Trie Based Subsumption and Improving the pi-Trie Algorithm*

Andrew Matusiewicz
Institute of Informatics, Logics, and Security Studies
Department of Computer Science
University at Albany
Albany, NY 12222
a_matusiewicz@cs.albany.edu
, Neil V. Murray
Institute of Informatics, Logics, and Security Studies
Department of Computer Science
University at Albany
Albany, NY 12222
nvm@cs.albany.edu
and Erik Rosenthal
Institute of Informatics, Logics, and Security Studies
Department of Computer Science
University at Albany
Albany, NY 12222
erikr@cs.albany.edu

Abstract

An algorithm that stores the prime implicates of a propositional logical formula in a trie was developed in [10]. In this paper, an improved version of that pi-trie algorithm is presented. It achieves its speedup primarily by significantly decreasing subsumption testing. Preliminary experiments indicate the new algorithm to be substantially faster and the trie based subsumption tests to be considerably more efficient than the clause by clause approach originally employed.

1 Introduction

Prime implicants were introduced by Quine [13] 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 [7]. All prime implicate algorithms of which the authors are aware make extensive use of clause set subsumption; improvements in both the pi-trie algorithm and its core subsumption operations are therefore relevant to all such applications.

Numerous algorithms have been developed to compute prime implicates — see, for example, [1, 2, 3, 5, 6, 8, 9, 12, 14, 18, 19]. Most use clause sets or truth tables as input, but rather few allow arbitrary formulas, such as the pi-trie algorithm introduced in [10]. This recursive algorithm stores the prime implicates in a trie — i.e., a labeled tree — and has a number of interesting properties, including the property 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 way the recursion assembles branches admits variations of the algorithm that compute only restricted sets of prime implicates, such as all positive or not containing specific variables. These variations significantly prune the search space during the computation, and experiments indicate that significant speedups are obtained. In this paper, the algorithm is enhanced while these properties are retained.

The primary improvement developed here is the elimination of unnecessary subsumption tests. This is accomplished by performing subsumption checks between tries whose branches represent clause sets. Experiments indicate the trie-based subsumption tests to be far superior to the clause by clause approach originally employed. This in turn yields a substantially faster pi-trie algorithm.

*This research was supported in part by the National Science Foundation under grants IIS-0712849 and IIS-0712752.

Basic terminology and the fundamentals of pi-tries are summarized in Section 2. The analysis that leads to the new pi-trie algorithm is developed in Section 3, and the trie-based set operations and experiments with them are described in Section 4. Finally, the new pi-trie algorithm and the results of experiments that compare the new algorithm with the original are presented in Section 5.

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.[1] Clauses are often referred to as sets of literals. 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, while, since a contradiction implies any clause, 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 [4]; a variation was introduced by Morrison in 1968 [11]. It is a tree in which each branch represents the sequence of symbols labeling the nodes[2] on that branch, in descending order. Tries have been used in a variety of settings, including representation of logical formulas — see, for example, [15]. 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 no possibility of confusion, the term *branch* will often be used for the clause it represents. Further, it will 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. The ternary representation will generally be assumed in this paper.

The reader is assumed to be familiar with resolution and subsumption [16]; the observations and Lemma 1 are 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 equivalent to the conjunction of its prime implicates.

[1] The term *clause* is also used for a conjunction of literals, especially with *disjunctive normal form*.

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

Let *resolution-subsumption* be the operation on clause sets defined by a single resolution step followed by removal of all subsumed clauses. Define a clause set $\mathcal{S}$ to be $prime$ if $\mathcal{S} = \mathcal{P}(\mathcal{F})$ for some logical formula $\mathcal{F}$; equivalently, $\mathcal{S} = \mathcal{P}(\mathcal{S})$.

