Efficient Query Processing with Compiled Knowledge Bases

Neil V. Murray[1] and Erik Rosenthal[2]

[1] Department of Computer Science, State University of New York, Albany, NY 12222, USA,
nvm@cs.albany.edu
[2] Department of Mathematics, University of New Haven, West Haven, CT 06516, USA,
erosenthal@newhaven.edu

Abstract. The goal of knowledge compilation is to enable fast queries. Prior approaches had the goal of small (i.e., polynomial in the size of the initial knowledge bases) compiled knowledge bases. Typically, query-response time is linear, so that the efficiency of querying the compiled knowledge base depends on its size. In this paper, a target for knowledge compilation called the ri*-trie* is introduced; it has the property that even if they are large they nevertheless admit fast queries. Specifically, a query can be processed in time *linear in the size of the query* regardless of the size of the compiled knowledge base.

1 Introduction

The last decade has seen a virtual explosion of applications of propositional logic. One is *knowledge representation*, and one approach to it is *knowledge compilation*. Knowledge bases can be represented as propositional theories, often as sets of clauses, and the propositional theory can then be *compiled*; i.e., preprocessed to a form that admits fast response to queries. While knowledge compilation is intractable, it is done once, in an off-line phase, with the goal of making frequent on-line queries efficient. Heretofore, that goal has not been achieved for arbitrary propositional theories.

A typical query of a propositional theory has the form, is a clause logically entailed by the theory? This question is equivalent to asking, is the conjunction of the theory and the negation of the clause unsatisfiable? Propositional logic is of course intractable (unless $\mathcal{NP} = \mathcal{P}$), so the primary goal of most research is to find relatively efficient deduction techniques. A number of languages — for example, *Horn sets*, *ordered binary decision diagrams*, sets of *prime implicates/implicants*, *decomposable negation normal form*, *factored negation normal form*, and *pairwise-linked formulas* — have been proposed as targets for knowledge compilation. (See, for example, [1, 3, 8, 10, 18, 21, 20, 29, 43, 49].

Knowledge compilation was introduced by Kautz and Selman [24]. They were aware of one issue that is not discussed by all authors: The ability to answer queries in time polynomial (indeed, often linear) in the size of the compiled theory is not very fast if the compiled theory is exponential in the size of the underlying propositional theory. Most investigators who have considered this issue focused on minimizing the size of the compiled theory, possibly by restricting or approximating the original theory. Another approach is considered in this paper: admitting large compiled theories —

stored off-line[3] — on which queries can be answered in time *linear in the size of the query*.

A data structure that has this property is called a *reduced implicate trie* or, more simply, an *ri-trie*, and is introduced in Section 3.2. These tries can be thought of as compact *implicate tries*, which are introduced in Section 3.1. Note that the target languages studied by the authors in [21, 34] are related but nevertheless distinct from this work; they enable response times linear only in the size of the compiled theory, which (unfortunately) can be exponentially large.

The Tri operator and RIT operator, which are the building blocks of ri-tries, are introduced and the appropriate theorems are proved in Section 4. A direct implementation of the RIT operator would appear to have a significant inefficiency. However, the algorithm developed in Section 5 avoids this inefficiency.

2 Preliminaries

For the sake of completeness, define an *atom* to be a propositional variable, a *literal* to be an atom or the negation of an atom, and a *clause* to be a disjunction of literals. Clauses are often referred to as sets of literals. Most authors restrict attention to *conjunctive normal form* (CNF) — a conjunction of clauses — but no such restriction is required in this paper.

Consequences expressed as minimal clauses that are implied by a formula are its *prime implicates*; (and minimal conjunctions of literals that imply a formula are its *prime implicants*). Implicates are useful in certain approaches to non-monotonic reasoning [27, 40, 46], where all consequences of a formula — for example, the support set for a proposed common-sense conclusion — are required. The implicants are useful in situations where satisfying models are desired, as in error analysis during hardware verification. Many algorithms have been proposed to compute the prime implicates (or implicants) of a propositional boolean formula [5, 14, 22, 23, 26, 39, 42, 48, 51].

An *implicate* of a logical formula is a clause that is entailed by the formula; i.e., a clause that contains a prime implicate. Thus, if $\mathcal{F}$ is a formula and C is a clause, then C is an implicate of $\mathcal{F}$ if (and only if) C is satisfied by every interpretation that satisfies $\mathcal{F}$. Still another way of looking at implicates is to note that asking whether a given clause is entailed by a formula is equivalent to asking whether the clause is an implicate of the formula. Throughout the paper, this question is what is meant by query.

3 A Data Structure That Enables Fast Query Processsing

The goal of knowledge compilation is to enable fast queries. Prior approaches had the goal of a small (i.e., polynomial in the size of the initial knowledge base) compiled knowledge base. Typically, query-response time is linear, so that the efficiency of querying the compiled knowledge base depends on its size. The approach considered in this

[3] The term *off-line* is used in two ways: first, for off-line memory, such as hard drives, as opposed to on-line storage, such as RAM, and secondly, for "batch preparation" of a knowledge base for on-line usage.

paper is to admit target languages that may be large as long as they enable fast queries. The idea is for the query to be processed in time *linear in the size of the query*. Thus, if the compiled knowledge base is exponentially larger than the initial knowledge base, the query must be processed in time logarithmic in the size of the compiled knowledge base. One data structure that admits such fast queries is called a *ri-trie* (for *reduced implicate trie*).

