An Anytime Symmetry Detection Algorithm for ROBDDs

Neil Kettle
University of Kent, UK
Email: njk4@kent.ac.uk

Andy King
University of Kent, UK
Email: a.m.king@kent.ac.uk

Abstract— Detecting symmetries is crucial to logic synthesis, technology mapping, detecting function equivalence under unknown input correspondence, and ROBDD minimization. State-of-the-art is represented by Mishchenko's algorithm. In this paper we present an efficient anytime algorithm for detecting symmetries in Boolean functions represented as ROBDDs, that output pairs of symmetric variables until a prescribed time bound is exceeded. The algorithm is complete in that given sufficient time it is guaranteed to find all symmetric pairs. The complexity of this algorithm is in $O(n^4+n|G|+|G|^3)$ where n is the number of variables and $|G|$ the number of nodes in the ROBDD, and it is thus competitive with Mishchenko's $O(|G|^3)$ algorithm in the worst-case since $n \ll |G|$. However, our algorithm performs significantly better because the anytime approach only requires lightweight data structure support and it offers unique opportunities for optimization.

I. Introduction

Symmetry detection has been important since the days of Shannon [1] who observed that symmetric functions have particularly efficient switch network implementations. Symmetry detection is no less important these days and knowledge of symmetric variables has many applications in logic synthesis [2, 3], technology mapping [4, 5], ROBDD minimization [6, 7] and detecting equivalence of Boolean functions for which input correspondence is unknown [8, 9].

The challenge in symmetry detection is to find efficient algorithms for detecting all symmetric variables pairs (x_i, x_j) of a given Boolean function $f(x_1 \ldots x_n)$, that is, find all pairs (x_i, x_j) such that $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots)$. The intuition being that f remains unchanged under the switching of the variables x_i and x_j. This symmetry is formally known as the first-order classical symmetry, or the non-skew non-equivalence symmetry [10]. It can be shown from Boole's expansion theorem [11] this is equivalent to checking equality of the co-factor pair $f|_{x_i \leftarrow 0, x_j \leftarrow 1} = f|_{x_i \leftarrow 1, x_j \leftarrow 0}$. This formulation shows that it is possible to find the set of all symmetric pairs by calling the co-factoring operation no more than $n^2 - n$ times, where n is the number of variables. Early work on detecting symmetric variables in Boolean functions has focussed on the computation of these co-factor pairs and symmetry detected by checking their equivalence [12]. The use of ROBDDs to represent Boolean functions enables co-factor equivalence to be checked in constant time, however, repeated co-factoring involves the creation and deletion of many intermediate ROBDD nodes and for very large ROBDDs this overhead can be prohibitive. This method is often referred to as the naïve method [12]. Möller, Mohnke and Weber [12] thus advocate the use of two preprocessing algorithms — two sieves — that detect pairs of asymmetric variables. These linear-time sieves significantly reduce the number of co-factor pairs that need to be computed. In general, however, the method still requires naïve co-factor computation, that is, calls to the standard co-factoring algorithm the complexity of which is in $O(|G| \lg |G|)$ [13]. Methods that rely on asymmetry sieves, such as those proposed in [7, 12], are said to be based upon the so-called *negative-thinking* paradigm [14]. That is, they obtain the symmetric variable pairs from the set of all variables pairs by systematically removing all asymmetric variable pairs.

Because of the cost of repeated co-factoring, many symmetry detection methods endeavor to avoid naïve co-factor computation. Möller *et al.* [12] and Panda *el al.* [6] detect all symmetries between variables adjacent in the variable order with an algorithm in $O(|G|)$. Rudell's dynamic variable reordering algorithm [15] has also been used to detect symmetries, although the aim is not symmetry detection *per se*, but ROBDD minimization. Rudell's algorithm considers each variable in turn moving it up and down in the variable ordering (subject to complexity limits) so as to minimize the ROBDD. Panda *et al.* [6] modify Rudell's algorithm to detect symmetries between variables that become adjacent when one of the variables is repositioned in the ROBDD variable ordering. Symmetric variables are then grouped, and any subsequent reordering that is applied is required to preserve a contiguous variable ordering within each group. This approach to symmetry detection does not require naïve co-factor computation, but there is no guarantee that all symmetries will be found. State-of-the-art is represented by Mishchenko's algorithm [14] that detects all symmetric variable pairs in a ROBDD in $O(|G|^3)$. (Note that this algorithm is parameterized by the underlying set representation that is used to store the variable pairs, and therefore this complexity result does not consider the complexity of the set operations themselves. Most conservatively, assuming all set operations are linear, the overall running time is at least $O(n^2|G|^3)$ since each set contains potentially $O(n^2)$ elements). Algorithms such as those of Mishchenko [14] and Panda *et al*. [6] are based on the so-called *positive-thinking* paradigm [14]. That is they compute variable pairs that are symmetric, and in the case of Mishchenko's algorithm, because of its completeness, those pairs not found to be symmetric are then known to be asymmetric.

