A New Algorithm for Computing Theory Prime Implicates Compilations

Pierre Marquis and **Samira Sadaoui**

CRIN-CNRS and INRIA-Lorraine
Bâtiment LORIA - Campus Scientifique, B.P. 239
F-54506 Vandœuvre-lès-Nancy Cedex, FRANCE
E-mail: {marquis, sadaoui}@loria.fr

Abstract

We present a new algorithm (called TPI/BDD) for computing the theory prime implicates compilation of a knowledge base Σ. In contrast to many compilation algorithms, TPI/BDD does not require the prime implicates of Σ to be generated. Since their number can easily be exponential in the size of Σ, TPI/BDD can save a lot of computing. Thanks to TPI/BDD, we can now conceive of compiling knowledge bases impossible to before.

Introduction

Many problems from various AI fields require propositional reasoning. Unfortunately, the computational complexity of querying a propositional knowledge base Σ is high (CLAUSE ENTAILMENT is co-NP complete); accordingly, every query answering algorithm runs in time exponential in the size of Σ in the worst case.

In order to circumvent the intractability of propositional reasoning, several approaches have been proposed so far. Among them is *equivalence-preserving compilation*. Compiling a knowledge base Σ consists in translating (compiling) it during an off-line preprocessing phase into an equivalent form (a compilation) from which on-line query answering is tractable. This approach allows on-line response time to be sped up as soon as the compilation cost is amortized over a sufficiently large set of queries.

In the following, we are concerned with *theory prime implicates compilations* of knowledge bases. In (Marquis 1995), the main advantages of such compilations w.r.t. on-line reasoning have been exhibited. Hereafter, we complete this study by focusing on the off-line compiling process. Indeed, equivalence-preserving compilation can prove helpful only if *the compilation time is reasonable for off-line computation* (del Val 1994). This motivates the search for efficient algorithms (w.r.t. off-line compiling).

From a theoretical point of view, it is very unlikely that a polynomial time compilation function exists; indeed, for every equivalence-preserving compilation technique, the size of the compiled knowledge base is *not polynomial* in the size of the original knowledge base in the worst case, unless $NP \subseteq P/poly$ (this is a direct consequence of Theorem 5 from (Kautz & Selman 1992)). Accordingly, our main objective is to *increase the number of knowledge bases which can be compiled in practice*, only.

To this end, we propose a new algorithm (called TPI/BDD) for computing the theory prime implicates compilation of a knowledge base Σ w.r.t. a tractable theory Φ. Actually, TPI/BDD computes the *minimal implicates* of Σ w.r.t. Φ, which coincide with its theory prime implicates each time Φ is a logical consequence of Σ.

TPI/BDD is based on PI/BDD (Madre & Coudert 1991), one of the most efficient prime implicates algorithm we found in the literature. Both algorithms rely on a BDD (Binary Decision Diagram) representation of Σ; particularly, they do not require Σ to be put into CNF in a preliminary step and they avoid redundancy by sharing identical formulas embedded at different places in Σ.

The key of TPI/BDD is to compute the minimal implicates of Σ recursively from the minimal implicates of its restriction formulas (i.e. the formulas rooted « below » Σ in the BDD). Thus, every implicate of a restriction formula of Σ which is not minimal w.r.t. $\models_\Phi$ can be removed as soon as it is generated. Accordingly, TPI/BDD *does not require the prime implicates of Σ to be computed;* this contrasts with the algorithm TPI pointed out in (Marquis 1995) *and with many prime implicates-based compilation functions*, including PI (Reiter & de Kleer 1987), FPI_0, FPI_1 (del Val 1994) and those given in (Mathieu & Delahaye 1994). Since the number of prime implicates of Σ can easily be exponential in the size of Σ, TPI/BDD can save a lot of computing. Thus, we show that the number of clauses generated by TPI/BDD is always smaller than the number of clauses generated by PI/BDD; as a consequence, the number of deduction checks (w.r.t. Φ) performed by TPI/BDD can be significantly lower than the number of standard subsumption checks performed by PI/BDD (even if this is not always the case). We present some preliminary experimental results providing evidence for the substantial computational savings achievable with this new algorithm. Because we can now conceive of compiling knowledge bases impossible to before, TPI/BDD gives to the theory prime implicates compilation technique a decisive advantage over prime implicates-based techniques.

The rest of this paper is organized as follows. After some formal preliminaries, prime implicates, theory prime implicates and minimal implicates are successively defined. On this ground, TPI/BDD is presented and illustrated. Finally, the performances of TPI/BDD are analyzed and compared with those offered by PI/BDD.