3.1 Implicate Tries

The trie is a well-known data structure introduced by Morrison in 1968 [30]; it is a tree in which each branch represents the sequence of symbols labeling the nodes[4] on that branch, in descending order. A prefix of such a sequence may be represented along the same branch by defining a special *end symbol* and assigning an extra child labeled by this symbol to the node corresponding to the last symbol of the prefix. For convenience, it is assumed here that the node itself is simply marked with the end symbol, and leaf nodes are also so marked. One common application for tries is a dictionary. The advantage is that each word in the dictionary is present precisely as a (partial) branch in the trie. Checking a string for membership in the dictionary merely requires tracing a corresponding branch in the trie. This will either fail or be done in time linear in the size of the string.

Tries have also been used to represent logical formulas, including sets of prime implicates [46]. 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 represented by the trie. But observe that this CNF formula introduces significant redundancy. In fact, the trie can be interpreted directly as an NNF formula, recursively defined as follows: A trie consisting of a single node represents the constant labeling that node. Otherwise, the trie represents the disjunction of the label of the root with the conjunction of the formulas represented by the tries rooted at its children.

When clause sets are stored as tries, space advantages can be gained by ordering the literals and treating the clauses as ordered sets. An n-literal clause will be represented by one of the $n!$ possible sequences. If the clause set is a set of implicates, then one possibility is to store only prime implicates — clauses that are not subsumed by others — because all subsumed clauses are also implicates and thus implicitly in the set. The space savings can be considerable, but there will in general be exponentially many prime implicates. Furthermore, to determine whether clause C is in the set, the trie must be examined for any subset of C; the literal ordering helps, but the cost is still proportional to the size of the trie.

Suppose instead that *all* implicates are stored; the resulting trie is called an *implicate trie*. To define it formally, let $p_1, p_2, ..., p_n$ be the variables that appear in the input knowledge base $\mathcal{D}$, and let q_i be the literal p_i or $\neg p_i$. Literals are ordered as follows: $q_i \prec q_j$ iff $i < j$. (This can be extended to a total order by defining $\neg p_i \prec p_i, 1 \leq i \leq n$. But neither queries nor branches in the trie will contain such complementary pairs.) The implicate trie for $\mathcal{D}$ is a tree defined as follows: If $\mathcal{D}$ is a tautology (contradiction), the

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

tree consists only of a root labeled 1 (0). Otherwise, it is a tree whose root is labeled 0 and has, for any implicate $C = \{q_{i_1}, q_{i_2}, \ldots, q_{i_m}\}$, a child labeled q_{i_1}, which is the root of a subtree containing a branch with labels corresponding to $C - \{q_{i_1}\}$. The clause C can then be checked for membership in time linear in the size of C, simply by traversing the corresponding branch.

Note that the node on this branch labeled q_{i_m} will be marked with the end symbol. Furthermore, given any node labeled by q_j and marked with the end symbol, if $j < n$, it will have as children nodes labeled q_k and $\neg q_k$, $j < k \leq n$, and these are all marked with the end symbol. This is an immediate consequence of the fact that a node marked with the end symbol represents an implicate which is a prefix (in particular, subset) of every clause obtainable by extending this implicate in all possible ways with the literals greater than q_j in the ordering.

3.2 Reduced Implicate Tries

Recall that for any logical formulas $\mathcal{F}$ and α and subformula $\mathcal{G}$ of $\mathcal{F}$, $\mathcal{F}[\alpha/\mathcal{G}]$ denotes the formula produced by substituting α for every occurrence of $\mathcal{G}$ in $\mathcal{F}$. If α is a truth functional constant 0 or 1 ($false$ or $true$), and if p is a negative literal, we will slightly abuse this notation by interpreting the substitution $[0/p]$ to mean that 1 is substituted for the atom that p negates.

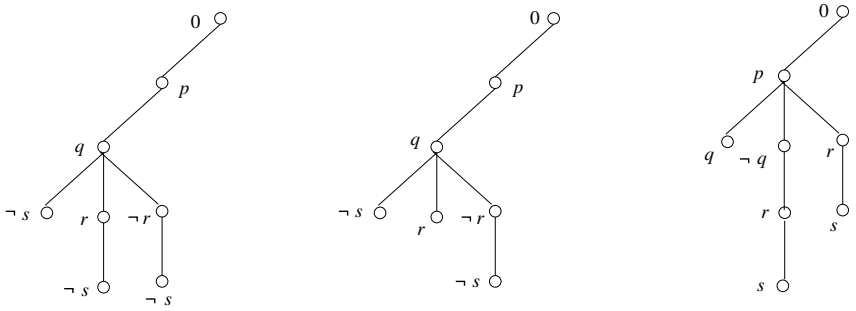
The following simplification rules[5] are useful (even if trivial).

SR5.	$\mathcal{F}[\mathcal{G}/\mathcal{G} \vee 0]$	$\mathcal{F}[\mathcal{G}/\mathcal{G} \wedge 1]$
SR6.	$\mathcal{F}[0/\mathcal{G} \wedge 0]$	$\mathcal{F}[1/\mathcal{G} \vee 1]$
SR8.	$\mathcal{F}[0/p \wedge \neg p]$	$\mathcal{F}[1/p \vee \neg p]$

If $C = \{q_{i_1}, q_{i_2}, \ldots, q_{i_m}\}$ is an implicate of $\mathcal{F}$, it is easy to see that the node labeled q_{i_m} will become a leaf if these rules are applied repeatedly to the subtree of the implicate trie of $\mathcal{F}$ rooted at q_{i_m}. Moreover, the product of applying these rules to the entire implicate trie until no applications of them remain will be a trie in which no internal nodes are marked with the end symbol and all leaf nodes are, rendering that symbol merely a convenient indicator for leaves. The result ot this process is called a *reduced implicate trie* or simply an ri*-trie*.

