Compilation for Critically Constrained Knowledge Bases

Robert Schrag
Department of Computer Sciences and Applied Research Laboratories
University of Texas at Austin
Internet: schrag@cs.utexas.edu
World-wide Web: http://www.cs.utexas.edu/users/schrag/

Abstract

We show that many "critically constrained" Random 3SAT knowledge bases (KBs) can be compiled into disjunctive normal form easily by using a variant of the "Davis-Putnam" proof procedure. From these compiled KBs we can answer all queries about entailment of conjunctive normal formulas, also easily — compared to a "brute-force" approach to approximate knowledge compilation into unit clauses for the same KBs. We exploit this fact to develop an aggressive hybrid approach which attempts to compile a KB exactly until a given resource limit is reached, then falls back to approximate compilation into unit clauses. The resulting approach handles all of the critically constrained Random 3SAT KBs with average savings of an order of magnitude over the brute-force approach.

Introduction

Consider the task of reasoning from a propositional knowledge base (KB) $\mathcal{F}$ which is expressed as a conjunctive normal formula (CNF). We are given other, query CNFs $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_N$ and asked, for each $\mathcal{Q}_i$, "Does $\mathcal{F}$ entail $\mathcal{Q}_i$?" In general, this reasoning task is co-NP-complete. Selman and Kautz (1991, 1996) introduce propositional *knowledge compilation* (KC) — transforming $\mathcal{F}$ into a "compiled" form $\mathcal{F}_C$ so that such propositional reasoning can be done tractably. KC as they propose it is *approximate* — sound but not complete. For queries which are not covered by $\mathcal{F}_C$, it is necessary to fall back to general, co-NP-complete reasoning to get the desired answer. KC usually has been viewed as an "off-line" process, occurring in the background, with ignorable cost. Nonetheless Kautz and Selman (1994) show that KC can pay for itself when the set of queries to be answered after compilation is large enough.

We introduce two new, related KC approaches which we compare to a straightforward "brute-force" approach to unit clause approximate KC using "critically constrained" KBs from Random 3SAT as a benchmark. First, we introduce an exact KC approach based on the very effective "Davis-Putnam" proof procedure for determining propositional satisfiability, or DP (Davis & Putnam 1960, Davis, Logemann, & Loveland 1962). We modify DP to collect an irredundant prime implicant cover — a disjunctive normal formula (DNF) equivalent to the input KB, which serves as $\mathcal{F}_C$. On the many KBs which are easy, this approach amortizes its compilation overhead about 50 times sooner and increases average query speed 2–3 orders of magnitude more than does the brute-force approach. Second, we exploit this fact to develop an aggressive hybrid approach which attempts to compile a KB exactly until a given resource limit is reached, then falls back to approximate compilation as necessary. The resulting approach handles all of the critically constrained Random 3SAT KBs with average savings of an order of magnitude on both time performance metrics mentioned above.

In Random 3SAT, highly "under-constrained" KBs have many solutions which typically are easy to find; KC for such KBs may be unnecessary — or even counter-productive, if a KC approach must enumerate all of their many observed prime implicates (Schrag & Crawford 1996). KBs which are "over-constrained" (*i.e.*, inconsistent) are uninteresting for most practical purposes — they entail any query. Thus, we may argue that KC is most useful when KBs are "critically constrained" — just "on the edge" of unsatisfiability — so that including some small number of additional randomly selected clauses would result with high probability in a CNF that was unsatisfiable. Consistent critically constrained KBs which occur at roughly 4.3 clauses/variable turn out to be just those for which KC is most effective and appropriate. That is, answering queries with the uncompiled KB $\mathcal{F}$ tends to be hard, but answering queries with the compiled KB $\mathcal{F}_C$ tends to average out as easy, even when (as in approximate KC) some queries not covered by $\mathcal{F}_C$ must be answered using $\mathcal{F}$. Critically constrained Random 3SAT KBs, on average, have relatively short prime implicants (Schrag & Crawford 1996) and relatively few models (Selman, Mitchell, & Levesque 1996). The former property contributes to the success of unit clause approximate KC;

the latter property contributes to the success of our DP-based exact KC.

