A Space-Efficient Algorithm for Finding Strongly Connected Components

David J. Pearce

School of Engineering and Computer Science
Victoria University, New Zealand
david.pearce@ecs.vuw.ac.nz

Abstract

Tarjan's algorihm for finding the strongly connected components of a directed graph is widely used and acclaimed. His original algorithm required at most $v(2 + 5w)$ bits of storage, where w is the machine's word size, whilst Nuutila and Soisalon-Soininen reduced this to $v(1 + 4w)$. Many real world applications routinely operate on very large graphs where the storage requirements of such algorithms is a concern. We present a novel improvement on Tarjan's algorithm which reduces the space requirements to $v(1 + 3w)$ bits in the worst case. Furthermore, our algorithm has been independently integrated into the widely-used SciPy library for scientific computing.

Keywords: Graph Algorithms, Strongly Connected Components, Depth-First Search.

1. Introduction

For a directed graph $D = (V, E)$, a *Strongly Connected Component (SCC)* is a maximal induced subgraph $S = (V_S, E_S)$ where, for every $x, y \in V_S$, there is a path from x to y (and vice-versa). Tarjan presented a now well-established algorithm for computing the strongly connected components of a digraph in time $\Theta(v+e)$ [14]. In the worst case, this needs $v(2+5w)$ bits of storage, where w is the machine's word size. Nuutila and Soisalon-Soininen reduced this to $v(1+4w)$ [10]. In this paper, we present for the first time an algorithm requiring only $v(1+3w)$ bits in the worst case. Furthermore, this algorithm has been independently integrated into the widely-used SciPy library for scientific computing specifically because of its ability to handle larger graphs in practice [13].

Tarjan's algorithm has found numerous uses in the literature, often as a subcomponent of larger algorithms, such as those for *transitive closure* [9], *compiler optimisation* [5], *program analysis* [1, 11] and for *bisimulation equivalence* [2] to name but a few. Of particular relevance is its use in model checking [7, 12], where the algorithm's storage requirements are a critical factor limiting the number of states which can be explored [8, 4].

2. Depth-First Search

Algorithm 1 presents a well-known procedure for traversing digraphs, known as Depth First Search (DFS). We say that an edge $v \rightarrow w$ is *traversed* if visit(w) is called from visit(v) and that the value of *index* on entry to visit(v) is the *visitation index* of v. Furthermore, when visit(w) returns we say the algorithm is *backtracking* from w to v. The algorithm works by traversing along some branch until a leaf or a previously visited vertex is reached; then, it *backtracks* to the most recently visited vertex with an unexplored edge and proceeds along this; when there is no such vertex, one is chosen from the set of unvisited vertices and this continues until the whole digraph has been

Algorithm 1 DFS(V,E)

1: $index = 0$
2: **for all** $v \in V$ **do** $visited[v] = false$
3: **for all** $v \in V$ **do**
4: **if** $\neg visited[v]$ **then** visit(v)

procedure visit(v)
5: $visited[v] = true$; $index = index + 1$
6: **for all** $v \to w \in E$ **do**
7: **if** $\neg visited[w]$ **then** visit(w)

