REDUCED IMPLICATE TRIES AND LOGICAL OPERATIONS

Andrew Matusiewicz, Neil V. Murray
ILS Inst., Department of Computer Science, State University of New York, Albany, NY 12222, USA
andrew.matusiewicz@gmail.com, nvm@cs.albany.edu
Erik Rosenthal
Department of Mathematics, University of New Haven, West Haven, CT 06516, USA
erosenthal@newhaven.edu

Keywords: Knowledge compilation; reduced implicate trie; updates.

Abstract: The reduced implicate trie (ri-trie), introduced in (Murray and Rosenthal, 2005), is a target language for knowledge compilation. It has the property that any query can be processed in time linear in the size of the query. In this paper, conjunction and negation are developed as update operations for ri-tries that do not require recompilation. Conjunction has been implemented, and experimental results, though preliminary, are promising. Conjoining a formula to an existing ri-trie by compiling the new formula and conjoining the tries is generally more efficient than compiling the conjunction of the two formulas.

1 INTRODUCTION

Several investigators have represented knowledge bases as propositional theories, typically as sets of clauses. However, since the question, Does \( \mathcal{NP} = \mathcal{P} \)? remains open — i.e., there are no known polynomial algorithms for problems in the class \( \mathcal{NP} \) — the time to answer queries is (in the worst case) exponential. The reduced implicate trie was developed (Murray and Rosenthal, 2005) as a solution to a problem posed by Kautz and Selman (Kautz and Selman, 1991). Their idea, known as knowledge compilation, was to pay the exponential penalty once by compiling the knowledge base into a target language that would guarantee fast response to queries. They specified that the size of the target language be polynomial in the size of the original theory, and that query response time be polynomial in the size of the compiled theory. The result would then be polynomial response time to all queries.

The reduced implicate trie (ri-trie) takes a different approach: Admit large compiled theories on which queries can be answered quickly. It has been shown that ri-tries guarantee response time linear in the size of the query. Thus queries of any knowledge base that can be “practically compiled” — i.e., can be built in reasonable time and space\(^1\) — can always be answered quickly.

In this paper, three update operations for the ri-trie that do not require recompilation are described. They are negation — i.e., finding the ri-trie of \( \neg F \) from the ri-trie of \( F \), conjunction and union — i.e., finding, respectively, the ri-trie of the conjunction and the union of two ri-tries.

Reduced implicate tries are reviewed in Section 2. In Sections 3.1 and 3.2, operations on and between ri-tries are introduced, and in Section 3.4, a preliminary implementation is described.

2 REDUCED IMPLICATE TRIES

The reader is assumed to be familiar with the terms atom, literal, clause, conjunctive normal form (CNF), implicate, and prime implicate. Recall that asking whether a given clause is entailed by a formula is equivalent to the question, Is the clause an implicate of the formula? The reader is also assumed to be familiar with the trie data structure, which has been to represent logical formulas, including sets of prime implicates (Reiter and de Kleer, 1987). 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.

\(^1\)Reasonable is a subjective term, presumably defined by the end user.
A tautology is logically equivalent to the empty sentence (empty conjunction) and thus has no implicates. A contradiction, on the other hand, is logically equivalent to the empty clause (empty disjunction). Thus all clauses are implicates, and the empty clause is the only prime implicate.

In this paper, we assume that a variable ordering has been selected, and that nodes along a branch are labeled consistently with that ordering.

A reduced implicate trie (ri-trie) is a trie whose branches represent the relatively prime implicants (Murray and Rosenthal, 2007a). If $\mathcal{F}$ is a logical formula, then a relatively prime implicate is one for which no proper prefix (with respect to the variable formula, then a relatively prime implicate is one for which no proper prefix (with respect to the given variable ordering). Thus all clauses are implicates, and the empty clause is the only prime implicate.

The ri-trie of a logical formula $\mathcal{F}$ can be obtained with the recursively defined RIT operator, introduced in (Murray and Rosenthal, 2005).

$$\text{RIT}(\mathcal{F}, V) = \begin{cases} \mathcal{F} & V = \emptyset \\ v_i \lor \text{RIT}(\mathcal{F}[0/v_i], V - \{v_i\}) & v_i \in V \\ \neg v_i \lor \text{RIT}(\mathcal{F}[1/v_i], V - \{v_i\}) & v_i \in V \\ \text{RIT}(\mathcal{F}[0/v_i] \lor \mathcal{F}[1/v_i], V - \{v_i\}) & \end{cases}$$

Note that the third conjunct of RIT is RIT of the disjunction of the first two. As a result, the next lemma tells us that the branches of the third subtrie are precisely those that appear in both of the first two. The notation $\text{Imp}(\mathcal{F})$ is used for the set of all implicates of $\mathcal{F}$.

