Heuristic NPN classification for large functions using AIGs and LEXSAT

Mathias Soeken[1], Alan Mishchenko[2], Ana Petkovska[1], Baruch Sterin[2], Paolo Ienne[1], Robert K. Brayton[2], and Giovanni De Micheli[1]

[1] EPFL, Lausanne, Switzerland
[2] UC Berkeley, CA, USA
mathias.soeken@epfl.ch

Abstract. Two Boolean functions are NPN equivalent if one can be obtained from the other by negating inputs, permuting inputs, or negating the output. NPN equivalence is an equivalence relation and the number of equivalence classes is significantly smaller than the number of all Boolean functions. This property has been exploited successfully to increase the efficiency of various logic synthesis algorithms. Since computing the NPN representative of a Boolean function is not scalable, heuristics have been proposed that are not guaranteed to find the representative for all functions. So far, these heuristics have been implemented using the function's truth table representation, and therefore do not scale for functions exceeding 16 variables.
In this paper, we present a symbolic heuristic NPN classification using And-Inverter Graphs and Boolean satisfiability techniques. This allows us to heuristically compute NPN representatives for functions with much larger number of variables; our experiments contain benchmarks with up to 194 variables. A key technique of the symbolic implementation is SAT-based procedure LEXSAT, which finds the lexicographically smallest satisfiable assignment. To our knowledge, LEXSAT has never been used before in logic synthesis algorithms.

1 Introduction

Researchers have intensively studied the classification of Boolean functions in the past. One of the frequently used classifications is based on NPN equivalence[7, 8, 5, 16, 11]. Two Boolean functions f and g are *Negation-Permutation-Negation (NPN) equivalent*, denoted $f =_{\mathrm{NPN}} g$, if one can be obtained from the other by negating (i.e., complementing) inputs, permuting inputs, or negating the output. This notion of equivalence is motivated by the logic representation because NPN equivalent functions are invariant to the "shape" of an expression. As an example, $(x_1 \lor x_2) \land \bar{x}_3 =_{\mathrm{NPN}} (\bar{x}_2 \lor x_3) \land x_1$ by replacing $x_1 \leftarrow \bar{x}_2, x_2 \leftarrow x_3$, and $\bar{x}_3 \leftarrow x_1$. This is especially true in frequently used logic representations, such as *And-Inverter Graphs (AIGs)* [9, 15], in which negations are represented by complemented edges. An *equivalence class* represents a set of functions such that each two functions belonging to the class are NPN equivalent. For each equivalence class, one function, called *representative*, is selected uniquely among all functions of the class.

Exact NPN classification refers to the problem of finding, for a Boolean function f, its representative $r(f)$ of an NPN class. Thus, an exact NPN classification algorithm can be used to decide NPN equivalence of two functions f and g, since $r(f) = r(g) \Leftrightarrow f =_{\text{NPN}} g$. Often $r(f)$ is chosen to be the function in the equivalence class that has the smallest truth table representation. The smallest truth table is the one with the smallest integer value, when considering the truth table as a binary number. To the best of our knowledge, there is no better way to exactly find $r(f)$ than to exhaustively enumerate all $n!$ permutations and all 2^{n+1} negations. To cope with this complexity, *heuristic NPN classification* has been proposed that computes $\tilde{r}(f) \geq r(f)$, i.e., it may not necessarily find the smallest truth table representation. Since $\tilde{r}(f) = \tilde{r}(g) \Rightarrow f =_{\text{NPN}} g$, heuristic NPN classification algorithms are nevertheless very helpful in many applications. For example, it has been applied to logic optimization [18, 21] and technology mapping [1, 16, 4, 10]. Finally, another option is to perform *Boolean matching* that directly checks if $f =_{\text{NPN}} g$, without first generating the representatives. Several algorithms solve this problem [1], but to the best of our knowledge, there is no good heuristics for Boolean matching without the representative.

The existing algorithms for heuristic NPN computation are implemented using truth tables [10, 5] and therefore can only be efficiently applied to functions with up to 16 variables. The operations that are performed on the functional representation in heuristic NPN classification algorithms are typically: i) negating an input, ii) negating an output, iii) swapping two inputs, and iv) comparing two functions. When using truth tables as an underlying representation, all these operations can easily be implemented.

In this paper, we propose an implementation of two existing heuristic NPN classification algorithms using AIGs and LEXSAT. While the first three of the above described operations are simple to implement in AIGs, the comparison of two functions is difficult. Thus, our algorithm uses LEXSAT, which is a variant of the Boolean satisfiability (SAT) problem in which the lexicographically smallest (or greatest) assignment is returned if the problem is satisfiable. Our experimental evaluations show that, by using AIGs as function representation together with LEXSAT, heuristic NPN classification can be applied to functions with up to 194 variables while providing the same quality as the truth table based implementation at the cost of additional runtime.

The paper is structured as follows. Section 2 introduces necessary background. Section 3 reviews two existing heuristic NPN classification algorithms. Section 4 describes the truth table based implementation of the algorithms and the proposed AIG based implementation using LEXSAT. Section 5 shows the results of the experimental evaluation and Sect. 6 discusses possible performance improvements. Section 7 concludes the paper.

2 Preliminaries

2.1 Boolean functions

We assume that the reader is familiar with Boolean functions. A *Boolean function* $f : \mathbb{B}^n \to \mathbb{B}$ evaluates bitvectors $x = x_1 \dots x_n$ of size n. If $f(x) = 1$, we call x a *satisfying assignment.* Given two assignments $x = x_1 \dots x_n$ and $y = y_1 \dots y_n$, we say that x is *lexicographically smaller* than y, denoted $x < y$, if there exists a $k \geq 1$ such that $x_k < y_k$ and $x_i = y_i$ for all $i < k$. We use the notation x^1 and x^0 to refer to x and $\bar{x}$, respectively.