Formal Preliminaries

Let PS be a finite set of propositional variables. $PROP_{PS}$ denotes the propositional language built in the usual way upon PS and the connectives.

The notions of interpretation over $PROP_{PS}$, model, insatisfiable formula, valid formula, logical entailment $\models$ and logical equivalence $\equiv$ are defined in the usual way. $PROP_{PS} / \equiv$ denotes the quotient set of $PROP_{PS}$ induced by $\equiv$. We shall note 0 (resp. 1) the representative of the class of all the insatisfiable (resp. valid) formulas in $PROP_{PS} / \equiv$.

Let Φ be a finite set of formulas. $\models_\Phi$ denotes the pre-ordering over $PROP_{PS}$ defined by $\sigma \models_\Phi \tau$ iff $\Phi \models \sigma \Rightarrow \tau$. Let $\equiv_\Phi$ denote the equivalence relation over $PROP_{PS}$ induced by $\models_\Phi$ and defined by $\sigma \equiv_\Phi \tau$ iff $\sigma \models_\Phi \tau$ and $\tau \models_\Phi \sigma$. $PROP_{PS} / \equiv_\Phi$ denotes the quotient set of $PROP_{PS}$ induced by $\equiv_\Phi$.

Given a set of clauses Σ, $\min(\Sigma, \models_\Phi)$ denotes the subset of Σ consisting of all its minimal elements w.r.t. the ordering $\models_\Phi$ over $PROP_{PS} / \equiv_\Phi$. The number of clauses in Σ is noted $\#\Sigma$.

The *Shannon expansion* of a propositional formula ψ around variable x is the formula $(\neg x \wedge \psi_0) \vee (x \wedge \psi_1)$ of $PROP_{PS} / \equiv$, where ψ_0 (resp. ψ_1) is the (so-called *direct restriction*) formula of ψ (w.r.t. x) obtained by replacing every occurrence of x in ψ by 0 (resp. 1). More generally, when S is a set of variables occurring in ψ, σ is a *restriction formula* of ψ (w.r.t. S) iff σ is obtained by substituting (uniformly) in ψ every occurrence of variables from S by 0 (or by 1). Shannon expansion can be iterated until all the variables occurring in ψ are eliminated. The corresponding decomposition tree is called a *Shannon tree* of ψ. In many situations, some nodes in this tree are redundant; they can be removed to produce a directed acyclic graph, called a *Binary Decision Diagram* (BDD) of ψ (see (Bryant 1992) for a survey).

Prime Implicates, Theory Prime Implicates and Minimal Implicates

Let Ψ and Φ be finite sets of formulas:

- A *prime implicate* (pi for short) of Ψ is a clause π s.t.:
  - $\Psi \models \pi$ holds, and
  - for every clause π', if $\Psi \models \pi'$ and $\pi' \models \pi$ hold, then $\pi' \equiv \pi$ holds.
- A *theory prime implicate* (tpi for short) of Ψ w.r.t. Φ is a clause π s.t.:
  - $\Psi \models_\Phi \pi$ holds, and
  - for every clause π', if $\Psi \models_\Phi \pi'$ and $\pi' \models_\Phi \pi$ hold, then $\pi' \equiv_\Phi \pi$ holds.
- A *minimal implicate* (mi for short) of Ψ w.r.t. Φ is a clause π s.t.:
  - $\Psi \models \pi$ holds, and
  - for every clause π', if $\Psi \models \pi'$ and $\pi' \models_\Phi \pi$ hold, then $\pi' \equiv_\Phi \pi$ holds.

$PI(\Psi)$ (resp. $TPI(\Psi, \Phi)$, $MinPI(\Psi, \Phi)$) will denote the set of pis (resp. tpis, resp. mis) of Ψ (resp. w.r.t. Φ).

Clearly enough, the pis of Ψ are the minimal elements w.r.t. $\models$ in the set of all the clauses implied by Ψ (the so-called *implicates* of Ψ). $TPI(\Psi, \Phi)$ is composed of the minimal elements w.r.t. $\models_\Phi$ in the set of all the implicates of $\Psi \cup \Phi$. $MinPI(\Psi, \Phi)$ is composed of the implicates of Ψ which are minimal w.r.t. $\models_\Phi$. Tpis (resp. mis) are considered up to $\equiv_\Phi$: only *one representative per equivalence class* is kept in $TPI(\Psi, \Phi)$ (resp. $MinPI(\Psi, \Phi)$).

Example 1 Let $\Sigma =_{def} \{p \vee q \vee r,\ r \vee s,\ \neg p \vee q,\ p \vee \neg q,\ \neg q \vee \neg r\}$ and $\Phi =_{def} \{\neg p \vee q,\ p \vee \neg q,\ \neg q \vee \neg r\}$.