**Lemma 1** Given logical formulas $\mathcal{F}$ and $\mathcal{G}$, $\text{Imp}(\mathcal{F}) \cap \text{Imp}(\mathcal{G}) = \text{Imp}(\mathcal{F} \lor \mathcal{G})$. □

Given two formulas $\mathcal{F}$ and $\mathcal{G}$, fix an ordering of the union of their variable sets, and let $\mathcal{T}_\mathcal{F}$ and $\mathcal{T}_\mathcal{G}$ be the corresponding ri-tries. The intersection of $\mathcal{T}_\mathcal{F}$ and $\mathcal{T}_\mathcal{G}$ is defined to be the ri-trie (with respect to the given variable ordering) that represents the intersection of the implicate sets. The intersection of two tries (with the same variable ordering) is produced by the INT operator introduced in (Murray and Rosenthal, 2007b).

**Theorem 1** Let $\mathcal{T}_\mathcal{F}$ and $\mathcal{T}_\mathcal{G}$ be the respective ri-tries of $\mathcal{F}$ and $\mathcal{G}$ (with the same variable ordering). Then $\text{INT}(\mathcal{T}_\mathcal{F}, \mathcal{T}_\mathcal{G})$ is the intersection of $\mathcal{T}_\mathcal{F}$ and $\mathcal{T}_\mathcal{G}$; in particular, $\text{INT}(\mathcal{T}_\mathcal{F}, \mathcal{T}_\mathcal{G})$ is the ri-trie of $\mathcal{F} \lor \mathcal{G}$ (with respect to the given variable ordering). □

Theorem 1 provides a formal basis for a definition of the RIT operator that produces ri-tries using intersection. It is obtained from the earlier definition by replacing the third conjunct by $\text{INT}(\mathcal{F}[0/v_i], V - \{v_i\})$, $\text{INT}(\mathcal{F}[1/v_i], V - \{v_i\})$).

## 3 UPDATING ri-TRIES

It is typical in the knowledge compilation paradigm to assume that the intractable part of the processing is done only once (or at least not very often). In the absence of an efficient updating technology, this favors knowledge bases that are stable; i.e., a single compilation is expected to provide a repository that remains useful over a large number of queries. The original knowledge base can always be modified and then recompiled, but in general this is expensive. As a result, updates that can be installed into the compiled knowledge base without recompiling have the potential to widen applicability considerably.

Four update operations for ri-tries were introduced in (Murray and Rosenthal, 2007b): Intersection, substitution of a truth constant, variable reordering, and conjunction of a clause. Two update operations are described in Sections 3.1 and 3.2: negation and conjunction.

### 3.1 Negation

The RIT operator by itself produces a trie for a formula $\mathcal{F}$ in which every leaf is labeled 0 or 1. Truth functional simplifications then yield the desired ri-trie. Without the simplifications, the trie is called a constant leaf trie (cl-trie). Merely swapping the 0’s and 1’s in a cl-trie will produce a representation of $\neg \mathcal{F}$, but this is not the cl-trie of $\neg \mathcal{F}$. The difficulty is the third conjunct. The NEG operator recursively simplifies the first two conjuncts and then applies the INT operator to produce the third. Below, the formal definition of NEG uses the representation of the trie $\mathcal{T}$ rooted at $p_i$ as the 4-tuple $(p_i, T^+, T^-, T^0)$, see (Murray and Rosenthal, 2007b).

$$\text{NEG}(r, 0, 0, 0) = \langle \neg r, 0, 0, 0 \rangle$$

$$\text{NEG}(r, p, 0, 0) = \langle r, \neg p, 0 \rangle$$

$$\text{NEG}(r, 0, \neg p, 0) = \langle r, p, 0 \rangle$$

$$\text{NEG}(r, p, (\neg p, T^+, T^-, T^0), (0, T^+, T^-, T^0)) = \langle r, \neg p, T^+, T^-, T^0, 0 \rangle$$

$$\text{NEG}(r, (p, T^+, T^-, T^0), p, (0, T^+, T^-, T^0)) = \langle \neg p, T^+, T^-, T^0, 0, \rangle$$