Lemma 1. A clause set $\mathcal{S}$ is a fixed point of resolution-subsumption iff $\mathcal{S}$ is prime. □

3 Prime Implicates under Truth-Functional Substitution

The pi-trie algorithm performs the recursion by substituting truth constants for variables to reduce the number of variables; this section contains an analysis of the relationship among $\mathcal{P}(\mathcal{F})$, $\mathcal{P}(\mathcal{F}[\alpha/v])$, and $(\mathcal{P}(\mathcal{F}))[\alpha/v]$, where $\alpha = 0, 1$, and v is a variable occurring in $\mathcal{P}(\mathcal{F})$. Partition $\mathcal{P}(\mathcal{F})$ into clause sets $\mathcal{S}^-$, $\mathcal{S}^+$, and $\mathcal{R}$, where $\mathcal{S}^-$ has the clauses containing $\neg v$, $\mathcal{S}^+$ has the clauses containing v, and the clauses of $\mathcal{R}$ contain neither. Observe that $\mathcal{P}(\mathcal{F})[1/v] = (N[1/v] \cup P[1/v] \cup \mathcal{R}[1/v])$. Then $\mathcal{S}^+[1/v] = \emptyset$ and $\mathcal{R}[1/v] = \mathcal{R}$. Also, $\mathcal{Q} = N[1/v]$ can be obtained by removing all occurrences of $\neg v$ from the clauses of $\mathcal{S}^-$. Let $\tilde{\mathcal{R}}$ be the the clauses in $\mathcal{R}$ not subsumed by any clause in $\mathcal{Q}$. The next lemma uses this notation.

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

Proof. Note first that, since $\mathcal{S}^+[1/v] = \emptyset$, $\mathcal{P}(\mathcal{F}[1/v])$ is logically equivalent to $\mathcal{Q} \cup \tilde{\mathcal{R}}$. Thus, by Lemma 1, it suffices to show the latter to be a fixed point under resolution-subsumption. Since $\tilde{\mathcal{R}} \subseteq \mathcal{R} \subseteq \mathcal{P}(\mathcal{F})$, no clause in $\tilde{\mathcal{R}}$ can subsume any other clause in $\tilde{\mathcal{R}}$. The same is true for $\mathcal{Q}$ since it is true for $\mathcal{S}^-$, and the clauses of $\mathcal{Q}$ are obtained from the clauses of $\mathcal{S}^-$ by removing $\neg v$.[3] To complete the proof, it suffices to show that resolution can produce only a subsumed clause. There are two cases to consider: resolving two clauses from $\tilde{\mathcal{R}}$ and resolving with at least one clause from $\mathcal{Q}$.

Case 1. Let C be the resolvent of two clauses in $\tilde{\mathcal{R}}$. 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 $\mathcal{R}$ and thus is either in $\tilde{\mathcal{R}}$ or is subsumed by a clause in $\mathcal{Q}$.

Case 2. Let C be the resolvent of two clauses with at least one from $\mathcal{Q}$. 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 $\tilde{C} \in N$, so $\tilde{C} - \{\neg v\}$ is in $\mathcal{Q}$ and subsumes C. Otherwise, $\tilde{C} \in \mathcal{R}$, and the analysis of Case 1 applies. □

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

Corollary. Partition $\mathcal{P}(\mathcal{F})$ into three sets: $\mathcal{S}^-$, the clauses containing $\neg v$, $\mathcal{S}^+$, the clauses containing v, and $\mathcal{R}$, the clauses containing neither. Let $\mathcal{Q} = \mathcal{S}^+[0/v] = \{C - \{v\} \mid C \in \mathcal{S}^+\}$, and let $\tilde{\mathcal{R}}$ be the clauses in $\mathcal{R}$ not subsumed by any clause in $\mathcal{Q}$. Then $\mathcal{P}(\mathcal{F}[0/v]) = \mathcal{Q} \cup \tilde{\mathcal{R}}$.

For the remainder of the paper, when it is clear that truth-functional substitution is for variable v, $\mathcal{F}[0/v]$ and $\mathcal{F}[1/v]$ will be denoted by $\mathcal{F}_0$ and $\mathcal{F}_1$, respectively.

Lemma 2 and its corollary say that, with respect to variable v, $\mathcal{P}(\mathcal{F})$ can be transformed into $\mathcal{P}(\mathcal{F}_\alpha)$ in polynomial time; moreover it places a limitation on the required checks for subsumption. Specifically, one must only check whether clauses in $\mathcal{Q}$ subsume clauses in $\mathcal{P}(\mathcal{F})$ that contain neither v nor $\neg v$, which takes time proportional to the product of the clause set sizes. The goal, however, is to transform $\mathcal{P}(\mathcal{F}_0)$ and $\mathcal{P}(\mathcal{F}_1)$ into $\mathcal{P}(\mathcal{F})$.