Definitions

We deal with propositional logic throughout. A *variable*, (*e.g.*, p), ranges over the values "true" and "false". A *literal* is a variable or its negation (*e.g.*, p, ¬p). A *clause* is a disjunction of literals (*e.g.*, (p ∨ ¬q ∨ r)). When convenient, we treat a clause as the set of its literals, or *vice versa*. A clause which contains both a literal and its negation (*e.g.*, (p ∨ ¬p ∨ q)) is *trivial*. (It is a tautology.) A *Horn clause* contains at most one positive literal. A *conjunctive normal formula* (CNF) $\mathcal{F}$ is a conjunction of clauses (*e.g.*, ((p ∨ ¬q ∨ r) ∧ (p ∨ s))). When convenient, we treat a CNF as the set of its clauses, or *vice versa*. An *implicate* of a formula $\mathcal{F}$ is a non-trivial clause C over the variables of $\mathcal{F}$ such that $\mathcal{F}$ implies C ($\mathcal{F} \Rightarrow C$). A set S *subsumes* a set S' if $S \subseteq S'$. *E.g.*, the clause (a ∨ b) subsumes the clause (a ∨ b ∨ c). A *prime implicate* of a formula $\mathcal{F}$ is an implicate C of $\mathcal{F}$ such that C is subsumed by (contains) no other implicate C' of $\mathcal{F}$.

A *term* is a conjunction of literals (*e.g.*, (p ∧ ¬q ∧ r)). When convenient, we treat a term as the set of its literals, or *vice versa*. A term which contains both a literal and its negation (*e.g.*, (p ∧ ¬p ∧ q)) is *contradictory*. A *disjunctive normal formula* (DNF) $\mathcal{F}$ is a disjunction of terms (*e.g.*, ((p ∧ ¬q ∧ r) ∨ (p ∧ s))). When convenient, we treat a DNF as the set of its terms, or *vice versa*. An *implicant* of a formula $\mathcal{F}$ is a non-contradictory term T over the variables of $\mathcal{F}$ such that T implies $\mathcal{F}$ ($T \Rightarrow \mathcal{F}$). A *prime implicant* of a formula $\mathcal{F}$ is an implicant T of $\mathcal{F}$ such that T is subsumed by (contains) no other implicant T' of $\mathcal{F}$.

A (prime) implicate/implicant *cover* of a formula $\mathcal{F}$ is a subset Φ of the set of all (prime) implicates/implicants of $\mathcal{F}$ such that Φ is logically equivalent to $\mathcal{F}$. A (prime) implicate/implicant cover Φ of $\mathcal{F}$ is *irredundant* if it contains no element ϕ such that $\Phi \backslash \{\phi\}$ is logically equivalent to $\mathcal{F}$. A *model* of a propositional formula $\mathcal{F}$ is an implicant of $\mathcal{F}$ which mentions every variable that is mentioned in $\mathcal{F}$.

The *resolution* operation combines two clauses A and B mentioning complementary literals l and l', respectively (*e.g.*, A = (p ∨ q), B = (p ∨ ¬q)), to produce a single *resolvent* clause which mentions all literals in either clause except the complementary ones ($[A - l] \cup [B - l'] =$ (p)). In addition, we require that $A \cup B$ include no pair of complementary literals besides $\{l, l'\}$; this excludes trivial clauses as inputs and outputs for resolution.

Instances of *Random 3SAT* are parameterized by two values: n, the number of variables; and m, the number of clauses which are to be generated. Clauses are generated by selecting a set of 3 variables randomly from among the n available variables with uniform probability $1/\binom{n}{3}$ and negating each with uniform probability 1/2.