The problem with existing symmetry detection methods is that they are either monolithic, inefficient, or incomplete. A monolithic algorithm has to be run to completion before it can return any answer; the value of such an algorithm is compromised if the running time is prohibitive. Mishchenko's [14] algorithm falls into this class. Practically all engineering tasks (and logic synthesis is no exception) require an acceptable answer to be found in a reasonable amount of time rather than the optimal answer in an exorbitant amount of time. This is relevant in the context of symmetry detection because the running time of the state-of-the-art algorithm [14] can exceed 12 hours on some ROBDDs of less than a million nodes (actually this was benchmark `simp12`). This motivates the need for a so-called anytime algorithm that will incrementally detect pairs of symmetric variables until some given time bound is exceeded. Symmetry detection algorithms [7, 12] based on naïve co-factor computation can be considered to be incremental but, alas, this approach is inefficient. The algorithm of Panda *et al*. [6] is an interesting example of an incremental algorithm that does not require co-factor computation but, unfortunately, the algorithm is incomplete for the purposes of symmetry detection.

In this paper we present a novel anytime algorithm for symmetry detection based on the negative-thinking paradigm, whose efficiency compares very favorably against that of Mishchenko in the case when all the pairs require to be enumerated. The algorithm demonstrates that, with careful construction, it is possible to detect symmetries incrementally without compromising efficiency. Our anytime algorithm is inspired by that of Mishchenko, but the correctness of our algorithm is surprisingly subtle in that it depends on paths not passing through given nodes in the ROBDD. For pedagogical purposes, two versions of the algorithm are presented: a simple version that contains a minimal number of components to ensure correctness; and a refined version that demonstrates how an incremental algorithm has computational advantages over a comparable monolithic algorithm. These two algorithms are respectively presented in Sections III and IV. An intriguing aspect of the anytime approach is that it permits transitivity to be fully exploited. It is well-known that if $(x_i, x_j), (x_j, x_k)$ are symmetric then so is (x_i, x_k) [16,17], but this observation has had scant consideration in the symmetry detection literature. Möller *et al.* [12, p 681] state that "we also use the fact that if $\{x_i, x_j\}$ and $\{x_j, x_k\}$ are pairs of symmetric variables, then $\{x_i, x_k\}$ is a pair of symmetric variables as well", seemingly missing the fact that if (x_i, x_j) are symmetric and (x_i, x_k) are asymmetric then (x_j, x_k) are asymmetric. Due to the way our anytime algorithm decomposes symmetry detection into a series of passes, one for each variable, we are free to apply asymmetry/symmetry propagation between each of these passes to reduce the expected cost of each pass. This is discussed in Section IV-C. Sections IV-A and IV-B show how the algorithm can be accelerated using more well-known techniques that relate to adjacent symmetries [12] and positive satisfy counts [8]. Extensive experimental results that are given in Section V demonstrate the value of these refinements, compare the algorithm against of that Mishchenko and demonstrate the anytime nature of the algorithm. The remainder of this paper is organized as follows: Section II presents definitions used within the paper and Section VI presents the concluding discussion. For clarity, we summarize our contributions as follows:

- The paper presents a novel incremental, anytime algorithm for symmetry detection based on the negative-thinking paradigm.
- In theory, the algorithm is in $O(n^4+n|G|+|G|^3)$ where $n \ll |G|$ (even considering the complexity of all set operations) which compares favorably against state-of-the-art [14].
- The paper shows that an anytime algorithm can put low computational demands on the underlying data-structures that represent pairs of symmetric variables. Thus anytime generality does not have to sacrifice efficiency, indeed the converse is true.
- The paper explains how an incremental anytime approach offers special opportunities for optimization, in that classical assymetry/symmetry sieves can precede the algorithm and assymetry/symmetry propagation techniques can be inserted into the main loop of the algorithm.
- The paper also reports a hitherto overlooked subtlety of symmetry detection: it seems that at least $O(n|G|)$ preprocessing steps must be performed before incremental symmetry detection may commence. Rather surprisingly, the correctness of our algorithm critically depends upon an $O(n|G|)$ asymmetry sieve [12], that relates to paths that can arise within an ROBDD in the presence of symmetries. (As a consequence, we conjecture that there is no way to construct an incremental, complete symmetry detection algorithm without first applying preprocessing).

II. Preliminaries

In this paper we consider completely specified Boolean functions $f : \{0,1\}^n \to \{0,1\}$ that are conventionally written as Boolean formulae defined over a variable set $X = \{x_1, \ldots, x_n\}$. The satisfy-count of an n-ary Boolean function f is defined as $\|f\| = |\{(b_1, \ldots, b_n) \mid f(b_1, \ldots, b_n) = 1\}|$ [13]. The (Shannon) co-factor of a function f w.r.t a variable x_i and a Boolean constant b is defined by $f|_{x_i \leftarrow b} = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$. Multiple variable co-factors can be defined inductively as $f_0 = f$, $f_i = f_{i-1}|_{x_i \leftarrow b_i}$ and $f|_{x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n} = f_n$. A function f over X is symmetric in a pair of variables (x_i, x_j) iff $f|_{x_i \leftarrow 0, x_j \leftarrow 1} = f|_{x_i \leftarrow 1, x_j \leftarrow 0}$, otherwise it is asymmetric in (x_i, x_j).

