Algorithmica (2008) 50: 120–158
DOI 10.1007/s00453-007-9079-5

Symbolic Graphs: Linear Solutions to Connectivity Related Problems

Raffaella Gentilini · Carla Piazza · Alberto Policriti

Received: 15 June 2006 / Accepted: 19 June 2006 / Published online: 8 December 2007
© Springer Science+Business Media, LLC 2007

Abstract The importance of symbolic data structures such as Ordered Binary Decision Diagrams (OBDD) is rapidly growing in many areas of Computer Science where the large dimensions of the input models is a challenging feature: OBDD based graph representations allowed to define truly new standards in the achievable dimensions for the Model Checking verification technique. However, OBDD representations pose strict constraints in the algorithm design issue. For example, Depth-First Search (DFS) is not feasible in a symbolic framework and, consequently, many state-of-the-art DFS based algorithms (e.g., connectivity procedures) cannot be easily rearranged to work on symbolically represented graphs. We devise here a symbolic algorithmic strategy, based on the new notion of *spine-set*, that is general enough to be the engine of linear symbolic step algorithms for both strongly connected components and biconnected components. Our procedures improve on previously designed connectivity symbolic algorithms. Moreover, by an application to the so-called "bad

This work is a revised and extended version of [22, 23]. It is partially supported by the projects PRIN 2005015491 and BIOCHECK.

R. Gentilini · C. Piazza · A. Policriti (✉)
Università di Udine (DIMI), Via Le Scienze 206, 33100 Udine, Italy
e-mail: policriti@dimi.uniud.it

R. Gentilini
e-mail: gentilin@dimi.uniud.it

C. Piazza
e-mail: piazza@dimi.uniud.it

R. Gentilini
Department of Computer Science Reactive Systems Group, Kaiserslautern University, Kaiserslautern, Germany

A. Policriti
Institute of Applied Geonomics, Odine, Italy

cycle detection problem", our technique can be used to efficiently solve the emptiness problem for various kinds of ω-automata.

Keywords Strongly connected components · Biconnected components · Ordered binary decision diagrams · Model checking · Massive graphs

1 Introduction

The problems tackled in this paper are two (very) classical ones: strongly connected components computation for directed graphs and biconnected components computation for undirected graphs [13, 26, 39]. The new ingredient motivating and justifying our efforts is the use of a specific *symbolic* data structure representing the input data and somehow forcing the basic operations performed by our algorithms. The importance of symbolic data structures such as the one we use, Ordered Binary Decision Diagrams (OBDD) [9, 10, 43], is rapidly growing in many areas of Computer Science where the large dimensions of the input models is a challenging feature: OBDD based graph representations allowed to achieve truly new standards in the dimensions for the Model Checking verification technique [12, 28, 35].

Many authors have taken into account the problem of precisely characterizing the kind of speed-up which an OBDD-based representation can give. We discuss in some detail such studies and the relative position of our contribution with respect to those works in Sect. 3.2. We point out here that the cost model we use is based on counting the number of symbolic steps (see [37]) and that the main motivation for using such a model is the practical observation that in applications (for example Model Checking) such cost is coherent with experimental results [4, 5, 12, 28, 37]. At a higher level of abstraction, we observe that when working with succinct representations (e.g., circuits, boolean formulae, OBDD, etc.) it is rather natural to consider basic operations different from the standard ones. In particular, if succinctness is obtained via a sharing mechanism (as in the case of the "Reduce" routine collapsing nodes in Binary Decision Trees to produce OBDDs [9, 10]), the *natural* operations to be used in estimating costs must operate on more than one represented object at a time. The post and pre operations—introduced in Sect. 3.1 and producing the set of nodes reachable in one forward or backward step, respectively, from a set of nodes—are *natural* in the above sense.

From an algorithmic point of view, OBDD representations pose strict constraints on the design techniques. For example, Depth-First Search (DFS) is not feasible (i.e., is not efficient) in a symbolic framework [4, 5, 12, 28, 37] and, consequently, many state-of-the art DFS based algorithms (e.g., connectivity procedures [13, 26, 39]) cannot be easily rearranged to work on symbolically represented graphs. We devise here a symbolic algorithmic *strategy*—based on the new notion of *spine-set*—that is general enough to be the engine of linear symbolic step algorithms for both strongly connected components and biconnected components. As the algorithms we present here exemplify, the notion of spine-set is a sort of a symbolic way to introduce an ordering among (set of) nodes. Exploiting such an order is easy and the complexity analysis results in a linear number of symbolic steps on the ground of a natural amortization of such steps' cost on the output production (see Sects. 5 and 6). An

additional feature of our approach—which we consider fundamental to obtain good practical performances and to determine a cost model based on symbolic steps—is the fact that the number of variables involved in the OBDDs used by our algorithms is small and remains constant throughout the entire execution (see Sect. 3.2).

The validity of our approach, which is mainly of theoretical interest for the moment, is witnessed by the lower complexity of our symbolic algorithm for computing strongly connected components (SCC) with respect to the existing symbolic procedures for the same problem [4, 5, 15, 41, 45]. More specifically, our $\mathcal{O}(V)$ symbolic steps SCC algorithm improves on the previous state-of-the-art SCC procedure in [4, 5], performing $\mathcal{O}(V \log(V))$ symbolic steps. We also provide, here, an $\mathcal{O}(V)$ symbolic steps solution to the biconnectivity problem, thereby proving the non singularity of our spine-set based approach. To the best of our knowledge there are not other symbolic algorithms for the biconnectivity problem, despite of its growing importance in areas such as Networks Design, where huge graphs are involved (see, e.g., [46]). Moreover, our procedures can be used to check the emptiness of various ω-automata and hence to solve the *bad cycle detection* problem in Model Checking in a linear number of symbolic steps (see Sect. 7).

The paper is organized as follows: Sect. 2 reviews some basic notions concerning graph connectivity. In Sect. 3 we present preliminary material on symbolic graph algorithms and we discuss some related works. In Sect. 4 we introduce the central notion, with respect to our approach, of *spine-set*. Sections 5 and 6 develop linear symbolic steps procedures for strongly connected and biconnected components analysis, respectively. Finally, in Sect. 7, we reuse spine-sets to solve the bad cycle detection problem for various ω-automata in a linear number of symbolic steps. Preliminary versions of some results presented in this work appeared in [22, 23].

2 Preliminaries

This section collects preliminary notions and notations, used in the rest of the paper, concerning graphs and connectivity.

Definition 1 (Graph) Let V be a finite set of vertices (vertex set) and $E \subseteq V \times V$ be a binary relation on V (edge set). The structure $G = \langle V, E \rangle$ is a *directed graph* (*digraph*). Moreover, if E is symmetric the structure $G = (V, E)$ is an *undirected graph*.

In undirected graphs the edges (u, v) and (v, u) are considered to be the same edge. We will simply write G to denote a graph that can be either directed or undirected. Given G having vertices in V and edges in E and given a set of vertices $U \subseteq V$ we use the notations $\mathsf{post}(U)$ and $\mathsf{pre}(U)$ to denote the set of nodes $\mathsf{post}(U) = \{v \mid \exists u \in U (u, v) \in E\}$ and $\mathsf{pre}(U) = \{v \mid \exists u \in U (v, u) \in E\}$, respectively. If G is undirected, then $\mathsf{post}(U)$ and $\mathsf{pre}(U)$ always coincide and we denote them by $\mathsf{img}(U)$. When $U = \{u\}$ is a singleton, with an abuse of notation, we will write $\mathsf{post}(u)$ ($\mathsf{pre}(u)$ and $\mathsf{img}(u)$) instead of $\mathsf{post}(\{u\})$ ($\mathsf{pre}(\{u\})$ and $\mathsf{img}(\{u\})$), respectively).

Given a graph G having V as set of vertices and edges in E and a subset U of V we use $E \restriction U$ to denote the set of edges incident onto U, i.e., $E \restriction U = \{(u, v) \mid u, v \in U \wedge (u, v) \in E\}$. A *subgraph* G' of G is a graph whose set of vertices is a subset U of V and whose set of edges is $E \restriction U$. A *path* p *in* G *of length* $n \geq 0$ is a sequence $p = (v_0, v_1, \ldots, v_n)$ of elements of V such that for each $0 \leq i < n$ it holds that $(v_i, v_{i+1}) \in E$. The *edges of the path* p are the edges (v_i, v_{i+1}) with $0 \leq i < n$. A *subpath* of p is a subsequence of p. A (simple) *cycle of* G is a path $p = (v_0, v_1, \ldots, v_n)$ such that $n > 0$ and $v_0 = v_n$. A path $p = (v_0, v_1, \ldots, v_n)$ is said to be *simple* if it does not contain a subpath which is a cycle. A node u *reaches* a node v (v *is reachable from* u) if there exists a path p whose first node is u and whose last node is v. The notion of reachability can be immediately extended to set of nodes.

Definition 2 (Backward and Forward Sets) Let $G = \langle V, E \rangle$ be a digraph and $U \subseteq V$ be a set of nodes. We define the *backward set* of U, denoted by $BW_G(U)$, as the set of nodes that reach U. Conversely, we define the *forward set* of U, denoted by $FW_G(U)$, as the set of nodes reachable from U.

When $U = \{v\}$ is a singleton we also use the notation $BW_G(v)$ ($FW_G(v)$) to denote $BW_G(U)$ ($FW_G(U)$, respectively).

2.1 Strong Connectivity of Digraphs

In a digraph $G = \langle V, E \rangle$ two nodes u and v are said to be *mutually reachable*, denoted by $u \leftrightsquigarrow v$, if both u reaches v and vice versa. The relation $\leftrightsquigarrow$ of mutual reachability is an equivalence relation over V.

Definition 3 (Strongly Connected Components) Given $G = \langle V, E \rangle$, consider the partition $\{V_1, \ldots, V_k\}$ of V induced by $\leftrightsquigarrow$. The *strongly connected components* of G are the subgraphs $\langle V_1, E \restriction V_1 \rangle, \ldots, \langle V_k, E \restriction V_k \rangle$.

A graph is said to be *strongly connected* if it consists of a unique strongly connected component. In the rest of this paper we use the notation $scc_G(v)$ (or simply $scc(v)$) to refer to the set of vertices U such that $v \in U$ and $\langle U, E \restriction U \rangle$ is a strongly connected component of G. $scc_G(v)$ is said to be *trivial* if it is equal to $\{v\}$ and $(v, v) \notin E$.

Definition 4 (Scc-Closed Vertex Set) Given $G = \langle V, E \rangle$, let $U \subseteq V$. U is said to be *scc closed* if, for each vertex $v \in V$, either $scc(v) \cap U = \emptyset$ or $scc(v) \subseteq U$.

Boolean combinations of scc-closed sets are scc closed. Lemma 1, whose proof is immediate (see [45]), relates some of the above defined notions and ensures the correctness of the symbolic scc-algorithms in [4, 5, 45].

Lemma 1 *Let $G = \langle V, E \rangle$ be a digraph 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*

$$scc_G(v) = FW_{G'}(v) \cap BW_{G'}(v).$$

2.2 Biconnectivity of Undirected Graphs

The notion of *biconnectivity* is the counterpart, in an undirected graph, of the notion of strong connectivity.

Consider $G = (V, E)$ and, from now on, assume that $|E| \geq 1$ and that G contains no self loops. Moreover, we suppose that G is connected. A vertex $a \in V$ is said to be an *articulation point* of G if there exist two distinct vertices $v \neq a$ and $w \neq a$ such that every path between v and w contains a. In other words, a is an articulation point if the removal of a splits G into two or more parts. $G = (V, E)$ is *biconnected* if it contains no articulation point. The biconnected components of $G = (V, E)$ can be defined upon the following equivalence relation $\circlearrowleft$ on E.

Definition 5 Let $G = (V, E)$. Two edges $e_1, e_2 \in E$ are in relation $\circlearrowleft$ if and only if either they are the same edge or there is a simple cycle in G containing both e_1 and e_2.

Definition 6 (Biconnected Components) Given $G = (V, E)$, consider the partition $\{E_1, \ldots, E_k\}$ of E induced by $\circlearrowleft$. For each $1 \leq i \leq k$, let V_i be the set of nodes occurring in the edges of E_i. The *biconnected components* of G are the subgraphs $(V_1, E_1), \ldots, (V_k, E_k)$.

In the rest of this paper we use the notation $bcc_G(v, u)$ (or simply $bcc(v, u)$) to indicate the biconnected component containing the edge (v, u) in G. Lemma 2, whose proof is immediate (see e.g., [1, 20]), gives useful information on biconnectivity.

Lemma 2 *For $1 \leq i \leq k$, let $G_i = (V_i, E_i)$ be the biconnected components of $G = (V, E)$. Then*:

1. *G_i is biconnected*;
2. *for all $i \neq j$, $V_i \cap V_j$ contains at most one vertex*;
3. *a is an articulation point of G if and only if $a \in V_i \cap V_j$ for some $i \neq j$.*

Definition 7 (Bcc-Closed Subgraph) Let $G' = (V', E')$ be a subgraph of $G = (V, E)$. G' is said bcc closed if, for each biconnected component of G, (V_i, E_i), either $E_i \cap E' = \emptyset$ or $E_i \subseteq E'$.

3 Symbolic Graph Algorithms

We review here some basic notions on OBDDs and symbolic graph algorithms. We then discuss a number of issues concerning both the design and the analysis of algorithms manipulating OBDD represented graphs.

Ordered Binary Decision Diagrams (OBDDs) [2, 9, 10, 27] are a fundamental data structure originally developed for efficiently storing and manipulating boolean functions. Any boolean function $f(x_1, \ldots, x_k)$ can be naturally represented by a *Binary Decision Tree* (*BDT*) of height k. In the BDT for $f(x_1, \ldots, x_k)$ each path defines a boolean assignment, $b_1 \ldots b_k$, for the variables of f and the leaves are labelled with

the boolean value $f(b_1, \ldots, b_k)$. Processing a BDT bottom up, we obtain a directed acyclic graph which compactly stores the same information through node-sharing [2, 27, 29, 43]. By introducing an ordering over the node labelling variables in such diagrams, Bryant [9, 10] showed how to produce a *canonical* representation and, consequently, a manipulation framework for boolean functions (the OBDDs). OBDDs can be used to symbolically represent each notion which is expressible as a boolean function, e.g., in *Symbolic Model Checking* [28] they are used to represent Kripke structures (graphs). Besides Model Checking, other typical areas of applications of OBDD techniques are logic synthesis [9, 10], VLSI design and CAD problems [29] and many others.

3.1 Symbolic Representation and Manipulation of Graphs

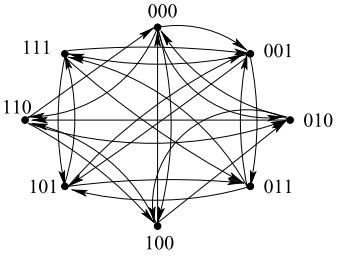
The way OBDDs are usually employed to represent a graph G, having vertices in V and edges in E, is based on the following observations:

- Each node is encoded as a binary number, i.e., $V = \{0, 1\}^v$. Hence, a set $U \subseteq V$ is a set of binary strings of length v whose characteristic (boolean) function, $\chi_U(u_1, \ldots, u_v)$, can be represented by an OBDD;
- $E \subseteq V \times V$ is a set of binary strings of length $2v$ whose characteristic (boolean) function, $\chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v)$, can be represented by an OBDD.

The actual number of nodes of an OBDD greatly varies and strongly depends on the variable ordering (see [29]). Indeed, even if there exist a number of effective heuristics to rearrange a given OBDD variable ordering [14], the problem of obtaining the best variable ordering has been proved NP-complete [6, 36].