- $PI(\Sigma) = \{\neg p \vee \neg r,\ \neg p \vee s,\ p \vee r,\ r \vee s,\ \neg p \vee q,\ q \vee r,\ p \vee \neg q,\ \neg q \vee \neg r,\ \neg q \vee s\}$ (up to $\equiv$).
- $TPI(\Sigma, \Phi) = MinPI(\Sigma, \Phi) = \{r \vee s, p \vee r\}$ (up to $\equiv_\Phi$).

The interplay between these three notions is made precise by Proposition 1.

Proposition 1 Let Ψ and Φ be finite sets of formulas.
1) $PI(\Psi) = TPI(\Psi, \{\}) = MinPI(\Psi, \{\})$ (up to $\equiv$).
2) $TPI(\Psi, \Phi) = \min(PI(\Psi \cup \Phi), \models_\Phi)$ (up to $\equiv_\Phi$).
3) $MinPI(\Psi, \Phi) = \min(PI(\Psi), \models_\Phi)$ (up to $\equiv_\Phi$).
4) $TPI(\Psi, \Phi) = MinPI(\Psi \cup \Phi, \Phi)$ (up to $\equiv_\Phi$).

As a straightforward consequence of Proposition 1(2) (resp. 1(3)), the number of tpis (resp. mis) of Ψ w.r.t. any Φ is always lower (or equal) to the number of pis of Ψ.

Corollary Let Ψ and Φ be finite sets of formulas. $\#TPI(\Psi, \Phi) \leq \#PI(\Psi)$ and $\#MinPI(\Psi, \Phi) \leq \#PI(\Psi)$.

Proposition 2 shows that the tpis (resp. mis) of a disjunction of formulas can be computed from the tpis (resp. mis) of its d'sjuncts. Several pi algorithms, including (Slagle, Chang & Lee 1970; Jackson & Pais 1990) are based on it when restricted to the pi situation (i.e. $\Phi = \{\}$).

Proposition 2 Let $\Psi_1, ..., \Psi_p$ and Φ be finite sets of formulas.
1) $TPI(\Psi_1 \vee ... \vee \Psi_p, \Phi) = \min(\{\pi_1 \vee ... \vee \pi_p;\ \pi_i \in TPI(\Psi_i, \Phi) \text{ for } i \in [1 .. p]\}, \models_\Phi)$ (up to $\equiv_\Phi$).
2) $MinPI(\Psi_1 \vee ... \vee \Psi_p, \Phi) = \min(\{\pi_1 \vee ... \vee \pi_p;\ \pi_i \in MinPI(\Psi_i, \Phi) \text{ for } i \in [1 .. p]\}, \models_\Phi)$ (up to $\equiv_\Phi$).

Proposition 3 shows that the pis (resp. mis) of a formula can be computed from the pis (resp. mis) of its direct restriction formulas and the pis (resp. mis) of their disjunction.

Proposition 3 Let Ψ and Φ be finite sets of formulas. Let $(\neg x \wedge \Psi_0) \vee (x \wedge \Psi_1)$ be the Shannon expansion of Ψ around any variable x occurring in Ψ.
1) $PI(\Psi) = PI(\Psi_0 \vee \Psi_1) \cup \{\neg x \vee \pi;\ \pi \in PI(\Psi_1) \text{ and } \forall \pi' \in PI(\Psi_0 \vee \Psi_1), \pi' \not\models \pi\} \cup \{x \vee \pi;\ \pi \in PI(\Psi_0) \text{ and } \forall \pi' \in PI(\Psi_0 \vee \Psi_1), \pi' \not\models \pi\}$ (up to $\equiv$).
2) $MinPI(\Psi, \Phi) = \min(MinPI(\Psi_0 \vee \Psi_1, \Phi) \cup (\{\neg x\} \vee MinPI(\Psi_1, \Phi)) \cup (\{x\} \vee MinPI(\Psi_0, \Phi)), \models_\Phi)$ (up to $\equiv_\Phi$).

Proposition 3(1) (pi situation) is (the dual of) the main property the algorithm PI/BDD relies on. It shows how some subsumption checks can be avoided between clauses π' from the three sets $PI(\Psi_0 \vee \Psi_1)$, $\{\neg x \vee \pi;\ \pi \in PI(\Psi_1)\}$, $\{x \vee \pi;\ \pi \in PI(\Psi_0)\}$. Indeed, such π' are pis (hence, no intra-set subsumption checks are necessary); additionally, every π' in $\{\neg x \vee \pi;\ \pi \in PI(\Psi_1)\}$ (resp. $\{x \vee \pi;\ \pi \in PI(\Psi_0)\}$) contains literal $\neg x$ (resp. x); subsequently, π' can neither subsume nor be subsumed by a clause from $\{x \vee \pi;\ \pi \in PI(\Psi_0)\}$ (resp. $\{\neg x \vee \pi;\ \pi \in PI(\Psi_1)\}$) (i.e. some inter-set subsumption checks can be avoided).