To that end, note first that $\mathcal{F} \equiv (v \vee \mathcal{F}_0) \wedge (\neg v \vee \mathcal{F}_1)$. So $\mathcal{P}(\mathcal{F})$ is logically equivalent to $(v \vee \mathcal{P}(\mathcal{F}_0)) \wedge (\neg v \vee \mathcal{P}(\mathcal{F}_1))$. Denote these conjuncts by J_0 and J_1, respectively; they can be regarded

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

as clause sets by distributing v ($\neg v$) over the clauses of $\mathcal{P}(\mathcal{F}_0)$ ($\mathcal{P}(\mathcal{F}_1)$). Observe that J_0 and J_1 are (separately) resolution-subsumption fixed points because by definition so are $\mathcal{P}(\mathcal{F}_0)$ and $\mathcal{P}(\mathcal{F}_1)$. Subsumption cannot hold between a clause in J_0 and one in J_1 because the former contains v and the latter, $\neg v$. So if $J_0 \cup J_1$ must be altered to produce a resolution-subsumption fixed point, the changes result (directly or indirectly) from resolutions having one parent from each. These can be restricted to resolving on v and $\neg v$ because any other produces a tautology. Note that each such resolvent is the union of a clause from $\mathcal{P}(\mathcal{F}_0)$ and one from $\mathcal{P}(\mathcal{F}_1)$.

It turns out to be sufficient to consider only resolvents formed by one such resolution. This is a consequence of the following theorem, which is a restated version of Theorem 1 from [10].

Theorem 1. Let $\mathcal{F}$ be a logical formula and let v be a variable in $\mathcal{F}$. Suppose E is a prime implicate of $\mathcal{F}$ not containing v. Then $E \subseteq (C \cup D)$, where $C \in \mathcal{P}(\mathcal{F}_0)$ and $D \in \mathcal{P}(\mathcal{F}_1)$. □

Theorem 1 and the discussion leading up to it suggest how $\mathcal{P}(\mathcal{F})$ can be computed from $\mathcal{P}(\mathcal{F}_0)$ and $\mathcal{P}(\mathcal{F}_1)$. It will be useful to denote $\mathcal{P}(\mathcal{F}_0)$ and $\mathcal{P}(\mathcal{F}_1)$ by $\mathcal{P}_0$ and $\mathcal{P}_1$, respectively, and to partition each into two subsets. Let $\mathcal{P}_0^{\supseteq}$ be those clauses in $\mathcal{P}_0$ that are subsumed by some clause in $\mathcal{P}_1$. Let $\mathcal{P}_0^{\bowtie}$ be the remaining clauses in $\mathcal{P}_0$. Similarly define $\mathcal{P}_1^{\supseteq}$, and $\mathcal{P}_1^{\bowtie}$.

Theorem 2. Let J_0, J_1, $\mathcal{P}_0$, $\mathcal{P}_1$, $\mathcal{P}_0^{\supseteq}$, $\mathcal{P}_0^{\bowtie}$, $\mathcal{P}_1^{\supseteq}$, and $\mathcal{P}_1^{\bowtie}$ be defined as above. Then

$$\mathcal{P}(\mathcal{F}) = (v \vee \mathcal{P}_0^{\bowtie}) \cup (\neg v \vee \mathcal{P}_1^{\bowtie}) \cup (\mathcal{P}_0^{\supseteq} \cup \mathcal{P}_1^{\supseteq}) \cup \mathcal{U}$$

where $\mathcal{U}$ is the maximal subsumption-free subset of $\{C \cup D \mid C \in \mathcal{P}_0^{\bowtie}, D \in \mathcal{P}_1^{\bowtie}\}$ in which no clause is subsumed by a clause in $\mathcal{P}_0^{\supseteq}$ or in $\mathcal{P}_1^{\supseteq}$.

Proof. By considering each type of resolvent of a clause in J_0 with one in J_1 with respect to its addition to $J_0 \cup J_1$, the composition of $\mathcal{P}(\mathcal{F})$ can be verified. So assume $\{v\} \cup C$ is resolved with $\{\neg v\} \cup D$ on variable v, where $C \in \mathcal{P}_0$ and $D \in \mathcal{P}_1$. Four types of resolutions are possible, characterized by the blocks in which C and D reside. In three cases, $C \in \mathcal{P}_0^{\supseteq}$ or $D \in \mathcal{P}_1^{\supseteq}$.