ROBDDs are obtained by inducing a total-order on X. A BDD is a rooted directed acyclic graph where each internal node is labeled with a Boolean variable. Each internal node has one successor node connected via an edge labeled 0, and another successor connected via an edge labeled 1. Each external (leaf) node is either 0 or 1. The Boolean function represented by a BDD can be evaluated for a given variable assignment by traversing the graph from the root, taking the 1 edge at a node when the variable is assigned to 1 and the 0 edge when the variable is assigned to 0. The external node reached in this traversal indicates the value of the Boolean function for the assignment. An OBDD is a BDD with the restriction that the label of a node is always less than the label of any internal node reachable via its successors. An ROBDD is an OBDD with the additional constraint that the successors of any internal node do not represent the same Boolean function. Note that any internal node of an ROBDD is itself the root of an ROBDD. An ROBDD f is symmetric in a pair of variables (x_i, x_j) iff the Boolean function it represents is symmetric in (x_i, x_j). Finally, let $|G|$ denote the number of internal nodes in a ROBDD G.

III. Anytime Symmetry Detection Algorithm

In this section we propose a novel, anytime approach to symmetry detection. The algorithm presented in Algorithm 1 contains the minimum number of components required so as to ensure correctness. The algorithm takes as input an ROBDD f and returns the set S of symmetric variable pairs. The algorithm is composed of two distinct procedures. `ComputeAsymmetry`(f) performs two depth-first search (dfs) traversals over the ROBDD f, to detect pairs of variables that are asymmetric (in the particular sense that is described in Section III-A). `RemoveAsymmetry`(f, i, C) filters a set of variables C whose symmetry relationship with variable x_i is unknown to return the set $C' \subseteq C$ of variables that are symmetric with x_i (this procedure is detailed in Section III-B).

Algorithm 1 ComputeSymmetricPairs(f)

$A \leftarrow$ `ComputeAsymmetry`(f)
$S \leftarrow \emptyset$
for $i = 1$ **to** $n-1$ **do**
  $C \leftarrow \{ j \mid (i,j) \notin (S \cup A) \wedge i < j\}$
  $D \leftarrow$ `RemoveAsymmetry`(f, i, C)
  $S \leftarrow S \cup \{(i,k),(k,i) \mid k \in D\}$
  $A \leftarrow A \cup \{(i,l),\ (l,i) \mid l \in C \setminus D\}$
return S

The call to `ComputeAsymmetry` initializes the set of asymmetric variable pairs A; S is initially empty. The remainder of the algorithm considers each of the n variables in turn. Firstly, a set C is constructed that contains all variables whose symmetry relation with x_i has not yet been ascertained. Secondly, the set of symmetric variables D returned from `RemoveAsymmetry` is used to extend S and A. Observe that the sets S and A can be augmented in $O(n)$ time when C and D are represented as arrays. Furthermore, observe that C can be constructed in $O(n)$ time when the sets of pairs S and A are represented as adjacency matrices. Finally, observe that actually only $n-1$ iterations of the loop are required because of the structure of C. Further details of these two procedures are given in Sections III-A and III-B.

A. Computing Asymmetries

The algorithm that initializes A is constructed from lemmas that detail how symmetric variables place structural constraints on ROBDDs [12]. For completeness, we state these lemmas below:

Lemma 1. *If an ROBDD f over a set of variables $X = \{x_1, \ldots, x_n\}$ is symmetric in the pair (x_i, x_j) and $i < j$, then every ROBDD rooted at a node labeled x_i must contain a node labeled x_j.*

Lemma 2. *If an ROBDD f over a set of variables $X = \{x_1, \ldots, x_n\}$ is symmetric in the pair (x_i, x_j) and $i < j$, then every path from the root of f to a node labeled x_j must visit a node labeled x_i.*

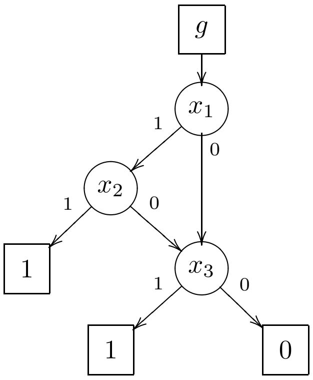

Fig. 1. The ROBDD g for the propositional formula $(x_1 \wedge x_2) \vee x_3$

Lemma 1 and Lemma 2 provide two conditions under which asymmetry can be observed. For any given node labeled x_i we can compute the set of all variables x_j that appear in a ROBDD that is rooted at that node, and any variable not appearing in this set is necessarily asymmetric with x_i. Furthermore, for any given node labeled x_j, we can compute the set of all variables x_i that appear on *all* paths from the root of the ROBDD to the node, and any variable not appearing in this set is asymmetric with x_j. The asymmetry conditions of Lemma 1 and Lemma 2 can be checked in two dfs traversals of the ROBDD, each traversal taking $O(n|G|)$ time.