The truth table of f is the bitstring obtained by concatenating the function values for the assignments $0 \dots 00, 0 \dots 01, \dots, 1 \dots 10, 1 \dots 11$ in the given order, i.e., the lexicographically smallest assignment $0 \dots 00$ corresponds to the most significant bit. The notion of lexicographic order can be transferred to truth tables. We also use f to refer to its truth table representation, if it is clear from the context. We use $\top$ and $\bot$ to refer to tautology and contradiction, respectively.

2.2 NPN equivalence

Definition 1 (NPN equivalence, [7]). *Let $g(x_1, \dots, x_n)$ be a Boolean function. Given a permutation $\pi \in S_n$ (S_n is the symmetric group over n elements) and a phase-bitvector $\varphi = (p, p_1, \dots, p_n) \in \mathbb{B}^{n+1}$, we define*

$$\mathit{apply}(g, \pi, \varphi) = g^p(x_{\pi(1)}^{p_1}, \dots, x_{\pi(n)}^{p_n}). \tag{1}$$

We say that two Boolean functions $f(x_1, \dots, x_n)$ and $g(x_1, \dots, x_n)$ are Negation-Permutation-Negation (NPN) equivalent*, if there exists $\pi \in S_n$ and $\varphi \in \mathbb{B}^{n+1}$ such that $f(x_1, \dots, x_n) = \mathit{apply}(g, \pi, \varphi)$, i.e., g can be made equivalent to f by $\underline{n}$egating inputs, $\underline{p}$ermuting inputs, or $\underline{n}$egating the output. We call the pair π, φ an* NPN signature*.*

NPN equivalence is an equivalence relation that partitions the set of all Boolean functions over n variables into a smaller set of NPN classes. As an example, all 2^{2^n} Boolean functions over n variables can be partitioned into 2, 4, 14, 222, 616126 NPN classes for $n = 1, 2, 3, 4, 5$ [8].

Definition 2 (NPN classes and representatives). *We refer to the* NPN class *of a function f as $[f]$ and define $f =_{\text{NPN}} g$, if and only if $[f] = [g]$. As the* representative *$r(f)$ of each NPN class $[f]$, we take the function with the smallest truth table. In other words, $r(f) \leq g$ for all $g \in [f]$.*

Example 1. The truth table of $x_1 \lor x_2$ is 0111, and its NPN representative is $x_1 \land x_2 = \overline{\bar{x}_1 \lor \bar{x}_2}$ which truth table is 0001. Note that $g =_{\text{NPN}} \mathit{apply}(g, \pi, \varphi)$ for all $\pi \in S_n$ and all $\varphi \in \mathbb{B}^{n+1}$. For a detailed introduction into NPN classification the reader is referred to the literature [19, 1].

Alg. 1. LEXSAT implementation from [13, Ex. 7.2.2.2-109].

```
   Input  : Boolean function $f(x_1, \dots, x_n)$
   Output : $\min f$ if $f \neq \bot$, otherwise *unsatisfiable*
1  set $s \leftarrow \mathrm{SAT}(f)$;
2  if $s =$ unsatisfiable then return unsatisfiable;
3  set $y_1, \dots, y_n \leftarrow s$ and $y_{n+1} \leftarrow 1$;
4  set $d \leftarrow 0$;
5  while true do
6      set $d \leftarrow \min\{j > d \mid y_j = 1\}$;
7      if $d > n$ then return $y_1, \dots, y_n$;
8      set $s \leftarrow \mathrm{SAT}(f, \{x_1^{y_1}, \dots, x_{d-1}^{y_{d-1}}, \bar{x}_d\})$;
9      if $s \neq$ unsatisfiable then set $y_1, \dots, y_n \leftarrow s$;
10 end
```

2.3 Lexicographic SAT

Definition 3 (Lexicographically smallest (greatest) assignment). *Let* $f : \mathbb{B}^n \to \mathbb{B}$ *with* $f \neq \bot$. *Then* $x \in B^n$ *is the* lexicographically smallest assignment *of* f*, if* $f(x) = 1$ *and* $f(x') = 0$ *for all* $x' < x$. *We denote this* x *as* $\min f$. *Analogously,* $x \in B^n$ *is the* lexicographically greatest assignment of f*, if* $f(x) = 1$ *and* $f(x') = 0$ *for all* $x' > x$. *We denote this* x *as* $\max f$.

Based on this definition, *lexicographic SAT* (LEXSAT) is a decision procedure that for a given function f, returns $\min f$ when the problem is satisfiable, or returns *unsatisfiable*, when $f = \bot$. LEXSAT is NP-hard and complete for the class $\mathsf{FP}^{\mathsf{NP}}$ [14]. Knuth [13] proposes an implementation that calls a SAT solver several times to refine the assignment, and which is described in Alg. 1. In the algorithmic description $\mathrm{SAT}(f, a)$ refers to the default SAT decision procedure for a Boolean function f and optional assumptions a that allows incremental solving. SAT returns *unsatisfiable*, when $f = \bot$, or a satisfying assignment, when the problem is satisfiable. Further, to use the SAT solver, we assume that f is translated into a CNF representation (e.g., using [22, 6]). By replacing $y_j = 1$ with $y_j = 0$ in Line 6 and $\bar{x}_d$ with x_d in Line 8, Alg. 1 can find $\max f$.