The KC Approach of Selman and Kautz

Selman and Kautz (1991, 1996) compile a KB into a pair $(\mathcal{F}_{\text{lb}}, \mathcal{F}_{\text{ub}}) = \mathcal{F}_{\text{C}}$, such that both $\mathcal{F}_{\text{lb}}$ and $\mathcal{F}_{\text{ub}}$ are sets of Horn clauses (Horn CNFs), and

$$\mathcal{F}_{\text{lb}} \Rightarrow \mathcal{F} \Rightarrow \mathcal{F}_{\text{ub}}.$$

$\mathcal{F}_{\text{lb}}$ is a (greatest) *lower bound* of $\mathcal{F}$. $\mathcal{F}_{\text{ub}}$ is a (least) *upper bound* of $\mathcal{F}$. In general, neither bound is equivalent to the KB itself, which is why this compilation approach is called "approximate." Propositional reasoning with these Horn clause CNFs can be done in linear time (Dowling & Gallier 1984).

To answer a query about a clause C in approximate KC, we use the following procedure. (A CNF query follows from a KB $\mathcal{F}$ iff every one of its clauses follows from $\mathcal{F}$.)

KC-QUERY($C, \mathcal{F}_{\text{lb}}, \mathcal{F}_{\text{ub}}, \mathcal{F}$)
1 **if** $\mathcal{F}_{\text{ub}} \Rightarrow C$ **then return**(true);
2 **if** $\mathcal{F}_{\text{lb}} \not\Rightarrow C$ **then return**(false);
3 **if** (DP($\mathcal{F} \cup \neg C$))) $\neq$ false) **then return**(false);
4 **return**(true);

Kautz and Selman (1994) specialize this approach so that $\mathcal{F}_{\text{lb}}$ and $\mathcal{F}_{\text{ub}}$ are sets of unit clauses — making compilation feasible for many KBs for which full Horn KC is not. Under this specialization, $\mathcal{F}_{\text{ub}}$ is the set of unit prime implicates of $\mathcal{F}$, and $\mathcal{F}_{\text{lb}}$ is equivalent to a single prime implicant.

Alternative KC Formulations

Many different forms are possible for $\mathcal{F}_{\text{C}}$, in either the approximate or the exact KC styles. We note that a cover of (prime) implicants supports exact query answering, using a linear-time procedure which we describe below. Del Val (1994) shows that unit resolution is complete for some subsets of the set of all prime implicates, but coverage alone is not a sufficient condition to make unit resolution complete for such a subset. Bounds for approximate KC must merely satisfy the scheme

$$\mathcal{F}_{\text{lb}} \Rightarrow \mathcal{F} \Rightarrow \mathcal{F}_{\text{ub}},$$

where $\mathcal{F}_{\text{lb}}$ is some lower bound on $\mathcal{F}$ and $\mathcal{F}_{\text{ub}}$ is some upper bound. Non-covers of (prime) implicants will serve as lower bounds; non-covers of (prime) implicates will serve as upper bounds.

DP-based Exact KC

Unlike the problem of generating prime implicates or a cover thereof for a CNF, the problem of generating prime implicants or a cover for a CNF has not received much attention. The former problem — where the input and the output have the same normal form — is important for circuit design, where it is known as "2-level logic minimization". (For a recent survey, see (Coudert 1994).) Algorithms for solving the latter problem which interests us seem to be directed primarily toward tasks in automated reasoning (Slagle,

Chang, & Lee 1970, Rymon 1994, Jackson & Pais 1990, Sochor 1991, Madre & Coudert 1991, Castell & Cayrol 1996). Among the cited algorithms, only that of Castell and Cayrol is at all feasible for generating the prime implicants for many of our critically constrained Random 3SAT KBs. All of these algorithms generally require computing the entire set of prime implicants before one can identify a covering subset. It seems worth trying to generate the cover directly, since this can be exponentially smaller than the full set of prime implicants.