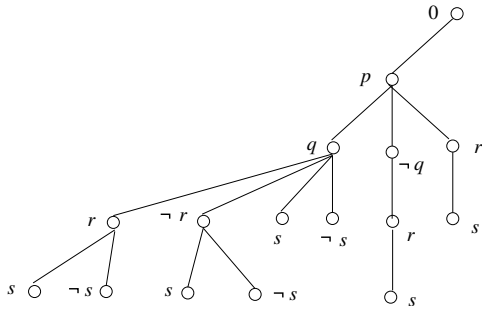
Consider an example. Suppose that the knowledge base $\mathcal{D}$ contains the variables p, q, r, s, in that order, and suppose that $\mathcal{D}$ consists of the following clauses: $\{p, q, \neg s\}$, $\{p, q, r\}$, $\{p, r, s\}$, and $\{p, q\}$. Initialize the ri-trie as a single node labeled 0 and then build it one clause at a time. After the first is added to the tree, its two supersets must also be added. The resulting ri-trie is on the left in the diagram below. Adding the second clause implies that the node labeled by r is also a leaf. Then all extensions of this branch are entailed by $\{p, q, r\}$ (and thus by $\mathcal{D}$), and the corresponding child is dropped resulting in the ri-trie in the center.

[5] The labels of these rules come from [21].



Adding the last two clauses produces the ri-trie on the right.

The complete implicate trie in the example is shown below. It has eight branches, but there are eleven end markers representing its eleven implicates (nine in the subtree rooted at q, and one each at the two rightmost occurrences of s.)



Observations (ri-tries).

1. The NNF equivalent of the *ri*-trie is

$$p \vee (q \wedge (\neg q \vee r \vee s) \wedge (r \vee s)).$$

2. In general, the length of a branch is at most n, the number of variables that appear in the original logical formula $\mathcal{D}$.
3. In order to have the determination of entailment of a given clause be linear in the size of the clause, enough branches must be included in the tree so that the test for entailment can be accomplished by traversing a single branch.
4. When a clause is added to the trie, the literal of highest index becomes the leaf, and *that branch need never be extended*; i.e., if a clause is being tested for entailment, and if a prefix of the clause is a branch in the trie, that the clause is entailed.
5. The ri-trie will be stored off-line. Even if it is very large, each branch is small — no longer than the number of variables — and can be traversed very quickly, even with relatively slow off-line storage.
6. If the query requires sorting, the search time will be $n \log n$, where n is the size of the query. But the sorting can be done in main memory, and the search time in the (off-line) ri-trie is still linear in the size of the query.

4 A Procedure for Computing ri-Tries

In this section a procedure for computing an ri-trie as a logical formula is presented. The *Tri operator* is described in Section 4.1; it can be thought of as a single step in the process that creates an ri-trie. The *RIT operator*, described in Section 4.2, produces the ri-trie with recursive applications of the Tri operator.

It will be convenient to assume that any constant that arises in a logical formula is simplified away with repeated applications of rules **SR5** and **SR6** (unless the formula is constant).

4.1 The Tri Operator

The Tri operator restructures a formula by substituting truth constants for a variable. Lemmas 1 and 2 and the corollary that follows provide insight into the implicates of the components of $\mathrm{Tri}(\mathcal{F}, p)$. Let $\mathrm{Imp}(\mathcal{F})$ denote the set of all implicates of $\mathcal{F}$.

The *tri-expansion*[6] of any formula $\mathcal{F}$ with respect to any atom p is defined to be

$$\mathrm{Tri}(\mathcal{F}, p) \;=\; (p \;\vee\; \mathcal{F}[0/p]) \;\wedge\; (\neg p \;\vee\; \mathcal{F}[1/p]) \;\wedge\; (\mathcal{F}[0/p] \;\vee\; \mathcal{F}[1/p]).$$

Lemma 1. Suppose that the clause C is an implicate of the logical formula $\mathcal{F}$, and that the variable p occurs in $\mathcal{F}$ but not in C. Then $C \in \mathrm{Imp}(\mathcal{F}[0/p]) \cap \mathrm{Imp}(\mathcal{F}[1/p])$.

Proof. Let I be an interpretation that satisfies $\mathcal{F}[1/p]$; we must show that I satisfies C. Extend I to $\tilde{I}$ by setting $\tilde{I}(p) = 1$. Clearly, $\tilde{I}$ satisfies $\mathcal{F}$[7], so $\tilde{I}$ satisfies C. But then, since p does not occur in C, I satisfies C. The proof for $\mathcal{F}[0/p]$ is identical, except that $\tilde{I}(p)$ must be set to 0. □

Lemma 2. Let $\mathcal{F}$ and $\mathcal{G}$ be logical formulas. Then $\mathrm{Imp}(\mathcal{F}) \cap \mathrm{Imp}(\mathcal{G}) = \mathrm{Imp}(\mathcal{F} \vee \mathcal{G})$.

Proof. Suppose first that C is an implicate of both $\mathcal{F}$ and $\mathcal{G}$. We must show that C is an implicate of $\mathcal{F} \vee \mathcal{G}$, so let I be an interpretation that satisfies $\mathcal{F} \vee \mathcal{G}$. Then I satisfies $\mathcal{F}$ or I satisfies $\mathcal{G}$, say $\mathcal{F}$. Then, since C is an implicate of $\mathcal{F}$, I satisfies C.