3 Heuristic NPN classification

To the best of our knowledge there is no efficient algorithm to find the representative $r(f)$ for a given Boolean function $f : \mathbb{B}^n \to \mathbb{B}$, and an exhaustive exploration of all functions in $[f]$ is required. More efficient algorithms can be found when using a heuristic to *approximate* the representative. Such heuristics do not visit all functions in $[f]$ and therefore are not guaranteed to find $r(f)$, only a function $\tilde{r}(f) \geq r(f)$ that is locally minimal to all visited ones is returned.

In the following subsections, we describe in detail two heuristic NPN classification algorithms from the literature for which truth table based implementations

Alg. 2. Flip-swap heuristic for NPN classification.

```
   Input  : Boolean function $f \in \mathbb{B}^n \to \mathbb{B}$
   Output : NPN signature $\pi \in S_n, \varphi \in \mathbb{B}^{n+1}$
1  set $\pi \leftarrow \pi_e$ and $\varphi \leftarrow 0 \dots 0$;
2  repeat
3      set $improvement \leftarrow 0$;
4      for $i = 0, \dots, n$ do
5          if $apply(f, \pi, \varphi \oplus 2^i) < apply(f, \pi, \varphi)$ then
6              set $\varphi \leftarrow \varphi \oplus 2^i$;
7              set $improvement \leftarrow 1$;
8          end
9      end
10     for $d = 1, \dots, n-2$ do
11         for $i = 1, \dots, n-d$ do
12             set $j \leftarrow i + d$;
13             if $apply(f, \pi \circ (i, j), \varphi) < apply(f, \pi, \varphi)$ then
14                 set $\pi \leftarrow \pi \circ (i, j)$;
15                 set $improvement \leftarrow 1$;
16             end
17         end
18     end
19 until $improvement = 0$;
20 return $\pi, \varphi$;
```

exists, e.g., in *ABC* [3]. The first heuristic [10] is called *flip-swap*, which in alternating steps flips single bits and permutes pairs of indices. The second algorithm [10] is called *sifting* (inspired by BDD sifting [20]), which considers only adjacent indices for flipping and swapping.

3.1 Flip-swap heuristic

Algorithm 2 shows the flip-swap heuristic, which in alternating steps first tries to flip single bits in the phase φ and then permutes pairs of indices in π. The phase is initialized to consist only of 0, and the permutation is initially the identity permutation π_e. The first **for**-loop, which flips bits (Line 4), also covers negating the whole function when $i = 0$. Note that all index pairs in the second **for**-loop (Line 10), which swaps inputs, are enumerated ordered by their distance such that adjacent pairs are considered first. Flipping and swapping is repeated until no more improvement, i.e., no smaller representative, can be found. The algorithm returns a permutation π and a phase-bitvector φ, which lead to the smallest function that is considered as representative $\tilde{r}(f)$ of the input function f. One can obtain $\tilde{r}(f)$ by executing $apply(f, \pi, \varphi)$.

Alg. 3. Sifting heuristic for NPN classification.

```
    Input  : Boolean function $f \in \mathbb{B}^n \to \mathbb{B}$
    Output : NPN signature $\pi \in S_n, \varphi \in \mathbb{B}^{n+1}$
 1  set $\pi \leftarrow \pi_e$ and $\varphi \leftarrow 0 \dots 0$;
 2  repeat
 3      set improvement $\leftarrow 0$;
 4      for $i = 1, \dots, n-1$ do                         /* in alternating order */
 5          set $\sigma, \hat{\sigma} \leftarrow \pi_e$ and $\psi, \hat{\psi} \leftarrow 0 \dots 0$;
 6          for $j = 1, \dots, 8$ do
 7              if $4 \mid j$ then
 8                  set $\sigma \leftarrow \sigma \circ (i, i+1)$;
 9              end
10              else if $2 \mid j$ then
11                  set $\psi \leftarrow \psi \oplus 2^{i+1}$;
12              end
13              else
14                  set $\psi \leftarrow \psi \oplus 2^i$;
15              end
16              if apply$(f, \pi \circ \sigma, \varphi \oplus \psi) <$ apply$(f, \pi \circ \hat{\sigma}, \varphi \oplus \hat{\psi})$ then
17                  set $\hat{\sigma} \leftarrow \sigma$ and $\hat{\psi} \leftarrow \psi$;
18                  set improvement $\leftarrow 1$;
19              end
20          end
21          set $\pi \leftarrow \pi \circ \hat{\sigma}$ and $\varphi \leftarrow \varphi \oplus \hat{\psi}$;
22      end
23  until improvement $= 0$;
24  return $\pi, \varphi$;
```

3.2 Sifting heuristic

Algorithm 3 shows the sifting heuristic that was presented by Huang et al. [10]. The idea is that a *window* is shifted over adjacent pairs of input variables $x_{\pi(i)}$ and $x_{\pi(i+1)}$. The adjacent variables are negated and swapped (Lines 6–15) in the following way that guarantees that all eight possibilities are obtained by applying simple operations, and the initial configuration is restored in the end.

$$xy \xrightarrow[j=1]{\bar{\square}\square} \bar{x}y \xrightarrow[j=2]{\square\bar{\square}} \bar{x}\bar{y} \xrightarrow[j=3]{\bar{\square}\square} x\bar{y} \xrightarrow[j=4]{\leftrightarrow} y\bar{x} \xrightarrow[j=5]{\bar{\square}\square} \bar{y}\bar{x} \xrightarrow[j=6]{\square\bar{\square}} \bar{y}x \xrightarrow[j=7]{\bar{\square}\square} yx \xrightarrow[j=8]{\leftrightarrow} xy \qquad (2)$$

Whenever j is a multiple of 4 (i.e., $j = 4$ and $j = 8$), the variables are swapped. Otherwise, whenever j is even (i.e., $j = 2$ and $j = 6$), the second variable is negated, and in all other cases the first variable is negated.

