Exploiting Modern #SAT-Solving Techniques to Generate Implicants

Christian Muise
Department of Computer Science, University of Toronto, Canada

February 10, 2009

Abstract

An implicant of a propositional boolean formula is a conjunction of propositions that entails a the formula. Implicants play an important role for many tasks in AI such as knowledge compilation, circuit minimization, and diagnosis. There are many classes of implicants that are of interest including prime, irredundant, and orthogonal implicants. In general, implicants are hard to compute because the size of both the intermediate and/or final results can be exponentially large. In this work we focus on how advances in modern #SAT solvers can be exploited to effectively compute a complete set of *orthogonal* implicants. We develop an orthogonal implicant compiler and demonstrate its potential through experimental evaluation. Preliminary results indicate that this approach is effective at generating the orthogonal implicants of a propositional theory represented in Conjunctive Normal Form.

1 Introduction

Knowledge compilation plays a crucial role in the field of automated reasoning [7, 9, 19, 29, 30]. When propositional theories are compiled into certain syntactic representations of a particular language, some intractable queries become polynomial in the size of the representation. Many target languages have been studied and some of the most common are the family of various forms of *implicants*. *Prime* implicants are popular for circuit minimization problems [6], while *irredundant* implicants are of particular interest since they are minimal representations of a propositional theory in implicant form [11]. *Orthogonal* implicants provide another compelling language that is efficient for enumerating all models of the theory. Regardless of type, the number of implicants required to represent a propositional theory may be exponential in the worst case. Additionally, most algorithms for computing the various forms of implicants may require exponential space, even if the final result is polynomial in size [22].

We propose an algorithm, ODNF-Search, that compiles a propositional theory from Conjunctive Normal Form (CNF)[1] to a set of orthogonal implicants, also referred to as *Orthogonal Disjunctive Normal Form* (ODNF). ODNF-Search is based on a well-known method by Davis, Putnam, Logemann, and Loveland (DPLL) [17]. If run exhaustively, DPLL can be used to calculate the number of models a propositional theory has – a problem referred to as #SAT [28]. Many important problems in computer

[1]CNF has become somewhat of a standard for automated reasoning systems since many forms of input can be easily transformed to CNF.

science can be encoded as SAT problems. As such, there has been tremendous effort to build efficient SAT and #SAT solvers, the fastest of which are based on DPLL. By basing our approach on DPLL, an algorithm with a long history of theoretic and implementational improvements, we are able to leverage many advances in state-of-the-art solvers. Such techniques include conflict analysis [32], non-chronological backtracking [31], pre-processing [18], implicit binary constraint propagation [34], and component caching [28].

The main contribution of our work is a CNF-to-ODNF compiler, c2o, that builds upon a state-of-the-art #SAT solver, sharpSAT [34]. The extended solver not only generates the number of solutions, but additionally a set of orthogonal implicants in ODNF that represents the theory. From a knowledge compilation standpoint, the language of orthogonal implicants allows us to answer questions in polynomial time (with respect to the size of the representation) about consistency, validity, clausal entailment, and solution distribution, among others [13]. Such information has proven useful in areas such as belief revision [12] and reliability theory [5]. Approaches with similar goals to the work presented here include *primeii* [3] and *c2d* [8], and we compare the approaches experimentally in Section 4.2 and algorithmically in Section 5.1. We experimentally validated our work and found that it shows a great deal of promise for the task of CNF-to-ODNF conversion.

Our modifications to the sharpSAT solver were done in the most general way to enable easy exploitation of future advances in the underlying #SAT-solving technology, and also to support the generation of related target knowledge compilation languages. While our approach shows a significant improvement over primeii, it is still outperformed in general by the c2d solver by Darwiche et al. [8]. In the future we hope to incorporate an efficient means of storing the partially constructed ODNF representation which could potentially speed up our solver by many orders of magnitude.

The remainder of the paper is organized as follows. Section 2 introduces the background material and discusses previous literature. In Section 3 we present our new method for generating the ODNF of a theory. Section 4 describes the implementational details of c2o and gives experimental results. Finally we provide conclusions and discussion of possible future work in Section 5.

2 Background

If we have an efficient means to reason with the set of solutions to a propositional theory, a number of AI tasks can be solved in polynomial time with respect to the size of our representation. This section provides the theoretical background needed to describe the ODNF representation and reviews related work in this area of research.

2.1 Basic Definitions

Following [27], we have the following definitions:

Definition 1 (Boolean Variable, Literal) *A* ***boolean variable*** *is a variable that can take on a value of either true or false. We will use the letters x and v (with subscripts) to denote variables. A* ***literal*** *is either a boolean variable (a positive literal), or a negated boolean variable (a negative literal). We will use the letter l (with subscripts) to denote literals.*

Definition 2 (CNF, DNF)
Conjunctive Normal Form (CNF) *is a conjunction of disjunctions of literals. Every propositional theory can be represented in conjunctive normal form. We will refer to a disjunction of literals as a* ***clause****.*
Disjunctive Normal Form (DNF) *is a disjunction of conjunctions of literals. Every propositional theory can be represented in disjunctive normal form. We will refer to a conjunction of literals as a* ***term****.*

Definition 3 (SAT) ***Satisfiability (SAT)*** *is the problem of deciding whether or not a propositional theory is satisfiable. A propositional theory is satisfiable if there exists an assignment to all of the variables such that the theory is satisfied.*

For a theory Σ with n variables, written in CNF, an assignment $X \in \{0,1\}^n$ to the variables satisfies the theory if it makes at least one literal true in every clause $c \in CNF(\Sigma)$. For a theory Σ written in DNF, an assignment $X \in \{0,1\}^n$ to the variables satisfies the theory if it makes all of the literals true in at least one term $t \in DNF(\Sigma)$.

The task of finding a satisfying assignment for a theory in DNF is trivial, but compiling a theory from CNF to DNF is, at worst, an exponential process generally referred to as DUALIZATION [23]. When we consider the task of finding the number of satisfying assignments for a theory represented in DNF, the task becomes complicated because multiple terms may represent the same satisfying assignment. The process of converting an arbitrary DNF into a form that can be used to count the number of assignments is generally referred to as ORTHOGONALIZATION [5] and can be exponential in the input size as well.

Definition 4 (Model, Partial Assignment) *A* ***model*** *of a propositional theory is a satisfying assignment. It consists of a complete assignment $X \in \{0,1\}^n$ to the variables such that the theory is satisfied. A* ***partial assignment*** *is an assignment to a subset of the variables.*

Sometimes we will say a partial assignment satisfies a theory Σ. By this we mean that the variables which are not assigned can be arbitrarily set to either *true* or *false*, and the resulting full assignment of the variables will be a model of Σ.