Example 1 Consider the graph in Fig. 1. To give an OBDD representation of such a graph, we need to encode its nodes with a three variables boolean code, $\langle x_1, x_2, x_3 \rangle$. Accordingly, six variables, namely $\langle x_1, x_2, x_3, x'_1, x'_2, x'_3 \rangle$, are necessary to give a symbolic representation of the graph relation.

We depict in Fig. 1, on the right, the OBDD encoding of the graph relation, with respect to the variable ordering $\langle x_3, x'_3, x_1, x_2, x'_1, x'_2 \rangle$. Using the alternative variable ordering $\langle x_1, x'_1, x_2, x'_2, x_3, x'_3 \rangle$, we would obtain a worst-case size OBDD for this example, having a number of nodes (edges) exponential in the number of variables



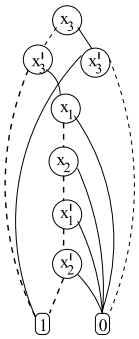


Fig. 1 A graph and the OBDD representation of its relation, with respect to the variable ordering $\langle x_3, x'_3, x_1, x_2, x'_1, x'_2 \rangle$

(or, equivalently, linear with respect to $|V| + |E|$). In fact, the first three levels of the resulting OBDD would be a complete binary tree. In [30, 34], the authors discuss a number of graph topologies for which there exists a variable ordering that guarantees an OBDD representation having logarithmic size with respect to the graph size.

Various packages have been developed to manipulate OBDDs: Somenzi's CUDD at Colorado University [38], Lind-Nielsen's BuDDy, Biere's ABCD package, Janssen's OBDD package from Eindhoven University of Technology, Yang's PBF package, Carnegie Mellon's OBDD package, the Berkeley's CAL [32], and K. Milvang-Jensen's parallel package BDDNOW. All these packages are endowed with a number of built-in operations such as equality test and the boolean operations $\cup, \cap, \setminus$.

Canonicity guarantees that equality tests can be considered constant time operations: 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 in time $\mathcal{O}(|B_1||B_2|)$ using dynamic programming [10] (if B is an OBDD, then $|B|$ denotes the number of its nodes). The same cost (linear in the size of the input OBDDs) is achieved by the other set-composition operations and by the operation $\mathsf{pick}(U)$ that picks an element from the set U [4, 5].

The digraph operations of post-image computation (post) and pre-image computation (pre), are usually programmed on top of the OBDD packages. Consider the boolean functions $\chi_U(y_1, \ldots, y_v)$ and $\chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v)$, representing the set of nodes $U \subseteq V$ and the relation E of the graph $G = (V, E)$. Then, the expression $\exists x_1 \ldots x_v(\chi_U(x_1 \ldots x_v) \wedge \chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v))$ gives the set of nodes reachable in one step from U. Similarly, the nodes reaching U in one step satisfy $\exists y_1, \ldots, y_v(\chi_U(y_1, \ldots, y_v) \wedge \chi_E(x_1, \ldots, x_v, y_1, \ldots, y_v))$. Both the formulas defined above allow us to obtain, in an undirected graph, the image-set of a given set of nodes U, $\mathsf{img}(U)$. Computing these expressions, also called *relational products*, has a worst-case complexity which is exponential in the number of variables of the OBDDs representing χ_U and χ_E [28, 37].

In practical cases the cost of the operations of (post/pre) image computation, even thought acceptable, is the crucial one. Hence, in the area of the symbolic algorithms, the operations post, pre, and img are referred as *symbolic steps* [37].

3.2 Symbolic Graph Algorithms Analysis

To take advantage of OBDD space-saving features, symbolic graph algorithms should operate on *sets* of nodes and/or edges, rather than on single graph elements. In fact, in this way every OBDD operation (hopefully) processes many nodes and edges in simultaneously. Moreover, computing the successors of one node has the same worst-case cost as computing the successors of a set of nodes. Operating on *sets* of graph

elements should be somehow *inherent* to a proper definition of symbolic algorithms and poses strict constraints on the possibility of rearranging classical algorithms in the symbolic setting.

Apart from the agreement on the above general suggestion [3, 37, 43], there is no definite assessment, in the literature, of what a *good* symbolic algorithm is, on how to evaluate complexity of symbolic procedures, as well as on how to compare symbolic algorithms. To this purpose, a natural question arising when graph problems are taken into consideration assuming a symbolic data representation, is the following: what is the complexity of the graph problem with respect to the dimension of the symbolic input representation? This issue was considered in [16] with respect to a very simple graph problem: the graph accessibility problem, consisting in determining the existence of a path connecting two given nodes in a graph. The authors of [16] showed that it is possible to encode every decision problem on a polynomial space bounded Turing machine into a GAP problem. The graph underling the encoding represents all possible configurations and transitions of the Turing machine and it is shown to have a succinct (polynomial in the number of variables) OBDD representation: hence the GAP problem is PSPACE complete with respect to the dimensions of a symbolic input representation. However, if the number of nodes of the manipulated (explicit) graph are considered, the above result simply assesses that it is very difficult to use OBDDs to define poly-logarithmic algorithms for GAP as well as more complex graph problems. It remains possible to devise OBDD-based algorithms that solve polynomially the above problems and "often" with sublinear space and/or time (with respect to the explicit graph dimension). In fact, these characteristics are shared by all the symbolic graph algorithms defined, to date, in the verification field.

Seen from a different perspective, the result in [16] suggests the inadequacy of classical complexity analysis, when dealing with symbolic procedures, due to the great distance between best and worst cases. For instance the GAP problem is poly-logarithmic in the best case and polynomial in the worst case, with respect to the dimensions of the explicitly represented graph. The need for new parameters to evaluate relative merits of symbolic algorithms have been considered by various authors. We review some of them below and, finally, we set the framework used in this paper to evaluate and compare symbolic algorithms.

Current Trends in Comparing Symbolic Algorithms Most of the authors in the verification and Model Checking communities, evaluate a symbolic algorithm counting the number of OBDD operations performed [11, 12, 18, 28]. In [4, 5, 37] the notion of *symbolic steps* was introduced, capturing the fact that symbolic operations based on relational product can cause an explosion of the input OBDD size. Thus, the asymptotic number of symbolic steps, i.e., of the operations based on relational product, is taken as a measure of symbolic algorithms performance.

Indeed, comparing symbolic procedures exclusively on the ground of bounds on the number of general, as well as specific, OBDD operations is not fair since the sizes of the manipulated OBDD are not taken into account. To consider OBDD sizes as well, it is possible to use, as a performance parameter, also the number of variables of intermediate OBDDs manipulated. The importance of having the variables number as low as possible was recognized by various authors [3, 18, 25, 37]. As a matter of

fact multiplying by a constant k the number of variables on which an OBDD depends, means exploding its worst case size S to S^k. The algorithms defined in this paper will use at most $2n$ variables, were n is the number of variables needed to encode the graph vertex set. Employing $2n$ variables allows us to compute on sets of nodes using pre and post operations and leaving the edge relation *untouched*. In a certain sense, hence, this is the minimum number of variables needed to get some information from the graph relation.

In [33, 34, 44] Sawitzki and Woelfel do not pose any restriction to the number of variables in the computed OBDD. As an example, the topological sort algorithm in [44] requires $4n$ variables and relies on computing the transitive closure of the input graph relation. The main aim of [33, 34, 44] is that of exploiting the power of graph relation composition to which it is possible to apply speedup strategies such as *iterative squaring* [11]. However, to avoid the problems related to the (likely) huge intermediate OBDD sizes, largely experimentally recognized in the verification community [4, 5, 37], [33, 34, 44] must impose strict constraints on their graph topology (only very regular graphs such as grids are considered). Computing on such regular graphs guarantees to have all intermediate OBDDs of size polynomial in the number of variables: the complexity analysis is then carried on in the classical way and the costs obtained are poly-logarithmic.

In this paper, we take seriously into account suggestions coming from the great deal of experimental work done in symbolic model checking. In fact, we do not want pose constraints on graph topologies. Following [4, 5, 12, 18, 28, 37], our graph connectivity symbolic algorithms will work on sets of nodes, rather than edges, thus minimizing the number of variables in intermediate OBDDs (as well as their worst case sizes). Being the number of variables fixed, we will use the asymptotic number of *symbolic steps* [4, 5] to compare algorithms.

3.3 Related Work on Symbolic Algorithms

As we already observed, since symbolic graph algorithms should work on sets of graph elements, it is not always possible to translate classical algorithms into efficient symbolic procedures. The simple problem of exploring a graph is significant to this phenomenon. Symbolically, the problem can be efficiently tackled in a breadth-first manner: sets of nodes having increasing distance from the source-node are considered and the diameter of the graph is a bound on the number of symbolic steps necessary to complete the visit. To perform Depth-First Search (DFS) instead, an order over the nodes must be computed. This order is associated to depth of nodes in the paths subsequently discovered. Thus, it is necessary to maintain a global vertex labelling discovering time and to subsequently take into consideration a node at a time according to such a labelling. As a result, DFS is intractable in a symbolic setting [4, 5, 37]. Recently, the strongly connected components computation (which, in the explicit setting, is linearly solved starting from a DFS visit) has been tackled in the area of verification. Motivated by the possibility of speeding up some model checking algorithms, the authors of [4, 5, 45] proposed two symbolic algorithms to find the strongly connected components of a directed graph. Both the algorithms compute the strongly connected component of a node, v, by breadth-first discovering, and then

intersecting, the forward-set and the backward-set of v. Moreover, both algorithms need the same number of variables, in that they both manipulate sets of nodes: the procedure in [4, 5] outperforms the one in [45], since it needs $\mathcal{O}(V \log(V))$ rather than $\mathcal{O}(V^2)$ symbolic steps.

The design of symbolic graph algorithms for *bisimulation* reduction has also been proposed in the area of Model Checking [7, 12, 17, 35]. The first symbolic bisimulation algorithms [7, 12] obtained the bisimulation equivalence of a graph by computing a minimum fix-point binary relation over the nodes. Later, [18] proposed bisimulation symbolic algorithms that do not manipulate relations but, rather, partitions of vertex sets: this way the number of variables involved in the computation is cut in half.

All the above symbolic procedures were deeply experimentally analyzed on a data set derived from graphs modeling systems in verification. A data set of random graphs, besides that of graphs deriving from Model Checking, was used to test the algorithm in [24]. The procedure symbolically solves the max-flow problem on 0-1 networks on graphs having more than 10^{30} nodes.

Finally, the authors in [33, 34, 44], developed a number of symbolic graph algorithms (topological sorting [44], max-flow on 0-1 networks, all pair shortest paths problem [34]) all relying on symbolic computation of transitive closure of graph relation. Transitive closure computation is well known, in Model Checking, to likely lead to huge intermediate OBDDs. Hence, even if it requires only $\log(n)$ iterations by *iterative squaring* it is usually avoided.

4 Spine-sets as Symbolic Counterpart to DFS

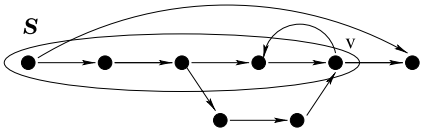
In this section we introduce the notion of *spine-set*. A spine-set allows us to implicitly encode an ordering, suitable for the efficient symbolic computation of both biconnected components and strongly connected components. In this sense, spine-sets are, in the design of connectivity algorithms, a sort of symbolic counterpart to the use of DFS.

A path $(v_0, \ldots, v_p)$ in a graph G is said a *chordless path* if and only if for all $0 \leq i < j \leq p$ such that $j - i > 1$, there is no edge from v_i to v_j in G.

Definition 8 (Spine-set) Consider a graph G having vertices in V and edges in E. Let $\mathcal{S} \subseteq V$. The pair $\langle \mathcal{S}, v \rangle$ is a *spine-set* of G if and only if G contains a chordless path whose set of vertices is $\mathcal{S}$ that ends at v. The node v is called the *spine-anchor* of the spine-set $\langle \mathcal{S}, v \rangle$.

Example 2 In Fig. 2 we provide a pictorial representation of a spine-set $\mathcal{S}$ and its anchor v in the case of a digraph.

Fig. 2 A Spine-set $\langle \mathcal{S}, v \rangle$



It is immediate to verify that a spine-set is associated to a unique chordless path. On this ground, we use the notation $v_0 \overline{\ldots} v_p$ to express the fact that $\langle \{v_0, \ldots, v_p\}, v_p \rangle$ is a spine-set of G associated to the chordless path $(v_0, \ldots, v_p)$. Though simple, Lemma 3 is significant in that it allows us to view a spine-set as an *implicitly ordered set*.

Lemma 3 *If* $v_0 \overline{\ldots} v_p$ *and* $p > 0$, *then* $\mathsf{pre}(v_p) \cap \{v_0, \ldots, v_p\} = \{v_{p-1}\}$ *and* $v_0 \overline{\ldots} v_{p-1}$.

Proof Each subpath of a chordless path is clearly a chordless path. Hence, $v_0 \overline{\ldots} v_{p-1}$ and $\{v_{p-1}\} \subseteq \mathsf{pre}(v_p) \cap \{v_0, \ldots, v_p\}$. Moreover, $\{v_{p-1}\} \supseteq \mathsf{pre}(v_p) \cap \{v_0, \ldots, v_p\}$ also holds in that, otherwise, there would be some edge from a spine-set node v_j to v_p, with $p - j > 1$. □

Section 4.1 relates spine-sets to the strongly connected components of a digraph. Similarly Sect. 4.2 relates spine-sets to the biconnected components of an undirected graph.

4.1 Spine-sets and Strongly Connected Components

In this section we consider a digraph $G = \langle V, E \rangle$.

Lemma 4 *Given the spine-set* $v_0 \overline{\ldots} v_p$ *there is a minimum* $0 \leq t \leq p$ *and a maximum* $0 \leq l \leq p$ *such that*:

1. $scc(v_p) \cap \{v_0, \ldots, v_p\} = \{v_t, \ldots, v_p\}$ *and* $scc(v_0) \cap \{v_0, \ldots, v_p\} = \{v_0, \ldots, v_l\}$;
2. *If* $t \neq 0$, *then* $v_0 \overline{\ldots} v_{t-1}$ *and* $\{v_{t-1}\} = \mathsf{pre}(scc(v_p) \cap \{v_0, \ldots, v_p\}) \cap \{v_0, \ldots, v_p\}$;
3. *If* $l \neq p$, *then* $v_{l+1} \overline{\ldots} v_p$.

Proof We prove Item 1. Let $scc(v_p)$ to contain the spine-set node v_j. We prove that, for all $j < k < p$, $v_k \in scc(v_p)$. Let $j < k < p$. By definition of spine-set, there is a path from v_k to v_p. By $v_j \in scc(v_p)$, there is a path from v_p to v_j and hence, using again the definition of spine-set, there is a path from v_p to v_k.

An analogous reasoning schema allows us to conclude that there exists a maximum $0 \leq l \leq p$ such that $scc(v_0) \cap \{v_0, \ldots, v_p\} = \{v_0, \ldots, v_l\}$.

Item 2 follows from Item 1, from definition of spine-set, and from the fact that each subpath of a chordless path is a chordless path.

Item 3 follows from Item 1 and from the fact that each subpath of a chordless path is a chordless path. □

By the above lemma, in a digraph the nodes of a spine-set can be assigned to their strongly connected components in the order induced on them by the spine-set. In the next section we state an analogous result in the framework of the biconnected components of an undirected graph. This does not come by chance: it depends on the fact that the order induced by a spine-set gives information on the *depth* of the

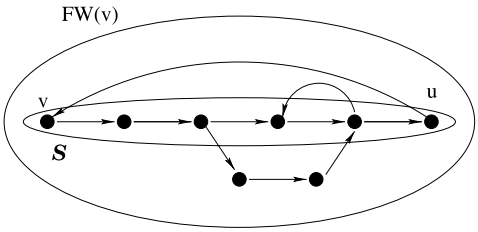


Fig. 3 A Skeleton $\langle \mathcal{S}, u \rangle$ in $FW(v)$

spine-set nodes in a path of the graph. Both the symbolic connectivity algorithms presented in the rest of this paper use spine-sets to *drive* the computation on opportune breadth-first discovered scc-closed (bcc-closed) subgraphs. Thus, although the vertex set is always explored in a breadth-first manner, globally the strongly connected components (biconnected components) are produced in a *piecewise depth-first order*.

