JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

Extracting Hardware Circuits from CNF Formulas

Master's Thesis

submitted in partial fulfillment of the requirements

for the academic degree

Diplom-Ingenieur

in the in the Master's Program

Computer Science

Submitted by:

Harald Seltner

At the:

Institute for Formal Models and Verification

Advisor:

Univ.-Prof. Dr. Armin Biere

Linz, July 2014

Abstract

SAT solvers can solve the Boolean satisfiability problem efficiently. For example, they are used for formal verification and other tasks in the field of Electronic Design Automation. Most solvers require input to be in conjunctive normal form (CNF). Logic circuits can be encoded in CNF efficiently using Tseitin transformation. Such a conversion usually causes information loss. The logic paths and gates are lost. Therefore, algorithms have been proposed that aim at reconstructing circuit structures from CNF. Using these techniques might allow to apply circuit-SAT techniques to arbitrary CNFs.

In this work we present the tool `cnf2aig` that can reconstruct circuits from CNFs and outputs them as and-inverter graphs. We give efficient algorithms for detecting the most common hardware gates in CNF. Further we have implemented a solution for the partial MAX-SAT problem that guarantees that the reconstructed circuit is maximal with respect to the gates our algorithms can detect. We show how we use a circuit fuzzer to test our tool. Concluding we give detailed benchmark results using the SAT competition 2013 application benchmarks.

Contents

1. Introduction

Thanks to their efficiency in solving the Boolean satisfiablity problems SAT solvers have become a standard tool in many applications. Despite the Boolean satisfiability problem being NP-compelete, SAT solvers can solve practical problems of interest within a reasonable amount of time. Although not restricted to this field SAT solvers can be used for formal verification in tasks in the area of electronic design automation (EDA).

Most modern SAT solvers expect input to be in conjunctive normal form (CNF). It is possible to transform an electronic circuit into a CNF encoding in linear time. This process is called Tseitin transformation [Tse83]. However, during the transformation of a circuit to CNF certain information is lost. This information can be useful for a SAT solver taking benefit of the circuit structure. Solvers have been implemented that use the circuit from which a CNF was derived and runtime has been decreased considerably.

This work is about extracting circuit structure from CNF. As mentioned above this information could be used to speed up SAT solvers. Ideally the SAT solver would simply use the circuit from which the CNF has been encoded but this circuit may not always be available. This is the case for most benchmarks used in SAT solver competitions. Also we might find circuit structures in CNFs which have not been encoded from electronic circuits.

1.1. Previous Work

In general information loss is unavoidable when encoding a circuit to CNF. Maybe due to this consideration there has only been little effort in extracting circuit structure from CNF. There have been works that extract equivalences [Li00] and simple AND and OR gates [OGMS02]. Roy et al. were the first to our knowledge who explicitly extract logic gates from CNF [RM04]. They introduce the notion of a CNF signature which basically is the CNF encoding of a logic gate. More recent

work in this topic was done by Zhaohui Fu and Sharad Malik [FM07] who not only extract logic gates but also guarantee to extract the biggest acyclic circuit possible.

Roy et al. use a generic graph matcher. Gates in a CNF are found based on sub-graph isomorphism. They do not give much detail about their implementation and they focus on finding (N)AND, (N)OR and NOT gates. They place strong restrictions on the occurences of XOR gates in order to extract them. Fu and Malik provide a more flexible approach. It is based on a gate library which describes the gates to extract. This makes the approach more flexible but less efficient than pattern matching that is specific to gate types. They further guarantee to extract a maximum acyclic circuit. For this purpose they use a SAT solver.

1.2. Outline of `cnf2aig`

This work is strongly based on the work by Fu and Malik [FM07]. We will provide a tool `cnf2aig` that is similar to the tool `cnf2ckt` developed by Fu and Malik. However their tool is not available and our implementation has several differences. In this work we will compare the results of the tools and run `cnf2aig` on new benchmarks from the SAT competition 2013.

The basic workflow of `cnf2aig` is shown in Figure 1.1. After giving basic definitions in Chapter 2 we will describe each step in the workflow in its own chapter. Starting in Chapter 3 we describe how to find gates encoded in a CNF. We give the CNF signatures for the most common gate types and algorithms to find them. We call this process *reverse Tseitin transformation* in reference to the process of encoding gates in CNF called Tseitin transformation. The result of this step is a set of potential gates found in the CNF, called matches. In Chapter 4 we describe how a SAT solver is used to find a subset of these matches that form a biggest acyclic circuit. As result we obtain a set of matches – a maximum cover. This representation of a logic circuit is then written as an and-inverter graph (AIG) as discussed in Chapter 5. We could have used another output format for the circuit. One of the reason to use an AIG is that it allows testing as shown in Chapter 6. Finally we present detailed benchmark results in Chapter 7.

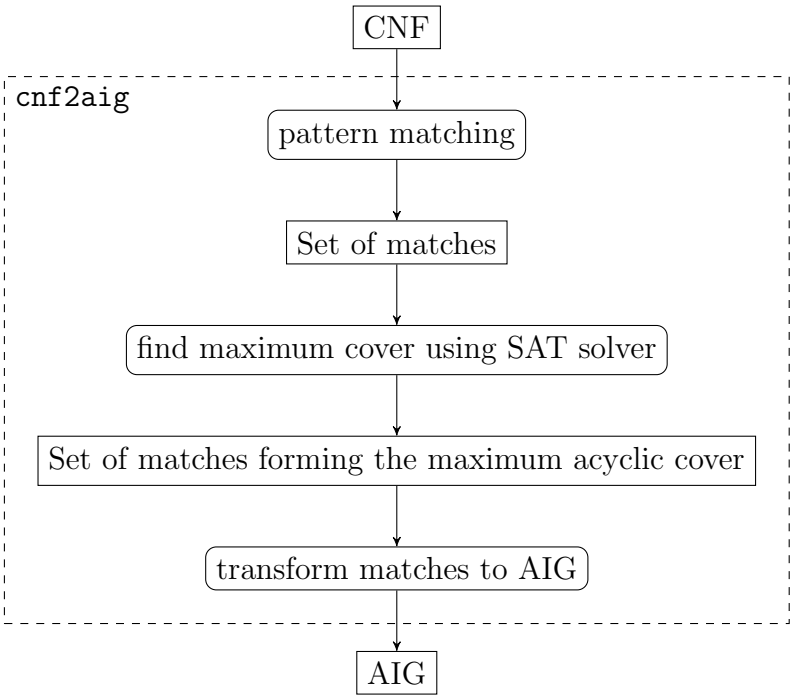


Figure 1.1.: Workflow of `cnf2aig`.

2. Basic Definitions

Before talking about the implementation of `cnf2aig` this chapter explains some concepts used in the rest of this thesis. In this chapter we define the problem of Boolean satisfiability (SAT) and introduce and-inverter graphs. We show what a SAT solver is and give some examples. Readers familiar with those concepts can skip this chapter.

2.1. SAT Problem

Colloquially speaking the Boolean satisfiability problem – also called propositional satisfiablity problem, usually simply denoted as SAT – is about deciding whether an assignment exists such that a propositional formula evaluates to true. In this section we give a formal definition of its concepts. Note that we use the terms propositional and Boolean synonymical.

Definition 1 (Boolean formula) *Let P be the set of all propositional variables. Then the set of Boolean formulas is defined by the following:*

1. $\top$ and $\bot$ are Boolean formulas.
2. If z is a variable of P, then z is a Boolean formula.
3. If Φ_1 and Φ_2 are Boolean formulas, then $(\Phi_1 \wedge \Phi_2)$, $(\Phi_1 \vee \Phi_2)$, $(\Phi_1 \Rightarrow \Phi_2)$ and $(\Phi_1 \Leftrightarrow \Phi_2)$ are Boolean formulas.
4. If Φ is a Boolean formula, then also $\neg\Phi$ is a Boolean formula.

The symbols $\top$ and $\bot$ are called truth values, where $\top$ can be read as true and $\bot$ as false.

A literal is a variable or a negated variable. We say that a literal l is positive if l is a variable and negative otherwise, i.e. if it is a negated variable. The negative

literal that is the negated varialbe v is written as $\neg v$. Instead of writing $\neg l$ we can also use $\bar{l}$. This allows more compact representations.

The set of all variables in a Boolean formual Φ is denoted as $Var(\Phi)$. An assignment of a Boolean formula Φ is a mapping $a : Var(\Phi) \to \{\top, \bot\}$. It maps the variables in Φ to truth values.

Definition 2 (Evaluation) *The value of a Boolean formula Φ under an assignment a is written as $\Phi[a]$. It is the truth value to which Φ evaluates under the assignment a. It is defined by the following:*

1. If Φ is $\top$ then $\Phi[a]$ is $\top$.
2. If Φ is $\bot$ then $\Phi[a]$ is $\bot$.
3. If Φ is a propositional variable, then $\Phi[a]$ is $a(\Phi)$.
4. $\neg\bot$ is $\top$ and $\neg\top$ is $\bot$.
5. If Φ is $\neg\Phi'$ then $\Phi[a]$ is $\neg\Phi'[a]$.
6. If Φ is $\Phi_1 \wedge \Phi_2$ then $\Phi[a]$ is $\top$ if both $\Phi_1[a]$ and $\Phi_2[a]$ are $\top$. Otherwise Φ is $\bot$.
7. If Φ is $\Phi_1 \vee \Phi_2$ then $\Phi[a]$ is $\top$ if at least one of $\Phi_1[a]$ or $\Phi_2[a]$ is $\top$. Otherwise Φ is $\bot$.
8. If Φ is $\Phi_1 \Rightarrow \Phi_2$ then $\Phi[a]$ is $\neg\Phi_1[a] \vee \Phi_2[a]$
9. If Φ is $\Phi_1 \Leftrightarrow \Phi_2$ then $\Phi[a]$ is $(\Phi_1[a] \Rightarrow \Phi_2[a]) \wedge (\Phi_2[a] \Rightarrow \Phi_1[a])$

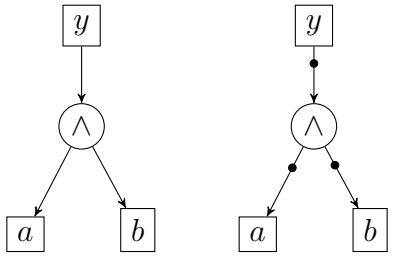
If a Boolean formula Φ evaluates to $\top$ under an assignment a then a is a *model* of Φ. A Boolean formula is called *satisfiable* if there exists a model for it. Otherwise it is *unsatisfiable*. Let A be the set of all assignments of Φ_1 and Φ_2. Two Boolean formulas Φ_1 and Φ_2 are called *logically equivalent* ($\Phi_2 \equiv \Phi_2$) if $\forall a \in A : \Phi_1[a] \Leftrightarrow \Phi_2[a]$ holds. Two Boolean formulas Φ_1 and Φ_2 are called *equisatisfiable* if and only if ($\Phi_1$is satisfiable $\Leftrightarrow$ $\Phi_2$is satisfiable)

A *clause* is a disjunction of literals. The clause with n literals $l_1, \ldots, l_n$ is $l_1 \vee l_2 \vee \cdots \vee l_n$. Note that the empty clause containing no literals evaluates to $\top$. A Boolean formula that is a conjunction of clauses is in *Conjunctive Normal Form* (CNF). For simplicity we simply call a Boolean formula that is in CNF a CNF.

A SAT solver takes a Boolean formula as input and computes whether the output is satisfiable or not. In addition most SAT solvers print the model found if the input is satisfiable. Most solvers take a CNF as input. Although this problem has been shown to be NP-complete modern SAT solvers are able to solve these problems for many interesting inputs.

2.2. And-Inverter Graphs

An *and-inverter graph* (AIG) can represent Boolean formulas. They are described in [KPKG02, KGP01]. We will use it to represent the logic circuit constructed by `cnf2aig`. An AIG is a directed acyclic graph. It consists of input nodes, conjunction nodes, outputs and edges. Each input node represents a Boolean variable. A conjunction node is connected to two child nodes with one edge each. An edge connects a conjunction node with an input node or a conjunction node. Each edge has one additional attribute that specifies whether the value of the child node connected is negated. Graphically this is represented by a dot on the edge. A conjunction node represents the conjunction of its two children. An output is similar to an edge to a node. Like an edge it can be negated. Figure 2.1 shows two examples of AIGs. In both cases y is the only output, a and b are input nodes and there is one conjunction node. Figure 2.1b shows how a disjunction is represented in an AIG. It uses the equivalence $a \vee b \Leftrightarrow \neg(\neg a \wedge \neg b)$.



(a) $y = a \wedge b$ (b) $y = a \vee b$

Figure 2.1.: Examples for and-inverter graphs.

3. Reverse Tseitin Transformation

Tseitin transformation can be used to convert a circuit into a CNF. In this chapter we show how the gates of the original circuit can be found in the created CNF. For this purpose we will introduce the *CNF signatures* of the most common gate types. Further we present algorithms for each of them that find such signatures in a CNF.

3.1. Tseitin Transformation

A lot of SAT instances are fully or partially derived from circuit encoded problems. The technique used for this purpose is called Tseitin transformation. It was first described by G.S. Tseitin in 1983 [Tse83]. It creates a CNF with size linear in the size of the circuit. This procedure allows to encode a circuit in CNF and add constraints. A SAT solver can then verify that the constraints are always respected.

Tseitin transformation works by adding an auxiliary variable – further called Tseitin variable – for each gate in the circuit. The Tseitin variable is then set equal to the logic function of the gate over the inputs. The resulting formula is then transformed to CNF.

An optimization to this technique exists. Plaisted and Greenbaum have shown that it suffices to consider one direction of the equivalence between the Tseitin variable and the logic function of the gate in [PG86]. Gates encoded using this technique are out of the scope of this work and will not be detected by `cnf2aig`. Recent work that deals with this problem is [GB13] by A. Goultiaeva and F. Bacchus. They deal with *Quantified Boolean Formulas* but the principle can be applied to Boolean formulas.

3.2. CNF Signatures

The clauses created from a gate by applying Tseitin transformation is called CNF signature. The term is introduced in [RM04] where some of the following CNF signatures are given too. Deriving the CNF signature of a gate is straight forward. The Tseitin variable of the gate is set equal to the logic function of the gate over its inputs. We will elaborate the derivation on the example of an OR gate with the inputs x_i and the Tseitin variable z:

$$
\begin{gathered}
z \Leftrightarrow \bigvee_i x_i \equiv \\
(z \Rightarrow \bigvee_i x_i) \wedge (\bigvee_i x_i \Rightarrow z) \equiv \\
(\neg z \vee \bigvee_i x_i) \wedge (z \vee \bigwedge_i \neg x_i) \equiv \\
(\neg z \vee \bigvee_i x_i) \wedge (\bigwedge_i z \vee \neg x_i)
\end{gathered}
$$

For a 2-input OR gate with the Tseitin variable z and the input a and b this gives the following CNF signature.

$$
\begin{gathered}
(\neg z \vee a \vee b) \wedge \\
(z \vee \neg a) \wedge \\
(z \vee \neg b)
\end{gathered}
$$

Using the same technique it is simple to deduce the following CNF signatures [RM04]:

$$z \Leftrightarrow \text{and}(x_1, \ldots, x_n) \equiv \left(z \vee \bigvee_{i=1}^{n} \neg x_i\right) \wedge \left(\bigwedge_{i=1}^{n} (\neg z \vee x_i)\right) \tag{3.1}$$

$$z \Leftrightarrow \text{nand}(x_1, \ldots, x_n) \equiv \left(\neg z \vee \bigvee_{i=1}^{n} \neg x_i\right) \wedge \left(\bigwedge_{i=1}^{n} (z \vee x_i)\right) \tag{3.2}$$

$$z \Leftrightarrow \text{or}(x_1, \ldots, x_n) \equiv \left(\neg z \vee \bigvee_{i=1}^{n} x_i\right) \wedge \left(\bigwedge_{i=1}^{n} (z \vee \neg x_i)\right) \tag{3.3}$$

$$z \Leftrightarrow \text{nor}(x_1, \ldots, x_n) \equiv \left(z \vee \bigvee_{i=1}^{n} x_i\right) \wedge \left(\bigwedge_{i=1}^{n} (\neg z \vee \neg x_i)\right) \tag{3.4}$$

The `cnf2aig` tool also matches buffer and inverter gates. The CNF signatures of them are simple. Equation (3.5) shows the CNF signature of a buffer and Equation (3.6) for an inverter.

$$z \Leftrightarrow \text{buffer}(x) \equiv z \Leftrightarrow x \qquad \equiv (\neg z \vee x) \wedge (z \vee \neg x) \tag{3.5}$$

$$z \Leftrightarrow \text{invert}(x) \equiv z \Leftrightarrow \neg x \qquad \equiv (z \vee x) \wedge (\neg z \vee \neg x) \tag{3.6}$$

The CNF signature of XOR and XNOR gates is less simple because encoding the *xor* function over n inputs in CNF requires 2^{n-1} clauses. For an XOR gate each clause contains an even number of negative literals. The following shows this for $n = 3$.

$$\text{xor}(a, b, c) \equiv (a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$$

To derive the CNF signature of XOR gates we consider the equality of the Tseitin variable z with the *xor* function over its inputs and derive the following:

$$\begin{gathered} z \Leftrightarrow \text{xor}(x_1, \ldots, x_n) \equiv \\ (z \Rightarrow \text{xor}(x_1, \ldots, x_n)) \wedge (\text{xor}(x_1, \ldots, x_n) \Rightarrow z) \equiv \\ (\neg z \vee \text{xor}(x_1, \ldots, x_n)) \wedge (z \vee \neg \text{xor}(x_1, \ldots, x_n)) \equiv \\ \neg \text{xor}(x_1, \ldots, x_n, z) \end{gathered}$$

Thus the CNF signatures for XOR and XNOR are the following. Recall that the CNF representation of the XOR function contains 2^{n-1} clauses. Each clause contains exactly n literals. An even number of them is negative. Thus the CNF signature of an XOR gate with n inputs contains 2^n clauses each containing all $n+1$ literals. An odd number of the literals is negative. The CNF signature of an XNOR gate also contains 2^n clauses. Each clause contains $n + 1$ literals, The difference to the signature of an XOR gate is that an *even* number of literals is negative in each clause.

$$z \Leftrightarrow \text{xor}(x_1, \ldots, x_n) \equiv \neg \text{xor}(x_1, \ldots, x_n, z) \tag{3.7}$$

$$z \Leftrightarrow \text{xnor}(x_1, \ldots, x_n) \equiv \text{xor}(x_1, \ldots, x_n, z) \tag{3.8}$$

A majority-of-three gate (MAJ3) is a gate type that is also frequently used, e.g. for the carry out in a full adder circuit. To be able to compare our results to [FM07] `cnf2aig` implements its detection. The output of such a gate is `true` if at least two of its inputs are `true`. The logic function of a MAJ3 gate is shown in Equation (3.9) which requires at least two inputs to be `true`. Similarly the negation is defined in

Equation (3.10) which requires that at least two of the inputs are `false`.

$$\text{maj3}(a,b,c) \equiv (a \vee b) \wedge (a \vee c) \wedge (b \vee c) \tag{3.9}$$

$$\neg\,\text{maj3}(a,b,c) \equiv (\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee \overline{c}) \wedge (\overline{b} \vee \overline{c}) \tag{3.10}$$

Using this representations it is easy to deduce the CNF signature of a MAJ3 gate:

$$\begin{gathered} z \Leftrightarrow \text{maj3}(a,b,c) \equiv \\ (z \Rightarrow \text{maj3}(a,b,c)) \wedge (\text{maj3}(a,b,c) \Rightarrow z) \equiv \\ (\neg z \vee \text{maj3}(a,b,c)) \wedge (z \vee \neg\,\text{maj3}(a,b,c)) \equiv \\ (\overline{z} \vee a \vee b) \wedge (\overline{z} \vee a \vee c) \wedge (\overline{z} \vee b \vee c) \wedge (z \vee \overline{a} \vee \overline{b}) \wedge (z \vee \overline{a} \vee \overline{c}) \wedge (z \vee \overline{b} \vee \overline{c}) \end{gathered}$$

Thus the CNF signature of a MAJ3 gate is the following as also given in [RM04]:

$$\begin{aligned} z \Leftrightarrow \text{maj3}(a,b,c) \equiv (\neg z \vee a \vee b) \wedge (\neg z \vee a \vee c) \wedge (\neg z \vee b \vee c) \wedge \\ (z \vee \neg a \vee \neg b) \wedge (z \vee \neg a \vee \neg c) \wedge (z \vee \neg b \vee \neg c) \end{aligned} \tag{3.11}$$

Note that the majority-of-three gate is different from all other gate types described so far in that its CNF signature has no clause that contains all of the inputs and the Tseitin variable. This has to be considered in the next chapter where we will present an algorithm for detecting gates in CNF formulas.

A gate that also has this property is the if-then-else (ITE) gate. It has three inputs c (condition), t (then) and e (else). It encodes the function "if c then t else e". Note that this gate is the only one discussed in this work where the order of the inputs matters. To derive the CNF signature of the ITE gate we construct a truth table as shown in Table 3.1. To construct the CNF representation from the truth table the lines where the rightmost column is 0 are considered. The literals of each line are negated forming a clause of the CNF representation. This CNF is minimized using the Karnaugh map shown in Figure 3.1. Note that two of the implicants are optional, they are marked by a dashed border. They are optional clauses in the CNF-representation of an ITE gate. The CNF signature is the following where the clauses in the brackets are optional:

$$\begin{aligned} z \Leftrightarrow \text{ite}(c,t,e) \equiv \\ (\neg c \vee \neg t \vee z) \wedge (\neg c \vee t \vee \neg z) \wedge (c \vee \neg e \vee z) \wedge (c \vee e \vee \neg z) \\ [\wedge(\neg t \vee \neg e \vee z) \wedge (t \vee e \vee \neg z)] \end{aligned} \tag{3.12}$$

c	t	e	ite(c,t,e)	z	$z \Leftrightarrow$ ite(c,t,e)
0	0	0	0	0	1
0	0	1	1	0	0
0	1	0	0	0	1
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	0	0	1
1	1	0	1	0	0
1	1	1	1	0	0
0	0	0	0	1	0
0	0	1	1	1	1
0	1	0	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	0	1	0
1	1	0	1	1	1
1	1	1	1	1	1

Table 3.1.: Truth table of an ITE gate.

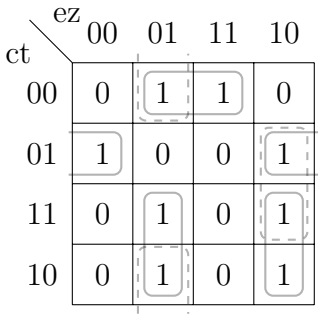


Figure 3.1.: Karnaugh map for the CNF of $z \Leftrightarrow$ ite(c,t,e).

3.3. Detecting CNF Signatures

So far we have seen what the CNF signatures of the most common gate types look like. We have identified two different approaches in the literature to find such signatures. [RM04] uses a generic graph matcher VFLib to find sub-graph isomorphisms. This has the drawback that the inputs are not treated as an unordered set. This results in $n!$ matches for a gate with n inputs even if the sequence of the inputs does not matter. Recall that if-then-else gates are the only type of gates considered in this work where the input sequence matters. [FM07] provides an approach that does not suffer from this problem. Instead a library that specifies the CNF signatures to look for is used with a custom algorithm.

We present custom algorithms for detecting each gate type. This allows to optimize each algorithm based on the CNF signature it detects. Similar to [FM07] the algorithms are based on the observation that most CNF signatures contain a clause that contains all the input variables and the output variable, known as the *key clause*. For example for an AND gate the key clause is a clause with at least three literals where exactly one literal is positive and all others are negative. Further we can see that all clauses in a CNF signatures contain the output variable.

3.3.1. Detect AND, NAND, OR and NOR gates

Detecting AND, NAND, OR and NOR gates is straight forward because we can easily identify a unique key clause for each of them. Recall the CNF signatures given in Equations (3.1), (3.2), (3.3) and (3.4). We observe that each of these CNF signatures has a key clause that contains all inputs and the output. We can determine the type of gate depending on the number of negative literals in the key clause. Let n be the number of literals and p be the number of positive literals in the key clause. We can then determine the gate type depending on p:

p	Gate Type f	Tseitin Variable	Example for $z \Leftrightarrow f(x_1, x_2, x_3)$
1	AND	the only positive literal	$z \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3$
0	NAND	cannot be determined	$\overline{z} \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3$
$n-1$	OR	the only negative literal	$\overline{z} \vee x_1 \vee x_2 \vee x_3$
n	NOR	cannot be determined	$z \vee x_1 \vee x_2 \vee x_3$

Table 3.2.: Associating a key clause to AND, NAND, OR and NOR gates.

```
1  find_and_or_nand_nor():
2      foreach Clause c that has at least 2 literals:
3          int z
4          GateType g
5          check_key_clause(c, out z, out g)
6          switch g
7              case AND, OR:
8                  if find_all_binary_clauses(c, z)
9                      record match
10             case NAND, NOR:
11                 foreach lit in c:
12                     if find_all_binary_clauses(c, lit)
13                         record match
14                         break
15
16 check_key_clause(Clause c, out int z, out GateType g):
17     switch (number of pos. literals in c):
18         case 1:
19             z = the one positive literal in c
20             g = AND
21         case 0:
22             g = NAND
23         case c.length - 1:
24             z = the one negative literal in c
25             g = OR
26         case c.length:
27             g = NOR
28         default:
29             g = NONE
30
31 find_binary_clauses(Clause c, int z):
32     foreach lit in c where c ≠ z:
33         if !find_clause(-lit, -z):
34             return false
35     return true
```

Listing 3.1: Algorithm to detect (N)AND and (N)OR gates

The table also shows what the key clause looks like when z is the Tseitin variable of the gate and the inputs are x_1, x_2, x_3. Based on this observation it is easy to write a function `check_key_clause` that takes a clause c and outputs a gate type if c is the key clause of an AND, NOR, OR or NOR gate. In case of OR and AND gates it also outputs the Tseitin variable z. The algorithm for detecting these gate types is given in Listing 3.1. It iterates over all the clauses in the CNF. For each clause c `check_key_clause` is invoked. If c is a key clause the other clauses of the CNF signature have to be found. In case of AND and OR gates the Tseitin variable is fixed at this point. For NAND and NOR gates each literal in c may represent the Tseitin variable. Thus the algorithm loops over all literals in c (line 11). If all the binary clauses are found a match is recorded.