Each iteration of the loop in Algorithm 1 considers a variable x_i and forms the set C from those variables whose symmetry relationship with variable x_i is not yet known. The validity of this decomposition into multiple passes, is justified by the proposition which itself is a consequence of the following lemma [12]:

Lemma 3. *A Boolean function f over a set of variables $X = \{x_1, \ldots, x_n\}$ is symmetric in the pair (x_i, x_j) iff both co-factors $f|_{x_k \leftarrow 0}$ and $f|_{x_k \leftarrow 1}$ are symmetric in the pair (x_i, x_j).*

Proposition 1. *If an ROBDD f over a set of variables $X = \{x_1, \ldots, x_n\}$ is symmetric in the pair (x_i, x_j) and $i < j$ iff*

- *every ROBDD rooted at a node labeled x_i is symmetric in (x_i, x_j) and,*
- *every path from the root to a node labeled x_j passes through a node labeled x_i.*

Proof. The proposition follows by applying the lemma inductively on the variables $x_{i-1}, \ldots, x_1$, though for brevity we consider only the first inductive step. Consider an ROBDD g whose root node is labeled with x_{i-1}. There are four cases to consider. First, the roots of both co-factors $g|_{x_{i-1} \leftarrow 0}$ and $g|_{x_{i-1} \leftarrow 1}$ are labeled x_i. By Lemma 3, g is symmetric in (x_i, x_j) iff $g|_{x_{i-1} \leftarrow 0}$ and $g|_{x_{i-1} \leftarrow 1}$ are symmetric in (x_i, x_j). Observe that every path from the root of g to x_j passes through a node labeled x_i. Second, the root of $g|_{x_{i-1} \leftarrow 0}$ is labeled with x_i whereas $g|_{x_{i-1} \leftarrow 1}$ is not. Again, g is symmetric in (x_i, x_j) iff $g|_{x_{i-1} \leftarrow 0}$ and $g|_{x_{i-1} \leftarrow 1}$ are symmetric in (x_i, x_j). Observe $g|_{x_{i-1} \leftarrow 1}$ is symmetric in (x_i, x_j) iff $g|_{x_{i-1} \leftarrow 1}$ contains no node labeled x_j, or equivalently, every path from the root of g to x_j passes through a node labeled x_i. The third and fourth cases are respectively analogous and similar to the second. $\square$

The proposition allows exhaustive checking to be decomposed into a series of passes; one pass for each variable x_i. The crucial point is that when the loop is entered, we have already removed all pairs of variables (x_i, x_j) such that there exists a path from the root to a node labeled x_j which does not pass through a node labeled x_i. Hence, for correctness, the body of the loop in Algorithm 2, must only check the first condition of the proposition. The counterexample given in Figure 1 illustrates the necessity of the second condition in the proposition, or put another way, it shows that correctness is compromised if the preprocessing is omitted from the algorithm. Observe that in Figure 1 that the variable pair (x_2, x_3) is symmetric in the ROBDD rooted at x_2, however (x_2, x_3) are asymmetric in the ROBDD g since there exists a path from the root of g (x_1) to the node x_3 that does not visit a node labeled x_2. In fact, disabling the preprocessing gives the following asymmetry and symmetry sets A_i and S_i after i iterations of the loop: $A_0 = S_0 = \emptyset$, $A_1 = \{(x_1, x_3), (x_3, x_1)\}$,$S_1 = \{(x_1, x_2), (x_2, x_1)\}$, $A_2 = A_1, S_2 = S_1 \cup \{(x_2, x_3), (x_3, x_2)\}$. Observe the erroneous pair (x_2, x_3) contained within S_2.

B. Removing Asymmetries

After the initial preprocessing, incremental symmetry detection can commence. The procedure given below takes as input an ROBDD f, a variable index i, and a set C of variable indices corresponding to those variables whose symmetry relation with variable i is unknown.

Algorithm 2 RemoveAsymmetry(f, i, C)

> **if** $C = \emptyset$ **then**
>   **return** $\emptyset$
> $j \leftarrow$ `index`(f)
> **if** $j > i \vee f = \mathbf{true} \vee f = \mathbf{false}$ **then**
>   **return** C
> **if** $j = i$ **then**
>   **return** `RemoveAsymmetryVar`$(f|_0, f|_1, C)$
> **else**
> $C \leftarrow$ `RemoveAsymmetry`$(f|_0, i, C)$
>   **return** `RemoveAsymmetry`$(f|_1, i, C)$

The function `index`(f) returns the index of the root node of f, that is, i if the root is labeled x_i. The test $j > i$ implements a form of early termination: if the test is satisfied then the ROBDD f can contain no node labeled x_i. The external nodes $\mathbf{true}$ and $\mathbf{false}$ also trigger early termination. At the heart of `RemoveAsymmetry` is a call to `RemoveAsymmetryVar` which encapsulates the logic to co-factor f_0 and f_1 so as to perform the symmetry check. The pseudo-code for this procedure is given in Algorithm 3. Whenever the call `RemoveAsymmetryVar` is reached, it examines the co-factors of f to remove variables from C that are asymmetric w.r.t x_i. First, consider the case when both root nodes of f_0 and f_1 are labeled with the same variable x_j. In this case we compute $f_0|_{x_j \leftarrow 1}$ and $f_1|_{x_j \leftarrow 0}$ and check for equivalence. Second, when f_0 is labeled with x_j and f_1 is labeled with x_k where $j < k$, the check reduces to $f_0|_{x_j \leftarrow 1} = f_1$. Third, the $k < j$ case is analogous to the second. The recursive calls follow the co-factor check because it is necessary to check symmetry across all variable assignments. Note that both `RemoveAsymmetry` and `RemoveAsymmetryVar` terminate as soon as $C = \emptyset$.

