On Subsumption Removal and On-the-Fly CNF Simplification

Lintao Zhang

Microsoft Research Silicon Valley Lab
1065 La Avenida Ave., Sunnyvale, CA 94043
lintaoz@microsoft.com

Abstract. Conjunctive Normal Form (CNF) Boolean formulas generated from resolution or solution enumeration often have much redundancy. It is desirable to have an efficient algorithm to simplify and compact such CNF formulas on the fly. Given a clause in a CNF formula, if a subset of its literals constitutes another clause in the formula, then the first clause is said to be subsumed by the second clause. A subsumed clause is redundant and can be removed from the original formula. In this paper, we present a novel algorithm to maintain a subsumption-free CNF clause database by efficiently detecting and removing subsumption as the clauses are being added. Furthermore, we present an algorithm that compact the database greedily by recursively applying resolutions that decrement the size of the clause database. Our experimental evaluations show that these algorithms are efficient and effective in practice.

1 Introduction

A propositional Boolean formula can be represented in Conjunctive Normal Form (CNF). A CNF formula is a conjunction (logic AND) of one or more *clauses*, each clause is a disjunction (logic OR) of one or more *literals*. A literal is either a positive or a negative occurrence of a Boolean *variable*. Almost all Boolean Satisfiability (SAT) solvers (e.g. [1] [2] [3] [4]) and Quantified Boolean Formula (QBF) solvers (e.g. [5] [6] [7] [8]) require that the input formula to be in CNF.

Given a clause in a CNF formula, if a subset of its literals constitutes another clause in the formula, then the first clause is said to be subsumed by the second one. Formally, we use $l(C)$ to denote the set of literals of clause C. Given clauses C_1 and C_2, if $l(C_1) \in l(C_2)$, then C_1 subsumes C_2 and C_2 is subsumed by C_1. A subsumed clause is redundant and can be removed from the original formula without changing the Boolean function it represents. Since redundant clauses consume memory and slow down the reasoning process of SAT or QBF solvers, it is desirable to make a CNF formula subsumption free by detecting and removing subsumed clauses. Modern SAT solvers and QBF solvers usually maintains a CNF database internally, and the database is modified constantly during the solving process due to mechanisms such as learning (e.g. [3] [7]) and resolution and expansion [9]. Therefore, it is desirable that subsumption removal can be performed on the fly.

Subsumption has been studied in the theorem proving community [10][11]. In particular, when a new clause is derived, all existing clauses in the database are checked against it to see whether they are subsumed. This check is called *backward subsumption*. The newly derived clause is also checked against all the existing clauses to see if it is subsumed by any existing clauses. This check is called *forward subsumption*. To maintain the subsumption-freeness of the clause database, both checks need to be performed.

SAT solving has been an intensively investigated research field for many years. Therefore, it is surprising that there is little work in the SAT literature that specifically addresses the problem of efficient implementation of subsumption detection and removal, or in a broader sense, on-the-fly CNF compaction. The common perception in the SAT community seems to be that subsumption detection and removal are expensive. Therefore, they are either ignored or only carried out periodically using naive algorithms during the SAT solving process.

For search based algorithms (e.g. [2] [5]), this approach is acceptable. First, search based solvers do not generate as many new clauses compared with resolution based solvers. Second, in modern search based SAT solvers all of the new clauses come from conflict driven learning [3]. Clauses generated from conflict driven learning cannot be subsumed by existing clauses (forward subsumed). This fact can be easily observed as follows. In modern SAT solvers, each learned clause is always an *asserting clause* [3], which is conflicting at current decision level and will become unit after backtracking to an earlier decision level. None of the existing clauses can subsume this clause because otherwise the clause that subsumes it would have been a conflicting clause at an earlier decision level, which is impossible. The third and most important reason that subsumption is usually ignored is because in a search based SAT solvers learned clauses can always be deleted if necessary. Many intelligent clause deletion heuristics take the usefulness of the clauses into account. Since subsumed clauses are not useful in the reasoning process, they have high probability of being garbage collected. Therefore, memory and run time overhead for the subsumed clauses can be kept under control.

