Computing strongly connected components in a linear number of symbolic steps

Raffaella Gentilini∗ Carla Piazza† Alberto Policriti∗

Abstract
We present an algorithm that computes in a linear number of symbolic steps (\(O(|V|)\)) the strongly connected components (SCCs) of a graph \(G = (V, E)\) represented by an Ordered Binary Decision Diagram (OBDD). This result matches the complexity of the (celebrated) Tarjan’s algorithm operating on explicit data structures. To date, the best algorithm for the above problem works in \(\Theta(|V| \log |V|)\) symbolic steps ([BGS00]).

1 Introduction
The problem of determining the strongly connected components (SCCs) of a graph is a “classical” one (see [AHU74, CLR90]) and finds applications in different areas of Computer Science such as Computer-Aided Design (CAD), Very Large Scale Integration (VLSI), Model Checking, to name only a few (close to the field of symbolic computation).

Using typical data structures (adjacency-lists or adjacency-matrixes) the SCCs can be computed in linear time (\(O(|V| + |E|)\)) using the elegant and efficient algorithm proposed by Tarjan in [Tar72]. Tarjan’s algorithm is mainly based on the ordering on \(V\) induced by depth-first visit.

Many recent applications involve graphs which cannot be handled with standard (explicit) representations since they are too large to be kept in main memory (see [Bry86], [BCM+92], and [MT98]). To handle such "oversized" problems, OBDDs—representing graphs symbolically—are employed (see Section 3). In this symbolic setting the manipulation of a single node is not possible any longer, rather symbolic algorithms have to deal with sets of nodes. The best symbolic algorithm to date for SCCs computation was proposed by Bloem, Gabow, and Someni in [BGS00] and it works in \(\Theta(|V| \log |V|)\) symbolic steps (see Section 3 for a definition of symbolic primitives).

Our starting point is precisely the fact that Tarjan’s algorithm (as pointed out in [BGS00]) is not applicable when working with OBDDs. This is due to the fact that such algorithm is based on depth-first visit and labelling of the nodes, i.e. it requires to take into consideration and store information for each node individually, while the strong point of OBDDs in saving space, is the fact that sets of nodes are manipulated (and split only when strictly necessary).

The algorithm we present here tries to mimic Tarjan’s algorithm in a symbolic setting. Even though we cannot explicitly label the nodes to obtain the strongly connected components, we can use the computation of all nodes reachable from a given one (the forward-set of a given node) to obtain a suitable ordering driving our computation.

The forward-set computation is relatively inexpensive (in symbolic terms) and will allow us to determine the above mentioned ordering which, notice, is in general different from the one in Tarjan’s procedure. The linear bound (\(O(|V|)\)) to the number of symbolic steps performed by the procedure is then obtained by amortizing the (symbolic) cost of the forward-set computations over the entire procedure which recursively invokes them.

2 Graphs and Strongly Connected Components
We introduce some basic notions concerning graphs and strongly connected components.

A graph, \(G = (V, E)\), is a pair where \(V\) is a finite set (the set of states) and \(E\) is a relation
Let a set $E$ we can safely assume that of induced by considering the subgraph $G = \langle V, E \rangle$ is the diameter $d_G$ of $G = \langle V, E \rangle$ is the maximum distance between two states in $V$. Finally, given $v \in V$, the forward-set of $v$ in $G = \langle V, E \rangle$ is $FW_G(v) = \{w \mid w \in V \land v \rightarrow w\}$. Conversely, the backward-set of $v \in V$, $BW_G(v)$, is $BW_G(v) = \{w \mid w \in V \land w \leftarrow v\}$.

The notion of strongly connected component (scc) is defined on the ground of the relation of mutual reachability: $\ldots \subseteq V \times V$. Given $u, v \in V$, we say that $u \iff v$ if both $u \rightarrow v$ and $v \rightarrow u$. It holds that $\iff$ is an equivalence relation over $V$: the set of strongly connected components of $G = \langle V, E \rangle$ is the partition over $V$ induced by $\iff$.

In the rest of this paper we use the notation $scc_G(v)$ (or simply $scc(v)$) to refer to the strongly connected component to which $v$ belongs. $scc_G(v)$ is said to be trivial if it is equal to $\{v\}$ and $v$ does not belong to any cycle in $G$.

Consider $U \subseteq V$, $U$ is to be said scc-closed if for all vertex $v \in V$, either $scc(v) \cap U = \emptyset$ or $scc(v) \subseteq U$. It is immediate to see that boolean combinations of scc-closed sets are scc-closed. The following lemma whose proof can be found in [XB99], relates some of the above defined notions and constitute the ground for the correctness of the symbolic scc-enumeration algorithms in [BGS00, XB99].

