# Certifying Algorithm for Strongly Connected Components

**Tadej Borovšak**[1], **Jurij Mihelič**[2]

[1]*XLAB d.o.o., Pot za Brdom 100, Ljubljana, Slovenia*
[2]*Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, Slovenia*
*E-mail: jurij.mihelic@fri.uni-lj.si*

## Abstract

*In the paper we focus on algorithms whose output is the solution to the input instance as well as the certificate that the computed solution is indeed correct; such algorithms are called certifying. Our center of interest is the problem of decomposing a directed graph into its strongly connected components. Several non-certifying algorithms are already known for the problem. Our contribution is a certifying algorithm for the problem as well as the corresponding checker algorithm. Additionally, we prove that the checker correctly verifies the output of the certifying algorithm.*

## 1 Introduction

Algorithms for solving problems defined on graphs are fundamental in computer science. Many such problems exist, see for example any introductory textbook on algorithms [1, 2] or for particular examples of graph problems [3, 4, 5]. One of the elementary graph problems is determining a decomposition of a given directed graph into *strongly connected components*. Many algorithms on directed graphs may benefit from handling each component separately [2].

Usually the algorithms used in real-life applications must be shown correct. There are several approaches to this, e.g., intuitive understanding, testing, formal proofs, automatic verification, and *certifying algorithms*. In this paper we focus on the latter.

A certifying algorithm is an algorithm that produces, with each output, a *certificate* or *witness* (easy-to-verify proof) that the particular output has not been compromised by a bug [6, 7]. A user can convince herself that the output is correct by inspecting the witness.

The process of checking the witness may be automated with a *checker* algorithm. The checker is an algorithm, which automatically verifies using the witness that the output is correctly corresponding to the input.

In practice, formal analysis of correctness of the algorithm may be too involved, but analyzing the checker may be much easier. Additionally, such verification of the algorithm on a paper does not prove anything about its implementation; it may contain bugs. On the other side, testing the implementation with a limited test set may leave out many test cases and miss some important bugs as complete testing is in practice infeasible. Notice also, that automatic verification of programs is still in its infancy: with many successes but also with a very demanding implementation.

In this regard, certifying algorithms are somehow in-between, yet their implementation effort is basically indistinguishable from the non-certifying ones. Nevertheless, additional effort must be put into the implementation of the checker, and to formally prove that checking actually works.

## 2 Strongly connected components

### 2.1 Problem definition

A *strongly connected component* of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that every pair of vertices $u, v \in C$ are reachable from each other, i.e., there is a path in $G$ from $u$ to $v$ as well as from $v$ to $u$. Notice also, the component $C$ being maximal means that no vertex can be added to $C$ such that $C$ would still be strongly connected. (Clearly, the notion of being "maximal" differs from "maximum", i.e., there may be many (maximal) strongly connected components of various sizes, yet the maximum strongly connected components are the largest among them.)

A *partition* of a set $S$ is a family $\{P_1, P_2, \ldots, P_s\}$ of nonempty subsets of $S$ such that every element in $S$ is a member of exactly one of the subsets, i.e., $P_i \neq \emptyset$ and $P_i \subseteq S$ for all $1 \leq i \leq s$ as well as $\bigcup_{1 \leq i \leq s} P_i = S$. Additionally, the sets in a partition are pairwise disjoint, i.e., $P_i \cap P_j = \emptyset$ for all $1 \leq i, j \leq s$ with $i \neq j$.

Given a directed graph $G = (V, E)$, the goal of the strongly connected components problem is to find a partition of the vertices $V$, such that each set of the partition is strongly connected component.

### 2.2 Non-certifying algorithms

There are several polynomial-time algorithms for solving the strongly connected components problem exactly. The two well-known are Kosaraju's algorithm [1, 2, 8] and Tarjan's algorithm [9]. Both algorithms are an application of a depth-first search on a graph. Their running time is $\Theta(|V(G)| + |E(G)|)$ provided that an input graph is represented by an adjacency matrix.

Because our certifying algorithm for finding strongly connected components is based on Tarjan's algorithm, let

us present it here briefly. The pseudo-code of the (certifying) algorithm is shown in Figures 4 and 5. In the depth-first traversal, every vertex gets two numbers assigned: index – when the vertex is first visited and incremented on every visit, and low – the lowest index encountered in the subsequently seen vertices. On vertex exit when index equals low, the current component is extracted from the stack. Such a vertex is called the component's *representative*.