Proposition 3(2) (mi situation) is the key property on which our algorithm TPI/BDD is based. Unlike PI/BDD, both intra-set and inter-set deduction checks (w.r.t. Φ) between clauses from the three sets $MinPI(\Psi_0 \vee \Psi_1, \Phi)$, $\{\neg x \vee \pi;\ \pi \in MinPI(\Psi_1, \Phi)\}$, $\{x \vee \pi;\ \pi \in MinPI(\Psi_0, \Phi)\}$ are required in the general case.

Thanks to Propositions 2 and 3, the pis (resp. mis) of Ψ (w.r.t. Φ) can be characterized from the pis (resp. mis) of its direct restriction formulas Ψ_0 and Ψ_1. Contrastingly, the tpis of Ψ w.r.t. Φ cannot be characterized from the tpis of its restriction formulas. Fortunately, we are interested in computing tpi compilations; in this situation, the tpis of Ψ w.r.t. Φ can be computed as the mis of Ψ w.r.t. Φ since tpis and mis coincide when $\Psi \models \Phi$ (see Proposition 1(4) and Example 1 for illustration).

A New Algorithm for TPI Compilations

Based on complexity considerations, our central motivation is to design a new tpi compilation algorithm which *does not require the pis of Σ to be generated*; indeed, because it does not satisfy this requirement, the algorithm TPI (Marquis 1995) appears as very time consuming, so impractical for many applications.

Additionally, TPI applies to CNF knowledge bases, only. While any formula can be put into CNF in linear time (introducing new variables), (Ngair 1993) shows that the cost of generating additional new pis (containing such new variables) usually outweighs the benefit of the compact encoding (an exponential number of useless pis can be generated). This is also true for tpis: in the general case, the restriction of accepting only CNF formulas as input leads to significant computational overhead. Accordingly, we are looking for a tpi compilation algorithm which *accepts any propositional formula as its input.*

An Outline of PI/BDD

Because tpis (and mis) are direct generalizations of pis (cf. Proposition 1(1)), we turn towards pi algorithms satisfying this last constraint in the objective of extending them to the tpi situation.

Though many pi algorithms can be found in the literature, only one algorithm, namely (Madre & Coudert 1991), fulfills this requirement. Since the algorithm sketched in (Madre & Coudert 1991) was oriented towards the generation of prime implicants (the dual notion of pi), we adapted it to compute pis and called the resulting algorithm PI/BDD.

Interestingly, PI/BDD is based on a BDD representation of Σ; because it avoids redundancy by sharing identical restriction formulas embedded at different places in Σ, it is one of the most efficient pi algorithm we can find (especially when non-normalized knowledge bases are considered).

Basically, PI/BDD relies on Propositions 2 (restricted to the pi situation) and 3(1). Some additional properties allow one to avoid useless computations; in particular, when Ψ_0 and Ψ_1 are the direct restriction formulas of Ψ (w.r.t. x), every clause of $PI(\Psi_0) \cap PI(\Psi_1)$ is a pi of Ψ. More generally, every clause of $PI(\Psi_0)$ (resp. $PI(\Psi_1)$) which is subsumed by a clause of $PI(\Psi_1)$ (resp. $PI(\Psi_0)$) is a pi of Ψ; subsequently, we can remove such clauses from their respective sets before computing $PI(\Psi_0 \vee \Psi_1)$ as $min(\{\pi_0 \vee \pi_1;\ \pi_i \in PI(\Psi_i)$ for $i \in [0 .. 1]\}, \models)$ (see (Madre & Coudert 1991) for details).

TPI/BDD

The key of TPI/BDD is to compute the mis of Σ recursively from the mis of its restriction formulas. Accordingly, every implicate of a restriction formula of Σ which is not minimal w.r.t. $\models_\Phi$ *can be removed as soon as it is generated.* To be more specific, TPI/BDD basically consists in computing recursively the mis of each restriction formula of Σ in a bottom-up way from its BDD representation, thanks to Propositions 2(2) and 3(2).