**Lemma 2.1.** Let $G = \langle V, E \rangle$ be a graph and consider the subgraph $G' = \langle U, E \mid U \rangle$ where $U \subseteq V$ is scc-closed. For all $v \in U$, both $FW_{G'}(v)$ and $BW_{G'}(v)$ are scc-closed and $scc_G(v) = FW_{G'}(v) \cap BW_{G'}(v)$

**3 OBDDs and Symbolic Primitives**

In this section we review some basic notions on OBDDs and computational complexity of symbolic procedures.

Binary Decision Diagrams (BDDs) are a fundamental data structure developed for efficiently storing boolean functions. General BDD’s were first introduced in [Lee59, Ake78]. Bryant, introducing in [Bry85] an ordering on the nodes of BDDs (OBDDs), attracted attention on the possibility of their use in logic design verification. OBDDs can be used to represent symbolically each notion which is expressible as a boolean function, for example, as it is usually done in Symbolic Model Checking, graphs.

Any boolean function $f(x_1, \ldots, x_k)$ can be represented by a binary tree of height $k$, whose leaves are labelled by 0 or 1. A path from the root to one leaf represents a boolean assignment $b_1 \ldots b_k$ for the variables $x_1, \ldots, x_k$. The label of the leaf will be 0 or 1 according to the boolean value of $f(b_1, \ldots, b_k)$. Such a tree is called Binary Decision Tree (BDT) for the function $f$. This BDT can be processed by a bottom-up algorithm (see [Bry85]) so as to obtain an acyclic graph that stores the same information in a more compact way, called (Reduced) OBDD for the function $f$. OBDDs are canonical representation for boolean functions since two boolean functions are equivalent if and only if they are associated to the same OBDD [Bry86].

The way OBDDs are usually employed to represent graphs is based on the following observations:

- we can safely assume that $V = \{0, 1\}^v$, i.e. each node is encoded as a binary number;
- a set $U \subseteq V$ is a set of binary strings of length $v$, hence its characteristic boolean function $\chi_U : \{0, 1\}^v \rightarrow \{0, 1\}$, where $\chi_U(u_1, \ldots, u_v) = 1$ $\iff \langle u_1, \ldots, u_v \rangle \in U$ is a boolean function, which can be represented using an OBDD;
- $E \subseteq V \times V$ is a set of binary strings of length $2v$ and hence, again, its characteristic function $\chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v) = 1$ $\iff \langle x_1, \ldots, x_v \rangle E(y_1, \ldots, y_v)$ is a boolean function, which can be represented using an OBDD.

The actual number of nodes of an OBDD varies greatly and strongly depends on the variable ordering (see [MT98]).

Various packages have been developed to manipulate OBDDs: Somenzi’s Cudd at Colorado University [Som01], Lind-Nielsen’s BuDDy, Biere’s ABCD package, Janssen’s OBDD package from Eindhoven University of
Technology, Carnegie Mellon’s OBDD package, the Berkeley’s CAL [SRBSV96], K. Milvang-Jensen’s parallel package BDDNOW, Yang’s PBF package. All these packages are endowed with a number of built-in operations such as equality test and the boolean operations $\cup$, $\cap$, $\setminus$.

Equality test can be considered a constant time operation: if $f$ and $g$ are represented by two OBDDs in the unique table (a table providing access to a unique representation for each OBDD used), then the functions are equal if and only if the variables associated to $f$ and $g$ are two pointers to the same location in the table.

Let us assume that $B_1$ and $B_2$ are the OBDDs representing the boolean functions $f_1(x_1, \ldots, x_k)$ and $f_2(x_1, \ldots, x_k)$, respectively. Then $B_1 \cup B_2$ is an OBDD that represents the function $f_1(x_1, \ldots, x_k) \lor f_2(x_1, \ldots, x_k)$ and can be computed by dynamic programming in time $O(|B_1||B_2|)$, (similarly for $\cap$ and $\setminus$) as explained in [Som99].

The graph operations of image computation (post) and pre-image computation (pre) are usually programmed on the top of the OBDD packages. Consider the boolean functions $\chi_A(y_1, \ldots, y_v)$ and $\chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v)$, representing the set of nodes $A \subseteq V$ and the relation $E$ of the graph $G = (V, E)$. Then, the boolean expression $\exists y_1 \ldots y_v (\chi_A(y_1, \ldots, y_v) \land \chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v))$ gives the set of nodes reachable in one step from $A$. Similarly, the formula $\exists x_1 \ldots x_v (\chi_A(y_1, \ldots, y_v) \land \chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v))$ represents the set of nodes having at least one edge to a vertex in $A$.