### 2.3 An example

Let us present the introduced notions with an example. Consider a directed graph in Figure 1. The strongly connected components of the graph are represented by the partition $\{\{a, b, c\}, \{d\}, \{e, f, g, h\}\}$.

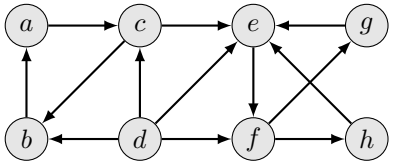


Figure 1: An example of a directed graph on eight vertices.

The trace of the execution of Tarjan's algorithm on the graph from Figure 1 is shown in Figure 2. The order in which the vertices are visited in the depth-first traversal is $a, c, b, e, f, g, h, d$; the corresponding index numbers are written above vertices before the colon. After the colon the low numbers are shown in the order as they are also updated by the algorithm.

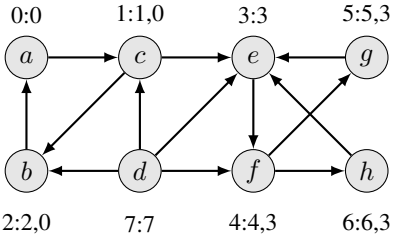


Figure 2: The trace of Tarjan's algorithm.

For example, consider vertex $b$. It is visited after $a$ and $c$, so its index=2. On the first visit its low number is assigned to 2, but when considering $b$'s neighbors (i.e., vertex $a$, which is already marked), its low number is updated to 0. This number is also propagated to vertex $c$ on traversal exit from vertex $b$. Notice also, that the first component to be found by the algorithm is $\{e, f, g, h\}$, proceeding by $\{d\}$, and finally $\{a, b, c\}$.

## 3 Certifying algorithm

In this section we present how to alter Tarjan's algorithm in order to obtain certifying algorithm. Our goal is to provide enough information to the checker to be able to efficiently verify the correctness of the solution. As explained in the next section, a naïve checker is not efficient enough.

Therefore, the extended output of the algorithm is a directed acyclic graph connecting the representatives of strongly connected components. An example of such graph for the strongly connected components of the graph from Figure 1 is shown in Figure 3.

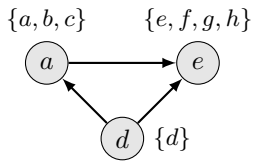


Figure 3: Directed acyclic graph of the component representatives.

Additional information is provided with each vertex (i.e., representative): vertices of the original graph contained in the corresponding component as well as two trees to speed up the checking procedure. The first tree consists of forward edges of the corresponding component, and the second one of backward edges.

The main traversal procedure is shown in Figure 4. It does not differ much from the classic depth-first traversal, apart from the initialization of two graphs, i.e., fw and bw, and a final step consisting of a construction of a directed acyclic graph connecting the representatives.

```
fun tarjan_certify is
    count = 0
    stack = []; result = []
    fw = Graph(); bw = Graph()

    for v ∈ V do
        if not marked(v) then
            connect(v)

    dag = construct_dag(result)
    return dag
```

Figure 4: The main traversal procedure.

In order to construct these two trees, the certifying algorithm builds two forests during the depth-first traversal: see Figure 5, the corresponding forests are denoted with fw (forward edges) and bw (backward edges). The former is extended with an edge on recursive calls, i.e., whenever a new neighboring vertex is visited, and the latter is extended when the low number of a vertex is updated.

To continue our previous example. Forward and backward trees for each component are shown in Figure 6. Notice that, the representatives of the components are $a$, $e$, and $d$.

## 4 Checking the output

In the following sections, we first present an efficient algorithm for checking the correctness of the output of the certifying Tarjan's algorithm. Then we prove the correctness of the checker.

```
fun connect(v) is
    index(v) ← low(v) ← count
    count ← count + 1
    stack.push(v)
    fw.addVertex(v)
    bw.addVertex(v)

    for x ∈ N+(v) do
        if not marked(x) then
            fw.addEdge(v, x)
            connect(x)
            if low(x) < low(v)) then
                low(v) ← low(x)
                bw.addEdge(v, x)
        else if x ∈ stack then
            if index(x) < low(v) then
                low(v) ← index(x)
                bw.addEdge(v, x)
        endif
    done

    if index(v) = low(v) then
        comp = []
        do
            comp.add(stack.pop())
        while stack.top()! = v
        result.add(comp, fw(comp), bw(comp))
    endif
```

Figure 5: The traversal procedure.