We attack the CNF prime implicant generation problem with a flexible core which we call DP*. In the pseudo-code that follows, CNFs are represented as sets of clauses; clauses and implicants are represented as sets of literals. "\" is the set difference operator. The brackets "[", "]" enclose a statement block.

```
DP*(CNF, σ)
1   UNIT-PROPAGATE(CNF, σ);
2   if ∅ ∈ CNF then return(false);
3       else if CNF = ∅ then
4           [PROCESS-IMPLICANT(σ);
5           return;]
6       else
7           [α ← CHOOSE-BEST-LITERAL(CNF);
8           DP*(CNF ∪ {{α}}, σ ∪ {α});
9           DP*(CNF ∪ {{¬α}}, σ ∪ {¬α});]
10  return(false);
```

DP* is specialized to compute one implicant or an irredundant prime implicant cover by providing different definitions of the procedure PROCESS-IMPLICANT. When so specialized, DP* serves as the core for ("standard") DP and DPPI (for "DP-PRIME-IMPLICANTS"), respectively. First, we discuss the operation of standard DP.

```
DP(CNF)
1   return(DP*(CNF, ∅));

PROCESS-IMPLICANT_DP(σ);
1   return-to(DP, σ);
```

In recursive calls (by value) to DP*, a candidate implicant σ is extended until its status as an implicant or a non-implicant can be definitely determined. CNF and σ are modified in calls (by name) to UNIT-PROPAGATE. In the implementation we use, CHOOSE-BEST-LITERAL always returns the positive literal for the variable which occurs in the greatest number of clauses of the shortest length in CNF. Note the non-local exit in Line 1 of PROCESS-IMPLICANT$_{DP}$: DP returns "false" or a set of literals whose conjunction is an implicant of its input argument. (If a variable is not included in this set, then a literal for it in either sense (positive or negative) may be included to form a larger, subsumed implicant or a model.)

```
UNIT-PROPAGATE(CNF, σ)
1   while ∃ω ∈ CNF: ω = {λ}
2       [σ ← σ ∪ {λ};
3       for ψ ∈ CNF \ω
4           [if λ ∈ ψ then CNF ← CNF \{ψ};
5               else if ¬λ ∈ ψ then
6                   [CNF ← CNF ∪ {ψ \ {¬λ}};
7                   CNF ← CNF \{ψ};]]]
```

UNIT-PROPAGATE adds the single literal λ in a unit clause ω to the literal set σ (Line 2). Then it simplifies CNF as follows. For each clause ψ besides ω in CNF, it removes ψ if ω subsumes ψ (Line 4), and it performs resolution on ψ and ω if possible, adding the resolvent and removing the now-subsumed ψ (Lines 5 through 7).

```
DPPI(CNF)
1   PRIMES ← ∅;
2   DP*(CNF, ∅);
3   RETURN(PRIMES);

PROCESS-IMPLICANT_DPPI(σ);
1   PRIMES ← PRIMES ∪ {ONE-PRIME(σ)};
2   unless REMAINING-RESOURCES()
3       [DPPI-INCOMPLETE ← true;
4       return-to(DPPI);]
```

DPPI is like DP, except that it does not terminate after finding one implicant of the input KB; it continues to explore the *entire* search tree, recording a prime (minimal) version of each implicant found along the way in the global variable PRIMES. The procedure REMAINING-RESOURCES detects the condition that allocated space or time resources have been exhausted, whereupon PROCESS-IMPLICANT$_{DPPI}$ sets a flag to indicate that compilation must become approximate (Line 3) and interrupts DPPI execution by exiting non-locally (Line 4). At termination, if uninterrupted, PRIMES contains an irredundant prime implicant cover of the input KB; it includes a prime implicant to cover (subsume) any model of the KB, and no other terms. This cover serves as the compiled form $\mathcal{F}_C$ for the CNF KB $\mathcal{F}$ in exact KC.

```
ONE-PRIME(σ, CNF)
1   PRIME? ← true;
2   for λ ∈ σ
3       [HIT-CLAUSES ← {ψ ∈ CNF : λ ∈ ψ};
4       if ¬(∃ψ ∈ HIT-CLAUSES : (ψ ∩ σ) = {λ})
5           then
6               [PRIME? ← false;
7               ONE-PRIME(σ \ {λ}, CNF);]]
8   if PRIME?
9       then return-to(PROCESS-IMPLICANT, σ);
```

