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 $G = \langle V, E \rangle$ 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 *odering* 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 *simbolically*—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 Somenzi 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, $G = \langle V, E \rangle$, is a pair where V is a finite set (the set of *states*) and E is a relation

*Dip. di Matematica e Informatica. Università di Udine. Via Le Scienze 206, 33100 Udine - Italy.

†Dip. di Informatica. Università Ca' Foscari di Venezia. Via Torino 155, 30172 Mestre (VE) - Italy.

over V (the set of *edges*). The graph $\langle U, E \restriction U\rangle$ is a subgraph of $G = \langle V, E\rangle$ if $U \subseteq V$ and $E \restriction U = E \cap (U \times U)$. A finite sequence of states, $\langle v_1, \ldots, v_p\rangle$, is said to be a *path* of *length* $p-1$ from u to v if $v_1 = u$, $v_p = v$, and $(v_{i-1}, v_i) \in E$ for all $2 \leq i \leq p$. Given two states $u, v \in V$, we say that u *reaches* v ($u \rightsquigarrow v$) whenever there exists a path from u to v. In case $u \rightsquigarrow v$, the *distance* from u to v is defined as the minimum length of a path from u to v, otherwise is ∞. The notion of distance allows to define the *diameter* of a graph: 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 $FW_G(v) = \{w \mid w \in V \wedge v \rightsquigarrow w\}$. Conversely, the *backward-set* of $v \in V$, $BW_G(v)$, is $BW_G(v) = \{w \mid w \in V \wedge w \rightsquigarrow v\}$.

The notion of *strongly connected component* (*scc*) is defined on the ground of the relation of *mutual reachability*: $\leftrightsquigarrow \subseteq V \times V$. Given $u, v \in V$, we say that $u \leftrightsquigarrow v$ if both $u \rightsquigarrow v$ and $v \rightsquigarrow u$. It holds that $\leftrightsquigarrow$ 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 $\leftrightsquigarrow$.

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 $G' = \langle U, E \restriction U\rangle$ 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 $\chi_U(u_1, \ldots, u_v) = 1 \Leftrightarrow \langle u_1, \ldots, u_v\rangle \in U$ 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

  $$\chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v) = 1 \Leftrightarrow \langle x_1, \ldots, x_v\rangle E \langle y_1, \ldots, y_v\rangle$$

  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 $f_1(x_1, \ldots, x_k) \vee f_2(x_1, \ldots, x_k)$ and can be computed by dynamic programming in time $O(|B_1||B_2|)$, (similarly for $\cap$ and $\setminus$)[1] as explained in [Som99].

The graph operations of image computation (post) and pre-image computation (post) 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 $\exists y_1 \ldots y_v(\chi_A(y_1, \ldots, y_v) \wedge \chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v))$ gives the set of nodes reachable in one step from A. Similarly, the formula $\exists x_1 \ldots x_v(\chi_A(y_1, \ldots, y_v) \wedge \chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v))$ represents the set of nodes having at least one edge to a vertex in A. Computing the above mentioned expressions, also called *relational products*, has a worst-case complexity which is exponential in the sizes of the OBDDs representing χ_A and χ_E [McM93, Som99].

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]).

[1] If B is an OBDD, then $|B|$ denotes the number of its nodes.

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 very 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 scc-closed 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 [PCPn96]) and for the *bad-cycle detection* problem in Model Checking (see [EL86, HKSV01]). 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, HKSV01] 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 rôle 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 counterexamples 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 $\mathcal{S} \subseteq V$. The pair $\langle \mathcal{S}, v \rangle$ is a spine-set of $G = \langle V, E \rangle$ iff $v \in \mathcal{S}$ and there is a bijection (certifying function) $f : \mathcal{S} \mapsto \{1, \ldots, |\mathcal{S}|\}$ such that*

1. $f(v) = |\mathcal{S}|$
2. $\forall u, w \in \mathcal{S}(uEw \Rightarrow f(w) = f(u)+1 \vee f(w) \leq f(u))$
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 $\langle \mathcal{S}, v \rangle$ has a unique certifying function.*

In virtue of the above lemma, we use the notation $v_1 \vec{\ldots} v_p$ to express the fact that $\langle \{v_1, \ldots, v_p\}, v_p \rangle$ is a spine-set of $G = \langle V, E \rangle$ 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 $v_1 \vec{\ldots} v_p$, then for all indexes $1 \leq j \leq p$ we have that $v_1 \vec{\ldots} v_j$ and $v_j \vec{\ldots} v_p$. In fact, consider an index $1 \leq j \leq p$. By $v_1 \vec{\ldots} v_p$ we easily obtain that the function mapping each $v_i \in \{v_1, \ldots, v_j\}$ to i is a certifying function for $\langle \{v_1, \ldots, v_j\}, v_j \rangle$. Symmetrically, the function mapping each $v_i \in \{v_j, \ldots, v_p\}$ to $i - j + 1$ is a certifying function for $\langle \{v_j, \ldots, v_p\}, v_p \rangle$.*

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 $G = \langle V, E \rangle$.

