Graph algorithms - Strongly connected components

The Kosaraju algorithm

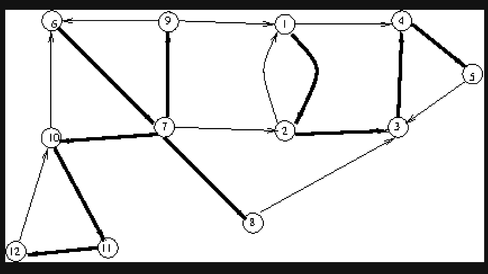
```
Input:
    G : directed graph
Output:
    comp : a map that associates, to each vertex, the ID of its strongly connected component

Subalgorithm DF1(Graph G, vertex x, Set& visited, Stack& processed)
    for y in Nout(x) do
        if y not in visited then
            visited.add(y)
            DF1(y)
        end if
    end for
    processed.push(x)

Algorithm:
    Stack processed
    Set visited

    for s in X do
        if s not in visited then
            visited.add(s)
            DF1(G, s, visited, processed)

    visited.clear()
    Queue q
    int c = 0
    while not processed.isEmpty() do
        s = processed.pop()
        if s not in visited then
            c = c + 1
            comp[s] = c
            q.enqueue(s)
            visited.add(s)
            while not q.isEmpty() do
                x = q.dequeue()
                for y in Nin(x) do
                    if y not in visited then
                        visited.add(y)
                        q.enqueue(y)
                        comp[y] = c
                    end if
                end for
            end while
        end if
    end while
```



Correctness

To clarify the terminology, at each time during the depth-first traversal, each vertex can be in one of 3 possible states:

- not visited yet (not yet in the *visited* set);
- on the execution stack (on the path from the current root to the current node);
- fully visited (and thus in the processed stack).

First, it is easy to show that a depth-first (DF) traversal started from a root visits all the vertices that are accessible from that root. It can ve shown by contradiction: assume a vertex that is accessible from the root, but never gets visited. Consider then a path from the root to that vertex; that path must have, at some point, a visited vertex followed by a non-visited vertex. But this means that, when that last visited vertex was visited, its successor must have been visited, too.

Next, for each SCC, we consider the *representative vertex* of the SCC the first vertex, from that SCC, to get (partially) visited. We claim two things:

- The representative of a SCC is the first vertex, from that SCC, to be visited, but the last to be fully processed and added to the stack. Consequently, it will also be the first to be taken out of the stack in the second phase.
- If SCC B is accessible from SCC A, then the representative of A is fully processed *after* the representative of B.

Indeed, let x be the representative of a SCC. At some point, x is processed, becoming effectively the root of DFS. Until it gets fully processed, all the vertices accessible from x are processed, unless at least a path to them goes through an already visited vertex. However, the path cannot go through any of the ancestors of x, because none is accessible from x, as x is the first vertex of its SCC that is processed. As for the fully visited vertices, all the vertices accessible from them are already fully processed. Therefore, all the vertices in the SCC of x are processed between the time x is first touched until the time x is finished.

.

Now consider the second phase. At each iteration of the main loop, a representative of a new SCC is picked up from the processed stack. Let's call x that vertex and A its SCC. Now, all the vertices from which x is accessible are either members of A or members of another SCC (let's call it Z) from which A is accessible. But, in the second case, the representative of Z must have been put in the processed stack after x and, therefore, it is already processed and its SCC retrieved.

On the reflexive-transitive closure and the reduced graph

The *reflexive-transitive closure* of a graph is the accessibility relation in that graph.

Given a graph, we can define the relation $x \sim y$ if x is accessible from y and y is accessible from x. Then, $\sim$ is an equivalence relation, and it defines a partitioning of the set of vertices in the graph. The parts are the strongly connected components.

Next, we can define an accessibility relation between components, by saying that component B is accessible from component A if there is a vertex in B that is accessible from a vertex in A. We immediately see that if there is a vertex in B that is accessible from a vertex in A, then each vertex in B is accessible from\ each vertex in A.

Finally, we can define a new graph in which the vertices are the SCC of the original graph and where we put an edge between two vertices if there is an edge between a vertex of the first component and a vertex of the second component. This is the *reduced graph* of the strogly connected components.

It is easy to see that, in the reduced graph, there are no cycles. If there was a cycle, the SCCs in that cycle would be a single SCC.

Tarjan's algorithm

Tarjan's algorithm is also based on performing a DFS in the graph, but it computes, for each vertex, the earliest ancestor vertex (closest to the root) that is directly reachable from that vertex or a descendent of that vertex in the DFS tree (this is called *lowlink*).

The SCC representatives are recognized by the fact that their lowlink is equal to themselves.

Actually retrieving the component is done as follows: a stack is maintained where a vertex is added when its processing starts. Then, when a vertex is recognized as SCC representative at the end of its processing, all the vertices up to and including the SCC representatives are poped out of the stack and marked as part of the SCC.

Radu-Lucian LUPŞA
2018-04-17