Recently some new SAT and QBF algorithms and applications began to appear in the literature that render subsumption detection and removal relevant. Subsumption removal is important for solvers or preprocessors based on resolution (recent examples include QBF solver Quantor [9] and SAT preprocessor NiVer [12]). Resolution based solvers are usually memory limited, and the resolution operation often generate large number of clauses that are subsumed by existing clauses. Another application where subsumption needs to be considered is SAT based solution enumeration (e.g. [13] [14] [15]), which tries to enumerate all solutions of a SAT instance (not simply counting). In both of these cases, newly added clauses cannot be removed if they are not redundant. Therefore, subsumed clauses must be detected and removed in order to free the occupied memory space and increase the capacity of the solver.

In this paper, we propose an efficient algorithm for on-the-fly subsumption detection and removal for CNF clause database. In [9], the author proposed a signature based algorithm for backward subsumption detection. We propose an alternative backward subsumption detection algorithm that does not incur memory overhead as signature based algorithm does. We also propose an efficient forward subsumption

detection algorithm inspired by the two-literal-watching algorithm proposed in SAT solver Chaff [4]. Unlike some of the existing works such as [16] and [17], our algorithms operate on a flat CNF clause database, sometimes called sparse-matrix representation [18]. Since most of the SAT and QBF solvers and preprocessors operates on a flat clause database, our algorithm can be directly applied without changing their native data structures.

To compact the CNF database even more. We propose an algorithm that recursively applies the combination of a slightly modified version of afore mentioned algorithms and resolution. Our proposed algorithm not only makes the clause database subsumption free, but also decremental resolution free. We define a clause database to be *decremental resolution free (DRF)* if no two clauses in the database can be resolved such that the resolvant subsumes either of these two clauses. Experimental results show that this algorithm often produces a more compact CNF database than subsumption removal alone.

This paper is organized as follows. Section 2 describes previous works on CNF database simplification. Section 3 describes our subsumption detection and removal algorithm in detail. Section 4 introduces the algorithm for making a CNF formula decremental resolution free. Section 5 experimentally evaluates the algorithms. Section 6 draws the conclusion and discusses some future works.

2 Previous Work

In this work what we really want to achieve is to efficiently remove redundancy in a CNF database, which is just a special case of Boolean formula simplification. Therefore, in addition to describe some previous works in subsumption detection in 2.1 and 2.2, we also briefly discuss related works in Boolean formula simplification in 2.3 and 2.4.

2.1 Using Trie and ZBDD to Represent CNF

Trie and ZBDD are both proposed as compact representations of CNF clause databases. Zhang proposed to use *trie* to store clauses in the SAT solver SATO [16]. A trie is a tree structure to represent a set in which each path from the root to a leaf corresponds to one key in the represented set. In [16], the trie used for representing the clause database is a ternary tree. Each internal node in the trie structure corresponds to a variable, and its three children edges are labeled Positive, Negative, and Don't Care. A leaf node is either *true* or *false*. Each path from root of the trie to a *true* leaf represents a clause. A trie is ordered if for every internal node V, Parent(V) has a smaller variable index than that of V. The ordered trie structure can detect duplicated and tail subsumed clauses in CNF database cheaply. A clause is said to be tail subsumed by another clause if its first portion of the literals (a prefix) is also a clause in the clause database.

An ordered trie has obvious similarities with Binary Decision Diagrams [19]. Chatalic and Simon proposed to use Zero-suppressed Binary Decision Diagrams

(ZBDDs [20]) to represent clauses [17]. A ZBDD representation of the clause database can detect not only tail subsumption but also head subsumption. Moreover, the authors proposed a subsumption elimination operator to make a CNF subsumption free and a subsumption-free union operator to combine two sets of subsumption-free clauses into a subsumption-free CNF. The operators work on a symbolically compressed dataset, therefore, they are supposed to be efficient.

