FACTA UNIVERSITATIS (NIŠ)
SER.: ELEC. ENERG. vol. 24, no. 3, December 2011, 325-339

# Linear algorithms for recognizing and parsing superpositional graphs

**Ahti Peder, Härmel Nestra, Jaan Raik, Mati Tombak, and Raimund Ubar**

**Abstract:** Structurally synthesized binary decision diagrams (SSBDD) are a special type of BDDs that are generated by superposition according to the structure of propositional formula. Fast algorithms for simulation, diagnostic reasoning and test generation running on SSBDDs exploit their specific properties. Hence the correctness of SSBDDs should be checked before using those algorithms. The problem of recognizing SSBDDs can be reduced to the problem of recognizing their skeleton, namely superpositional graphs, which are a proper subclass of binary graphs.

This paper presents linear time algorithms for testing whether a binary graph is a superpositional graph and for restoring the history of its generating process.

**Keywords:** BDD; binary graph; structurally synthesized binary decision diagram (SSBDD); recognition algorithm.

## 1 Introduction

Within the last two decades, binary decision diagrams (BDD) have become the state-of-the-art data structure in VLSI CAD for representation and manipulation of Boolean functions. They were first introduced for logic simulation in [1], and for test generation in [2, 3]. In 1986, Bryant proposed a new data structure called reduced ordered BDDs (ROBDDs) [4]. He showed simplicity of the graph manipulation and proved the model canonicity that made BDDs one of the most popular representations of Boolean functions [5–7].

Manuscript received November 14, 2011. An earlier version of this paper was presented at the Reed Muller 2011 Workshop, May 25-26, 2011, Gustavelund Conference Centre, Tuusula, Finland.

A.Peder and H.Nestra are with University of Tartu, Tartu, Estonia (e-mails: [ahti.peder, harmel.nestra@ut.ee). J.Raik, M.Tombak and R.Ubar are with Tallinn University of Technology, Tallinn, Estonia (e-mails: jaan@pld.ttu.ee, mati.tombak@ut.ee, and raiub@pld.ttu.ee).

Digital Object Identifier: 10.2298/FUEE1103325P

Structurally synthesized BDDs (SSBDD) form a subclass of BDDs that was proposed and developed with the goal to represent, simulate and analyze structural features of circuits [2, 8, 9]. The most significant difference between the function-based BDDs and structure-based SSBDDs is in the method how they are generated. While BDDs are created on the functional basis by Shannon expansion, or by other types of expansions (like Davio or Reed-Muller expansions) of Boolean functions, the SSBDD models are synthesized by superposition according to the structure of the propositional formula which extracts both functions and the data about the structure of the circuit. The linear complexity of the SSBDD model results from the fact that a digital circuit is represented as a system of SSBDDs, where for each fanout-free region (FFR) a separate SSBDD is generated. This allows hierarchical approach to diagnostic modeling of complex digital circuits. On the higher level, a circuit is represented as a network of FFR modules whereas the lower level modeling is carried out for FFRs using SSBDDs [9].

The speed and correctness of simulation, diagnostic reasoning and test generation algorithms running on SSBDDs have been achieved thanks to exploiting specific properties of SSBDDs. Thus the correctness of SSBDDs should be checked before using the mentioned algorithms.

However, it has been unclear up to now how to check if a given BDD belongs to the class of SSBDDs and how to restore the propositional formula represented by an SSBDD. This paper describes linear time algorithms for solving these problems. These algorithms are based on an equivalent description of superpositional graphs that does not use the notion of superposition [10].

The organization of the paper is as follows. In Section 2, formal definitions for SSBDDs and superpositional graphs are given. Section 3 reproduces necessary definitions and theorems (without proofs) from [10] and, in Section 4, the algorithms referred to in the previous paragraph are presented and explained. Section 5 contains additional reference to related work and Section 6 concludes the paper.

## 2 Structurally Synthesized Binary Decision Diagrams

Denote $B = \{0, 1\}$. Let a mapping $f : B^n \to B$ represent a Boolean function.

A *binary graph* is a directed acyclic connected graph with a root and two terminal nodes (sinks) labeled with 0 and 1 such that every internal (i.e., not terminal) node $v$ has exactly two immediate successors denoted by high$(v)$ and low$(v)$. An edge $a \to b$ is called 0-*edge* (1-*edge*) if low$(a) = b$ (high$(a) = b$). Binary graphs are skeletons of binary decision diagrams (BDD): a BDD is a binary graph in which internal nodes are labelled by propositional variables. We denote the label of a node $v$ by label$(v)$.