In order to avoid useless computations, some additional (simple) properties are also used. They generalize to the mis situation the properties used to improve PI/BDD, as evoked above. Thus:

- If π_0 and π_1 are mis of the same formula, performing deduction checks (w.r.t. Φ) between π_0 and π_1 is useless (both are necessarily minimal w.r.t. $\models_\Phi$).
- If clauses π_0 and π_1 are s.t. $\pi_0 \models_\Phi \pi_1$ then, for every clause π, $\pi_0 \vee \pi \models_\Phi \pi_1 \vee \pi$ holds.

These two properties allow us to minimize the number of deduction checks (w.r.t. Φ) that a naive implementation of Proposition 3(2) would require.

We are now ready to describe the algorithm TPI/BDD:

```
Function TPI/BDD(Ψ, Φ)
% Input  : two formulas Ψ and Φ s.t. Ψ ⊨ Φ.
% Output : the set TPI of tpis of Ψ w.r.t. Φ.
1. Compute a BDD of Ψ and put its nodes on a
   list L in a decreasing order w.r.t. their
   indexes
2. For every node n in L do
    Let Ψn be the restriction of Ψ at node n
    If n is a sink node
     then TPI ← {Ψn}
     else a/ let Ψn0 (resp. Ψn1) be the direct
             restriction formula of Ψn at node n0
             (resp. n1)
          b/ let MinPI0 (resp. MinPI1) be the set
             of mis of Ψn0 (resp. Ψn1) and let x
             be the variable labelling node n
          c/ TPI ← MinPI(MinPI0, MinPI1, Φ, x)
3. Return(TPI)
```

```
Function MinPI(MinPI0, MinPI1, Φ, x)
% Input  : the two sets MinPI0 (resp. MinPI1) of
mis of Ψ0 (resp. Ψ1) w.r.t. Φ and the variable x
s.t. (¬x ∧ Ψ0) ∨ (x ∧ Ψ1) is the Shannon expansion of
Ψ around x.
% Output : the set MinPI of mis of Ψ w.r.t. Φ.
1. Set impMinPI0, impMinPI1, nonimpMinPI0,
   nonimpMinPI1 to the empty set
2. For every clause π in MinPI0 do
    If there exists π' in MinPI1 s.t. π' ⊨Φ π
     then impMinPI0 ← {π} ∪ impMinPI0
     else nonimpMinPI0 ← {π} ∪ nonimpMinPI0
3. For every clause π in MinPI1 do
    If there exists π' in nonimpMinPI0 s.t. π' ⊨Φ π
     then impMinPI1 ← {π} ∪ impMinPI1
     else nonimpMinPI1 ← {π} ∪ nonimpMinPI1
4. MinPI1 ← min({π1 ∨ π2; π1 ∈ nonimpMinPI1,
                 π2 ∈ nonimpMinPI1}, ⊨Φ)
5. MinPI2 ← min({x ∨ π; π ∈ nonimpMinPI0} ∪
                {¬x ∨ π; π ∈ nonimpMinPI1}, ⊨Φ)
6. MinPI ← min2(min2(min2(impMinPI0, impMinPI1,
           ⊨Φ), MinPI1), ⊨Φ), MinPI2, ⊨Φ)
7. Return(MinPI)
```

impMinPI0 (resp. impMinPI1) denotes the subset of the mis of MinPI0 (resp. MinPI1) which are implied (w.r.t. Φ) by a mi of MinPI1 (resp. nonimpMinPI0); the complement of impMinPI0 (resp. impMinPI1) in MinPI0 (resp. MinPI1) is denoted nonimpMinPI0 (resp. nonimpMinPI1); min2(MI1, MI2, ⊨Φ) computes min(MI1 ∪ MI2, ⊨Φ) while avoiding intra-set deduction checks (i.e. every element of MI1 (resp. MI2) is assumed minimal w.r.t. ⊨Φ in MI1 (resp. MI2)).

Basically, the correctness of TPI/BDD w.r.t. tpi computation relies on Propositions 1(4), 2(2) and 3(2) and the simple properties given above.

TPI/BDD can be enhanced in several directions. Some improvements concern the BDD generation (step 1/). Particularly, Reduced BDDs (RBDDs for short), also referred to as free Boolean graphs or one-time branching programs (Bryant 1992), can be preferred to (standard) Reduced Ordered BDDs (ROBDDs); indeed, requiring that every path from the root to a sink of the BDD respect the same (total) ordering over the variables of Σ is unnecessary for our purpose. Interestingly, every ROBDD is a RBDD but the converse does not hold, so the size of the smallest RBDD of Σ is always smaller (or equal) to the size of its smallest ROBDD. In practice, instead of pointing out a fixed variable ordering before computing the BDD, the variables of Σ can be ordered *dynamically* in each branch of its Shannon tree while building up this tree, so as to minimize the expected depth of each branch. Such dynamic variable ordering can result in huge computational savings in the BDD generation. Particularly, when Σ is in CNF, branching rules used in the old good Davis&Putnam procedure for SAT (Hooker & Vinay 1995) may apply here advantageously.

