An Improved Incremental Algorithm for Generating Prime Implicates

Johan de Kleer
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto CA 94304 USA
email: dekleer@parc.xerox.com

Abstract

Prime implicates have become a widely used tool in AI. The prime implicates of a set of clauses can be computed by repeatedly resolving pairs of clauses, adding the resulting resolvents to the set and removing subsumed clauses. Unfortunately, this brute-force approach performs far more resolution steps than necessary. Tison provided a method to avoid many of the resolution steps and Kean and Tsiknis developed an optimized incremental version. Unfortunately, both these algorithms focus only on reducing the number of resolution steps required to compute the prime implicates. The actual running time of the algorithms depends critically on the number and expense of the subsumption checks they require. This paper describes a method based on a simplification of Kean and Tsiknis' algorithm using an entirely different data structure to represent the data base of clauses. The new algorithm uses a form of discrimination net called tries to represent the clausal data base which produces an improvement in running time on all known examples with a dramatic improvement in running time on larger examples.

1 Introduction

Prime implicates have become a widely used tool in AI. They can be used to implement ATMSs [9], to characterize diagnoses [2], to compile formulas for TMSs [3], to implement circumscription [5; 8], to give but a few examples. The prime implicates of a set of clauses can be computed by repeatedly resolving pairs of clauses, adding the resulting resolvents to the set and removing subsumed clauses. Unfortunately, this brute-force approach performs far more resolution steps than necessary. Tison [10] provided a method to avoid many of the resolution steps and Kean and Tsiknis [6] give an optimized incremental version. Both algorithms provide a significant advance as they substantially reduce the number of resolution steps required to compute the prime implicates of a set of clauses.

Unfortunately, both algorithms focus only on reducing the number of resolution steps required to compute the prime implicates. The actual running time of the algorithm also depends critically on the number and expense of the subsumption checks they requires. Reducing the number of resolutions certainly reduces the number of subsumption checks required. However, the number of subsumption checks required grows (see analysis in Section 3.2 and data in Section 4) faster than the square of the final number of prime implicates. As a result both algorithms are impractical on all but the tiniest examples.

Neither algorithm [6; 10] indicates how subsumption is to be performed, and any prime implicate algorithm is incomplete without such a specification. This paper proposes a method based on a simplification of Kean and Tsiknis' algorithms using an entirely different data structure to represent the data base of clauses in order to facilitate subsumption checking. This data structure is called trie [7] which has been explored extensively for representing dictionaries of words. (Tries are also used in ATMS implementations [1] to store the nogood data base.) Using tries to represent the data base dramatically improves the performance of prime implicate algorithms. The data in Section 4 shows that we can now construct the prime implicates for a much larger set of tasks.

Admittedly, no amount of algorithmic improvement can avoid the complexity produced by the sheer number of prime implicates. The number of prime implicates for many tasks grows relatively quickly so the approach is impractical for many applications. However, we can now conceive of computing prime implicates for applications impossible to before.

2 A brute-force algorithm

The key step in computing prime implicates consists of a resolution rule called consensus[6; 9; 10]. Given two clauses:

$$x \vee \beta,$$

$$\neg x \vee \gamma,$$

where x is a symbol and β and γ are (possibly empty) disjunctions of literals, the consensus of these two clauses with respect to x is the clause,

$$\beta \vee \gamma,$$

with duplicate literals removed. If the two clauses have more than one pair of complementary literals, then the consensus would contain complementary literals and is

discarded (since it is a logical tautology). The prime implicates of a set of clauses can be computed by repeatedly adding the consensus of any pair of clauses to the set and continually removing all subsumed clauses (until no further consensus and subsumption is possible).

The following algorithm finds the prime implicates of a set of clauses Q:

ALGORITHM **BRUTE-FORCE**(Q)

1. Let $\mathcal{P}$ (the result) be {}.
2. Take the first clause q off of Q. If none we are done.
3. If q is subsumed by any clause of $\mathcal{P}$, then go back to step 2.
4. Remove all clauses of $\mathcal{P}$ which are subsumed by q.
5. Try to compute the consensus of q and every clause in $\mathcal{P}$. Whenever the consensus exists, add it to Q.
6. Add q to $\mathcal{P}$.
7. Go to step 2.

