**Tarjan’s Algorithm**

**Iheb Gafsi**\*

INSAT Student

**Iheb.engineer@gmail.com**

**Definition:**

Tarjan's algorithm is a popular graph traversal algorithm used for finding strongly connected components in directed graphs. Developed by Robert Tarjan in 1972, this algorithm efficiently determines sets of vertices that have mutual paths between them.

The core idea behind Tarjan's algorithm is to perform a depth-first search (DFS) on the graph while maintaining information about the depth of each vertex and the earliest reachable ancestor of each vertex. Through this process, the algorithm can identify and group together vertices that form strongly connected components. Strongly connected components are essential for various graph-related problems, such as analyzing network structures, identifying cycles, and solving certain optimization tasks. Tarjan's algorithm achieves a linear runtime complexity of ), making it a powerful tool for graph analysis.

**Use cases:**

Tarjan's algorithm finds extensive applications in the domain of computer science and graph theory due to its ability to identify strongly connected components efficiently. One of its primary use cases is in identifying cycles within directed graphs, which is crucial for detecting potential deadlocks and analyzing iterative processes in various systems. Additionally, Tarjan's algorithm is employed in network analysis to identify groups of closely connected nodes or communities, facilitating efficient information flow and network optimization. This algorithm also plays a vital role in identifying critical points, such as articulation points and bridges, in undirected graphs. By pinpointing these critical points, it becomes possible to evaluate the robustness and resilience of complex systems. In summary, Tarjan's algorithm is a versatile and powerful tool for analyzing graphs and networks, enabling researchers and engineers to gain valuable insights into the underlying structures and connections present in diverse datasets and systems.

**Algorithm:**

Begin by marking each node's ID as unvisited.

Initiate a Depth-First Search (DFS) traversal. During traversal, assign an ID and a low-link value to each visited node. Additionally, keep track of visited nodes by adding them to a seen stack.

During the callback phase of DFS, check if the previous node exists in the stack. If it does, update the current node's low-link value to the minimum between its current value and the low-link value of the previous node.

After exploring all neighbors of a node, check if the current node initiated a connected component. If it did, remove nodes from the stack until the current node is reached, effectively identifying and extracting the strongly connected component.

1. #Variables

2. graph = Adjacency List

3. #Global Variables and constants

4. UNVISITED = -1

5.

6. class Tarjan:

7.     def findSccs(self, graph):

8.         self.n = len(graph)

9.         self.sccs =0

10.         self.id = 0

11.         self.ids = [UNVISITED] \* self.n

12.         self.low\_links = [0] \* self.n

13.         self.stacked = [False] \* self.n

14.         self.stack = []

15.

16.         for i in range(self.n):

17.             if self.ids[i] == UNVISITED:

18.                 self.dfs(i, graph)

19.         return self.low\_links

20.

21.     def dfs(self, i, graph):

22.         self.stack.append(i)

23.         self.stacked[i] = True

24.         self.ids[i] = self.low\_links[i] = self.id

25.         self.id += 1

26.         for arg in graph[i]:

27.             if self.ids[i] == UNVISITED: self.dfs(arg)

28.             if self.stacked[arg]: self.low\_links[i] = min(self.low\_links[arg],

29.                                                            self.low\_links[i])

30.         if self.ids[i] == self.low\_links[i]:

31.             node = self.stack.pop()

32.             while not node == i:

33.                 self.stacked[node] = False

34.                 self.low\_links[node] = self.ids[i]

35.                 node = self.stack.pop()

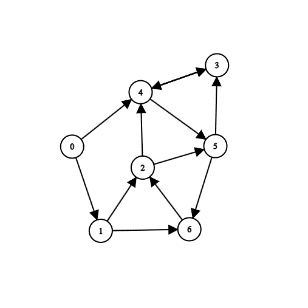
36.             self.sccs += 1

37.

38. print(Tarjan().findSccs(graph))

**Example:**

Here’s a small example illustrating an example of input outputs for the Tarjan’s Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

1. #Variables

2. graph =[

3.     [1, 4],

4.     [2, 6],

5.     [4, 5],

6.     [4],

7.     [5, 3],

8.     [3, 6],

9.     [2]

10. ]

11. #Global Variables and constants

12. UNVISITED = -1

13.

14. class Tarjan:

15.     def findSccs(self, graph):

16.         self.n = len(graph)

17.         self.sccs =0

18.         self.id = 0

19.         self.ids = [UNVISITED] \* self.n

20.         self.low\_links = [0] \* self.n

21.         self.stacked = [False] \* self.n

22.         self.stack = []

23.

24.         for i in range(self.n):

25.             if self.ids[i] == UNVISITED:

26.                 self.dfs(i, graph)

27.         return self.low\_links

28.

29.     def dfs(self, i, graph):

30.         self.stack.append(i)

31.         self.stacked[i] = True

32.         self.ids[i] = self.low\_links[i] = self.id

33.         self.id += 1

34.         for arg in graph[i]:

35.             if self.ids[i] == UNVISITED: self.dfs(arg)

36.             if self.stacked[arg]: self.low\_links[i] = min(self.low\_links[arg],

37.                                                            self.low\_links[i])

38.         if self.ids[i] == self.low\_links[i]:

39.             node = self.stack.pop()

40.             while not node == i:

41.                 self.stacked[node] = False

42.                 self.low\_links[node] = self.ids[i]

43.                 node = self.stack.pop()

44.             self.sccs += 1

45.

46. print(Tarjan().findSccs(graph))

The corresponding output is:

Python >> [0, 1, 2, 3, 3, 3, 2]