Further improvements of TPI/BDD are related to the mis generation itself. Thus, the mis computed at each node can be sorted w.r.t. their size, putting the smallest clauses first in the ordering (we can also use the heuristic given in (Jackson & Pais 1990) instead). The computational cost of this additional processing can be balanced by the savings it convcys at steps 2/ 3/ in MinPI (most general clauses can be found more quickly since the smallest clauses are often the most general ones). Additionally, when Σ is in CNF, TPI/BDD can be boosted by removing in a preliminary step every clause of Σ which is a logical consequence of Φ.

Interestingly, the BDD produced at step 1/ of TPI/BDD can be viewed as a compact encoding of an *implicant cover* of Σ, hence as a compilation of Σ (Schrag 1996). In many situations, the size of this BDD is exponentially greater than the size of the tpi compilation. However, if the size of the BDD is sufficiently low, we can decide to stop TPI/BDD at step 1/ and use the BDD as a compilation. We can also decidc to pursuc the tpi compiling while storing the BDD as an additional compilation of Σ. In this case, interleaving (or parallelizing) the search through both compilations can be a way to improve on-line query answering; indeed, implicant covers are specialized in queries π which are *not* consequences of Σ (pointing out an implicant π' of Σ s.t. π' $\not\models$ π is sufficient to return a « no » answer) while tpi compilations are specialized in queries π which are consequences of Σ (pointing out a tpi π' of Σ s.t. π' ⊨Φ π is sufficient to return a « yes » answer).

TPI/BDD at Work

A BDD of the knowledge base Σ of Example 1 is given in Figure 1. The pis (resp. mis) generated at each node of this BDD are given in Table 1.

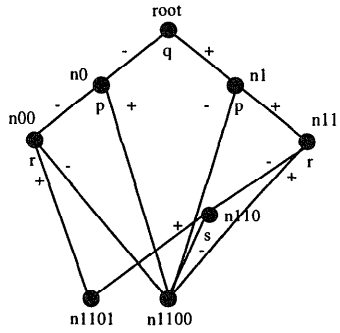

Figure 1. BDD of the running example.

Node	*Pis*	*Mis*
root	{¬p ∨ ¬r, ¬p ∨ s, p ∨ r, r ∨ s, ¬p ∨ q, q ∨ r, p ∨ ¬q, ¬q ∨ ¬r, ¬q ∨ s}	{r ∨ s, p ∨ r}
n0	{¬p, r}	{r}
n1	{p, ¬r, s}	{p, s}
n00	{r}	{r}
n11	{¬r, s}	{¬r, s}
n110	{s}	{s}
n1100	{0}	{0}
n1101	{1}	{1}

Table 1. Pis and mis of the running example.

Table 1 shows that performing deduction checks w.r.t. ⊨Φ allows one to remove at nodes n0, n1 and at the root of the BDD some clauses that are not removed when standard subsumption checks are considered. Particularly,

while PI/BDD generates 20 clauses (as the pis of the restriction formulas of Σ in the BDD) and performs 36 subsumption checks, TPI/BDD generates 11 clauses, only (as the mis of the restriction formulas of Σ w.r.t. Φ in the BDD) and performs 19 deduction checks, only.

Comparing PI and TPI Compilations

We estimate the performances of PI and TPI compilations w.r.t. off-line compiling as the computational resources (space and time) needed by the best algorithms (or, at least, some of the best algorithms) known at that time for achieving such compilations, namely PI/BDD and TPI/BDD.

Our objective is to *compare* both algorithms in a *computer-independent way*; particularly, the performances of such algorithms will not be given in CPU time and Mb consumed (since such scores are dependent of the computer and of the programming language that are used). Instead, more reliable scores are considered and we measure:

- the space complexity of PI/BDD and TPI/BDD as:
  - the number of clauses in the corresponding compilations, and
  - the total number of clauses which have been generated.
- the time complexity of PI/BDD and TPI/BDD as the total number of deduction checks which have been performed.

As far as space complexity is concerned, our evaluation takes into account the number of clauses in the compilations (since it is a lower bound of the space required to compute them), and the total number of clauses that have been generated; this number can be viewed as a upper bound of the space required to compute the compilation when no memory is released (this is the case in our SICStus PROLOG implementation).