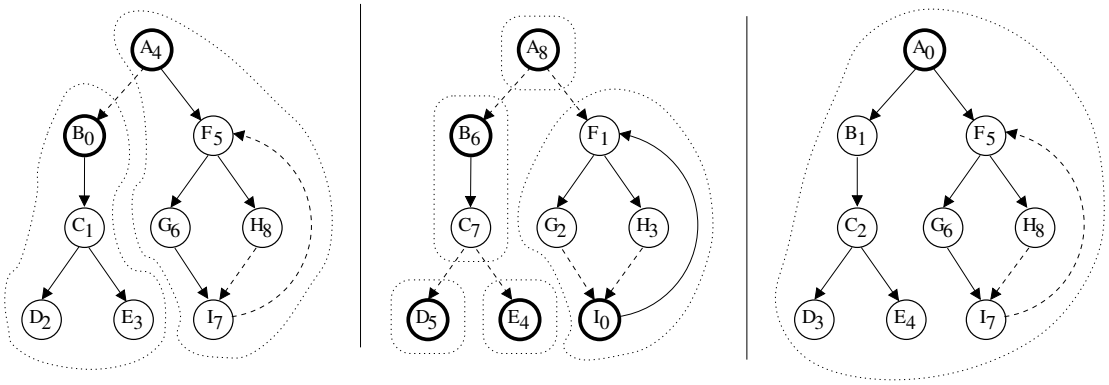

Figure 1: Illustrating three possible traversal forests for the same graph. The key is as follows: vertices are subscripted with their visitation index; dotted lines separate traversal trees; dashed edges indicate those edges not traversed; finally, bold vertices are tree roots.

explored. Such a traversal always corresponds to a series of disjoint trees, called *traversal trees*, which span the digraph. Taken together, these are referred to as a *traversal forest*. Figure 1 provides some example traversal forests.

Formally, $F = (I, T_0, \ldots, T_n)$ denotes a traversal forest over a digraph $D = (V, E)$. Here, I maps every vertex to its visitation index and each T_i is a traversal tree given by $(r, V_{T_i} \subseteq V, E_{T_i} \subseteq E)$, where r is its root. It is easy to see that, if visit(x) is called from the outer loop, then x is the root of a traversal tree. For a traversal forest F, those edges making up its traversal trees are *tree-edges*, whilst the remainder are *non-tree* edges. Non-tree edges can be further subdivided into *forward-*, *back-* and *cross-edges*:

Definition 1. *For a directed graph, $D = (V, E)$, a node x* reaches *a node y, written $x \overset{D}{\rightsquigarrow} y$, if $x = y$ or $\exists z.[x \to z \in E \ \wedge \ z \overset{D}{\rightsquigarrow} y]$. The D is often omitted from $\overset{D}{\rightsquigarrow}$, when it is clear from the context.*

Definition 2. *For a digraph $D = (V, E)$, an edge $x \to y \in E$ is a forward-edge, with respect to some tree $T = (r, V_T, E_T)$, if $x \to y \notin E_T \wedge x \neq y \wedge x \overset{T}{\rightsquigarrow} y$.*

Definition 3. *For a digraph $D = (V, E)$, an edge $x \to y \in E$ is a back-edge, with respect to some tree $T = (r, V_T, E_T)$, if $x \to y \notin E_T \wedge y \overset{T}{\rightsquigarrow} x$.*

Cross-edges constitute those which are neither forward- nor back-edges. A few simple observations can be made about these edge types: firstly, if $x \to y$ is a forward-edge, then $I(x) < I(y)$; secondly, cross-edges may be *intra-tree* (i.e. connecting vertices in the same tree) or *inter-tree*; thirdly, for a back-edge $x \to y$ (note, Tarjan called these *fronds*), it holds that $I(x) \geq I(y)$ and all vertices on a path from y to x are part of the same strongly connected component. In fact, it can also be shown that $I(x) > I(y)$ always holds for a cross-edge $x \to y$ (see Lemma 1, page 10).

Two fundamental concepts behind efficient algorithms for this problem are the *local root* (note, Tarjan called these LOWLINK values) and *component root*: the local root of v is the vertex with the lowest visitation index of any in the same component reachable by a path from v involving at most one back-edge; the root of a component is the member with lowest visitation index. The significance of local roots is that they can be computed efficiently and that, if r is the local root of v, then $r = v$ iff v is the root of a component (see Lemma 3, page 10). Thus, local roots can be used to identify component roots.

