Computing Prime Implicates by Pruning the Search Space and Accelerating Subsumption

Andrew Matusiewicz[1], Neil V. Murray[1], and Erik Rosenthal[1]

Institute for Informatics, Logics, & Security Studies,
Department of Computer Science,
University at Albany – SUNY,
Albany, NY 12222, USA
phone: +1 518-442-3393
a_matusi@cs.albany.edu
nvm@cs.albany.edu
erik.rosenthal@cs.albany.edu

Abstract. The *prime implicate trie* (pi-trie) of a logical formula is a tree whose branches are labeled with the prime implicates of the formula. An algorithm that builds the pi-trie from a logical formula was introduced in [12]; it builds the trie recursively and makes extensive use of subsumption testing. A more efficient version in which the use of subsumption is reduced is presented in this paper. The improved algorithm is illustrated with experiments.
The algorithm naturally lends itself to selecting restricted sets — for example, only positive prime implicates or only those of bounded length, both useful properties for some applications. Such restrictions can provide significant advantages in efficiency since the set of all prime implicates of a logical formula is typically exponential in size.

Key words: Propositional logic, prime implicate, subsumption, trie, clause.

1 Introduction

Prime implicants were introduced by Quine [17] as a means of simplifying propositional logical formulas in *disjunctive normal form* (DNF); prime implicates play the dual role in *conjunctive normal form* (CNF). Implicants and implicates have many applications, including abduction and program synthesis of safety-critical software [8]. All prime implicate algorithms of which the authors are aware make extensive use of clause set subsumption; improvements in the pi-trie algorithm and its core subsumption operations are thus relevant to all such applications.

Numerous algorithms have been developed to compute prime implicates — see, for example, [1, 3, 4, 6, 7, 9, 10, 16, 18, 23–25]. Most use clause sets or truth tables as input; the pi-trie algorithm introduced in [12] is one of rather few that accept arbitrary formulas as input. This recursive algorithm stores the prime implicates in a *trie* — a labeled tree. In this paper, the original algorithm is improved and shown to have a number of interesting properties.

One particularly nice property is that, at every stage of the recursion, once the subtrie rooted at a node is built, some superset of each branch in the subtrie is a prime implicate of the original formula. This property, along with the manner in which the recursion assembles branches, admits the use of filters in the algorithm to compute restricted sets of prime implicates — for example, only positive ones or those not containing specific variables — without computing any others. These filters prune the search space, often substantially, and experiments indicate that significant speedups can be obtained.

The improvements developed here[1] include a much cleaner description of the algorithm. Unnecessary subsumption tests are eliminated, in part by performing subsumption checks between tries whose branches represent clause sets. Experiments indicate that the new trie-based subsumption tests are superior to the clause by clause approach originally employed. The result is a substantially faster algorithm.

Basic terminology and the fundamentals of pi-tries are summarized in Section 2. The analysis that leads to the improved pi-trie algorithm is developed in Section 3, and trie-based set operations and experiments with them are described in Section 4. The improved pi-trie algorithm and the results of experiments that compare it with the original are presented in Section 5. Properties of the algorithm allowing selective computation are explained in Section 6. The efficiency improvements are evident in the outcomes of the experiments described in Section 7.

2 Preliminaries

The terminology used in this paper for logical formulas is standard: An *atom* is a propositional variable, a *literal* is an atom or the negation of an atom, and a *clause* is a disjunction of literals.[2] Clauses are often referred to as sets of literals; though it is sometimes convenient to represent them with logical notation. In this paper, either notation will be used according to which seems most natural. An *implicate* of a formula is a clause entailed by the formula, and a non-tautological clause is a *prime implicate* if no proper subset is an implicate. The set of prime implicates of a formula $\mathcal{F}$ is denoted $\mathcal{P}(\mathcal{F})$. Note that a tautology has no prime implicates, and the empty clause is the only prime implicate of a contradiction.

2.1 Background

The trie is a well-known data structure introduced by Fredkin in 1960 [5]; a variation was introduced by Morrison in 1968 [14]. It is a tree in which each branch represents the sequence of symbols labeling the nodes[3] on that branch, in descending order. Tries have been used in a variety of settings, including representation of logical formulas — see, for example, [19]. The nodes along each branch represent the literals of a clause, and the conjunction of all such clauses is a CNF equivalent of the formula. If there is

[1] The algorithm and some of the accompanying results were introduced without proof in [13].

[2] It is common in the literature to see the term *clause* used for a conjunction of literals, especially with disjunctive normal form.

[3] Many variations have been proposed in which arcs rather than nodes are labeled, and the labels are sometimes strings rather than single symbols.

no possibility of confusion, the term *branch* may be used for the clause it represents. It will generally be assumed that a variable ordering has been selected, and that nodes along each branch are labeled consistently with that ordering. A trie that stores all prime implicates of a formula is called a *prime implicate trie*, or simply a pi-trie.

It is convenient to employ a ternary representation of pi-tries, with the root labeled 0 and the ith variable appearing only at the ith level. If $v_1, v_2, \ldots, v_n$ are the variables, then the children of a node at level i are labeled v_{i+1}, $\neg v_{i+1}$, and 0, left to right. With this convention, any subtrie (including the entire trie) is easily expressed as a four-tuple consisting of its root and the three subtries. For example, the trie $\mathcal{T}$ can be written $\langle r, \mathcal{T}^+, \mathcal{T}^-, \mathcal{T}^0 \rangle$, where r is the label of the root of $\mathcal{T}$, and $\mathcal{T}^+$, $\mathcal{T}^-$, and $\mathcal{T}^0$ are the three (possibly empty) subtries representing, respectively, the set of prime implicates containing v, containing $\neg v$, and not containing the variable v at all. The ternary representation will be assumed in this paper unless otherwise stated.

The reader is assumed to be familiar with resolution and subsumption [20]; Lemma 1 and the following observations are either well known or obvious and are stated without proof.

Observations.

1. Each implicate of a logical formula is subsumed by at least one prime implicate.
2. $\mathcal{P}(\mathcal{F})$ is subsumption free.
3. Resolution is consequence complete (modulo subsumption) for propositional CNF formulas. Thus, if C is an implicate of $\mathcal{F}$, there is a clause D that subsumes C and can be derived from $\mathcal{F}$ by resolution. In particular, every prime implicate of $\mathcal{F}$ can be derived by resolution.
4. A formula is logically equivalent to the conjunction of its prime implicates.

Let $\mathcal{S}$ be a set of clauses. Then $\mathcal{S}$ is said to be $prime$ if $\mathcal{S}$ is the set of prime implicates of a logical formula $\mathcal{F}$. A *resolution-subsumption* operation on $\mathcal{S}$ consists of a single resolution step followed by removal of all subsumed clauses, and $\mathcal{S}$ is a *resolution-subsumption fixed point* if every resolution-subsumption operation on $\mathcal{S}$ leaves $\mathcal{S}$ unchanged.

Lemma 1. A clause set $\mathcal{S}$ is prime iff $\mathcal{S} = \mathcal{P}(\mathcal{S})$ iff $\mathcal{S}$ is a resolution-subsumption fixed point. In particular, if $\mathcal{S}$ is any set of clauses, then $\mathcal{P}(\mathcal{S})$ is the unique resolution-subsumption fixed point that is logically equivalent to $\mathcal{S}$. $\square$

3 Prime Implicates under Truth-Functional Substitution

The pi-trie algorithm computes $\mathcal{P}(\mathcal{F})$ recursively from $\mathcal{P}(\mathcal{F}[\alpha/v])$, where α, one of the constants 0 or 1, is substituted for every occurrence of the variable v in $\mathcal{P}(\mathcal{F})$. This section contains an analysis of the relationships among $\mathcal{P}(\mathcal{F})$, $\mathcal{P}(\mathcal{F}[1/v])$, and $\mathcal{P}(\mathcal{F}[0/v])$.

Fix one variable v and partition $\mathcal{P}(\mathcal{F})$ into clause sets $\mathcal{P}_v$, $\mathcal{P}_{\neg v}$, and $\mathcal{P}'$, where $\mathcal{P}_v$ is the set of all clauses containing v, $\mathcal{P}_{\neg v}$ all clauses containing $\neg v$, and $\mathcal{P}'$ the remaining clauses. Observe that $\mathcal{P}_v[1/v]$ is empty since every clause reduces to the constant 1, and $\mathcal{P}'[1/v] = \mathcal{P}'$, since v does not occur in $\mathcal{P}'$. Also, letting $\mathcal{P}_{\neg v,1} = \mathcal{P}_{\neg v}[1/v]$, $\mathcal{P}_{\neg v,1}$ can be obtained by removing $\neg v$ from each clause in $\mathcal{P}_{\neg v}$. Thus $\mathcal{P}(\mathcal{F})[1/v] = \mathcal{P}_{\neg v}[1/v] \cup \mathcal{P}_v[1/v] \cup \mathcal{P}'[1/v] = \mathcal{P}_{\neg v,1} \cup \mathcal{P}'$. If the set of clauses in $\mathcal{P}'$ not subsumed by any clause in $\mathcal{P}_{\neg v,1}$ is denoted $\tilde{\mathcal{P}}'$, then $\mathcal{P}(\mathcal{F}[1/v])$ is logically equivalent to $\mathcal{P}_{\neg v,1} \cup \tilde{\mathcal{P}}'$. The next lemma states that $\mathcal{P}_{\neg v,1} \cup \tilde{\mathcal{P}}'$ is in fact the set of prime implicates of $\mathcal{F}[1/v]$.