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

LEMMA 5.3. $v_1 \vec{\ldots} v_p \wedge p > 1 \Rightarrow E^{-1}(v_p) \cap \{v_1, \ldots v_{p-1}\} = \{v_{p-1}\} \wedge v_1 \vec{\ldots} v_{p-1}$

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 $v_1 \vec{\ldots} v_p$, there is a maximum $1 \leq l \leq p$ and a minimum $1 \leq t \leq p$ such that $scc(v_1) \cap \{v_1, \ldots, v_p\} = \{v_1, \ldots, v_l\} \wedge scc(v_p) \cap \{v_1, \ldots, v_p\} = \{v_t, \ldots, v_p\}$.*

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

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

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

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 $\langle \mathcal{S}, u \rangle$ is a spine-set in $G = \langle V, E \rangle$, then $FW(u) \cap scc(\langle \mathcal{S}, u \rangle) = scc(u)$.*

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 $v \in V$. $\langle \mathcal{S}, u \rangle$ is a skeleton of $FW(v)$ iff u is a node in $FW(v)$ whose distance from v is maximum and $\mathcal{S}$ 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 $\langle \mathcal{S}, u \rangle$ is a skeleton of the forward-set $FW(v)$, then $\mathcal{S} \subseteq FW(v)$.*

LEMMA 5.9. *Let $FW(v)$ be the forward-set of $v \in V$. If $\langle \mathcal{S}, u \rangle$ is a skeleton of $FW(v)$, then $\langle \mathcal{S}, u \rangle$ is a spine-set in $G = \langle V, E \rangle$.*

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 $\langle V, E \rangle$ and a pair $\langle \mathcal{S}, NODE \rangle$. $\langle \mathcal{S}, NODE \rangle$ is either $\langle \emptyset, \emptyset \rangle$ or $\mathcal{S} = \{v_1, \ldots, v_p\} \subseteq V$, $NODE = \{v_p\}$, with $v_1 \vec{\ldots} v_p$ (i.e. $\langle \{v_1, \ldots, v_p\}, v_p \rangle$ is a spine-set in $\langle V, E \rangle$).

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

In case ($\mathcal{S} \neq \emptyset$ and) $NODE = \{v_p\}$, v_p is chosen. Otherwise ($\mathcal{S} = \emptyset$) an arbitrary element $v \in V$ is picked[2] (assigning the singleton $\{v\}$ to $NODE$). Then the subprocedure **Skel_Forward** is invocated to compute the forward-set of $NODE$ together with a skeleton, $\langle \mathcal{S}', u' \rangle$, of such a forward-set. The local variable FW maintains the just mentioned forward-set whereas $New\mathcal{S}$ and $NewNODE$ maintain $\mathcal{S}'$ and $\{u'\}$, respectively. In line 8 the local variable SCC is initialized to be the singleton $NODE$ and then it is augmented with the strongly connected component containing NODE (loop of lines 9-10). In line 11 the partition of scc's is updated and finally the procedure is recursively called over:

1. the subgraph of $\langle V, E \rangle$ induced by $V \setminus FW$ and the spine-set of such a subgraph obtained from $\langle \mathcal{S}, NODE \rangle$ by subtracting SCC (cf. Lemma 5.5);
2. the subgraph of $\langle V, E \rangle$ induced by $FW \setminus SCC$ and the spine-set of such a subgraph obtained from $\langle New\mathcal{S}, NewNODE \rangle$ by subtracting SCC (cf. Lemma 5.6).

In Figure 2 the pseudocode of the subprocedure **Skel_Forward** is presented. It is used within the execution of **SCC-Find** to obtain the forward-set of a node together with a skeleton of such a forward-set. The parameters of such a routine are a graph $G = \langle V, E \rangle$ and a singleton $NODE = \{v\} \subseteq V$. The forward-set of the node in input, v, is simply computed with a symbolic breadth first search [Som99, RBS00] i.e. by a loop that discovers, in each iteration i, all the nodes of V having distance i from v. In [RBS00] the just mentioned sets are referred as *onion-rings* and are enqueued onto a set-priority-queue to produce a counterexample of minimum length. In this context, they are pushed onto a stack to produce a skeleton of $FW(v)$.