The `find_binary_clauses` function is simple. It takes the key clause and the Tseitin variable as input. Note that each CNF signature of a gate with i inputs consists out of the key clause and i binary clauses each containing the Tseitin variable and one of the inputs. Both of these literals need to have the opposite phase than they have in the key clause.

3.3.2. Detect Buffers and Inverters

Buffer and inverter gates have very small CNF signatures that consist only of two clauses. Each clause contains the input and the output variable. To detect inverter gates we search for a clause which consists of exactly two positive literals. Then we have to find a clause that has the same two literals but in negated form. If this clause can be found too, two possible matches were found because each of the two variables might be the output variable. The simple algorithm is shown below in Listing 3.2.

```
1 find_inverter():
2     foreach Clause c:
3         if c.length != 2 or c[0] < 0 or c[1] < 0:
4             continue
5         if !find_inverted_clause(c)
6             continue
7         record two inverter matches
```

Listing 3.2: Algorithm to detect inverter gates

Searching for buffer gates is very similar but one problem has to be solved. The CNF signature does not contain one unique key clause. Both clauses contain one

negative and one positive literal. To avoid detecting a buffer gate twice only the clause where the variable that occurs as a negative literal is less than the positive literal is treated as a key clause. Besides that the algorithm is the same as for inverter gates. It also records two matches after it finishes, one for each variable as the output. The algorithm is shown in Listing 3.3.

```
1  find_buffer():
2      foreach Clause c:
3          if c.length != 2 or sign(c[0]) == sign(c[1]):
4              continue
5          int pos = the positive literal in c
6          int neg = the negative literal in c
7          if neg > pos:
8              continue
9          if !find_inverted_clause(c)
10             continue
11         record two buffer matches
```

Listing 3.3: Algorithm to detect buffer gates

3.3.3. Detect XOR and XNOR Gates

To develop an algorithm for detecting XOR gates let us recall what the CNF signature looks like. As shown in equations (3.7) the CNF signature of an XOR gate with inputs $x_1, \ldots, x_i$ and Tseitin variable z is $\neg \operatorname{xor}(x_1, \ldots, x_i, z)$. This is a set of 2^i clauses. Each clause contains all inputs and the Tseitin variable. An odd number of literals are negative. Consider the following example which shows the CNF signature of a 3-input XOR gate with inputs a, b, c and Tseitin variable z.

As before, for better readability, an overbar is used instead of $\neg$.

$$
\begin{aligned}
&(\overline{a} \vee b \vee c \vee z) \wedge \\
&(a \vee \overline{b} \vee c \vee z) \wedge \\
&(a \vee b \vee \overline{c} \vee z) \wedge \\
&(a \vee b \vee c \vee \overline{z}) \wedge \\
&(\overline{a} \vee \overline{b} \vee \overline{c} \vee z) \wedge \\
&(\overline{a} \vee \overline{b} \vee c \vee \overline{z}) \wedge \\
&(\overline{a} \vee b \vee \overline{c} \vee \overline{z}) \wedge \\
&(a \vee \overline{b} \vee \overline{c} \vee \overline{z})
\end{aligned}
$$

For an XOR gate the unique key clause is the clause that contains only one positive literal. The CNF signature of an XNOR gate looks very similar. It contains the same number of clauses but each contains an even number of negative literals. The clause used as key clause contains zero negative literals.

```
1  find_xor_xnor():
2      foreach Clause c with at least 3 literals:
3          if c contains exactly one neg. literal:
4              if c already in XOR match: continue
5              find_other_xor_clauses(c, XOR)
6          else if c contains zero neg. literals:
7              find_other_xor_clauses(c, XNOR)
8
9  find_other_xor_clauses(Clause c, GateType g):
10     unsigned bitmask = 0
11     unsigned bound = 1 << c.length
12     while bitmask < bound:
13         bitmask++
14         if bitmask has odd nr. of high bits: continue
15         Clause f = flip_clause(c, bitmask)
16         if !find_clause(f)
17             return
18     record c.length matches of type g
19
20 flip_clause(Clause c, unsigned bitmask):
21     Clause f
22     for (int i = 0; i < len; i++):
```

```
23          if bitmask & 1:
24              f[i] = c[i] * -1
25          else:
26              f[i] = c[i]
27          bitmask >>= 1
28      return f
```

Listing 3.4: Algorithm to detect X(N)OR gates

The algorithm described in 3.4 is implemented in Lingeling [Bie13] but we are not aware of any description in the literature. The algorithm `find_xor_xnor` starts by finding a key clause. In case of an XOR match we have to make sure that this clause is not already part of an XOR match. For an XOR gate with n inputs there are $n+1$ possible key clauses that contain exactly one negative literal. This check is done in line 4.

After a key clause is found we have to find the other clauses that are part of the CNF signature. The `while` loop in `find_other_xor_clauses` produces all of these clauses in the variable f. This is done by using an unsigned variable *bitmask* that is incremented at each iteration of the while loop. The actual body of the loop is only entered when bitmask contains an odd number of bits that are 1. To produce f the key clause c and the bitmask are passed to `flip_clause`. The bits in bitmask define whether a literal in c should be flipped, i.e. negated. The least significant bit of bitmask specifies whether the first literal in c is negated or not. If this bit is 1 then the the first literal in c is negated and added to f. The i^{th} least significant bit in bitmask defines whether the i^{th} literal in c should be flipped or not. For example when c is $(a \vee b \vee \overline{c} \vee z)$ and bitmask is 5 which is 1001 in binary representation, the resulting flipped clause is $(\overline{a} \vee b \vee \overline{c} \vee \overline{z})$.

The while loop ends when bitmask has been used to create all clauses in the CNF signature except the key clause. This is the case when all bits in bitmask that correspond to a literal in c are zero and the next bit in bitmask is 1. If all clauses have been found matches are recorded. It is important to note that one match for each possible Tseitin variable is recorded.

Obviously this approach places restrictions on the size of X(N)OR gates that can be detected. Assuming *bitmask* has 32 bits the algorithm can find X(N)OR gates with up to 30 inputs. This number is big enough. A 30-input X(N)OR gate has a CNF signature of $2^{30} = 1073741824$ clauses. Such a CNF should not occur in practice.

3.3.4. Detect Majority-of-Three Gates

In contrast to all gate detection algorithms described above the detection of MAJ3 gates cannot rely on the fact that a key clause exists. Instead the fact that all clauses in the CNF signature contain the Tseitin variable is used. The CNF signature is described in Equation (3.11). The following listing assigns all clauses of the CNF signature to variables.

```
1 c1 =  a  b -z
2 c2 =  a  c -z
3 c3 =  b  c -z
4 c4 = -a -b  z
5 c5 = -a -c  z
6 c6 = -b -c  z
```

These variables are used in the algorithm to find MAJ3 gates given in Listing 3.5. It iterates over all variables z in the CNF. For each of these variables it iterates over the clauses where it occurs as a negative literal. Then it checks if this clause may be $c1$. If not, it continues with the next clause where $\neg z$ occurs. After this step the variables a, b and z are fixed. Then it tries to find $c4$ because it is the only clause besides $c1$ that does not contain c. Again, it can continue with the next clause if it is not found. Then the algorithm tries to find $c2$ and c. This is done by iterating over all clauses where a occurs. For each of this clauses v it is checked whether v can be $c2$. If so we have also found c. Now all of the variables in the CNF signature are assigned to an actual variable of the CNF. If v cannot be $c2$ or one of the other clauses cannot be found it continues with the next v. If all clauses of the CNF signature can be found the match is recorded.

Note that after finding $c1$ and $c2$ it suffices to search for $c2$. If we cannot find a variable c such that the clause $(a \vee c \vee \neg z)$ is in the CNF one could assume that we have to search for the clause $(b \vee c \vee \neg z)$ and use this as c. Then we could go on and search for $c3$, $c5$ and $c6$. This is not necessary because switching a and b in the CNF signature results in exactly the same clauses.

3.3.5. Detect If-then-else Gates

Finding if-then-else gates in a CNF is a similar challenge to finding majority-of-three gates. The CNF signature is given in Equation (3.12). The following listing

```
find_maj3():
  foreach Variable z:
    foreach Clause u where -z occurs:
      int a, b, c
      if !is_c1(u, z, out a, out b): continue  // c1
      if !find_clause(-a, -b, z): continue     // c4
      foreach Clause v where a occurs:
        if !is_c2(v, a, b, z, out c): continue // c2
        if !find_clause( b,  c, -z): continue  // c3
        if !find_clause(-a, -c,  z): continue  // c5
        if !find_clause(-b, -c,  z): continue  // c6
        record match

boolean is_c1(Clause c, int z, out int a, out int b);
  if c.length != 3:
    return false
  a = 0
  b = 0
  for (int i = 0; i < 3; i++):
    if c[i] == -z:
      continue
    if c[i] > 0:
      b = a
      a = c[i]
  return a != 0 and b != 0

boolean is_c2(Clause c, int a, int b int z, out int c):
  if c.length != 3:
    return false
  boolean found_a = false
  boolean found_z = false
  for (int i = 0; i < 3; i++):
    switch c[i]:
      case  a: found_a = true
      case -z: found_z = true
      case  b: return false
      default: c = c[i]
  return c > 0 and found_a and found_z
```

Listing 3.5: Algorithm to detect majority-of-three gates

assigns all clauses of the CNF signature to variables. These variables are used in the algorithm below.

```
1 c1 =  a  c -z
2 c2 = -a -b  z
3 c3 =  a -c  z
4 c4 = -a  b -z
5 c5 = -b -c  z  (optional)
6 c6 =  b  c -z  (optional)
```

The algorithm is shown in Listing 3.6. It iterates over all the variables and for each of them over all clauses u where the negated literal occurs. It first searches for $c4$. This is because a and b occur in different phases in this clause. If the clause u is a possible $c4$ we have fixed a and b. Then the algorithm searches for $c2$ which only contains variables that are fixed at this point. Next it searches for $c1$ by iterating over all clauses v where a occurs. After it has found $c1$ and c it searches for $c3$. If it cannot be found it proceeds with the next clause. If $c1$, $c2$, $c3$ and $c4$ have been found it continues to search for $c5$ and $c6$. These are optional clauses. If they are found they are added to the match.

```
1  find_ite():
2    foreach Variable z:
3      foreach Clause u where -z occurs:
4        int a, b, c
5        if !is_c4(u, z, out a, out b): continue  // c4
6        if !find_clause(-a, -b, z): continue     // c2
7        foreach Clause v where a occurs:
8          if !is_c1(v, a, b, z, out c): continue // c1
9          if !find_clause(a, -c, z): continue    // c3
10         find_clause(-b, -c,  z)                // c5
11         find_clause( b,  c, -z)                // c6
12         record match
```

Listing 3.6: Algorithm to detect if-then-else gates

4. Construction of a Maximum Acyclic Cover

In the previous chapter we have shown how to find CNF-signatures in a CNF formula. The result of this is a set of matches M. Each match m consists out of the following:

- A set of clauses $clauses(m)$ forming the CNF-signature of the match.
- A set of variables $in(m)$ that are input signals of the match. Note that in the case of ITE gates the set is ordered.
- A variable $out(m)$ that is the Tseitin variable of the match.
- A gate type $gate(m)$.

As mentioned in the previous chapter not all of these matches can be used to reconstruct the logic circuit. Think about the detection of XOR gates with n inputs. The XOR detection algorithm produces $n+1$ matches but only one of them actually occurs in the original circuit. To construct a maximum set of matches – the cover M_c – we further restrict the circuit to be acyclic as most circuits used in verification tasks are acyclic. Let C denote the set of all clauses in the input CNF formula and V the set of variables in the CNF. $S(c)$ is the set of matches where clause c is involved. [FM07] proposes four constraints on M_c:

1. Every signal in the circuit can be output of at most one match in the cover.
$$\forall v \in V, m_1 \in M_c, m_2 \in M_c : v = out(m_1) \wedge v = out(m_2) \Rightarrow m_1 = m_2$$
2. Each clause can be involved in at most one match in the cover.
$$\forall c \in C, m_1 \in M_c, m_2 \in M_c : c \in clauses(m_1) \wedge c \in clauses(m_2) \Rightarrow m_1 = m_2$$
3. Every matched clause has to be involved in at least one match in the cover.
$$\forall c \in C : \exists m \in M_c : S(c) \neq \emptyset \Rightarrow m \in M_c$$

4. The logic circuit formed by the matches in M_c has to be acyclic.

Note that constraint 2 and 3 are competing. Instead of enforcing constraint 3 we will maximize the number of matched clauses used in the cover. Similar to [FM07] `cnf2aig` uses a SAT solver to enforce these constraints. To avoid mixing up the two CNFs we will further call the CNF we try to extract circuits from the *original CNF* and the newly created CNF to enforce the constraint will be called W. In W we use boolean variables to denote whether a match is used in the cover or not. In this CNF we encode the constraints described above. If and only if the SAT solver assigns a variable to true the according match is part of the cover.

Constraint 1 and 2 can be encoded in a straight forward way. To enforce that only one of n matches can be true we have to add all possible clauses that contain two of the matches as negated literals. For example if the matches m_1, m_2 and m_3 share the same output variable we add the clauses $(\neg m_1 \vee \neg m_2) \wedge (\neg m_1 \vee \neg m_3) \wedge (\neg m_2 \vee \neg m_3)$ to W. In general this adds $\binom{n}{2} = (n-1)n/2$ clauses. The `cnf2aig` tool has an option to enable a more efficient encoding that is also used to encode constraint 3. This encoding uses a parallel counter. It is described in Section 4.1.1.

4.1. Maximizing the Cover

As already mentioned the third constraint targets to maximize the result. However in most cases we will not be able to use all matched clauses in the cover. Instead we will try to maximize the number of matched clauses used in the cover. For this we add clauses to W for each matched clause in the original CNF that contains all the Boolean variables that represent the matches the clause occurs in. Since we do not want to enforce all of these added clauses to be true, we introduce a new variable for each of these clauses that we add to it. We call this variable a *relaxation variable*. In the end we want as many relaxation variables to be false as possible.

More formally this can be described as follows. For each clause c in the original CNF we define a new relaxation variable $r(c)$ and add the clause $\bigvee S(c) \vee r(c)$ to W. If $r(c)$ is false then one of the matches $S(c)$ has to be used in the cover.

We now created the CNF W with a set of clauses that have to be satisfied (constraint 1 and 2) and a set of clauses that are *relaxable* (constraint 3). We want to maximize the number of these relaxable clauses that are satisfied. This is known as the partial MAX-SAT problem [FM06]. We use a different approach than Fu

and Malik to solve the problem. As already mentioned we add a relaxation variable to all relaxable clauses. We want to minimize the number of relaxation variables that are assigned to true.

The first step is to set the SAT solver's default decision of these variables to false. That implies that when the SAT solver cannot decide that a relaxation variable has to be true or false while traversing the search space, it will first try to satisfy the formula by setting the relaxation variable to false. Nevertheless the SAT solver may assign more relaxation variables to true than necessary.

The next step is to further decrease the number of true relaxation variable. We do this by adding a *cardinality constraint* over the relaxation variables. When k relaxation variables are assigned to true after the first call of the SAT solver we add a set of clauses to W that enforces that only $k-1$ variables may be true. If this new CNF cannot be satisfied we have already found the maximum cover in the SAT solver run before.

4.1.1. Encoding a Parallel Counter

The naive way to encode that only k out of n variables may be true in CNF needs $\binom{n}{k+1}$ clauses. The approach is to create all combinations of $k+1$ variables and for each of these encode that at least one of them has to be false. There exist more efficient encodings of cardinality constraints. C. Sinz describes and compares different encodings in [Sin05]. One of these strategies uses a parallel counter. It is based on work of Muller and Preparata [MP75] and is implemented in `cnf2aig`.

The idea is to create a counter of these variables as a logic circuit. The output of the counter is a binary representation of the number of true variables in the input. This counter is then Tseitin transformed into CNF and added to W. Let m denote the number of outputs of the counter. They are a binary representation of the number of true variables in the input. Last we have to add a comparator that is only satisfiable when the binary encoded number is less than or equal to k.

The maximum output of the counter with n inputs is n. This is the case when all n input variables are true. To encode n as a binary number we need $\lfloor \log n \rfloor + 1$ bits. Since we defined that the output consists of m variables we now know that $m = \lfloor \log n \rfloor + 1$. Let $s_0, \ldots, s_{m-1}$ denote the outputs of the counter. They form the binary representation of the number of true variables in the n input variables. s_0 depicts the least significant bit, s_{m-1} the most significant.

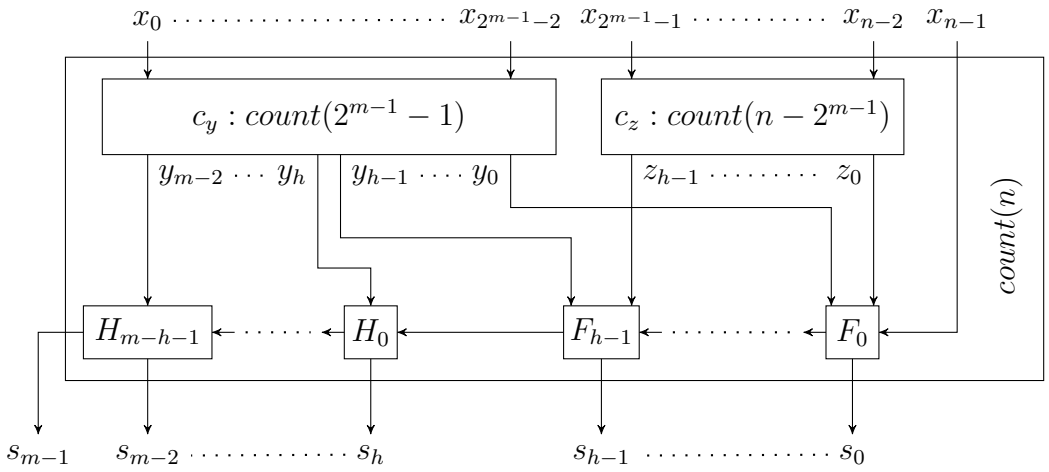


Figure 4.1.: Recursive parallel counter. F depicts a full adder, H a half adder and $count(t)$ a recursive parallel counter with t inputs

Consider for example 7 inputs of the counter are true. Let the number of inputs be 8. Then the number of outputs is $\lfloor \log 8 \rfloor + 1 = 4$. The outputs of the counter are: $s_0 = 1$, $s_1 = 1$, $s_2 = 1$ and $s_3 = 0$ which represents $0111_2 = 7_{10}$.

The implementation of the counter is a recursive parallel one. It is illustrated in Figure 4.1. The n inputs are split in two halves. Both are recursively processed. Both counters output a binary representation of the number of true input variables in their input. These two numbers are summed up by full and half adders forming the final output. To be more precise the inputs $x_0, \ldots, x_{n-1}$ are split in three parts:

- The left half consists of exactly $2^{m-1} - 1$ inputs. It is recursively processed by a counter c_y and produces exactly $m - 1$ outputs $y_0, \ldots, y_{m-2}$.
- The right half consists of 0 to $2^{m-1} - 1$ inputs. It is recursively processed by a counter c_z and produces h outputs $z_0, \ldots, z_{h-1}$ where $0 \leq h \leq m - 1$. Note that there exists cases where this counter does not produce any outputs.
- The last input variable x_{n-1} is summed up together with the results of the counters c_y and c_z.

The reason that the left half processes $2^{m-1} - 1$ inputs is that it should produce $m-1$ outputs. The maximum number that can be depicted in binary representation by $m - 1$ bits is $2^{m-1} - 1$ which is the number of inputs c_y processes. The right counter c_z processes the remaining inputs except the last one. The number of inputs is $n - 2^{m-1}$. It can be shown that this is always less or equal to the number

of inputs c_y processes. There are two extreme cases when c_z processes zero inputs and where it processes as many inputs as c_y. When n is a power of two, i.e. $\lfloor \log n \rfloor = \log n$, c_z processes zero inputs. When n is one less than a power of two, i.e. $\lfloor \log n + 1 \rfloor = \log n + 1$, c_z processes as many inputs as c_y.

The outputs of c_y and c_z and x_{n-1} are summed up by an m-bit adder using conventional full and half adders. A half adder takes two inputs and outputs the sum and a carry. A full adder takes three inputs and outputs a sum and a carry. The first full adder has the inputs x_{n-1}, y_0 and z_0. It produces the least significant bit of the output s_0. The carry of the full adder is input of the next full adder that also has y_1 and z_1 as input. The last full adder has a carry and y_{h-1} and z_{h-1} as inputs. If $m - 1 > h$, i.e. the left half has produced more outputs than the right half, the variables $y_h, \ldots, y_{m-2}$ are added using half adders. In the case where the right half processes no inputs, the m-bit adder consist only of half adders. When the right half processes as many inputs as the left half the adder consists only of full adders.

Figure 4.1 shows the recursive parallel counter with both full and half adders. [Sin05] shows that the number of full adders needed is $n - m$ and the number of half adders is at most $\lfloor \log n \rfloor$. It further gives the equations for full and half adders and their clauses resulting from Tseitin transformation. Note that Plaisted-Greenbaum transformation is used. A half adder with the inputs a and b and the outputs c (carry) and s (sum) adds the three clauses

$$(a \vee \neg b \vee s) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee s) \tag{4.1}$$

and a full adder adder with inputs a, b, c_{in} and outputs c and s the following seven clauses:

$$\begin{aligned}(a \vee b \vee \neg c_{in} \vee s) \wedge (a \vee \neg b \vee c_{in} \vee s) \wedge (\neg a \vee b \vee c_{in} \vee s) \wedge \\ (\neg a \vee \neg b \vee \neg c_{in} \vee s) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee \neg c_{in} \vee c) \wedge (\neg b \vee \neg c_{in} \vee c)\end{aligned} \tag{4.2}$$

Based on the idea described above a cardinality constraint generator has been implemented. It takes an interface of a SAT solver at construction time. It provides two functions: Create a counter over an array of variables and create a comparator that ensures that the number of true variables in the counter's input is less or equal than some number k. Obviously the counter has to be created before any comparator can be created. We now describe the algorithm to create the parallel counter. The comparator will be shown in the next subsection.

Listing 4.1 shows the algorithm that creates a parallel counter. It takes an integer pointer x that points to the sequence of input variables. The second parameter n

```
1  gen_par_count(int *x, int n, out int s0, out int m):
2      if n == 1:
3          s0 = x[0]
4          m = 1
5          return
6
7      int y0, z0, m_y, h = 0
8      m = floor log n
9      int ly = 2 ^ m - 1    // number of inputs for c_y
10     int lz = n - ly - 1   // number of inputs for c_z
11     gen_par_count(x, ly, out y0, out m2)
12     assert(m_y == m)
13     if lz > 0:
14         gen_par_count(x + ly, lz, out z0, out h)
15
16     int carry = x[n-1]
17     generate_adders(carry, y0, m, z0, h, out s0)
18
19 generate_adders(int c_in, int y, int m, int z, int h,
20                  out int s0):
21     int carry = nextVar
22     nextVar += m - 2      // reserve space for m - 2 carries
23     s0 = nextVar;
24     for i in [0...h-1]:
25         int sum = nextVar++
26         if carry == s0:
27             carry = nextVar++
28         gen_full_add(y0 + i, z0 + i, c_in, sum, carry)
29         c_in = carry++
30     for i in [h...m-2]:
31         int sum = nextVar++
32         if carry == s0:
33             carry = nextVar++
34         gen_half_add(y0 + i, c_in, sum, carry)
35         c_in = carry++
```

Listing 4.1: Algorithm to create a parallel counter

is the number of inputs. It outputs the first variable of the output $s0$ and the number of outputs m. The outputs are consecutive. Note that the listing uses the same variable names as Figure 4.1. At first the function checks for the base case where the number of inputs is one. In this case the output is trivial. The value of s_0 is set to the only input. The number of outputs m is one.

Then the inputs are split and $x_0, \ldots, x_{2^{m-1}-2}$ are counted recursively on line 11. It outputs y_0 and the number of outputs produced by c_y which has to be equal to $m-1$. This is asserted in the next line. The output variables $y_0, \ldots, y_{m-2}$ are consecutive. If c_z is to process any inputs this is done on line 14.

Finally, the adder is generated. The `generate_adders` function takes x_{n-1} as the first carry and generates an adder for the outputs of c_y and c_z. The function maintains the next variable that has not yet been used in the SAT solver in $nextVar$. Since the outputs $s_0, \ldots, s_{m-1}$ have to be consecutive it first reserves variables that will be used for the internal carries of the full and half adders. There are $m-2$ internal carries. Note that the last carry is s_{m-1}. It then generates the full adders for $z_0, \ldots, z_{h-1}$ and $y_0, \ldots, y_{h-1}$ in a for-loop on line 24. The next for-loop adds the half adders of the remaining outputs of c_y. Both for-loops have a special handling of the last carry. When $carry$ is the same as the first output variable s_0 the last full or half adder will be added. In this case the carry of the adder is used as the last output s_{m-1}. The function `gen_full_add` adds the seven clauses shown in Figure 4.2 that encode a full adder. It takes the following inputs in order: a, b, c_{in}, s, c. The function `gen_half_add` adds the three clauses shown in 4.1. The inputs are: a, b, s, c.

C. Sinz shows that a parallel counter over n variables adds at most

$$7n - 4\lfloor logn \rfloor - 7. \tag{4.3}$$

clauses to the CNF [Sin05]. He further mentions that the SAT solver has to solve the problem by search. We will see in the evaluation chapter that the problem created can actually be rather hard to solve.