Definition 5 (#SAT) ***#SAT*** *refers to the problem of calculating the total number of models of a propositional theory.*

Definition 6 (Entailment) *A propositional theory Σ is said to* ***entail*** *another propositional theory Σ' if every model of Σ is also a model of Σ', written as $\Sigma \models \Sigma'$.*

Definition 7 (Implicate, Implicant) *A disjunction of literals c (or clause) is an* ***implicate*** *of Σ iff $\Sigma \models c$. A conjunction of literals t is an* ***implicant*** *of Σ iff $t \models \Sigma$. When convenient, and the distinction between implicate and implicant is not ambiguous, we will refer to the literals in an implicate or implicant as a set (rather than a disjunction or conjunction respectively). An implicate (implicant) is said to be* ***prime*** *if no proper subset of the implicate (implicant) is also an implicate (implicant) of the theory.*

Every clause in the CNF representation of a theory Σ is an implicate of Σ, and similarly, every term in the DNF representation of a theory Σ is an implicant of Σ.

For a CNF representation of a theory, every implicant must **cover** the entire set of clauses. An implicant covers a clause if it contains a literal that is also part of the clause. This requirement exists because if an implicant φ did not cover a clause c in the CNF representation of a theory Σ, then a particular setting of variables would exist such that all of the literals in c could be falsified and φ would not be violated. This implies $\varphi \not\models \Sigma$, and hence φ cannot be an implicant of the theory.

Definition 8 (ODNF) *A DNF representation of a theory Σ is said to be* ***orthogonal*** *if every pair of terms in the DNF is mutually inconsistent. In other words, every pair of terms disagrees on at least one literal. We refer to a DNF with this property as Orthogonal Disjunctive Normal Form, or* ***ODNF***.[2]

Here we provide an example of a propositional theory Σ in CNF:

$$CNF(\Sigma) = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$$

The following is one of many possible DNF representations of the theory Σ:

$$\begin{aligned} DNF(\Sigma) = {} & (x_2 \wedge x_3) \vee (x_3 \wedge \neg x_4) \vee (x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge \neg x_3 \wedge x_4) \vee \\ & (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_4) \vee (\neg x_2 \wedge \neg x_3 \wedge x_4) \end{aligned}$$

Finally, we provide an ODNF representation of Σ. Note how every pair of implicants disagrees on at least one literal:

$$\begin{aligned} ODNF(\Sigma) \quad = \quad & (x_2 \wedge x_3) \vee (\neg x_2 \wedge x_3 \wedge \neg x_4) \vee \\ & (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_3 \wedge x_4) \end{aligned}$$

2.2 Knowledge Compilation

Implicants were first identified over half a century ago [24]. Prime implicants, in particular, have been heavily studied because of their ability to concisely represent a propositional theory; a property that has made them ideal for tasks such as circuit minimization [6]. Historically, prime implicants have received a large amount of attention, and a number of algorithms have been proposed to compute prime implicants from an enumeration of all solutions [21, 20, 24]. Similar work has been done to compile the complete set of prime implicants from an input of CNF [4, 14, 33] – a process referred to as DUALIZATION [23].

Orthogonal implicants have received less attention, but still provide enticing computational properties. A set of orthogonal implicants, or an ODNF representation, is closely related to the deterministic decomposable negation normal form (d-DNNF) as described in [13] which has an important location in the knowledge compilation map [9]. ODNF is a restricted subset of both d-DNNF and DNF – two languages that are not from the same lineage in the knowledge compilation map. A subset of the knowledge compilation map with ODNF included is shown in Figure 1.

In the knowledge compilation map, computational power is inherited from the top down – in Figure 1 the text outside of the boxes indicates problems that are polynomial for the given language that are not polynomial for any of its ancestors. The problems of interest are **Co**nsistency, **C**lausal **E**ntailment, **M**odel **E**numeration, **Va**lidity, valid **Imp**licant, Model **Coun**ting, **Eq**uality, and **S**entential **E**ntailment. The languages of interest are as follows:

[2]Since every term in ODNF is an implicant, and furthermore every pair of terms in ODNF are mutually inconsistent, we use the terms *orthogonal implicants* that describe a propositional theory and an *ODNF* representation of a theory to mean the same thing.

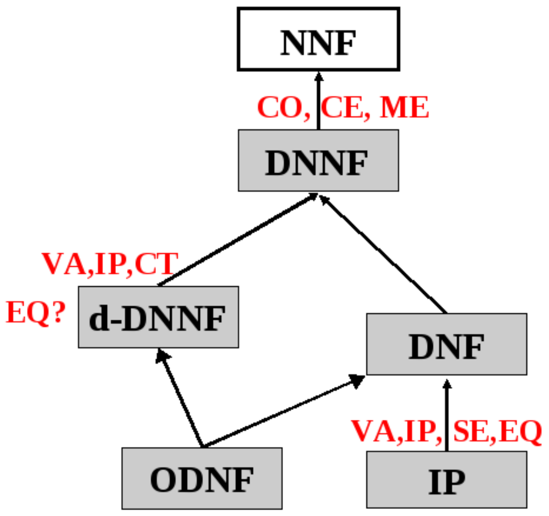

Figure 1: Knowledge Compilation Map

- Negation Normal Form (NNF): The family of boolean formulas that are built from the operators $\vee$, $\wedge$, and $\neg$, with the added restriction that all $\neg$ operators are at the literal level.
- Decomposable Negation Normal Form (DNNF): The family of boolean formulas that are NNF, but additionally have the property that the formula operands of $\wedge$ do not share variables.
- Disjunctive Decomposable Negation Normal Form (d-DNNF): The family of boolean formulas that are DNNF, but additionally have the property that the formula operands of $\vee$ are inconsistent.
- Disjunctive Normal Form (DNF): As described in Section 2.1.
- Orthogonal Disjunctive Normal Form (ODNF): As described in Section 2.1.
- Prime Implicants (IP): The complete set of prime implicants for a theory (as described in Section 2.1).

Since any two terms in ODNF are orthogonal, and any single term in ODNF does not contain the same variable twice, ODNF is actually a restricted form of d-DNNF. Additionally, since ODNF is a restricted form of DNF, it has the property of being *flat* – the NNF tree of $\vee$ and $\wedge$ operators for an ODNF representation has a height of precisely two [13].

Being a restricted version of d-DNNF, ODNF shares in all of the computational advantages that d-DNNF possesses and may contain further computational properties since the language is flat. The added restriction, however, means that ODNF is a less parsimonious language than d-DNNF.

3 Using DPLL for Computing ODNF

In this section, we turn to the problem of computing an ODNF representation from a CNF representation by exploiting state-of-the-art SAT-solving technology based on DPLL. More precisely, we exploit #SAT-solving technology – extensions of DPLL to perform model counting – to generate our ODNF representation. The motivation for using #SAT-solving techniques follows from the trace of an exhaustive DPLL used for solving #SAT, which represents a list of orthogonal implicants.