A *path* from node $u$ to node $v$ ($u \rightsquigarrow v$) is a sequence $w_0, \ldots, w_k$ of nodes where $w_0 = u$, $w_k = v$ and for each $0 \leq i < k$, $w_{i+1} = \text{high}(w_i)$ or $w_{i+1} = \text{low}(w_i)$. A 0-*path* (1-*path*) is a path which contains only 0-edges (1-edges).

Let $D$ be a binary decision diagram with variables $x_1, \ldots, x_n$. Every vector $\alpha \in B^n$ *activates* a path $w_0, \ldots, w_k$ in $D$ from the root to a terminal node: if $\alpha \vdash \text{label}(w_i)$ then $w_{i+1} = \text{high}(w_i)$, otherwise $w_{i+1} = \text{low}(w_i)$. The Boolean function $f_D(x_1, \ldots, x_n)$ represented by $D$ is defined as follows: $f(\alpha) = 1$ iff the path activated by $\alpha$ ends in terminal 1.

**Definition 1.** Propositional formula with variables $x_1, \ldots, x_n$ is defined inductively as follows:

1° every literal (i.e., $x_i$ or $\neg x_i$ for some $i$) is a propositional formula;

2° if $P$ and $Q$ are propositional formulae then $(P \& Q)$ and $(P \vee Q)$ are propositional formulae.

**Definition 2.** A *superposition* of a binary graph $E$ into a binary graph $G$ for an internal node $v$, denoted by $G[v \leftarrow E]$, is a graph obtained by deleting $v$ from $G$ and redirecting all edges pointing to $v$ to the root of $E$, all edges of $E$ pointing to terminal 1 to the node $\text{high}(v)$ and all edges of $E$ pointing to terminal 0 to the node $\text{low}(v)$.

In the figures, we draw 1-edges from left to right and 0-edges from up to down, without showing the labels 1 and 0. An example in Figure 1 characterizes the process of finding the graph $G[v \leftarrow E]$.

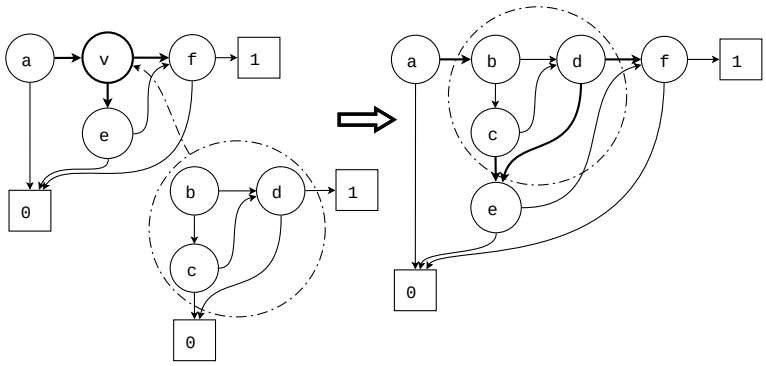


Fig. 1. The superposition $v \leftarrow E$ in the graph $G$.

Let $A$, $C$ and $D$ be binary graphs whose descriptions are shown in Figure 2.

**Definition 3.** The class *SPG* of *superpositional graphs* is defined inductively as follows:

1° graph $A$ is in $SPG$;

2° if $G \in SPG$ and $v$ is an internal node of $G$ then $G[v \leftarrow C] \in SPG$ and $G[v \leftarrow D] \in SPG$.

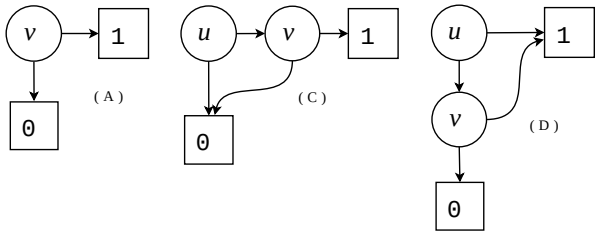


Fig. 2. Binary graphs $A$, $C$ and $D$.

Note that $C = A[v \leftarrow C] \in SPG$ and $D = A[v \leftarrow D] \in SPG$. We say that $v \leftarrow C$ and $v \leftarrow D$ are *elementary* superpositions. It can be shown that the class $SPG$ is closed under superposition [10].