4.1.2. Encoding a Comparator

After initializing the cardinality constraint generator and creating a counter over $x_0, \ldots, x_{n-1}$ it can be used to define an upper limit k on the number of true variables in $x_0, \ldots, x_{n-1}$. Recall that $s_0, \ldots, s_{m-1}$ are a binary representation of the number of true variables in $x_0, \ldots, x_{n-1}$ where s_0 represents the least significant

bit. A precondition to creating the comparator is that k can be expressed as an m-bit binary number, i.e. $k \leq n$.

In [Sin05] a recursive definition of an encoding of such a comparator is given. Let s be the number that is represented by $s_0, \ldots, s_{m-1}$. We write this number as $s_{m-1}s_{m-2}\ldots s_0$. Similarly we write k as $k_{m-1}k_{m-2}\ldots k_0$ so that k_0 is the least significant bit in k and k_{m-1} the most significant. Now we have to encode a comparator in a set of clauses that are only satisfiable when $s \leq k$. Note that when we create the comparator we know the value of k but not the value of s and thus not the values of $s_0, \ldots, s_{m-1}$.

The comparison starts with the most significant bits k_{m-1} and s_{m-1}. If k_{m-1} is zero s_{m-1} has to be zero too and further we have to ensure that $s_{m-1}\ldots s_0 \leq k_{m-1}\ldots k_0$. If k_{m-1} is one we have to consider the value of s_{m-1}. If s_{m-1} is zero the comparison succeeded, i.e. $s \leq k$. If s_{m-1} is one we have to continue and ensure that $s_{m-1}\ldots s_0 \leq k_{m-1}\ldots k_0$. Since the value of s_{m-1} is not known we add $\neg s_0$ to all clauses that will be added when comparing $s_{m-1}\ldots s_0 \leq k_{m-1}\ldots k_0$.

The idea described above can be extended to a formal recursive definition of the comparator that ensures that $s \leq k$. Let $L(i)$ be the boolean formula that encodes $s_i \ldots s_0 \leq k_i \ldots k_0$. It can be defined as:

$$L(0) = \begin{cases} \neg s_0 & \text{if } k_0 = 0 \\ \text{true} & \text{if } k_0 = 1 \end{cases}$$

$$L(i) = \begin{cases} \neg s_i \wedge L(i-1) & \text{if } k_0 = 0 \\ s_i \Rightarrow L(i-1) & \text{if } k_0 = 1 \end{cases}$$

This representation can be transformed to CNF directly. The implication $s_0 \Rightarrow L(i-1)$ can be thought of as adding $\neg s_0$ to all clauses produced by $L(i-1)$. The algorithm to encode $s \leq k$ is shown in Listing 4.2. It takes the first output of the counter s_0, the number of outputs m and k as input. The algorithm traverses the bits of k starting with the most significant bit k_{m-1}. On line 5 `ki` is assigned to k_i. "`>>` (i - 1)" represents a bit shift to the right by $i-1$ bits. "`&`" is a bitwise or. `ki` is either zero or one. `si` represents s_i. If `ki` is zero $-si$ is added to the clause `premises`. If ki is zero the clause which is the union of $\neg s_i$ and `premises` is added to W.

```
1  gen_less_than_comparator(int s0, int m, int k):
2      Clause premises = {}
3      for i in [m - 1 ... 0]:
4          int si = s0 + i
5          int ki = (k >> (i - 1)) & 1
6          if ki = 0:
7              Clause c = {-si} ∪ premises
8              add_clause_to_sat(c)
9          else: // ki is 1
10             premises = {-si} ∪ premises
11         k <<= 1;
```

Listing 4.2: Algorithm to create a binary comparator

4.2. Enforcing the Acyclic Property

We have so far shown how `cnf2aig` enforces the constraints one and two defined in the beginning of this chapter. Further we have shown how it maximises the third constraint. The last constraint enforces the result to be acyclic. This is necessary because we cannot determine the direction of XOR, XNOR, inverter and buffer gates. Consider the circuit shown in Figure 4.2 which is a simplified example from [FM07]. The inverter gate drawn with dashed lines has not been part of the original circuit. Instead the clauses $(y \vee x) \wedge (\neg y \vee \neg x)$ have been added as a constraint to the CNF. Then x and y are the outputs of the XOR gates connected by the inverter. This constraint is recognized as an inverter by `cnf2aig`. The actual outputs of the XOR gates a and b are x and y respectively as shown in Figure 4.2. However `cnf2aig` detects three possible XOR gates for each single XOR gate because it cannot determine which signal is the output. Thus a cycle may be formed. This cycle is marked in thick lines in the circuit.

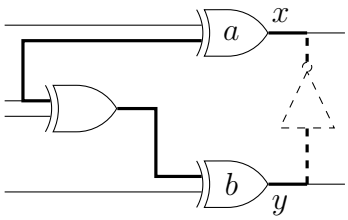


Figure 4.2.: XOR chain that may form a cycle.

To ensure the result is acyclic `cnf2aig` implements the same strategy used in

[FM07]. If the result proposed by the SAT solver is cyclic we add a blocking clause [McM02] to W. For example let m_a, m_b and m_i be the matches that form the circuit shown in Figure 4.2. To avoid this cycle we add the clause $(\neg m_a \vee \neg m_b \vee \neg m_i)$ to W. This inhibits that all three matches are selected by the next invocation of the SAT solver.

To detect cycles in the result of the SAT solver a simple depth first search (DFS) is used. Once a cycle is found the search ends and the blocking clause is added. This search is done in linear time. The downside is that only one of possibly many cycles is found. Therefore `cnf2aig` implements Tarjan's algorithm to find all strongly connected components [Tar72]. It finds all strongly components in the result. Each strongly connected component contains at least one cycle. Further each cycle is contained in a strongly connected component. By forbidding strongly components in the result we will eventually end up with a result that contains no cycle.

4.3. Algorithm to Create the Maximum Acyclic Cover

The techniques described above are used to construct the maximum acyclic cover using the matchings found in the CNF. The results respects the four constraints described in the beginning of this chapter as far as possible. The algorithm that creates the maximum cover is shown in Listing 4.3. It first creates a new instance of a SAT solver. The SAT solver used in this work is lingeling [Bie13]. It then adds clauses that encode constraints one and two as shown above. To maximize constraint 3 it adds clauses that enforce that all matched clauses are used in the cover but adds a relaxation variable to each of them (line 10).

Then the main loop starts. First the SAT solver is called to solve the formula. At the first invocation the result will always be satisfiable. Then it is checked whether the result contains any cycles. If it does a blocking clause is added. This ensures constraint four. If the result does not contain a cycle the cardinality constraint over the relaxation variables is reduced by one. For this the number of true relaxation variables in the result is counted. Let k be this number. Then the constraint that the number of true relaxation variables has to be less or equal than $k-1$ is added. This targets maximizing constraint 3.

Note that W is always satisfiable until the last call when it becomes unsatisfiable.

This approach is dual to the implementation in [FM07]. It has the advantage that the incremental features of the SAT solver can be exploited. An incremental relaxation strategy like implemented in [FM07] cannot keep learned clauses between calls to the SAT solver. The disadvantage of our approach is that the counter adds a considerable amount of clauses to W which makes it hard to solve as our experiments will show. It may appear that the approach is inefficient because the cardinality constraint is only decreased by one after each invocation. However experiments show that this is not the case. It is important that the default decision of the relaxation variables used by the SAT solver is false. Thus the SAT solver will first try to set the relaxation variables to false which leads to a small number of true relaxation variables after the first invocation of the SAT solver.

4.3.1. Reducing Relaxation Variables

In `cnf2aig` we implemented a small optimization to reduce the number of relaxation variables. When a set of clauses $c_1, \ldots, c_{n+1}$ forms the CNF signature of an n-input AND gate and none of the clauses is involved in any other gate `cnf2aig` will find one match m. For constraint 3 it will add the following $n+1$ clauses to the CNF:

$$\bigwedge_{i=1}^{n+1} m \vee r_i.$$

This adds $n+1$ relaxation variables $r_1, \ldots, r_{n+1}$ to the CNF. If a clause is involved in only one match we do not have to do this. Instead we simply add $\neg m$ as input of the counter $n+1$ times. This optimization can be triggered by passing the `--no-single-relax` option to `cnf2aig`.

We have now shown how to obtain a maximum acyclic cover out of all matches found in the CNF. The variables in the SAT solver which correspond to matches and are to true in the final satisfying assignment represent the matches that are part of the cover. These matches can now be written to a hardware circuit.

```
1  create_cover():
2      W = new SatSolver
3      add constraint 1 to W
4      add constraint 2 to W
5      foreach Clause c in the original CNF:
6          Clause b = {}
7          foreach Match m that contains c:
8              b = b ∪ {m}
9          int relax_var = nextVar++
10         b = b ∪ {relax_var}
11         add b to W
12
13     int k = number of relaxation variables
14     boolean satisfiable = true
15     while satisfiable:
16         satisfiable = solve(W)
17         if satisfiable:
18             find cycle and add blocking clause
19             if cycle found:
20                 continue
21             k = number of true relaxation variables
22             if no relaxation variables are true:
23                 current solution is final solution
24             remember current solution
25             decrement_cardinality_constraint(W, k - 1)
26
27     last remembered solution is the final solution
28
29 decrement_cardinality_constraint(SatSolver W, int k):
30     if counter has not been created:
31         add counter over relaxation variables to W
32     add comparator to enforce that number of true
       relaxation variables is ≤ k
```

Listing 4.3: Algorithm to create a the maximum acyclic cover.

5. Generate AIGER Output

The result of the previous chapter is a set of matches that form an acyclic cover. These matches represent a logic circuit that is encoded by the matched part of the CNF. By the matched part we mean the clauses in the CNF that are used in the matches in the cover. There may be unmatched clauses in the CNF which have to be dealt with. The final goal of cnf2aig is to transform the input (a CNF) into a different format (an AIG) preserving the truth value.

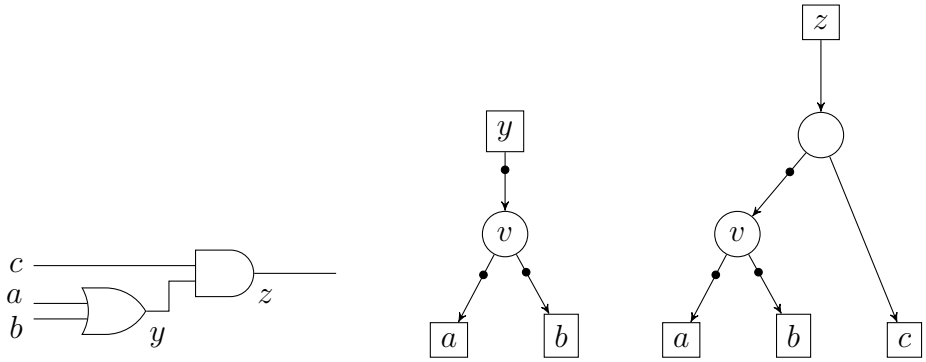
We have chosen the final output to be represented as an and-inverter-graph (AIG). The file format used is AIGER [Bie]. Compared to other formats this has the minor disadvantage that we cannot directly represent the gates like XOR in the output. However it is common to use AIGs to represent arbitrary circuits. It is easy to convert the gates detected by `cnf2aig` to AIGs as we will show in this chapter. Advantages are the tools that are provided to handle the AIGER format. We will use tools from the AIGER library for testing in Chapter 6.

An and-inverter-graph is a directed acyclic graph. Each node in an AIG has two ancestors and represents a conjunction of them. Edges in an AIG may be marked with a dot to indicate negation. We have given a more detailed definition of an AIG in Section 2.2.

5.1. Create AIGs From Matches

We will now show how `cnf2aig` transforms a match to an AIG. Recall that a match consists of a gate type, inputs and one output. In the algorithms shown in the next subsections m will represent the match. A match has n inputs $i_0, \ldots, i_{n-1}$. The output is represented by o.

For each match in the cover `cnf2aig` creates an AIG. Since `cnf2aig` iterates over all matches and creates AIGs for them we have to take special care of the output of such an AIG. Consider a match that represents a 2-input OR gate. The



(a) Circuit (b) OR AIG (c) Overall AIG

Figure 5.1.: AIG created from circuit.

output of this AIG is negated. This means that if the output of the AIG node is used as input in e.g. an AND gate, we have to use it negated. The situation is illustrated in Figure 5.1. Note that the output y is never really added to the AIG. Instead `cnf2aig` has to remember that when node v is used as input it has to be used negated.

In `cnf2aig` each variable is represented by an integer. For a positive literal this integer is positive. The negated literal is represented by a negative integer. In the AIGER library each AIG node is also represented by an integer. In the pseudo code used in this chapter we abstract the AIGER library to accept such integers to create AIG nodes. The AIGER API is shown in Listing 5.1. `aiger_and` takes an integer that represents an AIG node and the integers representing its children as input. The given AIG node has to be greater than 0. The children may be negative integers which denotes a negated edge to the child. For example, to add an AIG with the output 7 and the inputs 1 and 2 we call the function `aiger_and(7, 1, 2)`. As you can see a variable v represented by the integer n in the original CNF is also represented by the integer n in the final AIG.

```
1 aiger_and(int aig_node, int left, int right)
2 aiger_add_output(int output)
```

Listing 5.1: Abstraction of the AIGER API.

To remember which AIG nodes have to be connected with a negated edge when used as input `cnf2aig` maintains a map w. Technically it maps an integer to an integer. Functionally it maps each variable used in a match to the AIG node that should be used as input when this variable is used as input. If a variable is mapped to a negative integer it means that it has to be connected with a negated edge

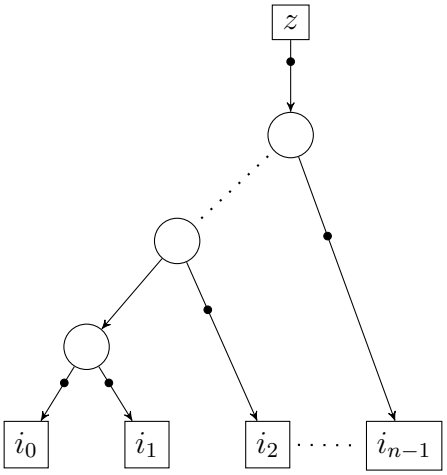


Figure 5.2.: AIG representing n-input OR gate.

when used as input.

5.1.1. AND, NAND, OR and NOR

Converting an AND match to an AIG is trivial. The only question is whether we want to create a balanced tree or not. To keep the algorithms as simple as possible the tree is not balanced. The AIG for an OR gate is given in Figure 5.2. The OR AIG is based on the following simple equivalence (De Morgan's law):

$$\bigvee_k i_k \Leftrightarrow \neg(\bigwedge_k \neg i_k)$$

Obviously the only difference to the AIG representing NOR is that the output z is not negated. Similarly the AIG for an AND gate would contain no negated edges and for a NAND gate only the edge to the output would be negated.

It is easy to see that the only difference for AIGs representing one of the gate types AND, NAND, OR or NOR is whether the edges from inputs and outputs are negated. Table 5.1 shows which gate has which edges negated when transformed to an AIG. When creating an AIG we do not care about whether the output has to be negated. As mentioned above this is only important when the AIG is used as input for something else.

The algorithm for converting these gate types is shown in Listing 5.2. It takes the inputs i, the number of inputs n and the output o as input. It further takes

Gate Type	invert input	invert output
AND	no	no
NAND	no	invert
OR	invert	invert
NOR	invert	no

Table 5.1.: Inversion of edges for AND, NAND, OR, NOR AIGs.

a pointer to a function as argument, which in turn takes an integer and returns an integer. This function is applied to the inputs of the match when adding them to the AIG. As shown in Figure 5.1 we have to pass a function that negates the integer for OR and NOR gates. For AND and NAND gates the identity function is passed.

```
1  and_or_nor_to_aig(int* i, int n, int o):
2      and_nand_or_nor_to_aig(i, n, o, invert)
3
4  and_and_nand_to_aig(int* i, int n, int o):
5      and_nand_or_nor_to_aig(i, n, o, ident)
6
7  and_nand_or_nor_to_aig(int* i, int n, int o,
8                         int (*inv_edge_input)(int)):
9      int t = inv_edge_input(i[0])
10     for k in [1..n - 2]:
11         int aig_node = nextNode++
12         aiger_and(aig_node, t, inv_edge_input(i[k]))
13         t = aig_node
14     aiger_and(o, t, inv_edge_input(i[k]))
15
16 int invert(int x):
17     return -x
18
19 int ident(int x):
20     return x
```

Listing 5.2: Algorithm to create AIG for AND, NAND, OR, NOR gates.

Note that the algorithm maintains a state in the variable *nextNode*. In the beginning it is the minimum integer that is not used in the original CNF.

5.1.2. XOR and XNOR

Constructing AIGs for XOR and XNOR gates is not that straight forward. The AIG for a 2-input XOR is given by the following equivalence:

$$a \oplus b \Leftrightarrow (a \vee b) \wedge (\neg a \vee \neg b) = \neg(\neg a \wedge \neg b) \wedge \neg(a \wedge b).$$

It is easy to see that this encoding needs two additional AIG nodes. In general for an n-input XOR gate $2(n-1)+1$ AIG nodes are added. The AIG constructed for a 2-input XOR gate is shown in Figure 5.3. The algorithm to construct AIGs representing XOR gates with an arbitrary number of inputs is shown in Listing 5.3. Just as Listing 5.2 it uses *nextNode* which is initialized with the minimum integer not used in the original CNF.

Note that XOR can be presented differently with an AIG using only 3 conjunction nodes. It uses the equivalence

$$a \oplus b \Leftrightarrow (\neg a \wedge b) \vee (a \wedge \neg b) = \neg(\neg(\neg a \wedge b) \wedge \neg(a \wedge \neg b)).$$

This representation uses a negated output. It is simpler to use the former representation which does not have a negated output. Further the tree created by Listing 5.3 is not balanced.

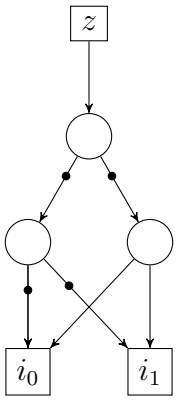


Figure 5.3.: AIG representing a 2-input XOR gate.

```
1  binary_xor_to_aig(int a, int b, int o):
2      int l = nextNode++
3      aiger_and(l, -a, -b)
4      int r = nextNode++
5      aiger_and(r, a, b)
6      aiger_and(o, l, r)
7
8  xor_to_aig(int* i, int n, int o):
9      int t = i[0]
10     for k in [1..n - 2]:
11         int aig_node = nextNode++
12         binary_xor_to_aig(t, i[k], aig_node)
13         t = aig_node
14     binary_xor_to_aig(t, i[k], o)
```

Listing 5.3: Algorithm to create AIG for XOR gates.

5.1.3. Majority-of-3 Gates

A majority-of-3 gate with the output z and inputs a, b and c can be represented by an AIG using the following formula.

$$\text{maj3}(a, b, c) \Leftrightarrow (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \Leftrightarrow \neg(\neg(a \wedge b) \wedge \neg(a \wedge c) \wedge \neg(b \wedge c))$$

The AIG is shown in Figure 5.4. The AIG nodes are named so that the reference to Listing 5.4 is easy to see. As you can see in the listing adding this AIG using the AIGER API is simple.

```
1  maj3_to_aig(int* i, int n, int o):
2      int r = nextNode++
3      int s = nextNode++
4      int t = nextNode++
5      int u = nextNode++
6      aiger_and(r , a, b)
7      aiger_and(s, a, c)
8      aiger_and(t, b, c)
9      aiger_and(u, -r, -s)
10     aiger_and(o, u, -t)
```

Listing 5.4: Constucting an AIG representing a MAJ3 gate.

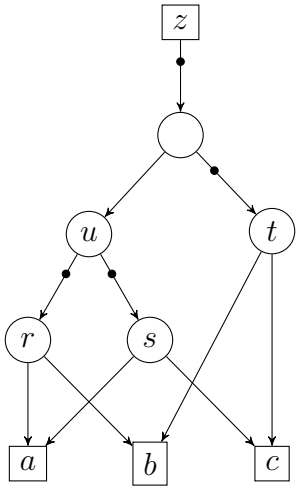


Figure 5.4.: AIG representing a MAJ3 gate.

5.1.4. If-then-else Gates

ITE gates are also easy to represent as an AIG. The equation is

$$\text{ite}(c, t, e) \Leftrightarrow (c \wedge t) \vee (\neg c \wedge e) \Leftrightarrow \neg(\neg(c \wedge t) \wedge \neg(\neg c \wedge e)).$$

The AIG is shown in Figure 5.5. As the figure shows the AIG needs two additional nodes. The output is negated, `cnf2aig` has to store this information in the mapping w as described above.

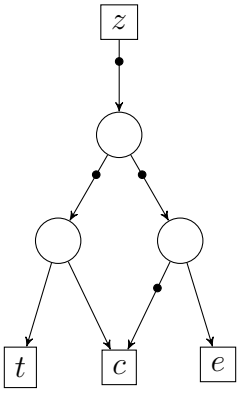
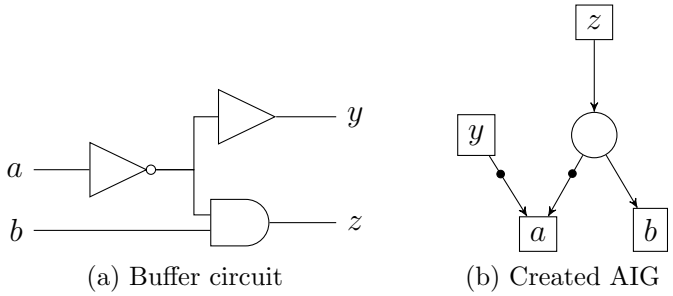


Figure 5.5.: AIG representing an if-then-else gate.



(a) Buffer circuit (b) Created AIG

Figure 5.6.: AIG created for buffer and inverter gates.

5.1.5. Buffers and Inverters

The way `cnf2aig` handles buffer and inverter gates is totally different from all other gate types. Instead of adding an AIG node the buffer's (resp. inverter's) input is used instead. In a preprocessing phase `cnf2aig` creates the mapping w as mentioned above that maps each variable to the AIG node to use when the variable is used as input somewhere. Obviously we can just map a buffer gate to its only input to achieve this.

Figure 5.6 shows an example of what AIG is creates for buffer and inverter gates. The circuit is shown in (a) and the created AIG in (b). You can see that the buffers and inverters are not directly reflected in the created AIG. Listing 5.5 shows the whole preprocessing step that creates the map w as described above. The algorithm can be seen as a variant of the union find algorithm described by R. E. Tarjan [Tar75].

The preprocessing first initializes w with 0 for each variable. This value is not a valid AIG node and will be set to a different value for all variables after preprocessing. For OR, NAND, XNOR, ITE and MAJ3 gates v will get mapped to $-v$ on line 27. Outputs of these gates have to be used with a negated edge when connected to further AIG nodes. For buffers v can in most cases be mapped to the input of the buffer gate. However this does not work if multiple buffers or inverters are chained. In this case we want to map v to the input of the first buffer or inverter in the chain.

The first input of a buffer-inverter chain is returned by $find\_aig\_out$. The function takes the output o of a match m as input and returns the AIG node that should be used instead of this node. For AND, NOR and XOR gate this is o. Outputs of these matches are used as-is. For OR, NAND, XNOR, ITE and MAJ3 it returns $-o$. Outputs of AIGs representing these gates have to be used with a

```
1  preprocess():
2      Map <int, int> w
3      foreach variable v in the CNF:
4          w.put(v, 0)
5      foreach Match m in the cover:
6          w.put(v, find_aig_out(m.inputs[0]))
7      foreach variable v in the CNF:
8          if w.get(v) = 0:
9              w.put(v, v)
10
11 int find_aig_out(int o):
12     Match m = the Match in the cover where o is output
13     switch m.type:
14         case BUFFER:
15             if w.get(m.inputs[0]) != 0:
16                 return w.get(m.inputs[0])
17             int a = find_aig_out(m.inputs[0])
18             w.put(o, a)
19             return a
20         case INVERTER:
21             if w.get(m.inputs[0]) != 0:
22                 return w.get(m.inputs[0])
23             int a = -find_aig_out(m.inputs[0])
24             w.put(o, a)
25             return a
26         case OR, NAND, XNOR, ITE, MAJ3:
27             return -o
28         default:
29             return o
```

Listing 5.5: Preprocessing phase to build w.

negated edge. For buffer and inverter gates the function proceeds recursively. The function $find\_aig\_out$ is called with the only input as parameter. In the case of a buffer the result of this call is returned. With an inverter the result is returned negated. A small optimization is added that stops the recursive search when a buffer or inverter is found that has already been processed. In this case the output of the buffer or inverter is not longer mapped to 0 and the result of $w.get(o)$ can be used as the result.

5.2. Dealing With Unmatched Clauses

We have now transformed all matches in the cover to an and-inverter graph. However we may have only covered a subset of the clauses in the original CNF. In most CNFs of interest we will have to deal with clauses that are unmatched. Even if the CNF was constructed from a hardware circuit and all these gates were detected the CNF will probably contain clauses that place constraints on the hardware circuit. After all this is the use case that a hardware circuit is encoded in CNF. These clauses will likely not form the signature of a gate. Figure 4.2 shows an example where this situation occurs.

In order to maintain satisfiability we have to add all unmatched clauses to the AIG. For this we have to create an AIG representing an OR for each unmatched clause. The literals of the clause are the input of the OR AIG. We have already shown how to construct an OR AIG in Section 5.1.1.