All eight configurations are evaluated and the best configuration is stored in $\hat{\sigma}$ and $\hat{\psi}$. This procedure is repeated as long as an improvement can be obtained. The window is moved in alternating order over the variables, i.e., first from left to right and then from right to left.

4 Implementations

The crucial parts in the algorithms presented in Sect. 3 are the checks in Lines 5 and 13 from Alg. 2, and Line 16 from Alg. 3, which derive the representative of the given function by applying the given permutation and phase. Four operations are required to perform these checks, which are i) negating an input, ii) negating an output, iii) swapping two inputs, and iv) comparing the two functions. This section describes two implementations for these four operations: i) based on truth tables which scales to functions with up to 16 variables and ii) based on AIGs and LEXSAT which scales for large functions.

4.1 Truth table based implementation

A truth table is represented as the binary expansion of a nonnegative number $t \in [0, 2^n)$. The most significant bit represents the assignment $0 \dots 0$ and the least significant bit represents the assignment $1 \dots 1$. In this representation the truth table of the function $x_1 \land x_2$ is 0001 and of $x_1 \lor x_2$ is 0111. The truth table for a variable x_{n-i} in a n-variable Boolean function is $\mu_{n,i} = \frac{2^{2^n}-1}{2^{2^i}+1}$ for $0 \le i < n$. For $n = 3$, we have 0000 1111, 0011 0011, and 0101 0101 for x_1, x_2, and x_3, respectively. The comparison of two functions given in their truth table representation is straightforward, e.g., by comparing their integer values. The other three operations can be implemented using well-known bitwise arithmetic as illustrated next. A truth table t for an n-variable Boolean function can be negated by flipping each bit, i.e., $t \leftarrow \bar{t}$. In order to negate the polarity of a variable x_i in a truth table t for an n-variable Boolean function, we compute $t \leftarrow \big((t \mathbin{\&} \mu_{n,i}) \ll 2^{n-i+1}\big) \mid \big((t \mathbin{\&} \bar{\mu}_{n,i}) \gg 2^{n-i+1}\big)$. The operations '&', '|', '$\ll$', '$\gg$' are bitwise AND, bitwise OR, logical left-shift, and logical right-shift. Two variables x_i and x_j with $i > j$ can be swapped in a truth table t by performing the two operations $t' \leftarrow (t \oplus (t \gg \delta)) \mathbin{\&} 2^j$ and $t \leftarrow t \oplus t' \oplus (t' \ll \delta)$, where $\delta = i - j$ and '$\oplus$' is bitwise XOR.

All the described operations can be implemented very efficiently when n is small. For example, a 6-variable function fits into a word that requires one memory cell on a 64-bit computer architecture. Almost all of the bitwise operations have a machine instruction counterpart that can be processed within one clock cycle. Warren, Jr. [23] and Knuth [12] describe all these bitwise manipulations and give more detailed equations.

4.2 AIG based implementation using LEXSAT

For large functions, we propose representing the function using the AIG data structure, which represents a logic network using two-input AND gates and edges connecting them. The edges may be complemented, representing inverters over these edges. For an AIG, negating the function, negating the polarity of a single variable, and swapping two variables is trivial. This can be achieved by complementing fanout edges from the primary output and primary inputs, and by swapping two AIG nodes of the corresponding primary inputs, respectively.

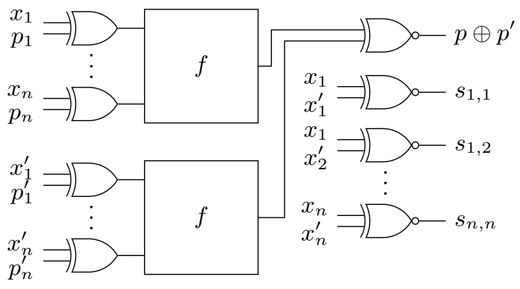

Fig. 1. Shared miter construction. The additional variables p_i and p'_i control the phase of the inputs and the output, while the additional variables $s_{i,j}$ allow permuting inputs.

Since the truth table is not explicitly represented by an AIG, the lexicographic comparison in Alg. 2 cannot be performed directly. We find that LEXSAT can be used to solve this problem. The next theorem, which is the main contribution of this paper, explains how the comparison can be done.

Theorem 1. *Let $g : \mathbb{B}^n \to \mathbb{B}$ and $h : \mathbb{B}^n \to \mathbb{B}$ be two Boolean functions. Then $g < h$, if and only if*

$$g \neq h \text{ and } g(\min(g \oplus h)) = 0.$$

Proof. '$\Leftarrow$': The condition can only hold if $g \neq h$. Then, $g \oplus h \neq \bot$ and its lexicographic smallest assignment $x = \min(g \oplus h)$ is the smallest assignment for which g and h differ. Hence, x is the first bit-position in which the truth table representations of g and h differ. (Recall that the most significant bit corresponds to all variables set to 0.) If $g(x) = 0$, then $h(x) = 1$, and g must be lexicographically smaller than h.

'$\Rightarrow$': Obviously, $g \neq h$. Let x be the smallest assignment for which g and h differ, i.e., $x = \min(g \oplus h)$. Since $g < h$, we have $g(x) = 0$. □

Based on Theorem 1, the following steps are necessary in an AIG based implementation to compute whether $g < h$.