**Definition 4.** A *structurally synthesized binary decision diagram* $\mathscr{D}(P)$ for a propositional formula $P$ is a superpositional graph defined inductively on the structure of $P$ as follows:

1° if $P$ is a literal $l$ then $\mathscr{D}(P)$ is a graph $A$ where the root is labelled by $l$;
2° if $P = Q\&R$ then $\mathscr{D}(P) = C[u \leftarrow \mathscr{D}(Q)][v \leftarrow \mathscr{D}(R)]$;
3° if $P = Q \vee R$ then $\mathscr{D}(P) = D[u \leftarrow \mathscr{D}(Q)][v \leftarrow \mathscr{D}(R)]$.

Binary graphs $A$, $C$, and $D$ in Figure 2 are SSBDDs for formulae $v$, $u\&v$ and $u \vee v$ respectively. The resulting SSBDD of Figure 1 corresponds to the formula

$$a\&\Big(\big((b \vee c)\&d\big) \vee e\Big)\&f\Big).$$

The notions of activated path and Boolean function represented by an SSBDD are similar to the case of BDD. The only difference is that, in order to choose the next node in the path, we have to evaluate literals instead of variables.

**Theorem 1.** *[11] A propositional formula $P$ and its SSBDD $\mathscr{D}(P)$ represent the same Boolean function.*

## 3 Theory of Superpositional Graphs

An SSBDD is a superpositional graph whose internal nodes are labelled with literals. Therefore, testing if a BDD is an SSBDD reduces to checking if its underlying binary graph is a superpositional graph.

Let $G$ be a binary graph with $n$ internal nodes. An obvious way for testing is to generate all sequences of superpositions of length $n-1$ and check if some of them ends up with $G$. There are $(n-1)! \cdot 2^{n-1}$ possible sequences of superpositions.

In [10], Peder and Tombak proved necessary and sufficient conditions for a binary graph to be a superpositional graph. These conditions do not involve the notion of superposition and allow to deduce linear time algorithms for testing superpositionality and for finding the sequence of superpositions that generates the given superpositional graph. We reproduce here the necessary definitions and theorems (without proofs) from [10].

**Definition 5.** A binary graph $G$ is *traceable* if there exists a directed path through all internal nodes of $G$ (Hamiltonian path).

A binary graph is acyclic, therefore, if the Hamiltonian path exists then it is unique.

**Theorem 2.** *Every superpositional graph is traceable.*

Theorem 2 gives a canonical enumeration of the nodes of a superpositional graph. Given the canonical enumerations of $G, H \in SPG$, we can test the isomorphism between $G$ and $H$ in time $O(n)$ (we must check the endpoints of all edges and there are $2n$ edges in an $n$-node binary graph).

Finding the Hamiltonian path of a binary graph $G$ is a special case of the classical task of topological sorting of the nodes of a graph.

**Definition 6.** A binary graph $G$ is *homogenous* if only one type of edges (i.e. either 1-edges only or 0-edges only) enter into every node $v \in V(G)$.

**Definition 7.** The edges $v_k \to v_p$ and $v_l \to v_r$ of a binary traceable graph are *crossing edges* if $k < l < p < r$.

**Definition 8.** We say that a binary traceable graph is *strongly planar* if there are no crossing 0-edges and no crossing 1-edges.

Strong planarity has the following intuitive interpretation: stretch the graph so that all nodes are in a straight line in their canonical order and there are no 0-edges above the line and no 1-edges below the line. If the binary graph is strongly planar then there are no crossing arrows in such drawing. Figure 3 depicts the superpositional graph before and after stretching.

It is obvious that if a binary graph is strongly planar then it is also planar, while the opposite does not hold in general.

**Definition 9.** We say that a binary traceable graph is 1-*cofinal (*0-*cofinal)* if all 1-edges (0-edges) starting between the endpoints of some 0-edge (1-edge) and crossing it end in the same node.

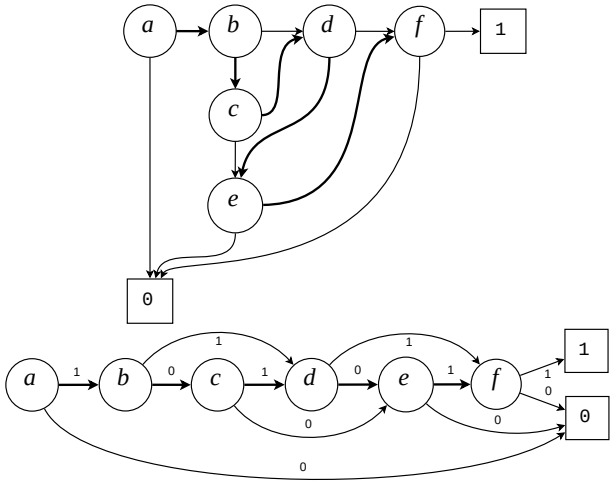


Fig. 3. A superpositional graph before and after stretching. Bold arrows mark the Hamiltonian path.