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

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

```
1  SCC-Find(V, E, S, NODE)
2    begin
3      set of nodes : FW, NewS, NewNODE, SCC
4                     V', E', S', NODE'

5      if V = ∅ then return;

- Determine the node for which the scc is computed -
6      if S = ∅ then NODE ← Pick(V);

- Compute the forward-set of the vertex in NODE togheter with a skeleton -
7      ⟨FW, NewS, NewNODE⟩ ← Skel_Forward(V, E, NODE);

- Determine the scc containing NODE -
8      SCC ← NODE;
9      while (( pre(SCC) ∩ FW) \ SCC) ≠ ∅ do
10       SCC ← SCC ∪ ( pre(SCC) ∩ FW);

- Insert the scc in the scc-Partition -
11     SCC-Partition ← SCC-Partition ∪ SCC

- First recursive call: computation of the scc's in V \ FW -
12     V' ← V \ FW;  E' ← E ↾ V';
13     S' ← S \ SCC;  NODE' ← pre(SCC ∩ S) ∩ (S \ SCC);
14     SCC-Find(V', E', S', NODE')

- Second recursive call: computation of the scc's in FW \ SCC -
15     V' ← FW \ SCC;  E' ← E ↾ V';
16     S' ← NewS \ SCC;  NODE' ← NewNODE \ SCC;
17     SCC-Find(V', E', S', NODE')
18   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 **Skel_Forward**$(V, E, \{v\})$ computes the forward-set of v and a skeleton of $FW(v)$.

LEMMA 7.1. *Let $G = \langle V, E\rangle$. Given $v \in V$, the procedure* **Skel_Forward**$(V, E, \{v\})$ *returns the triple $\langle FW(v), S', \{u\}\rangle$ where $\langle S', u\rangle$ is a skeleton of $FW(v)$.*

LEMMA 7.2. *Let $G = \langle V_G, E_G\rangle$ be a graph with $V_G \neq \emptyset$ and consider the execution of the algorithm* **SCC-Find** *on the parameters $\langle V_G, E_G, \emptyset, \emptyset\rangle$.*

1. *Each time line 7 in the code is executed:*
   (a) *$\langle V, E\rangle$ is a subgraph of $\langle V_G, E_G\rangle$ and V is scc-closed.*
   (b) *$NODE = \{v\} \subseteq V$ and if $S \neq \emptyset$, then $\langle S, v\rangle$ is a spine-set of $\langle V, E\rangle$*
2. *Each time line 11 in the code is executed:*
   (a) *$scc_G(v)$ is assigned to SCC and $\langle SCC, V \setminus FW, FW \setminus SCC\rangle$ is a partition over V.*

THEOREM 7.1. (SOUNDNESS) *Consider a graph $G = \langle V_G, E_G\rangle$. If $SCC \subseteq V_G$ is added to SCC_Partition within* **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$, *then SCC is a scc of G.*

```
1  Skel_Forward (V, E, NODE)
2    begin
3      set of nodes : LEVEL, FW, S', NODE';
4      STACK ← empty stack (of sets);
5      LEVEL ← NODE

- Compute the Forward-set and push onto STACK the onion rings -
6      while (LEVEL ≠ ∅) do  {
7          Push(STACK, LEVEL);
8          FW ← FW ∪ LEVEL;
9          LEVEL ← post(LEVEL) \ FW; }