Another important topic, at least from the point of view of this paper, is the additional storage requirements of Algorithm 1 over that of the underlying graph data structure. Certainly, v bits are needed for *visited*$[\cdot]$, where $v = |V|$. Furthermore, each activation record for visit$(\cdot)$ holds the value of v, as well as the current position in v's out-edge set. The latter is needed to ensure each edge is iterated at most once. Since no vertex can be visited twice, the call-stack can be at most v vertices deep and, hence, consumes at most $2vw$ bits of storage, where w is the machine's word size. Note, while each activation record may hold more items in practice (e.g. the return address), these can be avoided by using a *non-recursive* implementation (see §4). Thus, Algorithm 1 requires at most $v(1 + 2w)$ bits of storage. Note, we have ignored *index* here, since we are concerned only with storage proportional to $|V|$.

3. Improved Algorithm for Finding Strongly Connected Components

Tarjan's algorithm and its variants are based upon Algorithm 1 and the ideas laid out in the previous section. Given a directed graph $D = (V, E)$, the objective is to compute an array mapping vertices to component identifiers, such that v and w map to the same identifier iff they are members of the same component. Tarjan was the first to show this could be done in $\Theta(v + e)$ time, where $v = |V|$ and $e = |E|$. Tarjan's algorithm uses the *backtracking* phase of Depth-First Search to explicitly compute the local root of each vertex. An array of size $|V|$, mapping each vertex to its local root, stores this information. Another array of size $|V|$ is needed to map vertices to their visitation index. Thus, these two arrays consume $2vw$ bits of storage between them.

The key insight behind our improvement is that these arrays can, in fact, be combined into one. This array, *rindex*$[\cdot]$, maps each vertex to the visitation index of its local root. The outline of our new algorithm, `PEA_FIND_SCC1`, is as follows: on entry to visit(v), *rindex*$[v]$ is assigned the visitation index of v; then, after each successor w is visited, *rindex*$[v] = min($*rindex*$[v],$ *rindex*$[w])$. Figure 2 illustrates this. The algorithm determines which vertices are in the same component (e.g. B, C, D, E, G in Figure 2) in the following way: if, upon completion of visit(v), the local root of v is not v, then push v onto a stack; otherwise, v is the root of a component and its members are popped off the stack and assigned its unique component identifier. In Tarjan's original algorithm, the local root of a vertex was maintained explicitly and, hence, it was straightforward to determine whether a vertex was the root of some component or not. In our improved algorithm, this information is not available and, hence, we need another way of determining this. In fact, it is easy enough to see that the local root of a vertex v is v iff *rindex*$[v]$ has not changed after visiting any successor.

Pseudo-code for the entire procedure is given in Algorithm 2 and there are several points to make: firstly, *root* is used (as discussed above) to detect whether *rindex*$[v]$ has changed whilst visiting v (hence, whether v is a component