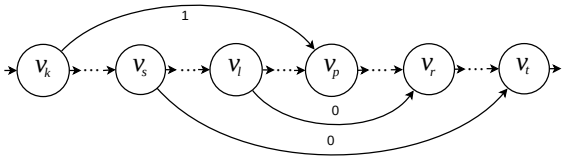


Fig. 4. Situation forbidden by 0-cofinality. For establishing 0-cofinality, one of the edges ending at $v_r$ and $v_t$ must be redirected to the other vertex.

Figure 4 illustrates the situation, forbidden by 0-cofinality.

**Definition 10.** A binary traceable graph is *cofinal* if it is both 1-cofinal and 0-cofinal.

**Lemma 1.** *If $G$ is a traceable strongly planar cofinal binary graph with $n > 2$ internal nodes then it can be represented as a superposition $G = H[w \leftarrow F]$ where $H$ and $F$ are binary graphs with at least 2 nodes.*

Algorithm 3 in Section 4 imitates the decomposition shown by the proof of this lemma, therefore we reproduce a sketch of the proof from [10] here.

*Proof.* Let $v_1, \ldots, v_n$ be the canonical sequence of internal nodes of the graph $G$. Suppose we have a subsequence $v_k, v_{k+1}, \ldots, v_l$ $(k < l)$ satisfying the following:

1. all edges from nodes $v_1, \ldots, v_{k-1}$ to the nodes of the subsequence are pointing to $v_k$;

2. all 1-edges from the nodes of the subsequence to nodes $v_{l+1},\ldots,v_n,1$ are pointing to the same node;
3. all 0-edges from the nodes of the subsequence to nodes $v_{l+1},\ldots,v_n,0$ are pointing to the same node.

Then we can construct binary graphs $H$ and $F$ as follows. The set of nodes and edges of graph $H$ are $V(H)=\{v_1,\ldots,v_{k-1},w,v_{l+1},\ldots,v_n,0,1\}$ where $w$ is a fresh node and

$$\begin{aligned} E(H) \;=\; & \{(u,v):\ u,v\in V(H)\setminus\{w\},(u,v)\in E(G)\}\cup \\ & \{(u,w):\ u\in\{v_1,\ldots,v_{k-1}\},(u,v_k)\in E(G)\}\cup \\ & \{(w,z):\ z\in\{v_{l+1},\ldots,v_n,0,1\},\exists i_{k\le i\le l}((v_i,z)\in E(G))\}, \end{aligned}$$

respectively. The set of nodes and edges of graph $F$ are $V(F)=\{v_k,\ldots,v_l,0,1\}$ and

$$\begin{aligned} E(F) \;=\; & \{(u,v):\ u,v\in V(F),(u,v)\in E(G)\}\cup \\ & \{(v_i,1):\ v_i\in V(F),\exists z_{z\in\{v_{l+1},\ldots,v_n,1\}}(\mathrm{high}(v_i)=z)\}\cup \\ & \{(v_i,0):\ v_i\in V(F),\exists z_{z\in\{v_{l+1},\ldots,v_n,0\}}(\mathrm{low}(v_i)=z)\}, \end{aligned}$$

respectively. By construction, $G=H[w\leftarrow F]$. There are the following four cases:

1. $\mathrm{high}(v_1)=1$. Then $\mathrm{low}(v_1)=v_2$ and the desired subsequence is $v_2,\ldots,v_n$.
2. $\mathrm{low}(v_1)=0$. Then $\mathrm{high}(v_1)=v_2$ and the desired subsequence is $v_2,\ldots,v_n$.
3. $\mathrm{high}(v_1)=v_{l+1}$ where $1<l\le n-1$. Then $\mathrm{low}(v_1)=v_2$ and the desired subsequence is $v_1,\ldots,v_l$.
4. $\mathrm{low}(v_1)=v_{l+1}$ where $1<l\le n-1$. Then $\mathrm{high}(v_1)=v_2$ and the desired subsequence is $v_1,\ldots,v_l$.