The algorithm for adding the unmatched clauses and the final output to the AIG is shown in Listing 5.6. It first iterates over all unmatched clauses and creates a list *outputs* of integers. This list represents the AIG nodes which are the outputs of the OR AIGs of each clause. If there are unmatched unit clauses the literal in the clause is directly added to *outputs* without creating an OR AIG (line 5). For all other clauses it creates an OR AIG using the function *add_aig_or* which is not shown in the listing (line 8). It simply adds an OR AIG over all the literals in the clause as shown in Section 5.1.1. Finally an AND AIG is created with all *outputs* as input (line 16). The output of this AIG is also the only final output of the AIG created by `cnf2aig`.

There are two special cases to handle. When there is no unmatched clause the output TRUE is added. This means that the original CNF was fully transformed to an acyclic circuit. Since there are no constraints the formula is always satisfiable. The other case is when there is exactly one unmatched clause. In this case the

```
1  add_unmatched_clauses():
2      List <int> outputs
3      foreach clause c NOT in the cover:
4          if c.length = 1:
5              outputs.add(c[0])
6          else:
7              int output = nextNode++
8              add_aig_or(clause, output)
9              outputs.add(-output)
10     if outputs.isEmpty():
11         aiger_add_output(TRUE)
12     else if outputs.size = 1:
13         aiger_add_output(outputs[0])
14     else:
15         int final_output = nextNode++
16         add_aig_and(outputs, final_output)
17         aiger_add_output(final_output)
```

Listing 5.6: Add unmatched clauses and output to AIG.

output of the OR AIG is the final output of the AIG.

6. Testing

The `cnf2aig` tool takes a CNF as input and produces an output in the AIGER format that represents the CNF. Before using the tool for evaluations we want to make sure that it produces AIGER output that actually represents the CNF given. For this purpose we developed "`cnfcktfuzz`" that fuzzes a circuit and produces CNF output by doing simple Tseitin transformations. Then `cnf2aig` is invoked with this CNF as input. The output is then transformed into CNF again by applying Tseitin transformations. The `aigtocnf` tool from the AIGER library is used for this purpose. If the fuzzed CNF and the CNF produced by `aigtocnf` are not equisatisfiable an error in the implementation of one of the tools has been found. The workflow is described in Figure 6.1.

This testing strategy is implemented in a simple script that executes a number of tests. The workflow starting with the first CNF is implemented in a script itself that takes the fuzzed CNF as argument. When an error is found this script can be used with a delta debugger like `cnfdd` [BLB10].

6.1. A Circuit Fuzzer

For creating a CNF input that consists out of many gates a fuzzer `cnfcktfuzz` (CNF Circuit Fuzzer) has been implemented. It creates a rectangular grid of gates which are then connected randomly as will be described in this section. To produce a realistic SAT instance the final outputs of the circuit are randomly connected in clauses consisting of three literals (*3-clauses*).

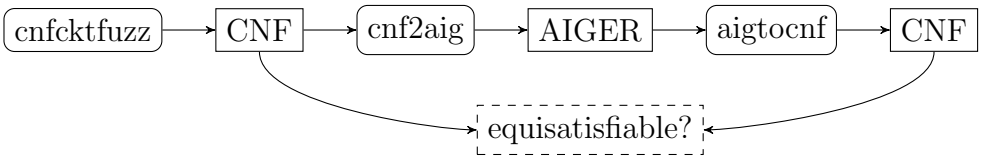


Figure 6.1.: Workflow of testing using cnfcktfuzz

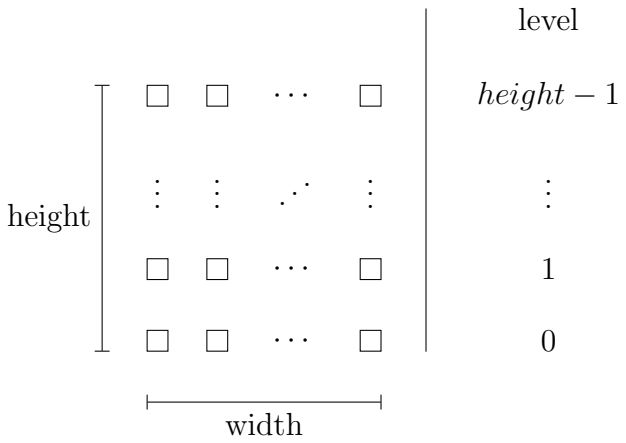


Figure 6.2.: The grid created by cnfcktfuzz

The fuzzer takes the width and the height of the grid of gates to create as parameters. Figure 6.2 shows the created grid which contains exactly $width{*}height$ gates. Each position in the grid is randomly assigned to a gate type. The type of gates produced are:

- AND, NAND
- OR, NOR
- XOR, XNOR
- ITE (if-then-else)
- Buffer
- Inverter
- MAJ3 (majority-of-3)

When filling the grid with gates the fuzzer starts at level 0. All inputs of gates at this level are inputs of the circuit. Then the next level is processed. Each gate is assigned a number of inputs. This number is fixed for some types of gates (ITE, Buffer, Inverter, MAJ3) and is random for all other types. The number of inputs depends on the level the gate is in and the width of the grid. After finding the number of inputs the inputs of a gate at level n where $n > 0$ are connected to the outputs of gates at lower levels.

The number of inputs assigned to level l is

$$\frac{2i}{l+1} \pm d$$

where i is the number of remaining inputs and d is a random deviation. The value of d depends on the width of the grid. Without using a random deviation this results in a linear distribution of the inputs of a gate g at level n with i inputs where $2i/n$ gates of level $n-1$ are inputs of g and 0 gates at level 0 are inputs of g.

6.2. Finding a Factor for the Number of 3-Clauses

The fuzzer should produce an equal amount of satisfiable and unsatisfiable CNFs. If it would only produce on kind of CNFs then `cnf2aig` could pass the tests by always producing this kind of output. To ensure an equal distribution of fuzzed satisfiable and unsatisfiable CNFs a number of random 3-clauses over the final outputs of the circuit is added. This number of 3-clauses added is the number of outputs multiplied by *factor*. This *factor* is configurable. Experiments have shown that for random 3-SAT instances this factor is approximately 4.3 [MSL92] for entirely random CNFs. However since the fuzzed circuit encodes relations of the variables used in the random 3-SAT clauses created the resulting CNF is much more likely to become unsatisfiable. Table 6.1 shows experiments with different values for *factor*. For each *factor* 100 circuits have been created and the percentage of satisfiable CNFs is determined. Figure 6.3 is created from Table 6.1b. It shows that with a *factor* of 0.8 an equal amount of satisfiable and unsatisfiable CNFs is produced.

factor	sat	unsat	%sat
0	97	3	97.00
0.1	97	3	97.00
0.2	90	10	90.00
0.3	73	27	73.00
0.4	64	36	64.00
0.5	62	38	62.00
0.6	58	42	58.00
0.7	54	46	54.00
0.8	42	58	42.00
0.9	44	56	44.00
1	35	65	35.00
1.1	38	62	38.00
1.2	22	78	22.00
1.3	28	72	28.00
1.4	20	80	20.00
1.5	15	85	15.00
1.6	25	75	25.00
1.7	6	94	6.00
1.8	8	92	8.00
1.9	10	90	10.00
2	1	99	1.00

(a) grid size $10 * 10$

factor	sat	unsat	%sat
0	97	3	97.00
0.1	96	4	96.00
0.2	90	10	90.00
0.3	79	21	79.00
0.4	76	24	76.00
0.5	72	28	72.00
0.6	62	38	62.00
0.7	60	40	60.00
0.8	39	61	39.00
0.9	46	54	46.00
1	36	64	36.00
1.1	33	67	33.00
1.2	23	77	23.00
1.3	25	75	25.00
1.4	16	84	16.00
1.5	17	83	17.00
1.6	15	85	15.00
1.7	14	86	14.00
1.8	7	93	7.00
1.9	9	91	9.00
2	6	94	6.00

(b) grid size $100 * 100$

Table 6.1.: Satisfiability of CNFs produced by cnfcktfuzz with different grid sizes

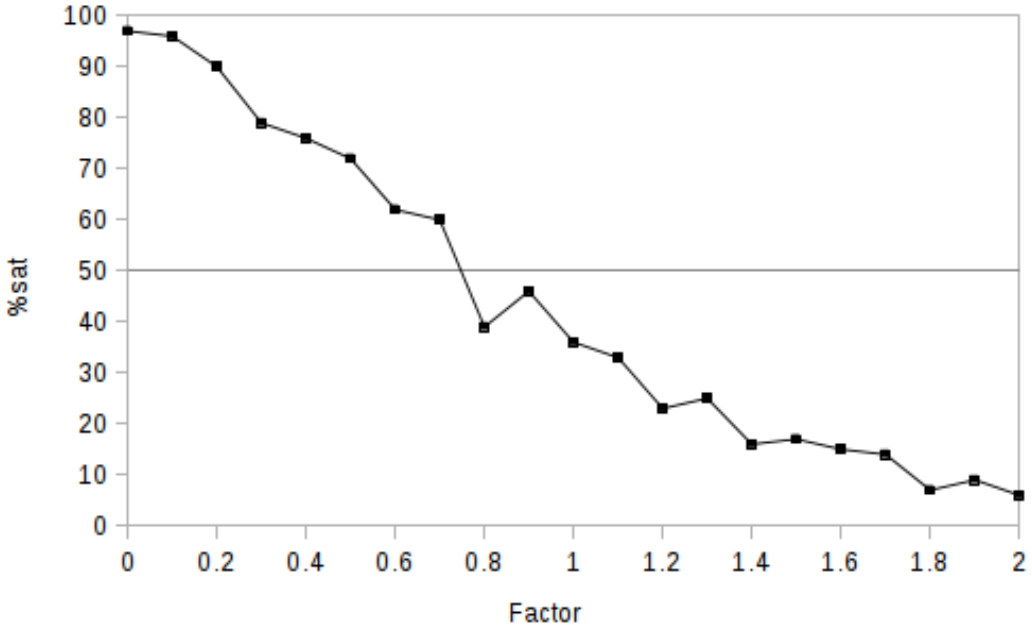


Figure 6.3.: Percentage of satisfiable CNFs produced by cnfcktfuzz

7. Evaluation

To evaluate our implementation we have run `cnf2aig` on several benchmarks. In this chapter we will show our results and give some interpretation. We will further see how different options can influence the behaviour and will give detailed results on run times and the number of gates detected in each benchmark. All benchmarks have been done on an Intel Core i7-2600 processor with 3.4 GHz. The timeout has been set to one hour and the memory limit to 10 GB.

At first we will compare our implementation to the results from [FM07]. Unfortunately the authors give only a few benchmark results and their implementation is not available.

7.1. Comparison to the Implementation by Fu and Malik

We have already outlined the different approach of our implementation compared to [FM07] in Section 4.1. We were able to find some of the benchmarks that have been used in their work and tested our implementation of them. The results are in Table 7.1. We have configured `cnf2aig` to search for the same gates as Fu and Malik did which are the following gates: AND, NAND, OR, NOR up to 10 inputs; XOR, XNOR up to 4 inputs; buffer, inverter and MAJ3 gates.

The first column gives the benchmark name. The second column lists the number of variables and clauses in the benchmark. To save space some columns contain two values on seperate lines. The third column lists the number of found matches in the first line and the number of matches used in the cover in the second line. The abbreviation "$\bigcirc$ Iter." stands for "Cyclic Iterations". This number represents how many times the SAT solver returned *satisfiable* and a cycle was found in the solution. "$\leq$ Iter." depicts the number of times the cardinality constraint was hardened, i.e. the SAT solver returned *satisfiable* and no cycle was found in the solution but the number of true relaxation variables was greater than 0. All timings

Benchmark	Variables / Clauses	Results of `cnf2aig`: Matches in Cover	Clauses in Cover	◯ Iter. / ≤ Iter.	Exec. T. / Match T.	Results of [FM07]: Matches in Cover	Clauses in Cover	◯ Iter. / ≥ Iter.
3bitadd31	8432 31310	0 0	0	0 0	0.063 0.002	0 0	0	46 0
VPN/clause10	2270930 8901946	2073518 -	-	0 1	oot 1.460	1517988 1370111	3977175	0 0
VPN/clause2	75528 272784	63142 -	-	0 1	oot 0.033	48682 43269	125329	0 0
VPN/clause4	267767 1002957	236267 -	-	0 1	oot 0.141	177968 158368	458219	0 0
VPN/clause6	683996 2623082	620373 -	-	0 1	oot 0.386	458858 411405	1192705	0 0
VPN/clause8	1461772 5687554	1335618 -	-	0 1	oot 0.931	980648 886207	2575231	0 0
hanoi6	4968 39666	203 167	567	0 1	0.172 0.010	214 732	732	0 31
IBM/k96	86698 444835	85392 -	-	23 1	oot 0.065	80837 55383	219376	0 192
IBM/k65	89465 398484	104600 -	-	1516 1	oot 0.079	80952 54433	192137	0 0
linvrinv8	1920 6337	5739 1317	4719	10 4	11.205 0.008	3111 1383	4917	3 0
longmult12	5974 18645	7076 4987	14977	0 3	100.119 0.002	6998 4987	14976	0 13
longmult14	7176 22389	8450 6045	18159	0 4	112.482 0.002	8360 6045	18158	0 15
longmult15	7807 24351	9167 6604	19840	0 5	168.655 0.003	9071 6604	19839	0 16
simon/Mat26	744 2464	1822 465	1651	0 4	1.386 0.003	1018 504	1766	0 1
simon/Mat317	24435 85050	71476 -	-	0 1	oot 0.083	39646 17776	64263	0 0
par32-5	3176 10325	11861 2808	8944	689 4	80.143 0.008	7268 3072	9736	0 0
pyhala40401	9638 31795	22279 5813	20480	0 5	150.850 0.019	12703 6384	22193	0 0
xorchain-1-16	46 122	142 -	-	384975 2	oot 0.000	92 30	118	50604 1

Table 7.1.: Results of `cnf2aig` run on benchmarks from [FM07].

are given in seconds. "Match T." is the time needed to find the matches. "Exec. T." is the overall execution time. Benchmarks that timed out are marked with "oot" (out of time). For comparison we give some of the results from [FM07] in the table. "$\geq$ Iter." is the number of times the cardinality constraint is relaxed.

Fu and Malik do not give actual timings of the overall process. They mention that some of their benchmarks timed out but they do not state the amount of time given. Unfortunately `cnf2aig` is not able to solve all benchmarks that have been solved by their implementation within the timeout of one hour. By looking at the timings one can see that nearly the whole time is spent in the SAT solver. The problem is that the CNF created is hard to solve. The majority of the CNF results from the clauses added to maximize constraint 3, i.e. the counter. The maximum number of relaxation variables added is 6754842 in the `clauses-10` benchmark. Applying Equation (4.3) we obtain 47283799 as an upper bound for the number of clauses added to encode the counter. Actually 47283772 clauses are added to encode the counter.

Compared to [FM07] our approach on solving the MAX-SAT problem is an under-approximation. We want to minimize the number of true relaxation varialbes. Fu and Malik assure this by first allowing zero of them to be true and iteratively loosening this constraint if the formula is unsolveable. Our approach is to allow all of the relaxation variables to be true in the beginning. However we set the default value for a relaxation variable to false. This results in a low number of true relaxation variables after the first iteration. We then only harden the constraint by allowing one less relaxation variable to be true. This approach turns out to only need a few iterations until the minimum number of true relaxation variables is hit.

As expected our matching time is much lower. Consider for example the benchmark `clauses-2`. Our implementation takes 35 μs versus Fu and Malik's implementation that takes 2.61 seconds.

Although our approach is different we expected the exact same results in which gates are extracted. However our results differ considerably. In the `clauses-2` benchmark we were able to extract more matches and use more clauses in the cover. On the other hand the `hanoi6` benchmark we extracted less matches and covered less clauses. Unfortunately we do not have an explanation for these variances. Although `cnf2aig` outputs more detailed numbers of what gate types are extracted we cannot compare these numbers to the results by Fu and Malik as they do not provide their implementation or more detailed statistics.

7.2. Evaluation on SAT Competition 2013 Benchmarks

We have run `cnf2aig` on application benchmarks from the SAT Competition 2013 [BBHJ]. In order to display compact tables we have numbered the benchmarks. The numbering is given in Table A.1. Table 7.2 shows some of the results of running `cnf2aig` on the benchmarks with the following options:

- Maximum number of X(N)OR inputs: 4
- Maximum number of (N)AND and (N)OR inputs: 10

The whole results are given in the appendix in Table B.1

Out of the 300 benchmarks 97 timed out within the timeout of 1000 seconds. We can see that nearly the whole time is spent in the SAT solver. In 241 of the benchmarks matches were found. The average percentage of clauses used in the cover is 13.43%. In most cases only a few iterations are needed. The maximum

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
005	13408 478484	74240 4608	50688	0 0	50688	0.066	7.743 4.559
aes4	708 2664	1212 416	1616	19 2	1616	0.001	1.587 1.572
aes6	596 2780	1060 320	2000	15 0	2000	0.001	0.099 0.084
apr1	6196 22741	18062 5898	22004	555 7	22004	0.006	413.695 410.985
apr2	3114 10827	10309 3056	10674	134 5	10674	0.007	112.603 112.238
biv1	973 5428	6737		44571 1	5182	0.009	oot
biv2	972 5432	6979		42712 1	5193	0.009	oot
esa	13842706 53616734			0 0			oom
gri1	330740 2750755	1800		1046 2	9900	0.051	oot
hit1	2260 29903	2154 959	5324	32 1	5324	0.002	15.603 15.493
hit2	2271 30273	2086 952	5154	21 1	5154	0.003	13.669 13.555
hw4	795369 2384932	778417 778417	2335251	0 0	2335251	0.331	11.152 8.747
md51	65604 273512	235919		367 0	257320	0.106	oot

Table 7.2.: Examples of `cnf2aig` run on SAT Competition 2013 benchmars.

number of cardinality iterations is 7. The number of cyclic iterations varies largely depending on the benchmarks. While the majority of the benchmarks cause zero cyclic iterations there exist benchmark types that need more of them. Examples are the *AProVE07**, *bivium-**, *grid-strips-grid-y-**, *gss-**, *hitag2-** and *md5_** benchmarks. Some of these benchmarks timed out so the number of cyclic iterations would probably rise if we increased the timeout.

The coverage varies largely. Most clauses are covered in the *hw** benchmarks with up to 98% in *hw4*. Three of the benchmarks ran out of memory which was limited to 10 Gigabytes.

7.2.1. Reducing Relaxation Variables

In Section 4.3.1 we have described an approach to reduce the number of clauses added by constraint 3. Table B.2 shows the results of the benchmarks that did not

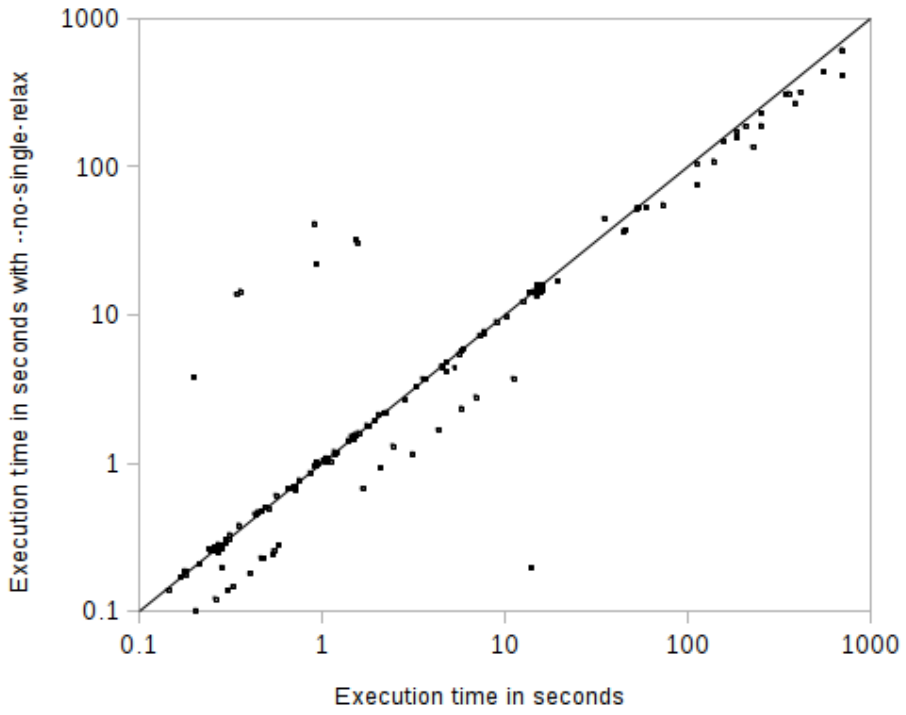


Figure 7.1.: Scatter plot of running `cnf2aig` with and without the `--no-single-relax` option.

time out. As in the benchmarks described above `cnf2aig` is set to find (N)AND and (N)OR gates with up to 10 inputs and X(N)OR gates with up to 4 inputs. Additionally "`--no-single-relax`" is passed. Figure 7.1 shows a scatter plot of the result. As you can see the results vary.

The column "Counter Inputs" lists the number of relaxation variables. Note that in the optimized version directly uses matches as counter inputs if possible. The column "Clauses Saved" lists how many clauses have not been added by constraint 3 because the match is directly used in the counter. This is also the number of how many new variables less are used in the CNF. "Exec. Time" is the overall execution time of the non optimized version and "Optimized Exec. Time" the overall execution time using the `--no-single-relax` option.

The approach reduces the size of the created CNF. Less variables and clauses are used. This causes a speedup of 1.50 in average. Using the optimization two additional benchmarks could be run within the timeout of 1000 seconds. Since there is no drawback in using this optimization it is the preferred way to use `cnf2aig`.

7.2.2. Number of Clauses Added

Since some of the benchmarks timed out it is interesting to investigate where the time is spent when extracting the hardware circuit. As mentioned before nearly the whole time is spent in the SAT solver. There are 4 reasons that may infect the size and hardness of the created CNF:

- Constraint 1 adds clauses to encode that at most one out of some variables may be true.
- Constraint 2 adds clauses to encode that at most one out of some variables may be true.
- Constraint 3 adds clauses in multiple steps:
  - For each clause involved in a match, add a clause to encode that at least one of these matches is true and add a relaxation variable to each clause.
  - If the best solution is not found immediately, add a counter over the relaxation variables.

  - For each iteration add a comparator

- For each cycle found add a blocking clause.

To examine which of the items mentioned above we consider which clauses are added when executing the benchmarks. Table B.3 shows how many clauses are added for which constraints. In average the distribution of which item above adds how many clauses is given in the following table:

Constraint 1	8.45 %
Constraint 2	40.44 %
Constraint 3	6.79 %
Counter	44.32 %
Cycles	0.79 %

Obviously constraint 2 and the counter added add the most clauses. Recall that constraint 2 is a cardinality constraint that is encoded in the naive way here. As described in Chapter 4 this ecnoding needs $\binom{n}{2} = (n-1)n/2$ clauses. On the other hand it is fast to decide by the SAT solver. By looking at the timed out benchmarks in Table B.3 we can identify two main problems: The number of cycles found and a huge counter encoding. Each cycle found leads to a new invocation of the SAT solver. When the counter is huge it is often the case that the first invocation of the SAT solver after adding the counter times out.

7.2.3. Reducing the Number of Clauses Added

The benchmarks above use a naive encoding for the at-most-one constraints 1 and 2. Since `cnf2aig` already implements a parallel counter to encode cardinality constraint we implemented the option `--ccg` that uses this *cardinality constraint generator* to encode constraints 1 and 2. Since we want to reduce the number of clauses this approach is only used when the number of matches involved in the at-most-one contraint succeeds a threshold t. For values less than t the naive encoding is used. Let n be the number of matches involved in the at-most-one constraint. The naive encoding adds

$$\binom{n}{2} = \frac{(n-1)n}{2} \tag{7.1}$$

clauses. The parallel counter and a comparator add

$$7n - 3\lfloor logn \rfloor - 6 \tag{7.2}$$

	without `--ccg`					with `--ccg`				
Bench-mark	Cycl. Iter.	Card. Iter.	Clauses Constr. 1	Clauses Constr. 2	Exec. time	Cycl. Iter.	Card. Iter.	Clauses Constr. 1	Clauses Constr. 2	Exec. time
005	0	0	635584	11070720	7.692	89	1	384864	354740	oot
006	0	0	635584	11070720	7.516	89	1	384864	354740	oot
biv1	97269	0	49045	856074	oot	44063	0	33254		oot
biv2	97726	0	51242	878104	oot	32112	1	34943	36284	oot
biv3	95881	0	50134	915510	oot	31007	1	33949	37198	oot
biv4	97265	0	50970	867644	oot	38286	1	34315	36252	oot
biv5	96579	0	49811	901378	oot	36198	1	33903	36788	oot
biv6	97781	0	49769	887340	oot	43579	0	34081		oot
biv7	56476	1	48546	886756	oot	37278	1	33242	36473	oot
biv8	97591	0	49533	878840	oot	35624	1	33427	36267	oot
biv9	96635	1	47614	866174	oot	32582	1	32895	36280	oot
hit5	34	1	1254	42572	14.303	30	1	1205	36866	9.512
hit15	39	1	1100	40826	15.076	37	1	1051	37286	10.044
hit19	30	1	1236	39532	13.768	29	1	1211	37286	10.113
md51	1133	0	37233626	10728886	oot	0	0	763049		oot
md52	1117	0	37080155	10726782	oot	0	0	762767		oot
md53	1103	0	37284641	10725786	oot	0	0	762932		oot
md54	1116	0	37353345	10733050	oot	0	0	763397		oot
md55	1051	0	38573167	10998612	oot	0	0	783365		oot
md56	1079	0	38416960	10996508	oot	0	0	783086		oot
md57	1057	0	38625094	10995512	oot	0	0	783245		oot
md58	1055	0	38695014	11002776	oot	0	0	783713		oot
md59	1052	0	38956497	11006716	oot	0	0	784190		oot

Table 7.3.: Evaluation of the `--ccg` option.

clauses. When more than 13 matches are involved in the constraint the latter approach adds less clauses. Since the naive encoding is more efficient to solve we have decided to use the parallel counter encoding when $n \geq 15$. The results are given in Table 7.3.Note that the benchmarks were run with the option `--no-single-relax` switched on. Only benchmarks where the number of clauses added differs are shown. This is only the case for 23 benchmarks. The results of the bechmarks *005* and *006* are very surprising. Using the `--ccg` option causes `cnf2aig` to run into a lot of cycles and the benchmarks time out. All benchmarks not shown here produce the same results as when run without `--ccg`.