$$\text{NEG}(r, (p, T^+, T^-, T^0), 0, 0) = \langle r, \neg p, T^+, T^-, T^0, 0, \rangle$$

$$\text{NEG}(r, \neg p, T^+, T^-, T^0, 0, \rangle = \langle r, \neg p, T^+, T^-, T^0, 0, \rangle$$

$$\text{NEG}(r, \neg p, T^+, T^-, T^0, 0, \rangle = \langle r, \neg p, T^+, T^-, T^0, 0, \rangle$$
NEG\( (r, 0, (\neg p, T^+, T^-, T^0), 0) \) = \\
\langle r, p, \text{NEG}(\neg p, T^+, T^-, T^0), \text{NEG}(0, T^+, T^-, T^0) \rangle \\
\text{otherwise} \quad \text{NEG}(r, T^+, T^-, T^0) = \\
\langle \text{NEG}(T^+), \text{NEG}(T^-), \text{INT}(\text{NEG}(T^+), \text{NEG}(T^-)) \rangle \\

For example, the \( ri \)-trie for \( p \lor q \) is shown on the left In Figure 1, and the \( cl \)-trie is on the right. The constants of the \( cl \)-trie are toggled on the left In Figure 2, and the \( cl \)-trie of \( \neg (p \lor q) \) is shown on the right.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{figure1.png}
\caption{The \( ri \)-trie and \( cl \)-trie for \( p \lor q \).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{figure2.png}
\caption{Negating the \( cl \)-trie for \( p \lor q \).}
\end{figure}

**Theorem 2** Let \( T_f \) be the \( ri \)-trie for formula \( f \) under a given variable ordering. Then \( \text{NEG}(T_f) \) is the \( ri \)-trie for \( \neg f \).

**3.2 Conjunction**

Suppose we have \( ri \)-tries \( T_f \) and \( T_g \) for formulas \( f \) and \( g \), respectively. We would like to compute the \( ri \)-trie for \( f \land g \).

In the case of a conjunction, any implicates of either conjunct is an implicates of the conjunction. However, the conjunction may have implicates that are entailed by neither of the conjuncts. (E.g., \textit{false} is an implicate of \( A \land \overline{A} \) but not of \( A \) nor of \( \overline{A} \).) In general, the implicates of a conjunction are a superset of the union of the implicates sets of the conjuncts.

In order to conjoin the \( ri \)-tries for formulas \( f \) and \( g \), first suppose that their first two subtrees can be conjoined pairwise. By definition, their intersection must represent the third subtree of the \( ri \)-trie for \( (f \land g) \). This subtree represents all implicates of \( (f \land g) \) that do not contain \( p \).

The operator \textsc{INT} takes two \( ri \)-tries as arguments under the assumption that they have the same variable ordering. The \textsc{CONJ} operator below employs the same convention and \( 4 \)-tuple notation.

\[
\text{CONJ}(T_f, T_g) = \begin{cases} 
T_f & T_g = \emptyset \lor \text{leaf}(T_g) \\
T_g & T_f = \emptyset \lor \text{leaf}(T_f) \\
\langle r, B^+, B^-, B^0 \rangle & \text{otherwise}
\end{cases}
\]

with \( r \) as the root label of both \( T_f \) and \( T_g \), and

\[
B^+ = \text{CONJ}(T_f^+, T_g^+), \quad B^- = \text{CONJ}(T_f^-, T_g^-), \quad \text{and} \quad B^0 = \text{INT}(B^+, B^-)
\]

**Theorem 3** Let \( T_f \) and \( T_g \) be the \( ri \)-tries for formulas \( f \) and \( g \), respectively. Then \( \text{CONJ}(T_f, T_g) \) is the \( ri \)-trie for \( f \land g \).

**3.3 Structure Sharing**

It is often convenient to assume that \( ri \)-tries are represented as trees. Consider however, an atomic formula whose variable has a high index. If the full \( ri \)-trie is represented as a tree, the size is exponential, as can be seen in the following lemma.

**Lemma 2** The \( ri \)-trie of a formula \( f = v_j \) with variables ordered by index has \( 3^{j-1} + \frac{3}{2} \) nodes. \( \square \)

In Figure 3, the \( ri \)-trie for \( v_3 \) is shown under each of the three representation schemes discussed in this section. The variable ordering is by index number.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{figure3.png}
\caption{The three \( ri \)-trie representations for \( v_3 \).}
\end{figure}

Structure sharing was applied to all identical subtrees in the development of our prototype compiler,
and the resulting reduction in size was orders of magnitude. In some cases, such as that of Lemma 2, this greedy structure sharing may make the difference between linear and exponential size.