In practical cases the cost of the operations post and pre, even thought acceptable, is the crucial one. Thus, in the area of the symbolic algorithms [Som99], the operations post and pre are referred as symbolic steps. On the ground of the above observation is somehow customary to measure the complexity of symbolic (graph) algorithms in terms of symbolic steps (see [BGS00, RBS00]).

4 Related Works

One of the first algorithms proposed to compute the scc’s of a graph using OBDDs as data structure can be found in [HMPS96] and it was mainly based on a previous computation of the transitive closure of the graph. As Xie and Beerel observe in [XB99] “computing the transitive closure has been shown to be computationally expensive in both CPU time and memory”. To save time and space a new algorithm which avoids the transitive closure computation is presented in [XB99]. The key observations behind the algorithm of Xie and Beerel are that both the forward-sets and backward-sets are sclosed and that the scc of a node $v$ is the intersection of its forward-set with its backward-set.

In the same paper the authors point out the importance of defining efficient scc-enumeration algorithms both for the symbolic home-state analysis of Petri nets (see [PCP96]) and for the badcycle detection problem in Model Checking (see [EL86, HKS90]). When measured in terms of nodes, the algorithm presented in [XB99] takes $\Theta(|V|^2)$ symbolic steps in the worst case.

In [BGS00] an algorithm with worst case complexity $\Theta(|V| \log |V|)$ steps is presented and it is shown how it can be applied to decide the emptiness of Street and Büchi automata. The improvements of the algorithm of Bloem, Gabow, and Somenzi w.r.t. to the one in [XB99] are obtained by interleaving the computation of the forward-set and of the backward-set of a node. The first of these two sets which converges is used to determine the two subgraphs on which the recursive calls are made. The algorithm presented is also compared with the algorithms in [EL86, HKS01] whose worst case complexity in terms of nodes is the same as the one in [XB99]. Moreover, as far as the application of the symbolic scc computation in the area of Model Checking is concerned we mention two fundamental papers by Fisler, Fraer, Vardi, and Z. Yang ([FFVY01]) and Ravi, Bloem and F. Somenzi ([RBS00]). In both works the role played by scc determination in solving the central problem of determining the bad computations in Model Checking is discussed in depth.

The algorithm we present here tackles the scc computation problem directly and provides a linear upper bound to the number of symbolic steps which is faster than any previously presented symbolic procedure. The main dif-
ference between our algorithm and the one presented in [BGS00] is that we introduce an order on the forward-set of a node which fully drives our recursive calls. The levels at the ground of our ordering have already been used in [RBS00] where were called onion-rings. In [RBS00] the levels are crucial to determine shorter counter-examples in the context of Model Checking.

5 Spine-sets and Skeletons

In this section we introduce the notion of spine-set and the one of skeleton of a forward-set: these notions will be used by our algorithm to compute in the right order forward-sets and scc’s. Clearly, we cannot maintain information for each node individually, rather we represent the ordering implicitly. Spine-sets are designed for this purpose.

**Definition 5.1. (Spine-set)** Let $S \subseteq V$. The pair $(S, v)$ is a spine-set of $G = (V, E)$ if $v \in S$ and there is a bijection (certifying function) $f: S \mapsto \{1, \ldots, |S|\}$ such that

1. $f(v) = |S|$
2. $\forall u, w \in S(uEw \Rightarrow f(w) = f(u) + 1 \lor f(w) \le f(u))$
3. $\forall u, w (f(w) = f(u) + 1 \Rightarrow uEw)$

The following lemma guarantees that a spine-set has a unique certifying function.

**Lemma 5.1.** A spine-set $(S, v)$ has a unique certifying function.

In virtue of the above lemma, we use the notation $v_1 \ldots v_p$ to express the fact that $(\{v_1, \ldots, v_p\}, v_p)$ is a spine-set of $G = (V, E)$ having as unique certifying function the bijection which maps each $v_i \in \{v_1, \ldots, v_p\}$ into $i$.