3 Improving efficiency

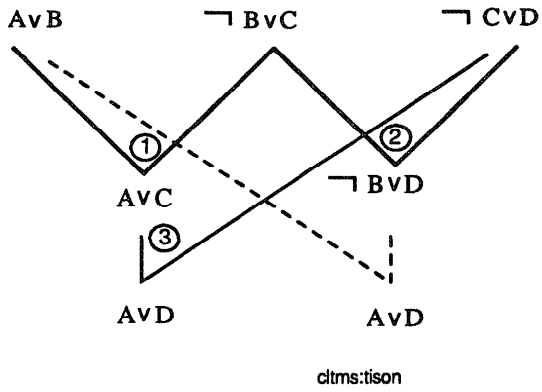
The algorithm presented in Section 2 is intuitively appealing but quite inefficient in practice. Constructing prime implicates is known to be NP-complete and therefore it is unlikely any really good algorithm exists. Nevertheless we can do dramatically better than the algorithm of Section 2. Through logical analysis we can eliminate a large number of the redundant consensus calculations. We can redesign the data structures to support addition and deletion of clauses relatively efficiently.

3.1 A more efficient consensus algorithm

When one observes the behavior of **BRUTE-FORCE**, almost all the clauses produced by the consensus calculation are subsumed by others. One reason for this is somewhat obvious: the consensus operation is commutative and associative when the consensi all exist (usually the case). For example, if we have three clauses α, β and γ, $consensus(\alpha, consensus(\beta, \gamma)) = consensus(consensus(\alpha, \beta), \gamma))$. For 3 clauses **BRUTE-FORCE** finds the same result in 3 distinct ways. Unfortunately, the number of ways to derive a result grows exponentially in the number of clauses used to produce the final result.

Tison [6; 10] introduced the following key intuition which suppresses the majority of the consensus calculations. To compute the prime implicates of a set of clauses, place an ordering on the symbols, then iterate over these symbols in order, doing all consensus calculations with respect to that symbol only. Once all the consensus calculations for a symbol have been made *it is never necessary to do another consensus calculation with respect to that symbol even for all the new consensus results which are produced later* (of course, when the user incrementally supplies the next clause the symbols must be reconsidered). Figure 1 provides a simple example. Suppose we are given clauses $A \vee B$, $\neg B \vee C$ and $\neg C \vee D$. The symbols are ordered: A, B and C. There are no consensus calculations available for A. There is only one (at 1 in the figure) consensus calculation available for B (on the first and second clauses). Finally, when processing C there are two (at 2 and 3 in the figure) consensus calculations available. One of those consensus calculations produces $\neg B \vee D$. Although this resolves with the first original clause, Tison's method tells us the result will be irrelevant because all the useful consensus calculations with respect to B have already been made.

Figure 1: Tison's Method



The following algorithm incorporates this ideas. The algorithm **IPIA** (this derives from the incremental Tison method presented and proved correct in [6]) takes a current set of prime implicates N and a set S of new clauses to add.

ALGORITHM **IPIA**(N,S)

1. Delete any $D \in N \cup S$ that is subsumed by another $D' \in N \cup S$.
2. Remove a smallest C clause from S. If none, return.
3. For each literal l of C, construct Π_l which contains all clauses of N which resolve successfully with C.
4. Let Σ be the set containing C.
5. Perform the following steps for each literal l of C.
   (a) For each clause in Σ which is still in N compute the consensus of it and every clause in Π_l which is still in N.
   (b) For every new consensus, discard it if it has been subsumed by $N \cup S$. Otherwise, remove any clauses in $N \cup S$ subsumed by it. Add the clause to N and Σ.

Comparing this algorithm to the previous one we see that a great many consensus computations are avoided:

- Consensus calculations with respect to a literal earlier in the order are ignored.
- Two clauses produced in the same main step (choice of C) are never resolved with each other.
- Consensus calculations with a D in the original N are ignored unless the consensus of D and C exists.

3.2 Implementing subsumption checking efficiently

Thus far we have been analyzing the logic of the prime implicate algorithms in order to improve their efficiency. However, all the algorithms we know of depend critically on subsumption checking and unless that is properly implemented all the CPU time will be spent checking subsumption.

A key observation is that we are maintaining a data base of unsubsumed clauses. We need to implement 3 transactions with this data base.