In the symbolic scc-algorithm, the scc-closed sets computed are forward-sets of a spine-anchor. Some important properties about such sets are stated in Lemma 5.

Lemma 5 *Let $\langle \mathcal{S}, u \rangle$ be a spine-set and $scc(\langle \mathcal{S}, u \rangle) = \bigcup_{w \in \mathcal{S}} scc(w)$. Then, $FW(u) \cap scc(\langle \mathcal{S}, u \rangle) = scc(u)$.*

Proof Clearly $FW(u) \cap scc(\langle \mathcal{S}, u \rangle) \supseteq scc(u)$. To prove the opposite inclusion, notice that since both $FW(u)$ and $scc(\langle \mathcal{S}, u \rangle)$ are scc closed, $FW(u) \cap scc(\langle \mathcal{S}, u \rangle)$ is scc closed. Let w be such that $scc(w) \subseteq FW(u) \cap scc(\langle \mathcal{S}, u \rangle)$. We prove that $scc(w) = scc(u)$. From the fact that $scc(w) \subseteq FW(u)$ we have that w is reachable from u. By $scc(w) \subseteq scc(\langle \mathcal{S}, u \rangle)$, there exists w' such that $w' \in \mathcal{S}$ and $scc(w) = scc(w')$. u is reachable from w' since $w' \in \mathcal{S}$ and $\langle \mathcal{S}, u \rangle$ is a spine-set. By $scc(w) = scc(w')$, we conclude that u is reachable from w. □

The notion of *skeleton of a forward set*, introduced below, relates spine-sets and forward-sets. In particular, a skeleton is a spine-set that will be used to drive the computation, in our symbolic scc-algorithm.

Definition 9 (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.

Example 3 In Fig. 3 we represent a skeleton of a forward-set.

Lemma 6 *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 $G = \langle V, E \rangle$.*

Proof It immediately follows from the definition of spine-set. □

4.2 Spine-sets and Biconnected Components

The results in the previous section establish a relationship over digraphs among spine-sets, strongly connected components, and forward-sets. Given $G = (V, E)$, in this

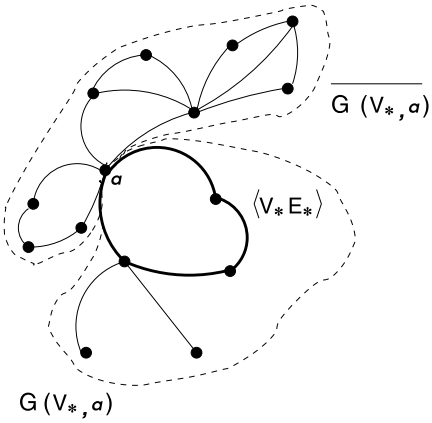


Fig. 4 Graphs $G(V_*, a)$, $\overline{G(V_*, a)}$

section we relate spine-sets, biconnected components, and the bcc-closed subgraphs introduced in Definition 10. More specifically Definition 10, below, introduces a canonical way of splitting G into two bcc-closed subgraphs. Consider an articulation point a. By Lemma 2 there are at least two biconnected components containing a. Our splitting will take place around one of them, say (V_*, E_*), with $a \in V_*$. As depicted in Fig. 4, one of the two subgraphs into which G gets split contains (V_*, E_*) (and it is called $G(V_*, a)$). The other subgraph includes all the remaining biconnected components containing a (and is called $\overline{G(V_*, a)}$). We denote by $\overset{G}{\rightsquigarrow}$ (the nodes in) a simple path in the graph G.

Definition 10 Let (V_*, E_*) be a biconnected component of $G = (V, E)$. If $a \in V_*$ is an articulation point of G, then $G(V_*, a)$ and $\overline{G(V_*, a)}$ are the subgraphs of G induced by the sets of nodes W and $(V \setminus W) \cup \{a\}$, respectively, where:

$$W = \{a\} \cup \{v | \exists \overset{G}{\rightsquigarrow} (a \overset{G}{\rightsquigarrow} v \wedge \overset{G}{\rightsquigarrow} \cap (V_* \setminus \{a\}) \neq \emptyset\}.$$

Lemma 7 *Consider a biconnected component (V_*, E_*) of $G = (V, E)$. If $a \in V_*$ is an articulation point of G, then*:

1. *The two subgraphs $G(V_*, a)$ and $\overline{G(V_*, a)}$ are bcc closed*;
2. *(V_*, E_*) is a subgraph of $G(V_*, a)$.*

Proof Let W be the vertex set of $G(V_*, a)$. There cannot be any edge (w, s) in G with $w \in W \setminus \{a\}$ and $s \in V \setminus W$, otherwise there would be a simple path traversing $V_* \setminus \{a\}$ from a to $s \in V \setminus W$. Hence, assume by contradiction that there exists a biconnected component in G, (V', E'), such that $V' \cap (W \setminus \{a\}) \neq \emptyset$ and $V' \cap (V \setminus W) \neq \emptyset$. As (V', E') is biconnected, for all z, t with $z \in V' \cap (W \setminus \{a\}), t \in V \setminus W$, there exists a path between z and t not containing a. We get to the contradiction that there exists an edge between a node in $W \setminus \{a\}$ and a vertex in $V \setminus W$. Item 2 directly follows from Definition 10. □

Lemma 8 is the counterpart of Lemmas 4 and 5 stated in the previous section for strongly connected components.

Lemma 8 *Given* $\overline{v_0 \dots v_p}$ *in* $G = (V, E)$, *let* V_* *be the vertex set of a biconnected component containing* v_p. *There is a minimum* $0 \leq t \leq p$ *such that*:

1. $V_* \cap \{v_0, \dots, v_p\} = \{v_i \mid t \leq i \leq p\}$;
2. *If* v_t *is an articulation point, then* $\overline{v_0 \dots v_t}$ *is a spine-set in* $\overline{G(V_*, v_t)}$;
3. *If* $a \neq v_t$, $a \in V_*$ *is an articulation point and* V' *is the vertex set of* $\overline{G(V_*, a)}$, *then* $V' \cap \{v_0, \dots, v_p\} \subseteq \{a\}$.

Proof Let us prove Item 1. By contradiction, suppose that there exists a sub-path of the chordless path $(v_1, \dots, v_p)$, $p = (v_i, \dots, v_j)$ with $|j - i| > 1$, in which the endpoints v_j and v_i belong to V_* and all the internal nodes do not belong to V_*. As V_* is connected, V_* contains a path from v_i to v_j, say q. By concatenating the paths p and q we obtain a simple cycle containing $v_{i+1}, \dots, v_{j-1}$. This ensures that all the nodes in p belongs to V_* and contradicts our hypothesis.

Item 2 immediately follows from the fact that each prefix of a chordless path is a chordless path.

Item 3. By Definition of v_t, since $a \in V^*$, a cannot be a v_i for $i < t$. Moreover, since $v_t \in V^*$ and a is an articulation point, from Definition 10 it follows that $\{v_0 \dots v_t\} \subseteq G(V^*, a)$. Hence $\{v_0 \dots v_p\} \subseteq G(V^*, a)$. Being a the unique vertex in common to $G(V^*, a)$ and $\overline{G(V^*, a)}$, we conclude that $\{v_0 \dots v_p\} \cap V' \subseteq \{a\}$. □

A *skeleton of* $\overline{G(V_*, a)}$, introduced below, is a particular spine-set in $\overline{G(V_*, a)}$ that will drive the computation in our symbolic bcc-algorithm.

Definition 11 (Skeleton of $\overline{G(V_*, a)}$) Consider $G = (V, E)$, a biconnected component (V_*, E_*) in G and an articulation point $a \in V_*$. $\langle \mathcal{S}, u \rangle$ is a skeleton of $\overline{G(V_*, a)}$ iff u is a node in $\overline{G(V_*, a)}$ having maximum distance from a and $\mathcal{S}$ is the set of nodes on a shortest path from a to u.

Lemma 9 *If* $\langle \mathcal{S}, u \rangle$ *is a skeleton of* $\overline{G(V_*, a)}$, *then* $\langle \mathcal{S}, u \rangle$ *is a spine-set in* $\overline{G(V_*, a)}$.

Proof It immediately follows from the definition of spine-set. □

5 Strong Connectivity on Symbolic Graphs

In this section we show how the notions introduced in Sects. 4 and 4.1 allow us to design a symbolic scc-algorithm performing a linear number of symbolic steps. Similarly to the other scc-algorithms in the literature, our algorithm does not provide as output the strongly connected component subgraphs, but their vertex sets, i.e., if $\langle V_1, E \restriction V_1 \rangle, \dots, \langle V_k, E \restriction V_k \rangle$ are the strongly connected components of $\langle V, E \rangle$ our output is $V_1, \dots, V_k$. From now on with a slight abuse of notation we refer to the sets $V_1, \dots, V_k$ as strongly connected components. We start by giving some intuitions

about the procedure we are going to present. In each iteration the strongly connected component of a node, v, is simply determined by first computing $FW(v)$ and then identifying those vertices 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 order associated to an opportune spine-set. More specifically, whenever a forward-set $FW(v)$ is built, a skeleton of such a forward-set, $v_0 \overline{\ldots} v_p$ with $v = v_0$, is also computed. The order induced by the skeleton is then used for the subsequent computations. Stated in other words, $scc(v_p)$ will be the first strongly connected component isolated in the scc-closed subset $FW(v) \setminus scc(v)$. In this way, the cost of the symbolic steps performed to produce $FW(v)$ is distributed over the computation of the strongly connected components of the nodes in the skeleton of $FW(v)$. This amortized analysis is the key point for the linear complexity of the algorithm.

5.1 The Linear Symbolic SCC Algorithm

With the above intuition, we outline in Table 1 the pseudo-code for our linear symbolic scc-algorithm. The parameters of the procedure depicted in Table 1 are a graph $G = \langle V, E \rangle$ and a pair $\langle \mathcal{S}, N \rangle$. $\langle \mathcal{S}, N \rangle$ is either $\langle \emptyset, \emptyset \rangle$ or $\mathcal{S} = \{v_0, \ldots, v_p\} \subseteq V$, $N = \{v_p\}$, with $v_0 \overline{\ldots} v_p$ (i.e., $\langle \{v_0, \ldots, v_p\}, v_p \rangle$ 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 to be computed is chosen.

In case $\mathcal{S} \neq \emptyset$ and $N = \{v_p\}$, v_p is chosen. Otherwise ($\mathcal{S} = \emptyset$) an arbitrary element $v \in V$ is picked (assigning the singleton $\{v\}$ to N). Then the subprocedure SKEL-FORWARD is called to compute the forward-set of the singleton N and 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 $newN$ maintain $\mathcal{S}'$ and $\{u'\}$, respectively. In line (6) the local variable SCC is initialized to be the singleton N and then it is augmented with the elements of the strongly connected component containing N (loop of lines (7), (8)). In line (9) 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 $\langle \mathcal{S}, N \rangle$ by subtracting SCC (cf. Item 2 in Lemma 4);
2. The subgraph of $\langle V, E \rangle$ induced by $FW \setminus SCC$ and the spine-set of such a subgraph obtained from $\langle new\mathcal{S}, newN \rangle$ by subtracting SCC (cf. Item 3 in Lemma 4).

In Table 2 the pseudo-code of the subprocedure SKEL-FORWARD is presented. It is used during the execution of SYMBOLIC-SCC 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 $N = \{v\} \subseteq V$. The forward-set of the node given as input, v, is simply computed with a symbolic breadth-first search [31, 37], i.e., by a loop that discovers, in each iteration i, all the nodes of V having distance i from v. In [31] the just mentioned sets, that are often referred as *onion-rings* [31, 37] in the verification area, 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 $FW(v)$.

Table 1 The scc-algorithm performing a linear number of symbolic steps

SYMBOLIC-SCC($V, E, \langle \mathcal{S}, N \rangle$)

(1) **if** $V = \varnothing$
(2) **then return**;

▷ *Determine the node for which the scc is computed*
(3) **if** $\mathcal{S} = \varnothing$
(4) **then** N $\leftarrow$ *pick*(V);

▷ *Compute the forward-set of the singleton N and a skeleton*
(5) $\langle FW, new\mathcal{S}, newN \rangle \leftarrow$ SKEL-FORWARD(V, E, N);

▷ *Determine the scc containing N*
(6) $SCC \leftarrow N$;
(7) **while** $((\mathsf{pre}(SCC) \cap FW) \setminus SCC) \neq \varnothing$ **do**
(8) $SCC \leftarrow SCC \cup (\mathsf{pre}(SCC) \cap FW)$;

▷ *Insert the scc in the scc-Partition*
(9) SCC-Partition $\leftarrow SCC$-Partition $\cup \{SCC\}$

▷ *First recursive call: computation of the scc's in* $V \setminus FW$
(10) $V' \leftarrow V \setminus FW$; $E' \leftarrow E \restriction V'$;
(11) $\mathcal{S}' \leftarrow \mathcal{S} \setminus SCC$; $N' \leftarrow \mathsf{pre}(SCC \cap \mathcal{S}) \cap (\mathcal{S} \setminus SCC)$;
(12) SYMBOLIC-SCC($V', E', \langle \mathcal{S}', N' \rangle$)

▷ *Second recursive call: computation of the scc's in* $FW \setminus SCC$
(13) $V' \leftarrow FW \setminus SCC$; $E' \leftarrow E \restriction V'$;
(14) $\mathcal{S}' \leftarrow new\mathcal{S} \setminus SCC$; $N' \leftarrow newN \setminus SCC$;
(15) SYMBOLIC-SCC($V', E', \langle \mathcal{S}', N' \rangle$)

Notice that the restriction of the edge relation in the two recursive calls is introduced only for the sake of clarity. As done in [4, 5] the results of the image and pre-image computations can be intersected with V', so that only the original edge relation needs to be stored.

5.2 Correctness and Complexity

The soundness and completeness of the algorithm in Table 1 are stated in Theorems 1 and 2, respectively. Throughout this section we will use the notation V_c^l to indicate the variable V on the entering of line l in the c-th execution of SYMBOLIC-SCC. V_c will denote the value of the parameter V to the c-th call to SYMBOLIC-SCC. An analogous notation will be adopted for all the variables in SYMBOLIC-SCC. Lemma 10, below, states that the subprocedure SKEL-FORWARD($V, E, \{v\}$) computes the forward-set of v and a skeleton of $FW(v)$.

Table 2 The procedure which computes the forward-set of a node and a skeleton

SKEL-FORWARD (V, E, N)

(1) Let `stack` be an empty stack of sets of nodes
(2) $L \leftarrow N$

▷ *Compute the Forward-set of N and push onto* `stack` *the onion rings*
(3) **while** $(L \neq \emptyset)$ **do**
(4) push(`stack`, L);
(5) $FW \leftarrow FW \cup L$;
(6) $L \leftarrow \mathsf{post}(L) \setminus FW$;

▷ *Determine a Skeleton of the Forward-set of N*
(7) $L \leftarrow \mathsf{pop}(\texttt{stack})$;
(8) $\mathcal{S}' \leftarrow N' \leftarrow \mathsf{pick}(L)$;
(9) **while** `stack` $\neq \emptyset$ **do**
(10) $L \leftarrow \mathsf{pop}(\texttt{stack})$;
(11) $\mathcal{S}' \leftarrow \mathcal{S}' \cup \mathsf{pick}(\mathsf{pre}(\mathcal{S}') \cap L)$;
(12) **return**$\langle FW, \mathcal{S}', N' \rangle$;

Lemma 10 *Let* $G = \langle V, E \rangle$. *Given* $v \in V$, SKEL-FORWARD$(V, E, \{v\})$ *returns the triple of sets* $\langle FW(v), \mathcal{S}', \{u\} \rangle$ *where* $\langle \mathcal{S}', u \rangle$ *is a skeleton of* $FW(v)$.