**Remark 5.1.** Let $\{v_1, \ldots, v_p\} \subseteq V$. If $v_1 \ldots v_p$, then for all indexes $1 \le j \le p$ we have that $v_1 \ldots v_j$ and $v_1 \ldots v_p$. In fact, consider an index $1 \le j \le p$. By $v_1 \ldots v_p$ we easily obtain that the function mapping each $v_i \in \{v_1, \ldots, v_j\}$ to $i$ is a certifying function for $(\{v_1, \ldots, v_j\}, v_j)$. Symmetrically, the function mapping each $v_i \in \{v_j, \ldots, v_p\}$ to $i - j + 1$ is a certifying function for $(\{v_j, \ldots, v_p\}, v_p)$.

Lemma 5.2 and Lemma 5.3 give some intuitions on the use of the notion of spine-set. In particular, Lemma 5.3 allows to view a spine-set as an implicitly ordered set and, as we said, this order will drive the enumeration of scc’s in our algorithm (see Section 6). Consider the subset of vertices $\{v_1, \ldots, v_p\} \subseteq V$ in the graph $G = (V, E)$.

**Lemma 5.2.** If $(v_1 \ldots v_p)$ is a path of minimum distance from $v$ to $u$ in $G$, then $(S = \{v_1, \ldots, v_p\}, u)$ is a spine-set in $G$.

**Lemma 5.3.** $v_1 \ldots v_p \land p \ge 1 \Rightarrow E^{-1}(v_p) \cap \{v_1, \ldots, v_{p - 1}\} = \{v_{p - 1}\} \cap v_1 \ldots v_{p - 1}$

The following results (Lemma 5.4, 5.5 and 5.6) link the notion of spine-set of a graph to that of strongly connected components.

**Lemma 5.4.** If $(v_1 \ldots v_p)$, there is a maximum $1 \le l \le p$ and a minimum $1 \le t \le p$ such that $scc(v_l) \cap \{v_1, \ldots, v_p\} = (v_1, \ldots, v_l) \land scc(v_p) \cap \{v_t, \ldots, v_p\} = (v_t, \ldots, v_p)$.

**Lemma 5.5.** Let $v_1 \ldots v_p$ and consider $S = \{v_1, \ldots, v_p\} \setminus scc(v_1)$: either $S = \emptyset$ or $E^{-1}(scc(v_p) \cap \{v_1, \ldots, v_p\}) \cap S = \{u\}$ and $(S, u)$ is a spine-set.

**Lemma 5.6.** Let $v_1 \ldots v_p$ and consider $S = \{v_1, \ldots, v_p\} \setminus (scc(v_1))$: either $S = \emptyset$ or $(S, v_p)$ is a spine-set.

**Definition 5.2.** Let $(S, u)$ be a spine-set in the graph $G = (V, E)$. The scc-set of $(S, u)$, $scc((S, u))$, is defined as $scc((S, u)) = \bigcup_{w \in S} scc(w)$.

On the ground of the above results and of Definition 5.2, we obtain the following Lemma 5.7 relating forward-sets and scc’s computation. The complexity analysis of the algorithm presented in Section 6 relies on Lemma 5.7.

**Lemma 5.7.** If $(S, u)$ is a spine-set in $G = (V, E)$, then $FW(u) \cap scc((S, u)) = scc(u)$.

With the above preliminaries we can introduce the notion of skeleton of a node’s forward-set. Intuitively, a skeleton is a spine-set whose nodes are used to amortize the cost of the computation of a forward-set and whose implicit order is used to drive the sequence of scc’s produced in output. Notice that Lemma 5.7 guarantees that the (inverse) order induced by a spine is the correct one for the scc’s computation by means of forward-sets.
**Definition 5.3. (Skeleton of FW(v))** Let FW(v) be the forward-set of the vertex v ∈ V. ⟨S, u⟩ is a skeleton of FW(v) iff u is a node in FW(v) whose distance from v is maximum and S is the set of nodes on a shortest path from v to u.

We conclude this section with the following two immediate observations.

**Lemma 5.8.** If ⟨S, u⟩ is a skeleton of the forward-set FW(v), then S ⊆ FW(v).

**Lemma 5.9.** Let FW(v) be the forward-set of v ∈ V. If ⟨S, u⟩ is a skeleton of FW(v), then ⟨S, u⟩ is a spine-set in G = ⟨V, E⟩.