1. Check whether clause x is subsumed by some clause of the data base.
2. Add clause x to the data base.
3. Remove all clauses from the data base which are subsumed by x.

To understand some of the complexities, consider the most obvious implementation: We could implement the subsumption check by a subset test and maintain the data base as a simple list. Using lists makes checking for subsumption of order the number of clauses, and thus the complexity of generating k prime implicates at least k^2 which is unacceptable.

Our implementation is based on an integration of two ideas. First, each clause is always represented in a canonical form. Second, the clause data base is represented as a discrimination tree. To achieve a canonical form for clauses we assign a unique integer id to each symbol. We order the literals of every clause in ascending order of their ids. (Complementary literals have the same id, but they can never appear in the same clause as this would produce a tautology.) This means that two sets of literals refer to the same clause if their ordered lists of literals are identical. For example, given symbols A, B and C with id's 1, 2 and 3, the clause,

$$A \vee B \vee C$$

is represented by the list,

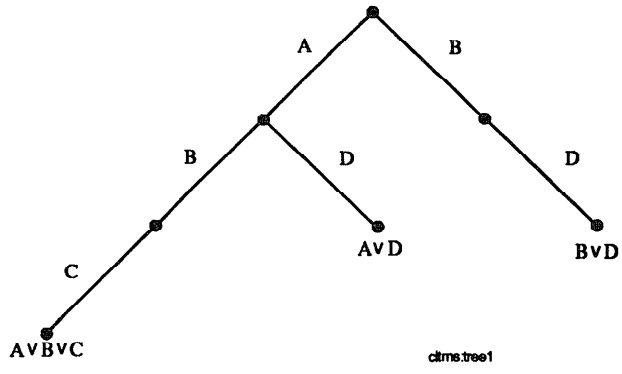
$$[A, B, C].$$

The representation of clauses is sensitive on the choice of id's. If the id's were 3, 1, 2 respectively, then the clause would be represented by the list,

$$[B, C, A].$$

This also makes it possible to test whether a clause of length n subsumes a clause of length m in at most $n + m$ comparisons. However, our algorithm never checks whether one clause subsumes another. Instead, our algorithm stores the canonical forms of clauses in a discrimination tree (or trie [7]). By storing canonical forms in a trie and a single clause can be checked against all existing clauses in a single operation.

Figure 2: Data base with 3 clauses.



The trie for clauses is relatively simple. Conceptually, it is a tree, all of whose edges are literals and whose leaves are clauses. The edges below each node in the trie are ordered by the id of the literal. Suppose that A, B, C, D and E are a sequence of nodes with ascending id's and the data base contains the three clauses:

$$A \vee B \vee C,$$
$$B \vee D,$$
$$A \vee D.$$

The resulting tree is illustrated in Figure 2. Because clauses are canonically ordered, our tries have the additional important property that the id of any edge is less than the id of any edge appearing below it at any depth. This property is heavily exploited in the update algorithms which follow.

The most commonly called procedure checks whether a clause is subsumed by one in the data base. Given an ordered set of literals, the recursive function SUBSUMED? checks whether the set of literals L is subsumed by trie N. The ordered literals are represented as an ordered list, and the trie by an ordered list of edges.

ALGORITHM SUBSUMED?(L, N)

1. If N is a terminal clause, return success.
2. Remove literals from the front of L until the id of the first edge of N is greater than that of the first literal of L.
3. If no literals remain (L is empty), return failure.
4. For each literal l of L do the following until success or the id of the first edge of N is no longer greater than that of l.

(a) If the first literal of N is l, recursively invoke **SUBSUMED?** on the remaining literals and the edges below the first element of N.

(b) If the recursive call returns success, return success.

5. Remove the first element of N.
6. Go to Step 2.

Suppose we want to check whether $D \vee E$ is subsumed by the data base of Figure 2. The root of the trie has 2 outgoing edges, A and B. D has a larger id than the top two edges of the trie (A and B), therefore **SUBSUMED?** immediately reports failure. Suppose we want to check whether $A \vee B \vee D$ is subsumed by the trie. The first edge from the root matches the first literal, so the recursive call tries to determine whether the remaining subclause $B \vee D$ is subsumed by the trie rooted from the edge below A. Again B matches, but D does not match C so the two recursive calls to **SUBSUMED?** fail. Finally, the top-level invocation of **SUBSUMED?** again recursively calls itself and finds a successful match.