□

**Theorem 3.** *A binary graph is superpositional iff it is a traceable strongly planar cofinal graph.*

For example, Figure 5 contains two binary graphs. In order to check whether they are superpositional graphs, we stretch them by the canonical ordering of nodes. The leftmost graph in Figure 5 is depicted in Figure 6 and the rightmost graph in Figure 7. It is obvious that both are traceable and strongly planar. The graph in Figure 6 is not cofinal since nodes $v$ and $w$ lie between the endpoints of the 1-edge $u\to x$ but 0-edges $v\to z$ and $w\to y$ point to different nodes. According to Theorem 3, the graph is not superpositional. The graph in Figure 7 is cofinal and, therefore, it is a superpositional graph.

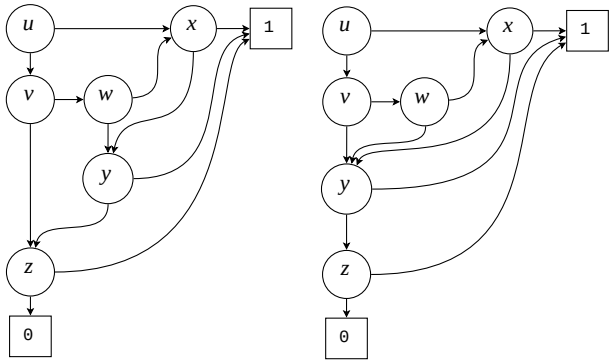


Fig. 5. Two binary graphs.

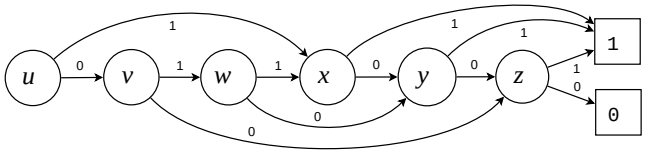


Fig. 6. Leftmost graph from Figure 5 in stretched form.

## 4 Algorithms

In this section, we present linear-time algorithms for three problems concerning superpositional graphs:

(I) testing whether a binary graph belongs to the class *SPG*;

(II) finding a sequence of superpositions that generates a given superpositional graph from trivial binary graphs (*C* and *D*);

(III) finding the propositional formula expressed by a given superpositional graph.

Time complexity is estimated w.r.t. the number *n* of internal nodes in the graph.

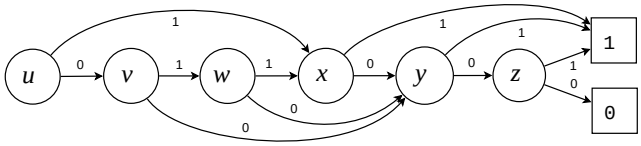


Fig. 7. Rightmost graph from Figure 5 in stretched form.

**Problem I: testing for any binary graph $G$ whether $G \in SPG$.**

According to Theorem 3 and the definitions given in Section 3, establishing superpositionality can be reduced to the following steps:

0. Checking whether $G$ is traceable and, if yes, finding the canonical order of its nodes.

1a. Checking whether $G$ is without crossing 0-edges.

1b. Checking whether $G$ is without crossing 1-edges.

2a. Checking whether $G$ is 0-cofinal.

2b. Checking whether $G$ is 1-cofinal.

At each step, a negative outcome implies also negative outcome of the whole task. If all steps are answered positively then $G$ is superpositional.

As mentioned above, step 0 is a special case of the task of topological sorting. This can be done in time linear w.r.t. the number of nodes. We describe algorithms for steps 1b and 2b below; steps 1a and 2a are dual to them. The algorithm descriptions make use of the following auxiliary functions:

- number$(v)$ for each node $v$ returns the index of node $v$ in the canonical enumeration of nodes. Suppose counting starts from 1; for terminal nodes $v$, number$(v)$ returns $n+1$;
- incom$_1(v)$ and incom$_0(v)$ for each internal node $v$ return the number of long 1-edges and 0-edges, respectively, incoming into $v$, where an edge is called *long* if it does not belong to the Hamiltonian path.

The values of these functions can be computed in one traversal of the graph and memorized at each node, therefore we may assume that they work with time $\mathcal{O}(1)$.

During the work of both algorithms, an auxiliary stack $L$ is used where pairs consisting of the target node and type (0 or 1) of an edge are stored. Furthermore, additional mutable fields $v.d_1$ and $v.d_0$ at every internal node $v$ are used.