**Lemma 3** The ri-trie of a formula \( F = v_j \) with variables ordered by index has, under greedy structure sharing, \( 3j + 1 \) nodes.

Formally, the nodes in an ri-trie are labeled with literals or the constant 0. The labeling scheme used in our implementation forgoes node labeling and instead uses edges labeled with ‘+’, ‘−’, or ‘0’. Variable indices for the nodes are inferred by the length of the path traversed in arriving at the node. The root is always labeled ‘0’. The child of a ‘+’ edge at level \( j \) has an inferred label of \( v_{j-1} \), the child of a ‘−’ edge at level \( j \) has an inferred label of \( \neg v_{j-1} \), and the child of any ‘0’ edge has an inferred label of ‘0’. We will refer to such a representation as a label-inferred ri-trie.

This convention allows more than just the merging of identical subtrees; subtrees that represent distinct formulas but that are structurally identical can sometimes also be merged. What is required is that one formula can be obtained by renaming the other, and that the renaming can be done merely by adding a constant to the indices of all variables.

**Lemma 4** The label-inferred ri-trie of a formula \( F = v_j \) with variables ordered by index, has, under greedy structure sharing, \( j + 1 \) nodes.

### 3.4 Experiments

The logical operations discussed in Sections 3.1 and 3.2 were added to the prototype compiler. (Disjunction is provided by the INToperator.) The system employs greedy structure sharing with label-inferred ri-tries. In Table 1, each row represents the average of five runs involving two random 3-CNF formulas over 25 variables.

The first two columns are the number of clauses in the two formulas. The next three columns are time in milliseconds for compiling the first formula, compiling the second formula and conjoining it to the ri-trie for the first, and conjoining the two formulas and compiling the conjunction. The last column is the size of the compiled ri-trie in nodes.

In all but one case, it is less costly to update the trie than to recompile with the new formula. The advantage improves as the number of clauses in the first formula increases. The conjecture is that smaller clause sets are, in a sense, more satisfiable; their ri-tries are larger, but the cost of computing them is closer to linear than exponential. Larger clause sets lead to smaller ri-tries that require much more computation, compile times increase and update times decrease.

<table>
<thead>
<tr>
<th># Clauses, 1st formula</th>
<th># Clauses, 2nd formula</th>
<th>Compile time, 1st formula</th>
<th>Compile time, 2nd formula &amp; conjoin</th>
<th>Conjoin both formulas &amp; compile</th>
<th># Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1</td>
<td>4898</td>
<td>4336</td>
<td>5320</td>
<td>139295</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>4754</td>
<td>2722</td>
<td>5702</td>
<td>127324</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>4822</td>
<td>4442</td>
<td>4364</td>
<td>70350</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>5520</td>
<td>2750</td>
<td>5766</td>
<td>82349</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>5538</td>
<td>1408</td>
<td>6294</td>
<td>56549</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>5626</td>
<td>2434</td>
<td>4740</td>
<td>12497</td>
</tr>
<tr>
<td>75</td>
<td>1</td>
<td>3056</td>
<td>958</td>
<td>2968</td>
<td>14971</td>
</tr>
<tr>
<td>75</td>
<td>5</td>
<td>3070</td>
<td>240</td>
<td>3038</td>
<td>8928</td>
</tr>
<tr>
<td>75</td>
<td>20</td>
<td>3072</td>
<td>1444</td>
<td>2356</td>
<td>1615</td>
</tr>
<tr>
<td>85</td>
<td>1</td>
<td>2244</td>
<td>78</td>
<td>2112</td>
<td>1963</td>
</tr>
<tr>
<td>85</td>
<td>5</td>
<td>2276</td>
<td>66</td>
<td>2348</td>
<td>1692</td>
</tr>
<tr>
<td>85</td>
<td>20</td>
<td>2260</td>
<td>1360</td>
<td>1794</td>
<td>159</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1886</td>
<td>24</td>
<td>1826</td>
<td>219</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>1866</td>
<td>28</td>
<td>1698</td>
<td>79</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>1860</td>
<td>1332</td>
<td>1754</td>
<td>8</td>
</tr>
</tbody>
</table>

These results are based on a prototype only and are very preliminary. But they indicate (not surprisingly) that updating operations are a potentially useful alternative to recompiling.

### REFERENCES