The time complexity of PI/BDD and TPI/BDD is weighed as the total number of deduction checks which have been performed, i.e. standard subsumption checks for PI/BDD and deduction w.r.t. Φ for TPI/BDD. Since our objective is to compare both algorithms, we do not include the time spent in computing the BDD. The time spent in generating the clauses could be considered as well but this would not change our analysis in depth: the time complexity of many pi algorithms (including PI/BDD) depends critically on deduction checking (de Kleer 1992). This is also the case for tpi algorithms (in particular, TPI/BDD).

Comparing PI/BDD and TPI/BDD Analytically

Obviously, for every formula Ψ rooted at node n in the BDD of Σ, both the time and the space required to compute the pis (resp. mis) of Ψ mainly depends on the number of pis (resp. mis) of its direct restriction formulas Ψ_0, Ψ_1.

Interestingly, for every formula Ψ and Φ, the number of mis of Ψ w.r.t. Φ *is always lower* (or equal) to the number of pis of Ψ (cf. Corollary to Proposition 1), and it can be exponentially smaller (Marquis 1995).

As a consequence, the space required by TPI/BDD for computing the tpi compilation of Σ w.r.t. Φ *is always smaller* than the space needed by PI/BDD for generating the pi compilation of Σ, independently of the way such resource is estimated. In the following, we shall show that the space savings can be very significant in practice.

Contrastingly, TPI/BDD does not necessarily yield time improvement over PI/BDD in all cases. Indeed, as explained before, some deduction checks that are not required in the pi situation must be done when mis are considered. In particular, when Φ is the empty theory, TPI/BDD may easily perform many useless deduction checks which would be avoided by PI/BDD. However, this is a pathological situation (but it clearly illustrates the influence of Φ in TPI/BDD). Although deduction checks w.r.t. Φ are more expensive than standard subsumption checks, *they succeed more frequently* (each time $\pi' \models \pi$ then $\pi' \models_\Phi \pi$ but the converse does not hold). Thus, many clauses can be given up at each node and *many deduction checks can be avoided* at the nodes occurring « above » (i.e. with lower indexes) in the BDD; actually, the simplification that occurs at each node can easily balance the computational overhead due to more expensive checks.

Preliminary Experimental Results

In order to complete the analysis given above, we performed some preliminary experiments. We implemented both PI/BDD and TPI/BDD in SICStus PROLOG 2.1 and tested them on CNF knowledge bases Σ drawn from diagnosis (adder) and qualitative physics (pipe 2, pipe 3, pipe 4) (Forbus & de Kleer 1993). As a tractable theory Φ, we considered the subset of all the Horn clauses of Σ for the pipe 2, pipe 3, pipe 4 examples and a unit-refutation complete logical consequence of Σ (achieved by FPI_0 compiling of a subset of Σ) for the adder example. For all these theories, checking whether $\pi' \models_\Phi \pi$ can be done in $O(|\Phi| + |\pi'| + |\pi|)$ time.

For each problem, both algorithms use the same BDD (corresponding to a fixed variable ordering for which variables are ranked according to their frequency in Σ: most frequent first).

Problem	*#PI(Σ)*	*#COMP$_\Phi$(Σ)*	*S_{lb}*
pipe 2	638	97	*6.6*
pipe 3	2360	210	*11.2*
pipe 4	6208	399	*15.6*
adder	9700	110	*88.2*

Problem	*#PI/BDD*	*#TPI/BDD*	*S_{ub}*
pipe 2	11489	841	13.7
pipe 3	86943	2837	30.6
pipe 4	1923520	8292	232.0
adder	425545	7241	58.8

Table 2. Space complexity (number of clauses).

Table 2 summarizes the space savings which can be achieved by TPI/BDD w.r.t. PI/BDD. Each line of the table lists the name of the problem, the number #PI(Σ) of

clauses in the pi compilation, the number $\#COMP_{\Phi}(\Sigma)$ of clauses in the tpi compilation (i.e. $\#TPI(\Sigma, \Phi) + \#\Phi$), the savings S_{lb} at the lower bound, i.e. $\#PI(\Sigma) / \#COMP_{\Phi}(\Sigma)$, and the total number `#PI/BDD` (resp. `#TPI/BDD`) of clauses generated by `PI/BDD` (resp. `TPI/BDD`). The last column S_{ub} indicates the global space savings at the upper bound (`#PI/BDD / #TPI/BDD`).

Problem	*#PI/BDD*	*#TPI/BDD*	*S*
pipe 2	1164672	14150	*82.3*
pipe 3	339251046	109377	*3101.7*
pipe 4	4162763406	578500	*7195.8*
adder	906294752	55200	*16418.4*

Table 3. Time complexity (number of deduction checks).