### 6 The Algorithm

In this section we show how the above introduced notions can be used to design a symbolic scc enumeration algorithm performing a linear number of symbolic steps. We start by giving some intuitions about the procedure. In each iteration the scc of a node, v, is simply determined by first computing FW(v) and then identifying those vertexes in FW(v) having a path to v. The choice of the node to be processed in any given iteration is driven by the implicit (inverse) order associated to an opportune spine-set. More specifically, whenever a forward-set (FW(w)) is built, a skeleton of such a forward-set is also computed. The order induced by the skeleton is then used for the subsequent computations. In this way, the symbolic steps performed to produce FW(w) are distributed over the scc computation of the nodes belonging to a skeleton of FW(w). This amortized analysis is the key point for the linear complexity of the algorithm.

With this intuition, we start describing the procedure in Figure 1 in more detail.

The parameters of the code in Figure 1 are a graph ⟨V, E⟩ and a pair ⟨S, NODE⟩. ⟨S, NODE⟩ is either ⟨∅, ∅⟩ or S = {v₁, . . . , vᵢ} ⊆ V, NODE = {vᵢ}, with v₁ . . . vᵢ (i.e. ⟨{v₁, . . . , vᵢ}, vᵢ⟩ is a spine-set in ⟨V, E⟩).

In case V is empty the routine terminates, otherwise the vertex for which the next strongly connected component is computed is chosen.

In case ⟨S ̸= ∅ and⟩ NODE = {vᵢ}, vᵢ is chosen. Otherwise ⟨S = ∅⟩ an arbitrary element v ∈ V is picked

---

2The function Pick(A) returns the singleton of an element of A.

---

Notice that the restriction of the transition relation in the two recursive calls is only for the sake of clarity. As done in [BGS00] the results of the image and pre-image computations are intersected with V’, so that we store in memory only the original transition relation.

### 7 Soundness, Completeness and Complexity

The soundness and completeness of the algorithm in Figure 1 are rather straightforward and are stated in Theorems 7.1 and...
SCC-Find($V, E, S, NODE$)
begin
set of nodes: $FW, NewS, NewNODE, SCC$

$V', E', S', NODE'$

if $V = \emptyset$ then return;
– Determine the node for which the scc is computed –
if $S = \emptyset$ then NODE ← Pick($V$);
– Compute the forward-set of the vertex in NODE together with a skeleton –
$⟨FW, NewS, NewNODE⟩ ← SkelForward(V, E, NODE);$
– Determine the scc containing NODE –
$SCC ← NODE;$
while $((pre(SCC) \cap FW) \setminus SCC) \neq \emptyset$ do
$SCC ← SCC \cup (pre(SCC) \cap FW);$
– Insert the scc in the scc-Partition –
$SCC$-Partition ← $SCC$-Partition $∪$ $SCC$
– First recursive call: computation of the scc’s in $V \setminus FW$ –
$V' ← V \setminus FW;$ $E' ← E \restriction V'$;
$S' ← S \setminus SCC;$ $NODE' ← pre(SCC \cap S) \cap (S \setminus SCC);$
$SCC-Find(V', E', S', NODE')$
– Second recursive call: computation of the scc’s in $FW \setminus SCC$ –
$V' ← FW \setminus SCC;$ $E' ← E \restriction V'$;
$S' ← NewS \setminus SCC;$ $NODE' ← NewNODE \setminus SCC;$
$SCC-Find(V', E', S', NODE')$
end

Figure 1: The scc algorithm with a linear number of symbolic steps.

7.2, respectively. Lemma 7.1 and Lemma 7.2 are preliminary to such theorems. In particular, Lemma 7.1, states that the subprocedure \texttt{Skel\_Forward}($V, E, \{v\}$) computes the forward-set of $v$ and a skeleton of $FW(v)$.

**Lemma 7.1.** Let $G = (V, E)$. Given $v \in V$, the procedure \texttt{Skel\_Forward}($V, E, \{v\}$) returns the triple $⟨FW(v), S', \{u\}⟩$ where $S', u$ is a skeleton of $FW(v)$.

**Lemma 7.2.** Let $G = (V_G, E_G)$ be a graph with $V_G \neq \emptyset$ and consider the execution of the algorithm \texttt{SCC\_Find} on the parameters $(V_G, E_G, \emptyset, \emptyset)$.
1. Each time line 7 in the code is executed:
   
   \begin{itemize}
   \item [(a)] $(V, E)$ is a subgraph of $(V_G, E_G)$ and $V$ is scc-closed.
   \item [(b)] $NODE = \{v\} \subseteq V$ and if $S \neq \emptyset$, then $⟨S, v⟩$ is a spine-set of $(V, E)$
   \end{itemize}
2. Each time line 11 in the code is executed:
   
   \begin{itemize}
   \item [(a)] $scg(v)$ is assigned to $SCC$ and $⟨SCC, V \setminus FW, FW \setminus SCC⟩$ is a partition over $V$.
   \end{itemize}

**Theorem 7.1.** (Soundness) Consider a graph $G = (V_G, E_G)$. If $SCC \subseteq V_G$ is added to $SCC$-Partition within \texttt{SCC\_Find}($V_G, E_G, \emptyset, \emptyset$), then $SCC$ is a scc of $G$. 
Consider a partition. Let\( O\) be the strongly connected components of\( G\).