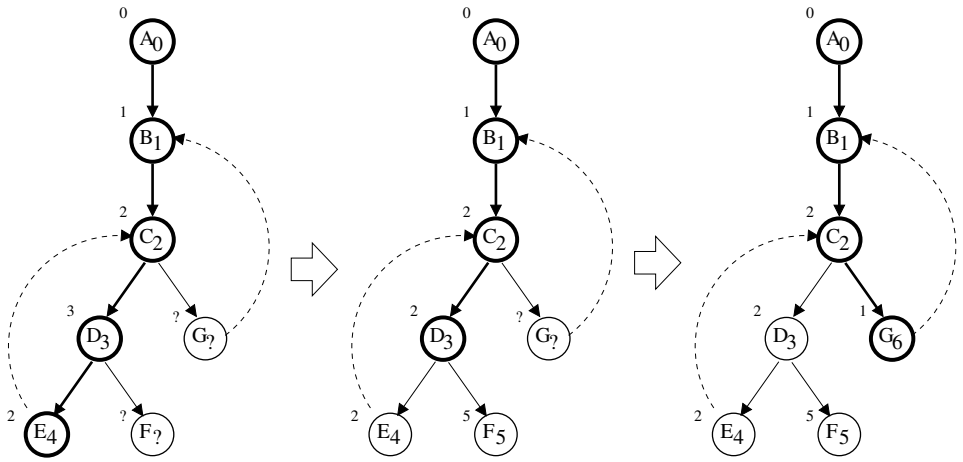

Figure 2: Illustrating the *rindex* computation. As before, vertices are subscripted with visitation index and dashed edges are those not traversed. The left diagram illustrates *rindex*$[\cdot]$ after the path $A \rightsquigarrow E$ has been traversed. On entry to visit(E), *rindex*$[E] = 4$ held, but was changed to *min*$(4,$ *rindex*$[C]) = 2$ because of the edge $E \rightarrow C$. In the middle diagram, visit(E) and visit(F) have completed (hence, the algorithm is backtracking) and *rindex*$[D]$ is *min*$(3,$ *rindex*$[E],$ *rindex*$[F]) = 2$. Likewise, *rindex*$[G] =$ *min*$(6,$ *rindex*$[B]) = 1$ in the right diagram because of $G \rightarrow B$. At this point, the algorithm will backtrack to A before terminating, setting *rindex*$[C] = 1$, *rindex*$[B] = 1$ and *rindex*$[A] = 0$ as it goes.

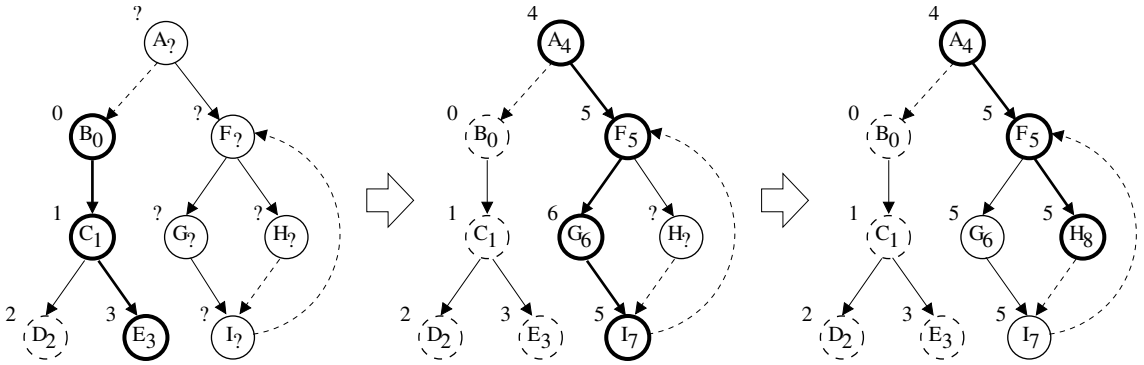

Figure 3: Illustrating why the *inComponent*$[\cdot]$ array is needed. As before, vertices are subscripted with their visitation index; dashed edges indicate those not traversed; finally, *inComponent*$[v] =$ *true* is indicated by a dashed border. In the leftmost diagram, we see that the traversal started from B and that D has already been assigned to its own component (hence, *inComponent*$[D] =$ *true*). In the middle diagram, the algorithm is now exploring vertices reachable from A, having assigned B, C, D and E to their own components. A subtle point is that, on entry to visit(A), *rindex*$[B] <$ *rindex*$[A]$ held (since $A \rightarrow B$ is a cross-edge). Thus, if *inComponent*$[\cdot]$ information was not used on Line 11 to ignore successors already assigned to a component, the algorithm would have incorrectly concluded *rindex*$[A] =$ *min*(*rindex*$[A],$ *rindex*$[B]) = 0$. In the final diagram, *inComponent*$[I] =$ *false* on entry to visit(H) because a vertex is not assigned to a component until its component root has completed.