When C is implemented as an array, the complexity of a single call to `RemoveAsymmetryVar` is $O(|G|^2)$. This follows since co-factor comparison and $C \setminus \{l\}$ are in $O(1)$, as is the test $C = \emptyset$ when C is augmented with a counter to record $|C|$. Overall, `RemoveAsymmetryVar` can only be invoked a total of $|G|$ times from within Algorithm 1, thus `RemoveAsymmetryVar` contributes $O(|G|^3)$ to the overall running time. The $n - 1$ calls to `RemoveAsymmetry` cumulatively cost $O(n|G|)$.

IV. Optimized Anytime Symmetry Detection Algorithm

In this section we propose a series of optimizations for Algorithm 1. This refined algorithm retains the incremental nature of the original algorithm, and in fact shows how this can be exploited by several optimizations. These optimizations seek to reduce the size of the set C, and hence the running time of the call

Algorithm 3 RemoveAsymmetryVar(f_0, f_1, C)

```
if C = ∅ then
  return ∅
if (f_0 = true ∨ f_0 = false) ∧ (f_1 = true ∨ f_1 = false) then
  return C
j ← index(f_0)
k ← index(f_1)
if j = k then
  (l, f_00, f_01, f_10, f_11) ← (j, f_0|_{j←0}, f_0|_{j←1}, f_1|_{k←0}, f_1|_{k←1})
else if j < k then
  (l, f_00, f_01, f_10, f_11) ← (j, f_0|_{j←0}, f_0|_{j←1}, f_1, f_1)
else
  (l, f_00, f_01, f_10, f_11) ← (k, f_0, f_0, f_1|_{k←0}, f_1|_{k←1})
if f_01 ≠ f_10 then
  C ← C \ {l}
C ← RemoveAsymmetryVar(f_00, f_10, C)
return RemoveAsymmetryVar(f_01, f_11, C)
```

RemoveAsymmetry(f, i, C), by enriching the sets A and S on-the-fly before, and between, iterations of the main loop. The symmetry sieve algorithms presented by [7, 12] give a way to refine the sets A and S before the loop is entered. When the loop is entered, it is possible to take advantage of the transitivity of the symmetry relation to add further pairs to A and S. The optimized symmetry detection algorithm presented in Algorithm 4 takes as input an ROBDD f and returns the set S of symmetric variable pairs. The new algorithm includes three additional procedures, namely, ComputeSatisfyCounts(f), ComputeAdjSymmetry(f) and SymmetryClosure(A, S) which are detailed in Sections IV-A and IV-B, IV-C respectively.

Algorithm 4 OptimizedSymmetricPairs(f)

```
A ← ComputeAsymmetry(f)
M ← ComputeSatisfyCounts(f)
for i = 1 to n do
  for j = i + 1 to n do
    if M(i) ≠ M(j) then
      A ← A ∪ {(i, j), (j, i)}
S ← ComputeAdjSymmetry(f)
for i = 1 to n − 2 do
  (A, S) ← SymmetryClosure(A, S)
  C ← { j | (i, j) ∉ (S ∪ A) ∧ i + 1 < j}
  D ← RemoveAsymmetry(f, i, C)
  S ← S ∪ {(i, k), (k, i) | k ∈ D}
  A ← A ∪ {(i, l), (l, i) | l ∈ C \ D}
return S
```

ComputeSatisfyCounts(f) returns a mapping M from variable indices to a natural number that can be used to distinguish pairs of asymmetric variables, that is, if $M(i) \neq M(j)$ then (x_i, x_j) are asymmetric. ComputeAdjSymmetry(f) returns the set of symmetric variable pairs for those pairs that are adjacent in the ROBDD ordering (which permits the number of loop iterations to be relaxed to $n-2$). Finally, SymmetryClosure(A, S) takes as input two sets A and S of variable pairs known to be asymmetric and symmetric respectively. Two new sets $A' \supseteq A$ and $S' \supseteq S$ are output that are derived by exploiting the transitivity of symmetry.

A. Positive Satisfy-Counts

A consequence of symmetry, which can also be used to detect asymmetry [8], relates to the satisfy count of one positive co-factor of a variable to the satisfy count of another:

Lemma 4. *If a Boolean function f over a set of variables $X = \{x_1, \ldots, x_n\}$ is symmetric in the pair (x_i, x_j), then $\|f|_{x_i \leftarrow 1}\| = \|f|_{x_j \leftarrow 1}\|$.*

Computing the satisfy counts of all co-factors can be realized using a single dfs traversal of the ROBDD in $O(n|G|)$ time [8]. Finding the resultant asymmetries requires n^2 comparisons in Algorithm 4, and thus the overall complexity of this phase is $O(n^2 + n|G|)$.

B. Adjacent Symmetries

The following lemma details a special case of symmetry, which relates to variables that are adjacent in the ROBDD ordering:

Lemma 5. *If a ROBDD f over a set of variables $X = \{x_1, \ldots, x_n\}$ is symmetric in the pair (x_i, x_{i+1}) iff $g|_{x_i \leftarrow 0, x_{i+1} \leftarrow 1} = g|_{x_i \leftarrow 1, x_{i+1} \leftarrow 0}$ holds for each ROBDD g that is rooted at a node labeled x_i*

This lemma leads to an $O(|G|)$ time algorithm that can detect all symmetry and asymmetry relationships between adjacent variables [12]. (In fact the algorithm of Möller *et al.* can be improved to detect asymmetry for a pair of non-adjacent variables, that is, a pair (x_i, x_k) is asymmetric if there exists a node g labeled x_i with successor nodes labeled x_k and x_l where $i+1 < k \leq l$ and $g|_{x_i \leftarrow 0, x_k \leftarrow 1} \neq g|_{x_i \leftarrow 1, x_k \leftarrow 0}$.)