7.2.4. Blocking Strongly Connected Components

As described in Section 4.2 `cnf2aig` has an option `--scc` to block strongly connected components instead of elementary cycles in the graph of matches. The complete results of this approach are shown in Table B.4. Table 7.4 shows interesting benchmarks.

We can see that with the `--scc` option enabled `cnf2aig` can solve three additional benchmarks within the timeout. These benchmarks are the *grid-strips**

Bench-mark	Variables Clauses	Matches in Cover	Clauses in Cover	Cyclic It. Cardin. It.	Cyclic It. (scc) Cardin. It. (scc)	Exec. Time	Exec. Time (scc)
aes6	596 2780	1060 320	2000	68 2	15 0	2.747	0.099
apr1	6196 22741	18062 5898	22004	1602 6	555 7	472.610	413.695
gri1	330740 2750755	1800 910	9020	438 10	1046 2	67.195	oot
gri2	404100 3361895	2200 1110	11020	539 9	722 1	80.133	oot
gri3	477460 3973035	2600 1310	13020	628 14	753 1	116.440	oot
gre	961 146909	0 0	0	0 0	0 0	0.443	0.431
hit1	2260 29903	2154 948	5280	29 2	32 1	14.072	15.603
hit8	2308 30662	2210 952	5179	15 1	16 0	13.864	0.283

Table 7.4.: Examples of `cnf2aig --scc` run on SAT Competition 2013 benchmars.

benchmarks. They all need many cyclic iterations but less with the `--scc` option enabled. In general the option does not influence runtime much. There are cases where the option causes a cardinality iteration. This increases runtime because the counter is created. Although this option looks promising for some sort of benchmarks it generally decreases performance. Considering only the benchmarks that did not timeout in both cases the average runtime increased from 35.45 seconds to 35.66. The average number of cyclic iterations increased from 6.96 to 12.74 and the average number of cardinality iterations from 0.33 to 0.42.

7.2.5. Allow Melting Literals

One of the reasons that makes it hard for Lingeling to solve the SAT problem is that nearly all variables in the CNF get *freezed*. We use the iterative feature of Lingeling. When we call the SAT solver it can use knowledge about the problem that it gathered in previous runs, e.g. learned clauses. However, since we may add blocking clauses when a cycle has been found, we need to freeze all variables representing matches in the CNF. This prevents Lingeling from removing the variable when it realizes that it is not needed anymore. Note that nearly the whole CNF consists of match variables. Only the internal variables of the counter do not have to be freezed. The outputs of the counter do have to be freezed.

We could create a new instance of the SAT solver each time we call it. Then freezing the match variables would not be necessary. However, this prevents Lingeling from learning across several invocations. We use a different approach. We

```
1 solve(SatSolver solver):
2     solver.solve()
3     if (conflict limit hit):
4         SatSolver clone = lglclone(solver)
5         lglsetopt(clone, "clim", -1);
6         clone.meltall()
7         clone.solve()
8         unclone(clone, solver)
```

Listing 7.1: Algorithm using the clim option of Lingeling.

start Lingeling with a conflict limit set. If Lingeling cannot solve the problem before it reaches this limit of conflicts it returns. If so we create a clone the SAT solver. We then tell the SAT solver that it may remove any variables (all variables are *melted*) and invoce this clone. This clone may now be able to solve the problem much faster because it can do optimizations that the freezing of most of the variables prevented.

In Chapter 4 we have shown the algorithm that uses the SAT solver to construct an acyclic maximum cover (Listing 4.3). When the clim option is used two things change. First the clim option is set on the SAT solver W. This causes the solver to return when it hits the configured number of conflicts. Second, we have to alter the solve function. In this function we create a clone of the solver if it hits the conflict limit. The function is shown in Listing 7.1.

The results are given in Table 7.5. Note that the –no-single-relax option is used to. The conflict limit is set to 100. The table only shows the 149 benchmarks where the conflict limit was hit at least once. A lot of the benchmarks only hit the conflict limit once. In nearly all cases this conflict limit is reached in the first call after the counter has been added. As you can see the option provides no real benefit. Only a few benchmarks, e.g. the *E0** benchmarks, benefit from the option.

Bench-mark	Variables	Clauses	Clim hit	Exec. Time with clim 100	Exec. Time without clim
pip7	39434	887706	1	50.709	55.379
dlx1	106013	1598301	2	oot	977.825
aaa1	25631	142227	4	347.034	105.582
aaa2	50277	283903	3	853.835	305.258
aaa3	53919	308235	4	oot	312.359
aes4	708	2664	6	7.502	1.596
aes5	708	2664	6	7.428	1.594
apr1	6196	22741	15	oot	317.949
apr2	3114	10827	21	oot	75.503
apr5	7729	29194	8	oot	442.341
e01	15364	2210893	1	564.979	614.493
e02	15364	2133873	1	563.236	604.189
e03	10420	395383	1	213.021	229.699
e04	13574	1301188	1	400.315	434.716
e05	9044	295685	1	144.723	155.913
hit1	2260	29903	6	73.672	15.122
hit2	2271	30273	10	116.023	14.115
hit3	2255	29932	24	210.830	14.298
hit4	2249	29810	28	257.880	14.991
hit5	2230	29319	10	121.843	14.303
hit6	2335	31221	6	75.914	14.437
hit9	2294	30374	9	116.060	15.807
hit10	2309	30899	7	85.114	14.135
hit11	2282	30268	12	136.364	15.632
hit12	2273	29849	12	154.172	14.690
hit13	2273	30142	1	12.196	13.690
hit14	2300	30362	11	127.218	13.241
hit15	2262	29727	28	252.866	15.076
hit16	2269	30014	18	210.902	15.564
hit17	2316	31013	8	103.547	14.829
hit18	2328	30885	3	37.928	14.423
hit19	2273	30117	5	64.040	13.768
hit20	2295	30490	4	49.383	14.426
hit21	2331	31022	4	54.193	15.999
hw10	45662	133180	1	246.462	268.277
ndh1	4020	466486	1	13.024	14.363
ndh2	4466	542457	1	15.921	16.972
p01	30447	126201	2	6.738	3.792
pb5	81651	358063	2	68.181	36.732
pb6	140866	627295	1	17.878	21.874
pb12	137600	614115	2	70.994	37.298
pb13	144485	647355	1	38.210	41.231
pb14	145862	654003	3	124.324	44.745
pb16	145943	654778	2	189.733	109.058
pb17	137243	614791	1	172.031	185.368
pb18	145259	653473	2	297.939	173.217
pb19	146595	659920	1	123.817	134.998
pb20	247418	1134514	2	223.076	148.542
pb21	236292	1080258	3	88.367	32.275
pb22	239734	1096970	2	61.149	30.383
pb24	216460	988171	1	175.779	188.988
rbc	2220	148488	1	3.875	4.156
rpo	2384	177941	1	4.056	4.410
vel4	96177	1814189	1	373.178	409.745

Table 7.5.: Examples of `cnf2aig --clim 100` run on SAT Competition 2013 benchmars.

8. Conclusion and Future Work

We have shown that it is possible to partly reconstruct logic circuit structures from CNF. If the type of gates are known and the algorithms can detect all gate types used in the circuit then the whole circuit can be reconstructed. We have combined different approaches on the problem and provided a general strategy to solve the partial MAX-SAT problem.

Detailed benchmarks show that it is possible to find circuit structures even if the CNF has not been encoded from a circuit. This may allow circuit SAT techniques to non circuit encoded CNFs.

The benchmarks of `cnf2aig` have shown that solving the partial MAX-SAT problem by encoding cardinality constraints using a parallel counter is possible and applicable to many inputs. Although the approach scales linearly it produces a hard problem which leads to a non solvable problem for big inputs within a reasonable amount of time.

Further optimization potential exists. Other techniques for solving the partial MAX-SAT problems will probably give better results. Further there exist more efficient at-most-one encodings like the *two product encoding* introduced in [Che10].

Another interesting research topic would be to port the approach described in this work to Quantified Boolean Formulas (QBF). QBFs are an extension to Boolean formulas and can be used to encode circuits as well.

A. Abbreviations for SAT Competition 2013 Benchmarks

The following table shows the abbreviations for the SAT Competition 2013 application benchmarks. These short names are used in the tables presenting our evaluation results.

Table A.1.: Numbering of SAT Competition 2013 benchmars.

Abbreviation	Bechmark Name
005	005
006	006
pip1	10pipe_k
pip2	10pipe_q0_k
pip3	11pipe_11_ooo
pip4	11pipe_q0_k
pip5	7pipe_k
pip6	8pipe_k
pip7	8pipe_q0_k
dlx1	9dlx_vliw_at_b_iq4
dlx2	9dlx_vliw_at_b_iq9
pip8	9pipe_k
vli1	9vliw_m_9stages_iq3_C1_bug10
vli2	9vliw_m_9stages_iq3_C1_bug1
vli3	9vliw_m_9stages_iq3_C1_bug4
vli4	9vliw_m_9stages_iq3_C1_bug6
vli5	9vliw_m_9stages_iq3_C1_bug7
vli6	9vliw_m_9stages_iq3_C1_bug8
vli7	9vliw_m_9stages_iq3_C1_bug9
aaa1	aaai10-planning-ipc5-pathways-13-step17
aaa2	aaai10-planning-ipc5-pathways-17-step20
aaa3	aaai10-planning-ipc5-pathways-17-step21
aaa4	aaai10-planning-ipc5-pipesworld-12-step15
aaa5	aaai10-planning-ipc5-TPP-21-step11
aaa6	aaai10-planning-ipc5-TPP-30-step11
acg1	ACG-15-10p0
acg2	ACG-15-10p1
acg3	ACG-20-5p0
acg4	ACG-20-5p1
aes1	aes_16_10_keyfind_3
aes2	aes_24_4_keyfind_2
aes3	aes_24_4_keyfind_4
aes4	aes_32_3_keyfind_1
aes5	aes_32_3_keyfind_2
aes6	aes_64_1_keyfind_1
apr1	AProVE07-02

Numbering of SAT Competition 2013 benchmars (continued).

Abbreviation	Bechmark Name
apr2	AProVE07-03
apr4	AProVE07-11
apr5	AProVE07-27
apr6	AProVE09-06
arc1	arcfour_initialPermutation_5_32
arc2	arcfour_initialPermutation_6_14
arc3	arcfour_initialPermutation_6_15
arc4	arcfour_initialPermutation_6_16
arc5	arcfour_initialPermutation_6_24
arc6	arcfour_initialPermutation_6_40
arc7	arcfour_initialPermutation_6_56
arc8	arcfour_initialPermutation_6_64
b041	b04_s_2_unknown_pre
b042	b04_s_unknown_pre
b043	b04_s_unknown
biv1	bivium-39-200-0s0-0x163b785faa4bfb1b3b894a9206768a6c3d5d6f038b3797c4c2-99
biv2	bivium-39-200-0s0-0x1b770901581bbb2863c83835583d7ce4e1fafd907076320542-34
biv3	bivium-39-200-0s0-0x28df9231b320bd56dfb68bfc7c3f0ca20dbae6b0eba535ad91-98
biv4	bivium-39-200-0s0-0x53e7622aad02b083b53dcd6a4a76f54a150ceb996ea1dfa300-63
biv5	bivium-39-200-0s0-0x5fa955de2b4f64d00226837d226c955de4566ce95f660180d7-30
biv6	bivium-39-200-0s0-0xdcfb6ab71951500b8e460045bd45afee15c87e08b0072eb174-43
biv7	bivium-40-200-0s0-0x66b619d7b8e447710bf43b794ded6cfaf1e75bb8a947e14c78-50
biv8	bivium-40-200-0s0-0x92fc13b11169afbb2ef11a684d9fe9a19e743cd6aa5ce23fb5-19
biv9	bivium-40-200-0s0-0xd447c33176b6b675fd5f8dc3a5deda46569dc34eedf37da020-6
blo1	blocks-blocks-36-0.130-NOTKNOWN
blo2	blocks-blocks-36-0.150-NOTKNOWN
bun1	b_unsat_pre
bun2	b_unsat
cou	countbitssrl032
ctl1	ctl_3082_415_unsat_pre
ctl2	ctl_3082_415_unsat
ctl3	ctl_3791_556_unsat
ctl4	ctl_4201_555_unsat_pre
ctl5	ctl_4201_555_unsat
ctl6	ctl_4291_567_10_unsat_pre
ctl7	ctl_4291_567_10_unsat
ctl8	ctl_4291_567_11_unsat_pre
ctl9	ctl_4291_567_11_unsat
ctl10	ctl_4291_567_12_unsat_pre
ctl11	ctl_4291_567_12_unsat
ctl12	ctl_4291_567_1_unsat_pre
ctl13	ctl_4291_567_1_unsat
ctl14	ctl_4291_567_2_unsat_pre
ctl15	ctl_4291_567_2_unsat
ctl16	ctl_4291_567_5_unsat
ctl17	ctl_4291_567_6_unsat_pre
ctl18	ctl_4291_567_6_unsat
ctl19	ctl_4291_567_7_unsat_pre
ctl20	ctl_4291_567_7_unsat
ctl21	ctl_4291_567_8_unsat_pre
ctl22	ctl_4291_567_8_unsat
ctl23	ctl_4291_567_9_unsat_pre
ctl24	ctl_4291_567_9_unsat
dat1	dated-10-11-u
dat2	dated-10-13-u
dat3	dated-5-13-u
dat4	dated-5-19-u
e01	E00N23
e02	E00X23

Numbering of SAT Competition 2013 benchmars (continued).

Abbreviation	Bechmark Name
e03	E02F20
e04	E02F22
e05	E04F19
esa	esawn_uw3.debugged
gri1	grid-strips-grid-y-3.045-NOTKNOWN
gri2	grid-strips-grid-y-3.055-NOTKNOWN
gri3	grid-strips-grid-y-3.065-SAT
gre	grieu-vmpc-31
gss1	gss-17-s100
gss2	gss-18-s100
gss3	gss-19-s100
gss4	gss-20-s100
gss5	gss-21-s100
gss6	gss-22-s100
gss7	gss-23-s100
gss8	gss-24-s100
gss9	gss-25-s100
gus1	gus-md5-08
gus2	gus-md5-11
hit1	hitag2-10-60-0-0x2201a94920a2d2e-8
hit2	hitag2-10-60-0-0x8edc44db7837bbf-65
hit3	hitag2-10-60-0-0xa04d664a73eac4d-66
hit4	hitag2-10-60-0-0xa360966c6eb75c4-62
hit5	hitag2-10-60-0-0xac23f1205f76343-96
hit6	hitag2-10-60-0-0xb7b72dfef34c17b-39
hit7	hitag2-10-60-0-0xbc15b17d0353413-10
hit8	hitag2-10-60-0-0xdf7fa6426edec07-17
hit9	hitag2-10-60-0-0xe14721bd199894a-99
hit10	hitag2-10-60-0-0xe6754daf48162bf-46
hit11	hitag2-10-60-0-0xfee9637399d85a2-78
hit12	hitag2-7-60-0-0x39ff85d4ef127de-52
hit13	hitag2-7-60-0-0x5f8ec0ffa4b15c6-25
hit14	hitag2-7-60-0-0xc048b9ebae66e9d-32-SAT
hit15	hitag2-7-60-0-0xe8fa35372ed37e2-80
hit16	hitag2-7-60-0-0xe97b5f1bee04d70-47
hit17	hitag2-8-60-0-0x880693399044612-25-SAT
hit18	hitag2-8-60-0-0xa3b8497b8aad6d7-42
hit19	hitag2-8-60-0-0xb2021557d918860-94
hit20	hitag2-8-60-0-0xdcdbc8bf368ee73-37
hit21	hitag2-8-60-0-0xfba1a41b5dfd7f7-52
hw1	hwmcc10-timeframe-expansion-k45-pdtpmsgoodbakery-tseitin
hw2	hwmcc10-timeframe-expansion-k45-pdtvissoap1-tseitin
hw3	hwmcc10-timeframe-expansion-k50-pdtpmsns2-tseitin
hw4	hwmcc12miters-nonopt-6s126
hw5	hwmcc12miters-nonopt-6s139
hw6	hwmcc12miters-nonopt-6s20
hw7	hwmcc12miters-nonopt-bob12m06
hw8	hwmcc12miters-nonopt-bob12s06
hw9	hwmcc12miters-opt-6s103
hw10	hwmcc12miters-opt-6s133
hw11	hwmcc12miters-opt-6s137
hw12	hwmcc12miters-opt-6s139
hw13	hwmcc12miters-opt-6s153
hw14	hwmcc12miters-opt-6s165
hw15	hwmcc12miters-opt-6s166
hw16	hwmcc12miters-opt-6s19
hw17	hwmcc12miters-opt-6s20
hw18	hwmcc12miters-opt-6s9
hw19	hwmcc12miters-opt-beempgsol2b1

Numbering of SAT Competition 2013 benchmars (continued).

Abbreviation	Bechmark Name
hw20	hwmcc12miters-opt-beempgsol5b1
hw21	hwmcc12miters-opt-bob12m04
hw22	hwmcc12miters-opt-bob12m06
hw23	hwmcc12miters-opt-bob12s02
hw24	hwmcc12miters-opt-bob12s04
hw25	hwmcc12miters-opt-bob12s06
ito1	itox_vc1033
ito2	itox_vc1130
kun	k_unsat
max1	maxor064
max2	maxor128
max3	maxxor032
max4	maxxor064
md51	md5_47_1
md52	md5_47_2
md53	md5_47_3
md54	md5_47_4
md55	md5_48_1
md56	md5_48_2
md57	md5_48_3
md58	md5_48_4
md59	md5_48_5
min1	minandmaxor128
min2	minxor128
min3	minxorminand064
min4	minxorminand128
ndh1	ndhf_xits_19_UNKNOWN
ndh2	ndhf_xits_21_SAT
nos1	nossum-sha-1128-001
nos2	nossum-sha-1128-002
nos3	nossum-sha-1128-003
nos4	nossum-sha-1128-004
nos5	nossum-sha-1128-005
nos6	nossum-sha-1128-006
nos7	nossum-sha-1128-007
nos8	nossum-sha-1128-008
nos9	nossum-sha-1128-009
nos10	nossum-sha-1128-010
nos11	nossum-sha-1144-001
nos12	nossum-sha-1144-002
nos13	nossum-sha-1144-003
nos14	nossum-sha-1144-005
nos15	nossum-sha-1144-007
nos16	nossum-sha-1144-008
nos17	nossum-sha-1144-009
nos18	nossum-sha-1160-001
nos19	nossum-sha-1160-002
nos20	nossum-sha-1160-003
nos21	nossum-sha-1160-007
nos22	nossum-sha-1160-008
nos23	nossum-sha-1160-009
nos24	nossum-sha1-23-64-003
nos25	nossum-sha1-23-64-004
nos26	nossum-sha1-23-80-001
nos27	nossum-sha1-23-96-003
nos28	nossum-sha1-23-96-007
ope	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT
p01	p01_lb_05
par1	partial-10-11-s

Numbering of SAT Competition 2013 benchmarks (continued).

Abbreviation	Bechmark Name
par2	partial-10-17-s
par3	partial-10-19-s
par4	partial-5-13-s
par5	partial-5-15-s
par6	partial-5-17-s
par7	partial-5-19-s
pb1	pb_200_03_lb_01
pb2	pb_200_03_lb_02
pb3	pb_200_03_lb_03
pb4	pb_200_05_lb_00
pb5	pb_200_10_lb_15
pb6	pb_300_01_lb_00
pb7	pb_300_02_lb_06
pb8	pb_300_02_lb_07
pb9	pb_300_03_lb_13
pb10	pb_300_04_lb_05
pb11	pb_300_04_lb_06
pb12	pb_300_05_lb_11
pb13	pb_300_05_lb_16
pb14	pb_300_05_lb_17
pb15	pb_300_06_lb_02
pb16	pb_300_09_lb_07
pb17	pb_300_10_lb_06
pb18	pb_300_10_lb_12
pb19	pb_300_10_lb_13
pb20	pb_400_02_lb_15
pb21	pb_400_03_lb_05
pb22	pb_400_03_lb_07
pb23	pb_400_04_lb_19
pb24	pb_400_05_lb_00
pb25	pb_400_09_lb_02
pb26	pb_400_09_lb_03
pb27	pb_400_09_lb_04
pb28	pb_400_09_lb_05
pb29	pb_400_10_lb_00
pos	post-cbmc-zfcp-2.8-u2
qqu	q_query_3_L70_coli.sat
rbc	rbcl_xits_14_SAT
rpo	rpoc_xits_15_SAT
sat1	SAT_dat.k100
sat2	SAT_dat.k65
sat3	SAT_dat.k80
sat4	SAT_dat.k85
slp1	slp-synthesis-aes-bottom13
slp2	slp-synthesis-aes-bottom14
slp3	slp-synthesis-aes-top25
slp4	slp-synthesis-aes-top26
slp5	slp-synthesis-aes-top27
smt1	smtlib-qfbv-aigs-ext_con_032_008_0256-tseitin
smt2	smtlib-qfbv-aigs-lfsr_004_127_112-tseitin
smt3	smtlib-qfbv-aigs-servers_slapd_a_vc149789-tseitin
tot	total-10-15-s
tra1	transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.020-NOTKNOWN
tra2	transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.030-NOTKNOWN
tra3	transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.040-NOTKNOWN

Numbering of SAT Competition 2013 benchmars (continued).

Abbreviation	Bechmark Name
tra4	transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.050-NOTKNOWN
tra5	transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.060-SAT
tra6	transport-transport-city-sequential-35nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.020-NOTKNOWN
tra7	transport-transport-city-sequential-35nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.030-NOTKNOWN
tra8	transport-transport-city-sequential-35nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.040-NOTKNOWN
tra9	transport-transport-three-cities-sequential-14nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.020-NOTKNOWN
tra10	transport-transport-three-cities-sequential-14nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.030-NOTKNOWN
tra11	transport-transport-two-cities-sequential-15nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.020-NOTKNOWN
tra12	transport-transport-two-cities-sequential-15nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.030-NOTKNOWN
tra13	transport-transport-two-cities-sequential-15nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.040-SAT
ucg	UCG-15-10p1
ur1	UR-15-10p0
ur2	UR-15-10p1
ur3	UR-20-10p1
ur4	UR-20-5p0
ur5	UR-20-5p1
uti1	UTI-15-10p1
uti2	UTI-20-10p0
uti3	UTI-20-10p1
uti4	UTI-20-5p1
vel1	velev-npe-1.0-9dlx-b71
vel2	velev-pipe-sat-1.0-b7
vel3	velev-pipe-sat-1.0-b9
vel4	velev-vliw-uns-4.0-9
vmp1	vmpc_29
vmp2	vmpc_30
vmp3	vmpc_32.renamed-as.sat05-1919
vmp4	vmpc_33
vmp5	vmpc_34
vmp6	vmpc_35.renamed-as.sat05-1921
vmp7	vmpc_36.renamed-as.sat05-1922
zfc	zfcp-2.8-u2-nh

B. Benchmark Results

Table B.1 shows the results of running `cnf2aig` on the benchmarks with the following options:

- Maximum number of X(N)OR inputs: 4
- Maximum number of (N)AND and (N)OR inputs: 10

See Section 7.2 for additional information and interpretation.

Table B.1.: Results of `cnf2aig` run on SAT Competition 2013 benchmarks.