Lemma 2. If $\mathcal{F}$ is any logical formula, then $\mathcal{P}(\mathcal{F}[1/v]) = \mathcal{P}_{\neg v,1} \cup \tilde{\mathcal{P}}'$.

Proof. By Lemma 1, since $\mathcal{P}(\mathcal{F}[1/v])$ is logically equivalent to $\mathcal{P}_{\neg v,1} \cup \tilde{\mathcal{P}}'$, it suffices to show $\mathcal{P}_{\neg v,1} \cup \tilde{\mathcal{P}}'$ to be a resolution-subsumption fixed point. Since $\tilde{\mathcal{P}}' \subseteq \mathcal{P}' \subseteq \mathcal{P}(\mathcal{F})$ — i.e., since $\tilde{\mathcal{P}}'$ is a set of prime implicates — no clause in $\tilde{\mathcal{P}}'$ can subsume any other clause in $\tilde{\mathcal{P}}'$. The same holds for $\mathcal{P}_{\neg v,1}$ since it holds for $\mathcal{P}_{\neg v}$, and the clauses of $\mathcal{P}_{\neg v,1}$ are obtained from the clauses of $\mathcal{P}_{\neg v}$ by removing $\neg v$.[4] Showing that resolving two clauses in $\mathcal{P}_{\neg v,1} \cup \tilde{\mathcal{P}}'$ must produce a subsumed clause will complete the proof. There are two cases to consider: resolving two clauses from $\tilde{\mathcal{P}}'$ and resolving with at least one clause from $\mathcal{P}_{\neg v,1}$.

Case 1. Let C be the resolvent of two clauses in $\tilde{\mathcal{P}}'$. Then C does not contain v and is subsumed by a clause $\tilde{C}$ in $\mathcal{P}(\mathcal{F})$. The clause $\tilde{C}$ is in $\mathcal{P}'$ and thus is either in $\tilde{\mathcal{P}}'$ or is subsumed by a clause in $\mathcal{P}_{\neg v,1}$.

Case 2. Let C be the resolvent of two clauses with at least one from $\mathcal{P}_{\neg v,1}$. Then $C \cup \{\neg v\}$ is the resolvent of the corresponding clauses from $\mathcal{P}(\mathcal{F})$, so there is a clause $\tilde{C} \in \mathcal{P}(\mathcal{F})$ that subsumes $C \cup \{\neg v\}$. If $\neg v \in \tilde{C}$, then $\tilde{C} \in \mathcal{P}_{\neg v}$, so $\tilde{C} - \{\neg v\}$ is in $\mathcal{P}_{\neg v,1}$ and subsumes C. Otherwise, $\tilde{C} \in \mathcal{P}'$, and the analysis of Case 1 applies. □

There is an entirely similar result when 0 is substituted for v.

Corollary. Let $\mathcal{P}_{v,0} = \mathcal{P}_v[0/v] = \{C - \{v\} \mid C \in \mathcal{P}_v\}$, and let $\hat{\mathcal{P}}'$ be the clauses in $\mathcal{P}'$ not subsumed by any clause in $\mathcal{P}_{v,0}$. Then $\mathcal{P}(\mathcal{F}[0/v]) = \mathcal{P}_{v,0} \cup \hat{\mathcal{P}}'$.

Henceforth, when the formula $\mathcal{F}$ and the specified variable v are clear, $\mathcal{F}[0/v]$ and $\mathcal{F}[1/v]$ will be denoted by $\mathcal{F}_0$ and $\mathcal{F}_1$, respectively, and $\mathcal{P}(\mathcal{F})$, $\mathcal{P}(\mathcal{F}_0)$, and $\mathcal{P}(\mathcal{F}_1)$ will be denoted by $\mathcal{P}$, $\mathcal{P}_0$, and $\mathcal{P}_1$, respectively.

It follows from Lemma 2 and its corollary that, with respect to variable v, $\mathcal{P}$ can be transformed into $\mathcal{P}_0$ (or into $\mathcal{P}_1$) in polynomial time. Moreover, the lemma places a limitation on the required checks for subsumption. Specifically, for $\mathcal{P}_0$, it is only necessary to check whether clauses in $\mathcal{P}_{v,0}$ subsume clauses in $\mathcal{P}$ that contain neither v nor $\neg v$. The time for these checks is proportional to the product of the clause set sizes. Unfortunately, this does not make transforming $\mathcal{P}_0$ and $\mathcal{P}_1$ into $\mathcal{P}$ a polynomial operation, which is a major focus of this paper.

Observe that $\mathcal{P} \equiv \mathcal{F} \equiv (v \vee \mathcal{F}_0) \wedge (\neg v \vee \mathcal{F}_1) \equiv (v \vee \mathcal{P}_0) \wedge (\neg v \vee \mathcal{P}_1)$. Let $\mathcal{P}_{0,v}$ and $\mathcal{P}_{1,\neg v}$ be the clause sets produced by, respectively, distributing v over $\mathcal{P}_0$ and $\neg v$ over $\mathcal{P}_1$. Then $\mathcal{P}_{0,v}$ and $\mathcal{P}_{1,\neg v}$ are (separately) resolution-subsumption fixed points because $\mathcal{P}_0$ and $\mathcal{P}_1$ are. Subsumption cannot hold between one clause in $\mathcal{P}_{0,v}$ and one

[4] It is possible that removing $\neg v$ could create a subsumption relationship to clauses in $\mathcal{P}'$, but such clauses are removed from $\mathcal{P}'$ to form $\tilde{\mathcal{P}}'$.

in $\mathcal{P}_{1,\neg v}$ since one contains v and the other $\neg v$. Thus if $\mathcal{P}_{0,v} \cup \mathcal{P}_{1,\neg v}$ is not a resolution-subsumption fixed point, producing a fixed point from it must begin with resolvents having one parent from each. These can be restricted to resolving on v and $\neg v$ because any other produces a tautology. It follows from Theorem 1 below that it is sufficient to consider only resolvents of this form.

The next three lemmas appeared in [15] or in [12]. All are straightforward; the first has surely has been observed independently by several authors. Proofs are included because they provide insight into the lemmas.

Lemma 3. The set of all implicates of $\mathcal{F} \vee \mathcal{G}$ is precisely the intersection of the implicate sets of $\mathcal{F}$ and $\mathcal{G}$.

Proof. Suppose first that C is an implicate of $\mathcal{F} \vee \mathcal{G}$, and let I be any interpretation that satisfies $\mathcal{F}$. Then I satisfies $\mathcal{F} \vee \mathcal{G}$, so I must satisfy C. Thus, C is an implicate of $\mathcal{F}$. Similarly, C is an implicate of $\mathcal{G}$; i.e., C is in the intersection.

Conversely, if C is an implicate of both $\mathcal{F}$ and $\mathcal{G}$, then any interpretation that satisfies $\mathcal{F} \vee \mathcal{G}$ obviously satisfies $\mathcal{F}$ or satisfies $\mathcal{G}$ and thus must satisfy C — i.e., C is an implicate of $\mathcal{F} \vee \mathcal{G}$. $\square$

Lemma 4. Let C be a clause not containing the variable v. Then C is an implicate of $\mathcal{F}$ iff C is an implicate of $\mathcal{F}_0 \vee \mathcal{F}_1$.

Proof. Suppose first that C is an implicate of $\mathcal{F}_0 \vee \mathcal{F}_1$. Then, by the last lemma, C is an implicate of both $\mathcal{F}_0$ and $\mathcal{F}_1$. To see that C is an implicate of $\mathcal{F}$, let I be an interpretation that satisfies $\mathcal{F}$. If $I(v) = 1$, then I satisfies $\mathcal{F}_1$, so I satisfies C, and if $I(v) = 0$, then I satisfies $\mathcal{F}_0$, so I satisfies C.

Conversely, suppose that C is an implicate of $\mathcal{F}$, and let I be an interpretation defined on the variables of $\mathcal{F}_0 \vee \mathcal{F}_1$ that satisfies $\mathcal{F}_0 \vee \mathcal{F}_1$, say I satisfies $\mathcal{F}_1$. Extend I by defining $I(v) = 1$. Then I satisfies $\mathcal{F}$, so I satisfies C. Since v is not in C, the value of $I(v)$ is not relevant.

Lemma 5. Let $\mathcal{F}$ and $\mathcal{G}$ be logical formulas, and let $C \in \mathcal{P}(\mathcal{F} \vee \mathcal{G})$. Then there exist prime implicates C_0 of $\mathcal{F}$ and C_1 of $\mathcal{G}$ such that $C = C_0 \cup C_1$.

Proof. Let D_0 be the subset of C that has the variables of C that occur in $\mathcal{F}$; similarly let D_1 be the subset of C that has the variables of C that are in $\mathcal{G}$. First, D_0 is an implicate of $\mathcal{F}$ (and D_1 is an implicate of $\mathcal{G}$): If I is an interpretation such that $I(\mathcal{F}) = 1$, extend I so that I falsifies all literals in $C - D_0$. Since $I(\mathcal{F}) = 1$, $I(\mathcal{F} \vee \mathcal{G}) = 1$, so $I(C) = 1$. Thus $I(D_0) = 1$.

Now let C_i be a prime subset of D_i ($i = 0, 1$), and consider $C_0 \cup C_1$. Since $C_0 \cup C_1 \subseteq C$, it suffices to prove that $C_0 \cup C_1$ is an implicate of $\mathcal{F} \vee \mathcal{G}$. Suppose $I(\mathcal{F} \vee \mathcal{G}) = 1$; then $I(\mathcal{F}) = 1$ or $I(\mathcal{G}) = 1$, say $I(\mathcal{F}) = 1$. Then, since C_0 is an implicate of $\mathcal{F}$, $I(C_0) = 1$, so $I(C_0 \cup C_1) = 1$. $\square$

The next theorem is a variant of the last lemma that is useful for the development of the main algorithm of this paper.