Algorithm 2 PEA_FIND_SCC1(V,E)

```
1:  for all v ∈ V do visited[v] = false
2:  S = ∅ ; index = 0 ; c = 0
3:  for all v ∈ V do
4:      if ¬visited[v] then visit(v)
5:  return rindex

procedure visit(v)
6:  root = true ; visited[v] = true                    // root is local variable
7:  rindex[v] = index ; index = index + 1
8:  inComponent[v] = false
9:  for all v → w ∈ E do
10:     if ¬visited[w] then visit(w)
11:     if ¬inComponent[w] ∧ rindex[w] < rindex[v] then
12:         rindex[v] = rindex[w] ; root = false
13: if root then
14:     inComponent[v] = true
15:     while S ≠ ∅ ∧ rindex[v] ≤ rindex[top(S)] do
16:         w = pop(S)                                 // w in SCC with v
17:         rindex[w] = c
18:         inComponent[w] = true
19:     rindex[v] = c
20:     c = c + 1
21: else
22:     push(S, v)
```

root); secondly, c is used to give members of a component the same component identifier; finally, the *inComponent*$[\cdot]$ array is needed for dealing with cross-edges. Figure 3 aims to clarify this latter point.

At first glance, Algorithm 2 appears to require $v(3+4w)$ bits of storage in the worst-case. This breaks down in the following way: v bits for *visited*; vw bits for *rindex*; vw bits for S (since a component may contain all of V); $2vw$ bits for the call-stack (as before); finally, v bits for *inComponent* and v bits for *root* (since this represents a boolean stack holding at most $|V|$ elements).

However, a closer examination reveals the following observation: let T represent the stack of vertices currently being visited (thus, T is a slice of the call stack); now, if $v \in T$ then $v \notin S$ holds and vice-versa (note, we can ignore the brief moment a vertex is on both, since it is at most one at any time). Thus, T and S can share the same vw bits of storage to give a total requirement of $v(3+3w)$ for Algorithm 2 (although this does require a non-recursive implementation as before — see §4).

Theorem 1. *Let $D = (V, E)$ be a directed graph. if Algorithm 2 is applied to D then, upon termination, rindex$[v] =$ rindex$[w]$ iff vertices v and w are in the same strongly connected component.*

Proof. Following Tarjan, we prove by induction the computation is correct. Let the induction hypothesis be that, for every vertex v where visit(v) has completed, *rindex*$[v]$ and *inComponent*$[v]$ are correct. That is, if *inComponent*$[v] =$ *true* then *rindex*$[v] =$ *rindex*$[w]$, for every w in v's component; otherwise, *inComponent*$[v] =$ *false* and *rindex*$[v]$ holds the visitation index of v's local root. Thus, k is the number of completions of visit$(\cdot)$. For $k = 1$, visit(x) has only completed for some vertex x. If x has no successors, *rindex*$[x]$ was assigned a unique component identifier and *inComponent*$[x]$ = *true*; otherwise *rindex*$[x] =$ *min*$\{I(y) \mid x \to y \in E\}$ and *inComponent*$[x]$ = *false*. Both are

correct because: a vertex with no successors is its own component; and any $x \to y$ is a back-edge since visit(y) has not completed.

For $k = n$, we have that visit$(\cdot)$ has completed n times. Let x be the vertex where visit(x) will complete next. Assume that, when Line 13 is reached, *rindex*$[x]$ holds the visitation index of x's local root. Then, the algorithm correctly determines whether x is a component root or not (following Lemma 3, which implies *rindex*$[x] = I(x)$ iff x is a component root). If not, *inComponent*$[x]$ =*false* and *rindex*$[x]$ is unchanged when visit(x) completes. If x is a component root, then the other members of its component are stored consecutively at the top of the stack. This is because otherwise some member u was incorrectly identified as a component root, or some non-member u was not identified as a component root (either implies *rindex*$[u]$ was incorrect during visit(u) at Line 13). Since the other members are immediately removed from the stack and (including x) assigned to the same unique component, the induction hypothesis holds.

Now, it remains to show that, on Line 13, *rindex*$[x]$ does hold the visitation index of x's local root. Certainly, if x has no successors then *rindex*$[x] = I(x)$ at this point. For the case that x has one or more successors then *rindex*$[x]$ = *min*$\{$*rindex*$[y] \mid x \to y \in E \wedge$ *inComponent*$[y]$ = *false*$\}$ at this point. To see why this is correct, consider the two cases for a successor y:

(i) *inComponent*$[y]$ =*true*. Let z be y's component root. It follows that visit(z) has completed and was assigned to the same component as y (otherwise some u, where visit(u) has completed, was identified as y's component root, implying *rindex*$[u]$ is incorrect). Now, x cannot be in the same component as y, as this implies $z \overset{T}{\rightsquigarrow} x$ (by Lemma 2) and, hence, that visit(z) had not completed. Thus, the local root of y cannot be the local root of x and, hence, $x \to y$ should be ignored when computing *rindex*$[x]$.