Algorithm 1 checks whether a given graph $G$ is without crossing 1-edges (step 1b in the schema above). After each execution of the body of the second for loop, stack $L$ contains the target node of each long 1-edge whose source node has been processed already but the target node not yet. The body of the while loop pops as many nodes from the stack as there exist long 1-edges pointing to the current node. If a target node different from the current node is popped, the stack must still contain the current node at least once; the edges that gave rise to these two records in the stack are crossing. Otherwise, the nodes popped during the while loop are precisely the nodes in the stack that are equal to the current node. If the graph contains a pair of crossing 1-edges, this is discovered while the algorithm is processing the target node of the first of them (whose numbers of endpoints are

lesser). Hence if the for loop finishes normally, the graph contains no crossing 1-edges.

**Algorithm 1.**

```
isWithoutCrossing1 (binary graph G) {
    for (all internal nodes v) v.d1 = 0;
    L = {};
    for (all internal nodes v in the canonical order) {
        while (v.d1 < incom1(v)) {
            (p,x) = L.pop();
            if (number(p) > number(v)) return false;
            v.d1++;
        } // while
        if (number(high(v)) > number(v) + 1)
        // if the 1-edge exiting from v_i is long
            L.push(high(v), 1);
    } // for
    return true;
}
```

During the work of Algorithm 1, bodies of the for loops are executed once for each internal node but the body of the while loop is executed once for each long 1-edge of the graph. As there is at most one long 1-edge outgoing from each node, the number of executions of the body of the while loop is linear and therefore the overall complexity of Algorithm 1 is linear as well.

Algorithm 2 for testing 1-cofinality is similar to the previous algorithm. It uses an additional variable *target* for holding the common target node of all 1-edges that start between the endpoints of the 0-edges that end at the node under consideration. If no outgoing 1-edge is detected yet then the variable holds the first in the canonical order node of $G$ (returned by function *start*).

Unlike Algorithm 1, this algorithm keeps track of both the long 0-edges and the long 1-edges. The while loop runs until all long 0-edges pointing to the current node are popped out of the stack. For the graph to be 1-cofinal, all target nodes of 1-edges met during this and having larger index than that of the current node must be equal since they all start between the endpoints of the 0-edge whose record is deepest in the stack. If no divergence of the target nodes is discovered, one of the records of the 1-edges (if there was any) must be pushed back to the stack since it can form a counterexample to 1-cofinality together with a 1-edge that will be encountered during a later execution of the while loop.

**Algorithm 2.**

```
isCofinal1 (strongly planar binary graph G) {
  for (all internal nodes v) v.d0 = 0;
  L = {};
  for (all internal nodes v in the canonical order) {
    target = start(G);
    while (v.d0 < incom0(v)) {
      (p,x) = L.pop();
      if (x == 1 and number(p) > number(v)) {
        if (target == start(G))   // the first outgoing 1-edge
          target = p;
        if (target<>p)   // two 1-edges pointing to different nodes
          return false;
      } // if
      if (p == v) v.d0++;
    } // while
    if (target<>start(G)) L.push(target,1);
    // push back one representative of the 1-edges
    if (number(high(v)) > number(v)+1)
      L.push(high(v),1);
    if (number(low(v)) > number(v)+1)
      L.push(low(v),0);
  } // for
  return true;
}
```

During the work of Algorithm 2, bodies of the for-loops are executed once for each internal node like in Algorithm 1. The body of the while-loop of Algorithm 2 is executed once for each long 0-edge of the graph but additionally once for each 1-edge stored in the stack. But note that, at processing each internal node, at most two 1-edges are pushed into the stack; hence the overall number of 1-edges popped does not exceed $2n$. Consequently, this algorithm works in linear time.

Altogether, we have obtained a method for checking superpositionality in linear time since each of the 5 steps works in linear time.

**Problem II: finding superpositions that generate a given $G \in SPG$.**

Algorithm 3 reconstructs a potential sequence of superpositions that would generate a given graph $G \in SPG$. Decomposition of the graph into two smaller ones is done in the way described in the proof of Lemma 1. Then both parts are decomposed

recursively. Graphs $C$ and $D$ are base cases with no further decomposition. (The graph $A$ is trivial and left out from the treatment.)

The output of the algorithm is given in the form of a binary tree, every node denoting either $C$ or $D$. According to the proof of Lemma 1, decomposition always results in two segments of consecutive nodes, whereby either the second segment is to be plugged for the last node of the first segment or the first segment is to be plugged for the first node of the second segment. In the decomposition tree, denote the two cases by arrows $\curvearrowleft$ and $\curvearrowright$, respectively. The node is not stored as it can be deduced from the direction of the arrow.