Let\( \text{SCC-Find}(G, \emptyset, \emptyset)\) be the recursive call to the procedure in Figure 1 in which\( scc(v)\) is computed and let\( \langle S_v, u \rangle\) be the skeleton of\( FW(v)\) determined by the subprocedure\( \text{Skel}\textunderscore\text{Forward}\) (cfr. Lemma 7.1). Lemma 7.3 allows to amortize steps 1-11 of the procedure\( \text{SCC-Find}(V, E, S, NODE)\) by charging a constant number of symbolic steps on each node in\( scc(v) \cup S_v\). Consider the execution of\( \text{SCC-Find}(G, E_G, \emptyset, \emptyset)\):

\[
\begin{align*}
1 & \text{ Skel\textunderscore Forward } (V, E, NODE) \\
2 & \text{ begin } \\
3 & \text{ set of nodes : LEVEL, FW, } S', NODE' \\
4 & \text{ STACK } \leftarrow \text{ empty stack (of sets) } \\
5 & \text{ LEVEL } \leftarrow \text{ NODE} \\
6 & \text{ Compute the Forward-set and push onto STACK the onion rings } \\
7 & \text{ while (LEVEL } \neq \emptyset\) do \\
8 & \text{ Push(STACK, LEVEL); } \\
9 & \text{ FW } \leftarrow \text{ FW } \cup \text{ LEVEL; } \\
10 & \text{ LEVEL } \leftarrow \text{ post(LEVEL) } \setminus \text{ FW; } \\
11 & \text{ Determine a Skeleton of the Forward-set } \\
12 & \text{ LEVEL } \leftarrow \text{ Pop(STACK); } \\
13 & \text{ } S' \leftarrow \text{ NODE' } \leftarrow \text{ Pick(LEVEL); } \\
14 & \text{ while STACK } \neq \emptyset \text{ do } \\
15 & \text{ LEVEL } \leftarrow \text{ Pop(STACK); } \\
16 & \text{ S' } \leftarrow \text{ S' } \cup \text{ Pick( pre(S') } \cap \text{ LEVEL); } \\
17 & \text{ end } \\
18 & \text{ end } \\
19 & \text{ return } \langle FW, S', NODE' \rangle; \\
\end{align*}
\]

Figure 2: The procedure for computing the forward-set of a node\( v\) together with a skeleton.

**Theorem 7.2. (Completeness)** Consider a graph\( G = \langle V_G, E_G \rangle\). If\( v \in V_G\), then when the procedure\( \text{SCC-Find}(G, E_G, \emptyset, \emptyset)\) terminates all the sccs of\( G\) belong to\( scc\text{-Partition}\).

On the ground Lemma 7.3, Theorem 7.3 proves that\( \text{SCC-Find}(G, E_G, \emptyset, \emptyset)\) computes the strongly connected components of\( (V_G, E_G)\) using\( O(|V_G|)\) symbolic steps. Let\( \text{SCC-Find}(V, E, S, NODE)\) be the recursive call to the procedure in Figure 1 in which\( scc(v)\) is computed and let\( \langle S_v, u \rangle\) be the skeleton of\( FW(v)\) determined by the subprocedure\( \text{Skel}\textunderscore\text{Forward}\) (cfr. Lemma 7.1). Lemma 7.3 allows to conclude that each node in\( V_G\) is charged of a constant number of symbolic steps in at most two distinct recursive calls to\( \text{SCC-Find}\).

**Theorem 7.3. (Complexity)** Consider a graph\( G = \langle V_G, E_G \rangle\). The procedure\( \text{SCC-Find}(G, E_G, \emptyset, \emptyset)\) runs in\( O(|V_G|)\) symbolic steps.

Figure 3 shows the relative sizes of the computed forward-sets and scc’s.\( FW(v)\) is already computed when\( \text{SCC-Find}(V, E, \{v_1, \ldots, v_p\}, v_p)\) is called and subsequently,\( FW(v_p)\) is computed and\( scc(v_p)\) is given in output.