Theorem 1. Let $\mathcal{F}$ be a logical formula, let v be a variable in $\mathcal{F}$, and suppose that E is a prime implicate of $\mathcal{F}$ not containing v. Then there exist $C \in \mathcal{P}_0$ and $D \in \mathcal{P}_1$ with $E = C \cup D$.

Proof. By Lemma 4, E is an implicate of both $\mathcal{F}_0$ and $\mathcal{F}_1$ and thus a superset of prime implicates C of $\mathcal{F}_0$ and D of $\mathcal{F}_1$. That is, E is a superset of $C \cup D$. Since E is prime and $C \cup D$ is an implicate of $\mathcal{F}$, E is in fact equal to $C \cup D$. $\square$

Theorem 1 and the discussion leading up to it suggest a method for computing $\mathcal{P}$ from $\mathcal{P}_0$ and $\mathcal{P}_1$: Perform all resolutions on v and $\neg v$ on pairs of clauses in $\mathcal{P}_{0,v} \cup \mathcal{P}_{1,\neg v}$ and then obtain the subsumption minimal subset.

Observations. Let $C \in \mathcal{P}_0$, and let $C_v = \{v\} \cup C$.

1. If every interpretation that satisfies $\mathcal{F}$ satisfies v, then $\{v\}$ is a prime implicate and $\mathcal{F}_0$ is unsatisfiable. Moreover, the remaining prime implicates — i.e., those that do not contain v — are precisely the prime implicates of of $\mathcal{F}_1$.
2. Though C_v may not be prime, it is an implicate of $\mathcal{F}$: Any interpretation that satisfies $\mathcal{F}$ either satisfies v, in which case it satisfies C_v, or it falsifies v, in which case it satisfies $\mathcal{F}_0$ and thus satisfies C.
3. If C_v is not a prime implicate of $\mathcal{F}$, then some prime implicate D of $\mathcal{F}$ (properly) subsumes C_v. If v were in D, then $D - \{v\}$ would be an implicate of $\mathcal{F}_0$ that properly subsumes C, which is impossible since C is a prime implicate of $\mathcal{F}_0$. But then D is an implicate of $\mathcal{F}_0$ that subsumes C — i.e., it must be the case that $D = C$. In particular, either C_v or C is a prime implicate of $\mathcal{F}$.
4. Suppose that $C \in \mathcal{P}$.
   (a) By Theorem 1, since $C \in \mathcal{P}_0$, there must be a prime implicate D of $\mathcal{P}_1$ such that $C = C \cup D$; i.e., $D \subseteq C$.
   (b) If D properly subsumes C, then D cannot be an implicate of $\mathcal{F}$ since C is. Thus $\{v\} \cup D$ is a prime implicate of $\mathcal{F}$.
   (c) If $D = C$, then D is a prime implicate of $\mathcal{F}$.

An immediate consequence of these observations is

Lemma 6. Let C be a prime implicate of $\mathcal{F}_0$, and let $C_v = \{v\} \cup C$. Then C_v is a prime implicate of $\mathcal{F}$ iff C is not subsumed by any implicate of $\mathcal{F}_1$, and C is a prime implicate of $\mathcal{F}$ iff C is subsumed by some implicate of $\mathcal{F}_1$. A similar result holds for prime implicates of $\mathcal{F}_1$. $\square$

Let $\mathcal{P}_0$ be partitioned into two subsets, $\mathcal{P}_0^{\supseteq}$, the clauses in $\mathcal{P}_0$ subsumed by some clause in $\mathcal{P}_1$, and $\mathcal{P}_0^{\nsupseteq}$, the remaining clauses in $\mathcal{P}_0$. Similarly, let $\mathcal{P}_1$ be partitioned into $\mathcal{P}_1^{\supseteq}$, clauses subsumed by clauses in $\mathcal{P}_0$, and $\mathcal{P}_1^{\nsubseteq}$, the remaining clauses. From Lemma 6, $(v \vee \mathcal{P}_0^{\nsubseteq}) \cup (\neg v \vee \mathcal{P}_1^{\nsubseteq})$ accounts for all prime implicates of $\mathcal{F}$ that contain v, and from Observation 4b, $\mathcal{P}_0^{\supseteq}$ and $\mathcal{P}_1^{\supseteq}$ are prime implicates that do not contain v. The remaining prime implicates of $\mathcal{F}$ also do not contain v and, by Theorem 1, must have the form $C \cup D$, where $C \in \mathcal{P}_0^{\nsupseteq}$ and $D \in \mathcal{P}_1^{\nsupseteq}$. This proves

Theorem 2. The set of all prime implicates of $\mathcal{F}$ is

$$\mathcal{P} = (v \vee \mathcal{P}_0^{\not\subseteq}) \cup (\neg v \vee \mathcal{P}_1^{\not\subseteq}) \cup \mathcal{P}_0^{\supseteq} \cup \mathcal{P}_1^{\supseteq} \cup \mathcal{U} \qquad (*)$$

where $\mathcal{U}$ is the maximal subsumption-free subset of $\{C \cup D \mid C \in \mathcal{P}_0^{\not\subseteq}, D \in \mathcal{P}_1^{\not\subseteq}\}$ in which no clause is subsumed by a clause in $\mathcal{P}_0^{\supseteq}$ or in $\mathcal{P}_1^{\supseteq}$. □

Example. Let $\mathcal{F}$ be the clause set

$$\{\{a, c, b\}, \{a, e\}, \{\neg a, c\}\{\neg a, \neg d\}, \{b, \neg d, e\}\}.$$

If $\mathcal{F}_0 = \mathcal{F}[0/a]$ and $\mathcal{F}_1 = \mathcal{F}[1/a]$, then $\mathcal{F}_0 = \{\{c, b\}, \{e\}\{b, \neg d, e\}\}$ and $\mathcal{F}_1 = \{\{c\}\{\neg d\}\{b, \neg d, e\}\}$. The prime implicate sets of $\mathcal{F}_0$ and of $\mathcal{F}_1$ are therefore $\mathcal{P}_0 = \{\{c, b\}\{e\}\}$ and $\mathcal{P}_1 = \{\{c\}\{\neg d\}\}$, respectively — see Figure 1.

$$\begin{tikzcd}
 & & \mathcal{F} = \{\{a,c,b\},\ \{a,e\},\ \{\neg a,c\},\ \{\neg a,\neg d\},\ \{b,\neg d,e\}\} \arrow[dl, "a=0"'] \arrow[dr, "a=1"] & & \\
 & \mathcal{F}_0 = \{\{c,b\},\ \{e\},\ \{b,\neg d,e\}\} & & \mathcal{F}_1 = \{\{c\},\ \{\neg d\},\ \{b,\neg d,e\}\} & \\
 & \mathcal{P}_0 = \{\{c,b\},\ \{e\}\} \arrow[dl] \arrow[dr] & & \mathcal{P}_1 = \{\{c\},\ \{\neg d\}\} \arrow[dl] \arrow[dr] & \\
\mathcal{P}_0^{\supseteq} = \{\{c,b\}\} & & \mathcal{P}_0^{\not\supseteq} = \{\{e\}\} \quad \mathcal{P}_1^{\supseteq} = \{\} & & \mathcal{P}_1^{\not\supseteq} = \{\{c\},\ \{\neg d\}\}
\end{tikzcd}$$

$$\begin{aligned}
a \vee \mathcal{P}_0^{\not\supseteq} &= \{\{a,e\}\} \\
\neg a \vee \mathcal{P}_1^{\not\supseteq} &= \{\{\neg a,c\},\ \{\neg a,\neg d\}\} \\
\mathcal{P}_0^{\supseteq} \cup \mathcal{P}_1^{\supseteq} &= \{\{c,b\}\} \\
\mathcal{U} &= \{\{e,c\},\ \{e,\neg d\}\}
\end{aligned}$$

$$\mathcal{P}(\mathcal{F}) = \{\{a,e\},\ \{\neg a,c\},\ \{\neg a,\neg d\},\ \{c,b\},\ \{e,c\},\ \{e,\neg d\}\}$$

Fig. 1.

The algorimithic details of computing these sets are presented in Section 5. The process relies heavily on subsumption checking and on formation of clause unions. The next section describes how these operations can exploit the trie data structure.

4 Trie-Based Operations

The original pi-trie algorithm, introduced in [12], employed a routine called PIT, which was based on a branch-by-branch analysis. The improved version, presented in the next section, also contains a routine named PIT. They use similar methods to construct $\mathcal{P}$ from $\mathcal{P}_0$ and $\mathcal{P}_1$, but the new version is based on Theorem 2 and uses the set operations of that theorem. The development is arguably more intuitive, and, more importantly, is more efficient.

One improvement comes from identifying $\mathcal{P}_0^{\supseteq}$ and $\mathcal{P}_1^{\supseteq}$ before considering any clauses as possible members of $\mathcal{U}$. This contrasts with the PIT routine of [12], in which branch by branch subsumption checks are based on prime marks. An unmarked branch can be combined with another to form a possible member of $\mathcal{U}$, only to be eventually discovered to represent a clause in (say) $\mathcal{P}_0^{\supseteq}$. The result is unnecessary subsumption checks.

There is a second improvement, perhaps a surprisingly effective one. Maintaining both $\mathcal{P}_0^{\supseteq}$ and $\mathcal{P}_0^{\not\subseteq}$ makes *prime marks* unnecessary. The original algorithm puts prime marks at some leaf nodes, and checking for their presence requires traversing the branch — almost as expensive as the subsumption check itself.