Both trie and ZBDD representations incur significant overhead for maintaining the clause database. They are not widely used in modern solvers. Most modern SAT solvers use a simple flat clause database and the subsumption elimination operators proposed in [17] are not applicable.

2.2. Signature Based Backward Subsumption Detection

Biere described a signature based backward subsumption detection algorithm for Quantor [9], a QBF solver is based on resolution and expand. A signature is a subset of a finite signature domain D. Each literal l is hashed to a value $h(l) \in D$. The signature $Sig(C)$ of a clause C is the union of the hash values of its literals. The signature of a literal $Sig(l)$ is the union of the signatures of the clauses in which it occurs, and is updated whenever a clause is added to the CNF. When a new clause C is added, its signature is calculated. If there exists a clause D which is subsumed by C, then the signature of C must be a subset of the signature of D, which is a subset of the signatures of all the literals in D. Therefore, a necessary condition for C to subsume any existing clause is that for all literals $l \in C$, $Sig(C) \in Sig(l)$. $Sig(l)$ is the current signature of l, before C is added to the current database.

If the necessary condition fails, no clause in the current CNF can be backward subsumed by the new clause and C can be added. This is called a *cache hit* [9]. Otherwise, in the cache miss case, we need to traverse all clauses of an arbitrary literal in C and explicitly check if C subsumes any of them. If any of the clauses are subsumed, they are removed from the CNF database. During the traversal, inclusion of signatures is a necessary condition and can be easily checked, because the hash value of a clause can be pre-calculated and stored.

After a clause is added, the signatures of all its literals have to be updated. However, if a clause is removed, hash collision does not allow subtracting its signature from all the signatures of its literals. Therefore, the old signature is kept as an over approximation. After certain number of clause removals, the accurate clause signature is recalculated.

One problem with the algorithm is that as the total number of literals in the database increases, the signature of a literal quickly fill up and approach the full domain set D, thus causing many cache misses. To avoid cache misses and increase performance, it is necessary to increase the domain size, which not only slows down the calculation of signature and matching, but also increase memory overhead which is undesirable because the applications we are interested in are often memory bound.

In [9] the author did not describe a forward subsumption detection algorithm. In fact, forward subsumption is invoked periodically by removing all clauses, flushing signatures and then adding back the clauses in reverse chronological order.

2.3 Prime Implicants and Irredundant Cover

A CNF formula is in fact a Product of Sums (POS) logic expression. The problem of subsumption detection and removal in CNF can be regarded as a restricted case of logic simplification. The problem of simplifying 2 level logic circuits has been well studied. In logic circuit simplification, the dual of POS, i.e. Sum of Products (SOP) expressions, are often studied instead. There are algorithms that can generate prime and irredundant covers for a Boolean function using e.g. iterative consensus and min-cover algorithms [21]. By applying similar algorithms to a CNF formula, we can also make it prime and irredundant. A clause is *prime* if deleting any literal from the clause changes the Boolean function represented by the CNF. The set of clauses in a CNF is *irredundant* if removing any of them changes the Boolean function it represents.

There are some attempts in using existing logic simplification techniques for CNF simplification (e.g. [15]). Unfortunately, the algorithms used for logic simplification are usually too expensive for CNF formulas, which often involve thousands of variables and clauses. Moreover, such techniques usually cannot be applied on the fly, which is often necessary for the applications we are interested in.

2.4 Simplification Using Trie and Hash

In [13], the authors described a technique to perform on-the-fly clause compaction and simplification for SAT based solution enumeration and quantification (the algorithm is described in terms of *cubes*, but the same principle can be applied for clauses). Whenever a clause is added to the database, it is checked against the database to see if there exists a clause that consists of the same set of variables as itself and the two clauses can be resolved. If there is such a clause, it is removed from the database and the resolvant (with one less literal) is added into the database recursively. The check is performed efficiently by utilizing a hash table and a trie data structure. The worst time complexity of the check is $O(n^2)$ where n is the number of variables.