We provide the necessary background on #SAT-solving techniques in Section 3.1 and describe our algorithm in Section 3.2.

3.1 Introduction to #SAT

When a theory is represented in CNF, one of the most well-known algorithms to solve SAT is DPLL. Algorithm 1 is the classic DPLL algorithm that includes unit propagation.

Algorithm 1 DPLL(CNF formula Σ): returns $true$ if Σ is satisfiable, and otherwise returns $false$.

```
1: $\Sigma := UnitProp(\Sigma)$
2: **if** $\Sigma$ does not contain any clauses **then**
3:     **return** $true$
4: **else if** $\Sigma$ contains an empty clause **then**
5:     **return** $false$
6: **end if**
7: $v :=$ choose-variable($\Sigma$)
8: **return** DPLL($\Sigma \cup v$) $\vee$ DPLL($\Sigma \cup \neg v$)
```

UnitProp(Σ) refers to unit propagation, an approach used to simplify the syntactic CNF representation in such a way that all models are preserved. The premise of unit propagation follows from the observation that if there is a unit clause in CNF, then all of the models must satisfy that single literal. We present a slightly modified version of unit propagation in Algorithm 2: $UnitProp'$. It has been augmented to return the set of literals that are propagated. While this information is not essential for solving SAT (or #SAT), it will be important for the task of finding the ODNF representation.

Moving from the problem of solving SAT to #SAT requires only a slight modification of the DPLL procedure. Conceptually, we do not stop once a satisfying assignment is found, but continue to exhaustively explore the search space. We will refer to this slightly modified version of DPLL as #DPLL, presented in Algorithm 3. The number of solutions represented by a partial assignment that fully satisfies the CNF is 2^m, where m is the number of unassigned variables [1].

There have been a number of improvements made to the DPLL and #DPLL algorithms, and in particular we take advantage of five of the most important which can be found in the #SAT solver, sharpSAT [34]. For clarity, these techniques to add extra functionality do not appear in Algorithm 3.

Pre-Processing

Before the search begins, a pre-processing step is used to quickly check if any variable setting is entailed by the theory. This is done by testing the effect of setting a variable

Algorithm 2 $UnitProp'$(CNF formula Σ): returns a modified version of Σ and the variables R that have been set.

```
1: $Q :=$ queue of all unit clauses in $\Sigma$
2: $R := \{\}$
3: while $Q$ is not empty do
4:   $uc :=$ pop the next unit clause from $Q$
5:   $l := literal(uc)$
6:   // Remove clauses that are satisfied by the literal
7:   for all $c \in \Sigma$ s.t. $l \in c$ do
8:     remove $c$ from $\Sigma$
9:   end for
10:  // Remove the opposite literal since it cannot satisfy a clause
11:  for all $c \in \Sigma$ s.t. $\neg l \in c$ do
12:    remove $\neg l$ from $c$
13:    if $c$ is empty then
14:      return $UNSAT, \{\}$
15:    end if
16:    if $c$ is now a unit clause then
17:      push $c$ on to $Q$
18:    end if
19:  end for
20:  // Record the propagated literal.
21:  $R := R \cup l$
22: end while
23: return $\Sigma, R$
```

Algorithm 3 #DPLL(CNF formula Σ): returns the number of models of Σ.

```
1: $\Sigma := UnitProp(\Sigma)$
2: if $\Sigma$ does not contain any clauses then
3:   $m := \#$ of unassigned variables
4:   return $2^m$
5: else if $\Sigma$ contains an empty clause then
6:   return 0
7: end if
8: $v :=$ choose-variable($\Sigma$)
9: return #DPLL($\Sigma \cup v$) + #DPLL($\Sigma \cup \neg v$)
```

to $true$ or $false$ and then applying unit propagation. If the resulting theory contains an empty clause (ie. is unsatisfiable), then we can conclude the opposite setting is entailed. This test is done for every variable before running the #DPLL procedure.

Additionally any unit propagation that can be achieved is carried out before the solving begins.

Conflict Analysis

Once a conflict is found in the search (ie. the partial assignment causes a clause to become empty), additional analysis is used to determine the precise cause for the conflict [32]. This is achieved by considering which settings in the partial assignment have led to the conflict. A number of different approaches to conflict analysis exist [2], but all produce a *conflict clause* – a clause added to the theory which, with the help of unit propagation, helps prevent future search in the same unsatisfiable area.

Non-Chronological Backtracking

Once a conflict is discovered, and a clause learned through conflict analysis, we can safely backtrack to the most recent decision level that unsets a literal in the conflict clause – a process known as non-chronological backtracking [31]. Normal backtracking would only backtrack to the latest decision level, but depending on the conflict analysis we may be able to safely backtrack further.

Implicit Binary Constraint Propagation

Similar to the pre-processing step described earlier, Implicit Binary Constraint Propagation (IBCP) is a method of testing whether or not certain variable settings lead to an unsatisfiable theory [34]. IBCP is employed during the search on a subset of the variables which have yet to be assigned. The determination of which variables should be included in the subset is a matter of ongoing research [16], but once tested if any are found to cause the theory to be unsatisfiable the opposite setting is added, as if a unit clause forcing the variable in that direction had already existed.

This form of inference is equivalent to enforcing Singleton Arc Consistency [26] on a subset of the variables which are considered important.

Component Caching

Arguably one of the most important contributions to #SAT-solving is component caching [28]. During the execution of #DPLL, when the theory represented in CNF can be partitioned into disjoint sets of clauses, such that no two sets share a variable, each set of clauses can be considered independently and the solutions combined (we refer to a disjoint set of clauses as a *component*). Additionally, certain components may appear more than once during the solving process so a component may be cached along with the number of solutions that it contains. When the same component is encountered in the future, the value is retrieved from the cache rather than solving the #SAT problem recursively on the component.

3.2 CNF-to-ODNF

The main difference between #DPLL and our approach is that we record and return sets of orthogonal implicants. #DPLL, on the other hand, only returns the model count.

Before we present the main algorithm, we start by providing some basic procedures / definitions that we will use to describe the algorithm:

- $term(l_1, l_2, \cdots)$: Represents the term $l_1 \wedge l_2 \wedge \cdots$.
- $literals(\varphi)$: The set of literals that are contained in term φ.
- $components(\Sigma)$: Calculates the set of disjoint components of a theory Σ.[3]
- $l_x, \neg l_x$: Respectively, the positive and negative literals corresponding to the variable x.

In order to generate an ODNF fully describing a theory Σ, we make use of the fact that every model of Σ must contain either $x = True$ or $x = False$ for any given variable x. Like the #DPLL algorithm, we partition Σ into two theories corresponding to when $x = True$ and when $x = False$, and recursively determine the ODNF of each theory. Recombining the results simply involves the union of the terms found in each theory: this is because the pair of theories is mutually inconsistent with respect to variable x, and therefore every term from one will be orthogonal to every term of the other.