A third improvement appears to be the most significant: Clause set operations can be realized recursively on entire sets, represented as tries.[5] Experiments indicate that the trie-based operations outperform branch-by-branch operations, and that the advantage increases with the size of the trie.

The following operators on clause sets F and G are defined as set operations, but the pseudocode realizes the operations assuming that the clause sets are represented as ternary tries. Recall that the trie $\mathcal{T}$ can be written $\langle r, \mathcal{T}^+, \mathcal{T}^-, \mathcal{T}^0 \rangle$, where r is the root label of $\mathcal{T}$, and $\mathcal{T}^+$, $\mathcal{T}^-$, and $\mathcal{T}^0$ are the three subtries. Tries with three empty children are called *leaves*.

1. *Subsumed*$(F, G) = \{C \in G \mid C$ is subsumed by some $C' \in F\}$
2. *XUnions*$(F, G) = \{C \cup D \mid C \in F,\ D \in G,\ C \cup D$ is not tautological $\}$

[5] Tries have been employed for (even first order) subsumption [22], but on a clause to trie basis, rather than the trie to trie basis developed here.

```
input  : Two clausal tries $T_1$ and $T_2$
output: $T$, a trie containing all the clauses in $T_2$ subsumed by some clause in $T_1$
if $T_1 = null$ or $T_2 = null$ then
    $T \leftarrow null$ ;
else if leaf($T_1$) then
    $T \leftarrow T_2$;
else
    $T \leftarrow$ new Leaf;
    $T^+ \leftarrow Subsumed(T_1^+, T_2^+) \cup Subsumed(T_1^0, T_2^+)$ ;
    $T^- \leftarrow Subsumed(T_1^-, T_2^-) \cup Subsumed(T_1^0, T_2^-)$ ;
    $T^0 \leftarrow Subsumed(T_1^0, T_2^0)$ ;
    if leaf($T$) then
        $T \leftarrow null$;
    end
end
return $T$;
```

Algorithm 1: *Subsumed*(T_1,T_2)

```
input  : Two clausal tries $T_1$ and $T_2$
output: $T$, a trie of the pairwise unions of the clauses in $T_1$ and $T_2$
if $T_1 = null$ or $T_2 = null$ then
    $T \leftarrow null$ ;
else if leaf($T_1$) then
    $T \leftarrow T_2$;
else if leaf($T_2$) then
    $T \leftarrow T_1$;
else
    $T \leftarrow$ new Leaf;
    $T^+ \leftarrow XUnions(T_1^+, T_2^+) \cup XUnions(T_1^0, T_2^+) \cup XUnions(T_1^+, T_2^0)$ ;
    $T^- \leftarrow XUnions(T_1^-, T_2^-) \cup XUnions(T_1^0, T_2^-) \cup XUnions(T_1^-, T_2^0)$ ;
    $T^0 \leftarrow XUnions(T_1^0, T_2^0)$;
    if leaf($T$) then
        $T \leftarrow null$;
    end
end
return $T$;
```

Algorithm 2: *XUnions*(T_1,T_2)

For convenience and readability, ordinary set union ($\cup$) and subtraction ($-$) have been employed here and in Section 5, but formal algorithmic definitions are not given. Union can easily be implemented recursively for the trie representation, but the resulting performance is improved only slightly over a straightforward iteration on clauses. Subtraction is also straightforward but is always employed with the result of a subsumption test. In practice, it is easier to extract the subsumed branches as a side effect during the subsumption test. Experiments involving subsumption testing are reported below. In Section 5, the improved and original versions of the pi-trie algorithms are compared.

The input for the experiments depicted in Figure 2 is a pair of n-variable CNF formulas, where $n \in \{10, 11, 12, 13, 14, 15\}$, with results averaged over 20 trials for each n. Each formula with n variables has $\lfloor \binom{n}{3}/4 \rfloor$ clauses of length 3, $\lfloor \binom{n}{4}/2 \rfloor$ clauses of length 4, and $\binom{n}{5}$ clauses of length 5. This corresponds to $\frac{1}{32}$ of the $2^k \binom{n}{k}$ possible clauses of length k for $k = 3, 4, 5$.

```
input  : Clause sets $F$ and $G$
output: The clauses in $G$ subsumed by some clause in $F$
$H \leftarrow \emptyset$;
for $C \in F$ do
    for $D \in G$ do
        if $C \subseteq D$ then $H \leftarrow H \cup \{D\}$;
    end
end
return $H$;
```

Algorithm 3: NaiveSubsumed(F,G)

The two clause sets are compiled into two tries for the application of *Subsumed* and into two lists for the application of *NaiveSubsumed*. For *Subsumed*, the runtimes for each n are graphed against the sum of the nodes in both input tries. *NaiveSubsumed* is graphed against the number of literal instances in both input formulas. This takes into account the more compact representation of clause sets provided by tries than by lists. It is thus inaccurate to graph both runtimes against a single parameter.

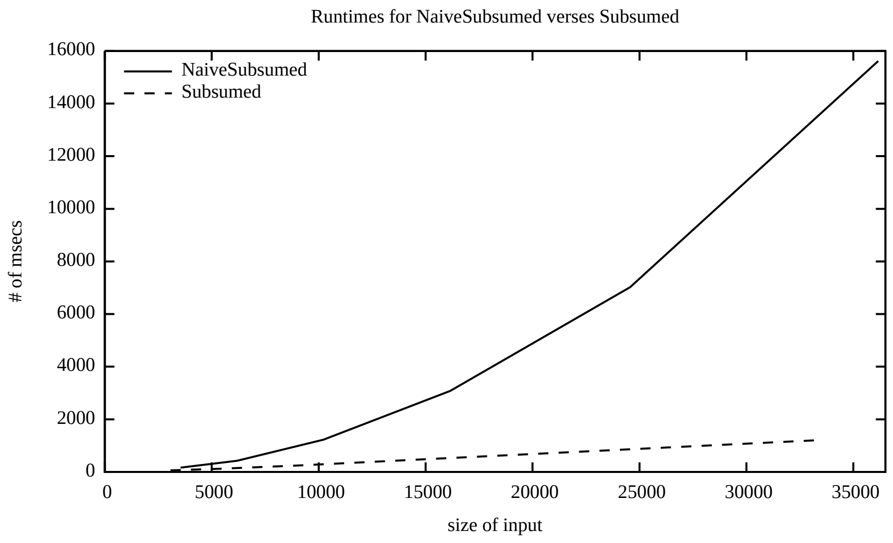

Fig. 2. *Subsumed* vs *NaiveSubsumed*

The ratio of runtimes changes as the input size increases, suggesting that the runtimes of *NaiveSubsumed* and *Subsumed* differ asymptotically; Lemma 7 offers additional evidence. Note first that a full ternary trie of depth h contains $\frac{3^{h+1}-1}{2}$ nodes. Thus *Subsumed*, which operates on two tries, has input size at most $3^{h+1}-1$, where h is the larger of the two heights.

Lemma 7. When applied to two ternary tries of depth at most h and thus combined size at most $n = 3^{h+1}-1$, *Subsumed* runs in time $O(n^{\frac{\log 5}{\log 3}}) \approx O(n^{1.465})$.

Proof. The algorithm *Subsumed* is recursive and operates on five pairs of children at each level, giving the recurrence relation $Z(h) = 5Z(h-1)$ with $Z(0) = 1$ and thus $Z(h) = 5^h$ for the runtime of *Subsumed* with respect to height. Expressing height in terms of size, $n \leq 2 \cdot \frac{3^{h+1}-1}{2} = 3^{h+1}-1$, which implies that $h \leq \log_3 \frac{n+1}{3}$. It follows that if $T(n)$ is the runtime of *Subsumed* as a function of size n, then $T(n) \leq Z(\log_3 \frac{n+1}{3})$. Hence,

$$T(n) = 5^{\log_3 \frac{n+1}{3}} = 5^{\frac{\log_5 \frac{n+1}{3}}{\log_5 3}} = \frac{n+1}{3}^{\frac{1}{\log_5 3}},$$

so that $T(n) = O(n^{\frac{1}{\log_5 3}}) = O(n^{\frac{\log 5}{\log 3}}) \approx O(n^{1.465})$. $\square$

This is less than *NaiveSubsumed*'s obvious runtime of $O(n^2)$ but still more than linear. Lemma 7 is interesting but the general upper bound may be quite different.

5 The Improved *pi*-trie Algorithm

Theorem 2 provides a simpler characterization of the *pi*-trie algorithm than the one developed in [12]. It reveals those subsumption checks that are required, and, by omission, those that are not. Nevertheless, the algorithm is not entirely transparent. However, the basic idea is rather simple. It can be viewed as a standard recursive divide-and-conquer algorithm, where each problem $\mathcal{F}$ is divided into subproblems $\mathcal{F}_0$ and $\mathcal{F}_1$ by substituting for the appropriate variable. The prime implicates $\mathcal{P}_0$ of $\mathcal{F}_0$ and $\mathcal{P}_1$ of $\mathcal{F}_1$ are computed recursively, and the prime implicates $\mathcal{P}$ of $\mathcal{F}$ are computed from these two sets. The set $\mathcal{P}$ is precisely the subsumption minimal subset of the clause set produced by saturating $\mathcal{P}_0$ and $\mathcal{P}_1$ with resolution. However, the algorithm presented in this section computes $\mathcal{P}$ from $\mathcal{P}_0$ and $\mathcal{P}_1$ in a parsimonious manner using Theorem 2.

