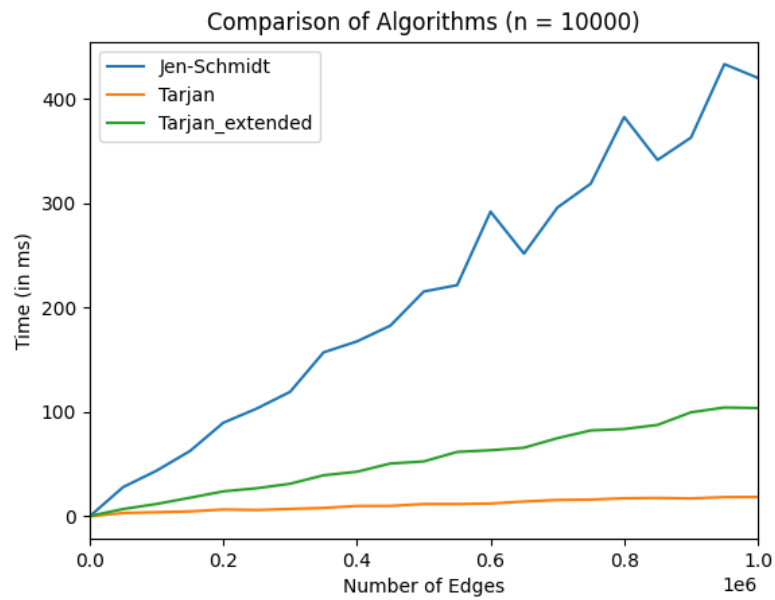
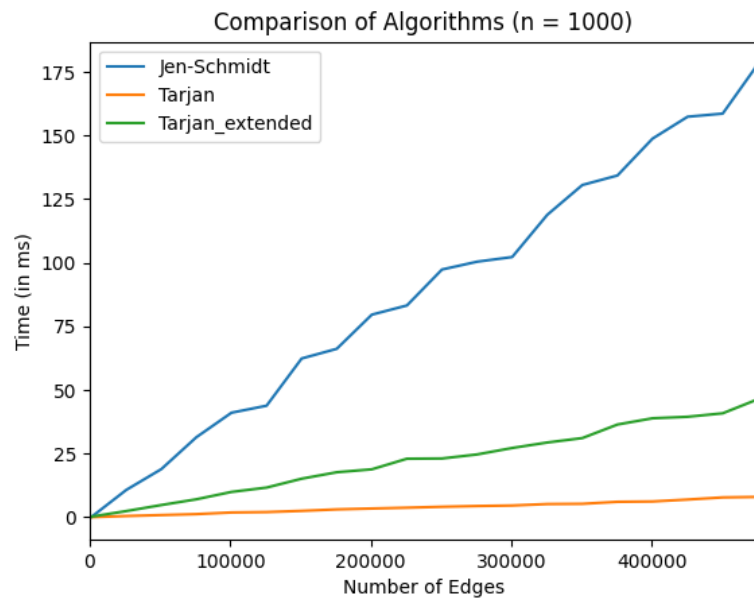
## 8 Comparative Study