Suppose now that $C \in \mathrm{Imp}(\mathcal{F} \vee \mathcal{G})$. We must show that $C \in \mathrm{Imp}(\mathcal{F})$ and that $C \in \mathrm{Imp}(\mathcal{G})$. To see that $C \in \mathrm{Imp}(\mathcal{F})$, let I be any satisfying interpretation of $\mathcal{F}$. Then I satisfies $\mathcal{F} \vee \mathcal{G}$, so I satisfies C. The proof that $C \in \mathrm{Imp}(\mathcal{G})$ is entirely similar. □

Corollary. Let C be a clause not containing p or $\neg p$, and let $\mathcal{F}$ be any logical formula. Then C is an implicate of $\mathcal{F}$ iff C is an implicate of $\mathcal{F}[0/p] \vee \mathcal{F}[1/p]$. □

[6] This is tri as in three, not as in trie; the pun is probably intended.

[7] It is possible that there are variables other than p that occur in $\mathcal{F}$ but not in $\mathcal{F}[1/p]$. But such variables must "simplify away" when 1 is substituted for p, so I can be extended to an interpretation of $\mathcal{F}$ with any truth assignment to such variables.

4.2 The RIT Operator

The ri-trie of a formula can be obtained by applying the Tri operator successively on the variables. Let $\mathcal{F}$ be a logical formula, and let the variables of $\mathcal{F}$ be $V = \{p_1, p_2, ..., p_n\}$. Then the RIT operator is defined by

$$\mathrm{RIT}(\mathcal{F}, V) \;=\; \begin{cases} \mathcal{F} & V = \emptyset \\ \begin{array}{c} p_i \;\vee\; \mathrm{RIT}(\mathcal{F}[0/p_i], V - \{p_i\}) \\ \wedge \\ \neg p_i \;\vee\; \mathrm{RIT}(\mathcal{F}[1/p_i], V - \{p_i\}) \\ \wedge \\ \mathrm{RIT}((\mathcal{F}[0/p_i] \;\vee\; \mathcal{F}[1/p_i]), V - \{p_i\}) \end{array} & p_i \in V \end{cases}$$

where p_i is the variable of lowest index in V.

Implicit in this definition is the use of simplification rules **SR5**, **SR6**, and **SR8**. Theorem 2 below essentially proves that the RIT operator produces ri-tries; reconsidering the above example illustrates this fact:

$$\mathcal{F} = \{p, q, \neg s\} \;\wedge\; \{p, q, r\} \;\wedge\; \{p, r, s\} \;\wedge\; \{p, q\},$$

so that $V = \{p, q, r, s\}$. Let $\mathcal{F}[0/p]$ and $\mathcal{F}[1/p]$ be denoted by $\mathcal{F}_0$ and $\mathcal{F}_1$, respectively. Since p occurs in every clause, $\mathcal{F}_0$ amounts to deleting p from each clause, and $\mathcal{F}_1 = 1$. Thus,

$$\begin{aligned} \mathrm{RIT}(\mathcal{F}, V) \;&=\; (p \;\vee\; \mathrm{RIT}(\mathcal{F}_0, \{q, r, s\})) \;\wedge\; (\neg p \;\vee\; 1) \;\wedge\; \mathrm{RIT}((\mathcal{F}_0 \;\vee\; 1), \{q, r, s\}) \\ &=\; (p \;\vee\; \mathrm{RIT}(\mathcal{F}_0, \{q, r, s\})) \end{aligned}$$

where

$$\mathcal{F}_0 \;=\; \{q, \neg s\} \;\wedge\; \{q, r\} \;\wedge\; \{r, s\} \;\wedge\; \{q\}.$$

Let $\mathcal{F}_0[0/q]$ and $\mathcal{F}_0[1/q]$ be denoted by $\mathcal{F}_{00}$ and $\mathcal{F}_{01}$, respectively. Observe that $\mathcal{F}_{00} = 0$, and $q \vee 0 = q$. Thus,

$$\mathrm{RIT}(\mathcal{F}_0, \{q, r, s\}) \;=\; q \;\wedge\; (\neg q \;\vee\; \mathrm{RIT}(\mathcal{F}_{01}, \{r, s\})) \;\wedge\; \mathrm{RIT}((0 \vee \mathcal{F}_{01}), \{r, s\}),$$

where

$$\mathcal{F}_{01} \;=\; \mathcal{F}_0[1/q] \;=\; 1 \;\wedge\; 1 \;\wedge\; (r \;\vee\; s) \;\wedge\; 1 \;=\; (r \;\vee\; s).$$

Observe now that $\mathcal{F}_{01}[0/r] = s$ and $\mathcal{F}_{01}[1/r] = 1$. Thus,

$$\mathrm{RIT}(\mathcal{F}_{01}, \{r, s\}) \;=\; (r \;\vee\; \mathrm{RIT}(s, \{s\})) \;\wedge\; (\neg r \;\vee\; 1) \;\wedge\; \mathrm{RIT}((s \vee 1), \{s\}).$$

Finally, since $\mathrm{RIT}(s, \{s\}) = s$, substituting back produces

$$\mathrm{RIT}(\mathcal{F}, V) \;=\; p \;\vee\; (q \;\wedge\; (\neg q \;\vee\; r \;\vee\; s) \;\wedge\; (r \;\vee\; s)),$$

which is exactly the formula obtained originally from the ri-trie.

If a logical formula contains only one variable p, then it must be logically equivalent to one of the following four formulas: $0, 1, p, \neg p$. The next lemma, which is trivial to prove from the definition of the RIT operator, says that in that case, $\mathrm{RIT}(\mathcal{F}, \{p\})$ is precisely the simplified logical equivalent.