Notice that counting boolean operations as well as symbolic steps would not change the asymptotic complexity of our algorithm.

It is also possible to express the complexity in terms of the diameter\( d_G\) and the size of the\( scc\text{-partition}\)\( N_G\) (cfr. Section 2). In detail, since in each iteration of\( \text{SCC-Find}\) a scc is
detected through a forward-set computation, the complexity expression $O(\min(|V_G|, N_G \times d_G))$ is immediately obtained.

We conclude this section by observing that several heuristics to optimize the implicit scc algorithm in Figure 1 are possible. In particular, the set $T \subseteq V_G$ of nodes belonging to trivial scc’s which do not reach any not-trivial scc could be quickly determined by the following fix-point pre-computation: while $(V_G \neq \text{pre}(V_G))$ do $(V_G \leftarrow \text{pre}(V_G)); \ T \leftarrow (V_G \setminus \text{pre}(V_G)) \cup T$. Each node in $T$ is a trivial scc of the graph in input, thus a non-expensive pre-processing determining $T$ a-priori (alternative to an explicit enumeration of each node in $T$ within the main procedure), would make the algorithm in Figure 1 somehow “more symbolic”.

8 Conclusions

The algorithm presented here allows to solve all those problems that are reducible to the so called bad-cycle detection in verification. For example, the emptyness problem for B"uchi and Street automata can be solved in symbolic linear time by our algorithm when reduced to a check of all possible scc’s of the input automaton. However, specialized version of the routine SCC-Find are, most probably, more suitable for such purposes (see, for example, Double-dfs for the explicit-case) and are currently under study.

The use of dfs-visit as sub-routine often suggests improvements in the complexity of many algorithms in the explicit setting. It would be interesting to see when and how the ideas and the technique proposed here allow such kind of optimizations in a symbolic setting.

References


Consider the recursive call to the procedure $\text{SCC-Find}(V, E, S, \text{NODE})$ (in the procedure $\text{SCC-Find}(V_G, E_G, \emptyset, \emptyset)$) in which $\text{sc}(v)$ is computed. Then, upon the execution of line 6, the singleton $\{v\}$ is assigned to $\text{NODE}$ (see the proof of Lemma 7.1). By Lemma 7.1, the forward-set of $v$, $\text{FW}(v)$, and a skeleton of $\text{FW}(v)$, $\langle S_v, v \rangle$, is obtained performing $\text{Skel Forward}(V, E, \{v\})$ in line 7. By Definition 5.3, $|S_v|$ is the maximum distance from $v$ to a node in $\text{FW}(v)$ i.e. $|S_v| = r$. The call to $\text{Skel Forward}(V, E, \{v\})$ in line 7 costs $2|S_v|$ symbolic steps. In fact, consider the code of the subroutine $\text{Skel Forward}$: each iteration of the while-loop in lines 6-9 enqueue a set onto the priority queue $Q$ and perform one symbolic step (line 9). Upon the return from the loop in lines 6-9 of $\text{Skel Forward}(V, E, \{v\})$, $Q$ has length equal to the maximum distance from $v$ to a node in $\text{FW}(v)$ i.e. equal to $|S_v|$ (Lemma 7.1). As far as the second while-loop of $\text{Skel Forward}(V, E, \{v\})$ is concerned, each iteration of such a loop dequeues a set from $Q$ and perform one symbolic step. Thus it is possible to amortize the cost of executing $\text{Skel Forward}(V, E, \{v\})$ within $\text{SCC-Find}(V, E, S, \text{NODE})$ charging 2 symbolic steps on each node in $S_v$. By Lemma 7.2 each symbolic step performed in the loop of lines 9-10 during $\text{SCC-Find}(V, E, S, \text{NODE})$ discovers at least one element in $\text{sc}(v)$. Hence, the while-loop in lines 9-10 costs at most $|\text{sc}(v)|$ symbolic steps and the thesis follows immediately.

**Proof of Theorem 7.3**

Let’s consider the execution of $\text{SCC-Find}(V_G, E_G, \emptyset, \emptyset)$. In each iteration of the procedure a node $v$ is chosen and its $\text{sc}$ is determined. Then, two recursive calls to the algorithm are performed. By Lemma 7.3, the cost of determining $\text{sc}(v)$ in one iteration can be amortized by charging a constant number of symbolic steps onto each node in $\text{sc}(v) \cup S_v$, where $S_v$ is the first component of the skeleton (of $\text{FW}(v)$) returned by $\text{Skel Forward}$. In detail, it is sufficient to charge three symbolic steps on each $u \in \text{sc}(v)$ and two symbolic steps on each $w \in S_v \setminus \text{sc}(v)$. We now prove that on the entering to each recursive call to $\text{SCC-Find}(V, E, S, \text{NODE})$, $S$ keeps the nodes of $V \subseteq V_G$ previously charged and each vertex in $S$ is charged of
two symbolic steps at most. We proceed by induction on the number of invocation to SCC-Find within SCC-Find(V_G, E_G, ∅, ∅).