Benchmark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
005	13408 478484	74240 4608	50688	0 0	50688	0.066	7.743 4.559
006	13408 478484	74240 4608	50688	0 0	50688	0.067	7.670 4.533
pip1	67300 3601247	5125 5125	16661	0 0	16661	0.164	5.798 0.051
pip2	77639 2082017	5787		0 4	19195	0.759	oot
pip3	88289 4187694	6883 6883	22379	0 0	22379	1.142	7.745 0.069
pip4	104244 3007883	7328		0 6	24145	1.281	oot
pip5	23909 751116	2523 2523	10516	0 0	10516	0.032	1.169 0.031
pip6	35065 1332773	3463 3463	14790	0 0	14790	0.058	2.048 0.044
pip7	39434 887706	3339 3317	11071	0 5	11123	0.230	73.544 71.937
dlx1	106013 1598301	10062		0 1	47469	0.109	oot
dlx2	482093 9676386	25518		0 1	133851	1.016	oot
pip8	49112 2317839	4436 4436	19221	0 0	19221	0.110	3.686 0.057
vli1	521182 13378625	33260 33260	149160	0 0	149160	31.102	54.519 0.478
vli2	521188 13378641	33260 33260	149160	0 0	149160	30.973	53.553 0.483
vli3	520721 13348117	33248 33248	149137	0 0	149137	30.595	53.511 0.472
vli4	521192 13378781	33260 33260	149160	0 0	149160	30.450	53.336 0.485

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
vli5	521147 13378010	33253 33253	149134	0 0	149134	30.829	53.931 0.460
vli6	521179 13378617	33260 33260	149160	0 0	149160	37.034	59.797 0.481
vli7	521187 13378624	33265 33265	149177	0 0	149177	30.844	53.748 0.480
aaa1	25631 142227	5352 4088	14470	0 2	14472	0.009	112.244 112.021
aaa2	50277 283903	9959 7482	25275	0 2	25277	0.022	347.534 347.100
aaa3	53919 308235	10329 7805	26407	0 2	26409	0.023	358.856 358.380
aaa4	63349 943492	8056		0 1	46824	0.039	oot
aaa5	99736 783991	3807		0 1	19271	0.028	oot
aaa6	314455 2920820	8318		0 1	41305	0.104	oot
acg1	262253 1131732	81846		0 1	279786	0.143	oot
acg2	268258 1155731	82565		0 1	281943	0.141	oot
acg3	324716 1390931	93883		0 1	319753	0.165	oot
acg4	331196 1416850	94282		0 1	320950	0.166	oot
aes1	1168 14656	1708 596	2224	20 0	2224	0.002	0.168 0.103
aes2	520 5968	872 312	1184	0 0	1184	0.000	0.030 0.008
aes3	520 5968	872 312	1184	0 0	1184	0.001	0.032 0.009
aes4	708 2664	1212 416	1616	19 2	1616	0.001	1.587 1.572
aes5	708 2664	1212 416	1616	19 2	1616	0.001	1.587 1.572
aes6	596 2780	1060 320	2000	15 0	2000	0.001	0.099 0.084
apr1	6196 22741	18062 5898	22004	555 7	22004	0.006	413.695 410.985
apr2	3114 10827	10309 3056	10674	134 5	10674	0.007	112.603 112.238
apr4	1004933 4426834	3460439		0 0	3970037	1.307	oot
apr5	7729 29194	25101		1775 9	28992	0.008	oot
apr6	77262 263137	250690		0 0	261884	0.093	oot
arc1	23149 87670	0 0	0	0 0	0	0.003	0.144 0.000
arc2	105743 405350	0 0	0	0 0	0	0.016	0.716 0.000
arc3	105744 405350	0 0	0	0 0	0	0.016	0.716 0.000
arc4	105745 405350	0 0	0	0 0	0	0.016	0.717 0.000
arc5	105753 405350	0 0	0	0 0	0	0.016	0.706 0.000

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
arc6	105769 405350	0 0	0	0 0	0	0.016	0.708 0.000
arc7	105785 405350	0 0	0	0 0	0	0.016	0.720 0.000
arc8	105793 405350	0 0	0	0 0	0	0.016	0.707 0.000
b041	123133 801488	50 25	50	0 0	50	0.082	1.489 0.001
b042	123133 792142	96 48	96	0 0	96	0.080	1.393 0.001
b043	290693 1262210	6 3	6	0 0	6	0.115	2.256 0.000
biv1	973 5428	6737		44571 1	5182	0.009	oot
biv2	972 5432	6979		42712 1	5193	0.009	oot
biv3	973 5570	6842		53697 1	5324	0.005	oot
biv4	971 5434	6911		51020 1	5188	0.009	oot
biv5	978 5514	6874		59114 1	5265	0.009	oot
biv6	974 5505	6878		48036 1	5201	0.009	oot
biv7	971 5469	6725		59569 1	5220	0.004	oot
biv8	970 5432	6775		48849 1	5191	0.009	oot
biv9	972 5450	6675		53248 1	5192	0.008	oot
blo1	530375 9511460	0 0	0	0 0	0	0.231	12.537 0.000
blo2	611755 10974540	0 0	0	0 0	0	0.266	14.344 0.000
bun1	80087 575617	0 0	0	0 0	0	0.055	1.057 0.000
bun2	113407 1010625	0 0	0	0 0	0	0.098	1.936 0.000
cou	18607 55724	11445 11445	34335	0 0	34335	0.005	0.174 0.114
ctl1	9520 97667	0 0	0	0 0	0	0.006	0.180 0.000
ctl2	12376 103844	0 0	0	0 0	0	0.007	0.178 0.000
ctl3	16541 111990	0 0	0	0 0	0	0.005	0.214 0.000
ctl4	14756 141683	0 0	0	0 0	0	0.010	0.269 0.000
ctl5	17374 144538	0 0	0	0 0	0	0.010	0.263 0.000
ctl6	15232 157735	0 0	0	0 0	0	0.012	0.300 0.000
ctl7	17850 154048	0 0	0	0 0	0	0.012	0.278 0.000
ctl8	15232 142785	0 0	0	0 0	0	0.010	0.253 0.000
ctl9	17850 147308	0 0	0	0 0	0	0.010	0.256 0.000

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
ctl10	15232 259243	0 0	0	0 0	0	0.025	0.512 0.000
ctl11	17850 156259	0 0	0	0 0	0	0.012	0.275 0.000
ctl12	15232 147244	0 0	0	0 0	0	0.010	0.254 0.000
ctl13	17850 149131	0 0	0	0 0	0	0.010	0.260 0.000
ctl14	15232 147245	0 0	0	0 0	0	0.010	0.280 0.000
ctl15	17850 149132	0 0	0	0 0	0	0.010	0.255 0.000
ctl16	17850 142693	0 0	0	0 0	0	0.009	0.262 0.000
ctl17	15232 134760	0 0	0	0 0	0	0.009	0.240 0.000
ctl18	17850 142697	0 0	0	0 0	0	0.009	0.252 0.000
ctl19	15232 167062	0 0	0	0 0	0	0.013	0.300 0.000
ctl20	17850 154654	0 0	0	0 0	0	0.011	0.280 0.000
ctl21	15232 142789	0 0	0	0 0	0	0.010	0.247 0.000
ctl22	17850 147312	0 0	0	0 0	0	0.010	0.254 0.000
ctl23	15232 169583	0 0	0	0 0	0	0.013	0.308 0.000
ctl24	17850 154655	0 0	0	0 0	0	0.012	0.272 0.000
dat1	141860 629461	98644		0 1	196992	0.089	oot
dat2	180608 815261	124944		0 1	249492	0.117	oot
dat3	138808 626501	96014		0 1	191742	0.086	oot
dat4	208002 986543	142309		0 1	284427	0.175	oot
e01	15364 2210893	11822 11730	35098	0 1	35098	1.316	702.848 691.057
e02	15364 2133873	11822 11730	35098	0 1	35098	1.392	696.306 685.944
e03	10420 395383	7760 7680	22960	0 1	22960	0.210	253.411 251.847
e04	13574 1301188	10340 10252	30668	0 1	30668	0.922	555.363 548.783
e05	9044 295685	6650 6574	19646	0 1	19646	0.142	186.627 185.420
esa	13842706 53616734			0 0			oom
gri1	330740 2750755	1800		1046 2	9900	0.051	oot
gri2	404100 3361895	2200		722 1	12100	0.061	oot
gri3	477460 3973035	2600		753 1	14300	0.071	oot
gre	961 146909	0 0	0	0 0	0	0.240	0.431 0.000

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
gss1	31318 94116	39999		6259 0	94052	0.016	oot
gss2	31364 94269	40251		6162 0	94205	0.016	oot
gss3	31435 94548	40729		6166 0	94484	0.016	oot
gss4	31503 94748	40923		6069 0	94684	0.017	oot
gss5	31613 95104	41255		6087 0	95040	0.016	oot
gss6	31616 95110	41257		6132 0	95046	0.017	oot
gss7	31711 95400	41451		6086 0	95336	0.016	oot
gss8	31821 95735	41669		6001 0	95671	0.017	oot
gss9	31931 96111	42062		5977 0	96047	0.016	oot
gus1	69397 226300	261107		0 0	226171	0.099	oot
gus2	69561 226787	261657		0 0	226658	0.098	oot
hit1	2260 29903	2154 959	5324	32 1	5324	0.002	15.603 15.493
hit2	2271 30273	2086 952	5154	21 1	5154	0.003	13.669 13.555
hit3	2255 29932	2008 961	5355	36 1	5355	0.002	15.341 15.232
hit4	2249 29810	2032 963	5377	59 1	5377	0.002	15.737 15.613
hit5	2230 29319	2187 945	5277	28 1	5277	0.003	15.291 15.178
hit6	2335 31221	2148 985	5377	33 1	5377	0.002	14.439 14.325
hit7	2321 30992	2132 963	5261	19 1	5261	0.003	14.051 13.934
hit8	2308 30662	2210 954	5187	16 0	5187	0.003	0.283 0.189
hit9	2294 30374	2178 973	5424	39 1	5424	0.002	16.005 15.889
hit10	2309 30899	2104 972	5267	20 1	5267	0.003	15.262 15.153
hit11	2282 30268	2034 975	5408	32 1	5408	0.002	14.972 14.859
hit12	2273 29849	2129 964	5437	42 1	5437	0.002	15.804 15.669
hit13	2273 30142	2191 939	5261	21 0	5261	0.002	0.340 0.237
hit14	2300 30362	2209 960	5296	24 1	5296	0.002	14.887 14.773
hit15	2262 29727	2158 950	5337	39 1	5337	0.002	16.031 15.919
hit16	2269 30014	2043 973	5490	49 1	5490	0.002	15.616 15.501
hit17	2316 31013	2035 984	5389	32 1	5389	0.002	16.008 15.892
hit18	2328 30885	2273 963	5299	21 0	5299	0.003	0.359 0.252

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
hit19	2273 30117	2185 952	5337	28 1	5337	0.002	14.833 14.724
hit20	2295 30490	2142 953	5331	33 1	5331	0.002	15.627 15.515
hit21	2331 31022	2104 985	5489	27 1	5489	0.002	15.149 15.037
hw1	98935 296405	53823		0 1	161382	0.042	oot
hw2	156058 466691	104818		0 1	313768	0.063	oot
hw3	88352 262658	55065		0 1	165138	0.040	oot
hw4	795369 2384932	778417 778417	2335251	0 0	2335251	0.331	11.152 8.747
hw5	618575 1794934	399093 399093	1197279	0 0	1197279	0.250	6.966 4.745
hw6	61555 183916	36359 36359	109077	0 0	109077	0.026	0.579 0.360
hw7	536882 1531966	289303		0 1	867903	0.227	oot
hw8	536832 1531816	289303		0 1	867903	0.229	oot
hw9	688509 2016502	441370		0 1	1324109	0.321	oot
hw10	45662 133180	27924 27923	83769	0 1	83771	0.020	391.087 390.752
hw11	723647 2070235	458615		0 1	1375844	0.350	oot
hw12	508510 1464739	326435 326435	979305	0 0	979305	0.219	5.827 4.006
hw13	58951 165355	30587		0 1	91760	0.023	oot
hw14	3098 8995	2355 2355	7065	0 0	7065	0.002	0.039 0.024
hw15	5448 15994	4162 4162	12486	0 0	12486	0.002	0.066 0.041
hw16	31141 90805	19926 19926	59778	0 0	59778	0.013	0.306 0.192
hw17	58614 175093	31742 31742	95226	0 0	95226	0.025	0.546 0.319
hw18	33563 98113	21505 21505	64515	0 0	64515	0.014	0.325 0.210
hw19	21464 61561	13011 13011	39033	0 0	39033	0.008	0.201 0.126
hw20	21465 61564	13012 13012	39036	0 0	39036	0.007	0.197 0.125
hw21	626166 1675696	299114		0 1	897338	0.258	oot
hw22	482635 1369225	326117		0 1	978347	0.216	oot
hw23	26233 77737	17008 17008	51024	0 0	51024	0.011	0.261 0.164
hw24	542367 1424299	295385		0 1	886151	0.234	oot
hw25	482548 1368964	326036		0 1	978105	0.208	oot
ito1	118700 348058	286725		326 0	245373	0.749	oot

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
ito2	152428 441729	358339		0 0	310183	0.814	oot
kun	7021 95854	0 0	0	0 0	0	0.006	0.182 0.000
max1	50996 151835	29553 29553	88659	0 0	88659	0.016	0.481 0.297
max2	200308 598619	116465 116465	349395	0 0	349395	0.069	2.104 1.359
max3	13240 39143	7667 7667	23001	0 0	23001	0.003	0.123 0.074
max4	51064 152039	29683 29683	89049	0 0	89049	0.016	0.464 0.290
md51	65604 273512	235919		367 0	257320	0.106	oot
md52	65604 273504	235839		404 0	257220	0.105	oot
md53	65604 273522	235899		389 0	257292	0.103	oot
md54	65604 273506	235979		413 0	257308	0.103	oot
md55	66892 279248	241537		359 0	262850	0.108	oot
md56	66892 279240	241457		355 0	262750	0.109	oot
md57	66892 279258	241517		360 0	262822	0.107	oot
md58	66892 279242	241597		341 0	262838	0.104	oot
md59	66892 279256	241697		352 0	262902	0.106	oot
min1	249327 746444	182387 182387	547161	0 0	547161	0.090	3.145 2.219
min2	54258 160469	35568 35568	106704	0 0	106704	0.016	0.544 0.354
min3	40042 119357	26731 26731	80193	0 0	80193	0.013	0.403 0.269
min4	153834 459965	102635 102635	307905	0 0	307905	0.054	1.670 1.155
ndh1	4020 466486	1911 1820	5369	0 1	5369	0.067	15.849 15.064
ndh2	4466 542457	2093 2002	5915	0 1	5915	0.087	19.491 18.610
nos1	4128 126548	6464 896	7424	0 0	7424	0.012	1.066 0.516
nos2	4128 126548	6464 896	7424	0 0	7424	0.012	1.061 0.518
nos3	4128 126548	6464 896	7424	0 0	7424	0.013	1.059 0.516
nos4	4128 126548	6464 896	7424	0 0	7424	0.012	1.062 0.515
nos5	4128 126548	6464 896	7424	0 0	7424	0.012	1.058 0.515
nos6	4128 126548	6464 896	7424	0 0	7424	0.012	1.077 0.528
nos7	4128 126548	6464 896	7424	0 0	7424	0.012	1.063 0.520
nos8	4128 126548	6464 896	7424	0 0	7424	0.012	1.067 0.515

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
nos9	4128 126548	6464 896	7424	0 0	7424	0.012	1.049 0.516
nos10	4128 126548	6464 896	7424	0 0	7424	0.012	1.077 0.530
nos11	4128 126564	6464 896	7424	0 0	7424	0.012	1.045 0.514
nos12	4128 126564	6464 896	7424	0 0	7424	0.011	1.060 0.513
nos13	4128 126564	6464 896	7424	0 0	7424	0.011	1.062 0.515
nos14	4128 126564	6464 896	7424	0 0	7424	0.012	1.123 0.514
nos15	4128 126564	6464 896	7424	0 0	7424	0.012	1.073 0.528
nos16	4128 126564	6464 896	7424	0 0	7424	0.012	1.076 0.526
nos17	4128 126564	6464 896	7424	0 0	7424	0.012	1.067 0.515
nos18	4128 126580	6464 896	7424	0 0	7424	0.011	1.063 0.513
nos19	4128 126580	6464 896	7424	0 0	7424	0.012	1.063 0.515
nos20	4128 126580	6464 896	7424	0 0	7424	0.012	1.071 0.517
nos21	4128 126580	6464 896	7424	0 0	7424	0.012	1.044 0.513
nos22	4128 126580	6464 896	7424	0 0	7424	0.013	1.048 0.515
nos23	4128 126580	6464 896	7424	0 0	7424	0.012	1.057 0.515
nos24	4288 132576	7776 960	8192	0 0	8192	0.012	1.185 0.591
nos25	4288 132576	7776 960	8192	0 0	8192	0.013	1.183 0.601
nos26	4288 132592	7776 960	8192	0 0	8192	0.012	1.179 0.591
nos27	4288 132608	7776 960	8192	0 0	8192	0.013	1.169 0.593
nos28	4288 132608	7776 960	8192	0 0	8192	0.013	1.202 0.607
ope	324116 1643601	10200		2464 1	20400	0.036	oot
p01	30447 126201	1290 750	2865	0 0	2865	0.007	0.200 0.010
par1	189456 850574	135508		0 1	265950	0.127	oot
par2	298900 1409628	211328		0 1	414900	0.271	oot
par3	330796 1585788	233292		0 1	458008	0.325	oot
par4	184675 842981	131311		0 1	257861	0.124	oot
par5	218494 1013700	154726		0 1	303576	0.168	oot
par6	252328 1189896	178334		0 1	350222	0.223	oot
par7	262705 1259225	185171		0 1	363683	0.254	oot

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
pb1	75341 324420	2046 1023	2046	0 0	2046	0.017	0.447 0.010
pb2	76315 329030	2056 1028	2056	0 0	2056	0.018	0.467 0.010
pb3	77289 333640	2066 1033	2066	0 0	2066	0.017	0.464 0.010
pb4	72356 312285	2030 1015	2030	0 0	2030	0.017	0.431 0.010
pb5	81651 358063	2819 1732	9054	0 1	9054	0.021	44.813 44.298
pb6	140866 627295	3334 1817	6334	0 0	6334	0.034	0.937 0.022
pb7	148565 663350	3096 1548	3096	0 0	3096	0.035	0.933 0.015
pb8	149934 669936	3106 1553	3106	0 0	3106	0.035	0.936 0.015
pb9	157913 708350	3166 1583	3166	0 0	3166	0.038	1.013 0.016
pb10	147726 660773	3084 1542	3084	0 0	3084	0.035	0.949 0.015
pb11	149079 667283	3094 1547	3094	0 0	3094	0.035	0.926 0.015
pb12	137600 614115	3758 2190	9667	0 1	9667	0.034	45.794 44.915
pb13	144485 647355	3818 2225	9822	0 0	9822	0.036	0.915 0.032
pb14	145862 654003	3830 2232	9853	0 1	9853	0.036	35.034 34.078
pb15	145487 648803	3056 1528	3056	0 0	3056	0.034	0.918 0.015
pb16	145943 654778	4326 2777	15992	0 1	15992	0.038	139.630 138.649
pb17	137243 614791	4614 3072	19608	0 1	19608	0.037	206.788 205.897
pb18	145259 653473	4704 3132	19992	0 1	19992	0.038	187.382 186.457
pb19	146595 659920	4719 3142	20056	0 1	20056	0.039	227.664 226.726
pb20	247418 1134514	5425 3335	17875	0 1	17875	0.063	156.792 155.185
pb21	236292 1080258	4487 2446	8537	0 0	8537	0.059	1.529 0.031
pb22	239734 1096970	4509 2458	8579	0 0	8579	0.060	1.557 0.032
pb23	263604 1207650	4228 2114	4228	0 0	4228	0.063	1.786 0.020
pb24	216460 988171	5624 3612	21624	0 1	21624	0.058	253.843 252.407
pb25	227643 1035554	4054 2027	4054	0 0	4054	0.055	1.446 0.020
pb26	229398 1044074	4064 2032	4064	0 0	4064	0.056	1.487 0.020
pb27	231153 1052594	4074 2037	4074	0 0	4074	0.055	1.522 0.020
pb28	232908 1061114	4084 2042	4084	0 0	4084	0.057	1.526 0.020
pb29	228738 1041009	4034 2017	4034	0 0	4034	0.057	1.476 0.020

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
pos	11483525 32697150	7316362		0 0		6.369	oom
qqu	218792 1020908	92726		0 1	278257	0.520	oot
rbc	2220 148488	1072 1005	2948	0 1	2948	0.017	4.792 4.561
rpo	2384 177941	1139 1072	3149	0 1	3149	0.021	5.328 5.052
sat1	1621762 6356704	1614717		0 0	4156655	1.241	oot
sat2	1049961 4112982	1045540		0 0	2690102	0.813	oot
sat3	1295022 5074584	1289497		0 0	3318635	0.997	oot
sat4	1376707 5395114	1370802		0 0	3528140	1.058	oot
slp1	19995 66333	45292		21 1	61563	0.022	oot
slp2	22886 76038	52373		19 1	70658	0.026	oot
slp3	71356 227126	114314		11 1	220749	0.066	oot
slp4	76943 245006	123454		12 1	238035	0.072	oot
slp5	82745 263586	132950		9 1	255985	0.078	oot
smt1	27224 68879	11922 11922	35766	0 0	35766	0.006	0.205 0.115
smt2	350506 878969	119076 119076	357228	0 0	357228	0.084	2.445 1.319
smt3	360364 1076507	258493 258493	775479	0 0	775479	0.148	4.324 3.034
tot	270120 1250036	185434		0 1	370242	0.218	oot
tra1	241290 1289940	0 0	0	0 0	0	0.024	1.760 0.000
tra2	361750 1934720	0 0	0	0 0	0	0.037	2.841 0.000
tra3	482210 2579500	0 0	0	0 0	0	0.049	3.564 0.000
tra4	602670 3224280	0 0	0	0 0	0	0.061	4.532 0.000
tra5	723130 3869060	0 0	0	0 0	0	0.074	5.616 0.000
tra6	630506 3386280	0 0	0	0 0	0	0.065	4.802 0.000
tra7	945406 5079060	0 0	0	0 0	0	0.095	7.297 0.000
tra8	1260306 6771840	0 0	0	0 0	0	0.132	10.239 0.000
tra9	756832 4116966	0 0	0	0 0	0	0.081	5.987 0.000
tra10	1134832 6175026	0 0	0	0 0	0	0.121	9.010 0.000
tra11	290655 1570705	0 0	0	0 0	0	0.030	2.204 0.000
tra12	435765 2355835	0 0	0	0 0	0	0.045	3.278 0.000

Results of `cnf2aig` run on SAT Competition 2013 benchmarks (continued).

Bench-mark	Variables / Clauses	Matches in Cover	Clauses in Cover	Cyclic It. / Cardin. It.	Relaxation Variables	Matching Time	Exec. Time / SAT Time
tra13	580875 3140965	0 0	0	0 0	0	0.061	4.522 0.000
ucg	200003 1019221	65419		0 1	230505	0.138	oot
ur1	199290 1005792	65412		0 1	230484	0.136	oot
ur2	199996 1019200	65412		0 1	230484	0.140	oot
ur3	259234 1387934	84325		0 1	296983	0.201	oot
ur4	224569 1190619	73207		0 1	257725	0.174	oot
ur5	224962 1204358	73207		0 1	257725	0.173	oot
uti1	200152 1019667	65583		0 1	230937	0.134	oot
uti2	259616 1374599	84091		0 1	296205	0.192	oot
uti3	260342 1391257	84456		0 1	297296	0.196	oot
uti4	225926 1207249	73326		0 1	258002	0.168	oot
vel1	889302 14582074	24599		0 1	99254	5.126	oot
vel2	118040 8804672	8114 8114	26078	0 0	26078	1.940	15.885 0.082
vel3	118038 8780591	8112 8112	26070	0 0	26070	1.968	15.850 0.082
vel4	96177 1814189	8511 8506	35337	0 2	35359	1.176	708.960 704.867
vmp1	841 120147	0 0	0	0 0	0	0.162	0.314 0.000
vmp2	900 133080	0 0	0	0 0	0	0.184	0.353 0.000
vmp3	1024 161664	0 0	0	0 0	0	0.282	0.488 0.000
vmp4	1089 177375	0 0	0	0 0	0	0.326	0.560 0.000
vmp5	1156 194072	0 0	0	0 0	0	0.398	0.654 0.000
vmp6	1225 211785	0 0	0	0 0	0	0.464	0.753 0.000
vmp7	1296 230544	0 0	0	0 0	0	0.560	0.861 0.000
zfc	10950109 32697150	7316362		0 0		6.368	oom

Table B.2 shows the results of running `cnf2aig` on the benchmarks with the following options:

- Maximum number of X(N)OR inputs: 4
- Maximum number of (N)AND and (N)OR inputs: 10
- Do not add clauses for constraint 3 that only contain one match

See Section 7.2.1 for additional information and interpretation.

Table B.2.: Results of `cnf2aig` run with –no-single-relax.

Bench-mark	Clauses	Counter Inputs	Clauses Saved	Exec. Time	Optimized Exec. Time
005	478484	50688	3840	7.743	7.692
006	478484	50688	3840	7.67	7.516
pip1	3601247	16661	16661	5.798	5.738
pip3	4187694	22379	22379	7.745	7.724
pip5	751116	10516	10516	1.169	1.194
pip6	1332773	14790	14790	2.048	2.123
pip7	887706	11123	11080	73.544	55.379
dlx1	1598301	47469	47464	oot	977.825
pip8	2317839	19221	19221	3.686	3.666
vli1	13378625	149160	149160	54.519	52.686
vli2	13378641	149160	149160	53.553	52.623
vli3	13348117	149137	149137	53.511	52.904
vli4	13378781	149160	149160	53.336	52.394
vli5	13378010	149134	149134	53.931	52.729
vli6	13378617	149160	149160	59.797	52.884
vli7	13378624	149177	149177	53.748	52.776
aaa1	142227	14472	11946	112.244	105.582
aaa2	283903	25277	20325	347.534	305.258
aaa3	308235	26409	21363	358.856	312.359
aes1	14656	2224	0	0.168	0.169
aes2	5968	1184	0	0.03	0.033
aes3	5968	1184	0	0.032	0.028
aes4	2664	1616	0	1.587	1.596
aes5	2664	1616	0	1.587	1.594
aes6	2780	2000	0	0.099	0.099
apr1	22741	22004	13648	413.695	317.949
apr2	10827	10674	6452	112.603	75.503
apr5	29194	28992	18422	oot	442.341
arc1	87670	0	0	0.144	0.137
arc2	405350	0	0	0.716	0.684
arc3	405350	0	0	0.716	0.662
arc4	405350	0	0	0.717	0.689
arc5	405350	0	0	0.706	0.692
arc6	405350	0	0	0.708	0.674
arc7	405350	0	0	0.72	0.67
arc8	405350	0	0	0.707	0.681
b041	801488	50	0	1.489	1.429
b042	792142	96	0	1.393	1.415
b043	1262210	6	0	2.256	2.195
blo1	9511460	0	0	12.537	12.255
blo2	10974540	0	0	14.344	14.224
bun1	575617	0	0	1.057	1.082

Results of `cnf2aig` run with –no-single-relax. (continued).