Lemma 3. Suppose that the logical formula $\mathcal{F}$ contains only one variable p. Then $\mathrm{RIT}(\mathcal{F}, \{p\})$ is logically equivalent to $\mathcal{F}$ and is one of the formulas $0, 1, p, \neg p$. □

For the remainder of the paper, assume the following notation with respect to a logical formula $\mathcal{F}$: Let $V = \{p_1, p_2, ...p_n\}$ be the set of variables of $\mathcal{F}$, and let $V_i = \{p_{i+1}, p_{i+2}, ..., p_n\}$. Thus, for example, $V_0 = V$, and $V_1 = \{p_2, p_3, ..., p_n\}$. Let $\mathcal{F}_t = \mathcal{F}[t/p_i], t = 0, 1$, where p_i is the variable of lowest index in $\mathcal{F}$.

Theorem 1. If $\mathcal{F}$ is any logical formula with variable set V, then $\mathrm{RIT}(\mathcal{F}, V)$ is logically equivalent to $\mathcal{F}$, and each branch of $\mathrm{RIT}(\mathcal{F}, V)$ is an implicate of $\mathcal{F}$.

Proof. We first prove logical equivalence. Proceed by induction on the number n of variables in $\mathcal{F}$. The last lemma takes care of the base case $n = 1$, so assume the theorem holds for all formulas with at most n variables, and suppose that $\mathcal{F}$ has $n + 1$ variables. Then we must show that

$$\mathcal{F} \equiv \mathrm{RIT}(\mathcal{F}, V) = \begin{array}{c} p_1 \vee \mathrm{RIT}(\mathcal{F}_0, V_1) \\ \wedge \\ \neg p_1 \vee \mathrm{RIT}(\mathcal{F}_1, V_1) \\ \wedge \\ \mathrm{RIT}((\mathcal{F}_0 \vee \mathcal{F}_1), V_1). \end{array}$$

By the induction hypothesis, $\mathcal{F}_0 \equiv \mathrm{RIT}(\mathcal{F}_0, V_1)$, $\mathcal{F}_1 \equiv \mathrm{RIT}(\mathcal{F}_1, V_1)$, and $(\mathcal{F}_0 \vee \mathcal{F}_1) \equiv \mathrm{RIT}((\mathcal{F}_0 \vee \mathcal{F}_1), V_1)$. Let I be any interpretation that satisfies $\mathcal{F}$, and suppose first that $I(p_1) = 1$. Then I satisfies p_1, $\mathcal{F}_1$, and $(\mathcal{F}_0 \vee \mathcal{F}_1)$, so I satisfies each of the three conjuncts of $\mathrm{RIT}(\mathcal{F}, V)$; i.e., I satisfies $\mathrm{RIT}(\mathcal{F}, V)$. The case when $I(p) = 0$ and the proof that any satisfying interpretation of $\mathrm{RIT}(\mathcal{F}, V)$ satisfies $\mathcal{F}$ are similar.

Every branch is an implicate of $\mathcal{F}$ since, by the distributive laws, $\mathrm{RIT}(\mathcal{F}, V)$ is logically equivalent to the conjunction of its branches. □

Lemma 4. Let C be an implicate of $\mathcal{F}$ containing the literal p. Then $C - \{p\} \in \mathrm{Imp}(\mathcal{F}[0/p])$.

Proof. Let I be an interpretation that satisfies $\mathcal{F}[0/p]$. Extend I by defining $I(p) = 0$. Then I satisfies $\mathcal{F}$, so I satisfies C. Since I assigns 0 to p, I must satisfy a literal in C other than p; i.e., I satisfies $C - \{p\}$. □

The theorem below says, in essence, that ri-tries have the desired property that determining whether a clause is an implicate can be done by traversing a single branch. If $\{q_1, q_2, ..., q_k\}$ is the clause, it will be an implicate iff for some $i \leq k$, there is a branch labeled $q_1, q_2, ..., q_i$. The clause $\{q_1, q_2, ..., q_i\}$ subsumes $\{q_1, q_2, ..., q_k\}$, but it is not an arbitrary subsuming clause. To account for this relationship, a *prefix* of a clause $\{q_1, q_2, ..., q_k\}$ is a clause of the of the form $\{q_1, q_2, ..., q_i\}$, where $0 \leq i \leq k$. Implicit in this definition is a fixed ordering of the variables; also, if $i = 0$, then the prefix is the empty clause.

Theorem 2. Let $\mathcal{F}$ be a logical formula with variable set V, and let C be an implicate of $\mathcal{F}$. Then there is a unique prefix of C that is a branch of $\mathrm{RIT}(\mathcal{F}, V)$.

Proof. Let $V = \{p_1, p_2, ...p_n\}$, and proceed by induction on n. Lemma 3 takes care of the base case $n = 1$.

Assume now that the theorem holds for all formulas with at most n variables, and suppose that $\mathcal{F}$ has $n+1$. Let C be an implicate of $\mathcal{F}$, say $C = \{q_{i_1}, q_{i_2}, ...q_{i_k}\}$, where q_{i_j} is either p_{i_j} or $\neg p_{i_j}$, and $i_1 < i_2 < ... < i_j$. We must show that a prefix of C is a branch in $\mathrm{RIT}(\mathcal{F}, V)$, which is the formula

$$\begin{array}{rcc} & & p_1 \ \vee \ \mathrm{RIT}(\mathcal{F}_0, V_1) \\ & & \wedge \\ \mathrm{RIT}(\mathcal{F}, V) & = & \neg p_1 \ \vee \ \mathrm{RIT}(\mathcal{F}_1, V_1) \\ & & \wedge \\ & & \mathrm{RIT}((\mathcal{F}_0 \ \vee \ \mathcal{F}_1), V_1). \end{array}$$

Observe that the induction hypothesis applies to the third branch. Thus, if $i_1 > 1$, there is nothing to prove, so suppose that $i_1 = 1$. Then q_1 is either p_1 or $\neg p_1$. Consider the case $q_1 = p_1$; the proof when $q_1 = \neg p_1$ is entirely similar. By Lemma 4, $C - \{p_1\}$ is an implicate of $\mathcal{F}_0$, and by the induction hypothesis, there is a unique prefix B of $C - \{p_1\}$ that is a branch of $\mathrm{RIT}(\mathcal{F}_0, V_1)$. But then $A = \{p_1\} \cup B$ is a prefix of C that is a branch of $\mathrm{RIT}(\mathcal{F}, V)$.