One important operation that we will require is the term crossproduct operator, $\otimes$. The operator $\otimes$ works on two sets of terms and results in the combination of all pairs of terms. Algorithm 4 describes the procedure that realizes the crossproduct operator.

Algorithm 4 $S_1 \otimes S2$: when S_1 and S_2 are sets of terms, returns all possible combinations of terms between the two sets.

```
1: $S = \{\}$
2: for all $\varphi_1 \in S_1$ do
3:   for all $\varphi_2 \in S_2$ do
4:     if $\varphi_1$ and $\varphi_2$ are consistent then
5:       $S := S \cup term(literals(\varphi_1) \cup literals(\varphi_2))$
6:     end if
7:   end for
8: end for
9: return $S$
```

When we use the $\otimes$ operator there are two special cases worth noting. First, if one of the operands is the empty set then the result of the operator will be the empty set as well. Second, if one of the operands is a set of just one term, then the operator simply combines that term with every term in the other set.

The way we partition a theory Σ into two theories is by first selecting a variable x to partition on, and then producing the two theories, $\Sigma \cup l_x$ and $\Sigma \cup \neg l_x$. After recursively solving for the ODNF of $\Sigma \cup l_x$ and $\Sigma \cup \neg l_x$, we compute the complete set of orthogonal implicants by taking their union.

One additional enhancement is to perform unit propagation and record a set V of all the variables that were assigned. Once we are about to return the union of orthogonal implicants from $\Sigma \cup l_x$ and $\Sigma \cup \neg l_x$, we merge V with the set of implicants (see line 9 in Alg. 5).

It should be noted that if Σ contains no clauses, then the set of orthogonal implicants describing Σ is simply the empty set (this becomes our base case). We now have all the

[3]Two components are disjoint if they do not share a variable.

pieces needed to describe the recursive procedure for generating the ODNF of a theory given in CNF. This is shown as Algorithm 5.

Algorithm 5 Simple-ODNF-Search(CNF formula Σ): returns a set of orthogonal implicants (ODNF) that is equivalent to Σ.

```
1: $\Sigma, V := UnitProp(\Sigma)$
2: if $\Sigma$ contains an empty clause then
3:     return $\{\}$
4: end if
5: $v$ := choose-variable($\Sigma$)
6: $S_t$ := Simple-ODNF-Search($\Sigma \cup l_v$)
7: $S_f$ := Simple-ODNF-Search($\Sigma \cup \neg l_v$)
8: if $V \neq \emptyset$ then
9:     return $(S_t \cup S_f) \otimes \{term(V)\}$
10: else
11:     return $(S_t \cup S_f)$
12: end if
```

Clearly, there is a strong similarity between Algorithms 5 and 3. $\#$DPLL has the property that every model is captured by one (and only one) leaf node in the search tree, but instead of recording how many solutions exist under a partial assignment, we record an orthogonal representation that fully describes the theory that results from a partial assignment.

Just as component analysis has been shown to provide significant improvements in solving $\#$SAT [28], we can leverage the same technique in the task of compiling ODNF. To do this, we take advantage of the following lemma:

Lemma 1 *If Σ is a disjoint theory made up of independent components $\Sigma_1, \Sigma_2, \cdots, \Sigma_k$, then we have:*

$$ODNF(\Sigma) = ODNF(\Sigma_1) \otimes ODNF(\Sigma_2) \otimes \cdots \otimes ODNF(\Sigma_k)$$

To see why this is the case, it is easier to consider only two disjoint theories, Σ_1 and Σ_2, and take advantage of the fact that the $\otimes$ operator is associative (this follows from the definition of $\otimes$). The ODNF of a component completely describes the set of variable assignments that satisfy that component. Since Σ_1 and Σ_2 are disjoint (with respect to variables), an assignment to $\Sigma_1 \cup \Sigma_2$ must be an assignment to both Σ_1 and Σ_2 individually. Taking the $\otimes$ operator thus provides a set of orthogonal implicants that fully describes $\Sigma_1 \cup \Sigma_2$.

Treating disjoint components separately allows us to divide the problem during compilation. Algorithm 6 demonstrates this improvement over Algorithm 5. On line 7 we treat each component individually, and combine the results together on lines 10-14.

As mentioned in Section 3.1, many other improvements from the $\#$SAT literature can be incorporated as well, and we discuss the implementational details in Section 4.1.

4 Implementation and Experimental Evaluation

We implemented our algorithm as an extension to sharpSAT; a C++ implementation of the $\#$DPLL algorithm. In this section, we describe the details of our implementation,

Algorithm 6 ODNF-Search(CNF formula Σ): returns a set of orthogonal implicants (ODNF) that is equivalent to Σ.

```
1: $\Sigma, V := UnitProp(\Sigma)$
2: **if** $\Sigma$ contains an empty clause **then**
3:     **return** $\{\}$
4: **end if**
5: $v :=$ choose-variable($\Sigma$)
6: $S := \{\}$
7: **for all** component $\phi \in components(\Sigma)$ **do**
8:     $S_t :=$ ODNF-Search ($\phi \cup l_v$)
9:     $S_f :=$ ODNF-Search ($\phi \cup \neg l_v$)
10:    **if** first time in the loop **then**
11:        $S := (S_t \cup S_f)$
12:    **else**
13:        $S := S \otimes (S_t \cup S_f)$
14:    **end if**
15: **end for**
16: **if** $V \neq \emptyset$ **then**
17:     **return** $S \otimes \{term(V)\}$
18: **else**
19:     **return** $S$
20: **end if**
```

followed by the experimental evaluation we performed to measure the effectiveness of our approach.

4.1 Implementation Details

In what follows, we describe five key components of c2o.

Pre-Processing

The pre-processing step of sharpSAT involves the potential discovery of variables that can take on only a single setting (referred to as a backbone variable [15]). During this phase, we keep track of every variable that is discovered and as a post-processing step we add all of these variable settings to each orthogonal implicant individually.

Conflict Analysis

Conflict analysis involves the addition of new clauses to the theory that prevent the solver from exploring the same (failed) search space again. Additional effort to maintain the orthogonal implicants is not required because of the following property:

Proposition 1 *If φ is an implicant of the theory Σ (ie. $\varphi \models \Sigma$), and ϕ is an implicate of Σ (ie. $\Sigma \models \phi$), then we can conclude $\varphi \models \phi$.*

This follows from the definition of $\models$ since every model of φ must also be a model of Σ, which in turn must also be a model of ϕ. Therefore, an implicant of a propositional theory must cover *any* implicate of the theory; including those added due to conflict analysis.