- Determine a Skeleton of the Forward-set -
10     LEVEL ← Pop(STACK);
11     S' ← NODE' ← Pick(LEVEL);
12     while STACK ≠ ∅ do
13         LEVEL ← Pop(STACK);
14         S' ← S' ∪ Pick( pre(S') ∩ LEVEL);
15     return⟨FW, S', NODE'⟩;
16   end
```

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* **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$ *terminates all the sccs of* G *belong to scc_Partition.*

On the ground Lemma 7.3, Theorem 7.3 proves that **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$ computes the strongly connected components of $\langle V_G, E_G \rangle$ using $O(|V_G|)$ symbolic steps. Let **SCC-Find**$(V, E, \mathcal{S}, NODE)$ be the recursive call to the procedure in Figure 1 in which $scc(v)$ is computed and let $\langle \mathcal{S}_v, u \rangle$ be the skeleton of $FW(v)$ determined by the subprocedure **Skel_Forward** (cfr. Lemma 7.1). Lemma 7.3 allows to amortize steps 1-11 of the procedure **SCC-Find**$(V, E, \mathcal{S}, NODE)$ by charging a constant number of symbolic steps on each node in $scc(v) \cup \mathcal{S}_v$. Consider the execution of **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$:

LEMMA 7.3. *Let* **SCC-Find**$(V, E, \mathcal{S}, NODE)$ *be the recursive call in which* $v \in V \subseteq V_G$ *is chosen as the node of which determining the scc.* $scc(v)$ *is computed using at most* $2r + |scc(v)|$ *symbolic steps where* r *is the maximum distance from* v *to a node in* $FW_{\langle V,E \rangle}(v)$.

Theorem 7.3 states that the worst-case complexity of the scc-symbolic algorithm of Figure 1 is linear in the size of states space of the input graph. The key point in the proof is showing that, in each recursive call to **SCC-Find**$(V, E, \mathcal{S}, NODE)$, $\mathcal{S}$ maintains the set of vertexes in $V \subseteq V_G$ that have been already charged of some symbolic step in the previous computation. On this ground, Lemma 5.7 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 **SCC-Find**.

THEOREM 7.3. (COMPLEXITY) *Consider a graph* $G = \langle V_G, E_G \rangle$. *The procedure* **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$ *runs in* $O(|V_G|)$ *symbolic steps.*

Figure 3 shows the relative sizes of the computed forward-sets and scc's . $FW(v_1)$ is already computed when **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*-partition N_G (cfr. Section 2). In detail, since in each iteration of **SCC-Find** a scc is

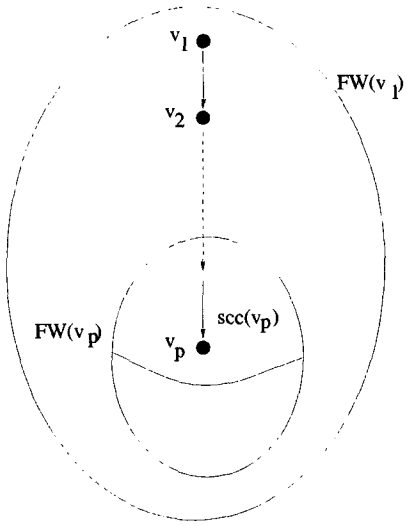

Figure 3: Relative sizes of the computed forward-sets.

detected through a forward-set computation, the complexity expression $O(\min(|V_G|, N_G * d_G))$ 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 $(V_G \neq pre(V_G))$ do $\{V_G \leftarrow pre(V_G);\ T \leftarrow (V_G \setminus pre(V_G)) \cup T\}$. Each node in T is a trivial scc of the graph in input, thus a non-expensive preprocessing 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 Büchi 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

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transaction on Computers*, 27(6):509–516, 1978.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BGS00] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In W. A. Hunt Jr. and S. D. Johnson, editors, *Proc. of Int. Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 37–54. Springer, 2000.

[Bry85] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proc. 22nd Design Automation Conference*, 1985.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, 1986.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[EL86] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proc. of First Annual Symp. on Logic in Computer Science (LICS'86)*, pages 267–278. IEEE, 1986.

[FFVY01] K. Fisler, R. Fraer, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 420–434. Springer-Verlag, Berlin, 2001.

[HKSV01] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using BDDs. *Formal Methods in System Design*, 18(2):131–140, 2001.

[HMPS96] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovin analysis of large finite state machines. *IEEE Transaction*

on Computer-Aided Design, 15(12):1479–1493, 1996.

[Lee59] C. Y. Lee. Binary decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[McM93] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[MT98] C. Meinel and T. Theobald. *Algorithms and data structures in VLSI Design: OBDD Foundations and Applications.* Springer-Verlag, Berlin, 1998.

[PCPn96] E. Pastor, J. Cortadella, and M. A. Peña. Structural methods to improve the symbolic analysis of Petri nets. In *Proc. of Conference on Application and Theory of Petri Nets*, volume 1639 of *Lecture Notes in Computer Science*, pages 26–45. Springer-Verlag, Berlin, 1996.