Bench-mark	Clauses	Counter Inputs	Clauses Saved	Exec. Time	Optimized Exec. Time
bun2	1010625	0	0	1.936	1.91
cou	55724	34335	34335	0.174	0.083
ctl1	97667	0	0	0.18	0.176
ctl2	103844	0	0	0.178	0.187
ctl3	111990	0	0	0.214	0.206
ctl4	141683	0	0	0.269	0.248
ctl5	144538	0	0	0.263	0.256
ctl6	157735	0	0	0.3	0.286
ctl7	154048	0	0	0.278	0.275
ctl8	142785	0	0	0.253	0.258
ctl9	147308	0	0	0.256	0.262
ctl10	259243	0	0	0.512	0.492
ctl11	156259	0	0	0.275	0.274
ctl12	147244	0	0	0.254	0.257
ctl13	149131	0	0	0.26	0.268
ctl14	147245	0	0	0.28	0.265
ctl15	149132	0	0	0.255	0.261
ctl16	142693	0	0	0.262	0.256
ctl17	134760	0	0	0.24	0.262
ctl18	142697	0	0	0.252	0.257
ctl19	167062	0	0	0.3	0.305
ctl20	154654	0	0	0.28	0.279
ctl21	142789	0	0	0.247	0.258
ctl22	147312	0	0	0.254	0.259
ctl23	169583	0	0	0.308	0.303
ctl24	154655	0	0	0.272	0.283
e01	2210893	35098	34914	702.848	614.493
e02	2133873	35098	34914	696.306	604.189
e03	395383	22960	22800	253.411	229.699
e04	1301188	30668	30492	555.363	434.716
e05	295685	19646	19494	186.627	155.913
gre	146909	0	0	0.431	0.452
hit1	29903	5324	2362	15.603	15.122
hit2	30273	5154	2338	13.669	14.115
hit3	29932	5355	2381	15.341	14.298
hit4	29810	5377	2371	15.737	14.991
hit5	29319	5277	2301	15.291	14.303
hit6	31221	5377	2443	14.439	14.437
hit7	30992	5261	2383	14.051	0.196
hit8	30662	5187	2353	0.283	0.197
hit9	30374	5424	2402	16.005	15.807
hit10	30899	5267	2407	15.262	14.135
hit11	30268	5408	2428	14.972	15.632
hit12	29849	5437	2383	15.804	14.69
hit13	30142	5261	2311	0.34	13.69
hit14	30362	5296	2374	14.887	13.241
hit15	29727	5337	2319	16.031	15.076
hit16	30014	5490	2410	15.616	15.564
hit17	31013	5389	2457	16.008	14.829
hit18	30885	5299	2367	0.359	14.423
hit19	30117	5337	2341	14.833	13.768
hit20	30490	5331	2339	15.627	14.426
hit21	31022	5489	2453	15.149	15.999
hw4	2384932	2335251	2335251	11.152	3.714
hw5	1794934	1197279	1197279	6.966	2.791
hw6	183916	109077	109077	0.579	0.279
hw10	133180	83771	83770	391.087	268.277
hw12	1464739	979305	979305	5.827	2.339

Results of `cnf2aig` run with –no-single-relax. (continued).

Bench-mark	Clauses	Counter Inputs	Clauses Saved	Exec. Time	Optimized Exec. Time
hw14	8995	7065	7065	0.039	0.022
hw15	15994	12486	12486	0.066	0.026
hw16	90805	59778	59778	0.306	0.137
hw17	175093	95226	95226	0.546	0.256
hw18	98113	64515	64515	0.325	0.145
hw19	61561	39033	39033	0.201	0.091
hw20	61564	39036	39036	0.197	0.088
hw23	77737	51024	51024	0.261	0.121
kun	95854	0	0	0.182	0.186
max1	151835	88659	88659	0.481	0.226
max2	598619	349395	349395	2.104	0.92
max3	39143	23001	23001	0.123	0.059
max4	152039	89049	89049	0.464	0.229
min1	746444	547161	547161	3.145	1.149
min2	160469	106704	106704	0.544	0.24
min3	119357	80193	80193	0.403	0.178
min4	459965	307905	307905	1.67	0.667
ndh1	466486	5369	5187	15.849	14.363
ndh2	542457	5915	5733	19.491	16.972
nos1	126548	7424	3840	1.066	1.049
nos2	126548	7424	3840	1.061	1.051
nos3	126548	7424	3840	1.059	1.037
nos4	126548	7424	3840	1.062	1.053
nos5	126548	7424	3840	1.058	1.044
nos6	126548	7424	3840	1.077	1.068
nos7	126548	7424	3840	1.063	1.063
nos8	126548	7424	3840	1.067	1.036
nos9	126548	7424	3840	1.049	1.079
nos10	126548	7424	3840	1.077	1.048
nos11	126564	7424	3840	1.045	1.042
nos12	126564	7424	3840	1.06	1.055
nos13	126564	7424	3840	1.062	1.062
nos14	126564	7424	3840	1.123	1.031
nos15	126564	7424	3840	1.073	1.03
nos16	126564	7424	3840	1.076	1.04
nos17	126564	7424	3840	1.067	1.048
nos18	126580	7424	3840	1.063	1.045
nos19	126580	7424	3840	1.063	1.027
nos20	126580	7424	3840	1.071	1.056
nos21	126580	7424	3840	1.044	1.03
nos22	126580	7424	3840	1.048	1.063
nos23	126580	7424	3840	1.057	1.034
nos24	132576	8192	3840	1.185	1.164
nos25	132576	8192	3840	1.183	1.164
nos26	132592	8192	3840	1.179	1.161
nos27	132608	8192	3840	1.169	1.142
nos28	132608	8192	3840	1.202	1.176
p01	126201	2865	1785	0.2	3.792
pb1	324420	2046	0	0.447	0.467
pb2	329030	2056	0	0.467	0.468
pb3	333640	2066	0	0.464	0.468
pb4	312285	2030	0	0.431	0.449
pb5	358063	9054	6880	44.813	36.732
pb6	627295	6334	3300	0.937	21.874
pb7	663350	3096	0	0.933	0.946
pb8	669936	3106	0	0.936	1.009
pb9	708350	3166	0	1.013	1.034
pb10	660773	3084	0	0.949	0.985

Results of `cnf2aig` run with –no-single-relax. (continued).

Bench-mark	Clauses	Counter Inputs	Clauses Saved	Exec. Time	Optimized Exec. Time
pb11	667283	3094	0	0.926	0.957
pb12	614115	9667	6531	45.794	37.298
pb13	647355	9822	6636	0.915	41.231
pb14	654003	9853	6657	35.034	44.745
pb15	648803	3056	0	0.918	0.962
pb16	654778	15992	12894	139.63	109.058
pb17	614791	19608	16524	206.788	185.368
pb18	653473	19992	16848	187.382	173.217
pb19	659920	20056	16902	227.664	134.998
pb20	1134514	17875	13695	156.792	148.542
pb21	1080258	8537	4455	1.529	32.275
pb22	1096970	8579	4477	1.557	30.383
pb23	1207650	4228	0	1.786	1.762
pb24	988171	21624	17600	253.843	188.988
pb25	1035554	4054	0	1.446	1.509
pb26	1044074	4064	0	1.487	1.527
pb27	1052594	4074	0	1.522	1.549
pb28	1061114	4084	0	1.526	1.548
pb29	1041009	4034	0	1.476	1.477
rbc	148488	2948	2814	4.792	4.156
rpo	177941	3149	3015	5.328	4.41
smt1	68879	35766	35766	0.205	0.101
smt2	878969	357228	357228	2.445	1.304
smt3	1076507	775479	775479	4.324	1.671
tra1	1289940	0	0	1.76	1.794
tra2	1934720	0	0	2.841	2.679
tra3	2579500	0	0	3.564	3.703
tra4	3224280	0	0	4.532	4.502
tra5	3869060	0	0	5.616	5.451
tra6	3386280	0	0	4.802	4.751
tra7	5079060	0	0	7.297	7.28
tra8	6771840	0	0	10.239	9.776
tra9	4116966	0	0	5.987	5.847
tra10	6175026	0	0	9.01	9.004
tra11	1570705	0	0	2.204	2.178
tra12	2355835	0	0	3.278	3.283
tra13	3140965	0	0	4.522	4.445
vel2	8804672	26078	26078	15.885	15.871
vel3	8780591	26070	26070	15.85	15.719
vel4	1814189	35359	35357	708.96	409.745
vmp1	120147	0	0	0.314	0.328
vmp2	133080	0	0	0.353	0.38
vmp3	161664	0	0	0.488	0.499
vmp4	177375	0	0	0.56	0.603
vmp5	194072	0	0	0.654	0.668
vmp6	211785	0	0	0.753	0.766
vmp7	230544	0	0	0.861	0.857

Table B.3 shows the results of how many clauses are added when running `cnf2aig` on the benchmarks with the following options:

- Maximum number of X(N)OR inputs: 4
- Maximum number of (N)AND and (N)OR inputs: 10
- Do not add clauses for constraint 3 that only contain one match

See Section 7.2.2 for additional information and interpretation.

Table B.3.: Clauses produced by `cnf2aig` run on SAT Competition 2013 benchmars.

Benchmark	Clauses added	Constr. 1	Constr. 2	Constr. 3	Counter	Cyclic
005	11753152	635584(5.41)	11070720(94.19)	46848(0.40)	(0.00)	0(0.00)
006	11753152	635584(5.41)	11070720(94.19)	46848(0.40)	(0.00)	0(0.00)
pip1	0	0()	0()	0()	()	0()
pip2	134492	43(0.03)	87(0.06)	87(0.06)	134275(99.84)	0(0.00)
pip3	0	0()	0()	0()	()	0()
pip4	169216	57(0.03)	114(0.07)	114(0.07)	168931(99.83)	0(0.00)
pip5	0	0()	0()	0()	()	0()
pip6	0	0()	0()	0()	()	0()
pip7	77885	21(0.03)	43(0.06)	43(0.06)	77778(99.86)	0(0.00)
dlx1	332202	3(0.00)	5(0.00)	5(0.00)	332189(100.00)	0(0.00)
dlx2	936871	3(0.00)	5(0.00)	5(0.00)	936858(100.00)	0(0.00)
pip8	0	0()	0()	0()	()	0()
vli1	0	0()	0()	0()	()	0()
vli2	0	0()	0()	0()	()	0()
vli3	0	0()	0()	0()	()	0()
vli4	0	0()	0()	0()	()	0()
vli5	0	0()	0()	0()	()	0()
vli6	0	0()	0()	0()	()	0()
vli7	0	0()	0()	0()	()	0()
aaa1	106905	620(0.58)	2526(2.36)	2526(2.36)	101233(94.69)	0(0.00)
aaa2	188105	1349(0.72)	4952(2.63)	4952(2.63)	176852(94.02)	0(0.00)
aaa3	196221	1350(0.69)	5046(2.57)	5046(2.57)	184779(94.17)	0(0.00)
aaa4	328266	133(0.04)	228(0.07)	228(0.07)	327677(99.82)	0(0.00)
aaa5	136820	591(0.43)	708(0.52)	708(0.52)	134813(98.53)	0(0.00)
aaa6	293345	1283(0.44)	1506(0.51)	1506(0.51)	289050(98.54)	0(0.00)
acg1	2056993	47879(2.33)	25356(1.23)	25356(1.23)	1958402(95.21)	0(0.00)
acg2	2072086	47879(2.31)	25356(1.22)	25356(1.22)	1973495(95.24)	0(0.00)
acg3	2347930	53300(2.27)	28228(1.20)	28228(1.20)	2238174(95.33)	0(0.00)
acg4	2356297	53300(2.26)	28228(1.20)	28228(1.20)	2246541(95.34)	0(0.00)
aes1	9156	580(6.33)	6352(69.38)	2224(24.29)	(0.00)	20(0.22)
aes2	5656	536(9.48)	3936(69.59)	1184(20.93)	(0.00)	0(0.00)
aes3	5656	536(9.48)	3936(69.59)	1184(20.93)	(0.00)	0(0.00)
aes4	18528	712(3.84)	4944(26.68)	1616(8.72)	11256(60.75)	19(0.10)
aes5	18528	712(3.84)	4944(26.68)	1616(8.72)	11256(60.75)	19(0.10)
aes6	15760	624(3.96)	13136(83.35)	2000(12.69)	(0.00)	15(0.10)
apr1	412443	51906(12.59)	198240(48.06)	8356(2.03)	153941(37.32)	312(0.08)
apr2	236271	32150(13.61)	125258(53.01)	4222(1.79)	74641(31.59)	41(0.02)
apr4	52600217	6795993(12.92)	44524842(84.65)	1279382(2.43)	(0.00)	0(0.00)
apr5	558696	47203(8.45)	298054(53.35)	10570(1.89)	202869(36.31)	38(0.01)
apr6	5456029	444766(8.15)	3087206(56.58)	90986(1.67)	1833071(33.60)	0(0.00)

Clauses produced by `cnf2aig` run on SAT Competition 2013 benchmars (cont.).

Bench-mark	Clauses added	Constr. 1	Constr. 2	Constr. 3	Counter	Cyclic
arc1	0	0()	0()	0()	()	0()
arc2	0	0()	0()	0()	()	0()
arc3	0	0()	0()	0()	()	0()
arc4	0	0()	0()	0()	()	0()
arc5	0	0()	0()	0()	()	0()
arc6	0	0()	0()	0()	()	0()
arc7	0	0()	0()	0()	()	0()
arc8	0	0()	0()	0()	()	0()
b041	101	1(0.99)	50(49.50)	50(49.50)	(0.00)	0(0.00)
b042	193	1(0.52)	96(49.74)	96(49.74)	(0.00)	0(0.00)
b043	12	0(0.00)	6(50.00)	6(50.00)	(0.00)	0(0.00)
biv1	909717	49045(5.39)	856074(94.10)	4598(0.51)	(0.00)	97269(10.69)
biv2	933954	51242(5.49)	878104(94.02)	4608(0.49)	(0.00)	97726(10.46)
biv3	970386	50134(5.17)	915510(94.34)	4742(0.49)	(0.00)	95881(9.88)
biv4	923226	50970(5.52)	867644(93.98)	4612(0.50)	(0.00)	97265(10.54)
biv5	955857	49811(5.21)	901378(94.30)	4668(0.49)	(0.00)	96579(10.10)
biv6	941717	49769(5.28)	887340(94.23)	4608(0.49)	(0.00)	97781(10.38)
biv7	976419	48546(4.97)	886756(90.82)	4644(0.48)	36473(3.74)	56476(5.78)
biv8	932985	49533(5.31)	878840(94.20)	4612(0.49)	(0.00)	97591(10.46)
biv9	954686	47614(4.99)	866174(90.73)	4618(0.48)	36280(3.80)	96635(10.12)
blo1	0	0()	0()	0()	()	0()
blo2	0	0()	0()	0()	()	0()
bun1	0	0()	0()	0()	()	0()
bun2	0	0()	0()	0()	()	0()
cou	0	0()	0()	0()	()	0()
ctl1	0	0()	0()	0()	()	0()
ctl2	0	0()	0()	0()	()	0()
ctl3	0	0()	0()	0()	()	0()
ctl4	0	0()	0()	0()	()	0()
ctl5	0	0()	0()	0()	()	0()
ctl6	0	0()	0()	0()	()	0()
ctl7	0	0()	0()	0()	()	0()
ctl8	0	0()	0()	0()	()	0()
ctl9	0	0()	0()	0()	()	0()
ctl10	0	0()	0()	0()	()	0()
ctl11	0	0()	0()	0()	()	0()
ctl12	0	0()	0()	0()	()	0()
ctl13	0	0()	0()	0()	()	0()
ctl14	0	0()	0()	0()	()	0()
ctl15	0	0()	0()	0()	()	0()
ctl16	0	0()	0()	0()	()	0()
ctl17	0	0()	0()	0()	()	0()
ctl18	0	0()	0()	0()	()	0()
ctl19	0	0()	0()	0()	()	0()
ctl20	0	0()	0()	0()	()	0()
ctl21	0	0()	0()	0()	()	0()
ctl22	0	0()	0()	0()	()	0()
ctl23	0	0()	0()	0()	()	0()
ctl24	0	0()	0()	0()	()	0()
dat1	1579860	69000(4.37)	66000(4.18)	66000(4.18)	1378860(87.28)	0(0.00)
dat2	2000942	87400(4.37)	83600(4.18)	83600(4.18)	1746342(87.28)	0(0.00)
dat3	1537723	67160(4.37)	64240(4.18)	64240(4.18)	1342083(87.28)	0(0.00)
dat4	2280326	99360(4.36)	95040(4.17)	95040(4.17)	1990886(87.31)	0(0.00)
e01	245972	0(0.00)	184(0.07)	184(0.07)	245604(99.85)	0(0.00)
e02	245972	0(0.00)	184(0.07)	184(0.07)	245604(99.85)	0(0.00)
e03	160959	0(0.00)	160(0.10)	160(0.10)	160639(99.80)	0(0.00)
e04	214938	0(0.00)	176(0.08)	176(0.08)	214586(99.84)	0(0.00)
e05	137739	0(0.00)	152(0.11)	152(0.11)	137435(99.78)	0(0.00)

Clauses produced by `cnf2aig` run on SAT Competition 2013 benchmars (cont.).

Bench-mark	Clauses added	Constr. 1	Constr. 2	Constr. 3	Counter	Cyclic
esa	0	()	()	0()	()	0()
gri1	71913	890(1.24)	900(1.25)	900(1.25)	69223(96.26)	0(0.00)
gri2	87913	1090(1.24)	1100(1.25)	1100(1.25)	84623(96.26)	0(0.00)
gri3	103904	1290(1.24)	1300(1.25)	1300(1.25)	100014(96.26)	0(0.00)
gre	0	0()	0()	0()	()	0()
gss1	195806	46986(24.00)	143220(73.14)	5600(2.86)	(0.00)	17460(8.92)
gss2	201339	48791(24.23)	146834(72.93)	5714(2.84)	(0.00)	17335(8.61)
gss3	215365	55037(25.56)	154434(71.71)	5894(2.74)	(0.00)	17441(8.10)
gss4	216976	54516(25.13)	156480(72.12)	5980(2.76)	(0.00)	17092(7.88)
gss5	223369	56769(25.41)	160510(71.86)	6090(2.73)	(0.00)	17297(7.74)
gss6	223365	56765(25.41)	160510(71.86)	6090(2.73)	(0.00)	17024(7.62)
gss7	226171	57779(25.55)	162246(71.74)	6146(2.72)	(0.00)	16653(7.36)
gss8	229261	58925(25.70)	164128(71.59)	6208(2.71)	(0.00)	16387(7.15)
gss9	239227	63463(26.53)	169432(70.82)	6332(2.65)	(0.00)	16540(6.91)
gus1	4167566	697166(16.73)	3366620(80.78)	103780(2.49)	(0.00)	0(0.00)
gus2	4175301	698113(16.72)	3373164(80.79)	104024(2.49)	(0.00)	0(0.00)
hit1	77419	977(1.26)	36282(46.86)	2962(3.83)	37198(48.05)	42(0.05)
hit2	73000	878(1.20)	33292(45.61)	2816(3.86)	36014(49.33)	21(0.03)
hit3	68985	680(0.99)	27922(40.48)	2974(4.31)	37409(54.23)	44(0.06)
hit4	70138	775(1.10)	28782(41.04)	3006(4.29)	37575(53.57)	48(0.07)
hit5	83668	1254(1.50)	42572(50.88)	2976(3.56)	36866(44.06)	34(0.04)
hit6	77119	848(1.10)	35762(46.37)	2934(3.80)	37575(48.72)	35(0.05)
hit7	41220	864(2.10)	37478(90.92)	2878(6.98)	(0.00)	9(0.02)
hit8	48959	1075(2.20)	45050(92.02)	2834(5.79)	(0.00)	9(0.02)
hit9	80118	917(1.14)	38278(47.78)	3022(3.77)	37901(47.31)	37(0.05)
hit10	76207	800(1.05)	35748(46.91)	2860(3.75)	36799(48.29)	21(0.03)
hit11	73632	612(0.83)	32248(43.80)	2980(4.05)	37792(51.33)	42(0.06)
hit12	76659	956(1.25)	34666(45.22)	3054(3.98)	37983(49.55)	44(0.06)
hit13	94167	1174(1.25)	53286(56.59)	2950(3.13)	36757(39.03)	18(0.02)
hit14	81034	1133(1.40)	39974(49.33)	2922(3.61)	37005(45.67)	28(0.03)
hit15	82230	1100(1.34)	40826(49.65)	3018(3.67)	37286(45.34)	39(0.05)
hit16	71489	652(0.91)	29400(41.13)	3080(4.31)	38357(53.65)	32(0.04)
hit17	70793	624(0.88)	29584(41.79)	2932(4.14)	37653(53.19)	28(0.04)
hit18	86152	1212(1.41)	44988(52.22)	2932(3.40)	37020(42.97)	21(0.02)
hit19	81050	1236(1.52)	39532(48.77)	2996(3.70)	37286(46.00)	30(0.04)
hit20	79127	935(1.18)	37956(47.97)	2992(3.78)	37244(47.07)	28(0.04)
hit21	76834	684(0.89)	34764(45.25)	3036(3.95)	38350(49.91)	39(0.05)
hw1	1129857	111(0.01)	87(0.01)	87(0.01)	1129572(99.97)	0(0.00)
hw2	2198535	887(0.04)	686(0.03)	686(0.03)	2196276(99.90)	0(0.00)
hw3	1156411	421(0.04)	57(0.00)	57(0.00)	1155876(99.95)	0(0.00)
hw4	0	0()	0()	0()	()	0()
hw5	0	0()	0()	0()	()	0()
hw6	0	0()	0()	0()	()	0()
hw7	6075362	151(0.00)	6(0.00)	6(0.00)	6075199(100.00)	0(0.00)
hw8	6075362	151(0.00)	6(0.00)	6(0.00)	6075199(100.00)	0(0.00)
hw9	9268655	1(0.00)	1(0.00)	1(0.00)	9268652(100.00)	0(0.00)
hw10	586301	0(0.00)	1(0.00)	1(0.00)	586299(100.00)	0(0.00)
hw11	9630791	1(0.00)	1(0.00)	1(0.00)	9630788(100.00)	0(0.00)
hw12	0	0()	0()	0()	()	0()
hw13	642235	5(0.00)	1(0.00)	1(0.00)	642228(100.00)	0(0.00)
hw14	0	0()	0()	0()	()	0()
hw15	0	0()	0()	0()	()	0()
hw16	0	0()	0()	0()	()	0()
hw17	0	0()	0()	0()	()	0()
hw18	0	0()	0()	0()	()	0()
hw19	0	0()	0()	0()	()	0()
hw20	0	0()	0()	0()	()	0()
hw21	6281267	6(0.00)	4(0.00)	4(0.00)	6281253(100.00)	0(0.00)

Clauses produced by `cnf2aig` run on SAT Competition 2013 benchmars (cont.).

Bench-mark	Clauses added	Constr. 1	Constr. 2	Constr. 3	Counter	Cyclic
hw22	6848389	74(0.00)	4(0.00)	4(0.00)	6848307(100.00)	0(0.00)
hw23	0	0()	0()	0()	()	0()
hw24	6202961	6(0.00)	4(0.00)	4(0.00)	6202947(100.00)	0(0.00)
hw25	6846695	73(0.00)	3(0.00)	3(0.00)	6846616(100.00)	0(0.00)
ito1	6454544	3040896(47.11)	3272120(50.69)	141528(2.19)	(0.00)	949(0.01)
ito2	7396083	3235167(43.74)	3980246(53.82)	180670(2.44)	(0.00)	415(0.01)
kun	0	0()	0()	0()	()	0()
max1	0	0()	0()	0()	()	0()
max2	0	0()	0()	0()	()	0()
max3	0	0()	0()	0()	()	0()
max4	0	0()	0()	0()	()	0()
md51	48068638	37233626(77.46)	10728886(22.32)	106126(0.22)	(0.00)	1133(0.00)
md52	47913047	37080155(77.39)	10726782(22.39)	106110(0.22)	(0.00)	1117(0.00)
md53	48116573	37284641(77.49)	10725786(22.29)	106146(0.22)	(0.00)	1103(0.00)
md54	48192509	37353345(77.51)	10733050(22.27)	106114(0.22)	(0.00)	1116(0.00)
md55	49680495	38573167(77.64)	10998612(22.14)	108716(0.22)	(0.00)	1051(0.00)
md56	49522168	38416960(77.58)	10996508(22.21)	108700(0.22)	(0.00)	1079(0.00)
md57	49729342	38625094(77.67)	10995512(22.11)	108736(0.22)	(0.00)	1057(0.00)
md58	49806494	38695014(77.69)	11002776(22.09)	108704(0.22)	(0.00)	1055(0.00)
md59	50071945	38956497(77.80)	11006716(21.98)	108732(0.22)	(0.00)	1052(0.00)
min1	0	0()	0()	0()	()	0()
min2	0	0()	0()	0()	()	0()
min3	0	0()	0()	0()	()	0()
min4	0	0()	0()	0()	()	0()
ndh1	37871	0(0.00)	182(0.48)	182(0.48)	37507(99.04)	0(0.00)
ndh2	41693	0(0.00)	182(0.44)	182(0.44)	41329(99.13)	0(0.00)
nos1	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos2	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos3	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos4	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos5	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos6	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos7	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos8	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos9	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos10	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos11	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos12	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos13	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos14	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos15	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos16	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos17	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos18	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos19	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos20	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos21	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos22	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos23	1005184	18560(1.85)	983040(97.80)	3584(0.36)	(0.00)	0(0.00)
nos24	1197376	25664(2.14)	1167360(97.49)	4352(0.36)	(0.00)	0(0.00)
nos25	1197376	25664(2.14)	1167360(97.49)	4352(0.36)	(0.00)	0(0.00)
nos26	1197376	25664(2.14)	1167360(97.49)	4352(0.36)	(0.00)	0(0.00)
nos27	1197376	25664(2.14)	1167360(97.49)	4352(0.36)	(0.00)	0(0.00)
nos28	1197376	25664(2.14)	1167360(97.49)	4352(0.36)	(0.00)	0(0.00)
ope	157986	5070(3.21)	5100(3.23)	5100(3.23)	142716(90.33)	0(0.00)
p01	22360	211(0.94)	1080(4.83)	1080(4.83)	19989(89.40)	0(0.00)
par1	2144976	95794(4.47)	96144(4.48)	91494(4.27)	1861544(86.79)	0(0.00)
par2	3345712	149177(4.46)	149894(4.48)	142444(4.26)	2904197(86.80)	0(0.00)

Clauses produced by `cnf2aig` run on SAT Competition 2013 benchmars (cont.).