The base case is when substitution yields a constant, so that $\mathcal{P}_0 = \{\{\}\}$ or $\mathcal{P}_1 = \{\}$. The remainder of the algorithm consists of combining $\mathcal{P}_0$ and $\mathcal{P}_1$ to form $\mathcal{P}$ with a routine called PIT. The version of PIT presented in [12] does subsumption checking branch by branch, whereas in this paper it is performed between entire tries. These trie-based operations — for example, $Subsumed$ and *XUnions* — is what makes the new algorithm more efficient.

```
input  : A boolean formula $\mathcal{F}$ and a list of its variables $V = \langle v_1, \ldots, v_k \rangle$
output: The clause set $\mathcal{P}$ — the prime implicates of $\mathcal{F}$
if $\mathcal{F} = 1$ then
    return $\emptyset$ ; // Tautologies have no prime implicates.
else if $\mathcal{F} = 0$ then
    return $\{\emptyset\}$ ; // $\mathcal{P}(0)$ contains only the empty clause.
else
    $\mathcal{F}_0 \leftarrow \mathcal{F}[0/v_1]$;
    $\mathcal{F}_1 \leftarrow \mathcal{F}[1/v_1]$;
    $V' \leftarrow \langle v_2, \ldots, v_k \rangle$;
    return PIT( prime($\mathcal{F}_0, V'$) , prime($\mathcal{F}_1, V'$) , $v_1$ );
end
```

Algorithm 4: prime($\mathcal{F}$,V)

```
input  : Clause sets $\mathcal{P}_0 = \mathcal{P}(\mathcal{F}_0)$ and $\mathcal{P}_1 = \mathcal{P}(\mathcal{F}_1)$, variable $v$
output: The clause set $\mathcal{P} = \mathcal{P}((v \vee \mathcal{F}_0) \wedge (\neg v \vee \mathcal{F}_1))$
$\mathcal{P}_0^{\supseteq} \leftarrow$ Subsumed($\mathcal{P}_1, \mathcal{P}_0$) ; // Initialize $\mathcal{P}_0^{\supseteq}$
$\mathcal{P}_1^{\supseteq} \leftarrow$ Subsumed($\mathcal{P}_0, \mathcal{P}_1$) ; // Initialize $\mathcal{P}_0^{\supseteq}$
$\mathcal{P}_0^{\not\subseteq} \leftarrow \mathcal{P}_0 - \mathcal{P}_0^{\supseteq}$;
$\mathcal{P}_1^{\not\subseteq} \leftarrow \mathcal{P}_1 - \mathcal{P}_1^{\supseteq}$;
$\mathcal{U} \leftarrow$ XUnions($\mathcal{P}_0^{\not\subseteq}, \mathcal{P}_1^{\not\subseteq}$);
$\mathcal{U} \leftarrow \mathcal{U} -$ SubsumedStrict($\mathcal{U}, \mathcal{U}$);
$\mathcal{U} \leftarrow \mathcal{U} -$ Subsumed($\mathcal{P}_0^{\supseteq}, \mathcal{U}$);
$\mathcal{U} \leftarrow \mathcal{U} -$ Subsumed($\mathcal{P}_1^{\supseteq}, \mathcal{U}$);
return  $((v \vee \mathcal{P}_0^{\not\subseteq}) \cup (\neg v \vee \mathcal{P}_1^{\not\subseteq}) \cup \mathcal{P}_0^{\supseteq} \cup \mathcal{P}_1^{\supseteq} \cup \mathcal{U})$;
```

Algorithm 5: PIT($\mathcal{P}_0$,$\mathcal{P}_1$,v)

Figure 3 compares the *pi*-trie algorithm from [12] to the updated version using the recursive $Subsumed$ and $XUnion$ operators. The input for both algorithms is 15-variable 3-CNF formulas with varying numbers of clauses, and the runtimes are averaged over 20 trials. The great discrepancy between runtimes requires that they be presented in log scale; it is explained in part by Figure 2, which compares the runtime of *Subsumed* to Algorithm 5, a naïve subsumption algorithm. The performance of the two systems converges as the number of clauses increases. With more clauses, formulas are unsatisfiable with probability approaching 1. As a result, the base cases of the prime algorithm are encountered early, and subsumption in the PIT routine plays a less dominant role, diminishing the advantage of the improved algorithm.

6 Selective Computation

There are situations in which it is desirable to restrict attention to a subset of prime implicates having a certain property, for example, being positive or having length at most

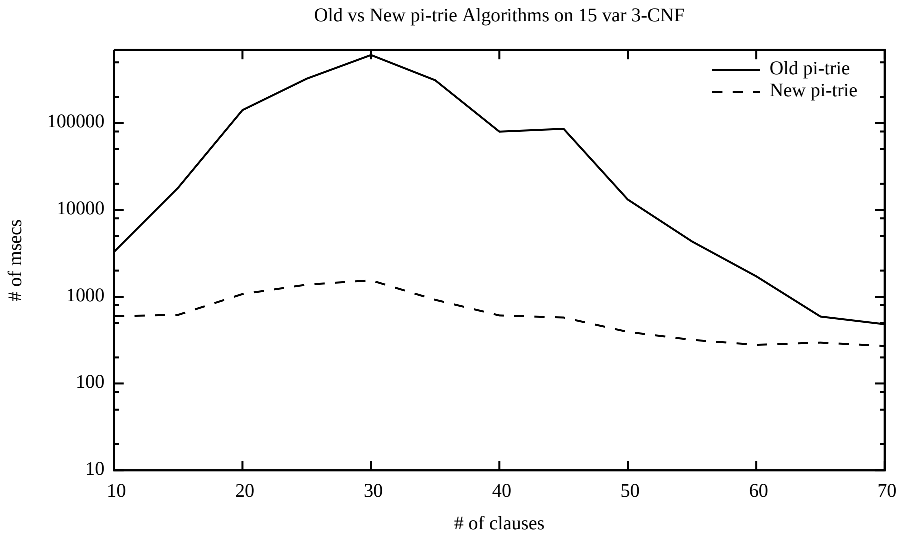

Fig. 3. Old vs Improved *pi*-trie algorithm

four. The desired subset can always be selected from the entire set of prime implicates, but generating only the prime implicates of interest is potentially more efficient.

6.1 Subset/Superset Invariance

A property Q of sets is said to be *superset invariant* if whenever $B \supseteq A$, $Q(A) \rightarrow Q(B)$; Q is *subset invariant* if whenever $B \subseteq A$, $Q(A) \rightarrow Q(B)$. The complement property, denoted $\overline{Q}$, is defined in the obvious way: $\overline{Q}(X) = \neg Q(X)$. The following lemma is immediate.

Lemma 8. If Q is superset (subset) invariant, then $\overline{Q}$ is subset (superset) invariant. $\Box$

Clauses (and prime implicates) are sets of literals. Thus, *containing no more than three literals*, *containing only positive literals*, and *being a Horn clause* are examples of subset invariant properties of clauses; *containing the variable* p_3 and *containing both positive and negative literals* are examples of superset invariant properties. *Having an equal number of positive and negative literals* is neither subset nor superset invariant.

It turns out that the pi-trie algorithm is particularly amenable to tuning for generation of only prime implicates satisfying subset invariant properties. The reason is that clauses computed at any stage of the algorithm always contain as subsets clauses computed at an earlier stage. Thus any clause with the desired subset invariant property cannot be produced from an earlier clause having the complement property. Looked at in another way, the complement property is superset invariant, so supersets of clauses having the unwanted property also have the unwanted property. Thus partially constructed branches that persist lead only to supersets of themselves, and so partial branches having the complement property can "grow" only into larger branches that also have the complement property. In particular, no branch with the desired property is lost by pruning branches with the complement property.

6.2 Pruning for Subset Invariant Properties

Let Q be a property defined on clauses. Then the *filter of Q*, denoted ϕ_Q, is the operation that produces the subset of S that satisfies Q:

$$\phi_Q(S) = \{C \in S \mid Q(C)\} .$$

Lemma 9 is immediate.

Lemma 9. Given a property Q and a set S of clauses, $\phi_Q(\phi_Q(S)) = \phi_Q(S)$; i.e., $\phi_Q(S)$ is a fixed point of ϕ_Q. $\square$

For the remainder of the paper, any property under discussion will be assumed to hold for some set of clauses and therefore, if subset invariant, to hold for the empty clause. The pi-trie algorithm is particularly amenable to being tuned to generate only prime implicants satisfying subset invariant properties.

Lemma 10. Let Q be a subset invariant property, and let $S1$ and $S2$ be clause sets. Then $\phi_Q(\mathit{Subsumed}(S1, S2)) = \mathit{Subsumed}(\phi_Q(S1), \phi_Q(S2))$

Proof. Suppose $C \in \phi_Q(\mathit{Subsumed}(S1, S2))$. Then C has property Q by definition of ϕ_Q. Also, by definition of $\mathit{Subsumed}$, $C \in S2$ and is subsumed by some clause C' in $S1$. Since Q is subset invariant, C' has property Q. So $C \in \phi_Q(S2)$ and $C' \in \phi_Q(S1)$. Thus by definition, $C \in \mathit{Subsumed}(\phi_Q(S1), \phi_Q(S2))$.