Table 3 gives the time complexity of `PI/BDD` (resp. `TPI/BDD`) estimated as the total number `#PI/BDD` (resp. `#TPI/BDD`) of deduction checks that have been performed. The rightmost column S still gives the relative savings, i.e. the reduction in deduction checks that is achieved (`#PI/BDD / #TPI/BDD`).

Table 2 and Table 3 indicate dramatic improvements in the computational resources needed to produce compilations (between one and over four orders of magnitude). For all the problems we tested, the fact that deduction checks w.r.t. Φ are more expensive than standard subsumption checks (with a $O(|\Phi|)$ time overhead per check in the worst case, only) has been easily balanced by the savings obtained in the number of checks. Experimentally, we observed that the number of deduction checks required grows faster than the square of the final number of tpis (this was noted in (de Kleer 1992) for the pi situation). This may explain why the time savings pointed out in Table 3 are significantly greater than the corresponding space savings given in Table 2.

Actually, an additional Le_Lisp 15.2 implementation of `PI/BDD` with memory release optimization has been necessary to derive these tables; indeed, our SPARC 20/50 station failed in computing the pis of Σ for all the problems, except `pipe 2`, using the PROLOG implementation of `PI/BDD` (this explains why we do not give absolute run time savings in Table 3). Contrastingly, our PROLOG implementation of `TPI/BDD` succeeded in compiling all the problems (though it has not been particularly optimized w.r.t. the `PI/BDD` one). Accordingly, from a practical point of view (i.e. once the programming language and the computer used for the experiments have been fixed), `TPI/BDD` can achieve the compilation of knowledge bases which cannot be compiled using `PI/BDD`.

The analysis clearly shows that, since no amount of algorithmic improvement can avoid the complexity produced by the sheer number of pis, pi compilation (and every compilation technique which requires pi computation) is impractical for many applications ((de Kleer 1992) already pointed it out). Contrastingly, our experiments show that tpi compilations can prove much more economical than pis-based compilation techniques from a computational point of view. Of course, a much more comprehensive experimental evaluation would be needed to see whether such improvements can be reasonably expected in the general case.

Conclusion

The main contribution of this paper is a new algorithm for computing theory prime implicates compilations of propositional knowledge bases Σ. Because it does not require the prime implicates of Σ to be generated, `TPI/BDD` can save a lot of computing.

This work must be extended in several directions. A first direction concerns the empirical validation of this approach; following (Schrag 1996), we plan to check tpi compilations against critically constrained knowledge bases and large real-world problems. A second direction is related to the performances of the various optimizations of `TPI/BDD` mentioned in the paper. Finally, beyond knowledge compilation, investigating the role of tpis in diagnosis and machine learning is another topic for further research.

References

Bryant, R.E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293-318, 1992.

De Kleer, J. An improved incremental algorithm for generating prime implicates. Proc. AAAI'92, 780-785, San Jose (CA), 1992.

Del Val, A. Tractable databases: how to make propositional unit resolution complete through compilation. Proc. KR'94, 551-561, Bonn, 1994.

Forbus, K. and de Kleer, J. *Building Problem Solvers*. MIT Press, Cambridge (MA), 1993.

Hooker, J.N. and Vinay, V. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359-383, 1995.

Jackson, P. and Pais, J. Computing prime implicants. Proc. CADE'90, 543-557, Kaiserslautern, 1990.

Kautz, H. and Selman, B. Forming concepts for fast inference. Proc. AAAI'92, 786-793, San Jose (CA), 1992.

Madre, J.-C. and Coudert, O. A logically complete reasoning maintenance system based on a logical constraint solver. Proc. IJCAI'91, 294-299, Sydney, 1991. See also an extended version entitled « A complete reasoning maintenance system based on typed decision graphs », BULL research report, 1-26, Louveciennes.

Marquis, P. Knowledge compilation using theory prime implicates, Proc. IJCAI'95, 837-843, Montreal, 1995.

Mathieu, Ph. and Delahaye, J.-P. A kind of logical compilation for knowledge bases. *Theoretical Computer Science*, 131:197-218, 1994.

Ngair, T.H. A new algorithm for incremental prime implicate generation. Proc. IJCAI'93, 46-51, Chambery, 1993.

Reiter, R. and de Kleer, J. Foundations of assumption-based truth maintenance systems: preliminary report. Proc. AAAI'87, 183-188, Seattle (WA), 1987.

Schrag, R. Compilation for critically constrained knowledge bases. Proc. AAAI'96 (this issue), Portland (OR), 1996.

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):304-310, 1970.