C. Symmetry Propagation

The final lemma can be obtained by recalling that a function f remains unchanged under the switching of any symmetric variables:

Lemma 6. *If a Boolean function f over a set of variables $X = \{x_1, \ldots, x_n\}$ is symmetric in the pairs (x_i, x_j) and (x_j, x_k) then f is also symmetric in the pair (x_i, x_k).*

This transitivity result provides a way of enriching the set S, that is, if $(x_i, x_j), (x_j, x_k) \in S$ then it follows that (x_i, x_k) is also a symmetric pair. Further, given $(x_i, x_j) \in S, (x_i, x_k) \in A$ then it follows that the pair (x_j, x_k) is asymmetric, that is, A can possibly be enriched too. This follows since if (x_j, x_k) is symmetric then by the lemma it follows that (x_i, x_k) is symmetric, which is a contradiction. Adding those variable pairs to A and S which can be inferred through transitivity is not dissimilar to computing the transitive closure of a binary relation. This motivates adapting the Floyd-Warshall [18, 19] all-pairs-shortest-path algorithm to this task by representing the sets of pairs A and S as an adjacency matrix of n^2 size. The pseudo-code for this algorithm is given in Algorithm 5.

Algorithm 5 SymmetryClosure(A, S)

```
for i = 1 to n do
  for j = i + 1 to n do
    for k = 1 to n do
      if (k, i) ∈ S ∧ (k, j) ∈ S then
        S ← S ∪ {(j, i), (i, j)}
      else if (k, i) ∈ A ∧ (k, j) ∈ S then
        A ← A ∪ {(j, i), (i, j)}
      else if (k, i) ∈ S ∧ (k, j) ∈ A then
        A ← A ∪ {(j, i), (i, j)}
return (A, S)
```

The complexity of Algorithm 5 is in $O(n^3)$ since membership check and single element insertion can be performed in $O(1)$ time for an adjacency matrix representation. Note that although the worst-case running time is not dependent on the number of symmetries present, larger symmetry sets induce more propagation which reduces the overall running time.

TABLE I
EXPERIMENTAL RESULTS

Circuit	# In	# Out	Σ|G|	|S|	read	naïve	[14]	§ III	A	A+B	A+B+C
pair	173	137	118066	1910	0.20	132.46	6.62	2.37	2.18	2.16	2.08
s4863	153	104	126988	547	2.63	20.60	5.30	1.41	1.08	1.01	0.82
s9234.1	247	250	4434504	3454	20.14	>7200	1407.20	183.84	158.36	145.94	141.26
s38584.1	1464	1730	150554	15629	3.70	337.59	16.70	3.12	3.04	3.01	2.80
C880	60	26	600998	262	8.29	704.54	13.90	7.75	6.84	5.63	5.20
C3540	50	22	4618194	81	21.80	>7200	132.72	71.64	68.23	66.08	65.04
simp10	105	1	722074	19	58.45	>7200	661.70	65.28	47.53	43.90	40.88
simp12	117	1	758330	23	76.23	>7200	>7200	105.67	61.94	59.87	57.59
simp14	120	1	562326	36	70.38	>7200	1114.29	75.75	38.48	36.17	30.63
hom06	104	1	1176845	20	65.22	>7200	274.90	115.66	91.70	88.31	81.50
hom08	95	1	893312	16	56.48	>7200	135.79	67.79	54.99	50.89	49.00
hom10	130	1	309221	29	29.98	>7200	1510.32	35.85	33.39	31.61	31.21
ca004	53	1	782640	2	5.40	>7200	147.97	31.35	12.33	12.33	12.10
ca008	96	1	682617	16	20.40	>7200	326.92	53.54	44.69	43.05	42.78
ca016	107	1	861209	26	60.10	>7200	305.11	72.68	59.96	50.90	50.80
urquhart2_25	48	1	722657	5	3.06	>7200	70.50	26.22	20.23	20.21	17.95
urquhart3_25	62	1	1771025	24	6.22	>7200	>7200	82.98	81.14	76.97	72.80
urquhart4_25	68	1	1736705	27	5.96	>7200	>7200	83.44	81.84	76.48	72.02
rope_0002	54	1	634914	3	3.06	>7200	192.77	22.48	18.53	18.47	18.50
rope_0004	62	1	1052214	10	4.73	>7200	487.26	41.71	39.70	37.90	37.82
rope_0006	61	1	759039	13	3.14	>7200	657.74	35.78	30.76	30.64	30.68
ferry8	111	1	290127	30	78.35	>7200	95.15	30.10	29.56	23.21	22.99
ferry10	116	1	539419	38	88.08	>7200	1866.62	70.34	69.84	54.19	53.42
ferry12	123	1	277291	36	47.96	>7200	142.10	37.63	37.50	30.98	30.95
gripper10	125	1	393485	28	69.08	>7200	261.32	52.97	50.53	45.38	44.74
gripper12	129	1	667877	43	50.95	>7200	368.50	106.32	102.87	85.43	84.90
gripper14	118	1	767735	40	47.29	>7200	415.57	111.49	110.40	73.48	71.34