The algorithm described in [13] can only detect possible simplification between two clauses that has the same number of literals. In contrast, in section 4 we describe an algorithm that can detect and simplify two clauses that have unequal number of literals as long as the resolvant subsumes at least one of the clauses. Moreover, the algorithm in Section 4 also detects and removes subsumption on the fly.

3 Subsumption Detection and Removal Algorithms

In this section, we describe our proposed subsumption detection and elimination algorithms. The backward subsumption detection algorithm leveraging efficient set intersection operations is described in section 3.1. The forward subsumption detection algorithm using one literal watching is described in section 3.2. In section 3.3 we describe strengthened versions of the forward subsumption algorithm. The forward

and backward subsumption algorithms are combined for subsumption elimination in Section 3.4 to achieve the goal of maintaining a subsumption free CNF database on the fly.

3.1 Backward Subsumption Detection

Given a clause *C*, the backward subsumption detection algorithm finds all clauses whose literals are supersets of the literals in *C*. For each literal *l*, we keep a list of clauses in which this literal occurs, we denote this set *ClauseSet(l)*. The pseudo code for backward subsumption detection is shown in the Algorithm 1.

```
BackwardSubsumedBy ( C )
{
    S = Set of All the Clauses;
    For each literal l in C {
        S = S ∩ ClauseSet ( l );
        If S is empty then break;
    }
    return S;
}
```

Algorithm 1. Backward Subsumption Detection

This algorithm can be implemented efficiently. Set intersection has a linear run time complexity to the total number of elements in the sets. Therefore, the worst case complexity for finding clauses backward subsumed by clause *C* is linear to the total number of clauses in which the literals in *C* occur. In practice, the literals in a clause are sorted in ascending sequence with regard to the number of clauses they appear in. The iteration often ends with just a couple of iterations when S becomes empty.

The advantage of this algorithm compared with the signature base algorithm is that it does not incur any overhead for storing signatures. This is attractive because the applications that we are targeting are usually memory bound. Note that the set of clauses a literal occurs has to be kept by both methods. In fact, it is often maintained by the existing SAT solvers or preprocessors for various other reasons. Therefore, this data structure can be regarded as free.

3.2 Forward Subsumption Detection

Given a clause *C*, forward subsumption detection algorithm finds out if there exists a clause in the current database that is consisted of a subset of the literals in *C*. If such a clause exists, then clause *C* is subsumed by it and should not be added into the CNF.

The algorithm for forward subsumption detection is based on the observation that if a clause *C* is a conflicting clause, then all the clauses that subsume it must also be conflicting clauses. Assume we set all literals in *C* to be *false*. If clause C_1 subsumes *C*, then all literals in C_1 must also be *false*.

Our algorithm to detect whether such C_1 exists is inspired by the 2-literal watching algorithm described in [4]. To detect if a clause is conflicting under a set of variable assignment, we only need to detect whether it contains at least one literal that is not false. We call this algorithm one-literal watching algorithm. Each clause has one literal marked as being watched. Each literal l has a list containing the set of clauses with watched literals corresponding to it. We denote the list as *WatchClauses(l)*.

```
IsForwardSubsumed ( C )
{
    for each literal l in C {
        mark l;
        for each clause C1 in WatchClause(l) {
            l1 = a literal in C1 that is not marked;
            if (no such l1) {
                unmark all literals in C;
                return true;
            }
            else {
                remove C1 from WatchClause(l);
                put C1 in WatchClause(l1);
            }
        }
    }
    unmark all literals in C;
    return false;
}
```

Algorithm 2. Forward Subsumption Detection

Given a clause C, the algorithm *IsForwardSubsumed(C)* returns *true* or *false* depending on whether the clause C is forward subsumed. The complexity of forward subsumption detection for a clause C is about the same as applying n variable assignments (i.e. implications) on the CNF database for a SAT solver, where n is the number of literals in clause C.

