Data Structures and Algorithms

Lecture 38

Aniket Basu Roy

2026-04-29 Wed

Contents

1 Agenda

1.1 Tarjan's Algorithm for Strongly Connected Components

- SCCs recap

- DFS-based approach: discovery index and low-link value
- Pseudocode
- Worked example
- Correctness (informal)
- Time complexity

2 Strongly Connected Components: Recap

A **strongly connected component (SCC)** of a directed graph $G = (V, E)$ is a maximal set $C \subseteq V$ such that for every pair $u, v \in C$ there is a directed path from u to v and from v to u.

Goal: partition V into its SCCs efficiently.

Tarjan's algorithm (1972) does this in $O(V + E)$ using a single depth-first search.

3 Key Idea

Recall that a DFS on a directed graph classifies edges as:

- **Tree edges:** (u, v) where v is first discovered from u.
- **Back edges:** (u, v) where v is an ancestor of u in the current DFS path.
- **Forward/cross edges:** all other edges.

Observation. Each SCC is a subtree of the DFS tree (possibly extended by back edges returning to ancestors still on the current DFS path).

A vertex v is the **root** of its SCC in the DFS tree if no vertex in its DFS subtree can reach a proper ancestor of v (i.e., no back edge escapes the subtree).

4 Algorithm Data Structures

For each vertex v maintain:

- v.index: discovery time of v (global counter, starting at 1, incremented on each first visit).

- v.lowlink: smallest index value reachable from v's DFS subtree via at most one back edge to an ancestor still on the auxiliary stack.
- v.onStack: true if v is currently on the auxiliary stack S.

A global stack S holds all vertices visited but not yet assigned to a completed SCC.

4.1 Lowlink Update Rules

Initialize v.lowlink $\leftarrow$ v.index when v is first visited.

After exploring each edge (v, w):

- **Tree edge** (w unvisited): recurse into w; then v.lowlink $\leftarrow \min(v.\text{lowlink}, w.\text{lowlink})$.
- **Back edge to ancestor on stack** (w.onStack = true): v.lowlink $\leftarrow \min(v.\text{lowlink}, w.\text{index})$. (Use w.index, not w.lowlink – this is deliberate and from the original paper.)
- **Edge to completed SCC** (w.onStack = false, already finished): do nothing.

4.2 Root Condition

After all edges from v are explored:

$$v.\text{lowlink} = v.\text{index} \implies v \text{ is the root of an SCC.}$$

Pop the stack down to and including v; the popped vertices form one SCC.

5 Pseudocode

```
TARJAN-SCC(G)
1.  index_counter = 1
2.  S = empty stack
3.  for each v in G.V
4.      v.index   = undefined
5.      v.onStack = false
6.  for each v in G.V
7.      if v.index is undefined
8.          STRONGCONNECT(G, v)
```

```
STRONGCONNECT(G, v)
 1. v.index   = index_counter
 2. v.lowlink = index_counter
 3. index_counter = index_counter + 1
 4. push v onto S
 5. v.onStack = true
 6. for each edge (v, w) in G.E
 7.     if w.index is undefined              // tree edge
 8.         STRONGCONNECT(G, w)
 9.         v.lowlink = min(v.lowlink, w.lowlink)
10.     else if w.onStack                    // back edge to ancestor on stack
11.         v.lowlink = min(v.lowlink, w.index)
12. if v.lowlink == v.index                 // v is root of an SCC
13.     SCC = empty set
14.     repeat
15.         w = pop from S
16.         w.onStack = false
17.         add w to SCC
18.     until w == v
19.     output SCC
```

6 Worked Example

```
V = {1, 2, 3, 4},   E: 1->2, 2->3, 3->1, 3->4

1 --> 2 --> 3 --> 4
^           |
+-----------+
```

DFS starts from vertex 1. Edges are explored left-to-right.

Event	vertex	index	lowlink	Stack S
Visit 1	1	1	1	[1]
Visit 2 (tree edge $1 \to 2$)	2	2	2	[1, 2]
Visit 3 (tree edge $2 \to 3$)	3	3	3	[1, 2, 3]
Back edge $3 \to 1$; update lowlink	3	3	1	[1, 2, 3]
Visit 4 (tree edge $3 \to 4$)	4	4	4	[1, 2, 3, 4]
4 is root ($4 = 4$); pop $\{4\}$	4	4	4	[1, 2, 3]
3 done; lowlink $\min(1, 4) = 1$	3	3	1	[1, 2, 3]
2 done; lowlink $\min(2, 1) = 1$	2	2	1	[1, 2, 3]
1 done; lowlink $\min(1, 1) = 1$	1	1	1	[1, 2, 3]
1 is root ($1 = 1$); pop $\{1, 2, 3\}$	1	1	1	[]

6.1 Output

- SCC $\{4\}$ (trivial; vertex 4 has no outgoing edges).
- SCC $\{1, 2, 3\}$ (the 3-cycle).

Condensation DAG: $\{1, 2, 3\} \to \{4\}$.

Note: Tarjan's algorithm outputs SCCs in **reverse topological order** of the condensation.

7 Correctness (Informal)

Two key claims establish correctness.

1. **A vertex v is assigned to an SCC only after its entire SCC has been explored.** Any vertex w reachable from v and able to reach back an ancestor of v is in the same SCC as v. Such a w lowers v.lowlink below v.index, preventing premature SCC output.
2. **When v.lowlink = v.index, all vertices above v on the stack belong to v's SCC.** No vertex above v can lower its own lowlink below v.index, because any path escaping back further would have already lowered v.lowlink too.

8 Time Complexity

Each vertex is pushed onto S once and popped once. Each edge (v, w) is examined once inside STRONGCONNECT.

Operation	Total cost
Push/pop each vertex	$O(V)$
Examine each edge	$O(E)$

$$\boxed{T_{\text{Tarjan}} = O(V + E).}$$

This matches the cost of a single DFS.

9 Questions

- In the worked example, draw the condensation DAG and verify that the output order of SCCs is a valid reverse topological ordering of it.
- Why do we update v.lowlink with w.index (not w.lowlink) for the back-edge case?
- Explain why vertices that are already in a completed SCC (w.onStack = false) are ignored when updating lowlink.