If $C \in \mathit{Subsumed}(\phi_Q(S1), \phi_Q(S2))$, then $C \in \phi_Q(S2))$ and there is a clause C' in $\phi_Q(S1)$ that subsumes C; both C and C' have property Q; $C \in S1$ and $C' \in S2$. So $C \in \mathit{Subsumed}(S1, S2)$, C has property Q, and thus $C \in \phi_Q(\mathit{Subsumed}(S1, S2))$. $\square$

Corollary 1. If Q is a subset invariant property and $S1$ and $S2$ are clause sets, then $\phi_Q(\phi_Q(\mathit{Subsumed}(S1, S2))) = \phi_Q(\mathit{Subsumed}(\phi_Q(S1), \phi_Q(S2)))$ and so from Lemma 9, $\phi_Q(\mathit{Subsumed}(S1, S2)) = \phi_Q(\mathit{Subsumed}(\phi_Q(S1), \phi_Q(S2)))$.

Corollary 2. Lemma 10 and Corollary 1 hold with $\mathit{SubsumedStrict}$ in place of $\mathit{Subsumed}$.

The corollaries capture a crucial property of the $\mathit{Subsumed}$ and $\mathit{SubsumedStrict}$ operators. For any two sets $S1$ and $S2$, the clauses produced by these operators that satisfy Q are determined entirely by those members of $S1$ and of $S2$ that satisfy Q. The next four lemmas establish this property for an additional four operations.

Lemma 11. Let Q be a subset invariant property, and let $S1$ and $S2$ be clause sets. Then $\phi_Q(S1 - S2) = \phi_Q(S1) - \phi_Q(S2)$.

Proof. Suppose $C \in \phi_Q(S1 - S2)$. Then $C \in S1 - S2$, $C \in S1$, $C \notin S2$, and C has property Q. Therefore, $C \in \phi_Q(S1)$ and $C \notin \phi_Q(S2)$, and so $C \in \phi_Q(S1) - \phi_Q(S2)$.

The proof that $\phi_Q(S1) - \phi_Q(S2) \subseteq \phi_Q(S1 - S2)$ is similar. $\square$

Corollary. If Q is a subset invariant property and $S1$ and $S2$ are clause sets, then $\phi_Q(S1 - S2) = \phi_Q(\phi_Q(S1) - \phi_Q(S2))$.

The proof of the next lemma is similar to that of Lemma 11.

Lemma 12. Let Q be a subset invariant property, and let $S1$ and $S2$ be clause sets. Then $\phi_Q(S1 \cup S2) = \phi_Q(S1) \cup \phi_Q(S2)$. □

Corollary: $\phi_Q(S1 \cup S2) = \phi_Q((\phi_Q(S1) \cup \phi_Q(S2))$.

Lemma 13. Let Q be a subset invariant property, and let $S1$ and $S2$ be clause sets. Then $\phi_Q(\textit{XUnions}(S1, S2)) = \phi_Q(\textit{XUnions}(\phi_Q(S1), \phi_Q(S2)))$.

Proof. Suppose $C \in \phi_Q(\textit{XUnions}(S1, S2))$. Then $C \in \textit{XUnions}(S1, S2)$ and has property Q. Also, $C = C1 \cup C2$, where $C1 \in S1$ and $C2 \in S2$. Since Q is subset invariant, both $C1$ and $C2$ have property Q. As a result, $C1 \in \phi_Q(S1)$ and $C2 \in \phi_Q(S2)$, Thus $C = C1 \cup C2 \in \textit{XUnions}(\phi_Q(S1), \phi_Q(S2))$, and because C has property Q, $C \in \phi_Q(\textit{XUnions}(\phi_Q(S1), \phi_Q(S2)))$.

The proof that $\phi_Q(\textit{XUnions}(\phi_Q(S1), \phi_Q(S2))) \subseteq \phi_Q(\textit{XUnions}(S1, S2))$ is similar. □

Note that if $\textit{XUnions}(\phi_Q(S1), \phi_Q(S2))$ is left unfiltered, it is not equal to $\phi_Q(\textit{XUnions}(S1, S2))$ since *XUnions* does not preserve a subset invariant property.

The next lemma completes the analysis of the interaction between filtering and clause set operations in PIT.

Lemma 14. Let Q be a subset invariant property, let G be a clause set, and let ℓ be a literal. Then $\phi_Q(\ell \wedge S) = \phi_Q(\ell \wedge \phi_Q(S))$.

Proof. Suppose $C \in \phi_Q(\ell \wedge S)$; $C - \{\ell\}$ is in S and has property Q, so $C - \{\ell\} \in \phi_Q(S)$. Thus $C \in \phi_Q(\ell \wedge \phi_Q(S))$. By a similar argument, $\phi_Q(\ell \wedge \phi_Q(S)) \subseteq \phi_Q(\ell \wedge S)$. □

Theorem 3 is essentially Theorem 3 from [13], but the complete proof is presented here.

Let PIT^Q be the PIT routine modified with a new return statement:

return $\quad \phi_Q(\ (v \wedge \mathcal{P}_1^{\not\supseteq})\ \cup\ (\neg v \wedge \mathcal{P}_0^{\not\supseteq})\ \cup\ \mathcal{P}_0^{\supseteq}\ \cup\ \mathcal{P}_1^{\supseteq}\ \cup\ \mathcal{U}\ )\ ;$

Let prime^Q to be the prime routine except that PIT^Q is employed in place of PIT, and the recursive calls to prime become calls to prime^Q.

Theorem 3. Let Q be a subset invariant property, and let $\mathcal{F}$ be a propositional formula with variable set V. Then ϕ_Q(prime($\mathcal{F}, V$)) = prime$^Q(\mathcal{F}, V)$.

Proof. Since ϕ_Q(prime($\mathcal{F}, V$)) is the set of prime implicants of $\mathcal{F}$ with property Q, it must be shown that prime$^Q(\mathcal{F}, V))$ is also this set; proceed by induction on n, the number of invocations of PIT^Q.

If $n = 0$, then only the base cases of prime^Q apply, and the resulting clause set is either empty or contains only the empty clause, which has property Q.

Assume now that the theorem holds for n invocations of PIT^Q, and suppose that $n+1$ are required. In the return statement of the last invocation, $\mathcal{P}_1^{\not\supseteq}$ and $\mathcal{P}_1^{\supseteq}$ have been computed with at most n invocations of PIT^Q. By the induction hypothesis, their union is the set of prime implicates of $\mathcal{F}_1$ having property Q. The same holds for $\mathcal{P}_0^{\not\supseteq}$ and $\mathcal{P}_0^{\supseteq}$ with respect to $\mathcal{F}_0$. Finally, Lemmas $10-14$ and Theorems 2 and 3 imply the filtered clauses being returned are precisely the prime implicates of $\mathcal{F}$ having property Q. $\square$

Restricted sets of prime implicates are useful in several settings. One is abductive reasoning, where prime implicates are relevant to the problem of finding an explanation $\mathcal{E}$ of an observation $\mathcal{O}$ in light of a given theory Σ. Typically, an explanation is required to be non-redundant in the sense that removal of any one literal invalidates the explanation. This can be assured by restricting attention to prime implicates of Σ. However, certain explanations may be preferred; for example, shortest or positive explanations are often desirable. Having at most a given number of literals is a subset invariant property, and so is having only positive literals.

Another preference may be that an explanation is sought in which attention is restricted to a certain set of propositions, say $p_1, \ldots, p_k$. Having only literals involving these variables is a subset invariant property. By placing these variables first in the ordering, substanative work involving the PIT routine occurs in only the first k levels of the pi-trie. Branches that may arise beyond this level will all be pruned, and the subsumption operations of PIT will never execute. The resulting computation is handled entirely by base cases. In fact, the node labeled by p_k remains if and only if all branches of the recursion leading away from it return zero. In other words, the formula under consideration at p_k is $\Sigma[\alpha_1/p_1, \alpha_2/p_2, \ldots, \alpha_k/p_k]$, where $\alpha_i \in \{0,1\}$, and it is sufficient to run a SAT solver on this formula.

Perhaps surprisingly, prime implicates turn out to be useful in the systhesis of correct code for systems having *polychronous data flow* (both synchronous and asynchronous data flow). In [8], a *calculus of epochs* is developed that provides scheduling computations at individual threads and at the correct synchronization points among threads. The proposed epoch calculus requires computations of prime implicates of formulas that arise from the relationships among *instants* at which events occur. Prime implicates in this setting must be positive — no negative literals present — and the most useful ones contain as few literals as possible. Positive, unit prime implicates are particularly useful. Of course, these are all subset invariant properties.

6.3 Pruning for Superset Invariant Properties

Containing both positive and negative literals is an example of a superset invariant property for which there is no obvious way to prune the search space to produce only prime implicates with that property. It is true that if a partially constructed branch has this property, then final branches that arise containing it will also have the property. But the difficulty is that partial branches not having this property cannot be pruned because they might give rise to branches with the property. It appears to be necessary to compute the full pi-trie and then cull for the desired prime implicates.

There may be better approaches for some superset invariant properties. Consider the property of containing a specific literal p. Suppose p to be positive and the first variable in the formula $\mathcal{F}$. Let $\mathcal{T}$ be the pi-trie of $\mathcal{F}$, and let its three branches be $\mathcal{T}^+$, $\mathcal{T}^-$, and $\mathcal{T}^0$. Then the set of prime implicates containing p are represented precisely by $\mathcal{T}^+$. Unfortunately, $\mathcal{T}^+$ cannot be computed without computing $\mathcal{T}^0$ since some prime implicates of $\mathcal{F}[1/p]$ may be subsumed by branches in $\mathcal{T}^0$. Nonetheless, significant computational savings are possible. First, the PIT function need not actually build the third subtrie $\mathcal{T}^0$. Moreover, branches in $\mathcal{T}^+$ are checked only for being subsumed by branches in $\mathcal{T}^-$, not for whether they subsume branches in $\mathcal{T}^-$. Additionally, the subsumption checks normally done between branches in $\mathcal{T}^0$ may be skipped.

