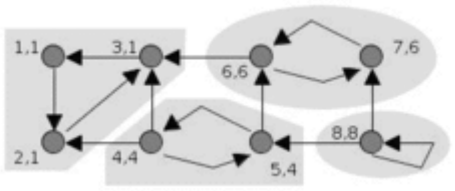
WIKIPEDIA
The Free Encyclopedia

Tarjan's strongly connected components algorithm

Tarjan's strongly connected components algorithm is an algorithm in graph theory for finding the strongly connected components (SCCs) of a directed graph. It runs in linear time, matching the time bound for alternative methods including Kosaraju's algorithm and the path-based strong component ?? algorithm. The algorithm is named for its inventor, Robert Tarjan.[1]

See CPH chapter 17

Tarjan's strongly connected components algorithm



Tarjan's algorithm animation

Data structure	Graph
Worst-case performance	$O(\|V\|+\|E\|)$

Overview

The algorithm takes a directed graph as input, and produces a partition of the graph's vertices into the graph's strongly connected components. Each vertex of the graph appears in exactly one of the strongly connected components. Any vertex that is not on a directed cycle forms a strongly connected component all by itself: for example, a vertex whose in-degree or out-degree is 0, or any vertex of an acyclic graph.

The basic idea of the algorithm is this: a depth-first search (DFS) begins from an arbitrary start node (and subsequent depth-first searches are conducted on any nodes that have not yet been found). As usual with depth-first search, the search visits every node of the graph exactly once, declining to revisit any node that has already been visited. Thus, the collection of search trees is a spanning forest of the graph. The strongly connected components will be recovered as certain subtrees of this forest. The roots of these subtrees are called the "roots" of the strongly connected components. Any node of a strongly connected component might serve as a root, if it happens to be the first node of a component that is discovered by search.

Stack invariant

Nodes are placed on a stack in the order in which they are visited. When the depth-first search recursively visits a node v and its descendants, those nodes are not all necessarily popped from the stack when this recursive call returns. The crucial invariant property is that a node remains on the stack after it has been visited if and only if there exists a path in the input graph from it to some node earlier on the stack. In other words, it means that in the DFS a node would be only removed from the stack after all its connected paths have been traversed. When the DFS will backtrack it would remove the nodes on a single path and return to the root in order to start a new path.

At the end of the call that visits v and its descendants, we know whether v itself has a path to any node earlier on the stack. If so, the call returns, leaving v on the stack to preserve the invariant. If not, then v must be the root of its strongly connected component, which consists of v together with any nodes later on the stack than `v` (such nodes all have paths back to `v` but not to any earlier node, because if they had paths to earlier nodes then `v` would also have paths to earlier nodes which is false). The connected component rooted at `v` is then popped from the stack and returned, again preserving the invariant.

Bookkeeping

Each node `v` is assigned a unique integer `v.index`, which numbers the nodes consecutively in the order in which they are discovered. It also maintains a value `v.lowlink` that represents the smallest index of any node on the stack known to be reachable from `v` through `v`'s DFS subtree, including `v` itself. Therefore `v` must be left on the stack if `v.lowlink < v.index`, whereas v must be removed as the root of a strongly connected component if `v.lowlink == v.index`. The value `v.lowlink` is computed during the depth-first search from `v`, as this finds the nodes that are reachable from `v`.

The lowlink is different from the lowpoint, which is the smallest index reachable from `v` through any part of the graph.[1]:156[2]

The algorithm in pseudocode

```
algorithm tarjan is
    input: graph G = (V, E)
    output: set of strongly connected components (sets of vertices)

    index := 0
    S := empty stack
    for each v in V do
        if v.index is undefined then
            strongconnect(v)

    function strongconnect(v)
        // Set the depth index for v to the smallest unused index
        v.index := index
        v.lowlink := index
        index := index + 1
        S.push(v)
        v.onStack := true

        // Consider successors of v
        for each (v, w) in E do
            if w.index is undefined then
                // Successor w has not yet been visited; recurse on it
                strongconnect(w)
                v.lowlink := min(v.lowlink, w.lowlink)
            else if w.onStack then
                // Successor w is in stack S and hence in the current SCC
                // If w is not on stack, then (v, w) is an edge pointing to an SCC already
found and must be ignored
                // The next line may look odd - but is correct.
                // It says w.index not w.lowlink; that is deliberate and from the original
paper
                v.lowlink := min(v.lowlink, w.index)

        // If v is a root node, pop the stack and generate an SCC
        if v.lowlink = v.index then
            start a new strongly connected component
            repeat
                w := S.pop()
                w.onStack := false
                add w to current strongly connected component
            while w ≠ v
            output the current strongly connected component
```

The `index` variable is the depth-first search node number counter. `S` is the node stack, which starts out empty and stores the history of nodes explored but not yet committed to a strongly connected component. This is not the normal depth-first search stack, as nodes are not popped as the search returns up the tree;

they are only popped when an entire strongly connected component has been found.