1) Create an AIG for the *miter* $m(x_1, \dots, x_n) = g(x_1, \dots, x_n) \oplus h(x_1, \dots, x_n)$ by matching the inputs and pairing the outputs using an XOR operation [2].
2) Encode the AIG for m as a CNF.
3) Solve LEXSAT for the variables $x_1, \dots, x_n$ by assuming the output of m to be 1.
4) If a satisfying assignment x exists and if simulating $g(x)$ returns 0, then $g < h$.

In our use of this procedure, we will obtain g and h from *apply*(f, π, φ) and *apply*(f, π', φ'), respectively, and the result will determine whether the current permutation and phase-bitvectors will be updated.

Simple miter and shared miter approach. The lexicographic comparison has to be done several times by following the above mentioned steps that require to build and encode the miter each time. We call this the *simple miter* approach. Contrary to this approach, we propose a *shared miter* approach in which the miter is only created once and can be reconfigured. Then, steps 1) and 2) are performed only once. The shared miter is equipped with additional *inputs* $p_1, \dots, p_n, p'_1, \dots, p'_n$ and *outputs* $p, p', s_{1,1}, s_{1,2}, \dots, s_{n,n}$ to reconfigure w.r.t. different permutations and phases. These inputs and outputs can be assigned using assumption literals in the LEXSAT calls in step 3).

The details are illustrated in Fig. 1. The assumption literals to control the phase for inputs and outputs are $(p, p_1, \dots, p_n) = \varphi$ and $(p', p'_1, \dots, p'_n) = \varphi'$. Instead of assuming the output of m to be 1 in step 3), it is assumed to be $\bar{p} \oplus p'$ (note the XNOR gate at the outputs). To take input permutation into consideration, we assume $s_{i,j} = 1$, if $\pi_i = \pi'_j$. This is ensured by having a quadratic number of XNOR gates for each pair of inputs x_i and x'_j. The lexicographically smallest assignment is determined with respect to $x_1, \dots, x_n$. Technically, the simulation in step 4) is not required since the simulation value is contained in the satisfying assignment, however, the runtime required for simulation is negligible. As the experimental results from Sect. 5 show, the simple miter approach and the shared miter approach trade off solving time against encoding time: the LEXSAT instances in the simple miter approach are simpler to solve, however, more time is spent for encoding the AIG of the miter for each lexicographic comparison.

Encoding the AIG. The process of translating an AIG into a conjunctive normal form (CNF), which is a set of clauses, is called *encoding*. In order to call LEXSAT, the AIG of the miter needs to be encoded into a CNF. Several techniques for encoding exist, but the most conventional one is the *Tseytin encoding* [22] that introduces a new variable for each gate, and expresses the relation of the output to the inputs using clauses. For example, an AND gate $c = a \wedge b$ is encoded with the three clauses $(a \vee \bar{c})(b \vee \bar{c})(\bar{a} \vee \bar{b} \vee c)$. Since AIGs contain only AND gates, the SAT formula consists of clauses that have the form as the three clauses of the AND gate, with literals inverted with respect to complemented edges.

Particularly for AIGs, one can do much better by using logic synthesis techniques to derive a smaller set of clauses. We call this technique *EMS encoding* [6] due to the authors' last names. EMS encoding applies cut-based technology mapping in which the objective is not area or delay but clause count. Several applications indicate that this encoding is particularly desirable when CNFs are generated from circuits and have a positive effect on the SAT solving runtime [6]. However, due to the overhead of technology mapping, the EMS encoding requires more runtime than the Tseytin encoding.

Table 1. Evaluating the quality of the heuristics for small functions.

# Variables	# Functions	# Classes / Runtime (s)					
		Exact		Flip-swap		Sifting	
6	40195	191	13941.11	441	5.65	368	24.26
8	81864	1274	> 4h	2409	51.30	2251	127.55
10	19723	1707	> 4h	2472	56.76	2360	81.25

5 Experiments

We have implemented all presented algorithms using *CirKit*[3] in the commands '*npn*' and '*satnpn*'. All experiments are carried out on an Intel Xeon E5 CPU with 2.60 GHz and 128 GB main memory running Linux 3.13. First, Sect. 5.1 shows results of an experiment that compares different heuristics against the exact NPN classification algorithm. However, this can only be done for small functions, for which the exact algorithm finishes in reasonable time. On the other hand, Sect. 5.2 shows results of an experiment that evaluates scalability by applying the heuristics to larger functions beyond the applicablity of truth table based implementations.

5.1 Quality evaluation

We applied both the truth table based implementation and the AIG based implementation of both heuristics to small Boolean functions that were harvested using structural cut enumeration in all instances of the MCNC, ISCAS, and ITC benchmark sets as suggested by Huang et al. [10] Based on the computed representative, all functions are partitioned into a smaller set of classes. Table 1 shows the number of classes generated by an exact algorithm for NPN classification and by each of the two heuristic algorithms presented in Sect. 3. Although the heuristic NPN classification algorithms can be applied to truth tables up to 16 variables, exact classification does not scale since all $n! \cdot 2^{n+1}$ permutations and phases have to be evaluated in the worst case. Runtimes are obtained from the truth table based methods. For example, the exact and heuristic classifications were applied to 40195 distinct functions of six variables. When partitioning this set based on the NPN representatives, 191 classes are obtained using exact NPN classification. The flip-swap heuristic cannot determine all representatives correctly and partitions the 40195 functions into 441 classes. The sifting heuristics shows better results and returns 368 classes.

The best quality is reached with exact classification, but it cannot be applied efficiently to larger functions due to the exponential search space. Nevertheless, the heuristic classification can have a huge contribution in some logic synthesis algorithms. For example, some optimizations which require a large computational effort can often be limited to only representatives of each NPN class. In the

[3] github.com/msoeken/cirkit

Table 2. Benchmark properties.