### 4.1 Checker algorithm

The input to checker algorithm is the output of a certifying algorithm. Denote with $G = (V, E)$ the original input graph on which the strongly connected components are to be found, and recall that, the output of the certifying algorithm is a directed acyclic graph $H = (U, F)$ of representatives as well as forward and backward tree witnesses. To check the output we proceed as follows.

First, perform a topological sort of the graph $H$. For example, the topological order of vertices of the graph from Figure 3 is $d, a, e$. This can be done in $\Theta(|V(H)| + |E(H)|)$ time [2]. Afterwards, in the reverse topological order perform the following checks for each component $C$ assigned to the representative:

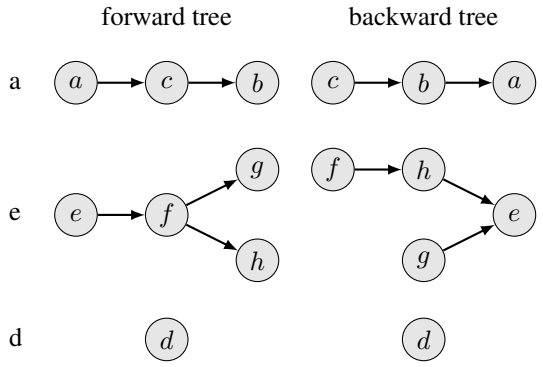


Figure 6: Forward and backward trees for each component.

**C1** Check if all vertices in the component $C$ are also present in the input graph $G$. Here the set of vertices in both forward and backward trees must be the same as well as it must be a subset of $V$. The time complexity of this step is $\Theta(|V(C)|)$.

**C2** Check if all edges in the component's both forward and backward trees are also present in the input graph $G$. This step takes $\Theta(|V(C)|)$ time.

**C3** Check if all vertices in the component $C$ are reachable from each other (in both directions). To do this the forward and backward trees are used for depth-first traversal. Thus, the times complexity of this step is $\Theta(|V(C)|)$.

**C4** Check if all vertices, which are reachable in the original graph $G$ from the component representative, are actually all in this component. To do this, we again employ depth-first traversal on the graph $G$. Notice that the search for reachable vertices is constrained to the corresponding component (because we use reverse topological order). Hence, this step takes $\Theta(|V(C)| + |E(G_C)|)$ time, where $G_C$ is vertex-induced subgraph in $G$ of vertices in $C$.

If all the checks evaluate successfully, then remove the component vertices (and corresponding edges) from the original graph $G$. Afterwards, proceed to the next component. Finally, when the loop terminates the graph $G$ should be empty.

Observe that, the preforming the checks for one component $C$ altogether takes $\Theta(|V(C)| + |E(G_C)|)$ time. Each component has to be checked, hence, the total time complexity of the checker is $\Theta(|V(G)| + |E(G)|)$.

### 4.2 Checker correctness

In order to prove that the algorithm indeed computed the correct result several properties of the output need to hold. In particular, for the resulting components $\mathcal{C}$, we need to show that:

**P1** The components $\mathcal{C}$ contain all the vertices of the input graph $G$, i.e., $V \subseteq \cup_{C \in \mathcal{C}}$.

**P2** The components $\mathcal{C}$ contain only the vertices of the input graph $G$, i.e., $\cup_{C \in \mathcal{C}} \subseteq V$.

**P3** The components $\mathcal{C}$ are pairwise disjoint, i.e., for two components $A, B \in \mathcal{C} \implies A \cap B = \emptyset$.

**P4** The components $\mathcal{C}$ are strongly connected, i.e., there exists a path from $u$ to $v$, where $(u, v) \in A \times A$ for each $A \in \mathcal{C}$.

**P5** The components $\mathcal{C}$ are maximal, i.e., there is no path from $u$ to $v$ or $v$ to $u$, where $u$ and $v$ are vertices from two distinct components in $\mathcal{C}$.