Computing a prime version of each discovered implicant has the advantages that the implicants comprising $\mathcal{F}_C$ can be smaller and that fewer implicants may be needed overall, since one prime implicant may subsume multiple discovered implicants. The greedy algorithm

ONE-PRIME will visit at most $O(|\sigma|^2)$ candidate implicants before reaching its non-local exit in Line 9.

Our procedure for answering CNF KB entailment queries using a set of (prime) implicants makes use of the fact that every implicate C of a KB $\mathcal{F}$ must have a non-null intersection with every implicant of $\mathcal{F}$. (This follows immediately from the definitions for implicate and implicant.) This can be checked in time linear in the cardinality of $\mathcal{F}_C$ (or $\mathcal{F}_{\text{lb}}$, when we use the procedure DPPI-KC-QUERY in approximate KC).

```
DPPI-KC-QUERY(C, F_C)
1   for σ ∈ F_C
2       [if (σ ∩ C = ∅) then return(false)];
```

Brute-force Approximate KC

As our baseline method for the generation of upper bounds for approximate KC, we use the following, "brute-force" procedure.

```
BRUTE-FORCE-UB(F)
1   Λ ← LITERALS-OF(F);
2   F_ub ← ∅;
3   for λ ∈ Λ
4       [unless ({¬λ} ∈ F_ub)
5           [if (DP(F ∪ {¬λ}) = false)
6               then F_ub ← F_ub ∪ {λ};]]
```

LITERALS-OF returns a set of the literals in its input CNF. For each represented literal, a refutation check (Line 5) determines whether this literal appears in a unit implicate, if (Line 4) its negation does not already appear in the upper bound being collected (Line 6).

In the following, we assume for convenience that the KBs input to KC procedures are consistent.

```
BRUTE-FORCE-KC(F)
1   σ ← ONE-PRIME(DP(F));
2   F_lb ← {σ};
3   BRUTE-FORCE-UB(F);
```

BRUTE-FORCE-KC compiles an input KB into a lower bound $\mathcal{F}_{\text{lb}}$ including a single prime implicant generated by DP and an upper bound $\mathcal{F}_{\text{ub}}$ generated using the brute-force approach.

Figure 1 shows execution time histograms of DPPI (given unlimited resources) and BRUTE-FORCE-KC for critically constrained Random 3SAT KBs of 100 variables. Execution times, in milliseconds, are rounded to their nearest base 2 log. The histogram modes for DPPI and BRUTE-FORCE-KC fall in the buckets for 2^{10} milliseconds and 2^{16} milliseconds, respectively — a difference of about two (base 10) orders of magnitude. DPPI does compile, exactly, most of the KBs relatively easily. On the other hand, we see here that DPPI also can take orders of magnitude *more* time than BRUTE-FORCE-KC. DPPI's run-time is correlated with the number of prime implicants it finds, so the size of a compiled KB $\mathcal{F}_{\text{lb}}$ can be huge. If we can avoid this situation, we will be able to exploit fast, exact KC for critically constrained KBs opportunistically, falling back to approximate KC where necessary. In the next section, we describe a hybrid approach which combines resource-limited DPPI processing and a more aggressive approach to unit clause approximate KC than is embodied in BRUTE-FORCE-KC. We call this approach "bootstrapped" KC.