Suppose first that $C \in \mathcal{P}_0^{\supseteq}$. Then there is a clause $D' \in \mathcal{P}_1$ that subsumes C. So the resolvent of $\{v\} \cup C$ with $\{\neg v\} \cup D'$ is C. All other resolutions involving $\{v\} \cup C$ result in a superset of C; a subset of C cannot be produced because $\mathcal{P}_0$ and $\mathcal{P}_1$ are prime implicate sets. So C is in $\mathcal{P}(\mathcal{F})$ and $\{v\} \cup C$ is not. This accounts for clauses in $(v \vee \mathcal{P}_0^{\bowtie})$ and in $\mathcal{P}_0^{\supseteq}$.

Similarly, the resolvents of $\{\neg v\} \cup D$ account for clauses in $(\neg v \vee \mathcal{P}_1^{\bowtie})$ and in $\mathcal{P}_1^{\supseteq}$. To summarize, all clauses in $\mathcal{P}_0^{\supseteq}$ and in $\mathcal{P}_1^{\supseteq}$ are in $\mathcal{P}(\mathcal{F})$, and the corresponding clauses of $v \vee \mathcal{P}_0^{\supseteq}$ and $\neg v \vee \mathcal{P}_1^{\supseteq}$ are not.

Now suppose $C \in \mathcal{P}_0^{\bowtie}$ and $D \in \mathcal{P}_1^{\bowtie}$. The resolvent $C \cup D$ may subsume or be subsumed by others of this type. It can also be subsumed by, but cannot subsume,[4] a clause from $\mathcal{P}_0^{\supseteq}$ or from $\mathcal{P}_1^{\supseteq}$. By removing such subsumed clauses from all clauses of this type, the set $\mathcal{U}$ results. □

4 Operations on Clause Sets

In [10], a branch by branch analysis leads to the PIT routine of the pi-trie algorithm introduced there. Theorem 2 in this paper is fundamentally the same. Each leads naturally to a method with which $\mathcal{P}(\mathcal{F})$ can be constructed from $\mathcal{P}_0$ and $\mathcal{P}_1$. However, Theorem 2 provides a set oriented characterization based on resolution and subsumption. The resulting development is arguably more intuitive. More importantly, the set oriented view has led to the more efficient version of the algorithm reported here.

[4] Easily shown by contradiction from the properties of $\mathcal{P}_0$ and $\mathcal{P}_1$.

One improvement results 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 [10] 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) $\mathcal{P}_0^{\supseteq}$; unnecessary subsumption checks result. A second improvement also results that is surprisingly effective. By handling $\mathcal{P}_0^{\supseteq}$ and $\mathcal{P}_0^{\bowtie}$ as separate sets, prime marks are unnecessary. Since the trie representation is being used, prime marks reside at leaf nodes. Checking for their presence requires traversing the branch, and this is almost as expensive as the subsumption check itself. (Using clause lists would allow first, rather than last, literals of a clause to be marked. But then the space economy of the trie would be lost.)

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

We define the following operators on clause sets F and G.

$\mathit{Subsumed}(F, G) = \{C \in G \mid C \text{ is subsumed by some } C' \in F\}$

$\mathit{Unions}(F, G) = \{C \cup D \mid C \in F,\ D \in G,\ C \cup D \text{ is not tautological }\}$

These definitions are purely set-theoretic. The pseudocode below realizes the operations assuming that 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*.

Algorithm 1: *Subsumed*(T_1,T_2)

```
input  : Two clausal tries T1 and T2
output: T, a trie containing all the clauses in T2 subsumed by some clause in T1
if T1 = null or T2 = null then
    T ← null ;
else if leaf(T1) then
    T ← T2;
else
    T ← new Leaf;
    T+ ← Subsumed(T1+, T2+) ∪ Subsumed(T1^0, T2+) ;
    T− ← Subsumed(T1−, T2−) ∪ Subsumed(T1^0, T2−) ;
    T^0 ← Subsumed(T1^0, T2^0) ;
    if leaf(T) then
        T ← null;
return T;
```

For convenience and readability, ordinary set union ($\cup$) and subtraction ($-$) have been employed in the pseudocode. Union can 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 easiest to extract the subsumed branches as a side effect during the subsumption test. Experiments involving both pi-tries and subsumption testing in isolation are reported in Section 5.