(ii) *inComponent*$[y]$ = *false*. Let z be y's component root. By a similar argument to above, visit(z) has not completed and, hence, $z \overset{T}{\rightsquigarrow} x$. Therefore, x is in the same component as y since $y \rightsquigarrow z$ and, hence, *rindex*$[y]$ should be considered when computing *rindex*$[x]$.

□

4. Further Improvements

In this section, we present three improvements to Algorithm 2 which reduce its storage requirements to $v(1 + 3w)$ by eliminating *inComponent*$[\cdot]$ and *visited*$[\cdot]$. To eliminate the *inComponent*$[\cdot]$ array we use a variation on a technique briefly outlined by Nuutila and Soisalon-Soininen [10]. For *visited*$[\cdot]$, a simpler technique is possible.

The *inComponent*$[\cdot]$ array distinguishes vertices which have been assigned to a component and those which have not. This is used on Line 11 in Algorithm 2 to prevent *rindex*$[w]$ being assigned to *rindex*$[v]$ in the case that w has already been assigned to a component. Thus, if we could ensure that *rindex*$[v] \leq$ *rindex*$[w]$ always held in this situation, the check against *inComponent*$[w]$ (hence, the whole array) could be safely removed. When a vertex v is assigned to a component, *rindex*$[v]$ is assigned a component identifier. Thus, if component identifiers were always greater than other *rindex*$[\cdot]$ values, the required invariant would hold. This amounts to ensuring that *index* $< c$ always holds (since *rindex*$[\cdot]$ is initialised from *index*). Therefore, we make several specific changes: firstly, c is

Algorithm 3 PEA_FIND_SCC2(V,E)

```
1: for all v ∈ V do rindex[v] = 0
2: S = ∅ ; index = 1 ; c = |V| − 1
3: for all v ∈ V do
4:     if rindex[v] = 0 then visit(v)
5: return rindex

procedure visit(v)
6: root = true                                   // root is local variable
7: rindex[v] = index ; index = index + 1
8: for all v→w ∈ E do
9:     if rindex[w] = 0 then visit(w)
10:    if rindex[w] < rindex[v] then rindex[v] = rindex[w] ; root = false
11: if root then
12:    index = index − 1
13:    while S ≠ ∅ ∧ rindex[v] ≤ rindex[top(S)] do
14:        w = pop(S)                             // w in SCC with v
15:        rindex[w] = c
16:        index = index − 1
17:    rindex[v] = c
18:    c = c − 1
19: else
20:    push(S, v)
```

initialised to $|V|-1$ (rather than 0) and decremented by one (rather than incremented) whenever a vertex is assigned to a component; secondly, *index* is now decremented by one whenever a vertex is assigned to a component. Thus, the invariant *index* $< c$ holds because $c \geq |V|-x$ and *index* $< |V|-x$, where x is the number of vertices assigned to a component.

Pseudo-code for the recursive version of our algorithm is shown in Algorithm 3. To eliminate the *visited*$[\cdot]$ array we have used *rindex*$[v] = 0$ to indicate a vertex v is unvisited. In practice, this can cause a minor problem in the special case of a graph with $|V| = 2^w$ vertices and a traversal tree of the same depth ending in a self loop. This happens because the algorithm attempts to assign the last vertex an *index* of 2^w, which on most machines will wrap-around to zero. This can be overcome by simply restricting $|V| < 2^w$, which seems reasonable given that it's providing a potentially large saving in storage.