Non-Chronological Backtracking

Non-chronological backtracking unsets a number of variables based on a conflict found during search. Backtracking away from a sub-space that has undiscovered solutions would invalidate the final solution count. Because of this, sharpSAT is particular in how far it backtracks once a conflict is found. This guarantees that our approach does not miss orthogonal implicants during the execution of non-chronological backtracking. The modifications required for this portion of the sharpSAT solver involved properly clearing out the orthogonal implicants that have become invalidated (ie. when one of the components is unsatisfiable in lines 7-14 of Algorithm 6).

Implicit Binary Constraint Propagation (IBCP)

The implementation of IBCP in sharpSAT allows us to treat any variable setting due to IBCP as a propagated literal. This is handled as described in Algorithm 6.

Component Caching

In sharpSAT, only the number of solutions is cached with a component. This was extended so that the set of implicants associated with a component was stored in a separate cache. We stored a pointer to this set of implicants with the cached component so when the component is encountered again, we would not need to recalculate the set of implicants associated with it.

When the component cache uses too much memory (a run-time setting for sharpSAT), the component cache is purged of old components. When this occurs, we also delete the set of implicants associated with that component. If the same component is encountered later in the search, both the solution count and set of orthogonal implicants will be re-computed.

The data structure used to store a set of implicants is a naive implementation and this led to the bottleneck of our CNF-to-ODNF converter. Improving this is a point of future work we intend to pursue.

4.2 Experimental Results

To evaluate our algorithm, we investigated three separate issues:

1. How does our implementation compare with other approaches that perform the same task?
2. Which #SAT technologies have an impact on the efficiency of finding orthogonal implicants?
3. Is there correlation between c2o and sharpSAT run-time performance?

Experiments were conducted on a Linux desktop with a Dual Core 2.13GHz processor and 2GB of memory. The propositional theories used were either from SATLIB[4] or generated as random 3SAT problems at a 4.25 clause-to-variable ratio.

During the execution of c2o we limited the memory allowed to 1.5GB. c2o was implemented as a modified version of sharpSAT,[5] and is written in C++ and compiled with

[4] `http://www.satlib.org/`
[5] `http://www2.informatik.hu-berlin.de/~thurley/sharpSAT/index.html`

GNU GCC. Primeii is written in common lisp, and the interpreter used was SBCL.[6] c2d is available for download as a binary.[7]

Algorithm Comparison

In order to see how our solver compares with other approaches that convert CNF-to-ODNF, we tested it on a range of inputs with two other programs: *primeii* [3] and *c2d* [8]. Further discussion on primeii and c2d is provided in Section 5.1. All solvers were run with their default settings and a twenty minute time limit.

As an initial step, primeii uses a variation of the DUALIZATION algorithm to compute a set of prime implicants. The set of prime implicants does not include all possible prime implicants, but enough to fully describe the theory. The software then uses a graph-based approach to convert the set of prime implicants into ODNF.

The c2d software uses a DPLL-based search and records the trace of the search space to compile a d-DNNF representation. The ODNF representation can be extracted from the d-DNNF by enumerating all possible paths in the d-DNNF structure. The complexity of this final process is linear in the size of the ODNF representation.

The problems used for this experiment were random 3-SAT instances with a 4.25 clause-to-variable ratio. The number of variables considered was 100, 150, 175, and 200. Twenty problems from each size class were generated, and the run-times for each of the three solvers recorded.

The means and standard deviations of the solvers' run-times are shown in Figure 2 – the error bars indicate $\pm$ one standard deviation of the mean run-time. Primeii was unable to solve any problems with more than 100 variables in a twenty minute time-out. Additionally, at sizes 175 and 200, c2o ran out of memory on two of the twenty instances (their values are removed from the average / standard deviation calculations).

As an initial step, c2d performs a decomposition of the clausal graph that causes a delay before beginning to solve the problem. This led to c2o outperforming c2d on instances with 175 variables, but in general c2d scales better with problem size, mainly due to the heavy memory usage by c2o. Despite this drawback with c2o, we found the difference in run-time between c2o and c2d to not be statistically significant at these problem sizes. A pairwise t-test was used to verify this, and the results for each problem size are provided in Table 1. For all problem sizes, the statistical comparison was not significant at $p \leq 0.005$.

Problem Size	Mean Difference (seconds)
100	0.118
150	1.384
175	-0.461
200	8.420

Table 1: Pairwise t-test Comparison – None of the differences are significant at $p \leq 0.005$

[6]http://www.sbcl.org/

[7]http://reasoning.cs.ucla.edu/c2d/

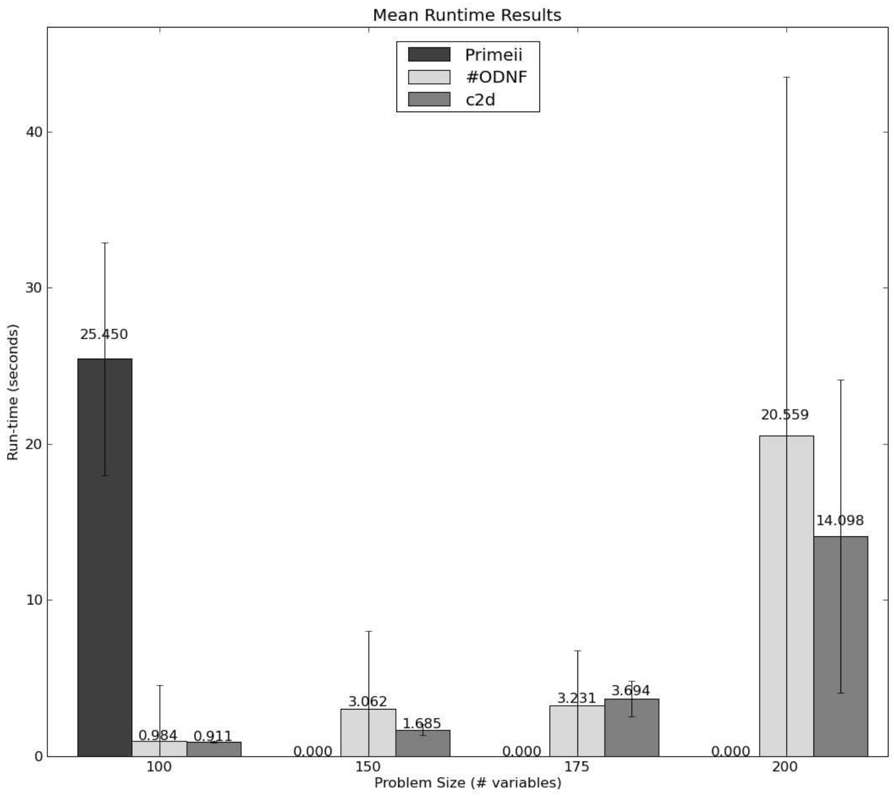

Figure 2: Mean Run-times

Parameter Effect

In order to show the effect of using different #SAT-solving technologies in our solver, we performed an analysis of variance (ANOVA) using the R statistical package [25]. For a particular instance, we ran c2o with every possible parameter setting; in total 32 different variations. The data sets used were the uniform random instance problems and flat graph colouring problems from SATLIB (the number of problems in each data set is 94 and 50 respectively).