In the algorithm description below, $\text{size}(G)$ returns the number of internal nodes of $G$. Functions $\text{succ}(v)$ and $\text{pred}(v)$ return the successor and predecessor of $v$ in the canonical order, respectively. Function $\text{jump}(v)$ returns the target of the long edge starting from $v$. Furthermore, $v$.**clone**() makes a copy of node $v$ and returns it. The Python-like notation $G[:v]$ denotes the part of graph $G$ that contains only nodes that occur before $v$ (excl.) in the canonical order. Likewise, $G[v:]$ denotes the part of graph $G$ from $v$ (incl.) to the end.

**Algorithm 3.** *(Sketch.)*

**decompose** *(non-trivial superpositional graph G)* {
  **if** $(G == C)$ **return** *leaf "C";*
  **if** $(G == D)$ **return** *leaf "D";*
  **if** $(\text{number}(\text{jump}(\text{start}(G))) > \text{size}(G))$ {
    $w = \text{succ}(\text{start}(G))$.**clone**();
    $H = G[:\text{succ}(\text{succ}(\text{start}(G)))]$;
    $F = \{w\} \cup G[\text{succ}(\text{succ}(\text{start}(G))):]$;
    **return** $decompose(H) \curvearrowleft decompose(F)$;
  } **else** {
    $w = \text{start}(G)$.**clone**();
    $H = \{w\} \cup G[\text{jump}(\text{start}(G)):]$;
    $F = G[:\text{jump}(\text{start}(G))]$;
    **return** $decompose(F) \curvearrowright decompose(H)$;
  }
}

Checking whether a graph is $C$ or $D$ can be performed in constant time. Clearly, function jump can operate in constant time (it suffices to check the two edges going out from the node). The place at the Hamiltonian path where the graph is to be divided into two, as well as the sizes of the two parts, can therefore be found within constant time. The sizes can be given together with the graph to all recursive calls of *decompose*, so the size can always be found in constant time.

There is no need to do the actual division of the set of nodes, one just has to remember the borders. Each edge that is pointing beyond the border must be interpreted as going to the terminal corresponding to its type. This way, all operations except for the recursive calls can be performed in time $\mathscr{O}(1)$. Since there must be less than $n$ places of decomposition, the complexity of the algorithm is $\mathscr{O}(n)$.

**Problem III: finding a propositional formula encoded by a given $G \in SPG$.**

The straightforward way to find the formula would be to construct the decomposition tree and process this, replacing occurrences of $C$ and $D$ with conjunctions and disjunctions of two variables and performing all superpositions as substitutions. This relies on the following fact.

**Theorem 4.** *Let $P$, $Q$ be propositional formulae where all literals are different variables and let $x$ be a variable occurring in $P$. Then $\mathscr{D}(P[x \leftarrow Q]) = \mathscr{D}(P)[v \leftarrow \mathscr{D}(Q)]$ where, in the l.h.s., $P[x \leftarrow Q]$ denotes the substitution of $Q$ into $P$ for $x$ and, in the r.h.s., $v$ denotes the node of $\mathscr{D}(P)$ labeled by $x$.*

*Proof.* By induction on the structure of formula $P$, using the following property: if $H, G, F \in SPG$ and $v$, $u$ are nodes in $H$, $G$, respectively, then $(H[v \leftarrow G])[u \leftarrow F] = H[v \leftarrow G[u \leftarrow F]]$. (This property can be proven by straightforward case study.) □

Alas, complexity of a substitution on a formula tree depends linearly on the depth of the variable node which is asymptotic to $n$ in the worst case. For example, the superpositional graphs where the endpoints of each long edge are at distance 2 along the Hamiltonian path give rise to decomposition trees of form $G_1 \curvearrowright (G_2 \curvearrowright \ldots \curvearrowright G_l)$ where $G_i$ are alternately $C$ and $D$ (the first can be either one). If the substitutions encoded by the decomposition tree are followed naively, each new substitution is performed one level deeper, leading to quadratic time complexity.

Therefore, a linear-time solution must reorder the superpositions. For that, Algorithm 4 below uses an auxiliary function *helper* that, additionally to the decomposition tree, takes two formulas as arguments, the first of which has to be later substituted for the leftmost variable and the other for the rightmost variable. This way, the only direct substitutions are made into the elementary formulas that correspond to $C$ and $D$ and the overall time complexity becomes linear. The *helper* function uses pattern matching on the decomposition tree structure, expressed by the **switch** and **case** keywords.