Finally, we present a non-recursive implementation of Algorithm 3 as, strictly speaking, this is required to obtain our reduced memory requirements in practice. Algorithm 4 gives pseudo-code for the imperative version of Algorithm 3 and a reference implementation in Java is also provided [6]. Unfortunately, Algorithm 4 is somewhat harder to understand than its recursive counterpart. The key is that vS and iS replace the call-stack from the recursive version and, intuitively, can be considered to hold "continuations". Here, the current vertex being visited is on the top of the vS stack, whilst the index of its next out-edge to be traversed is on the top of the iS stack. The procedure visitLoop() is responsible for progressively traversing all out-edges of a given vertex. To avoid the recursive call used in Algorithm 3, the next vertex to visit is placed onto the vS/iS stack in beginEdge() before visitLoop() returns. On subsequent calls to visitLoop(), the vertex being visited is loaded off the vS/iS stack so as to continue where it left off. The edge index, i, identifies both the next vertex to visit and also the vertex which was last visited (if one exists). This allows the necessary processing to be performed once an edge has been traversed,

and is done in finishedEdge().

5. Related Work

Tarjan's original algorithm needed $v(2 + 5w)$ bits of storage in the worst case. This differs from our result primarily because (as discussed) separate arrays were needed to store the visitation index and local root of each vertex. In addition, Tarjan's algorithm could place unnecessary vertices onto the stack S. Nuutila and Soisalon-Soininen addressed this latter issue [10]. However, they did not observe that their improvement reduced the storage requirements to $v(2+4w)$ (this corresponds to combining stacks S and T, as discussed in Section 3). They also briefly suggested that the *inComponent*$[\cdot]$ array could be eliminated, although did not provide details. Finally, Gabow devised an algorithm similar to Tarjan's which (essentially) stored local roots using a stack rather than an array [3]. As such, its worst-case storage requirement is still $v(2 + 5w)$.

Acknowledgements. Thanks to K. Siaulys for some useful corrections on Algorithm 4.

[1] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Language Systems (TOPLAS)*, 12(3):341–395, 1990.

[2] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.

[3] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, May 2000.

[4] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proceedings of the Confernece on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 205–219. Springer, 2004.

[5] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proceedings of the ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 869–878. Society for Industrial and Applied Mathematics, 2004.

[6] `http://github.com/DavePearce/StronglyConnectedComponents/`.

[7] Serge Haddad, Jean-Michel Ilié, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In *Proceedings of the conference on Automated Technology for Verification and Analysis*, pages 196–210. Springer, 2004.

[8] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–95, 1997.

[9] Y. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, 1993.

[10] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, January 1994.

[11] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient Field-Sensitive Pointer Analysis for C. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2004.

Algorithm 4 PEA_FIND_SCC3(V,E)

1: **for all** $v \in V$ **do** *rindex*$[v] = 0$
2: $vS = \emptyset$; $iS = \emptyset$ *index* $= 1$; $c = |V| - 1$
3: **for all** $v \in V$ **do**
4: **if** *rindex*$[v] = 0$ **then** visit(v)
5: **return** *rindex*

procedure visit(v)
6: beginVisiting(v)
7: **while** $vS \neq \emptyset$ **do**
8: visitLoop()

procedure visitLoop()
9: v = top(vS) ; i = top(iS)
10: **while** $i \leq |E(v)|$ **do**
11: **if** $i > 0$ **then** finishEdge$(v, i-1)$
12: **if** $i < |E(v)| \wedge$ beginEdge(v, i) **then return**
13: $i = i + 1$
14: finishVisiting(v)

procedure beginVisiting(v)
15: push(vS, v) ; push$(iS, 0)$
16: *root*$[v]$ = *true* ; *rindex*$[v]$ = *index* ; *index* = *index* $+ 1$

procedure finishVisiting(v)
17: pop(vS) ; pop(iS)
18: **if** *root[v]* **then**
19: *index* = *index* $- 1$
20: **while** $vS \neq \emptyset \ \wedge$ *rindex*$[v] \leq$ *rindex*$[top(vS)]$ **do**
21: w = pop(vS)
22: *rindex*$[w] = c$
23: *index* = *index* $- 1$
24: *rindex*$[v] = c$
25: $c = c - 1$
26: **else**
27: push(vS, v)