Adding a clause to the data base is very simple. Our algorithm exploits the fact that the clause to be added is not itself subsumed by some other clause, and that any clause it subsumes has been removed from the data base.

ALGORITHM **ADD-TO-TRIE**(L, N)

1. Remove edges of the front of N, until the id of the first edge of N is greater or equal to the first literal of L.
2. If the label of the first edge of N is the same as that of L, recursively call **ADD-TO-TRIE** with the remainder of L, the edges underneath the first edge of N and return.
3. Construct the edges to represent the literals of L and return, and side-effect the trie such that it appears just before the current position N.

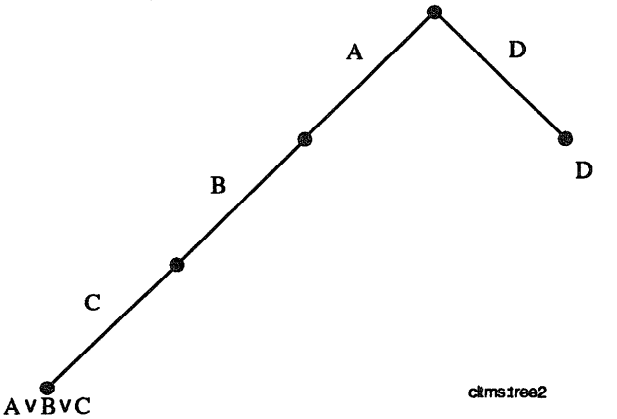
The potentially most expensive operation and the one which requires the greatest care is the third basic update on the trie. Here we are given a clause, not subsumed by the trie and we must remove from the trie all clauses subsumed by it.

ALGORITHM **REMOVE-SUBSUMED**(L, N)

1. If there are no literals, delete the entire trie represented by N and return.
2. If we are at a leaf of the trie, return.
3. For each edge e of N, do the following.

(a) If the label of e is the first literal of L, then recursively call **REMOVE-SUBSUMED** with the rest of the literals and the edges below e.

Figure 3: Trie with 2 clauses.



(b) If the label of e is lower than that of the first literal of L, then recursively call **REMOVE-SUBSUMED** on the *same* literals but the edges below e.

As an extreme case suppose we want to remove all clauses subsumed by D from Figure 2. Because D is last in the ordering, the algorithm simply searches in left-to-right depth-first order removing all clauses containing D. After adding the clause D, the resulting trie is illustrated by Figure 3.

4 Results

Table 1 table summarizes the performance improvement of **IPIA** produced using a trie. Each line of the table lists the name of the task, the number of clauses which specify the problem, the number of prime implicates of these clauses, the number of subsumption checks the non-trie version of **IPIA** requires, the running times of the two algorithms, the average fraction of the trie searched during subsumption checks, the fraction of resolvents which are not subsumed by the trie, and the number of non-terminal nodes in the final trie. The timings are obtained on a Symbolics XL1200. The Lisp code is not particularly optimized as it is designed for a text book [4].

The non-trie version of **IPIA** is so slow that it is not possible to actually time its performance on larger examples. We instrumented the trie version to estimate the number of subsumption checks that would have to be made by assuming that, on average, each subsumption check would check half of the current clauses. In all the cases we have tried, this estimate is within 25% of the actual number of subsumption checks so this estimate seems fairly reliable. We estimate the timing of the non-trie version using 10 microseconds per subsumption check which is the fastest we've ever seen the non-trie algorithm perform. The implementation is written in basic Common Lisp and runs in Lucid and Franz as well. Both the code and the examples

Task	Clauses	PIs	Subsumption Checks	t-list(s)	t-TRIE(s)	Fraction Searched	Fraction New	Trie Size
Two-Pipes	54	88	13124	.22	.06	.081	.50	54
Adder	50	8400	3.8×10^{10}*	3.8×10^{5}*	721	.009		16207
K33	9	73	5166	.12	.03	.123	1	24
K44	16	641	506472	5.5	.46	.031	1	160
K55	25	7801	9.1×10^{7}*	900*	8.6	.003	1	1560
K54	20	1316	1730120	22.6	1.1		1	263
K66	36	117685	2.4×10^{10}*	2.4×10^{5}*	224		1	
K67	42	823585	1.3×10^{15}*	1.4×10^{10}*	2541		1	137264
Regulator	106	2814	3.7×10^{9}*	3.7×10^{4}*	85		.06	1365
BD	151	1907	1.1×10^{9}*	1.1×10^{4}*	38		.067	2493

Table 1: Comparison of the old and improved algorithms. "*" indicates estimates.

are available from the author.