The results for the uniform random problem set and flatgraph problem set are summarized in Table 2. The value indicates the probability that using the associated parameters does not affect the run-time – the lower the probability, the more significant changing the parameter settings affects the running time. Results that are very significant (≤ 0.001) are indicated in bold, and results that are mildly significant (≤ 0.05) are indicated in italics. We only include the parameter settings that were significant for either data set. Abbreviations used for the five parameter settings are pre-processing (pp), conflict analysis (ca), non-chronological backtracking (ncb), component caching (cc), and implicit binary constraint propagation (ibcp).

For both problem sets we found component caching to play a significant *negative* role. Upon further investigation, this was found to be caused by the overhead required for maintaining the cache, which was seldom used. In many of the instances for both

Factor(s)	Uniform Pr(>F)	Flatgraph Pr(>F)
ca	$\mathbf{2.220 \times 10^{-13}}$	*1.525×10^{-2}*
cc	$\mathbf{1.414 \times 10^{-10}}$	$< \mathbf{2.000 \times 10^{-16}}$
ibcp	$\mathbf{2.530 \times 10^{-7}}$	2.374×10^{-1}
ca:cc	$\mathbf{2.936 \times 10^{-4}}$	4.531×10^{-1}

Table 2: ANOVA Results for Uniform Random Instance and Flat Graph Colouring Problem Sets

Factor	Uniform	Flatgraph
pp	$-$	$-$
ca	$\times$	$-$
ncb	$-$	$-$
cc	$\times$	$\times$
ibcp	$\surd$	$-$

Table 3: Tukey HSD Results: '$\surd$' and '$\times$' indicate that using the technology had a positive and negative impact (respectively), while '$-$' indicates there was no significant difference in performance.

data sets, there were only a small number of successful cache hits – rendering this feature harmful for usage when generating orthogonal implicants. A more formal analysis of parameter impact was achieved by conducting a Tukey HSD test for each individual setting. The result of this test is presented in Table 3. All tests were performed at a significance level of $p \leq 0.005$.

For the uniform random problem set, we found that the use of conflict analysis hurt the performance as well, and the combination of both conflict analysis and component caching was significant in negatively affecting the run-time. Implicit binary constraint propagation, on the other hand, had a positive impact on the solver's efficiency for uniform random problems.

In contrast to the uniform random problems, we found that conflict analysis provided a minor improvement for the solvers' efficiency on the flat graph colouring problems. The Tukey HSD test did not show a significant result at $p \leq 0.005$, but it was close with $p = 0.0152$.

Correlation to sharpSAT

We would expect improvements to the task of #SAT to also help in our approach to the task of CNF-to-ODNF conversion. One indicator of this would be a correlation between the power of the unmodified sharpSAT and c2o with respect to different parameter settings. For a given instance we measured the run-time on each of the 32 different parameter settings for both sharpSAT and c2o. A correlation coefficient, r, was then calculated from the run-times of each solver.

Table 4 summarizes the mean and standard deviation of all r-values (one for each instance in the data set). The data sets used were the same as the previous experiment.

For the uniform random problem set, we generally found a positive correlation when comparing the run-times. A large majority of instances in the uniform random data set were very strongly correlated (r-value of at least 0.9). We found that those

statistic \ data set	uniform	flat-graph
Mean Correlation	0.606	-0.004
STD Correlation	0.357	0.180

Table 4: Correlation Results: Values indicate the mean and standard deviation of r-values which are computed on a per-instance basis.

with a smaller correlation coefficient tended to have a very small standard deviation of run-time over all parameter settings. We suspect that with such a small deviation, the significance of parameter settings on the run-time is arbitrary and would lead to a low correlation coefficient.

To demonstrate at a high level how the two solvers compare on the uniform random problem set, we plot the run-times for every problem instance on every parameter setting in Figure 3. A portion of the data points are magnified to show the general correlation between the run-times of the two solvers, and a line of $y = x$ is included for reference.

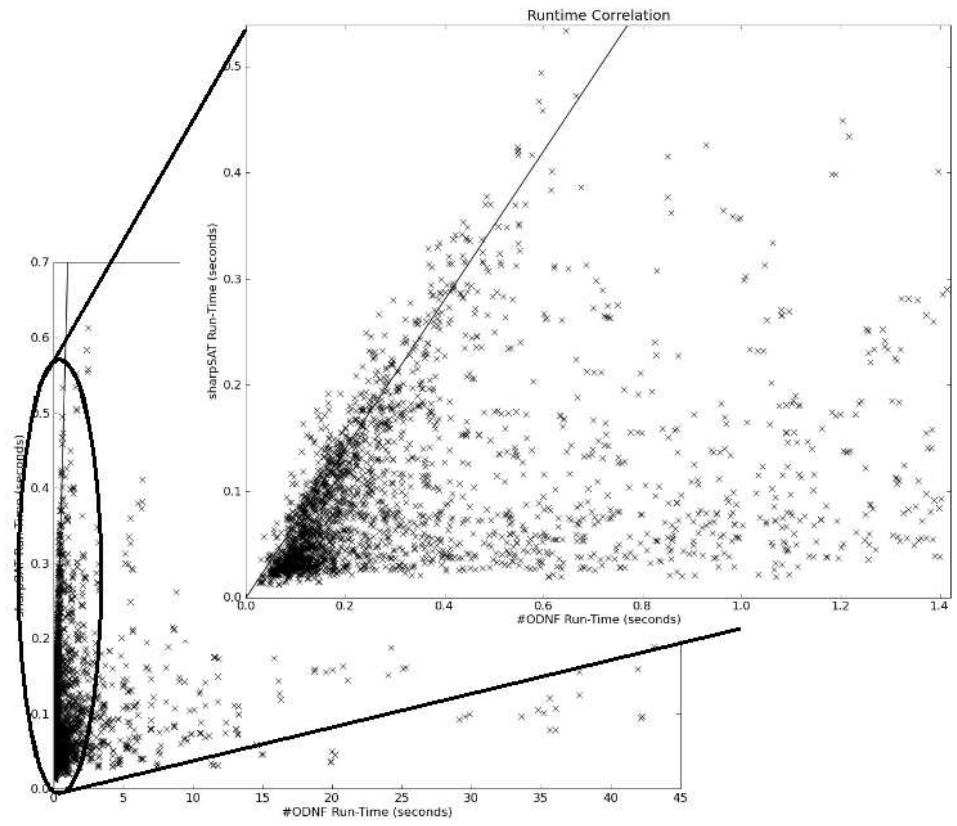

Figure 3: Run-time Comparison on Uniform Random Problems

For the flat graph colouring data set a meaningful comparison was not possible. The reason the correlation given in Table 4 is so low is due to the extremely low deviation on sharpSAT's performance with this data set. This floor effect is caused by the extremely small run-time needed for sharpSAT to solve the problems. To demonstrate this, we ran sharpSAT on the flat graph data set with all 32 parameter settings giving us a set of run-times R_1, and then repeated the experiment to give us a set of run-times R_2. We would expect to see a correlation between R_1 and R_2, but we find that, with an r-value of 0.051, there is very little correlation. The differences in run-times of sharpSAT are dominated by the noise caused by individual executions of the software. In contrast, checking the correlation between two successive runs of c2o on the flat graph problem set yields an r-value of 0.999.