Notice that, the first three properties, i.e., P1, P2, and P3, together prove that the components $C$ are a partition of the input graph vertices.

A naïve approach to checking the five conditions is inefficient (i.e., super linear), since, for example, proving

that there is no path from $u$ to $v$ may involve performing a full graph traversal. Additionally, one must do this for many pairs $u$ and $v$.

Now we show that the checker correctly checks the above conditions. We summarize our result in the following theorem.

**Theorem 1.** *The checks C1, C2, C3, and C4 are sufficient to prove the properties P1, P2, P3, P4, and P5.*

We prove the theorem separately for each property.

*Proof of P1.* Assume that a vertex $x \in V$ is not contained in any of the components. Then the checker will not remove $x$ from the graph $G$. Hence, the graph $G$ will not be empty at the end, and the check will fail. □

*Proof of P2.* This property is examined by the check C1. □

*Proof of P3.* Assume that the components $A, B \in \mathcal{C}$ are not disjoint. Let $x \in A \cap B$. Without loss of generality, assume that $A$ is processed before $B$ for it is preceding $B$ in a reverse topological order. The checker algorithm removes $x$ from the graph when done processing $A$. However, when $B$ is processed, the check C1 fails, since $x$ is no longer part of the graph. □

*Proof of P4.* This property is examined by the check C3, which verifies that all vertices of the component are reachable in the forward tree from the component representative. Additionally, all vertices of the component are also reachable from the component representative in the reversed backward tree, which ensures that the component representative is reachable form all vertices. □

*Proof of P5.* First, recall that the components form a directed acyclic graph. Observe, if there would be a cycle in such a graph, the components on the cycle are actually all part of the same component, i.e., they are not maximal. The checker first proves that the graph of components is indeed directed acyclic graph, by performing topological sort (which only works properly on directed acyclic graphs).

Proof that the paths between different components do not exist stems from the check C4. Since the components are processed in a reverse topological order, the depth-first traversal in the original graph $G$ visits only vertices in this component. Additionally, the component is removed from the graph $G$ at the end of the loop. □

## 5 Conclusion

In the paper we redesigned a well-known Tarjan's algorithm for finding strongly connected components of a given directed graph in order to obtain a certifying algorithm, which additionally to the solution of the problem outputs also the certificate that the solution is correct. Since the changes made to the original algorithm are minor, we expect that the resulting algorithm has very similar performance in practice than the original. Asymptotically, both algorithms have the same time complexity.

Notice also, that there exists another well-known algorithm for the problem, namely Kosaraju's algorithm: we believe that it could also be extended to become certifying by the use of similar techniques as described above.

## Acknowledgments

This work was partially supported by the Slovenian Research Agency and the projects "Parallel and distributed systems" and "Graph optimization methods for Data Analysis and Data Mining".

## References

[1] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. Data Structures and Algorithms. Addison-Wesley, 1983.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, 3rd edition. The MIT Press, 2009.

[3] Jurij Mihelič, Borut Robič. K-center problemi. Zbornik enajste mednarodne Elektrotehniške in računalniške konference ERK 2002, B:3-6, 23.-25. september 2002, Portorož, Slovenija.

[4] Jurij Mihelič, Borut Robič. Algoritmi za razmeščanje centrov. Elektrotehniški vestnik letn. 70(3):162-166, 2003.

[5] Jurij Mihelič, Borut Robič. Algoritmi za problem najmanjšega vozliščnega pokritja. Zbornik trinajste mednarodne elektrotehniške in računalniške konference ERK 2004, B:115-118, 27. - 29. september 2004, Portorož, Slovenija.

[6] R. M. McConnell, K. Mehlhorn, S. Näher, P. Schweitzer. Certifying Algorithms, 2010.

[7] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah, Pascal Schweitzer. An Introduction to Certifying Algorithms. Information Technology 53(6):287-293, 2011.

[8] Micha Sharir. A strong connectivity algorithm and its applications to data flow analysis. Computers and Mathematics with Applications 7(1):67–72, 1981.

[9] R. E. Tarjan, Depth-first search and linear graph algorithms. SIAM Journal on Computing 1 (2):146–160, 1972.