To complete the proof, we must show that A is the only such prefix of C. Suppose to the contrary that D is another prefix of C that is a branch of $\mathrm{RIT}(\mathcal{F}, V)$. Then either D is a prefix of A or A is a prefix of D; say that D is a prefix of A. Let $D = \{p_1\} \cup E$. Then E is a prefix of B in $\mathrm{RIT}(\mathcal{F}_0, V_1)$, which in turn means that E is a prefix of $C - \{p_1\}$. But we know from the inductive hypothesis that $C - \{p_1\}$ has a unique prefix in $\mathrm{RIT}(\mathcal{F}_0, V_1)$, so $E = B$, so $D = A$. If A is a prefix of D, then it is immediate that E is a prefix of $C - \{p_1\}$ in $\mathrm{RIT}(\mathcal{F}_0, V_1)$, and, as before, $E = B$, and $D = A$. □

The corollaries below are immediate because of the uniqueness of prefixes of implicates in $\mathrm{RIT}(\mathcal{F}, V)$.

Corollary 1. Every prime implicate of $\mathcal{F}$ is a branch in $\mathrm{RIT}(\mathcal{F}, V)$. □

Corollary 2. Every subsuming implicate (including any prime implicate) of a branch in $\mathrm{RIT}(\mathcal{F}, V)$ contains the literal labeling the leaf of that branch. □

5 An Algorithm for Computing ri-Tries

In this section, an algorithm that produces ri-tries is developed using pseudo-code. The algorithm relies heavily on Lemma 2, which states that $\mathrm{Imp}(\mathcal{F}_0 \vee \mathcal{F}_1) = \mathrm{Imp}(\mathcal{F}_0) \cap \mathrm{Imp}(\mathcal{F}_1)$; i.e., the branches produced by the third conjunct of the RIT operator are precisely the branches that occur in both of the first two (ignoring, of course, the root labels p_i and $\neg p_i$). The algorithm makes use of this lemma rather than directly implementing the RIT operator; in particular, the recursive call $\mathrm{RIT}((\mathcal{F}[0/p_i] \ \vee \ \mathcal{F}[1/p_i]), V - \{p_i\})$

is avoided. This is significant because that call doubles the size of the formula *along a single branch*.

No attempt was made to make the algorithm maximally efficient. For clarity, the algorithm is designed so that the first two conjuncts of the RIT operator are constructed in their entirety, and then the third conjunct is produced by parallel traversal of the first two.

The algorithm employs two functions: *rit* and *buildzero*. The nodes of the trie consist of five fields: *label*, which is the name of the literal that occurs in the node; *parent*, which is a pointer to the parent of the node; and *plus*, *minus*, and *zero*, which are pointers to the three children. The function rit recursively builds the first two conjuncts of the RIT operator and then calls buildzero, which recursively builds the third conjunct from the first two.

The reader may note that the algorithm builds a ternary tree rather than an n-ary trie. The reason is that the construction of the subtree representing the third conjunct of the RIT operator sets the label of the root to 0. This is convenient for the abstract description of the algorithm; it is straightforward but tedious to write the code without employing the zero nodes.

Observations.

1. Recall that each (sub-)trie represents the disjunction of the label of its root with the conjunction of the sub-tries rooted at its children.
2. If either of the first two sub-tries are 1, then that sub-trie is empty. The third is also empty since it is the intersection of the first two.
3. If any child of a node is 0, then the node reduces to a leaf. In practice, in the algorithm, this will only occur in the first two branches.
4. If both of the first two sub-tries are leaves, then they are deleted by **SR8** and the root becomes a leaf.
5. No pseudocode is provided for the straightforward routines “makeleaf”, “leaf”, and “delete”. The first two are called on a pointer to a trienode, the third is called on a trienode.