Table 1 clearly indicates the dramatic performance achievement produced using a trie data structure to represent the clausal data base. The actual **IPIA** algorithm in [6] includes additional optimizations to the version we have presented in this paper. Those optimizations reduce the number of resolutions and subsumption checks. However, these optimizations entail additional bookkeeping and are known to produce incorrect results in some cases. Therefore, we restrict our comparison to the basic version of **IPIA**.

The tasks labeled "Two-Pipes" and "Regulator" come from qualitative physics. The tasks labeled "Adder" and "BD" come are diagnosis problems. The tasks labeled "K"nm are taken from [6].

The analysis clearly shows that the trie-based algorithm yields substantial improvement in all cases. The final columns in the table provide some insight into why performance improves so dramatically. One hypothesis for the good performance is that most new clauses are subsumed by others and therefore immediately found in the trie and that the data structure would perform poorly if there were few subsumptions. The data does not substantiate this intuition. The column labeled "Fraction New" indicates the fraction of the new clauses which are not subsumed by the current trie. We see even in the worst case where the new clause is never subsumed, that the trie-based algorithm is far superior. The column labeled "Fraction Searched" shows the average fraction of the nonterminal nodes in the trie actually searched. This data suggests that the central advantage of using tries is that subsumption checks need only examine a small fraction of trie.

We have attempted to invent tasks which would defeat the advantage of using tries and have not been able to find any. Tries would perform poorly for a task in which all subsumption checks failed and all the nodes in the trie have to be scanned for each failing subsumption check. To achieve this the trie would have to contain a very high density of clauses. It is difficult to devise an initial set of clauses whose prime implicates densely populate the space. Moreover, the fact that clauses resolve with each to produce new ones means that if the clause set becomes too dense the clause set is likely to be reduced by subsumptions (a dense clause set is also likely to be inconsistent).

The performance of the trie-based **IPIA** is relatively sensitive to the choice of ids (although performance is always much better than the non-trie version). The choice of ids affects the canonicalized forms of the clauses and hence the size of the trie. To lower the size of the trie, the most common symbols should have lower id. Although it is impossible to tell initially which symbols will occur more commonly in the prime implicates, we use the number of occurences in the initial clause set as a guide. **IPIA** performs noticably better if less common symbols are processed first (i.e., in steps 2 and 5).

References

[1] de Kleer, J., An assumption-based truth maintenance system, *Artificial Intelligence* **28** (1986) 127–162. Also in *Readings in NonMonotonic Reasoning*, edited by Matthew L. Ginsberg, (Morgan Kaufmann, 1987), 280–297.

[2] de Kleer, J., Mackworth A., and Reiter R., Characterizing Diagnoses, in: *Proceedings AAAI-90*, Boston, MA (1990) 324–330. Extended version will appear in *Artificial Intelligence* Journal.

[3] de Kleer, J., Exploiting locality in a TMS, AAAI-90, Boston, MA (1990) 254–271.

[4] Forbus, K., and de Kleer, J., *Building problem solvers* (MIT Press, 1992).

[5] Ginsberg, M.L., A Circumscriptive Theorem Prover. *Proceedings of the second international workshop on Non-Monotonic Reasoning.* Springer, LNCS 346, 100–114, (1988).

[6] Kean, A. and Tsiknis, G., An incremental method for generating prime implicants/implicates, *Journal of Symbolics Computation* **9** (1990) 185–206.

[7] Knuth, D.E., *The art of computer programming* (Addison-Wesley, Reading, MA, 1972).

[8] Raiman, O., and de Kleer, J., A Minimality Maintenance System, submitted for publication, 1992.

[9] Reiter, R. and de Kleer, J., Foundations of assumption-based truth maintenance systems: Preliminary report, *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA (July, 1987), 183–188.

[10] Tison, P., Generalized consensus theory and application to the minimization of boolean functions, *IEEE transactions on electronic computers* **4** (August 1967) 446-456.