More generally, suppose that the prime implicates of interest are those that contain literals $l_1, l_2, \ldots, l_k$. This is of course superset invariant. The corresponding k variables may be put first in the ordering. Then, designated prime implicates are all found in the subtrie of the pi-trie for $\mathcal{F}$ at the end of the path labeled by these literals.

The subtrie computed initially that is rooted at l_k is the pi-trie for $\mathcal{F}[0/l_1, 0/l_2, \ldots, 0/l_k]$. Some of the branches in this subtrie represent, along with $l_1, \ldots, l_{k-1}$, implicates of $\mathcal{F}$ that are not prime implicates because l_k (and possibly other literals) is redundant. This is what is normally discovered in the PIT routine running on the subtrie rooted at l_{k-1}. In other words, to remove these non-prime implicates, the sibling subtrie rooted at $\overline{l_k}$ is required. A non-prime branch would normally be moved (without its l_k root) to the zero sibling of these two subtries. As with the single variable case, this zero subtrie need not actually be built; subsumption is checked in only one direction between the first two subtries, and the subsumption tests required within the third can be skipped.

To summarize, subsumption can be restricted to one direction between the pairs of subtries $(l_1, \overline{l_1}), (l_2, \overline{l_2}), \ldots, (l_k, \overline{l_k})$. For $1 \leq i \leq k$, branches in the subtrie rooted at l_i are checked only for being subsumed by branches in the subtrie rooted at $\overline{l_i}$. Furthermore, the additional work required to build the zero sibling subtries of each of these k pairs can be skipped. This avoids building one zero subtrie at each of the first k levels of the trie — see Figure 4. Were the trie balanced with N levels, it would have 3^N nodes, and building R of these can be avoided, where $1/3 \leq R = \sum_{i=1}^{k} 1/3^i < 1/2$.

7 Experiments with Subset Invariant Properties

The improved pi-trie algorithm is implemented in Java with two subset invariant filters: maximum clause length and designated forbidden literals. Its performance has been compared with two others.[6] The first comes from a system of Zhuang, Pagnucco and Meyer [26] that implements belief revision using prime implicates. Their system contains a routine that generates prime implicates from CNF clause sets. This routine, also in Java, was isolated and used without alteration; it will be referred to as **BRPrime**.

[6] It is surprisingly difficult to find publicly available prime implicate generators. Substantial email inquiries based on publications revealed only the system of Zhuang, Pagnucco and Meyer [26] that implements belief revision using prime implicates. That system was much less efficient than a simple prototype implemented by the first author, which in turn was much less efficient than the original pi-trie algorithm, available at http://www.cs.albany.edu/ritries. (A public release of the improved algorithm is under development.)

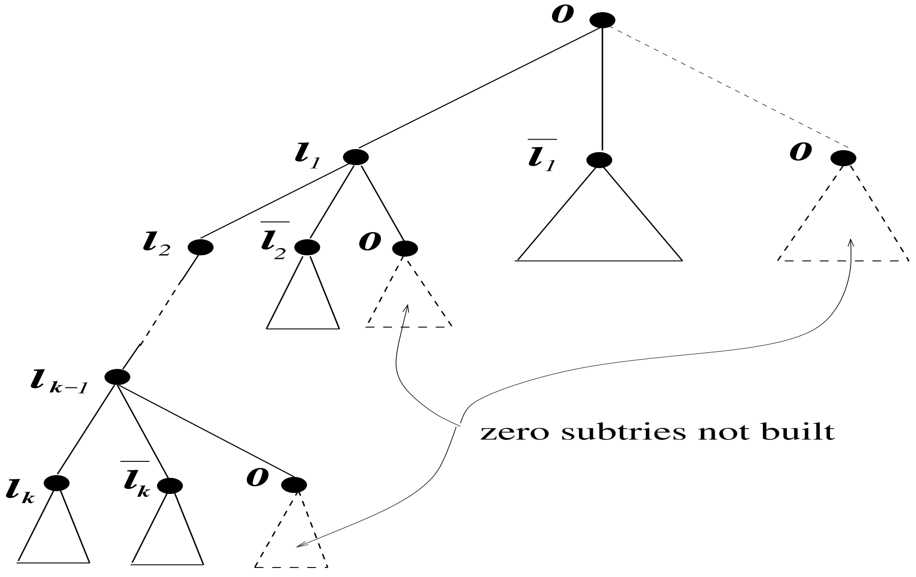

Fig. 4. Building the pi-Trie for Prime Implicates Having $\mathbf{l_1, \ldots, l_k}$.

The **BRPrime** routine generates primes from a list L of clauses by maintaining a list P of candidates for primality. The first clause C in L is checked for subsumption with each clause in P. Any subsumed clauses are removed from P, and if any prime candidate subsumes C, C is discarded. If C is not subsumed, it is appended to P, and all resolvents of C with clauses from P are appended to L. The process terminates when L is empty or when the empty clause is deduced.

The second comparison was made with a somewhat simpler resolution-based prime implicate generator called **ResPrime**. Also implemented in Java, it exhaustively iterates resolution and subsumption checking.

The input for all experiments is a variable number of random 3-CNF clauses from a constant alphabet size — see [21] for a good experimental analysis of prime implicates in random 3-CNF. Figure 5 shows a comparison of runtimes of **BRPrime**, **ResPrime**, pi-trie, and pi-trie filtered to exclude clauses of length greater than 2.

The experiments indicate that **BRPrime** is considerably slower than the others, especially on formulas with more clauses. Figure 6 reveals a bit more detail from the same data by removing **BRPrime**. **ResPrime** outperforms the pi-trie algorithm on small input, when the number of clauses is small, which is to be expected. The number of possible resolutions on small clause sets is correspondingly small, whereas the pi-trie algorithm does not assume the input to be CNF and thus does not take advantage of this syntactic regularity. Probabilistically, as the number of clauses increases the formulas become less and less satisfiable, and so in general the recursive substitution that drives the pi-trie algorithm requires fewer substitutions to reach a contradiction, allow-

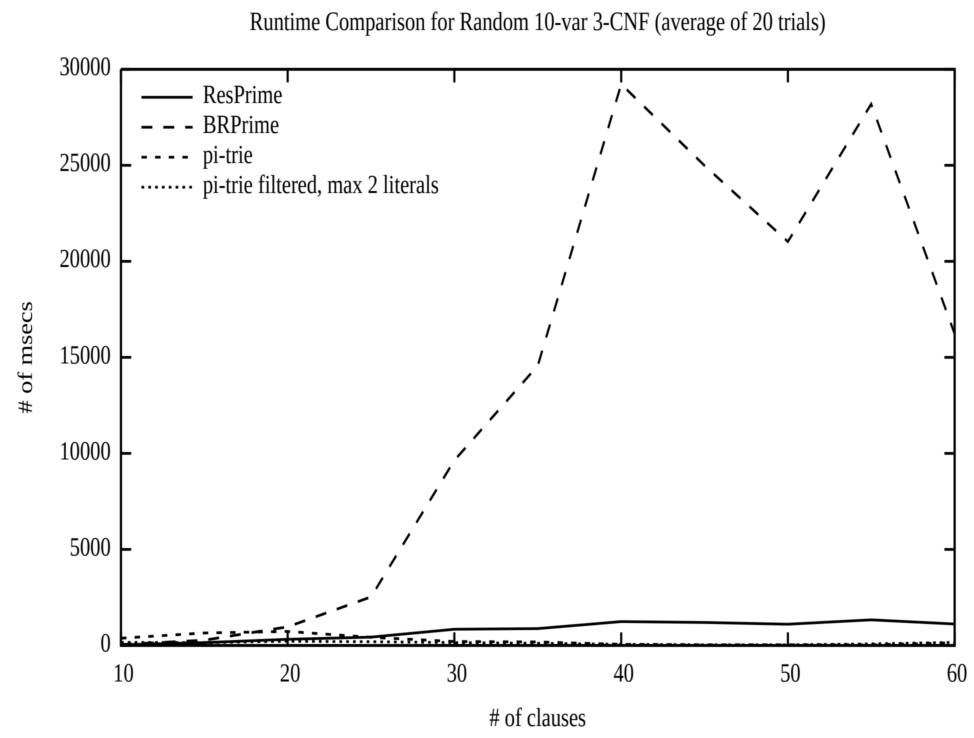

Fig. 5. 10 var 3-CNF

ing faster runtimes. At the same time, the resolution algorithm is required to process more and more clauses and their resolution pairs, and so runs slower.

The pi-trie algorithm with filtering offers dramatic improvements in performance on searches for subset invarient prime implicates. Figure 7 has the results of an experiment with a 13-variable 3-CNF formula. Two filters are used: The first is *max length 2*, and the second is exclusion of clauses containing any of the literals v_3, v_5, v_6, or $\neg v_7$.

Significant efficiency gains are obtained by applying the filter during generation (as opposed to generating all prime implicates and then filtering, as most other algorithms require). In fact, besides [11], which uses a novel integer linear programming approach to discover small prime implicants (easily transformed to a prime implicate algorithm via duality), the authors are aware of only one algorithm that allows filtering during generation of prime implicates based on size or on designated literals. That is the algorithm of Simon and del Val[23], which makes use of zero-based binary decision diagrams (ZBDDs). Those techniques may be applicable to the pi-trie algorithm and are being studied.