Proof Upon the termination of each iteration, i, of the first loop in SKEL-FORWARD, it holds that:

- L maintains the set of nodes at distance i from $\{v\}$;
- L is pushed onto the stack of sets of nodes, `stack`, and it is merged with FW.

It follows that the first loop in SKEL-FORWARD is executed d_v times, where d_v is the maximum distance, from v, of a vertex in V.

Upon termination of this loop, FW maintains the forward-set of $\{v\}$. Moreover, `stack` has length d_v and w belongs to the set in position i of `stack` if and only if w has distance i from v. Thus line (8) assigns, to both N' and $\mathcal{S}'$, a singleton $\{u\}$, where u is a node having maximum distance from v. Then, the while loop of lines (9)–(11) augments $\mathcal{S}'$ with the set of nodes on a minimum distance path from v to u. By Definition 9 we conclude that $\langle \mathcal{S}', u \rangle$ is a skeleton of $FW(v)$. □

Lemma 11 *Let* $G = \langle V, E \rangle$. *Consider the execution of* SYMBOLIC-SCC *on* $(V, E, \langle \emptyset, \emptyset \rangle)$. *In each recursive call*, c, *to* SYMBOLIC-SCC:

(a) $\langle V_c, E_c \rangle$ *is a subgraph of* $\langle V, E \rangle$ *and* V_c *is scc closed*;
(b) $N_c^5 = \{v\} \subseteq V$ *and if* $\mathcal{S}_c \neq \emptyset$, *then* $\langle \mathcal{S}_c, N_c \rangle$ *is a spine-set of* $\langle V_c, E_c \rangle$;
(c) $scc_G(v)$ *is assigned to* SCC_c^9 *and* $\langle SCC_c^9, V_c^9 \setminus FW_c^9, FW_c^9 \setminus SCC_c^9 \rangle$ *is a partition of* V_c.

Proof By induction on the number of recursive calls to SYMBOLIC-SCC within SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$. The base case is immediate. We assume our lemma true for the first c recursive calls and we sketch here the inductive step. Let SYMBOLIC-SCC$(V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'} \rangle)$ be the recursive call from which the $(c+1)$-st call to SYMBOLIC-SCC is executed (hence $c' \leq c$).

Inductive Step (Item (a)) V^9_{c+1} is either $V^9_{c'} \setminus FW^9_{c'}$ or $FW^9_{c'} \setminus SCC^9_{c'}$. Hence, by inductive hypothesis, $\langle V_{c+1}, E_{c+1} \rangle$ is a subgraph of $\langle V, E \rangle$ and V_{c+1} is a scc-closed subset of V.

Inductive Step (Item (b)) The values of $\mathcal{S}_{c+1}$ and N_{c+1} depend on whether the $(c+1)$-st call to SYMBOLIC-SCC corresponds to the first or to the second recursive call within SYMBOLIC-SCC$(V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'} \rangle)$. In the first case, if $\mathcal{S}_{c'} = \emptyset$, then line (11) initializes $\langle \mathcal{S}^{c+1}, N^{c+1} \rangle$ to $\langle \emptyset, \emptyset \rangle$. Otherwise, by inductive hypothesis, $\langle \mathcal{S}_{c'}, N_{c'} = \{v\} \rangle$ is a spine-set in $\langle V_{c'}, E_{c'} \rangle$ and $SCC^9_{c'} = scc_G(v)$. Hence, by Lemma 4 (Item 2) and Lemma 5, lines (10)–(11) in SYMBOLIC-SCC initialize $\langle \mathcal{S}_{c+1}, N_{c+1} \rangle$ to a spine-set in $\langle V_{c+1}, E_{c+1} \rangle$. In the second case, the inductive hypothesis, Lemma 4 (Item 3), Lemma 6, and Lemma 10 ensure that lines (13)–(14) in SYMBOLIC-SCC initialize $\langle \mathcal{S}_{c+1}, N_{c+1} \rangle$ to a spine-set in $\langle V_{c+1}, E_{c+1} \rangle$.

Inductive Step (Item (c)) By Lemma 10 and by inductive hypothesis, we have that the strongly connected component of v in G is assigned to SCC^9_{c+1} and the forward-set of v in $\langle V_{c+1}, E_{c+1} \rangle$ is assigned to FW^9_{c+1}. Thus, $\langle SCC^9_{c+1}, V^9_{c+1} \setminus FW^9_{c+1}, FW^9_{c+1} \setminus SCC^9_{c+1} \rangle$ is a partition of V_{c+1}. □

Theorem 1 (Soundness) *Consider $G = \langle V, E \rangle$. If $SCC \subseteq V$ is added to SCC-Partition within* SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$, *then SCC is a scc of G.*

Proof It directly follows from Lemma 11 and Lemma 1. □

Theorem 2 (Completeness) *Let $G = \langle V, E \rangle$. If $v \in V$, then upon termination of* SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$, *all the scc's of G belong to SCC-Partition.*

Proof By Lemma 11, in each recursive call, SYMBOLIC-SCC$(V_c, E_c, \langle \mathcal{S}_c, N_c \rangle)$, within SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$:

- $V_c \subset V$;
- A strongly connected component of G, $scc(v)$, is computed;
- SYMBOLIC-SCC is called over $(V', E', \langle \mathcal{S}', N' \rangle)$ and $(V'', E'', \langle \mathcal{S}'', N'' \rangle)$, where $\langle V', V'', scc(v) \rangle$ is a partition of V.

The thesis immediately follows. □

Lemma 12, below, shows that each node can be inserted at most two times into a spine-set, within the execution of SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$. Lemma 12 is preliminary to Theorem 3 assessing the complexity of SYMBOLIC-SCC.

Lemma 12 *Consider $G = \langle V, E \rangle$. Within* SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$, *each $v \in V$ is inserted at most two times into a spine-set.*

Proof Consider the c-th recursive call, SYMBOLIC-SCC($V_c, E_c, \langle \mathcal{S}_c, N_c \rangle$), within SYMBOLIC-SCC($V, E, \emptyset, \emptyset$). We proceed by induction on the number of recursive calls and we prove that, $\forall v \in V_c$:

(a) If $v \in V_c \setminus \mathcal{S}_c$, then v has never been inserted into a spine-set in the first $c-1$ recursive calls;
(b) If $v \in \mathcal{S}_c$, then v has been inserted exactly once into a spine-set;
(c) In lines (1)–(9) of SYMBOLIC-SCC($V_c, E_c, \langle \mathcal{S}_c, N_c \rangle$), $v \in V_c$ is inserted into a spine-set at most once;
(d) If $v \in \mathcal{S}_c$ is inserted into a spine-set in lines (1)–(9) of SYMBOLIC-SCC($V_c, E_c, \langle \mathcal{S}_c, N_c \rangle$), then $SCC_c^9 = scc(v)$.

The thesis immediately follows from Items (a)–(d). We sketch here the proof of the inductive step for Items (a)–(d) (the base case is immediate).

Consider $v \in V_c$, with $c > 1$. Item (c) follows from the fact that a node can be inserted into a spine-set only in line (4) of SKEL-FORWARD; it follows from Lemma 10 that the sets considered on such lines are mutually disjoint.

To prove the inductive step for Items (a)–(b), assume that the c-th recursive call to SYMBOLIC-SCC was called by the c'-th execution of the procedure. If $c = c' + 1$, then, by Lemma 11, $V_c = V_{c'} \setminus FW(u)$ and $\mathcal{S}_c = \mathcal{S}_{c'} \setminus scc(u)$, where $\{u\} = N_{c'}^5$. Hence, Items (a)–(b), for the inductive step, directly follow from the inductive hypothesis and from the fact that the nodes inserted into a spine-set, within the c'-th execution of SKEL-FORWARD, belong to $FW(u)$. Otherwise ($c > c' + 1$), the c-th execution of SYMBOLIC-SCC corresponds to the second recursive call within SYMBOLIC-SCC($V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'} \rangle$)). In this case, note that the two sets of vertices considered by the two recursive calls within SYMBOLIC-SCC($V_{c'}, E_{c'}, \langle \mathcal{S}_{c'}, N_{c'} \rangle$) are disjoint. Hence, for all c'' with $c' < c'' < c$, v cannot be inserted into a spine-set within SYMBOLIC-SCC($V_{c''}, E_{c''}, \langle \mathcal{S}_{c''}, N_{c''} \rangle$) (as $v \notin V_{c''}$). By Lemma 4, $FW(N_{c'}^5) \cap \mathcal{S}_{c'} \subseteq scc(N_{c'}^5)$, i.e., $V_c = FW(N_{c'}^5) \setminus scc(N_{c'}^5)$ is disjoint from $\mathcal{S}_{c'}$. Moreover, the nodes in $\langle \mathcal{S}_c, N_c \rangle$ are nodes inserted into a spine by the c'-th recursive call, and not yet assigned to their scc's . Thus, by inductive hypothesis, we have the validity of Items (a)–(b) for the inductive step.

We finally obtain the inductive step for Item (d). By Lemma 10, if $\langle \mathcal{S}_{c'}, N_{c'} \rangle$ is not an empty spine-set, then $\langle new\mathcal{S}_c^5, newN_c^5 \rangle$ is a spine set in $FW(N_{c'})$. Hence, by Lemma 5, $\mathcal{S}_{c'} \cap new\mathcal{S}_c^5 \subseteq scc(N_c^5)$ and we obtain the validity of Item (d) for the c-th recursive call. □

By the previous lemma, the linear number of symbolic steps performed by SYMBOLIC-SCC is immediate. In fact, each symbolic step in the loop of lines (7)–(8) of SYMBOLIC-SCC assigns a set of nodes to its scc. Hence the global number of these symbolic steps is $\mathcal{O}(|V|)$. The remaining symbolic steps, within SKEL-FORWARD, are proportional to the number of insertions into a spine-set. By Lemma 12 the number of these insertions is bounded by $\mathcal{O}(|V|)$.

Theorem 3 (Complexity) *Let* $G = \langle V, E \rangle$. SYMBOLIC-SCC($V, E, \langle \emptyset, \emptyset \rangle$) *runs in* $\mathcal{O}(|V|)$ *symbolic steps*.

Proof Lines (1)–(4) and (9)–(15) of SYMBOLIC-SCC perform a constant number of symbolic steps. Since each recursive call produces a strongly connected component, lines (1)–(4) and (9)–(15) of SYMBOLIC-SCC perform, globally, $\mathcal{O}(|V|)$ symbolic steps. Each symbolic step in the loop of lines (7)–(8) in the main procedure assigns a set of nodes to its scc. Hence the global number of these symbolic steps is $\mathcal{O}(|V|)$. The remaining symbolic steps are performed by the procedure SKEL-FORWARD. Within an execution of SKEL-FORWARD, the number of symbolic steps is proportional to the number of sets popped from the stack upon the second loop termination. Whenever a set is popped from the stack, an insertion into a spine is performed. Hence, by Lemma 12, the global cost (in term of symbolic steps) of executing SKEL-FORWARD within SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$, is linear in the size of V. □

Corollary 1, below, strengthens the complexity result stated in Theorem 3. In particular, rather than considering only symbolic steps, all OBDD operations are considered and parameters such as the diameter of the graph and the number of strongly connected components are used in the analysis.

Corollary 1 SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$ *requires* $\mathcal{O}(\min(|V|, dN))$ *symbolic operations, where* d *is the diameter of* $G = \langle V, E \rangle$ *and* N *is the number of scc's in* G.

Proof Each recursive call to SYMBOLIC-SCC produces a new strongly connected component, hence the number of recursive calls is bounded by N. We prove that each recursive call requires at most $\mathcal{O}(d)$ symbolic steps. By an inductive argument on the number of recursive calls we can easily show that for each recursive call, SYMBOLIC-SCC$(V_c, E_c, \langle \mathcal{S}_c, N_c \rangle)$, $d_c \leq d$, where d_c is the diameter of $\langle V_c, E_c \rangle$. The base case is immediate, for the inductive step there are two cases to be considered:

1. $V_{c+1} = FW_{\langle V_c, E_c \rangle}(v), v \in V_c$;
2. $V_{c+1} = V_c \setminus FW_{\langle V_c, E_c \rangle}(v), v \in V_c$.

In both cases it can be easily proved that, if $a, b \in V_{c+1}$ and z is a node in a path of $\langle V_c, E_c \rangle$ from a to b, then $z \in V_{c+1}$. From the above property it immediately follows that $d_{c+1} \leq d_c$ since, given any pair of nodes, $a, b \in V_{c+1}$, each path connecting a and b in $\langle V_c, E_c \rangle$ belongs to $\langle V_{c+1}, E_{c+1} \rangle$.

Since the main loops in SYMBOLIC-SCC and SKEL-FORWARD roughly perform a reachability analysis, a bound of $\mathcal{O}(d)$ symbolic steps for each recursive call to SYMBOLIC-SCC immediately follows. A global bound of $\mathcal{O}(dN)$ symbolic steps can be further derived and strengthened, by Theorem 3, to $\mathcal{O}(\min(|V|, dN))$ symbolic steps.

We finally observe that each iteration of each loop both in SYMBOLIC-SCC and in SKEL-FORWARD contains at least one symbolic step. Hence, the global number of OBDD operations in SYMBOLIC-SCC is asymptotically bounded by the global number of symbolic steps in SYMBOLIC-SCC. We conclude that SYMBOLIC-SCC$(V, E, \langle \emptyset, \emptyset \rangle)$ requires $\mathcal{O}(\min(|V|, dN))$ symbolic operations. □

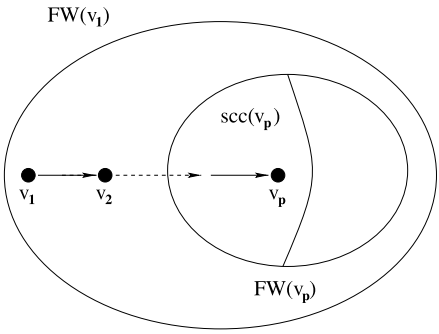


Fig. 5 Relative sizes of the computed forward-sets

Similar results have been proved in [4, 5] where it is shown that the complexity of their algorithm is $\mathcal{O}(\min(|V|\log(V), dN))$ while the complexity of the algorithm presented in [45] can be bounded by $\mathcal{O}(\min(|V|^2, dN))$.

Figure 5 shows the relative sizes of the computed forward-sets and scc's. $FW(v_1)$ and $scc(v_1)$ are already computed when SYMBOLIC-SCC$(V, E, \{v_l, \ldots, v_p\}, v_p)$ is called and subsequently, $FW(v_p)$ is computed and $scc(v_p)$ is given in output.

We conclude this section by observing that several heuristics to optimize the implicit algorithm in Table 1 are possible. In particular, as done in [4, 5, 31], the set $T \subseteq V$ of nodes belonging to trivial scc's which do not reach any non trivial scc could be quickly determined by the following fix-point pre-computation:

$$\text{while } (V \neq pre(V)) \text{ do } \{V \leftarrow pre(V); T \leftarrow (V \setminus pre(V)) \cup T\}.$$

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 in the main procedure), would make the algorithm in Table 1 somehow "more symbolic".