In the constructed formula, variables are not distinguished for keeping the algorithm maximally simple (each variable is marked by *var*). The real variables can be determined afterwards within one traversal by examining the formula side by side with the original SSBDD (the canonical order in the graph corresponds to the pre-order of the formula tree).

**Algorithm 4.**

```
toFormula (superpositional graph G) {
    if (G == A) return var;
    return helper(decompose(G),var,var);
}

helper (decomposition tree T, formula L, formula R) {
    switch(T) {
        case leaf "C": return L&R;
        case leaf "D": return L ∨ R;
        case U ↶ V: return helper(U,L,helper(V,var,R));
        case U ↷ V: return helper(V,helper(U,L,var),R);
    }
}
```

## 5 Related work

The construction of superpositional graphs is related to that of some other combinatorial objects that have been investigated before, for instance bracketings and separable permutations [12,13] . Thus problem II is closely related to, for instance, the problem of parsing bracketings or checking whether a given permutation is separable by finding a parse tree for it if possible. There are linear algorithms also for these problems (for instance, [13] presents such an algorithm for separable permutations). Typically, the main difficulty in solving this kind of problems comes from the fact that the exact decomposition point cannot be determined in constant time. In the case of superpositional graphs, finding a decomposition point is trivial due to the preprocessing that extracts the Hamiltonian path.

## 6 Conclusion

In this paper, linear time algorithms are given for testing if a given binary graph is superpositional and for restoring a sequence of superpositions that generates a given superpositional graph. These algorithms can be used to improve the efficiency of the algorithms used for diagnostic modelling of logic circuits with SSBDDs.

### Acknowledgments

The work has been supported by Estonian Science Foundation grants 7068, 7483, 7543, EC FP7 IST project DIAMOND, and Research Centre CEBE funded by EU Structural Funds.

The authors thank their colleagues Peeter Laud and Reimo Palm from University of Tartu for their help in improving the superpositionality test algorithms.

The authors would like to thank referees for carefully reading the manuscript and giving their valuable suggestions.

## References

[1] C. Lee, "Representation of switching circuits by binary decision diagrams," *Bell. Syst. Tech. Journal*, vol. 38, pp. 985–999, 1959.

[2] R. Ubar, "Test generation for digital circuits using alternative graphs," in *Proc. Tallinn Technical University*, no. 409, Tallinn TU, Tallinn, Estonia, 1976, pp. 75–81, in Russian.

[3] S. B. Akers, "Functional testing with binary decision diagrams," *J. of Design Automation and Fault-Tolerant Computing*, vol. 2, pp. 311–331, 1978.

[4] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transaction on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[5] S. Minato, *BDDs and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[6] T. Sasao and M. Fujita, *Representations of Discrete Functions*. Kluwer Academic Publishers, 1996.

[7] R. Drechsler and B. Becker, *Binary Decision Diagrams, Theory and Implementation*. Kluwer Academic Publishers, 1998.

[8] R. Ubar, "Test synthesis with alternative graphs," *IEEE Design and Test of Computers*, pp. 48–57, 1996.

[9] A. Jutman, J. Raik, and R. Ubar, "SSBDDs: Advantageous model and efficient algorithms for digital circuit modeling, simulation and test," in *Proc. 5th Int. Workshop on Boolean Problems*, Freiberg, Germany, 2002, pp. 157–166.

[10] A. Peder and M. Tombak, "Superpositional graphs," *Acta et Commentationes Universitatis Tartuensis de Mathematica*, vol. 13, pp. 51–64, 2009. [Online]. Available: http://www.ut.ee/~ahtip/LOG_COMP/PederTombakACTA2009.pdf

[11] A. Jutman, A. Peder, J. Raik, M. Tombak, and R. Ubar, "Structurally synthesized binary decision diagrams," in *Proc. 6th International Workshop on Boolean Problems*, Freiberg University, Germany, 2004, pp. 271–278.

[12] R. P. Stanley, "Hipparchus, Plutach, Schröder and Hough," *Am. Math. Monthly*, vol. 104, pp. 344–350, 1997.

[13] P. Bose, J. Buss, and A. Lubiw, "Pattern matching for permutations," *Information Processong Letters*, vol. 65, pp. 277–283, 1998.