[5] Tries have been employed for (even first order) subsumption [17], but on a clause to trie basis, rather than the trie to trie basis developed here.

Algorithm 2: *Unions*(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;
  $T^+ \leftarrow$ *Unions*$(T_1^+, T_2^+) \cup$ *Unions*$(T_1^0, T_2^+) \cup$ *Unions*(T_1^+, T_2^0) ;
  $T^- \leftarrow$ *Unions*$(T_1^-, T_2^-) \cup$ *Unions*$(T_1^0, T_2^-) \cup$ *Unions*(T_1^-, T_2^0) ;
  $T^0 \leftarrow$ *Unions*(T_1^0, T_2^0);
  if *leaf*(T) **then**
    $T \leftarrow null$;
return T;

5 The *pi*-trie Algorithm with set-wise operations

Theorem 2 leads to an alternate, simpler characterization of the *pi*-trie algorithm. We can view the algorithm in the standard divide-and-conquer framework, where each problem $\mathcal{F}$ is divided into subproblems $\mathcal{F}_0, \mathcal{F}_1$ by substitution on the appropriate variable (see the pseudocode for prime). The base case of this is where substitution yields a constant, which gives us $\mathcal{P}(0) = \{\{\}\}$ or $\mathcal{P}(1) = \{\}$.

Algorithm 3: prime($\mathcal{F}$,V)

input : A boolean formula $\mathcal{F}$ and a list of its variables $V = \langle v_1, \ldots, v_k \rangle$
output: The clause set $\mathcal{P}(\mathcal{F})$ — 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 $\{\{\}\}$; `//` $\mathcal{P}(0)$ `is the set of just 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);

The rest of the algorithm consists of combining $\mathcal{P}_0$ and $\mathcal{P}_1$ to form $\mathcal{P}(\mathcal{F})$. This is done both here and in [10] by a routine called PIT. But here, it is based on the $Subsumed$ and $Unions$ operators. The $SubsumedStrict$ operator produces only clauses with proper subsets as subsuming clauses. It is similar in principle to $Subsumed$, but requires additional flags for bookkeeping.

Figure 1 compares the *pi*-trie algorithm from [10] to the updated version using the recursive *Subsumed* and *Unions* operators.[6] The input for both algorithms is 15-variable 3-CNF with varying num-

[6]It is surprisingly difficult to find publicly available prime implicate generators. Substantial email inquiries based on publications produced only the system of Zhuang, Pagnucco and Meyer [20] that implements belief revision using prime implicates.

Algorithm 4: PIT(F_0,F_1,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 $F = \mathcal{P}(\mathcal{F})$
$\mathcal{P}_0^{\supseteq} \leftarrow$ *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` $\mathcal{P}_0^{\supseteq}$
$\mathcal{P}_0^{\bowtie} \leftarrow \mathcal{P}_0 - \mathcal{P}_0^{\supseteq}$;
$\mathcal{P}_1^{\bowtie} \leftarrow \mathcal{P}_1 - \mathcal{P}_1^{\supseteq}$;
$\mathcal{U} \leftarrow$ *Unions*$(\mathcal{P}_0^{\bowtie}, \mathcal{P}_1^{\bowtie})$;
$\mathcal{U} \leftarrow \mathcal{U} -$ *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 $F = v \vee \mathcal{P}_0^{\bowtie} \cup \neg v \vee \mathcal{P}_1^{\bowtie} \cup \mathcal{U}$;

bers 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 new algorithm.

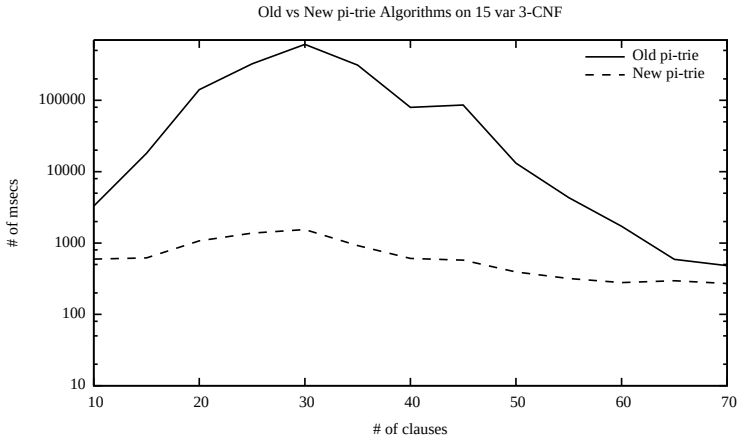

Figure 1: Old vs New *pi*-trie algorithm

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 new algorithm is under development.)