Bench-mark	Clauses added	Constr. 1	Constr. 2	Constr. 3	Counter	Cyclic
par3	3693348	164667(4.46)	165482(4.48)	157252(4.26)	3205947(86.80)	0(0.00)
par4	2079393	92794(4.46)	93020(4.47)	88660(4.26)	1804919(86.80)	0(0.00)
par5	2448664	109394(4.47)	109864(4.49)	104474(4.27)	2124932(86.78)	0(0.00)
par6	2823910	125856(4.46)	126400(4.48)	120200(4.26)	2451454(86.81)	0(0.00)
par7	2932379	130681(4.46)	131210(4.47)	124810(4.26)	2545678(86.81)	0(0.00)
pb1	4097	5(0.12)	2046(49.94)	2046(49.94)	(0.00)	0(0.00)
pb2	4117	5(0.12)	2056(49.94)	2056(49.94)	(0.00)	0(0.00)
pb3	4137	5(0.12)	2066(49.94)	2066(49.94)	(0.00)	0(0.00)
pb4	4065	5(0.12)	2030(49.94)	2030(49.94)	(0.00)	0(0.00)
pb5	68291	645(0.94)	2174(3.18)	2174(3.18)	63298(92.69)	0(0.00)
pb6	50632	302(0.60)	3034(5.99)	3034(5.99)	44262(87.42)	0(0.00)
pb7	6197	5(0.08)	3096(49.96)	3096(49.96)	(0.00)	0(0.00)
pb8	6217	5(0.08)	3106(49.96)	3106(49.96)	(0.00)	0(0.00)
pb9	6337	5(0.08)	3166(49.96)	3166(49.96)	(0.00)	0(0.00)
pb10	6173	5(0.08)	3084(49.96)	3084(49.96)	(0.00)	0(0.00)
pb11	6193	5(0.08)	3094(49.96)	3094(49.96)	(0.00)	0(0.00)
pb12	74488	624(0.84)	3136(4.21)	3136(4.21)	67592(90.74)	0(0.00)
pb13	75680	634(0.84)	3186(4.21)	3186(4.21)	68674(90.74)	0(0.00)
pb14	75916	636(0.84)	3196(4.21)	3196(4.21)	68888(90.74)	0(0.00)
pb15	6117	5(0.08)	3056(49.96)	3056(49.96)	(0.00)	0(0.00)
pb16	119283	1226(1.03)	3098(2.60)	3098(2.60)	111861(93.78)	0(0.00)
pb17	144873	1527(1.05)	3084(2.13)	3084(2.13)	137178(94.69)	0(0.00)
pb18	147711	1557(1.05)	3144(2.13)	3144(2.13)	139866(94.69)	0(0.00)
pb19	148181	1562(1.05)	3154(2.13)	3154(2.13)	140311(94.69)	0(0.00)
pb20	134647	1246(0.93)	4180(3.10)	4180(3.10)	125041(92.87)	0(0.00)
pb21	68256	407(0.60)	4082(5.98)	4082(5.98)	59685(87.44)	0(0.00)
pb22	68595	409(0.60)	4102(5.98)	4102(5.98)	59982(87.44)	0(0.00)
pb23	8461	5(0.06)	4228(49.97)	4228(49.97)	(0.00)	0(0.00)
pb24	160935	1600(0.99)	4024(2.50)	4024(2.50)	151287(94.01)	0(0.00)
pb25	8113	5(0.06)	4054(49.97)	4054(49.97)	(0.00)	0(0.00)
pb26	8133	5(0.06)	4064(49.97)	4064(49.97)	(0.00)	0(0.00)
pb27	8153	5(0.06)	4074(49.97)	4074(49.97)	(0.00)	0(0.00)
pb28	8173	5(0.06)	4084(49.97)	4084(49.97)	(0.00)	0(0.00)
pb29	8073	5(0.06)	4034(49.97)	4034(49.97)	(0.00)	0(0.00)
pos	1263322	286008(22.64)	943642(74.70)	33672(2.67)	(0.00)	0(0.00)
qqu	1952289	2067(0.11)	1839(0.09)	693(0.04)	1947690(99.76)	0(0.00)
rbc	20841	0(0.00)	134(0.64)	134(0.64)	20573(98.71)	0(0.00)
rpo	22245	0(0.00)	134(0.60)	134(0.60)	21977(98.80)	0(0.00)
sat1	6784444	2115318(31.18)	4097931(60.40)	571195(8.42)	(0.00)	0(0.00)
sat2	4403564	1371410(31.14)	2662034(60.45)	370120(8.41)	(0.00)	144(0.00)
sat3	5424464	1690298(31.16)	3277871(60.43)	456295(8.41)	(0.00)	0(0.00)
sat4	5764459	1796553(31.17)	3482886(60.42)	485020(8.41)	(0.00)	0(0.00)
slp1	1205440	332569(27.59)	423838(35.16)	18186(1.51)	430847(35.74)	22(0.00)
slp2	1423622	413808(29.07)	494188(34.71)	21100(1.48)	494526(34.74)	18(0.00)
slp3	3291452	991654(30.13)	723580(21.98)	31080(0.94)	1545138(46.94)	12(0.00)
slp4	3597044	1113709(30.96)	783620(21.79)	33572(0.93)	1666143(46.32)	10(0.00)
slp5	3919365	1245369(31.77)	846052(21.59)	36160(0.92)	1791784(45.72)	10(0.00)
smt1	0	0()	0()	0()	()	0()
smt2	0	0()	0()	0()	()	0()
smt3	0	0()	0()	0()	()	0()
tot	2969474	129720(4.37)	124080(4.18)	124080(4.18)	2591594(87.27)	0(0.00)
tra1	0	0()	0()	0()	()	0()
tra2	0	0()	0()	0()	()	0()
tra3	0	0()	0()	0()	()	0()
tra4	0	0()	0()	0()	()	0()
tra5	0	0()	0()	0()	()	0()
tra6	0	0()	0()	0()	()	0()
tra7	0	0()	0()	0()	()	0()

Clauses produced by `cnf2aig` run on SAT Competition 2013 benchmars (cont.).

Bench-mark	Clauses added	Constr. 1	Constr. 2	Constr. 3	Counter	Cyclic
tra8	0	0()	0()	0()	()	0()
tra9	0	0()	0()	0()	()	0()
tra10	0	0()	0()	0()	()	0()
tra11	0	0()	0()	0()	()	0()
tra12	0	0()	0()	0()	()	0()
tra13	0	0()	0()	0()	()	0()
ucg	1712030	47879(2.80)	25356(1.48)	25356(1.48)	1613439(94.24)	0(0.00)
ur1	1711886	47879(2.80)	25356(1.48)	25356(1.48)	1613295(94.24)	0(0.00)
ur2	1711886	47879(2.80)	25356(1.48)	25356(1.48)	1613295(94.24)	0(0.00)
ur3	2205127	61391(2.78)	32476(1.47)	32476(1.47)	2078784(94.27)	0(0.00)
ur4	1913717	53300(2.79)	28228(1.48)	28228(1.48)	1803961(94.26)	0(0.00)
ur5	1913717	53300(2.79)	28228(1.48)	28228(1.48)	1803961(94.26)	0(0.00)
uti1	1715115	47880(2.79)	25386(1.48)	25386(1.48)	1616463(94.25)	0(0.00)
uti2	2199758	61392(2.79)	32514(1.48)	32514(1.48)	2073338(94.25)	0(0.00)
uti3	2207403	61393(2.78)	32516(1.47)	32516(1.47)	2080978(94.27)	0(0.00)
uti4	1915741	53302(2.78)	28268(1.48)	28268(1.48)	1805903(94.27)	0(0.00)
vel1	694799	30(0.00)	43(0.01)	43(0.01)	694683(99.98)	0(0.00)
vel2	0	0()	0()	0()	()	0()
vel3	0	0()	0()	0()	()	0()
vel4	247464	0(0.00)	37(0.01)	2(0.00)	247425(99.98)	0(0.00)
vmp1	0	0()	0()	0()	()	0()
vmp2	0	0()	0()	0()	()	0()
vmp3	0	0()	0()	0()	()	0()
vmp4	0	0()	0()	0()	()	0()
vmp5	0	0()	0()	0()	()	0()
vmp6	0	0()	0()	0()	()	0()
vmp7	0	0()	0()	0()	()	0()
zfc	1263322	286008(22.64)	943642(74.70)	33672(2.67)	(0.00)	0(0.00)

Table B.4 compares the blocking of strongly connected components to the blocking of cycles.

Table B.4.: Evaluation of `cnf2aig --scc` on SAT Competition 2013 benchmarks.

Benchmark	Cyclic Itera.	Cardin. Itera.	Cyclic It. (scc)	Cardin. It. (scc)	Exec. Time	Exec. Time (scc)
005	0	0	0	0	7.572	7.743
006	0	0	0	0	7.561	7.670
pip1	0	0	0	0	5.822	5.798
pip2	0	4	0	4	oot	oot
pip3	0	0	0	0	7.730	7.745
pip4	0	6	0	6	oot	oot
pip5	0	0	0	0	1.207	1.169
pip6	0	0	0	0	2.166	2.048
pip7	0	5	0	5	73.985	73.544
dlx1	0	1	0	1	oot	oot
dlx2	0	1	0	1	oot	oot
pip8	0	0	0	0	3.791	3.686
vli1	0	0	0	0	53.760	54.519
vli2	0	0	0	0	53.677	53.553
vli3	0	0	0	0	53.474	53.511
vli4	0	0	0	0	53.427	53.336
vli5	0	0	0	0	53.877	53.931
vli6	0	0	0	0	53.518	59.797
vli7	0	0	0	0	53.312	53.748
aaa1	0	2	0	2	111.991	112.244
aaa2	0	2	0	2	349.024	347.534
aaa3	0	2	0	2	362.140	358.856
aaa4	0	1	0	1	oot	oot
aaa5	0	1	0	1	oot	oot
aaa6	0	1	0	1	oot	oot
acg1	0	1	0	1	oot	oot
acg2	0	1	0	1	oot	oot
acg3	0	1	0	1	oot	oot
acg4	0	1	0	1	oot	oot
aes1	20	0	20	0	0.087	0.168
aes2	0	0	0	0	0.040	0.030
aes3	0	0	0	0	0.031	0.032
aes4	15	1	19	2	1.704	1.587
aes5	15	1	19	2	1.699	1.587
aes6	68	2	15	0	2.747	0.099
apr1	1602	6	555	7	472.610	413.695
apr2	111	5	134	5	101.473	112.603
apr4	0	0	0	0	oot	oot
apr5	1585	4	1775	9	oot	oot
apr6	0	0	0	0	oot	oot
arc1	0	0	0	0	0.144	0.144
arc2	0	0	0	0	0.668	0.716
arc3	0	0	0	0	0.657	0.716
arc4	0	0	0	0	0.704	0.717
arc5	0	0	0	0	0.662	0.706
arc6	0	0	0	0	0.677	0.708
arc7	0	0	0	0	0.664	0.720
arc8	0	0	0	0	0.669	0.707
b041	0	0	0	0	1.439	1.489
b042	0	0	0	0	1.440	1.393
b043	0	0	0	0	2.153	2.256
biv1	66572	0	44571	1	oot	oot
biv2	68744	0	42712	1	oot	oot

Bench-mark	Cyclic Itera.	Cardin. Itera.	Cyclic It. (scc)	Cardin. It. (scc)	Exec. Time	Exec. Time (scc)
biv3	67344	0	53697	1	oot	oot
biv4	66184	0	51020	1	oot	oot
biv5	68228	0	59114	1	oot	oot
biv6	68739	0	48036	1	oot	oot
biv7	69066	0	59569	1	oot	oot
biv8	64374	0	48849	1	oot	oot
biv9	70339	0	53248	1	oot	oot
blo1	0	0	0	0	12.139	12.537
blo2	0	0	0	0	14.179	14.344
bun1	0	0	0	0	1.067	1.057
bun2	0	0	0	0	1.928	1.936
cou	0	0	0	0	0.174	0.174
ctl1	0	0	0	0	0.178	0.180
ctl2	0	0	0	0	0.181	0.178
ctl3	0	0	0	0	0.218	0.214
ctl4	0	0	0	0	0.250	0.269
ctl5	0	0	0	0	0.256	0.263
ctl6	0	0	0	0	0.304	0.300
ctl7	0	0	0	0	0.275	0.278
ctl8	0	0	0	0	0.252	0.253
ctl9	0	0	0	0	0.260	0.256
ctl10	0	0	0	0	0.536	0.512
ctl11	0	0	0	0	0.293	0.275
ctl12	0	0	0	0	0.259	0.254
ctl13	0	0	0	0	0.261	0.260
ctl14	0	0	0	0	0.256	0.280
ctl15	0	0	0	0	0.259	0.255
ctl16	0	0	0	0	0.249	0.262
ctl17	0	0	0	0	0.261	0.240
ctl18	0	0	0	0	0.283	0.252
ctl19	0	0	0	0	0.304	0.300
ctl20	0	0	0	0	0.291	0.280
ctl21	0	0	0	0	0.258	0.247
ctl22	0	0	0	0	0.265	0.254
ctl23	0	0	0	0	0.328	0.308
ctl24	0	0	0	0	0.275	0.272
dat1	0	1	0	1	oot	oot
dat2	0	1	0	1	oot	oot
dat3	0	1	0	1	oot	oot
dat4	0	1	0	1	oot	oot
e01	0	1	0	1	697.253	702.848
e02	0	1	0	1	693.649	696.306
e03	0	1	0	1	252.503	253.411
e04	0	1	0	1	542.221	555.363
e05	0	1	0	1	188.550	186.627
esa	0	0	0	0	oom	oom
gri1	438	10	1046	2	67.195	oot
gri2	539	9	722	1	80.133	oot
gri3	628	14	753	1	116.440	oot
gre	0	0	0	0	0.443	0.431
gss1	4454	0	6259	0	oot	oot
gss2	4405	0	6162	0	oot	oot
gss3	4429	0	6166	0	oot	oot
gss4	4453	0	6069	0	oot	oot
gss5	4379	0	6087	0	oot	oot
gss6	4299	0	6132	0	oot	oot
gss7	4400	0	6086	0	oot	oot
gss8	4333	0	6001	0	oot	oot
gss9	4321	0	5977	0	oot	oot
gus1	0	0	0	0	oot	oot

Bench-mark	Cyclic Itera.	Cardin. Itera.	Cyclic It. (scc)	Cardin. It. (scc)	Exec. Time	Exec. Time (scc)
gus2	0	0	0	0	oot	oot
hit1	29	2	32	1	14.072	15.603
hit2	26	3	21	1	13.108	13.669
hit3	50	3	36	1	12.743	15.341
hit4	61	1	59	1	12.986	15.737
hit5	50	3	28	1	14.414	15.291
hit6	36	2	33	1	15.541	14.439
hit7	20	1	19	1	15.130	14.051
hit8	15	1	16	0	13.864	0.283
hit9	54	1	39	1	16.151	16.005
hit10	35	1	20	1	14.501	15.262
hit11	35	4	32	1	15.450	14.972
hit12	47	3	42	1	15.492	15.804
hit13	14	0	21	0	0.255	0.340
hit14	30	1	24	1	14.509	14.887
hit15	48	2	39	1	13.684	16.031
hit16	59	3	49	1	14.722	15.616
hit17	43	3	32	1	14.780	16.008
hit18	18	1	21	0	14.509	0.359
hit19	30	1	28	1	14.545	14.833
hit20	23	1	33	1	13.628	15.627
hit21	32	1	27	1	13.301	15.149
hw1	0	1	0	1	oot	oot
hw2	0	1	0	1	oot	oot
hw3	0	1	0	1	oot	oot
hw4	0	0	0	0	11.123	11.152
hw5	0	0	0	0	6.908	6.966
hw6	0	0	0	0	0.590	0.579
hw7	0	1	0	1	oot	oot
hw8	0	1	0	1	oot	oot
hw9	0	1	0	1	oot	oot
hw10	0	1	0	1	390.673	391.087
hw11	0	1	0	1	oot	oot
hw12	0	0	0	0	5.749	5.827
hw13	0	1	0	1	oot	oot
hw14	0	0	0	0	0.039	0.039
hw15	0	0	0	0	0.064	0.066
hw16	0	0	0	0	0.307	0.306
hw17	0	0	0	0	0.537	0.546
hw18	0	0	0	0	0.330	0.325
hw19	0	0	0	0	0.200	0.201
hw20	0	0	0	0	0.202	0.197
hw21	0	1	0	1	oot	oot
hw22	0	1	0	1	oot	oot
hw23	0	0	0	0	0.271	0.261
hw24	0	1	0	1	oot	oot
hw25	0	1	0	1	oot	oot
ito1	101	0	326	0	oot	oot
ito2	0	0	0	0	oot	oot
kun	0	0	0	0	0.185	0.182
max1	0	0	0	0	0.474	0.481
max2	0	0	0	0	2.101	2.104
max3	0	0	0	0	0.121	0.123
max4	0	0	0	0	0.482	0.464
md51	373	0	367	0	oot	oot
md52	414	0	404	0	oot	oot
md53	384	0	389	0	oot	oot
md54	399	0	413	0	oot	oot
md55	15	0	359	0	oot	oot
md56	0	0	355	0	oot	oot

Bench-mark	Cyclic Itera.	Cardin. Itera.	Cyclic It. (scc)	Cardin. It. (scc)	Exec. Time	Exec. Time (scc)
md57	341	0	360	0	oot	oot
md58	376	0	341	0	oot	oot
md59	330	0	352	0	oot	oot
min1	0	0	0	0	3.042	3.145
min2	0	0	0	0	0.547	0.544
min3	0	0	0	0	0.395	0.403
min4	0	0	0	0	1.735	1.670
ndh1	0	1	0	1	15.804	15.849
ndh2	0	1	0	1	19.390	19.491
nos1	0	0	0	0	1.066	1.066
nos2	0	0	0	0	1.061	1.061
nos3	0	0	0	0	1.059	1.059
nos4	0	0	0	0	1.047	1.062
nos5	0	0	0	0	1.065	1.058
nos6	0	0	0	0	1.070	1.077
nos7	0	0	0	0	1.047	1.063
nos8	0	0	0	0	1.071	1.067
nos9	0	0	0	0	1.061	1.049
nos10	0	0	0	0	1.068	1.077
nos11	0	0	0	0	1.054	1.045
nos12	0	0	0	0	1.076	1.060
nos13	0	0	0	0	1.072	1.062
nos14	0	0	0	0	1.054	1.123
nos15	0	0	0	0	1.053	1.073
nos16	0	0	0	0	1.065	1.076
nos17	0	0	0	0	1.074	1.067
nos18	0	0	0	0	1.061	1.063
nos19	0	0	0	0	1.052	1.063
nos20	0	0	0	0	1.039	1.071
nos21	0	0	0	0	1.046	1.044
nos22	0	0	0	0	1.058	1.048
nos23	0	0	0	0	1.050	1.057
nos24	0	0	0	0	1.157	1.185
nos25	0	0	0	0	1.163	1.183
nos26	0	0	0	0	1.192	1.179
nos27	0	0	0	0	1.171	1.169
nos28	0	0	0	0	1.187	1.202
ope	3	1	2464	1	oot	oot
p01	0	0	0	0	0.192	0.200
par1	0	1	0	1	oot	oot
par2	0	1	0	1	oot	oot
par3	0	1	0	1	oot	oot
par4	0	1	0	1	oot	oot
par5	0	1	0	1	oot	oot
par6	0	1	0	1	oot	oot
par7	0	1	0	1	oot	oot
pb1	0	0	0	0	0.481	0.447
pb2	0	0	0	0	0.483	0.467
pb3	0	0	0	0	0.498	0.464
pb4	0	0	0	0	0.441	0.431
pb5	0	1	0	1	44.924	44.813
pb6	0	0	0	0	0.898	0.937
pb7	0	0	0	0	0.956	0.933
pb8	0	0	0	0	0.971	0.936
pb9	0	0	0	0	1.003	1.013
pb10	0	0	0	0	0.960	0.949
pb11	0	0	0	0	0.954	0.926
pb12	0	1	0	1	45.461	45.794
pb13	0	0	0	0	0.967	0.915
pb14	0	1	0	1	35.846	35.034

Bench-mark	Cyclic Itera.	Cardin. Itera.	Cyclic It. (scc)	Cardin. It. (scc)	Exec. Time	Exec. Time (scc)
pb15	0	0	0	0	0.928	0.918
pb16	0	1	0	1	141.262	139.630
pb17	0	1	0	1	208.646	206.788
pb18	0	1	0	1	189.678	187.382
pb19	0	1	0	1	229.122	227.664
pb20	0	1	0	1	157.412	156.792
pb21	0	0	0	0	1.639	1.529
pb22	0	0	0	0	1.650	1.557
pb23	0	0	0	0	1.774	1.786
pb24	0	1	0	1	249.821	253.843
pb25	0	0	0	0	1.502	1.446
pb26	0	0	0	0	1.581	1.487
pb27	0	0	0	0	1.602	1.522
pb28	0	0	0	0	1.538	1.526
pb29	0	0	0	0	1.504	1.476
pos	0	0	0	0	oom	oom
qqu	0	1	0	1	oot	oot
rbc	0	1	0	1	4.732	4.792
rpo	0	1	0	1	5.291	5.328
sat1	0	0	0	0	oot	oot
sat2	0	0	0	0	oot	oot
sat3	0	0	0	0	oot	oot
sat4	0	0	0	0	oot	oot
slp1	1	1	21	1	oot	oot
slp2	1	1	19	1	oot	oot
slp3	0	1	11	1	oot	oot
slp4	0	1	12	1	oot	oot
slp5	0	1	9	1	oot	oot
smt1	0	0	0	0	0.202	0.205
smt2	0	0	0	0	2.428	2.445
smt3	0	0	0	0	4.277	4.324
tot	0	1	0	1	oot	oot
tra1	0	0	0	0	1.803	1.760
tra2	0	0	0	0	2.754	2.841
tra3	0	0	0	0	3.661	3.564
tra4	0	0	0	0	4.541	4.532
tra5	0	0	0	0	5.512	5.616
tra6	0	0	0	0	4.773	4.802
tra7	0	0	0	0	7.173	7.297
tra8	0	0	0	0	9.898	10.239
tra9	0	0	0	0	5.833	5.987
tra10	0	0	0	0	8.860	9.010
tra11	0	0	0	0	2.201	2.204
tra12	0	0	0	0	3.288	3.278
tra13	0	0	0	0	4.547	4.522
ucg	0	1	0	1	oot	oot
ur1	0	1	0	1	oot	oot
ur2	0	1	0	1	oot	oot
ur3	0	1	0	1	oot	oot
ur4	0	1	0	1	oot	oot
ur5	0	1	0	1	oot	oot
uti1	0	1	0	1	oot	oot
uti2	0	1	0	1	oot	oot
uti3	0	1	0	1	oot	oot
uti4	0	1	0	1	oot	oot
vel1	0	1	0	1	oot	oot
vel2	0	0	0	0	15.824	15.885
vel3	0	0	0	0	15.801	15.850
vel4	0	2	0	2	709.139	708.960
vmp1	0	0	0	0	0.315	0.314

Bench-mark	Cyclic Itera.	Cardin. Itera.	Cyclic It. (scc)	Cardin. It. (scc)	Exec. Time	Exec. Time (scc)
vmp2	0	0	0	0	0.380	0.353
vmp3	0	0	0	0	0.497	0.488
vmp4	0	0	0	0	0.569	0.560
vmp5	0	0	0	0	0.672	0.654
vmp6	0	0	0	0	0.765	0.753
vmp7	0	0	0	0	0.855	0.861
zfc	0	0	0	0	oom	oom

Listings

List of Figures

List of Tables

Bibliography

[BBHJ] Adrian Balint, Anton Belov, M Heule, and Matti Järvisalo. SAT competition 2013. *satcompetition.org/2013*.

[Bie] Armin Biere. The AIGER and-inverter graph (AIG) format. *Available at fmv.jku.at/aiger*.

[Bie13] Armin Biere. Lingeling, plingeling and treengeling entering the SAT competition 2013. *Proceedings of SAT Competition*, pages 51–52, 2013.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 44–57. Springer, 2010.

[Che10] Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation*, 2010.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 252–265. Springer, 2006.

[FM07] Zhaohui Fu and S. Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pages 37–42, Jan 2007.

[GB13] Alexandra Goultiaeva and Fahiem Bacchus. Recovering and utilizing partial duality in QBF. In *Theory and Applications of Satisfiability Testing–SAT 2013*, pages 83–99. Springer, 2013.

[KGP01] Andreas Kuehlmann, Malay K Ganai, and Viresh Paruthi. Circuit-based boolean reasoning. In *Design Automation Conference, 2001.*

Proceedings, pages 232–237. IEEE, 2001.

[KPKG02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(12):1377–1394, 2002.

[Li00] Chu Min Li. Integrating equivalency reasoning into davis-putnam procedure. *AAAI/IAAI*, 2000:291–296, 2000.

[McM02] Ken L McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification*, pages 250–264. Springer, 2002.

[MP75] David E Muller and Franco P Preparata. Bounds to complexities of networks for sorting and for switching. *Journal of the ACM (JACM)*, 22(2):195–201, 1975.

[MSL92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 459–465. AAAI Press, 1992.

[OGMS02] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from CNF formulas. In *Principles and Practice of Constraint Programming-CP 2002*, pages 185–199. Springer, 2002.

[PG86] David A Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[RM04] Jarrod A. Roy and Igor L. Markov. Restoring circuit structure from SAT instances. In *International Workshop on Logic Synthesis*, 2004.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming-CP 2005*, pages 827–831. Springer, 2005.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union

algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[Tse83] G.S. Tseitin. On the complexity of derivation in propositional calculus. In JörgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.