Benchmark	Inputs	Outputs	Max. inputs
c432	36	7	36
c499	41	32	41
c880	60	26	45
c1355	41	32	41
c1908	33	25	33
c2670	233	140	119
c3540	50	22	50
c5315	178	123	67
c7552	207	108	194

Table 3. Runtime (in seconds) and number of LEXSAT calls (in millions) of the heuristics for the benchmark *c7552*. In bold are the number of SAT calls (in millions).

Heuristic	LEXSAT	Single miter				Shared miter			
	calls	Tseytin		EMS		Tseytin		EMS	
Flip-swap	0.8M	4416.10	**57M**	5348.17	**63M**	62357.90	**22M**	29184.40	**23M**
Sifting	1.8M	9450.02	**123M**	12018.30	**135M**	76582.10	**49M**	44896.10	**47M**

case of 6-variable functions, when using the sifting heuristic, it means that such computation only needs to be performed for 368 functions, instead of 40195.

We observed that both the truth table based and the AIG based implementation generate the same results in terms of quality. However, the AIG based implementation showed significantly worse performance when applied to small functions. This is due to the extra overhead spent on encoding and SAT solving which is only amortized for larger functions for which the truth table based implementation is not scalable. For large functions, the exponential size representation of truth tables becomes the bottleneck.

5.2 Scalability evaluation

In order to demonstrate scalability of the AIG based implementation of the heuristic NPN classification algorithms, as well as to evaluate the different implementation options, we have applied both heuristics to the combinational instances from the ISCAS benchmark suite. Since these benchmarks realize multiple output Boolean functions we ran the algorithm on each output cone separately. The reported results are cumulative for all outputs. Table 2 shows number of inputs and outputs of the used benchmarks, as well as the maximum number of inputs in any of the output cones. Since the runtimes for *c7552* are comparably high, we report them separately in Table 3 and allow a better scaling of the other benchmarks in the plots. As the main objective is runtime in this experiment, there is no direct comparison of both heuristics. A qualitative comparison of both approaches has already been provided in the previous section.

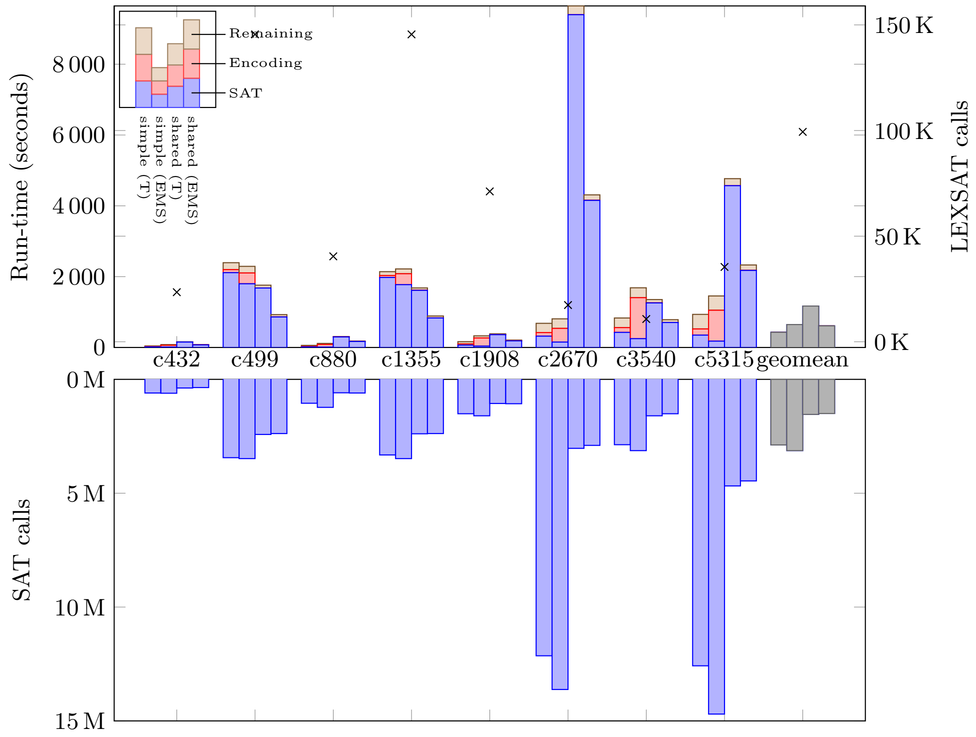

Fig. 2. Experimental results for flip-swap heuristic. Bars show the runtime (top) and the number of SAT calls (bottom). Cross marks show the number of LEXSAT calls.

The results of the experiments are presented in the plots in Fig. 2 for the flip-swap heuristic and in Fig. 3 for the sifting heuristic. Both plots are organized in the same way. We ran each benchmark in four configurations: simple miter with Tseytin encoding, simple miter with EMS encoding, shared miter with Tseytin encoding, and shared miter with EMS encoding. For each configuration, we plot the runtime in seconds in the upper axis and the number of total SAT calls in the lower axis. The runtime is separated into three parts: the lowest (blue) part shows the runtime spent on SAT solving, the middle (red) part shows the runtime spent on encoding, and the upper (brown) part shows the remaining runtime. Finally, cross marks show the number of LEXSAT calls in the upper axis. Note that the number of LEXSAT call is identical in each configuration. The last entry in the plots gives the geometric mean of the overall runtimes and SAT calls for each configuration.