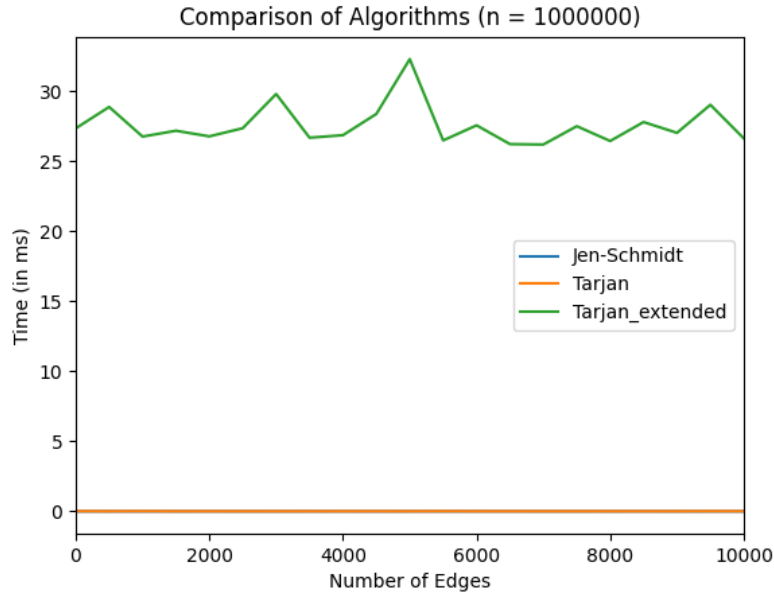
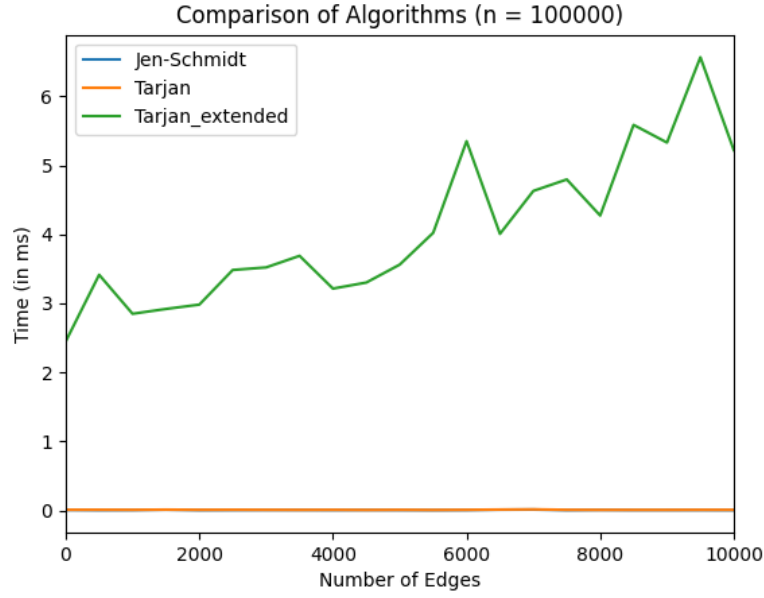
These are the runtimes which we got for some of the testcases on which we ran:

$n$	$m$	Jen Schmidt (in ms)	Tarjan (in ms)	Tarjan-Extended (in ms)
10	10	0.007	0.016	0.009
10	20	0.014	0.010	0.007
10	40	0.030	0.006	0.025
100	100	0.003	0.012	0.025
100	1088	0.736	0.029	0.148
100	2076	0.775	0.048	0.250
100	4052	1.729	0.082	0.415
100	4793	1.632	0.081	0.411
1000	1000	0.002	0.005	0.198
1000	125875	43.818	2.012	11.633
1000	300700	102.246	4.586	27.195
1000	400600	148.776	6.176	38.871
1000	475525	177.999	7.964	46.341
10000	50000	28.033	3.168	7.002
10000	150000	62.598	4.725	17.808
10000	450000	182.826	9.992	50.620
10000	650000	251.908	14.189	65.762
10000	1000000	420.418	18.555	103.792
100000	500	0.002	0.008	3.414
100000	3000	0.004	0.007	3.521
100000	6000	0.003	0.007	5.351
100000	10000	0.002	0.007	5.222
1000000	1000	0.004	0.018	26.742
1000000	3500	0.005	0.007	26.661
1000000	7000	0.004	0.008	26.176
1000000	10000	0.004	0.012	26.611

Table 1: Performance Comparison of Jen Schmidt, Tarjan, and Tarjan-Extended Algorithms







## 9 Conclusion

A general conclusion can be made that Tarjan's algorithm can be preferred over Jen-Schmidt on dense graphs with edges much greater than the number of vertices. On the other hand, for sparse graphs with edges much smaller than that of number of vertices, Jen-Schmidt performs much better than the case of Tarjan's. This is supported by similar results in the plots. General observations:

- For  $n \leq 10000$ , the algorithms are tested on sparse to dense graphs. As the number

of edges increases for some fixed  $n$ , time for Jen-Schmidt increases by a large amount as compared to Tarjan. Some abnormalities are observed in the case of  $n=10$ , as the times are very low and are dependent on the specific input of the graph. As we increase number of vertices, we get a general pattern/trend in the times. Tarjan extended takes more time than Tarjan as finding all biconnected components and printing them is more time consuming. While Jen-Schmidt even takes much more time than that of Tarjan-extended.

- For  $n > 10000$ , the algorithms are tested on sparse graphs. The times of both Tarjan and Jen-Schmidt are much lower than that of Tarjan-extended(as most of the sparse graphs are not biconnected, so Tarjan and Jen-Schmidt prune most of the steps of computation while Tarjan extended iterates over all the vertices making the time higher). For the case of Tarjan extended the times are comparable for all values of  $m$  up to 10000, as the major time complexity is because of the number of vertices here. For the case of Tarjan, the algorithm stops on finding one articulation points and answers "No". Similarly in the case of Jen-Schmidt due to very less number of edges, very less number of ears would be formed. This would lead to very small computation over the edges. As the edges are less in number to vertices, this computation is very negligible in comparison to that of Tarjan-extended.