4.3 Discussion

We have established a correlation between the efficiency of c2o and sharpSAT over all parameter settings, but we have also shown the negative impact component caching has on c2o. This is somewhat surprising since the use of component caching has been credited for much of the success of modern #SAT solvers.

One initial distinction is that the success attributed to component caching in #SAT solvers typically includes the use of disjoint component analysis – the process of analyzing each component individually and combining the solutions. This approach is an aspect of sharpSAT that cannot be disabled. Therefore, regardless of parameter setting, c2o will take advantage of disjoint component analysis.

Under the assumption that disjoint component analysis is used, component caching will have a far lower positive impact on the efficiency of a #SAT solver. During the execution on some classes of problems, repeated components may occur very few times, or even never. In these situations, component caching only hinders the efficiency, and in general the problem sets considered in our experimental evaluation fell under this category. In the case of c2o, this negative impact is magnified due to the overhead involved in maintaining the orthogonal implicants while using the component cache.

To verify this was the cause of component caching's behaviour, we performed an analysis of variance on the unmodified sharpSAT software. We found that similar to c2o, component caching had a negative, though statistically insignificant, impact on the solver's efficiency.

5 Discussion and Future Work

In this paper we proposed an algorithm, ODNF-Search, that compiles a propositional theory from CNF to a set of orthogonal implicants. Our implementation of the ODNF-Search algorithm, c2o, is able to compile a set of orthogonal implicants by leveraging advances in modern #SAT-Solver technology.

We compared c2o with two other solvers that have the ability to compute orthogonal implicants: primeii and c2d. c2o clearly outperformed primeii, demonstrating its ability to solve much larger problems, and when compared to c2d we found there to be little statistical difference. However, we expect that c2d would out-perform c2o on problems with more than 200 variables.

We demonstrated the intrinsic properties of our solver by analyzing precisely which #SAT technologies contribute to the performance of c2o. We found that component caching significantly hindered the solver's performance. This was in part due to the limited amount of cache hits during the execution of c2o on the problems considered, but mainly due to the overhead involved in storing partially constructed orthogonal implicants. Conflict analysis had both a positive and negative impact on the performance of c2o depending on the problem type, and for randomly generated problems implicit binary constraint propagation proved beneficial.

Finally, we attempted to determine whether or not the performance of c2o coincides with the performance of sharpSAT on similar problems. Results give a strong indication that new advances in #SAT-solving technology – specifically improvements to the sharpSAT software – will boost the efficiency of c2o.

5.1 Related Work

In Section 4, we compared the performance of c2o with two other pieces of software, primeii, and c2d. Here we elaborate on the differences between the approaches underlying these different software systems.

Based on the classic DUALIZE algorithm [23], Bittencourt proposed a method, called *primeii*, for computing a form similar to ODNF [3]. The first step of primeii is to produce a set of prime implicants – the set generated does not include every prime implicant, but enough to fully describe the theory. This is achieved by performing an A* search through the space of partial implicants where each state in the search space corresponds to a partial assignment of the variables. Neighbouring states correspond to the literals that can be added to the partial assignment such that more clauses are covered by the partial implicant – recall that a partial assignment satisfies a theory in CNF when every clause has a literal corresponding to a variable setting. Final states in the search correspond to prime implicants of the theory.

Primeii works by maintaining a subset of the prime implicants that are sufficient to describe the theory. Once a complete set of prime implicants is generated, the prime implicants are compiled into an orthogonal form by repeated application of the inclusion-exclusion principle [10] – implicants that describe an overlapping set of models are broken up to become orthogonal. The motivation for Bittencourt's approach is to provide exact information about the distribution of solutions in order to aid in the process of Belief Revision. The key difference of our approach is that we do not compute an intermediate form of prime implicants prior to generating an orthogonal representation.

Recent work by Darwiche et al. [8] points out how a similar approach to the one presented here can be used to compile a theory into d-DNNF using a solver that they have constructed called *c2d*. c2d records the search trace of the exhaustive DPLL algorithm, and structures the trace into d-DNNF form. However, before the DPLL procedure begins, c2d pre-computes sets of variables that cause the theory to decompose into disjoint components. This information is stored in a structure referred to as a *d-tree*, and it is used to guide the variable ordering heuristic in DPLL as well as indicate when the theory has become disjoint.

The decisions made by the DPLL algorithm correspond to or-nodes in the d-DNNF representation, and the d-tree corresponds to and-nodes in the d-DNNF representation. Additionally, any literals found through inference are also recorded as and-nodes in the d-DNNF representation. There are two key differences between c2o and c2d. Firstly, the ODNF, which our method explicitly generates, is implicitly represented by the d-DNNF that is generated by c2d. Second, we take advantage of sharpSAT's dynamic component analysis technology to compute disjoint components on-the-fly rather than pre-computing where the decomposition will occur.

Related to the task of compiling ODNF is the minimization of disjunctive normal forms. In this case the input is already in DNF, and the task is to find an equivalent

DNF which has as few terms as possible. Minimization involves removing redundant implicants and finding a representation that covers as many solutions as possible with little overlap between terms. The task of minimizing the DNF form from an ODNF expression, or total enumeration of satisfying assignments, is typically referred to as two-level circuit minimization [6]. Approaches include the ESPRESSO [21] and the Quine-McCluskey [24] algorithms. While the work presented in this paper does not perform two-level circuit minimization, it can be used to pre-process a theory described in CNF into a form that logic minimization algorithms can handle.

5.2 Future Work

There are a number of different areas that we would like to investigate further. Here we outline some of the more interesting ones.

Clause Learning

At a high level we have investigated the impact that conflict analysis has on c2o and shown that it has a negative impact on the solvers efficiency. However, the question remains as to whether or not certain types of clause learning can provide a benefit for the task of orthogonal implicant generation.

There are a number of clause learning schemes available [2], and there is a possibility that the chosen scheme could have an impact on the size of ODNF representations that would be generated. Intuitively, clauses that cause the search space to be as shallow as possible should lead to smaller implicants. This could alter the trade-offs already known to exist between the different clause learning schemes.