3.3 Strengthening Forward Subsumption Detection

A clause C is forward subsumed if an existing clause in the CNF subsumes C. If C is forward subsumed, then it is redundant and should not be added to the database. However, even if C is not subsumed, it may still be redundant. A clause is redundant if current CNF implies it. It is easy to prove that given Boolean formulas f_1 and f_2, if $\neg f_1 \wedge f_2$ is false, then f_2 implies f_1. Subsumption is a specific case of redundancy. We can easily devise other methods to detect redundancy.

A simple way to strengthen the forward subsumption detection algorithm is to apply full Boolean Constraint Propagation (BCP) when setting literals in C to be false. If it leads to a conflict, then C is redundant and should not be added to the CNF. The complexity of this strengthened forward subsumption detection algorithm is about the same as applying n variable branches (i.e. decisions) on the CNF database, where n is the number of literals in C. It is obviously more costly than simple forward subsumption detection, but potentially more effective.

We can strengthen the algorithm even more. If after setting all literals in *C* to *false* and applying BCP do not produce a conflicting clause, we can apply a SAT solver on the resulting CNF to see if it is satisfiable. If it is unsatisfiable, then *C* is redundant and should not be added to the CNF. Obviously, the complexity of this strengthened version of redundancy removal is in the same order of a SAT solving, which can be very expensive. Alternatively, we can set a time limit on the SAT solver, abort the solving when time out, and conservatively assume the clause to be irredundant.

There is a nice trade off between run time and quality of result for different strengthened versions of redundancy removal algorithms. It is up to the users to determine which version is the best fit for their particular applications.

3.4 Maintaining a Subsumption Free CNF Database

By combining the forward and backward subsumption detection algorithms, we obtain the algorithm for maintaining a subsumption free CNF database. Clauses are added one by one into the database. As clause enters the database, subsumed clauses are being detected and removed. Our experiments show that forward subsumption detection is much cheaper than backward subsumption detection. Therefore, when a clause enters the database, we first call procedure *IsForwardSubsumed*. If the clause is subsumed by existing clauses in the database, the clause is discarded and no further check is necessary. Otherwise, if it is not subsumed, *BackwardSubsumption* is called to check whether it subsumes any existing clauses. If so, the subsumed clauses are deleted and the new clause is added to the CNF database. The pseudo codes are shown in Algorithm 3.

```
AddClauseRemoveSubsumption ( C )
{
    if ( IsForwardSubsumed ( C ) )
        return;
    S = BackwardSubsumedBy ( C );
    for each clause C1 in S
        remove C1 from database;
    add clause C into the CNF database
}
```

Algorithm 3. Maintaining a Subsumption-Free Clause Database

4 Maintaining a Decremental Resolution Free CNF Database

A subsumption free CNF can often be simplified further by a resolution followed by subsumption. For example, consider a CNF containing clauses ($a \vee b$) and ($a \vee \neg b \vee c$). It is easy to observe that by resolving these two clauses, the resolvant ($a \vee c$) subsumes the second clause, therefore, the formula can be simplified as ($a \vee b$) $\wedge$ ($a \vee c$). We will call a resolution that generates a clause that subsumes at least one of its parent clauses a *decremental* resolution.

```
AddClauseAndMaintainDRF ( C )
{
    sub_cls = clauses that contain subsets
              of variables in C;
    d0_sub  = clauses in sub_cls that are
              distance 0 from C;
    if (d0_sub is not empty) // clause C is subsumed
        return;
    d1_sub  = clauses in sub_cls that are
              distance 1 from C;
    if ( d1_sub is not empty ) {
        C1 = arbitrarily choose one clause in d1_sub;
        C0 = resolvant of C and C1;
        AddClauseAndMaintainDRF ( C0 );
        return;
    }
    sup_cls = clauses that contain supersets
              of variables in C;
    d0_sup  = clauses in sup_cls that are
              distance 0 from C;
    d1_sup  = clauses in sup_cls that are
              distance 1 to C;
    remove all clauses in d0_sup from CNF;
    if ( d1_sup is not empty ) {
        for each C1 in d1_sup {
            remove C1 from CNF;
            C0 = resolvant of C and C1;
            AddClauseAndMaintainDRF ( C0 );
        }
        AddClauseAndMaintainDRF ( C );
        return;
    }
    add clause C to CNF;
}
```