The ri-Trie Algorithm.

```
declare( structure trienode(      lit: label,
                                parent: ↑trienode,
                                  plus: ↑trienode,
                                 minus: ↑trienode,
                                  zero: ↑trienode);

         RItrie:trienode);


input(G);                    {The logical formula G has variables p_1, p_2, ..., p_n}
RItrie ← rit(G, 1, 0);


function rit(G: wff, polarity, varindex: integer): ↑ trienode;
    N ← new(trienode);
    if varindex=0 then n.lit ← 0                          {root of entire trie is 0}
        else if polarity = 0 then N.lit ← −p_varindex
                else N.lit ← p_varindex
    Gplus ← G[0/p_varindex];
    Gminus ← G[1/p_varindex];
    if (Gplus = 0 or Gminus = 0)
        then makeleaf(N); return(↑N);                          {Observation 3.}
    if Gplus = 1 then N.plus ← nil; N.zero ← nil           {Observation 2.}
        else N.plus ← rit(Gplus, 1, varindex+1);
            if N.plus.lit = 1 then delete(N.plus↑); N.plus ← nil;
    if Gminus = 1 then N.minus ← nil; N.zero ← nil         {Observation 2.}
        else N.minus ← rit(Gminus, 0, varindex+1);
            if N.minus.lit = 1 then delete(N.minus↑)
            if N.plus = nil then N.lit ← 1; makeleaf(N);
    if (leaf(N.plus) and leaf(N.minus))                     {Observation 4.}
        then delete(N.plus); delete(N.minus); makeleaf(N); return(↑N);
    if (N.plus ≠ nil and N.minus ≠ nil)
        then N.zero ← buildzero(N.plus, N.minus);
    N.zero↑.parent ← ↑N;
    N.zero↑.lit ← 0
    return(↑N);
end rit;
```

```
function buildzero(N1, N2, ↑trienode): ↑trienode;
    Nzero ← new(trienode);
    Nzero.lit ← N1↑.lit;
    if leaf(N1) then Nzero.(plus, minus, zero) ← N2↑.(plus, minus, zero);
        return(↑Nzero)
    if leaf(N2) then Nzero.(plus, minus, zero) ← N1↑.(plus, minus, zero);
        return(↑Nzero);

    if (N1↑.plus = nil or N2↑.plus = nil)
        then Nzero.plus ← nil
        else Nzero.plus ← buildzero(N1↑.plus, N2↑.plus);
    if (N1↑.minus = nil or N2↑.minus = nil)
        then Nzero.minus ← nil
        else Nzero.minus ← buildzero(N1↑.minus, N2↑.minus);
    if (N1↑.zero = nil or N2↑.zero = nil)
        then Nzero.zero ← nil
        else Nzero.zero ← buildzero(N1↑.zero, N2↑.zero);
    if leaf(↑Nzero) then delete(↑Nzero); return(nil)
        else begin
            if Nzero.plus ≠ nil then Nzero.plus↑.parent ← ↑Nzero;
            if Nzero.minus ≠ nil then Nzero.minus↑.parent ← ↑Nzero;
            if Nzero.zero ≠ nil then Nzero.zero↑.parent ← ↑Nzero;
            return(↑Nzero)
        end

end buildzero.
```

A final straightforward observation is that by employing the dual of the Tri operator (say DTri),

$$\mathrm{DTri}(\mathcal{F}, p) \;=\; (p \wedge \mathcal{F}[1/p]) \vee (\neg p \wedge \mathcal{F}[0/p]) \vee (\mathcal{F}[1/p] \wedge \mathcal{F}[0/p]).$$

tries that store implicants may be analogously built.

6 Future Work

The ri-trie has the very nice property that no matter how large the trie is, any query can be answered in time linear in the size of the query. This gives rise to a number of questions. Under what circumstance will the ri-trie be small enough[8] to be practical? How easy is it to maintain and update ri-tries? The size of an ri-trie is surely dependent on the variable ordering — are there good heuristics for setting the variable ordering? Might a "forest" of ri-tries be effective?

An ri-trie is considerably smaller than the full implicate trie because whenever a prefix of an implicate is also an implicate, the trie may be truncated at the last node

[8] In this context, a large room full of large hard drives qualifies as small enough.

of the prefix. But knowing that any subset of a clause is an implicate is enough to determine that the clause is itself an implicate. Can this fact be used to reduce the trie further? Are there other insights that might provide further size reduction?

The answers to these questions will assist with implementation, but the results of this paper are adequate to begin experiments with ri-tries.

References

1. Bryant, R. E., Symbolic Boolean manipulation with ordered binary decision diagrams, *ACM Comput. Surv.* **24, 3** (1992), 293–318.
2. Bibel, W. On matrices with connections. *J. ACM* **28**,4 (1981), 633 – 645.
3. Cadoli, M., and Donini, F. M., A survey on knowledge compilation, *AI Commun.* **10** (1997), 137–150.
4. Chin, R. T., and Dyer, C. R. Model-based recognition in robot vision. *ACM Computing Surveys* 18(1) (Mar. 1986) 67–108.
5. Coudert, O. and Madre, J. Implicit and incremental computation of primes and essential implicant primes of boolean functions. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, (1992) 36-39.
6. Coudert, O. and Madre, J. A new graph based prime computation technique. In *Logic Synthesis and Optimization* (T. Sasao, Ed.), Kluwer (1993), 33–58.
7. D'Agostino, M., Gabbay, D. M., Hähnle, R., and Posegga, J., *Handbook of Tableau Methods*, Kluwer Academic Publishers, 1999.
8. Darwiche, A., Compiling devices: A structure-based approach, Proc. *Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR98)*, Morgan-Kaufmann, San Francisco (1998), 156–166.
9. Darwiche, A., Model based diagnosis using structured system descriptions, *Journal of A.I. Research*, **8**, 165-222.
10. Darwiche, A., Decomposable negation normal form, *J.ACM* **48,4** (2001), 608–647.
11. Darwiche, A. and Marquis, P., A knowledge compilation map, *J. of AI Research* **17** (2002), 229–264.
12. Davis, M. and Putnam, H. A computing procedure for quantification theory. *J.ACM*, **7** (1960), 201–215.
13. de Kleer, J. Focusing on probable diagnosis. *Proc. of AAAI-91, Anaheim, CA*, pages 842–848, 1991.
14. de Kleer, J. An improved incremental algorithm for computing prime implicants. *Proceedings of AAAI-92*, San Jose, CA, (1992) 780–785.
15. de Kleer, J., Mackworth, A. K., and Reiter, R. Characterizing diagnoses and systems, *Artificial Intelligence*, 32 (1987), 97–130.
16. de Kleer, J. and Williams, B. Diagnosing multiple faults, *Artificial Intelligence*, 56:197–222, 1987.
17. Fitting, M., *First-Order Logic and Automated Theorem Proving (2^{nd} ed.)*, Springer-Verlag, New York, (1996).
18. Forbus, K.D. and de Kleer, J., *Building Problem Solvers*, MIT Press, Cambridge, Mass. (1993).
19. Gomes, C. P., Selman, B., and Kautz, H., Boosting combinatorial search through randomization, *Proc. AAAI-98* (1998), Madison, Wisconsin, 431-437.
20. Hai, L. and Jigui, S., Knowledge compilation using the extension rule, *J. Automated Reasoning*, 32(2), 93-102, 2004.