[RBS00] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt Jr. and S. D. Johnson, editors, *Proc. of Int. Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 143–160. Springer, 2000.

[Som99] F. Somenzi. *Calculational System Design*, volume 173 of *Nato Science Series F: Computer and Systems Sciences*, chapter Binary Decision Diagrams, pages 303–366. IOS Press, 1999.

[Som01] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.1.* 2001. Available at http://vlsi.colorado.edu/ fabio/CUDD/.

[SRBSV96] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance bdd package based on exploiting memory hierarchy. In *Proc. of ACM/IEEE Design Automation Conference*, 1996.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[XB99] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proc. of Int. Conference on Computer Aided Design (CAD'99)*, pages 37–40, 1999.

Appendix

All the proofs are available from `http://www.dimi.uniud.it/~gentilin/sscc.ps`. In this section we report only the proofs of the complexity results.

Proof of Lemma 7.3

Consider the recursive call to the procedure **SCC-Find**$(V, E, \mathcal{S}, NODE)$ (in the procedure **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$) in which $scc(v)$ is computed. Then, upon the execution of line 6, the singleton $\{v\}$ is assigned to $NODE$ (see the proof of Lemma 7.1). By Lemma 7.1, the forward-set of v, $FW(v)$, and a skeleton of $FW(v)$, $\langle \mathcal{S}_v, u \rangle$, is obtained performing **Skel_Forward**$(V, E, \{v\})$ in line 7. By Definition 5.3, $|\mathcal{S}_v|$ is the maximum distance from v to a node in $FW(v)$ i.e. $|\mathcal{S}_v| = r$. The call to **Skel_Forward**$(V, E, \{v\})$ in line 7 costs $2|\mathcal{S}_v|$ symbolic steps. In fact, consider the code of the subroutine **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 **Skel_Forward**$(V, E, \{v\})$, Q has length equal to the maximum distance from v to a node in $FW(v)$ i.e. equal to $|\mathcal{S}_v|$ (Lemma 7.1). As far as the second while-loop of **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 **Skel_Forward**$(V, E, \{v\})$ within **SCC-Find**$(V, E, \mathcal{S}, NODE)$ charging 2 symbolic steps on each node in $\mathcal{S}_v$. By Lemma 7.2 each symbolic step performed in the loop of lines 9-10 during **SCC-Find**$(V, E, \mathcal{S}, NODE)$, discovers at least one element in $scc(v)$. Hence, the while-loop in lines 9-10 costs at most $|scc(v)|$ symbolic steps and the thesis follows immediately.

Proof of Theorem 7.3

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

two symbolic steps at most. We proceed by induction on the number of invocation to **SCC-Find** within **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$. As far as the base case is concerned, upon the invocation of **SCC-Find**$(V_G, E_G, \emptyset, \emptyset)$ 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**$(V^{i+1}, E^{i+1}, \mathcal{S}^{i+1}, NODE^{i+1})$. Let $\langle V^j, E^j, \mathcal{S}^j, NODE^j \rangle$ 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 $j \leq i$). Let $\{v\}$ be the singleton assigned to $NODE^j$ upon the j-th time line 6 is executed and let $\mathcal{S}_v$ be the set assigned to $New\mathcal{S}^j$, 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 $w \in scc(v)$ and two symbolic steps on each $u \in \mathcal{S}_v \setminus scc(v)$. 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**$(V^j, E^j, \mathcal{S}^j, NODE^j)$. Hence $V^{i+1} = V^j \setminus FW(v)$ and $\mathcal{S}^{i+1} = \mathcal{S}^j \setminus scc(v)$ i.e. $\mathcal{S}^{i+1} \cap scc(v) = \emptyset$. From Lemma 5.7, Lemma 5.8, and Lemma 5.9 $\mathcal{S}^{i+1} \cap \mathcal{S}_v = \emptyset$ and $V^{i+1} \cap \mathcal{S}_v = \emptyset$ 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**$(V^{i+1}, E^{i+1}, \mathcal{S}^{i+1}, NODE^{i+1})$ is entered, $\mathcal{S}^{i+1} \subset \mathcal{S}^j$ keeps all nodes previously charged and that each vertex in $\mathcal{S}^{i+1}$ 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**$(V^j, E^j, \mathcal{S}^j, NODE^j)$. Hence $V^{i+1} = FW(v) \setminus scc(v)$ and $\mathcal{S}^{i+1} = \mathcal{S}_v \setminus scc(v)$. From Lemma 5.7, Lemma 5.8, and Lemma 5.9 $\mathcal{S}^j \cap \mathcal{S}^{i+1} = \emptyset$ and $V^{i+1} \cap \mathcal{S}^j$ follow. Hence, exploiting the inductive hypothesis, on the entering to **SCC-Find**$(V^{i+1}, E^{i+1}, \mathcal{S}^{i+1}, NODE^{i+1})$, only nodes in $\mathcal{S}^{i+1}$ have been previously charged. Moreover, as $\mathcal{S}^{i+1} = \mathcal{S}_v \setminus scc(v)$, from Lemma 7.3 it follows that nodes in $\mathcal{S}^{i+1}$ are charged of two symbolic steps (within the j-th iteration of the procedure).

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