Prime Implicants

Instead of recursively generating orthogonal implicants, one could imagine returning a set of prime implicants. In Algorithm 6 we could alter lines 17 and 19 to convert the implicants into prime forms before returning them. This can be achieved by removing literals from each individual implicant until they minimally cover the theory at that step in the algorithm.

This approach would likely cause the solver to take more time when generating implicants, but the final form would be strictly smaller than ODNF. For tasks that require only a set of prime implicants, this form of compilation may generate a representation much smaller than ODNF. Note, however, that the prime implicants generated would not be the set of all prime implicants of the theory.

Compact Representation

A potential inefficiency to our current approach is that we cache partial solutions explicitly rather than in a compact symbolic form, resulting in a large amount of unnecessary memory use. As an alternative, we would like to investigate storing partial results in symbolic form and reconstructing solutions at the end of the process.

More specifically, instead of recording all of the literals that are set during the #DPLL process, we can simply store the decisions made along the way. The final representation would be in d-DNNF, but would *not* represent the theory. However, along with the knowledge of what inference techniques were used (unit propagation, IBCP, etc.), we would be able to reconstruct a d-DNNF similar to what c2d generates, or explicitly list a set of orthogonal implicants similar to what c2o generates. This approach is advantageous because compiling to the full d-DNNF or ODNF could be done in a post-processing step.

Only considering the decision variables would generate a representation strictly smaller than the d-DNNF generated by c2d. Intuitively, this is because the reduced d-DNNF generated would be an induced subgraph of the d-DNNF generated by c2d. The proposed approach would also substantially speed up c2o by requiring far less overhead to maintain the representation during execution. The final d-DNNF would represent a tree of backdoor sets [15], where every root-to-leaf path would represent a backdoor set with respect to the inference used. Enumerating all of the orthogonal implicants can be achieved by enumerating all root-to-leaf paths that end at a positive leaf, and performing the indicated inference on the backdoor set to find out which literals must be additionally in the orthogonal implicant.

Flat d-DNNF Computational Power

Finally, as noted in Section 2.2, ODNF is a flat form of the d-DNNF language. From a knowledge compilation perspective, there may be certain types of queries that are easier to solve in this flat form. We hope to investigate whether or not any existing problems are easily solved by a knowledge base represented in ODNF form.

References

[1] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and Complexity Results for# SAT and Bayesian Inference. In *Annual Symposium on Foundations of Computer Science*, volume 44, pages 340–351. IEEE Computer Society Press, 2003.

[2] P. Beame, H. Kautz, and A. Sabharwal. Understanding the Power of Clause Learning. In *International Joint Conference On Artificial Intelligence*, volume 18, pages 1194–1201. Lawrence Erlbaum Associates Ltd, 2003.

[3] G. Bittencourt. Combining Syntax and Semantics through Prime Form Representation. *Journal of Logic and Computation*, 18(1):13, 2008.

[4] G. Bittencourt, J. Marchi, and R.S. Padilha. A Syntactic Approach to Satisfaction. *Fourth Workshop on the Implementation of Logics*, 2003.

[5] E. Boros, Y. Crama, O. Ekin, P.L. Hammer, T. Ibaraki, and A. Kogan. Boolean Normal Forms, Shellability, and Reliability Computations. *SIAM Journal on Discrete Mathematics*, 13(2):212–226, 2000.

[6] R.K. Brayton. *Logic Minimization Algorithms for VLSI Synthesis*. Springer, 1984.

[7] M. Cadoli. A survey on knowledge compilation. *AI Communications*, 10(3):137–150, 1997.

[8] A. Darwiche. New Advances in Compiling CNF to Decomposable Negation Normal Form. In *Proceedings of European Conference on Artificial Intelligence*, pages 328–332, 2004.

[9] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[10] S.S. Epp. *Discrete mathematics with applications*. Belmont, CA: Thomson-Brooks/Cole, 2004.

[11] M.J. Ghazala. Irredundant Disjunctive and Conjunctive Forms of a Boolean Function. *IBM Journal of Research and Development*, 1(2):171, 1957.

[12] S.O. Hansson. *A Textbook of Belief Dynamics: Theory Change and Database Updating*. Kluwer Academic Publishers, 1999.

[13] J. Huang and A. Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 29:191–219, 2007.

[14] P. Jackson and J. Pais. Computing prime implicants. *Proceedings of the tenth international conference on Automated deduction table of contents*, pages 543–557, 1990.

[15] P. Kilby, J. Slaney, S. Thiebaux, and T. Walsh. Backbones and Backdoors in Satisfiability. In *Proceedings of the National Conference on Artificial Intelligenc*, volume 20, page 1368. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[16] C.M. Li. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, 1997.

[17] M.D.G. Logemann and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, pages 394–397, 1962.

[18] I. Lynce and J.P. Marques-Silva. Probing-Based Preprocessing Techniques for Propositional Satisfiability. In *Proc. the IEEE International Conference on Tools with Artificial Intelligence (ICTAI03)*, 2003.

[19] P. Marquis. Knowledge Compilation Using Theory Prime Implicates. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 837–845. LAWRENCE ERLBAUM ASSOCIATES LTD, 1995.

[20] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli. A new exact minimizer for two-level logic synthesis. *New Trends in Logic Synthesis and Optimization. Kluwer Academic Publishers*, 1992.

[21] P.C. McGeer, J.V. Sanghavi, R.K. Brayton, and A.L. Sangiovanni-Vicentelli. ESPRESSO-SIGNATURE: a new exact minimizer for logic functions. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 1(4):432–440, 1993.

[22] P.B. Miltersen, J. Radhakrishnan, and I. Wegener. On converting CNF to DNF. *Theoretical Computer Science*, 347(1-2):325–335, 2005.

[23] I.B. Pyne and E.J. McCluskey Jr. The Reduction of Redundancy in Solving Prime Implicant Tables. *IRE Trans. EC-II*, 4:473, 1962.

[24] W.V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62(9):627–631, 1955.

[25] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0.

[26] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science, 2006.

[27] S.J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2003.

[28] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[29] B. Selman and H. Kautz. Knowledge compilation using Horn approximations. In *Proceedings of AAAI-91*, pages 904–909, 1991.

[30] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *Journal of the Association for Computing Machinery*, 43(2):193–224, 1996.

[31] J.P.M. Silva and K.A. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *Proceedings of the 31st annual conference on Design automation*, pages 705–711. ACM Press New York, NY, USA, 1994.

[32] J.P.M. Silva and K.A. Sakallah. Conflict Analysis in Search Algorithms for Propositional Satisfiability. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[33] J.R. Slagle, C.L. Chang, and RCT Lee. A New Algorithm for Generating Prime Implicants. *Transactions on Computers*, 100(19):304–310, 1970.

[34] M. Thurley. sharpSAT-Counting Models with Advanced Component Caching and Implicit BCP. In *Ninth International Conference on Theory and Applications of Satisfiability Testing*, 2006.