Algorithm 5: NaiveSubsumed(F,G)

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** $H \leftarrow H \cup \{D\}$;
return H;

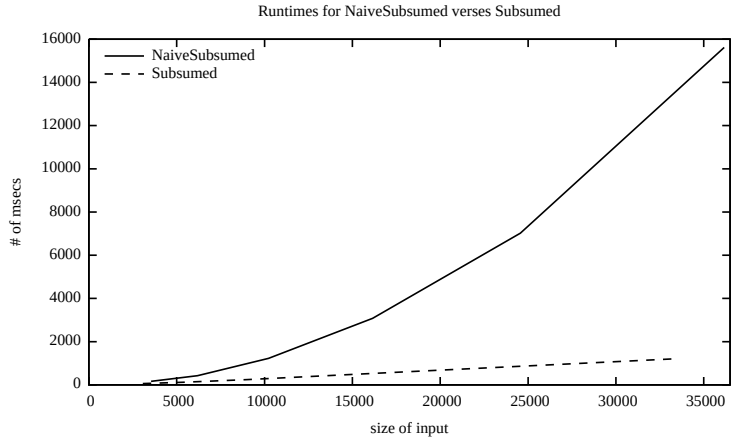

Figure 2: *Subsumed* vs *NaiveSubsumed*

The input for Figure 2 is a pair of n-variable CNF formulas where $n \in \{10, \ldots, 15\}$ 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 $k = 3, 4, 5$.

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 accout the fact that in general tries are a more compact representation of a clause set than a list, so it is inaccurate to graph both runtimes against a single parameter.

It can be seen that the ratio of the runtimes changes as the input size increases – this suggests that the runtimes of *NaiveSubsumed* and *Subsumed* differ asymptotically. Additional evidence is supplied by the following lemma:

Lemma 3. *Subsumed*, when applied to two full ternary tries of depth h and combined size $n = 2(\frac{3^{h+1}-1}{2})$, runs in time $O(n^{\frac{\log 5}{\log 3}}) \approx O(n^{1.465})$.

Proof. At each level, *Subsumed* recurses on five pairs of children, 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, from $n = 2 \cdot \frac{3^{h+1}-1}{2} = 3^{h+1} - 1$ we get $h = \log_3 \frac{n+1}{3}$. This allows us to obtain $T(n) = Z(\log_3 \frac{n+1}{3})$ as an expression for runtime with regard to size. Thus we have

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

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

This is less than *NaiveSubsumed*'s obvious runtime of $O(n^2)$ but still more than linear. Lemma 3 is interesting but the general upper bound may be quite different.

References

[1] Guilherme Bittencourt. Combining syntax and semantics through prime form representation. *Journal of Logic and Computation*, 18:13–33, 2008.

[2] 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.

[3] J. de Kleer. An improved incremental algorithm for computing prime implicants. In *Proc. AAAI-92*, pages 780–785, San Jose, CA, 1992.

[4] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[5] 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.

[6] 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.

[7] 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.

[8] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9:185–206, 1990.

[9] 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.

[10] 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.

[11] D.R. Morrison. Patricia — practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[12] T. Ngair. A new algorithm for incremental prime implicate generation. In *Proc. IJCAI-93, Chambery, France, (1993)*, 1993.

[13] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[14] 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.

[15] 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.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[17] 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*.

[18] 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.

[19] T. Strzemecki. Polynomial-time algorithm for generation of prime implicants. *Journal of Complexity*, 8:37–63, 1992.

[20] Zhi Qiang Zhuang, Maurice Pagnucco, and Thomas Meyer. Implementing iterated belief change via prime implicates. In *Australian Conference on Artificial Intelligence*, pages 507–518, 2007.