6 Biconnectivity on Symbolic Graphs

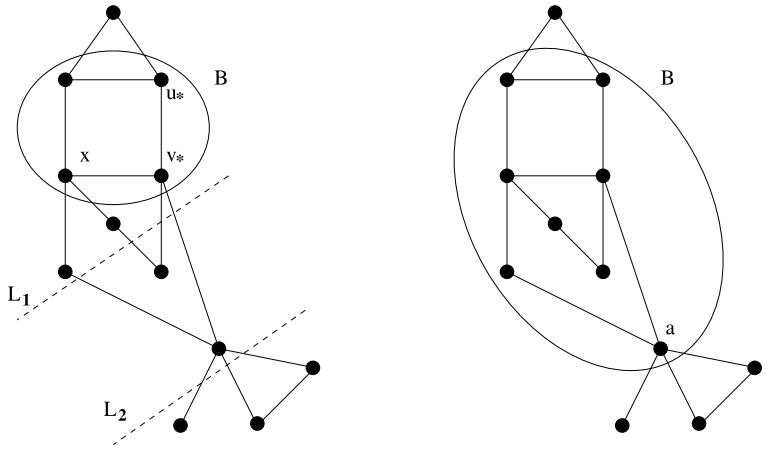
In this section we focus on biconnectivity. Relying again on spine-sets, we outline a symbolic bcc-algorithm performing a linear number of symbolic steps for computing biconnected components. The algorithm we propose uses a rather simple strategy to find the nodes of each biconnected component in an OBDD-represented graph $G = (V, E)$. Yet, this strategy would have a quadratic performance (in the size of the set of vertices) if not properly combined with the notion of spine-set. Note how this is perfectly symmetric to what happens in the symbolic scc-algorithm in Sect. 5.

Given an edge $(u_*, v_*) \in E$, let V_* be the vertex set of the biconnected component containing the edge $(u_*, v_*) \in E$. The strategy for building V_* relies on extending the set of vertices B, initialized as $B = \{u_*, v_*\}$, maintaining the invariant below:

$$\forall v \in B(v \in \{u_*, v_*\} \vee \exists \text{ a cycle linking } v, u_*, \text{ and } v_* \text{ in B}). \tag{1}$$

Invariant (1) ensures that $B \supseteq \{u_*, v_*\}$ induces a biconnected subgraph with $B \subseteq V_*$. Under the above mentioned invariant, a safe increasing of B is obtained

Table 3 Augmenting B under invariant (1)



by adding to B all nodes on simple paths between two nodes of B (simple paths *reaching back* B). The search for these paths could naturally take place from a node $x \in B$ linked to some node outside B: an *exploration point* for B.

We depict in Table 3 a successful attempt of augmenting $B \subseteq V_*$ looking for paths that source from the *exploration point* $x \in B$, cross $V \setminus B$, and terminate in a node of B other than x. First, the set L_1 containing the successors of $\{x\}$ in $V \setminus B$ is computed and then levels at increasing distance from L_1 are discovered. If a level L_p intersecting $B \setminus \{x\}$ is encountered, this guarantees the existence of at least one simple path between two nodes of B.

If the overall process allows discovering only vertices outside B or equal to the exploration point, the attempt to grow $B \supseteq \{u_*, v_*\}$ fails. In this case, invariant (1) ensures that the *exploration point* involved in the visit is an articulation point of G. This situation is sketched in the bottom of Table 3, if we choose the node a as the next *exploration point* for B. Always in Table 3, the set of nodes that can be retrieved using a as exploration point is the vertex set of the bcc-closed subgraph $\overline{G(V_*, a)}$. This set of nodes can be ignored while extending B to compute V^*, but it can be used to localize the (recursive) computation of subsequent biconnected components.

Summarizing, the above ideas lead to a recursive procedure in which the process of building the vertex set of each biconnected component (V_*, E_*) roughly results in subsequent breadth-first visits from a node of $B \subseteq V_*$. Each visit either augments $B \subseteq V_*$ or discovers a bcc-closed subgraph not containing (V_*, E_*) on which a recursive call can be made.

The problem, with the above approach, is that the global number of symbolic steps performed exploring the graph from the articulation points, is $\Theta(|V|^2)$ in the worst case.

Spine-sets allow us to discover the biconnected components in a depth-first search order and hence to have a linear global number of symbolic steps. The way spines are involved in the strategy described above is the following. Whenever the visit from an exploration point $\{a\}$ of B results in a set of p levels outside B, these levels are

used to build a skeleton of the graph $\overline{G(V_*, a)}$. This skeleton, say $\overline{av_1 \dots v_p}$, keeps track of the number of symbolic steps necessary to compute $\overline{G(V_*, a)}$. The order implicitly maintained in it can be used for subsequent computation, so that the cost of computing $\overline{G(V_*, a)}$ is amortized onto the cost of assigning each node in $\overline{av_1 \dots v_p}$ to its biconnected components. The linear performance of the full algorithm is a consequence of the following two facts:

1. The spine-driven order in which biconnected components are computed ensures that the global number of symbolic steps necessary to obtain the subgraphs for the recursive calls is $\mathcal{O}(|V|)$ (i.e., the spine-sets generated during the entire algorithm collect at most $\mathcal{O}(|V|)$ nodes);
2. The remaining symbolic steps assign at least one node to its biconnected components.

We conclude this section preamble observing that even though the use of spine-set is central with respect to the computational complexity of both our symbolic scc and bcc algorithms, the latter turns out slightly more complicated.

Example 4 Consider the graph depicted in Table 4, and its biconnected components, $\langle V_1, E_1\rangle$, $\langle V_2, E_2\rangle$, $\langle V_3, E_3\rangle$,$\langle V_4, E_4\rangle$. We illustrate in Tables 5(a)–(c) and 6(d)–(i) the use of *spine-sets* in our algorithm. In the above mentioned tables, we will use the symbol X to mark the nodes which subsequently play the role of exploration points; the white vertices will represent nodes inserted into a spine. In Table5(a) the edge (e, d) is chosen to initialize the vertex set of the first biconnected component. The node d is selected as the exploration point to extend $B = \{e, d\}$. In Table 5(b) the simple path reaching back B, $\langle d, c, b, e\rangle$, is added to B and the only remaining exploration point is the vertex d. There is no simple path reaching back $B = \{d, c, b, e\}$ from d and 5 symbolic steps are necessary to discover $\overline{G(V_1, d)}$: one to determine the only exploration point d and four to determine nodes at distance one, two, three,

Table 4 A graph and its biconnected components

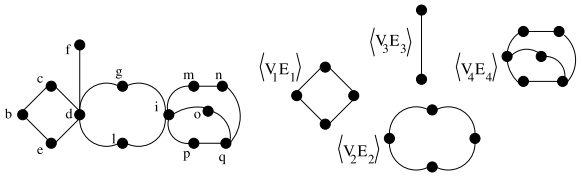


Table 5 Application of the linear symbolic bcc algorithm (part 1)

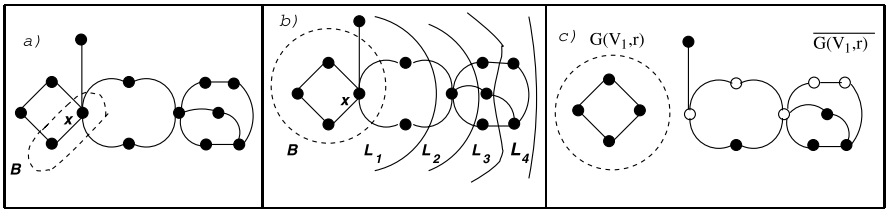
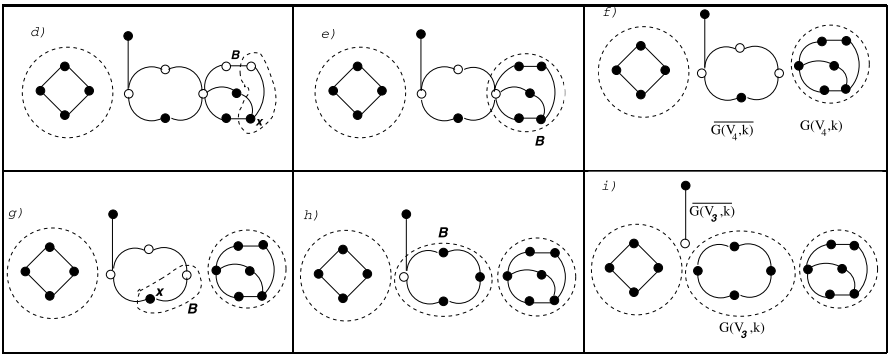


Table 6 Application of the linear symbolic bcc algorithm (part 2)



and four from d, respectively (see Table 5(b)). Thus, a spine-set in $\overline{G(V_1, d)}$ is built by selecting nodes on a path traversing levels L_1, L_2, L_3, L_4. In Table 5(c) a recursive call to the procedure is made over the graph $\overline{G(V_1, d)}$ coupled with its spine $\langle\{d, g, i, m, n\}, \{n\}\rangle$ and the computation of the biconnected component containing $\{d, c, b, e\}$ is localized onto $G(V_1, d)$ (finishing immediately because there are no exploration points). Tables 6(d)–(i) show how subsequent biconnected components computation is driven by the spine-set generated. In Table 6(d) the edge (n, q), whose endpoint n is the spine-anchor, is used to initialize the vertex set of a new biconnected component, $B = \{n, q\}$. The node q is chosen as the exploration-point. Note that, during the subsequent processing, the spine-set represents a chordless path having in common with B only the last node (until it gets empty). Moreover, we are careful to choose an exploration point other than the spine-anchor and we extend B in two phases: first, we detect simple paths reaching back either B or a node in the spine; then we extend B with the maximum "spine suffix" which is a simple path reaching back B. These technicalities ensure that, once a node is inserted in a spine-set, it is involved in some symbolic step if and only if it is assigned to a biconnected component. Note that, without the latter property, we would not be able to guarantee that a global linear number of symbolic steps is sufficient to manipulate spine-sets (i.e. to obtain the subgraphs for the recursive calls). In Table 6(e), B is augmented with the nodes internal to the paths $\langle n, q, o, i\rangle$ and $\langle n, q, p, i\rangle$ between the exploration point n and the spine-anchor i. The spine suffix $\{i, m, n\}$ is also added to B and the spine-set is updated to $\langle\{d, g, i\}, \{i\}\rangle$. At this point, there are no further potential exploration points other than the spine-anchor. This ensures that B is the vertex set of a biconnected component and that the spine-anchor, i, is an articulation point. Moreover the remaining spine is a spine in $\overline{G(V_4, k)}$. Thus, in Table 6(f) a recursive call on $\overline{G(V_4, k)}$ coupled with the spine-set $\langle\{d, g, i\}, \{i\}\rangle$ is made. During this recursive call the biconnected component $\langle V_2, E_2\rangle$ is isolated. The last recursive call produces $\langle V_3, E_3\rangle$.

Example 4 should suggest how the use of spine-sets allows us to avoid rediscovering again and again the same $\circlearrowleft$-closed subgraphs while isolating each biconnected components. In particular in the proposed example, the $\circlearrowleft$-closed subgraph having vertices $\{i, m, n, o, p, q\}$ would be explored three times (from some

exploration point) if the biconnected components were produced in the order $\langle V_1, E_1\rangle, \langle V_2, E_2\rangle, \langle V_3, E_3\rangle, \langle V_4, E_4\rangle$.

6.1 The Linear Symbolic Biconnectivity Algorithm

The algorithm SYMBOLIC-BCC in Table 7 implements the ideas outlined in the previous section and efficiently retrieves the vertex set of each biconnected component in a symbolically represented graph.

SYMBOLIC-BCC is a recursive procedure that takes as input a graph, $G = (V, E)$, and a pair of sets of nodes $\langle \mathcal{S}, N\rangle$. The pair $\langle \mathcal{S}, N\rangle$ is either composed of empty sets or $\mathcal{S} = \{v_0, \ldots, v_p\}$ and $N = \{v_p\}$, with $\overline{v_0 \ldots v_p}$ a spine in G. If $V \neq \emptyset$, lines (1)–(4) initialize the vertex set of a biconnected component, B, to the endpoints of an edge, $\{v_*, u_*\}$. If the spine is not empty, then one of the endpoints of this edge is the spine-anchor. Line (5) updates the spine to $\overline{v_0 \ldots v_{p-1}}$ in the case in which $B = \{v_{p-1}, v_p\}$. This ensures the property that the spine represents a chordless path sharing only its last node with B. Maintaining this property allows us to extend B in two phases, involving the nodes in the spine-set only to add them to B. The first phase consists essentially of the procedure VISIT: the detection of paths reaching back $B \cup S$. The procedure EAT-SPINE realizes the second phase. Both the procedures are called within the main loop of SYMBOLIC-BCC in line (7). Each iteration of such a loop operates on a bcc-closed subgraph that contains $B = \{v_*, u_*\}$, whose nodes are maintained in the variable V. Precisely, each loop execution either refines V or augments B. The refinement of V is done in case a bcc-closed subgraph disjoint from B is discovered; the augmentation of B is performed in case paths connecting two nodes in B are retrieved. Hence, in line (8) an exploration point, X, is chosen; the procedure VISIT is then called (line (9)) to explore V from X, obtaining exactly either a set, C, to augment B, or the set of vertices, V', of a bcc-closed subgraph and a spine-set in it $\langle \mathcal{S}', N'\rangle$ (line (9)). In the first case, the else branch of the if statement in line (10) is executed. Otherwise, the then branch is entered and a new recursive call on $\langle V', E \restriction V'\rangle$ is made on line 11. The loop terminates when B contains at most the spine-anchor, N, as a node linked to some vertex outside B. The latter condition ensures that B, plus at most the nodes on a spine-suffix, represents a bcc. If necessary, a spine-suffix is used to complete B in line (14), and the bcc built is finally added to the bcc partition (line (15)). Lines (16)–(17) perform a final conclusive recursive call in case $V \neq B$.

6.1.1 The subprocedure VISIT

The subprocedure VISIT, in Table 8, gets as input the exploration point $X = \{x\}$ selected in line (5) in SYMBOLIC-BCC, the set B, the graph (V, E) and its spine $\langle \mathcal{S}, N\rangle$. The procedure uses a stack of vertex sets to keep track of the levels at increasing distance from the set of nodes, outside $B \cup \mathcal{S}$, linked to the exploration point. Within each iteration of the loop in lines (8)–(10) of VISIT, a new level is pushed onto `stack` while either no new node can be discovered or a level intersecting $B \cup \mathcal{S}$ is detected. In the first case the set of vertices discovered is that of a bcc-closed subgraph used for the next recursive call.