Algorithm 4. Maintaining a DRF Clause Database

Formally, we define the *distance* between two clauses C_1 and C_2 as the number of variables that occur in both C_1 and C_2 but are in different polarities. Given two clauses C_1 and C_2 with distance 1, the *resolvant* is the clause that contains all literals in C_1 and C_2 except the distance 1 literals. Furthermore, if the set of variables in C_1 is a subset of the variables in C_2, then such a resolution is a *decremental resolution* because the resolvant subsumes C_2. Given a CNF database, if no decremental resolution is possible between any pair of clauses in the database, then the CNF is *decremental resolution free (DRF).*

Notice that an algorithm that maintains a decremental resolution free database may not necessarily generate a more compact representation than the algorithm that maintains a subsumption free database. Suppose we add three clauses ($a \vee \neg d$), ($a \vee b \vee c \vee d$), and ($c \vee d$) into the database, in that order. If we maintain a subsumption free database, since the second clause is subsumed by the third one, we will remove it and obtain a CNF with 2 clauses and 4 literals. On the other hand, if we maintain a DRF database, the second clause resolves with the first clause resulting in a new clause ($a \vee b \vee c$), which is not subsumed by the third clause. Therefore, the end result is a CNF formula with 3 clauses and 7 literals.

The algorithm to maintain a DRF CNF is built upon algorithms similar to the algorithms used for subsumption detection. In subsumption detection, we need to find clauses that contain subsets (as in forward subsumption) or supersets (as in backward subsumption) of the *literals* in a clause *C*. We can modify both of the algorithms slightly to find clauses that contain subsets or supersets of the *variables* in a clause *C*. This can be achieved easily. E.g. in the backward subsumption algorithm, we keep a list of clauses for each variable instead of each literals as *ClauseSet(v)*. In forward subsumption algorithm, we set marks on variables instead of literals. The pseudo code for adding a clause into a DRF clause database is shown in Algoirthm 4.

Algorithm *AddClauseAndMaintainDRF* adds a clause into a CNF formula if and only if it can make sure that there is no possible subsumption or decremental resolution between the clause to be added and the clauses in current CNF. If subsumption or decremental resolution is possible, it eliminates them and then makes a recursive call to itself to perform the check against the new CNF. One interesting implementation detail that needs to be pointed out is that after `d1_sup` is calculated, some of the clauses in it may be deleted by subsequent recursive calls. Therefore, whenever a clause from `d1_sup` is accessed in the `foreach` loop, it must be checked to make sure that the clause is still valid in current CNF.

5 Experimental Results

In this section, we report some preliminary experimental results to show the feasibility and effectiveness of our CNF simplification algorithms. The algorithms are implemented in C++. The program is compiled with Microsoft Visual Studio .Net 2003 and run on a dual 2.8G Intel Xeon processors machine with 2 Gig main memory running Windows XP. Only a single processor is used since the program is not multi-threaded. The CNF formulas we use for this experiment include SAT benchmarks generated from formal verification (1dlx_c_mc_ex_bp_f, c7552, longmult15), logic planning (bw_large.d) and a random generated 3-SAT instance (3SAT_100_400).

The simple application we choose to test the CNF simplification techniques is as follows. We randomly choose a subset of the variables in a CNF formula and eliminate them one by one using resolution. The algorithm we use for variable elimination is the same as the well known algorithm used by Davis and Putnam [1] for SAT solving. Table 1 shows the statistics of the original CNF formulas and the number of variables eliminated for each of the formulas. In Table 1 we also show the statistics of the final CNF formulas generated if we do not apply any CNF simplification techniques during the variable elimination process. This means that all of the clauses generated from resolution are added into the clause database.