The simple miter approach is often faster compared to the shared miter approach, especially for larger benchmarks. It can be seen that percentage of runtime spent on SAT solving is much smaller for the simple miter approach, since more time is spent on generating and encoding the miter. In the simple miter approaches the time spent on encoding can matter. Although the EMS

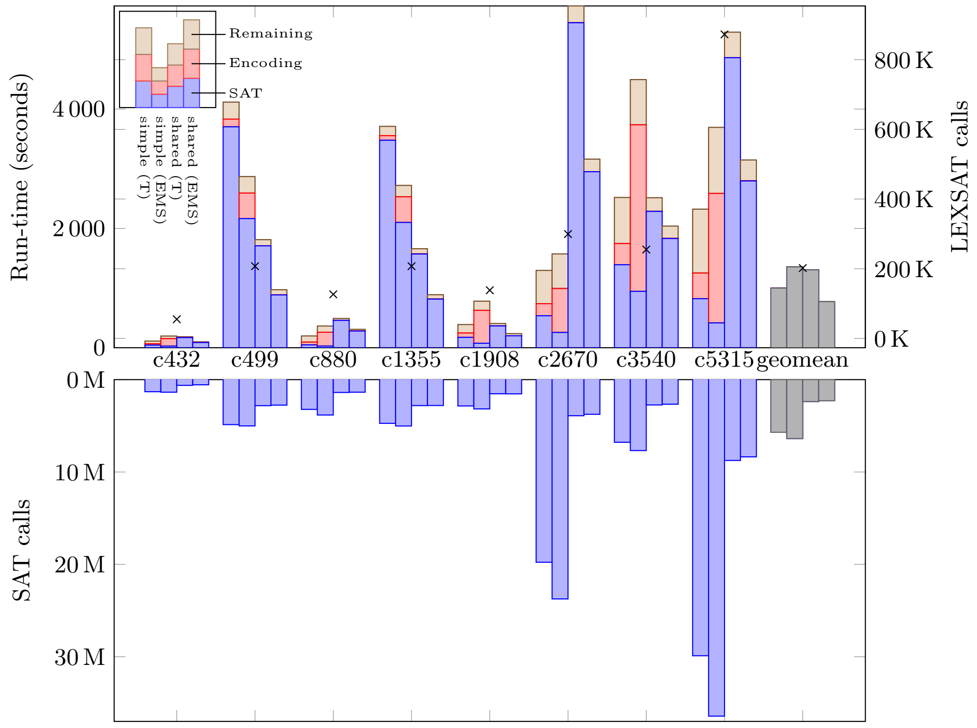

Fig. 3. Experimental results for sifting heuristic. Bars show the runtime (top) and the number of SAT calls (bottom), while cross marks show the number of LEXSAT calls.

encoding can reduce the runtime on SAT solving significantly (see, e.g., *c2670*, *c3540*, and *c5315*) the encoding time becomes the new bottleneck and eventually results in an overall larger runtime. This effect is not evident in the shared miter approach where a very small percentage of time is spent on encoding. The EMS method, due to the improved encoding, and the resulting improvement in SAT solving, reduces the overall runtime by about half overall. Note also that the overall number of SAT calls is about two times larger in the simple miter approach compared to the shared miter approach.

Challenges to performance. Symmetric or functionally independent variables can cause a problem for the algorithm, since they do not change the function after permutation and negation, respectively. For two symmetric variables x_i and x_j we have $\mathit{apply}(f, \pi, \varphi) = \mathit{apply}(f, \pi \circ (i, j), \varphi)$ and for a functionally independent variable x_i we have $\mathit{apply}(f, \pi, \varphi) = \mathit{apply}(f, \pi, \varphi \oplus 2^i)$. Consequently, the LEXSAT calls for the comparisons in the heuristic yield UNSAT and correspond to equivalence checking of two equivalent circuits. In practice, the situation may be slightly better due to several structural similarities of both circuits. In the experimental evaluation, this problem became evident for the multiplier benchmark *c6288* making a heuristic NPN classification based on SAT infeasible.

6 Possible improvements

In this section, we discuss several improvements that are not considered in the current implementation, but could be help reduce runtime more.

Symmetry and functional dependency. One can circumvent the problem with symmetric and functional independent variables by setting a time limit to the execution of a LEXSAT call. If the timeout is reached one can try random simulation as a last resort and, if this also fails, then one can proceed while not updating the current permutation π and phase φ. This shows positive effects on the overall runtime but can degrade the quality by increasing the number of distict equivalence classes.

Partial EMS encoding. The heuristics update the current permutation and phase by swapping two variables or negating one variable. Consequently, in most of the cases only small parts of the updated circuit change. The mapping of the unchanged part of the circuit and the resulting CNF stay the same and do not need to be recomputed. Making the EMS encoding aware of changes in the circuit reduces runtime. Since in case of EMS, the encoding often is the predominant part of the overall runtime, a significant speedup can be expected.

Avoid miter construction. Note that $\max f < \max g$ implies $f < g$. The other direction is not true which can readily be seen from $f = 1010$ and $g = 1100$. This check may be faster than first constructing the miter and calling LEXSAT on it. We have tried to integrate this check as a preprocessing step before the miter construction in the simple miter approach. However, as in most cases we had $\max f = \max g$, this resulted in a higher overall runtime. The check can be better integrated using a thread, that is started at the same time of the miter construction, and is terminated when a conclusive answer is found faster.

