WIKIPEDIA

Kosaraju's algorithm

In computer science, **Kosaraju-Sharir's algorithm** (also known as **Kosaraju's algorithm**) is a linear time algorithm to find the strongly connected components of a directed graph. Aho, Hopcroft and Ullman credit it to S. Rao Kosaraju and Micha Sharir. Kosaraju suggested it in 1978 but did not publish it, while Sharir independently discovered it and published it in 1981. It makes use of the fact that the transpose graph (the same graph with the direction of every edge reversed) has exactly the same strongly connected components as the original graph.

Contents

The algorithm

The primitive graph operations that the algorithm uses are to enumerate the vertices of the graph, to store data per vertex (if not in the graph data structure itself, then in some table that can use vertices as indices), to enumerate the out-neighbours of a vertex (traverse edges in the forward direction), and to enumerate the in-neighbours of a vertex (traverse edges in the backward direction); however the last can be done without, at the price of constructing a representation of the transpose graph during the forward traversal phase. The only additional data structure needed by the algorithm is an ordered list L of graph vertices, that will grow to contain each vertex once.

If strong components are to be represented by appointing a separate root vertex for each component, and assigning to each vertex the root vertex of its component, then Kosaraju's algorithm can be stated as follows.

1. For each vertex u of the graph, mark u as unvisited. Let L be empty.
2. For each vertex u of the graph do `Visit(u)`, where `Visit(u)` is the recursive subroutine:

    If u is unvisited then:

    1. Mark u as visited.
    2. For each out-neighbour v of u, if v is unvisited, do `Visit(v)`.
    3. Prepend u to L.

    Otherwise do nothing.

3. For each element u of L in order, do `Assign(u,u)` where `Assign(u,root)` is the recursive subroutine:

    If u has not been assigned to a component then:

    1. Assign u as belonging to the component whose root is *root*.
    2. For each in-neighbour v of u, do `Assign(v,root)`.

    Otherwise do nothing.

Trivial variations are to instead assign a component number to each vertex, or to construct per-component lists of the vertices that belong to it. The unvisited/visited indication may share storage location with the final assignment of root for a vertex.

The key point of the algorithm is that during the first (forward) traversal of the graph edges, vertices are prepended to the list L in post-order relative to the search tree being explored. This means it does not matter whether a vertex *v* was first visited because it appeared in the enumeration of all vertices or because it was the out-neighbour of another vertex *u* that got visited; either way *v* will be prepended to L before *u* is, so if there is a forward path from *u* to *v* then *u* will appear before *v* on the final list L (unless *u* and *v* both belong to the same strong component, in which case their relative order in L is arbitrary).

This means, that each element *n* of the list can be made to correspond to a block L[i_{n-1}: i_n], where the block consists of all the vertices reachable from vertex *n* using just outward edges at each node in the path. It is important to note that no vertex in the block beginning at *n* has an inward link from any of the blocks beginning at some vertex to its right, i.e., the blocks corresponding to vertices i_n, i_{n+1}, … *N* in the list. This is so, because otherwise the vertex having the inward link(say from the block beginning at $n' \geq i_{n+1}$)would have been already visited and pre-pended to L in the block of *n'*, which is a contradiction. On the other hand, vertices in the block starting at *n* **can have** edges pointing to the blocks starting at some vertex in $\{i_n, i_{n+1}, \dots N\}$.

Step 3 of the algorithm, starts from L[0], assigns all vertices which point to it, the same component as L[0]. Note that these vertices can only lie in the block beginning at L[0] as higher blocks can't have links pointing to vertices in the block of L[0]. Let the set of all vertices that point to L[0] be In(L[0]). Subsequently, all the vertices pointing to these vertices, In(In(L[0])) are added too, and so on till no more vertices can be added.

There is a path to L[0], from all the vertices added to the component containing L[0]. And there is a path to all the vertices added from L[0], as all those lie in the block beginning at L[0](which contains all the vertices reachable from L[0] following outward edges at each step of path). Hence all these form a single strongly connected component. Moreover, no vertex remains, because, to be in this strongly connected component a vertex must be reachable from L[0] and must be able to reach L[0]. All vertices that are able to reach L[0], if any, lie in the first block only, and all the vertices in first block are reachable from L[0]. So the algorithm chooses all the vertices in the connected component of L[0].

When we reach vertex v = L[*i*], in the loop of step 3, and *v* hasn't been assigned to any component, we can be sure that all the vertices to the left have made their connected components; that *v* doesn't belong to any of those components; that *v* doesn't point to any of the vertices to the left of it. Also, since, no edge from higher blocks to *v*'s block exists, the proof remains same.

As given above, the algorithm for simplicity employs depth-first search, but it could just as well use breadth-first search as long as the post-order property is preserved.

The algorithm can be understood as identifying the strong component of a vertex *u* as the set of vertices which are reachable from *u* both by backwards and forwards traversal. Writing $F(u)$ for the set of vertices reachable from *u* by forward traversal, $B(u)$ for the set of vertices reachable from *u* by backwards traversal, and $P(u)$ for the set of vertices which appear strictly before *u* on the list L after phase 2 of the algorithm, the strong component containing a vertex *u* appointed as root is

$$B(u) \cap F(u) = B(u) \setminus (B(u) \setminus F(u)) = B(u) \setminus P(u).$$

Set intersection is computationally costly, but it is logically equivalent to a double set difference, and since $B(u) \setminus F(u) \subseteq P(u)$ it becomes sufficient to test whether a newly encountered element of $B(u)$ has already been assigned to a component or not.

Complexity

Provided the graph is described using an adjacency list, Kosaraju's algorithm performs two complete traversals of the graph and so runs in Θ(V+E) (linear) time, which is asymptotically optimal because there is a matching lower bound (any algorithm must examine all vertices and edges). It is the conceptually simplest efficient algorithm, but is not as efficient in practice as Tarjan's strongly connected components algorithm and the path-based strong component algorithm, which perform only one traversal of the graph.

If the graph is represented as an adjacency matrix, the algorithm requires $O(V^2)$ time.

References

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, 3rd edition. The MIT Press, 2009. ISBN 0-262-03384-4.
- Micha Sharir. A strong-connectivity algorithm and its applications to data flow analysis. *Computers and Mathematics with Applications* 7(1):67–72, 1981.

External links

- Good Math, Bad Math: Computing Strongly Connected Components (http://scienceblogs.com/goodmath/2007/10/30/computing-strongly-connected-c/)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Kosaraju%27s_algorithm&oldid=1095805060"

This page was last edited on 30 June 2022, at 13:42 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License 3.0; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.