Table 7 The bcc-algorithm performing a linear number of symbolic steps

SYMBOLIC-BCC($V, E, \langle \mathcal{S}, N \rangle$)

▷ *Initialize the vertex set of a bcc with the endpoints of an edge*
(1) **if** $N \neq \emptyset$
(2) **then** $B \leftarrow N$
(3) **else** $B \leftarrow \mathsf{pick}(V)$
(4) $B \leftarrow \mathsf{pick}(\mathsf{img}(B)) \cup B$

▷ *Ensure that* $B \cap \mathcal{S} = N$
(5) **if** $(B \setminus \mathcal{S} = \emptyset)$
(6) **then** $\mathcal{S} \leftarrow \mathcal{S} \setminus N$; $N \leftarrow \mathsf{img}(N) \cap \mathcal{S}$

▷ *Extend the vertex set B*
(7) **while** (∃ an exploration-point other then the spine-anchor) **do**
▷ *Choose an exploration point other than the spine-anchor*
(8) $X \leftarrow \mathsf{pick}(\mathsf{img}(\mathsf{img}(B) \cap (V \setminus B)) \cap (B \setminus N))$
▷ *Explore outside B from X. Obtain a set, C, of new nodes for B or*
▷ *the vertex set* $V' \not\supseteq B$ *of a bcc-closed graph and a spine in it,* $\langle \mathcal{S}', N' \rangle$
(9) $\langle C, V', \langle \mathcal{S}', N' \rangle \rangle \leftarrow$ VISIT($V, E, \langle \mathcal{S}, N \rangle, B, X$)
(10) **if** $(V' \neq \emptyset)$
▷ $X = \{x\}$ *contains an articulation point*
▷ *Recursive call on* $\overline{G(B, x)}$ *coupled with a skeleton*
(11) **then** SYMBOLIC-BCC($V', E \restriction V', \langle \mathcal{S}', N' \rangle$)
(12) $V \leftarrow (V \setminus V') \cup X; \quad E \leftarrow E \restriction V$
▷ *Else augment B with nodes on paths reaching back* $B \cup S$
(13) **else** $B \leftarrow B \cup C$
▷ *Extend B with a spine-suffix*
(14) $\langle B, \langle \mathcal{S}, N \rangle \rangle \leftarrow$ EAT-SPINE($V, E, \langle \mathcal{S}, N \rangle, B$)

(15) Return the vertex set of the biconnected component built in B

▷ *Recursive call in case* $B \neq V$
(16) **if** $(V \setminus B \neq \emptyset)$
(17) **then** $V \leftarrow (V \setminus B) \cup N$; $E \leftarrow E \restriction V$; SYMBOLIC-BCC($V, E, \langle \mathcal{S}, N \rangle$)

Lines (16)–(18) build a spine-set in such a subgraph by suitably selecting a node for each level popped out from `stack`. In the second case, the set of vertices discovered contains at least one simple path between the exploration point and a vertex in $B \cup \mathcal{S}$. The loop at lines (12)–(13) detects the vertex sets of those paths containing exactly one node for each level pushed onto `stack` and whose last node belong to $B \cup \mathcal{S}$. These nodes are assigned to the set C (line (13)) that will be added to B in line (13) of the main algorithm.

Table 8 The subprocedure VISIT used within the linear symbolic bcc-algorithm

VISIT$(V, E, \langle \mathcal{S}, N\rangle, B, X)$

▷ *Initialization*
(1) Let `stack` be an empty stack of vertex sets
(2) $L \leftarrow \mathsf{img}(X) \cap (V \setminus (B \cup \mathcal{S}))$ ▷*Initialize frontier of vertex set discovered from X*
(3) **if** $(L = \emptyset)$
(4) **then return** $(\emptyset, \emptyset, \langle \emptyset, \emptyset\rangle)$

▷ *Explore the vertex set from X*
(5) $U \leftarrow V \setminus (X \cup L)$; ▷*Initialize the undiscovered nodes' set*
(6) $C \leftarrow \emptyset$; ▷*Initialize nodes on simple paths reaching back* $B \cup \mathcal{S}$
(8) **while** $(C = \emptyset \wedge L \neq \emptyset)$ **do** ▷ *Stop as soon as* $(B \cup S) \setminus X$ *is intersected*
(9) $\mathsf{push}(L, \mathtt{stack})$; $L \leftarrow \mathsf{img}(L) \cap U$; $U \leftarrow U \setminus L$;
(10) $C \leftarrow L \cap ((B \cup \mathcal{S}) \setminus X))$

▷ *Return either a nodes' set to augment the bcc partially collected in B,*
▷ *or a bcc-closed subgraph for a new recursive call*
(11) **if** $(C \neq \emptyset)$
▷ *Case 1. The bcc can be augmented: collect in C paths reaching back* $B \cup \mathcal{S}$
(12) **then while** $(\mathsf{Notempty}(\mathtt{stack}))$ **do**
(13) $C \leftarrow \mathsf{img}(C) \cap \mathsf{pop}(\mathtt{stack})$
(14) **return** $(C, \emptyset, \langle \emptyset, \emptyset\rangle)$
▷ *Case 2. X is an articulation point: prepare the next recursive call*
(15) **else** $V' \leftarrow X \cup (V \setminus (B \cup U))$
▷ *Build spine in the bcc-closed subgraph discovered*
(16) $\mathsf{pick}(\mathsf{pop}(\mathtt{stack}))$
(17) **while** $(\mathsf{NotEmpty}(\mathtt{stack}))$ **do**
(18) $\mathcal{S}' \cup \mathsf{pick}(\mathsf{img}(\mathcal{S}') \cap (\mathsf{pop}(\mathtt{stack})))$
(19) **return** $\emptyset, V', \langle \mathcal{S}' \cup X, N'\rangle)$

6.1.2 The subprocedure EAT-SPINE

The purpose of the subprocedure EAT-SPINE, in Table 9, is that of augmenting B with the maximum spine-set that reaches back B. This ensures invariant (1) upon the termination of each SYMBOLIC-BCC loop iteration. EAT-SPINE gets as input a bcc-closed subgraph, $\langle V, E\rangle$, a bcc subset $B \subseteq V$, and a spine-set in $\langle V, E\rangle$, $\langle \mathcal{S}, N\rangle$, such that $B \cap \mathcal{S} = N$. Precisely, the procedure adds to B the spine-suffix, $v_i, \ldots, v_p$, having maximum length and such that $v_i \in B$.

6.2 Correctness and Complexity

The soundness and completeness of the algorithm in Table 7 are stated in Theorems 4 and 5, respectively. Throughout this section we will use the notation V_c^l to indicate the variable V within the c-th recursive call to SYMBOLIC-BCC, on line l. We will

Table 9 The subprocedure EAT-SPINE

EAT-SPINE$(V, E, \langle \mathcal{S}, N\rangle, B)$

(1) $C \leftarrow \mathsf{img}(B \setminus N) \cap \mathcal{S}$
(2) **while** $(C \neq \emptyset)$ **do** $\triangleright$ *Update B with maximum spine suffix reaching back B*
(3) $\quad B \leftarrow B \cup (\mathsf{img}(N) \cap \mathcal{S})$
(4) $\quad C \leftarrow C \setminus N; \quad \mathcal{S} \leftarrow \mathcal{S} \setminus N; \quad N \leftarrow \mathsf{img}(N) \cap \mathcal{S};$
(5) **return** $(B, V', \langle \mathcal{S}' \cup X, N'\rangle)$

denote V_c the value of the parameter V to the c-th recursive call to SYMBOLIC-BCC. Whenever it will be also necessary to specify the loop iteration number, j, in SYMBOLIC-BCC, we will write $V^l_{\langle c,j\rangle}$. An analogous notation will be adopted for all the variables in SYMBOLIC-BCC.

The framework for the following Lemma is the execution of the subprocedure VISIT$(V, E, \langle \mathcal{S}, N\rangle, B, \{x\})$, where:

- $\langle \mathcal{S}, N\rangle$ is a spine in $G = (V, E)$;
- $B \subseteq V$ induces a biconnected subgraph of G such that $B \cap \mathcal{S} = N$;
- $\{x\} \subset B$ is an exploration point of B.

Lemma 13 *Let* $V', \langle \mathcal{S}', N'\rangle, C$ *be the sets returned by* VISIT$(V, E, \langle \mathcal{S}, N\rangle, B, \{x\})$, *where* $G = (V, E)$. *Assume that* V_* *is the vertex set of the biconnected component containing* B. *Then, only one of the following three cases is possible*:

1. x *is an articulation point,* $V' \neq \emptyset$ *is the vertex set of* $\overline{G(V_*, x)}$ *and* $\langle \mathcal{S}', N'\rangle$ *is a spine-set in* $\overline{G(V_*, x)}$;
2. $C \neq \emptyset$ *and the nodes in* C *belong to some simple paths from* X *to* $B \cup \mathcal{S}$ *traversing* $V \setminus (B \cup \mathcal{S})$;
3. $V' = C = \mathcal{S}' = N' = \emptyset$ *and* $\mathsf{img}(\{x\}) \subseteq (B \cup \mathcal{S})$.

Proof Consider the pseudo-code for VISIT in Table 8. Line (2) initializes the set of vertices, say L_1, having distance 1 from x and which neither belong to B nor to the spine. If $L_0 = \emptyset$, then $\mathsf{img}(\{x\}) \subseteq (B \cup \mathcal{S})$ and VISIT returns only empty sets (line (4)). Hence, case 3 applies.

If $L_1 \neq \emptyset$, then the loop in lines (8)–(10) is executed. This loop discovers, level by level in a breadth-first manner, nodes at increasing distance from L_1, without using the exploration point x. The loop stops when, either from the last level of discovered nodes, say L_p, no new vertex can be reached, or it holds that $L_p \cap (B \cup S) \neq \emptyset$. On the first condition, we obtain that all the nodes that can be discovered from L_1, without using x, belong to $V \setminus B$: thus x is an articulation point and the set of nodes discovered is the vertex set of $\overline{G(V_*, x)}$. In this case the else branch of the if statement in line (11) is executed and the just mentioned set of vertices is assigned to V' (line (15)). Lines (16)–(18) build a spine-set in $\overline{G(V_*, x)}$ using a (chordless) path which traverses each level discovered and reaches a node of maximal distance from x. Hence, case 1 applies.

Suppose instead that the loop in lines (8)–(10) finishes discovering a level of nodes L_p with $L_p \cap (\mathcal{S} \cup B) \neq \emptyset$. In such a case C is equal to $L_p \cap (B \cup \mathcal{S})$ on the end

of the loop. Hence, the then branch of the if statement is executed. Lines (12)–(13) complete C with the nodes on all the paths (of length p) from x to $L_p \cap (\mathcal{S} \cup B)$ traversing all the levels discovered by breadth-first search. Thus, case 2 applies. □

Relying on Lemma 13, Lemma 14 provides four invariants for SYMBOLIC-BCC ensuring its correctness.

Lemma 14 *Consider the c-th recursive call within* SYMBOLIC-BCC$(V, E, \langle \emptyset, \emptyset \rangle)$, SYMBOLIC-BCC$(V_c, E_c, \langle \mathcal{S}_c, N_c \rangle)$. *If the guard of the loop in line* (7) *is checked* j_c *times within* SYMBOLIC-BCC$(V_c, E_c, \langle \mathcal{S}_c, N_c \rangle)$, *then for all* $1 \leq j \leq j_c$:

1. $(V^7_{\langle c,j \rangle}, E^7_{\langle c,j \rangle})$ *is a connected and bcc-closed subgraph of* (V, E);
2. $(B^7_{\langle c,j \rangle}, E \restriction B^7_{\langle c,j \rangle})$ *is biconnected*;
3. *If* $j > 1$, *then either* $|B^7_{\langle c,j \rangle}| > |B^7_{\langle c,j-1 \rangle}|$ *or* $|V^7_{\langle c,j \rangle} \setminus V_*| < |V^7_{\langle c,j-1 \rangle} \setminus V_*|$, *where* $(V_*, E \restriction V_*)$ *is the biconnected component of* (V, E) *such that* $B^7_{\langle c,j \rangle} \subseteq V_*$;
4. $\langle \mathcal{S}^7_{\langle c,j \rangle}, N^7_{\langle c,j \rangle} \rangle$ *is a spine-set in the graph* $(V^7_{\langle c,j \rangle}, E^7_{\langle c,j \rangle})$ *and, if* $\mathcal{S}^7_{\langle c,j \rangle} \neq \emptyset$, *then* $\mathcal{S}^7_{\langle c,j \rangle} \cap B^7_{\langle c,j \rangle} = N^7_{\langle c,j \rangle}$.

Proof We follow an inductive argument on $\langle c, j \rangle$, where the usual lexicographical order is used to compare pairs. The base case ($\langle c, j \rangle = \langle 1, 1 \rangle$) is immediate, we sketch below the inductive step.

Let us assume Items (a)–(d) true for all (consistent) pairs $\langle 1, 1 \rangle \leq \langle c', j' \rangle \leq \langle c, j \rangle$. There are two cases to be considered depending on which is the minimum consistent pair greater than $\langle c, j \rangle$.

In the first case $\langle c, j+1 \rangle$ is the minimum consistent pair greater than $\langle c, j \rangle$ (i.e., the while-guard is checked more than j times in the c-th recursive call to SYMBOLIC-BCC). Within each iteration of the loop in SYMBOLIC-BCC the procedure VISIT is executed. We consider each of the three possible cases, by Lemma 13, for the triple returned by VISIT: $C, V', \langle \mathcal{S}', N' \rangle$. If all the sets are empty, then, by Lemma 13, $\mathsf{img}(X^{10}_{\langle c,j \rangle}) \subseteq \mathcal{S}^{10}_{\langle c,j \rangle} \cup B^{10}_{\langle c,j \rangle}$. By the choice of $X = \{x\}$ in line (8), some node of the spine-set other than the spine-anchor is linked to x. The procedure EAT-SPINE detects the least spine-set node, v_l, with the above property, subtracts $\{v_l + 1, \ldots, v_p\}$ from the spine-set, and adds $\{v_l, \ldots, v_p\}$ to B. Hence, $B^7_{\langle c,j+1 \rangle} = B \cup \{v_l \ldots v_p\}$ is biconnected, $B^7_{\langle c,j+1 \rangle} \cap \mathcal{S}^7_{\langle c,j+1 \rangle} = N^7_{\langle c,j+1 \rangle}$ and Items (a)–(d) are satisfied for $\langle c, j+1 \rangle$ when VISIT returns only empty sets. We can also easily handle the case in which the set $V'_{\langle c,j \rangle}$ returned by VISIT is not empty. By Lemma 13 if $V'^{10}_{\langle c,j \rangle}$ is not empty, then $X^7_{\langle c,j \rangle} = \{x\}$ is an articulation point and $V'^{10}_{\langle c,j \rangle}$ is the vertex set of $\overline{G(V_*, x)}$. Line (12) in SYMBOLIC-BCC subtracts this set of vertices from V. This way Items 1(a), 1(c) get satisfied for $\langle c, j+1 \rangle$. Items 1(b), 1(d) also hold since B and the spine-set can be modified only within EAT-SPINE which, as already discussed, maintains them. Finally, if VISIT returns only the set $C^{10}_{\langle c,j \rangle}$ not empty, line (13) in SYMBOLIC-BCC adds $C^{10}_{\langle c,j \rangle}$ to $B^7_{\langle c,j \rangle}$. Thus, B gets enlarged and Lemma 13 ensures that B is added of the nodes on some simple paths from X to $B \cup S$ traversing $V \setminus (B \cup \mathcal{S})$. At this point, EAT-SPINE completes B with the maximal suffix of the chordless path defined by $\langle \mathcal{S}, N \rangle$ reaching back $B \setminus N$. It follows that $B^7_{\langle c,j+1 \rangle}$ induces a biconnected subgraph and all the items in the lemma are satisfied.

In the second case the minimum consistent pair greater than $\langle c, j\rangle$ is $\langle c+1, 1\rangle$ and we must consider the c'-th recursive execution of SYMBOLIC-BCC from which the c-th execution was called ($c' \leq c$). If the just mentioned invocation was made during j'-th iteration in the loop of c'-th call to SYMBOLIC-BCC, then $V^7_{\langle c+1,1\rangle}$ and $\langle \mathcal{S}^7_{\langle c+1,1\rangle}, N^7_{\langle c+1,1\rangle}\rangle$ were built within the procedure VISIT. Thus, by Lemma 13, the graph induced by $V^7_{\langle c+1,1\rangle}$ is connected and bcc closed and $\langle \mathcal{S}^7_{\langle c+1,1\rangle}, N^7_{\langle c+1,1\rangle}\rangle$ is a spine in it. The subgraph $(B^7_{\langle c+1,1\rangle}, E \restriction B^7_{\langle c+1,1\rangle})$ is biconnected, since in lines (1)–(4) of each recursive call to SYMBOLIC-BCC the set B is initialized to the endpoints of an edge. Otherwise, the $(c+1)$-st recursive call to SYMBOLIC-BCC must be done in line 11 in the c'-th execution of SYMBOLIC-BCC. Hence, the validity of Items (a)–(d) on $\langle c+1, 1\rangle$ easily follows from inductive hypothesis. □

Theorem 4 (Correctness) *The algorithm* SYMBOLIC-BCC$(V, E, \langle\emptyset, \emptyset\rangle)$ *computes the vertex sets of the biconnected components in* $G = (V, E)$.