As far as the base case is concerned, upon the invocation of SCC-Find(V_G, E_G, ∅, ∅) no node in V_G has been previously charged of any symbolic step. For the inductive step, consider the i + 1-th recursive call to SCC-Find(V^i+1, E^i+1, S^i+1, NODE^i+1). Let (V^j, E^j, S^j, NODE^j) be the parameters of the (j-th) recursive call to the procedure SCC-Find from which the i + 1 call to SCC-Find is executed (hence j ≤ i). Let \{v\} be the singleton assigned to NODE^j upon the j-th time line 6 is executed and let S_v be the set assigned to NewS_v upon the j-th time line 7 is executed. By Lemma 7.3, the cost of executing lines 1-11 of SCC-Find for the j-th time can be amortized by charging three symbolic steps on each w ∈ scc(v) and two symbolic steps on each u ∈ S_v \ scc(v). We have two cases to take in consideration. For the first case, suppose that the i + 1 invocation to SCC-Find corresponds to the first recursive call within the execution of SCC-Find(V^j, E^j, S^j, NODE^j). Hence V^{i+1} = V^j \ FW(v) and S^{i+1} = S^j \ scc(v) i.e. S^{i+1} = S_v \ scc(v) = ∅. From Lemma 5.7, Lemma 5.8, and Lemma 5.9 S^{i+1} ∩ S_v = ∅ and V^{i+1} ∩ S_v = ∅ follow. Thus, in this case iteration j of the algorithm charge no node in V^{i+1}. Exploiting the inductive hypothesis we obtain that, when SCC-Find(V^{i+1}, E^{i+1}, S^{i+1}, NODE^{i+1}) is entered, S^{i+1} ⊆ S^j keeps all nodes previously charged and that each vertex in S^{i+1} is charged of two symbolic steps at most. For the second case, we have that the i + 1 invocation to SCC-Find corresponds to the second recursive call within the execution of SCC-Find(V^j, E^j, S^j, NODE^j). Hence V^{i+1} = FW(v) \ scc(v) and S^{i+1} = S_v \ scc(v). From Lemma 5.7, Lemma 5.8, and Lemma 5.9 S^j ∩ S^{i+1} = ∅ and V^{i+1} ∩ S^j follow. Hence, exploiting the inductive hypothesis, on the entering to SCC-Find(V^{i+1}, E^{i+1}, S^{i+1}, NODE^{i+1}), only nodes in S^{i+1} have been previously charged. Moreover, as S^{i+1} = S_v \ scc(v), from Lemma 7.3 it follows that nodes in S^{i+1} are charged of two symbolic steps (within the j-th iteration of the procedure).

With this preliminary, consider a recursive call to SCC-Find(V, E, S, NODE) within SCC-Find(V_G, E_G, ∅, ∅). If S = ∅, then no node in V ⊆ V_G has been charged of any symbolic step in previous recursive calls. Otherwise (S ≠ ∅) S keeps the nodes previously used to amortize some symbolic step and each vertex in S is charged of two symbolic steps at most. By Lemma 7.1, if S ≠ ∅, then NODE = {p} and (S, p) is a spine-set in (V, E). Moreover, p is chosen as the node of which determining the strongly connected component (line 6). If (S_p, t) is the skeleton of FW(p) obtained executing Skel Forward(V, E, {p}), then, by Lemma 5.7, S_p ∩ S = scc(p). Thus we have that the nodes in S_p \ scc(p) are charged for the first time (within the execution of SCC-Find(V_G, E_G, ∅, ∅)) with 2 symbolic steps. The nodes in S \ S_p ⊆ scc(p) have been previously charged: they are charged of three symbolic steps more within the execution of the procedure SCC-Find(V, E, S, NODE). However they will never be encountered again (because they belong to scc(p)). Finally, nodes in (S_p \ S) ∩ scc(p) are charged for the first time. Hence, we have that each node is charged of 5 symbolic steps at most, overall the entire execution of SCC-Find(V_G, E_G, ∅, ∅).