Using simulation to skip some SAT calls. For simplicity, consider the computation of $\max f$ for one function, as in the above subsection, rather than for the miter of two functions, as elsewhere in the paper. Assume that we found $\max f$, which is an assignment of n input variables x such that $f(x) = 1$, and $f(y) = 0$ for all assignments $y > x$. Now take assignment x and generate n assignments, which are distance-1 from x, by flipping the value of one input at a time. Perform bit-parallel simulation of f using these distance-1 assignments and observe the output of f. If $f(d) = 1$ for some distance-1 assignment d, we know that flipping the corresponding input cannot reduce $\max f$, and so we can skip the SAT call. On the other hand, if $f(d) = 0$, flipping the corresponding input may lead to a smaller $\max f$. The reduction in $\max f$ is possible if flipping this input does not create a new 1 for a lexicographically larger assignment, which has a 0 before (follows from the fact that the original assignment is $\max f$). As a result, in the case when $f(d) = 0$, we need a LEXSAT call to check whether flipping this input leads to an improvement in $\max f$. However, when $f(d) = 1$ the call can be skipped to reduce the runtime.

Native LEXSAT solver. A considerably faster LEXSAT algorithm can be obtained by directly modifying the SAT solver (see Ex. 7.2.2.2-275 in [13]). To this end, instead of using the assumption interface of MiniSAT, as we did in this paper, we can modify decision heuristics of the SAT solver to always perform decisions on the input variables in the order, which is imposed by the user of the LEXSAT algorithm. Decisions on other variables can be performed in any order.

Using binary search in LEXSAT. Another possibility to improve the runtime of LEXSAT implementation is to use an approach based on binary search. Instead of trying to append to the array of assumptions one literal at a time, we can begin by putting one half of all literals, then one quarter, and so on. The binary search approach can be taken one step further: We can profile and see how often, on average, the first step of binary search leads to a SAT or UNSAT call. Based on the result, we could try to make the first step to be, say, 25% or 75% of the total number of assumptions, instead of 50% (as in the naïve binary search). This way, we could have a better "success rate" of searching, which could lead to a faster LEXSAT implementation.

7 Conclusions

In this paper, we have shown how heuristic NPN classification algorithms can be implemented based on AIGs and SAT instead of truth tables. The new implementation can be applied to larger functions. The key aspect of the proposed approach is finding the lexicographically smallest assignment of a SAT instance using the LEXSAT algorithm. An experimental evaluation shows that using the AIG based implementation, the heuristic NPN classification algorithms can be applied to functions with hundreds of inputs. Our current implementation is preliminary and there are several possibilities to reduce the overall runtime, some of them have been outlined in Sect. 6.

Acknowledgments. This research was supported by H2020-ERC-2014-ADG 669354 CyberCare, by the German Academic Exchange Service (DAAD) in the PPP 57134066, and partly by the NSF/NSA grant "Enhanced equivalence checking in cryptoanalytic applications" at University of California, Berkeley.

References

1. Benini, L., Micheli, G.D.: A survey of Boolean matching techniques for library binding. ACM Trans. Design Autom. Electr. Syst. 2(3), 193–226 (1997)
2. Brand, D.: Verification of large synthesized designs. In: Proceedings of the International Conference on Computer Aided Design. pp. 534–37 (1993)
3. Brayton, R.K., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proceedings of the International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 6174, pp. 24–40. Springer (2010)
4. Chatterjee, S., Mishchenko, A., Brayton, R.K., Wang, X., Kam, T.: Reducing structural bias in technology mapping. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25(12), 2894–903 (2006)
5. Debnath, D., Sasao, T.: Efficient computation of canonical form for Boolean matching in large libraries. In: Proceedings of the Asia and South Pacific Design Automation Conference. pp. 591–96 (2004)
6. Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing. pp. 272–86 (2007)
7. Goto, E., Takahasi, H.: Some theorems useful in threshold logic for enumerating Boolean functions. In: International Federation for Information Processing Congress. pp. 747–52 (1962)
8. Harrison, M.A.: Introduction to Switching and Automata Theory. McGraw-Hill, New York (1965)
9. Hellerman, L.: A catalog of three-variable Or-inverter and And-inverter logical circuits. IEEE Transactions on Electronic Computers 12, 198–223 (1963)
10. Huang, Z., Wang, L., Nasikovskiy, Y., Mishchenko, A.: Fast Boolean matching based on NPN classification. In: Proceedings of the 2013 International Conference on Field Programmable Technology. pp. 310–13 (2013)
11. Katebi, H., Markov, I.L.: Large-scale Boolean matching. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition. pp. 771–76 (2010)
12. Knuth, D.E.: The Art of Computer Programming, Volume 4A. Addison-Wesley, Reading, Massachusetts (2011)
13. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability. Addison-Wesley, Reading, Massachusetts (2015)
14. Krentel, M.W.: The complexity of optimization problems. Journal of Computer and System Sciences 36(3), 490–509 (1988)
15. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21(12), 1377–94 (2002)
16. Mailhot, F., Micheli, G.D.: Algorithms for technology mapping based on binary decision diagrams and on Boolean operations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12(5), 599–620 (1993)
17. Mishchenko, A.: Enumeration of irredundant circuit structures. In: Proceedings of the 23rd International Workshop on Logic and Synthesis (2014)
18. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In: Proceedings of the 43rd Design Automation Conference. pp. 532–36 (2006)
19. Muroga, S.: Logic design and switching theory. John Wiley & Sons Inc., New York (1979)

20. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of the International Conference on Computer Aided Design. pp. 42–47 (1993)
21. Soeken, M., Amarù, L.G., Gaillardon, P., De Micheli, G.: Optimizing majority-inverter graphs with functional hashing. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition. pp. 1030–1035 (2016)
22. Tseytin, G.S.: On the complexity of derivation in propositional calculus. In: Slisenko, A.P. (ed.) Studies in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics, pp. 115–25. Springer (1970)
23. Warren, Jr., H.S.: Hacker's Delight. Addison-Wesley, Reading, Massachusetts, 2 edn. (2012)