Proof Soundness follows directly from Lemma 14. Precisely, items 1–3 in Lemma 14 ensure that, upon termination of the loop in line (7) of each recursive call to SYMBOLIC-BCC, the vertex set of a biconnected component in $G = (V, E)$ is collected in B. Completeness follows from the fact that lines (16)–(17) and line (11) perform all the recursive calls necessary to have the entire graph considered for bcc partitioning. □

Theorem 5 states that SYMBOLIC-BCC$(V, E, \langle\emptyset, \emptyset\rangle)$ needs $\mathcal{O}(|V|)$ symbolic steps to compute $\{V_1, \ldots, V_n\}$, where $\{\langle V_1, E_1\rangle \cdots \langle V_n, E_n\rangle\}$ are the biconnected components in $G = (V, E)$. As already anticipated, the complexity result is a consequence of the following considerations. First, the global number of symbolic steps spent to detect simple paths reaching back a partial biconnected component is $\mathcal{O}(|V_1| + \cdots + |V_n|)$. Moreover, the global number of symbolic steps spent to obtain the subgraphs for the recursive calls is proportional to the number of insertions of nodes in a spine. By Lemma 16 also the global number of insertions of nodes into a spine is $\mathcal{O}(|V_1| + \cdots + |V_n|)$. Note that $(|V_1| + \cdots + |V_n|) \geq |V|$ because of the articulation points, however, by Lemma 15, given below, it holds that $(|V_1| + \cdots + |V_n|) = \mathcal{O}(|V|)$.

Lemma 15 *Consider the set* A_G *of articulation points of* $G = (V, E)$. *Given* $a \in A_G$ *let* $m_G(a)$ (*the* multiplicity *of the articulation point* a) *be the number of biconnected components of* G *that contain* a. *Then*

$$\sum_{a \in A} m_G(a) \leq 2|V| - 4.$$

Proof By induction on the number of biconnected components of G, n. If $n = 1$, there are no articulation points and $2|V| - 4 \geq 2 * 2 - 4 = 0$. Otherwise, by item 3 in Lemma 2, there is at least one articulation point in G. Let a be an articulation point of G and let $\langle V_*, E_*\rangle$ be a biconnected component of G such that $a \in V_*$. We use the inductive hypothesis on the two subgraphs $G_1 = \langle V_1, E_1\rangle = G(V_*, a)$ and $G_2 =$

$\langle V_2, E_2\rangle = \overline{G(V_*, a)}$. By Definition 10 and Lemma 7 it follows that $V_1 \cap V_2 = \{a\}$ and:

$$\forall w \in V_i \setminus \{a\}(w \in A_{G_i} \wedge m_{G_i}(w) = k) \quad \Leftrightarrow \quad (w \in A_G \wedge m_G(w) = k),$$

for $i = 1, 2$. By definition of $G(V_*, a) = G_1$, a cannot be an articulation point in G_1. If a is an articulation point in G_2, $m_G(a) = m_{G_2}(a) + 1$. Otherwise $m_G(a) = 2$. Thus:

$$\begin{aligned}\sum_{a\in A} m_G(a) &\leq \sum_{a\in A_{G_1}} m_{G_1}(a) + \sum_{a\in A_{G_2}} m_{G_2}(a) + 2\\ &\leq (2|V_1| - 4) + (2|V_2| - 4) + 2 = 2|V| + 2 - 8 + 2 = 2|V| - 4. \quad \square\end{aligned}$$

Lemma 16 *Let v be a node belonging to m biconnected components of $G = (V, E)$. During the entire execution of* SYMBOLIC-BCC$(V, E, \langle\emptyset, \emptyset\rangle)$, *$v$ is inserted at most m times in a spine-set.*

Proof Consider the c-th recursive call to SYMBOLIC-BCC within execution of SYMBOLIC-BCC$(V, E, \langle\emptyset, \emptyset\rangle)$. By induction on the number of recursive calls, we prove that $\forall v \in V_c$, in the first $c - 1$ (recursive) calls to SYMBOLIC-BCC:

- $v \in V_c$ has been inserted at least once in a spine if and only if $v \in \mathcal{S}_c$;
- If $v \in V_c$ has been inserted m times in a spine then v belongs to at least m distinct biconnected components. Moreover, in this case $m - 1$ biconnected components containing v have been already produced in output.

Base: Immediate as $\langle\mathcal{S}_1, N_1\rangle = \langle\emptyset, \emptyset\rangle$.

Inductive Step: Let $c > 1$ and $v \in V_c$. Assume that the c-th call to SYMBOLIC-BCC was done during the c'-th execution of SYMBOLIC-BCC, which builds the biconnected component set of vertices $V_{(*,c')}$. Let $a_1, \ldots, a_l$ be the articulation points of $(V_{c'}, E_{c'})$ that are contained in $V_{(*,c')}$. By Lemmas 13 and 14, the subgraphs of $G_{c'} = (V_{c'}, E_{c'})$ involved in the recursive calls of SYMBOLIC-BCC$(V_{c'}, E_{c'}, \langle\mathcal{S}_{c'}, N_{c'}\rangle)$, are $\overline{G_{c'}(V_{(*,c')}, a_k)}$, where $a_k \in \{a_1, \ldots, a_l\}$. These subgraphs have no vertex in common, thus $\forall c' < c'' < c$, $v \notin V_{c''}$ and we can safely avoid considering the recursive calls between the c'-th one and the c-th one for the inductive step. If the c-th recursive call to SYMBOLIC-BCC was made after the loop of the c'-th execution of SYMBOLIC-BCC, then $\langle\mathcal{S}_c, N_c\rangle$ is a "prefix" of $\langle\mathcal{S}_{c'}, N_{c'}\rangle$ and v was not involved in any operation of insertion into a spine-set in the c'-th execution of SYMBOLIC-BCC. In this case the inductive hypothesis (on c') ensures the inductive step.

Otherwise, the c-th recursive call to SYMBOLIC-BCC must be done within an iteration of the loop internal to the c'-th execution of SYMBOLIC-BCC. In this case $\langle\mathcal{S}_c, N_c\rangle$ must have been produced within VISIT, in the process of exploring from an articulation point of $V_{(*,c')}$. By Lemma 8, there is at most one vertex in $\mathcal{S}_{c'} \cap V_c$. Thus, we can conclude what follows on the ground of the inductive hypothesis on c'. If $v \in V_c \setminus \mathcal{S}_c$, then, when the c-th recursive call is done, v has not yet been inserted in any spine. If $v \in \mathcal{S}_c \wedge v \notin \mathcal{S}_{c'}$, then when the c-th recursive call is done v has been inserted only once in a spine-set. Finally, in case $v \in \mathcal{S}_{c'} \cap \mathcal{S}_c$ we have that $\{v\} = \mathcal{S}_{c'} \cap \mathcal{S}_c$: if v has been inserted m times in a spine-set upon the c'-th execution

of SYMBOLIC-BCC, then v has been inserted $m+1$ times in a spine-set upon the c'-th execution of SYMBOLIC-BCC. Moreover, as the c'-th recursive call produces $V_{(*,c')} \supseteq \{v\}$ we have that m biconnected components containing v have been already been produced in output upon the execution of SYMBOLIC-BCC$(V_c, E_c, \langle \mathcal{S}_c, N_c \rangle)$. □

Theorem 5 (Complexity) *The algorithm* SYMBOLIC-BCC$(V, E, \langle \emptyset, \emptyset \rangle)$ *runs in* $\mathcal{O}(|V|)$ *symbolic steps*.

Proof The complexity of the algorithm can be computed by counting the global number of iterations of the loops in the subprocedures VISIT and EAT-SPINE. As far as EAT-SPINE is concerned, each time an iteration of its loop is performed a new node is assigned to a biconnected component. Thus the global number of iterations of the EAT-SPINE loop is $\mathcal{O}(|V_1| + \cdots + |V_n|)$, where $\{\langle V_1, E_1 \rangle \cdots \langle V_n, E_n \rangle\}$ are the biconnected components of G.

During each execution of VISIT the number of iterations of the first loop is the same as the number of iterations of the second loop. This number, say p, corresponds to the number of disjoint levels of nodes pushed onto the stack used within VISIT. Upon the termination of VISIT, either, for each level in the stack, some nodes are assigned to a biconnected component, or, for each level in the stack, a vertex is inserted into a spine-set. It follows, by Lemma 16, that the global number of iterations of the VISIT loop is $\mathcal{O}(|V_1| + \cdots + |V_n|)$, where $\{\langle V_1, E_1 \rangle \cdots \langle V_n, E_n \rangle\}$ are the biconnected components of G.

By Lemma 15 $\mathcal{O}(|V_1| + \cdots + |V_n|) = (\mathcal{O}|V|)$. □

Remark 1 Notice that also the global number of operations on OBDDs, performed by SYMBOLIC-BCC, is $\mathcal{O}(|V|)$. In fact, each iteration in each loop of SYMBOLIC-BCC contains at least one symbolic step. Thus the global number of symbolic operations is asymptotically bounded by the global number of symbolic steps. Since each biconnected component is built performing possibly more than one reachability analysis within the procedure VISIT it is not possible, instead, to obtain a bound of $\mathcal{O}(Nd)$ symbolic operations, where N is the number of biconnected components in G and d is the diameter of G.

7 Applications to Model Checking

Model checking [12] is an automatic verification technique used to formally prove properties of both hardware and software systems. In Model Checking the system correctness specification is represented as a temporal logic formula and the verification algorithm checks whether the formula is true in the model representing the system. The latter can be given either as a labeled transition system (a *Kripke structure*) or as an automaton (automata-based Model Checking [42]). Different algorithms have been developed depending on the temporal logic language and on the—either explicit or symbolic—representation of the model. For a complete introduction to Model Checking we refer the reader to [12] which presents all the basic techniques. With our respect, Model Checking can be seen as both an hopeful departure point and a

sure final arrival land for applicability of symbolic graph algorithms. In fact, on the one hand the success of symbolic Model Checking [11, 28] inspires the possibility of achieving similar outstanding new standard problem dimensions in other areas dealing with graph computation. On the other hand, Model Checking can be an application field of symbolic algorithmic solutions to classical graph problems.

In this section, we exactly discuss how our SYMBOLIC-SCC algorithm, as well as the spine-set tool introduced, can be applied to symbolic model checking. In particular, we will develop some variants of the procedure SYMBOLIC-SCC targeted to solve, in a linear number of symbolic steps, the emptiness language problem on various kinds of ω-automata [40]. Both linear temporal logic (LTL) Model Checking and fair computational tree temporal logic (CTL) Model Checking can be reduced to some kind of ω-automata language emptiness problem (cf. Sects. 7.1 and 7.2). Hence, our final achievement is a set of scc-based algorithms for fair CTL and LTL Model Checking, performing in a linear number of symbolic steps. A strategy similar to our was recently exploited by Bloem et al. in [4, 5] to design $\mathcal{O}(V \log(V))$ symbolic steps algorithms for the same purposes, based on their $\mathcal{O}(V \log(V))$ symbolic steps SCCs procedure. With respect to the number of symbolic steps, the above results outperform on symbolic algorithms currently implemented in symbolic model checkers tools, such as VIS and SMV [8, 28]. In fact, the latter procedures are base on an algorithm by Emerson and Lie [15], that uses a nested alternate fix-point computation, and performs $\Theta(V^2)$ symbolic steps in the worst case [4, 5]. We finally point out that, despite we believe that our results have an intrinsic theoretical interest, whether the number of symbolic steps is the "right" parameter to consider in comparing symbolic algorithms practical performances, is a problem deserving a major experimental effort.

7.1 Symbolic Model Checking, Büchi Language Emptiness, and Spine-sets

In the automata based approach to LTL Model Checking [42], both the model of the system and the negation of the LTL formula encoding the correctness requirement, $\neg\phi$, are translated into Büchi automata, $\mathcal{A}$ and $\mathcal{A}_{\neg\phi}$. Then, the language emptiness of the Büchi automaton $\mathcal{A} \times \mathcal{A}_{\neg\phi}$ is checked: in fact, the composed automaton accepts an input if and only if the system does not satisfy the correctness specification. The Büchi emptiness problem is in turn equivalent to the detection of a cycle, reachable from the initial states and containing a *final* state: for this reason, in the above context the algorithmic task of determining the language emptiness for $\mathcal{A} \times \mathcal{A}_{\neg\phi}$ is called *bad cycle detection*.

More precisely, (the language of) a Büchi automaton is defined as follows:

Definition 12 A *Büchi Automaton*, $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$, is a tuple in which:

- Σ is a finite alphabet;
- V is a finite set of states;
- $\Delta : V \times \Sigma \times V$ is the transition relation;
- V_0 is the set of initial states;
- $F \subseteq V$ is a set of nodes that represents the acceptance condition.

A *run* of the automaton $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$, on the infinite word $\alpha \in \Sigma^\omega$, is a mapping $\rho : \{0, 1, \ldots, \omega\} \mapsto V$, such that $\rho(0) \in V_0 \wedge \forall i \geq 0((\rho(i), \alpha[i], \rho(i+1)) \in \Delta)$. In a Büchi automaton $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$ a run is *accepting* if and only if $inf(\rho) \cap F \neq \emptyset$, where $inf(\rho)$ represents the set of states infinitely occurring as images in ρ. The *language* of $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \rangle$ consists of those words $\alpha \in \Sigma^\omega$ for which there exists an accepting run.

Lemma 17 [40] *The language of* $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, F \subseteq V \rangle$ *is not empty if and only if the graph,* $\langle V, E = \cup_{a \in \Sigma} \Delta(a) \rangle$*, contains a cycle of states that can be reached through a path starting from an initial state,* $q \in V_0$.

In the explicit case the characterization in Lemma 17 allows us to use SCC decomposition to design an explicit Büchi language emptiness algorithm, linear in the size of $\mathcal{A} \times \mathcal{A}_{\neg\phi}$. Let V' be the set of states reachable from V_0 using the relation $E = \bigcup_{a \in \Sigma} \Delta(a)$. Once obtained the scc's in $\langle V', E \rangle$, using, e.g., linear Tarjan algorithm [39], it is only necessary to check if there is a non trivial strongly connected component containing a final state.

As stated in the preamble to this section, the symbolic bad cycle detection algorithms integrated in most tools for symbolic Model Checking are based on a procedure by Emerson and Lie [15] performing in $\Theta(V^2)$ symbolic steps in the worst case [4, 5]. Lemma 17, combined with our symbolic linear algorithm for scc's decomposition can be used to cut down to $\mathcal{O}(V)$ the number of symbolic steps required to perform symbolic bad cycle detection, with a final strategy resembling the one used in the explicit case. More precisely, it is possible to obtain a linear symbolic algorithm for Büchi language emptiness detection by simply substituting line (9) of SYMBOLIC-SCC, with the following instruction, that checks if the outlined strongly connected component is not trivial and contains a final state:

(9′) **if** (post$(F \cap SCC) \cap SCC \neq \emptyset$) **then return** "language NOT EMPTY"

Line (9′) performs a constant number of symbolic steps and thus, clearly, it does not change the complexity of SYMBOLIC-SCC.

7.2 Spine-sets Applied to Fair Model Checking: Generalized Büchi and Street Automata Language Emptiness

Fairness is a crucial assumption in the context of correctness analysis of many systems. Fairness constraints are usually classified into *weak fairness* constraints and *strong fairness* constraints [19, 21]. Weak fairness guarantees that no enabled transition is postponed forever. For example, when verifying a communication protocol over reliable channels only fair execution paths, in which no message happens to be continuously sent but never received, should be considered. Strong fairness imposes that if an action is infinitely often enabled, then it will be infinitely often taken and it is used, for example, in the analysis of synchronous interactions.