V. EXPERIMENTAL RESULTS

To assess the efficiency of the anytime approach, the algorithm, complete with all its refinements was implemented using the CUDD [20] Decision Diagram package. The rationale for this choice of library was that the Extra DD library [21], which implements Mishchenko's algorithm, also uses CUDD. Experiments were performed on an UltraSPARC IIIi 900MHz based system with 16GB RAM under the Solaris 9 Operating System. All programs — the CUDD package, the Extra library, and our algorithm — were compiled with the GNU C Compiler version 3.3.0 with `-O3` enabled. The algorithms were run against a range of MCNC and ISCAS benchmark circuits of varying size [22], as well as several other benchmarks derived from the SAT literature. All timings were averaged over four runs and are given in seconds. Table I presents the results of these tests, the first four columns of Table I give the circuit name, number of inputs, outputs and the total number of nodes over all outputs respectively. Column five indicates the total number of all symmetric pairs found over each of the outputs of the circuit. Column six gives the time in seconds to read in the benchmark circuit and construct the ROBDD applying variable sifting. The remaining six columns give the runtimes required to compute all symmetric and asymmetric pairs. The first of these is the naïve method for computing all co-factor pairs. The second is Mishchenko's implementation of his own algorithm [21]. The third column is the unoptimized algorithm presented in Section III. The remaining three columns relate to the refinements presented in Section IV, that is, with the optimizations of Sections IV-A, IV-B and IV-C cumulatively enabled. The rationale for implementing the naïve method was to verify the implementation of our algorithm and Mishchenko's; the performance numbers are included to quantify the value of Mishchenko's algorithm. Note, that these figures present Mishchenko's algorithm in best light since when garbage collection is enabled the performance of Mishchenko's implementation can degrade, presumably because of its extensive use of ZDDs [23] to represent sets. For example, the circuit pair requires 33.40s compared to 6.62s with garbage collection disabled. Enabling garbage collection has no perceivable impact on our algorithm.

Figure 2 illustrates the outcome of some experiments designed to explore the anytime nature of the algorithm. In these experiments, the optimized algorithm was stopped after progressively larger timeouts were exceeded. The graphs display the number of symmetries found against these timeouts. Future work will investigate whether reordering the iterations in the main loop, for example, choosing i with the largest number of unknowns, increases the proportion of symmetries found early in the search.

VI. DISCUSSION

This paper presents a novel anytime symmetry detection algorithm, that is capable of detecting all symmetric variable pairs. The startling speed-ups over Mishchenko's algorithm stem from our use of a single static adjacency matrix rather than sets of pairs that are repeatedly generated. It is important to appreciate that there is no obvious way to re-engineer Mishchenko's algorithm to use a static adjacency matrix. This is because Mishchenko's algorithm is a bottom-up, divide and conquer algorithm that derives the solution to a problem by obtaining, and combining, the solutions to several sub-problems. Mishchenko [14, p 1590] points out that caching of the answers to these sub-problems is required to reduce the computational complexity from exponential to polynomial yet this requires multiple data structures to be maintained. By contrast, the anytime approach merely has to mark nodes as visited in any of the ROBDD traversals. Moreover, the only set operations that the anytime algorithm require are atomic

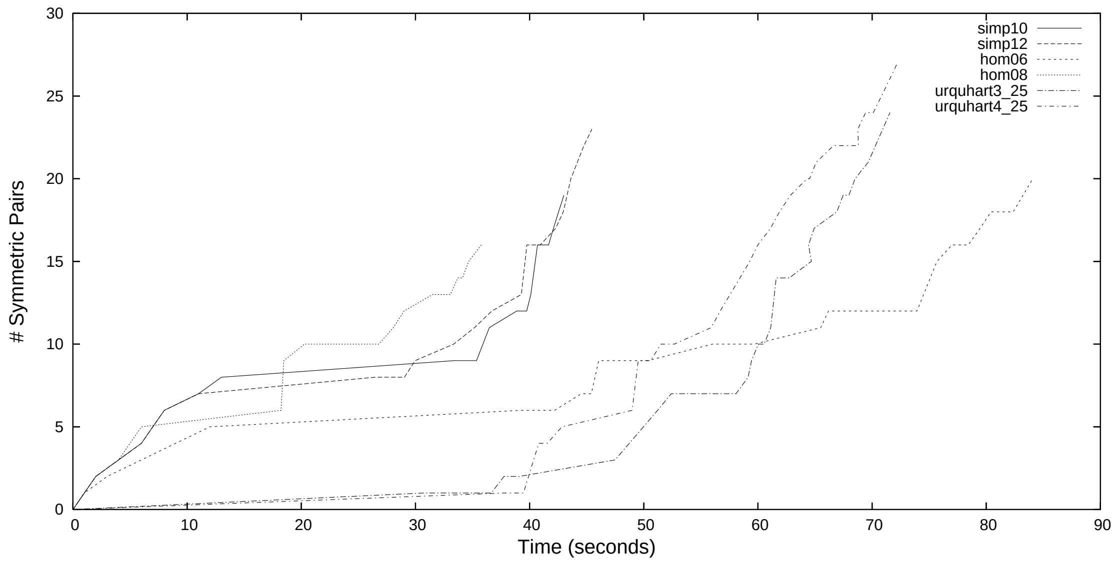

Fig. 2. Time against number of symmetries

$O(1)$ insertions and deletions, which finesses the otherwise $O(n^2)$ overhead of set intersection and union. This partly explains the speed of the anytime approach.