procedure beginEdge(v, k)
28: $w = E(v)[k]$
29: **if** $rindex[w] == 0$ **then**
30: pop(iS) ; push$(iS, k+1)$
31: beginVisiting(w)
32: **return** *true*
33: **else**
34: **return** *false*

procedure finishEdge(v, k)
35: $w = E(v)[k]$
36: **if** $rindex[w] < rindex[v]$ **then** $rindex[v] = rindex[w]$; *root*$[v]$ = *false*

[12] Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, and Denis Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In *Proceedings of the confernece on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 668–682. Springer, 2013.

[13] `http://www.scipy.org/`.

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

Appendix A. Appendix

For completeness, we provide in this Section proofs of several key points first shown by Tarjan [14]:

Lemma 1. *Let $D = (V, E)$ be a digraph and $F = (I, T_0, \ldots, T_n)$ a traversal forest over D. If $x \to y$ is a cross-edge then $I(x) > I(y)$.*

Proof. Suppose this were not the case. Then, $I(x) < I(y)$ (note, $x \neq y$ as self-loops are back-edges) and, hence, x was visited before y (recall visitation index is defined in terms of *index* in Algorithm 1, where it is increased on every visit and never decreased). Thus, when visit(x) was invoked, *visited*[y] = *false*. This gives a contradiction because either visit(x) invoked visit(y) (hence $x \to y$ is a tree-edge) or $\exists z.[x \overset{T_i}{\rightsquigarrow} z]$ and visit(z) invoked visit(y) (hence, $x \to y$ is a forward-edge). □

Lemma 2. *Let $D = (V, E)$ be a digraph and $F = (I, T_0, \ldots, T_n)$ a traversal forest over D. If $S = (V_S \subseteq V, E_S \subseteq E)$ is a strongly connected component with root r, then $\exists T_i \in F. \left[\forall v \in V_s.[r \overset{T_i}{\rightsquigarrow} v]\right]$.*

Proof. Suppose not. Then there exists an edge $v \to w \notin E_{T_i}$ where $v, w \in V_s \wedge r \overset{T_i}{\rightsquigarrow} v \wedge r \overset{T_i}{\rightsquigarrow} w$ (otherwise, w is not reachable from r and, hence, cannot be in the same component). It follows that $I(w) < I(v)$, because otherwise visit(v) would have invoked visit(w) (which would imply $v \to w \in E_{T_i}$). Since $v \in T_i$, we know that $r \overset{T_i}{\rightsquigarrow} u$, for any vertex u where $I(r) \leq I(u) \leq I(v)$ (since all vertices traversed from r are allocated consecutive indices). Thus, $I(w) < I(r)$ (otherwise $r \overset{T_i}{\rightsquigarrow} w$) which gives a contradiction since it implies r is not the root of S. □

Lemma 3. *Let $D = (V, E)$ be a digraph, $S = (V_S \subseteq V, E_S \subseteq E)$ a strongly connected component contained and r_v the local root of a vertex $v \in V_S$. Then, $r = v$ iff v is the root of S.*

Proof. Let r_S be the root of S. Now, there are two cases to consider:

i) If $v = r_S$ then $r_v = v$. This must hold as $r_v \neq v$ implies $I(r_v) < I(v)$ and, hence, that $v \neq r_S$.

ii) If $r_v = v$ then $v = r_S$. Suppose not. Then, $I(r_S) < I(r_v)$ and, as S is an SCC, $r_v \rightsquigarrow r_S$ must hold. Therefore, there must be some back-edge $w \to r_S \in E$, where $r_v \rightsquigarrow w \wedge I(r_S) < I(r_v) \leq I(w)$ (otherwise, r_v could not reach r_S). This is a contradiction as it implies r_S (not r_v) is the local root of v. □