As far as weak fairness is concerned, CTL Model Checking considers weak fairness by modeling systems with a variant of Büchi automata (introducing just

enough of second order expressivity). Namely, weak fairness constraints are introduced by means of a family of sets of states $\mathcal{F} = \langle F_1, \ldots, F_p \rangle$, that represent the acceptance condition of a generalized Büchi automaton $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, \mathcal{F} = \langle F_1, \ldots, F_k \rangle \rangle$. A run in a generalized Büchi automaton is accepting if and only if $\forall i = 1, \ldots, k$ $(\inf(\rho) \cap F_i \neq \emptyset)$. Restricting the evaluation of CTL formulas to weak fair paths, corresponds to solve a generalized Büchi emptiness problem. In turn, non emptiness language of a generalized Büchi automaton can be characterized as the presence of a reachable cycle of states, containing at least one element for each $F_i \in \mathcal{F}$. Symbolic CTL model checkers as VIS and SMV [8, 28], use, again, the procedure by Emerson and Lie [15] to solve the above problem in $\mathcal{O}(V^2)$ symbolic steps. It is possible to obtain a linear symbolic algorithm for generalized Büchi emptiness algorithm by simply substituting line (9) of SYMBOLIC-SCC, with the following group of instructions:

(9*) **if** (for all $F_i \in \mathcal{F}$ $(F_i \cap SCC \neq \emptyset)) \wedge (\mathsf{post}(SCC) \cap SCC \neq \emptyset)$
then return "language not empty"

Line (9*) performs a constant number of symbolic steps and thus, clearly, it does not change the complexity of SYMBOLIC-SCC.

Moving from weak to strong fairness constraints, in CTL model checking, corresponds to solve a language emptiness problem for Street automata. A Street automaton differs from a generalized Büchi one only in the acceptance condition $\mathcal{F} = \langle (P_1, V_1), \ldots, (P_n, V_n) \rangle$, constituted by a family of pairs of vertices sets. A run ρ of a Street automaton is *accepting* if for each pair of subsets in the acceptance condition, (P_i, V_i), it holds that $\inf(\rho) \cap P_i \neq \emptyset \rightarrow \inf(\rho) \cap V_i \neq \emptyset$. The language of a Street automaton, $\mathcal{A}$, is not empty if and only if $\mathcal{A}$ contains a cycle, $C = \langle s_1 \ldots s_n \rangle$, such that for each pair (P_i, V_i) of subsets in the acceptance condition it holds that $(\{s_1, \ldots, s_n\} \cap P_i \neq \emptyset \rightarrow \{s_1, \ldots, s_n\} \cap V_i \neq \emptyset)$.

On this ground, once a strongly connected component of a Street automaton is computed, determining whether it witnesses language non emptiness is a bit more tricky than in the case of Büchi automata. In fact, given a pair of subsets in the acceptance condition, (P_i, V_i), if a strongly connected component, S, intersects only P_i it is still possible that it witnesses language not emptiness: the reason is that a cycle which does not intersect neither P_i nor V_i can be contained in such strongly connected component. If there exists such a cycle, it should be searched recursively in the strongly connected components of $S \setminus P_i$. The authors of [4, 5] combined the above ideas with their strategies to obtain scc decomposition in $\mathcal{O}(V \log(V))$ symbolic steps: the result is a symbolic algorithm to check language emptiness for Street automata performing $\mathcal{O}(V \log(V))$ symbolic steps. Adapting the overall approach to deal with our scc symbolic algorithm allows us to design a procedure which solves the same problem in a linear number of symbolic steps. More precisely, it is only necessary to substitute line (9), in our SYMBOLIC-SCC algorithm, with a call to the subprocedure depicted in Table 11, that implements the recursive scc analysis outlined above. The final linear symbolic algorithm checking language emptiness for Street automata is reported in Table 10. Given a Street automaton $\mathcal{A} = \langle \Sigma, V, \Delta, V_0, \mathcal{F} \rangle$, the procedure SYMSA takes in input the induced graph $G = \langle V, E \rangle$, where $E = \bigcup_{a \in \Sigma} \Delta(a)$, a

Table 10 The algorithm SYMSA, determining in a linear number of symbolic steps if the language accepted by a given Street automaton is empty

SYMSA$(V, E, \langle \mathcal{S}, N\rangle, \mathcal{F} = \langle (P_1, V_1), \ldots, (P_k, V_k)\rangle)$

(1) **if** $V = \varnothing$
(2) **then return**;

▷ *Determine the node for which the scc is computed*
(3) **if** $\mathcal{S} = \varnothing$
(4) **then** $\mathrm{N} \leftarrow pick(\mathrm{V})$;

▷ *Compute the forward-set of the singleton N and a skeleton*
(5) $\langle FW, new\mathcal{S}, newN\rangle \leftarrow$ SKEL-FORWARD(V, E, N);

▷ *Determine the scc containing N*
(6) $SCC \leftarrow N$;
(7) **while** $((\mathsf{pre}(SCC) \cap FW) \setminus SCC) \neq \varnothing$ **do**
(8) $SCC \leftarrow SCC \cup (\mathsf{pre}(SCC) \cap FW)$;

▷ *Check the computed scc to determine language emptiness*
(9) CHECKSCC$(SCC, V, E, \mathcal{F} = \langle (P_1, V_1), \ldots, (P_k, V_k)\rangle)$

▷ *First recursive call: computation of the scc's in* $V \setminus FW$
(10) $V' \leftarrow V \setminus FW$; $E' \leftarrow E \restriction V'$;
(11) $\mathcal{S}' \leftarrow \mathcal{S} \setminus SCC$; $N' \leftarrow \mathsf{pre}(SCC \cap \mathcal{S}) \cap (\mathcal{S} \setminus SCC)$;
(12) SYMSA$(V', E', \langle \mathcal{S}', N'\rangle)$

▷ *Second recursive call: computation of the scc's in* $FW \setminus SCC$
(13) $V' \leftarrow FW \setminus SCC$; $E' \leftarrow E \restriction V'$;
(14) $\mathcal{S}' \leftarrow new\mathcal{S} \setminus SCC$; $N' \leftarrow newN \setminus SCC$;
(15) SYMSA$(V', E', \langle \mathcal{S}', N'\rangle)$

Table 11 The subroutine CHECKSCC, called by the procedure SYMSA

CHECKSCC$(SCC, V, E, \mathcal{F} = \langle (P_1, V_1), \ldots, (P_k, V_k)\rangle)$

(1) **if** (for all$(P_i, V_i) \in \mathcal{F}(SCC \cap P_i \neq \emptyset \rightarrow SCC \cap V_i \neq \emptyset)$)
(2) **then return** "language not empty"
(3) $C \leftarrow SCC$;
(4) **for each** $(P_i, V_i) \in \mathcal{F}(P_i \cap SCC \neq \emptyset \wedge V_i \cap SCC = \emptyset)$ **do**
(5) $C \leftarrow C \setminus P_i$;
(6) **if** $(C \neq \emptyset)$ **then** SYMSA$(C, E \restriction C, \mathcal{F}, \langle \emptyset, \emptyset\rangle)$;

spine-set in G, and the family $\mathcal{F} = \langle (P_1, V_1), \ldots, (P_k, V_k)\rangle$ defining the automaton acceptance conditions.

Assuming that the number of fairness constraints is a constant with respect to the automaton states space, the linear number of symbolic steps required in the worst

case by the algorithm SymSA is proved on the ground of the following considerations:

1. Each line in the subroutine CheckSCC, but line 6 requires at most a $\mathcal{O}(k)$ symbolic steps, where k is the number of fairness constraints;
2. Given a strongly connected component of the automaton, *SCC*, line 6 is executed at most $\mathcal{O}(k)$ times on (a subset) of it: in fact, each recursive call subtracts at least one vertex from it (in line 5). Thus, given a strongly connected component of the automaton, *SCC*, line 6 globally requires $\mathcal{O}(SCC)$ symbolic steps.

8 Conclusions

In this paper, we tackled the problem of designing efficient graph connectivity algorithms, assuming a symbolic data representation by means of OBDDs. We devised a symbolic algorithmic strategy, based on the so called *spine-set* notion, that is general enough to be the engine of linear symbolic steps algorithms for both strongly connected components and biconnected components. Thus, to some extent, spine-sets can be viewed as the symbolic counterpart of the well known DFS preprocessing speedup, fundamental in various classical algorithms. We see our work only as a first step in a wider research, aiming at understanding the relationships between explicit and symbolic algorithms, and at defining a whole set of tools and methods to both analyzing and designing efficient symbolic algorithms.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison–Wesley, Reading (1974)
2. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **C-27**(6), 509–516 (1978)
3. Bloem, R.P.: Search Techniques and Automata for Symbolic Model Checking. University of Colorado (2001)
4. Bloem, R.P., Gabow, H.N., Somenzi, F.: An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Proc. of Int. Conference on Formal Methods in Computer-Aided Design (FMCAD'00). LNCS, vol. 1954, pp. 37–54. Springer, Berlin (2000)
5. Bloem, R.P., Gabow, H.N., Somenzi, F.: An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. Form. Methods Syst. Des. **28**, 1–20 (2005)
6. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
7. Bouali, A., de Simone, R.: Symbolic bisimulation minimization. In: von Bochmann, G., Probst, D.K. (eds.) Proc. of Int. Conference on Computer Aided Verification (CAV'92). LNCS, vol. 663, pp. 96–108. Springer, Berlin (1992)
8. Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G., Villa, T.: VIS: a system for verification and synthesis. In: Alur, R., Henzinger, T.A. (eds.) Proc. of Int. Conference on Computer Aided Verification (CAV'96). LNCS, vol. 1102, pp. 428–432. Springer, Berlin (1996)
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
10. Bryant, R.E.: Symbolic boolean manipulation with ordered binary decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)

11. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. of IEEE Symp. on Logic in Computer Science (LICS'90), pp. 1–33. IEEE Computer Society, Los Alamitos (1990)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
13. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
14. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Softw. Tools Technol. Transf. **3**, 103–112 (2001)
15. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional mu-calculus. In: Proc. of IEEE Symp. on Logic in Comuter Science (LICS'86), pp. 267–278. IEEE Computer Society, Los Alamitos (1986)
16. Feigenbaum, J., Kannan, S., Vardi, M.Y., Viswanathan, M.: The complexity of problems on graphs represented as OBDDs. Chic. J. Theor. Comput. Sci. **47**, 1–25 (1999)
17. Fisler, K., Vardi, M.Y.: Bisimulation and model checking. In: Pierre, L., Kropf, T. (eds.) Proc. of Correct Hardware Design and Verification Methods (CHARME'99). LNCS, vol. 1703, pp. 338–341. Springer, Berlin (1999)
18. Fisler, K., Vardi, M.Y.: Bisimulation minimization and symbolic model checking. Form. Methods Syst. Des. **21**(1), 39–78 (2002)
19. Francez, N.: Fairness. Springer, Berlin (1986)
20. Frank, A.: Connectivity and network flows. In: Handbook of combinatorics, vol. 1, pp. 875–917. Elsevier, Amsterdam (1995)
21. Gabbay, D.M., Pnueli, A., Shelah, S., Stavi, J.: On the temporal basis of fairness. In: Proc. of ACM Sym. on Principles of Programming Languages (POPL'80), pp. 163–173. ACM, New York (1980)
22. Gentilini, R., Policriti, A.: Biconnectivity on symbolically represented graphs: a linear solution. In: Proc. of Int. Symposium on Algorithms and Computation (ISAAC'03). LNCS, vol. 2906, pp. 554–564. Springer, Berlin (2003)
23. Gentilini, R., Piazza, C., Policriti, A.: Computing strongly connected components in a linear number of symbolic steps. In: Proc. of Int. Symposium on Discrete Algorithms (SODA'03), pp. 573–582. ACM, New York (2003)
24. Hachtel, G.D., Somenzi, F.: A symbolic algorithms for maximum flow in 0-1 networks. Form. Methods Syst. Des. **10**(2/3), 207–219 (1997)
25. Hojati, R., Touati, H.J., Kurshan, R.P., Brayton, R.K.: Efficient *mega*-regular language containment. In: von Bochmann, G., Probst, D.K. (eds.) Proc of Int. Conference on Computer Aided Verification (CAV'92). LNCS, vol. 663, pp. 396–409. Springer, Berlin (1992)
26. Hopcroft, J.E., Tarjan, R.E.: Efficient algorithms for graph manipulation [h] (algorithm 447). Commun. ACM **16**(6), 372–378 (1973)
27. Lei, C.Y.: Representation of switching circuits by binary-decision programs. Bell Syst. Tech. J. **38**, 985–999 (1959)
28. McMillan, K.L.: Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic, Dordrecht (1993)
29. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design. OBDD—Foundations and Applications. Springer, Berlin (1998)
30. Nunkesser, R., Woelfel, P.: Representation of Graphs by OBDDs. In: Proc. of Int. Symposium on Algorithms and Computation (ISAAC'05). LNCS, vol. 3827, pp. 1132–1142. Springer, Berlin (2005)
31. Ravi, K., Bloem, R.P., Somenzi, F.: A comparative study of symbolic algorithms for the computation of fair cycles. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Proc. of Int. Conference on Formal Methods in Computer-Aided Design (FMCAD'00). LNCS, vol. 1954, pp. 143–160. Springer, Berlin (2000)
32. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package based on exploiting memory hierarchy. In: Proc. of ACM/IEEE Design Automation Conference, 1996
33. Sawitzki, D.: Implicit flow maximization by iterative squaring. In: Proc. of Int. Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM2004). LNCS, vol. 2932, pp. 301–313. Springer, Berlin (2004)
34. Sawitzki, D.: A symbolic approach to the all-pairs shortest-paths problem. In: Proc. of Int. Conference on Graph-Theoretic Concepts in Computer Science (WG 2004). LNCS, vol. 3357, pp. 154–167. Springer, Berlin (2004)
35. Schneider, K.: Verification of Reactive Systems. Springer, Berlin (2004)
36. Sieling, D.: The nonapproximability of OBDD minimization. Inf. Comput. **172**(2), 103–138 (2002)

37. Somenzi, F.: Binary decision diagrams. In: Calculational System Design, Nato Science Series F: Computer and Systems Sciences, vol. 173, pp. 303–366. IOS Press, Amsterdam (1999)
38. Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.3.1 (2001). Available at http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html
39. Tarjan, R.E.: Depth first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
40. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, vol. B, pp. 133–191. MIT Press, Cambridge (1990)
41. Touati, J.H., Brayton, R.K., Kurshan, R.P.: Testing language containment for omega-automata using BDD's. Inf. Comput. **118**(1), 101–109 (1995)
42. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Logics for Concurrency: Structure versus Automata. LNCS, vol. 1043, pp. 238–266. Springer, Berlin (1996)
43. Wegener, I.: BDDs design, analysis, complexity, and applications. Discrete Appl. Math. **138**, 229–251 (2004)
44. Woelfel, R.: Symbolic topological sorting with OBDDs. J. Discrete Algorithms (2006, to appear) (also in MFCS'03, LNCS, vol. 2747, pp. 671–680)
45. Xie, A., Beerel, P.: Implicit enumeration of strongly connected components and an application to formal verification. IEEE Trans. Comput. Aided Design Integrated Circuits Syst. **19**, 1225–1230 (2000)
46. Yuan, L., Gui, C., Chuah, C., Mohapatra, P.: Applications and design of heterogeneous and or broadband sensor networks. In: Proc. of IEEE First Workshop on Broadband Advanced Sensor Networks. IEEE Press, New York (2003)