21. Hähnle, R., Murray N.V., and Rosenthal, E. Normal Forms for Knowledge Compilation. *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, (ISMIS '05, Z. Ras ed.), Lecture Notes in Computer Science, Springer (to appear).
22. Jackson, P. and Pais, J., Computing prime implicants. *Proceedings of the 10th International Conference on Automated Deductions*, Kaiserslautern, Germany, July, 1990. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 449 (1990), 543-557.
23. Jackson, P. Computing prime implicants incrementally. *Proceedings of the 11th International Conference on Automated Deduction*, Saratoga Springs, NY, June, 1992. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 607 (1992) 253-267.
24. Kautz, H. and Selman, B., A general framework for knowledge compilation, in *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK)*, Kaiserslautern, Germany (July, 1991).
25. Kautz, H. and Selman, B., Proceedings of the Workshop on Theory and Applications of Satisfiability Testing, *Electronic Notes in Discrete Mathematics* **9** (2001), Elsevier Science Publishers.
26. Kean, A. and Tsiknis, G. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation* **9** (1990), 185-206.
27. Kean, A. and Tsiknis, G. Assumption based reasoning and clause management systems. *Computational Intelligence* **8**,1 (1992), 1–24.
28. Loveland, D.W. *Automated Theorem Proving: A Logical Basis*. North-Holland, New York, (1978).
29. Marquis, P., Knowledge compilation using theory prime implicates, Proc. *Int'l Joint Conf. on Artificial Intelligence (IJCAI)* (1995), Morgan-Kaufmann, San Mateo, Calif, 837-843.
30. Morrison, D.R. PATRICIA — practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, **15**,4, 514–34, 1968.
31. Murray, N.V. and Rosenthal, E. Inference with path resolution and semantic graphs. *J. ACM* **34**,2 (1987), 225–254.
32. Murray, N.V. and Rosenthal, E. Dissolution: making paths vanish. *J.ACM* **40**,3 (July 1993), 504–535.
33. Murray, N.V. and Rosenthal, E. On the relative merits of path dissolution and the method of analytic tableaux, Theoretical Computer Science 131 (1994), 1-28.
34. Murray N.V. and Rosenthal, E. Duality in Knowledge Compilation Techniques. *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, (ISMIS '05, Z. Ras ed.), Lecture Notes in Computer Science, Springer (to appear).
35. Murray, N.V. and Ramesh, A. An application of non-clausal deduction in diagnosis. *Proceedings of the Eighth International Symposium on Artificial Intelligence*, Monterrey, Mexico, October 17-20, 1995, 378–385.
36. Murray, N.V., Ramesh, A. and Rosenthal, E. The semi-resolution inference rule and prime implicate computations. *Proceedings* of the *Fourth Golden West International Conference on Intelligent Systems*, San Fransisco, CA, (June 1995) 153–158.
37. Murray, N.V. and Rosenthal, E. "Tableaux, Path Dissolution, and Decomposable Negation Normal Form for Knowledge Compilation." *Proceedings* of the *International Conference TABLEAUX 2003 – Analytic Tableaux and Related Methods*, Rome, Italy, September 2003. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 2796, 165-180.
38. Nelson, R. J. 'Simplest normal truth functions', *Journal of Symbolic Logic* **20**, 105-108 (1955).
39. Ngair, T. A new algorithm for incremental prime implicate generation. *Proc of IJCAI-93*, Chambery, France, (1993).
40. Przymusinski, T. C. An algorithm to compute circumscription. *Artificial Intelligence* **38** (1989), 49-73.

41. Ramesh, A. D-trie: A new data structure for a collection of minimal sets. Technical Report SUNYA-CS-95-04, December 28, 1995.
42. Ramesh, A., Becker, G. and Murray, N.V. CNF and DNF considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning* **18**,3 (1997), Kluwer, 337–356.
43. Ramesh, A. and Murray, N.V. An application of non-clausal deduction in diagnosis. *Expert Systems with Applications* **12**,1 (1997), 119-126.
44. Ramesh, A. and Murray, N.V. Parameterized Prime Implicant/Implicate Computations for Regular Logics. *Mathware & Soft Computing*, Special Issue on Deduction in Many-Valued Logics IV(2):155-179, 1997.
45. Reiter, R. A theory of diagnosis from first principles. *Artificial Intelligence*, **32** (1987), 57-95.
46. Reiter, R. and de Kleer, J. Foundations of assumption-based truth maintenance systems: preliminary report. *Proceedings of the 6th National Conference on Artificial Intelligence*, Seattle, WA, (July 12-17, 1987), 183-188.
47. Sieling, D., and Wegener, I., Graph driven BDDs – a new data structure for Boolean functions, *Theor. Comp. Sci.* **141** (1995), 283-310.
48. Slagle, J. R., Chang, C. L. and Lee, R. C. T. A new algorithm for generating prime implicants. *IEEE transactions on Computers* **C-19**(4) (1970), 304-310.
49. Selman, B., and Kautz, H., Knowledge compilation and theory approximation, *J.ACM* **43,2** (1996), 193-224.
50. Selman, B., Kautz, H., and McAllester, D., Ten challenges in propositional reasoning and search, *Proc of IJCAI-97*, Aichi, Japan (1997).
51. Strzemecki, T. Polynomial-time algorithm for generation of prime implicants. *Journal of Complexity* **8** (1992), 37-63.
52. Ullman, J.D. *Principles of Database Systems*, Computer Science Press, Rockville, 1982.