Table 2 shows the statistics for the variable elimination when we apply subsumption detection and removal technique described in section 3. Whenever a clause is added to the clause database, we check for subsumption and remove the subsumed clauses so that the CNF is subsumption free using Algorithm 3. The columns under

Formula	Num. Vars Eliminated	Original Formula			After Resolution		Time (s)
		Num Vars	Num Cls	Num Lits	Num Cls	Num Lits	
3SAT_100_400	10	100	400	583	19333	180120	0.453
1dlx_c_mc_ex_bp_f	40	776	3725	10045	154180	2854150	4.641
c7552	700	7652	20423	46377	247714	3702725	10.172
bw_large.d	100	6325	131973	294118	212530	1102847	3.641
longmult15	250	7807	24351	58557	692817	4728551	12.656

Table 1. Variable Elimination by Resolution: No CNF Simplification

"Total Added" are the number of clause and literals added to the clause database (i.e. number of clauses and literals passed to the routine *AddClauseRemoveSubsumption*). Columns under "Forward" and "Backward" show the number of clauses and literals removed due to forward and backward subsumption, respectively. "Final Result" shows the statistics of the final formula obtained. The differences between the sums of "Forward", "Backward" and "Final Result", and "Total Added" are the clauses removed during the resolution process. From the result we find that for our application, usually many more clauses are forward subsumed than backward subsumed. Therefore, for this application, applying forward subsumption detection and removal is very important and should not be foregone. However, there are certain applications where forward subsumption is not interesting, as discussed in the introduction section of this paper. For those applications, we can apply backward subsumption removal directly without invoking the forward subsumption detection algorithm.

Formula	Total Added		Forward		Backward		Final Result		Time (s)
	Num Cls	Num Lits	Num Cls	Num Lits	Num Cls	Num Lits	Num Cls	Num Lits	
3SAT_100_400	3782	25919	2028	16489	141	915	1430	7810	0.203
1dlx_c_mc_ex_bp_f	69686	1073171	39240	706487	5406	83234	22836	260403	9.719
c7552	49259	2352969	3157	144807	869	4238	35316	1368639	18.406
bw_large.d	201832	919829	28673	436138	1462	5792	168678	469222	7.703
longmult15	121059	500805	60649	304472	1061	3425	57353	186920	5.047

Table 2. Variable Elimination: Applying Subsumption Removal

Table 3 shows the statistics for applying the DRF algorithm described in Section 4 during the variable elimination process. Again, under "Total Added" are the clauses and literals added to the CNF database, and the "Final Result" columns show the final CNF obtained. Compare Table 3 with table 2, we find that by maintaining the CNF to be DRF, we usually obtain smaller CNF formulas. Maintaining a CNF to be DRF is more expensive than subsumption removal. However, compare the run time under "Time" column for all three tables, we find that the performances are acceptable for this application because the great reduction in number of clauses added offsets the costs for simplification.

Formula	Total Added		Final Result		Time (s)
	Num Cls	Num Lits	Num Cls	Num Lits	
3SAT_100_400	3007	17578	1235	5933	0.165
1dlx_c_mc_ex_bp_f	60665	817435	20967	211399	10.438
c7552	48708	2328013	35276	1355117	28.547
bw_large.d	189187	715388	166021	441140	12.422
longmult15	119450	445434	54761	160775	6.875

Table 3. Variable Elimination: Maintaining DRF

6 Conclusions and Future Work

In this paper we discuss methods to eliminate redundancy in a CNF clause database on-the-fly. We describe an algorithm for subsumption detection and elimination. We then describe an algorithm that uses the subsumption detection and elimination algorithm to make a CNF database *decremental resolution free*. The purpose of this work is to automatically compact and simplify CNF formulas derived from operations such as resolution and solution enumeration. Such work is important because many new applications of SAT solvers such as quantification elimination and abstraction refinement are performed by either resolution or solution enumeration.