Another source of speedup, in the anytime approach, is its amenability to optimization by enriching the A and S sets on-the-fly. One would think that computing the transitive closure is prohibitively expensive, but close inspection of the SPARC assembler revealed that the GNU compiler was able to generate very tight code from the regular structure of the closure algorithm.

Finally, Mishchenko's algorithm [14] is capable of detecting all four basic types of symmetry, namely, non-skew non-equivalence symmetry — the notion of symmetry considered in this paper — (NE), non-skew equivalence symmetry (E), skew non-equivalence symmetry (!NE) and the skew equivalence symmetry (!E). In this more general setting, a pair of variables are asymmetric if they do not satisfy any of these four symmetry types. A key component of our optimized algorithm, `SymmetryClosure`, can be straightforwardly generalized to infer these transitive symmetries by using a 4-bit encoding to indicate which symmetry types apply. A lookup-table of $(4^2) \times (4^2) = 256$ entries can then be used to obtain the transitive symmetry types without any impact on the asymptotic running time.

Acknowledgments We thank Arnaud Gotlieb and Peter Schachte for useful discussions. This work was supported, in part, by EPSRC Grant-EP/C015517 and British Council Grant-PN 05.021.

References

[1] C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *AIEE Trans.*, vol. 57, pp. 713–723, 1938.

[2] B. G. Kim and D. L. Dietmeyer, "Multilevel Logic Synthesis of Symmetric Switching Functions," *IEEE Trans. Computer-Aided Design*, vol. 10, no. 4, pp. 436–446, 1991.

[3] C. R. Edward and S. L. Hurst, "A Digital Synthesis Procedure Under Function Symmetries and Mapping Methods," *IEEE Trans. Comput.*, vol. C-27, no. 11, pp. 985–997, 1978.

[4] F. Mailhot and G. De Micheli, "Technology Mapping Using Boolean Matching and Don't Care Sets," in *European Design Automation Conference*, 1990, pp. 212–216.

[5] Y. T. Lai, S. Sastry, and M. Pedram, "Boolean Matching Using Binary Decision Diagrams with Applications to Logic Synthesis and Verification," in *International Conference on Computer-Aided Design*, 1992, pp. 452–458.

[6] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams," in *International Conference on Computer-Aided Design*, 1994, pp. 628–631.

[7] C. Scholl, D. Möller, P. Molitor, and R. Drechsler, "BDD Minimization Using Symmetries," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 2, pp. 81–100, 1999.

[8] J. Mohnke and S. Malik, "Permutation and Phase Independent Boolean Comparison," *INTEGRATION, The VLSI Journal*, vol. 16, pp. 109–129, 1993.

[9] D. I. Cheng and M. Marek Sadowska, "Verifying Equivalence of Functions with Unknown Input Correspondence," in *European Design Automation Conference*, 1993, pp. 272–277.

[10] V. N. Kravets and K. A. Sakallah, "Generalized Symmetries in Boolean Functions," in *International Conference on Computer-Aided Design*, 2000, pp. 526–532.

[11] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[12] D. Möller, J. Mohnke, and M. Weber, "Detection of Symmetry of Boolean functions Represented by ROBDDs," in *International Conference on Computer-Aided Design*, 1993, pp. 680–684.

[13] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[14] A. Mishchenko, "Fast Computation of Symmetries in Boolean Functions," *IEEE Trans. Computer-Aided Design*, vol. 22, no. 11, pp. 1588–1593, 2003.

[15] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," in *International Conference on Computer-Aided Design*, 1993, pp. 42–47.

[16] D. Bochmann and B. Steinbach, *Logikenwurf mit XBOOLE: Algorithmen und Programme*. Berlin: Verlag Technik GmBH, 1996.

[17] C. C. Tsai and M. Marek Sadowska, "Generalized Reed-Muller Forms as a Tool to Detect Symmetries," *IEEE Trans. Comput.*, vol. 45, no. 1, pp. 33–40, 1996.

[18] R. W. Floyd, "Algorithm 97: Shortest Path," *Commun. ACM*, vol. 5, no. 6, p. 345, 1962.

[19] S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, 1962.

[20] F. Somenzi, "CUDD Package, Release 2.4.0." [Online]. Available: http://vlsi.colorado.edu/∼fabio/CUDD/cuddIntro.html

[21] A. Mishchenko, "Extra Library of DD Procedures." [Online]. Available: http://www.ee.pdx.edu/∼alanmi/research/extra.htm

[22] "Lgsynth93 Benchmark Set." [Online]. Available: http://www.bdd-portal.org/benchmarks/

[23] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," in *Design Automation Conference*, 1993, pp. 272–277.