The results described in Figure 7 for formulas having 20 clauses are noteworthy: The raw pi-trie algorithm averaged 22,430 msecs, whereas the “max length 2” filter averaged only 1,170 msecs. One reason for the runtime advantage is that 13-variable

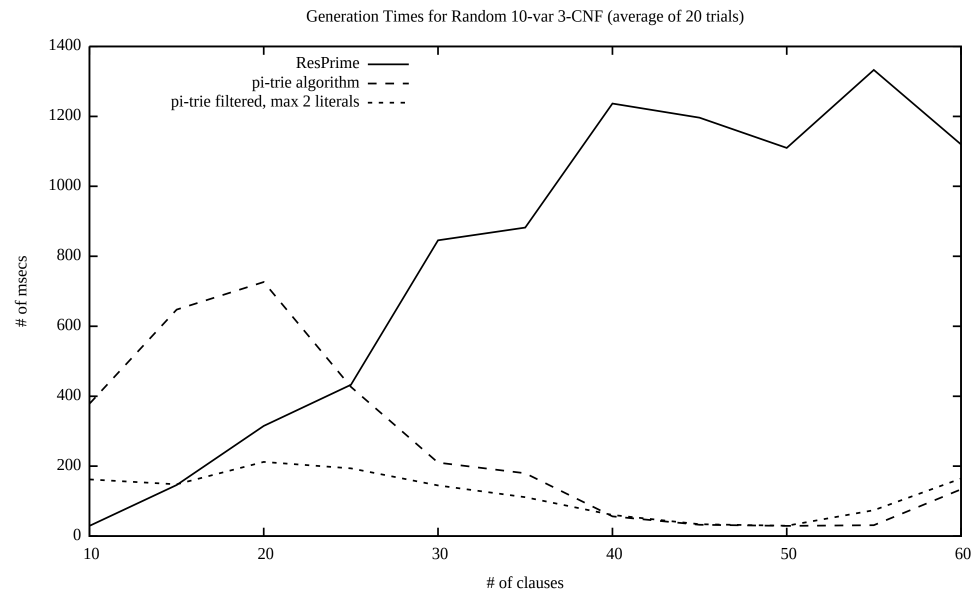

Fig. 6. 10 var 3-CNF

20-clause formulas have relatively long prime implicates. The average number of prime implicates is 213.3, while the average number with length at most 2 is 0.7.

Generating all prime implicates is considerably more difficult (time-wise) than deciding satisfiability: The number of prime implicates of a formula, and thus the size of the output, is typically exponential [2]. As a result, these experiments were performed on relatively small input compared to the examples that modern DPLL-based SAT solvers can handle.

Suppose now that a filter is applied that prunes all prime implicates having length greater than some fixed k. Then branches whose length is greater than k are never built. If n is the number of variables, the number of branches of length k or less is at most $\binom{n}{k} \cdot 3^k$, which is polynomial in n of degree at most k. Since all data used by the algorithm is stored in the trie, this proves

Theorem 4. Let $\mathcal{F}$ be a formula with n variables, and let Q be a subset invariant property satisfied only by clauses of length k or less. Then if the pi-trie algorithm employs PIT^Q, it runs in polynomial space. $\square$

Placing an upper bound on length or restricting prime implicates to those containing only literals from a given set will thus allow the pi-trie algorithm to run in polynomial space. Note that for any one problem instance, a restriction to designated literals amounts to prohibiting the complement set of literals. But if the problem is parameterized on the size of the complement set, polynomial space computation does not result because, for a fixed k, as n increases, so does admissible clause length.

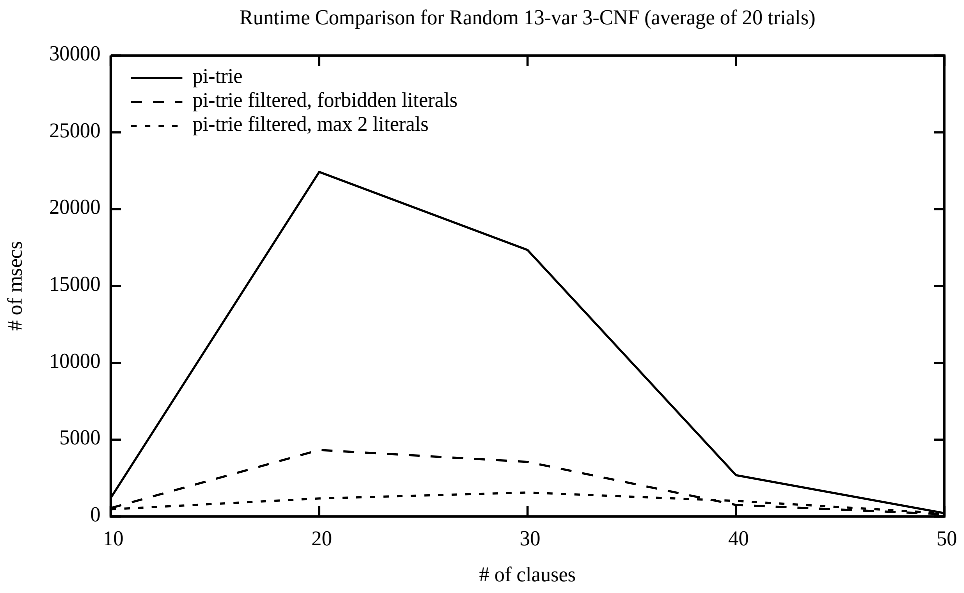

Fig. 7. *pi*-trie Filtering

8 Funding

This work was supported by the National Science Foundation [IIS-0712849, IIS-0712752]; and the Air Force Research Laboratory [FA8750-10-1-0206 – Subaward 450044-19191].

References

1. Guilherme Bittencourt. Combining syntax and semantics through prime form representation. *Journal of Logic and Computation*, 18:13–33, 2008.
2. A. Chandra and G. Markowsky. On the number of prime implicants. *Discrete Mathematics*, 24:7–11, 1978.
3. O. Coudert and J. Madre. Implicit and incremental computation of primes and essential implicant primes of boolean functions. In *29th ACM/IEEE Design Automation Conference*, pages 36–39, 1992.
4. J. de Kleer. An improved incremental algorithm for computing prime implicants. In *Proc. AAAI-92*, pages 780–785, San Jose, CA, 1992.
5. E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.
6. Peter Jackson. Computing prime implicants incrementally. In *Proc.* 11^{th} *International Conference on Automated Deduction, Saratoga Springs, NY, June, 1992*, pages 253–267, 1992. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 607.
7. Peter Jackson and J. Pais. Computing prime implicants. In *Proc.* 10^{th} *International Conference on Automated Deductions, Kaiserslautern, Germany, July, 1990*, volume 449, pages 543–557, 1990. In Lecture Notes in Artificial Intelligence, Springer-Verlag, Vol. 449.
8. B.A. Jose, S.K. Shukla, H.D. Patel, and J.P. Talpin. On the deterministic multi-threaded software synthesis from polychronous specifications. In *Formal Models and Methods in Co-Design (MEMOCODE'08), Anaheim, California, June 2008*, 2008.
9. A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9:185–206, 1990.
10. V.M. Manquinho, P.F. Flores, J.P.M. Silva, and A.L. Oliveira. Prime implicant computation using satisfiability algorithms. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, Newport Beach, U.S.A., November, 1997*, pages 232–239, 1997.
11. Joao P. Marques-Silva. On computing minimum size prime implicants. In *in International Workshop on Logic Synthesis*, 1997.
12. A. Matusiewicz, N.V. Murray, and E. Rosenthal. Prime implicate tries. In *Proceedings of the International Conference TABLEAUX 2009 - Analytic Tableaux and Related Methods, Oslo, Norway, July 2009*, pages 250–264, 2009. In Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol. 5607.
13. A. Matusiewicz, N.V. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. In *Proc. International Symposium on Methodologies for Intelligent Systems - ISMIS, Warsaw, Poland, June, 2011*, 2011. Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol 6804, 203-213.
14. D.R. Morrison. Patricia — practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
15. N.V. Murray and E. Rosenthal. Efficient query processing with compiled knowledge bases. In *Proc. International Conference TABLEAUX 2005 – Analytic Tableaux and Related Methods, Koblenz, Germany, September 2005*, pages 231–244, 2005. In Lecture Notes in Artificial Intelligence, Springer-Verlag, Vol. 3702.
16. T. Ngair. A new algorithm for incremental prime implicate generation. In *Proc. IJCAI-93, Chambery, France, (1993)*, 1993.
17. W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.
18. Anavai Ramesh, George Becker, and Neil V. Murray. Cnf and dnf considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18(3):337–356, 1997.

19. Ray Reiter and Johan de Kleer. Foundations of assumption-based truth maintenance systems: preliminary report. In *Proc. 6th National Conference on Artificial Intelligence, Seattle, WA, (July 12-17, 1987)*, pages 183–188, 1987.
20. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
21. Robert Schrag and James M. Crawford. Implicates and prime implicates in random 3-SAT. *Artificial Intelligence*, 81(1-2):199–222, 1996.
22. Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In *Proceedings of the IJCAR 2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland, July 2004*, 2004.
23. Laurent Simon and Alvaro del Val. Efficient consequence finding. In *IJCAI'01, Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 359–365, 2001.
24. J. R. Slagle, C. L. Chang, and R. C. T. Lee. A new algorithm for generating prime implicants. *IEEE transactions on Computers*, C-19(4):304–310, 1970.
25. T. Strzemecki. Polynomial-time algorithm for generation of prime implicants. *Journal of Complexity*, 8:37–63, 1992.
26. Zhi Qiang Zhuang, Maurice Pagnucco, and Thomas Meyer. Implementing iterated belief change via prime implicates. In Mehmet A. Orgun and John Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 507–518. Springer, 2007.