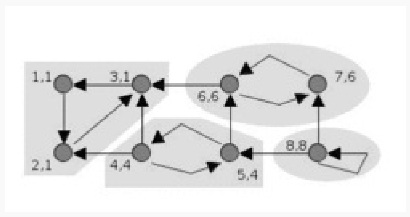
Tarjan's strongly connected components algorithm

From Wikipedia, the free encyclopedia

Tarjan's Algorithm is an algorithm in graph theory for finding the strongly connected components of a graph. Although proposed earlier, it can be seen as an improved version of Kosaraju's algorithm, and is comparable in efficiency to the path-based strong component algorithm. Tarjan's Algorithm is named for its discoverer, Robert Tarjan.[1]

Tarjan's strongly connected components algorithm



Tarjan's algorithm animation

Data structure	Graph
Worst case performance	$O(\lvert V\rvert + \lvert E\rvert)$

Contents [hide]

Overview [edit]

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 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 explored. 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 the root, if it happens to be the first node of the component that is discovered by the search.

Stack invariant [edit]

The nodes are placed on a stack in the order in which they are visited. When the depth-first search recursively explores a node `v` and its descendants, those nodes are not all necessarily popped from the stack before this recursive call returns. The crucial invariant property is that a node remains on the stack after exploration if and only if it has a path to some node earlier on the stack.

At the end of the call that explores `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 later nodes on the stack (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). This entire component is then popped from the stack and returned, again preserving the invariant.

Bookkeeping [edit]

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 (roughly speaking) the smallest index of any node known to be reachable from `v`, 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 algorithm in pseudocode [edit]

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

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

  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
        v.lowlink  := min(v.lowlink, w.index)
      end if
    end for

    // 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
      until (w = v)
      output the current strongly connected component
    end if
  end function
```

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. Note that 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.

Remarks [edit]

1. Complexity: The Tarjan procedure is called once for each node; the forall statement considers each edge at most twice. The algorithm's running time is therefore linear in the number of edges and nodes in G, i.e. $O(|V|+|E|)$.
2. The test for whether `w` is on the stack should be done in constant time, for example, by testing a flag stored on each node that indicates whether it is on the stack.
3. 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.[2]

4. Tarjan's algorithm was mentioned as one of his favorite implementations by Knuth appearing in his book The Stanford GraphBase, pages 512–519. He considered this as one of the most beautiful algorithms with a quote [3]

> 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 [edit]

1. ^ Tarjan, R. E. (1972), "Depth-first search and linear graph algorithms", *SIAM Journal on Computing* **1** (2): 146–160, doi:10.1137/0201010
2. ^ Harrison, Paul. "Robust topological sorting and Tarjan's algorithm in Python". Retrieved 9 February 2011.
3. ^ Harrison, Knuth. "Twenty Questions for Donald Knuth".

External links [edit]

- Implementation of Tarjan's Algorithm in .NET
- Implementation of Tarjan's Algorithm in .NET (GitHub)
- Implementation of Tarjan's Algorithm in PHP
- Another implementation of Tarjan's Algorithm in Python
- Implementation of Tarjan's Algorithm in Javascript
- Implementation of Tarjan's Algorithm in Clojure
- Implementation of Tarjan's Algorithm in C++

Categories: Graph algorithms | Graph connectivity