Many things are unexplored in this paper. Our algorithm is still not powerful enough to produce prime and irredundant CNF database. Traditional methods to generate prime and irredundant implicants are too expensive to be practical in this setting. How to balance the runtime cost and the quality of the result for on-the-fly CNF simplification is a very interesting problem worth much further investigation.

Moving further along this line, it is interesting to see if it is possible to efficiently generate formulas that have fewer restrictions than CNF. CNF is essentially a two level circuit, which is known to blow up in size for certain Boolean functions. It would be interesting to see if multilevel representations for a Boolean function can be efficiently derived during the solution enumeration or resolution process.

References

[1] M. Davis and H. Putnam, A computing procedure for quantification theory, *Journal of the ACM*, vol. 7, pp. 201-215, 1960.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. In *Communications of the ACM, 5:394-397, 1962*

[3] João P. Marques-Silva and Karem A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability, In *IEEE Tran. on Computers, vol. 48, 506-521, 1999*

[4] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT Solver, In *Proceedings of the Design Automation Conference, 2001*

[5] M. Cadoli, M. Schaerf, A. Giovanardi and M. Giovanardi. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation, in *Highlights of Satisfiability Research in the Year 2000, IOS Press, 2000*

[6] H. Kleine-Büning, M. Karpinski and A. Flögel. Resolution for quantified Boolean formulas. In *Information and Computation, 117(1):12-18, 1995*

[7] L. Zhang and S. Malik, Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver, In *Proc. of 8th International Conference on Principles and Practice of Constraint Programming (CP2002)*. Ithaca, NY, Sept. 2002.

[8] E. Giunchiglia, M. Narizzano and A. Tacchella,. Qube: a system for Deciding Quantified Boolean Formulas Satisfiability,. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR), 2001*

[9] A. Biere. Resolve and Expand. In *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, 2004.

[10] G. Gottlob, A. Leitsch, On the efficiency of subsumption algorithms, Journal of the ACM (JACM), Volume 32, Issue 2, pg. 280-295, April 1985

[11] A. Voronkov, "Implementing Bottom-up Procedures with Code Trees: a Case Study of Forward Subsumption", *Uppsala University, Computing Science Department, UPMAIL Technical Report 88*, 1994

[12] Sathiamoorthy Subbarayan and Dhiraj K Pradhan, NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances, In *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, 2004.

[13] P. Chauhan, E. M. Clarke, D. Kroening, Using SAT Based Image Computation for Reachability Analysis, *Technical Report CMU-CS-03-151, Carnegie Mellon University, School of Computer Science*, July, 2003

[14] Ken L. McMillan, Applying SAT Methods in Unbounded Symbolic Model Checking, in Proc. *14th International Conference on Computer Aided Verification (CAV02)*, Copenhagen, Denmark, July 2002

[15] Hyeong Ju Kang, In-Cheol Park, SAT-Based Unbounded Symbolic Model Checking, in *Proc. 40th Design Automation Conference (DAC03)*, 2003

[16] H. Zhang, "SATO: An efficient propositional prover," presented at *International Conference on Automated Deduction (CADE)*, 1997.

[17] Philippe Chatalic, Laurent Simon, Multi-Resolution on Compressed Sets of Clauses, in *Twelfth International Conference on Tools with Artificial Intelligence (ICTAI'00)*, 2000

[18] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers", In *Proc. of 8th International Conference on Computer Aided Deduction(CADE 2002*, Copenhagen, Denmark, July 2002

[19] Randy Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers, Vol. C - 35, No. 8, pp. 677 - 691*. August, 1986,

[20] S. Minato, "Zero-Suppressed BDDs for Set Manipula-tion in Combinatorial Problems," in *Proc. of the Design Automation Conference (DAC93), pp. 272-277*, 1993

[21] G. Hachtel and F. Somenzi, "Logic Sysntheiss and Verification Algorithms," Kluwer Academic Publishers, 1996