The outermost loop searches each node that has not yet been visited, ensuring that nodes which are not reachable from the first node are still eventually traversed. The function `strongconnect` performs a single depth-first search of the graph, finding all successors from the node `v`, and reporting all strongly connected components of that subgraph.

When each node finishes recursing, if its lowlink is still set to its index, then it is the root node of a strongly connected component, formed by all of the nodes above it on the stack. The algorithm pops the stack up to and including the current node, and presents all of these nodes as a strongly connected component.

`v.lowlink := min(v.lowlink, w.index)` is the correct way to update `v.lowlink` if `w` is on stack. Because `w` is on the stack already, `(v, w)` is a back-edge in the DFS tree and therefore `w` is not in the subtree of `v`. Because `v.lowlink` takes into account nodes reachable only through the nodes in the subtree of `v` we must stop at `w` and use `w.index` instead of `w.lowlink`.

opposite of this

↳ If we use w.lowlink, then we are adding SCC of which w is part of to SCC of which v is part of.

Complexity

Time Complexity: The Tarjan procedure is called once for each node; the forall statement considers each edge at most once. The algorithm's running time is therefore linear in the number of edges and nodes in G, i.e. $O(|V| + |E|)$.

In order to achieve this complexity, the test for whether `w` is on the stack should be done in constant time. This may be done, for example, by storing a flag on each node that indicates whether it is on the stack, and performing this test by examining the flag.

BAKWAS !!!! . . . .

Space Complexity: The Tarjan procedure requires two words of supplementary data per vertex for the `index` and `lowlink` fields, along with one bit for `onStack` and another for determining when `index` is undefined. In addition, one word is required on each stack frame to hold `v` and another for the current position in the edge list. Finally, the worst-case size of the stack `S` must be $|V|$ (i.e. when the graph is one giant component). This gives a final analysis of $O(|V| \cdot (2 + 5w))$ where w is the machine word size. The variation of Nuutila and Soisalon-Soininen reduced this to $O(|V| \cdot (1 + 4w))$ and, subsequently, that of Pearce requires only $O(|V| \cdot (1 + 3w))$.[3][4]

Additional remarks

invariant !!! . . .

While there is nothing special about the order of the nodes within each strongly connected component, one useful property of the algorithm is that no strongly connected component will be identified before any of its successors. Therefore, the order in which the strongly connected components are identified constitutes a reverse topological sort of the DAG formed by the strongly connected components.[5]

Donald Knuth described Tarjan's SCC algorithm as one of his favorite implementations in the book *The Stanford GraphBase*.[6]

He also wrote:[7]

> The data structures that he devised for this problem fit together in an amazingly beautiful way, so that the quantities you need to look at while exploring a directed graph are always magically at your fingertips. And his algorithm also does topological sorting as a byproduct.

References

1. Tarjan, R. E. (1972), "Depth-first search and linear graph algorithms", *SIAM Journal on Computing*, **1** (2): 146–160, CiteSeerX 10.1.1.327.8418 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.327.8418), doi:10.1137/0201010 (https://doi.org/10.1137%2F0201010)
2. "Lecture #19: Depth First Search and Strong Components" (https://www.cs.cmu.edu/~15451-f18/lectures/lec19-DFS-strong-components.pdf) (PDF). *15-451/651: Algorithms Fall 2018*. Carnegie Mellon University. pp. 7–8. Retrieved 9 August 2021.
3. Nuutila, Esko (1994). "On Finding the Strongly Connected Components in a Directed Graph". *Information Processing Letters*. **49** (1): 9–14. doi:10.1016/0020-0190(94)90047-7 (https://doi.org/10.1016%2F0020-0190%2894%2990047-7).
4. Pearce, David. "A Space Efficient Algorithm for Detecting Strongly Connected Components". *Information Processing Letters*. **116** (1): 47–52. doi:10.1016/j.ipl.2015.08.010 (https://doi.org/10.1016%2Fj.ipl.2015.08.010).
5. Harrison, Paul. "Robust topological sorting and Tarjan's algorithm in Python" (http://www.logarithmic.net/pfh/blog/01208083168). Retrieved 9 February 2011.
6. Knuth, *The Stanford GraphBase*, pages 512–519.
7. Knuth, Donald (2014-05-20). *Twenty Questions for Donald Knuth* (http://www.informit.com/articles/article.aspx?p=2213858&WT.mc_id=Author_Knuth_20Questions).

External links

- Rosetta Code (https://rosettacode.org/wiki/Tarjan), showing implementations in different languages
- PHP implementation of Tarjan's strongly connected components algorithm (https://github.com/Vacilando/php-tarjan)
- JavaScript implementation of Tarjan's strongly connected components algorithm (https://github.com/Vacilando/js-tarjan)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Tarjan%27s_strongly_connected_components_algorithm&oldid=1179692125"

-