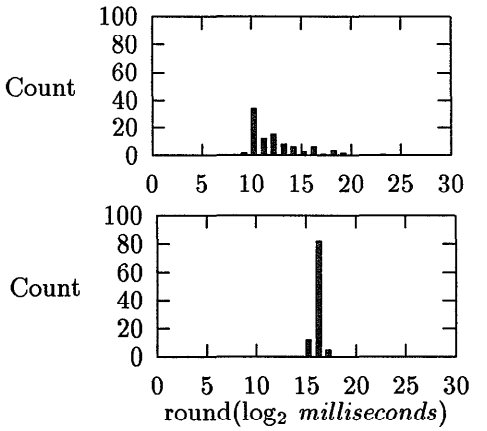

Figure 1: Histograms of compilation times for 100 consistent critically constrained Random 3SAT KBs of 100 variables, under exact KC procedure DPPI (top) and approximate KC procedure BRUTE-FORCE-KC (bottom).

Bootstrapped Approximate KC

The key idea of bootstrapped KC is to use KC-style querying to accelerate the compilation of the KC upper bound, appealing to whatever partial bounds have so far been accumulated, in a "bootstrapping" fashion.

```
BOOTSTRAPPED-KC(F)
1   F_lb ← DPPI(F);
2   if DPPI-INCOMPLETE
3       then BOOTSTRAPPED-UB(F);
```

We fall back to bootstrapped approximate KC when DPPI reaches its established resource limits. Then we call BOOTSTRAPPED-UB.

```
BOOTSTRAPPED-UB(F, F_lb)
1   Λ ← LITERALS-OF(F);
2   F_ub ← ∅;
3   for λ ∈ Λ
4       [unless ((¬λ} ∈ F_ub) ∨
5                ¬ DPPI-KC-QUERY({λ}, F_lb))
6           [if DP(F ∪ {¬λ}) then
7               [F_ub ← F_ub ∪ {λ};
8               σ ← ∅;
9               UNIT-PROPAGATE(F, σ);
10              for π ∈ σ [F_ub ← F_ub ∪ {π};]]]]
```

We use the lower bound gained from the truncated run of DPPI to circumvent (Line 5) the refutation

check for candidate implicates (Line 6) whenever possible. This means that the DPPI work is not wasted; the more prime implicants we have in $\mathcal{F}_{\text{lb}}$, the more candidate implicates we may be able to rule out without the refutation check.[1] When there are significant numbers of unit prime implicates, the ones that have been generated so far in the KC upper bound are another great source of efficiency. Knowing that a unit clause is implied, we use it to simplify the input KB using UNIT-PROPAGATE (Line 9), adding any further unit prime implicates this simplification uncovers to $\mathcal{F}_{\text{ub}}$ (Line 10). The refutation checks become easier as the simplified KB becomes smaller. To exploit this savings in fallback query processing, we also use the ultimate simplified KB in our bootstrapped version of the KC-QUERY procedure.

BOOTSTRAPPED-KC-QUERY($C, \mathcal{F}_{\text{lb}}, \mathcal{F}_{\text{ub}}, \mathcal{F}$)
1 **if** $\neg$DPPI-KC-QUERY($C, \mathcal{F}_{\text{lb}}$) **then return**(false);
2 **if** $\neg$DPPI-INCOMPLETE **then return**(true);
3 **if** $\mathcal{F}_{\text{ub}} \Rightarrow C$ **then return**(true);
4 **if** (DP($\mathcal{F} \cup \neg C$) $\neq$ false) **then return**(false);
5 **return**(true);

A Comparative Evaluation

We use the following quantities to describe and compare the performance of different KC approaches. These are most applicable when a common satisfiability algorithm is the core procedure for each of the subject operations: upper and lower bounds computation; uncompiled query processing; and fallback query processing for approximate KC, when the compiled bounds are not sufficient. In each of these operations, we use a Lisp version of the "tableau" implementation of DP (Crawford & Auton 1993), which we have modified slightly to produce the variants we have described. We performed all our experiments on a Sun SPARC-5 workstation using Allegro Common Lisp, measuring execution times with the function `get-internal-run-time`.

Q_U — time/query, uncompiled
Q_C — time/query, compiled
$\alpha = Q_C/Q_U$ — query time ratio
C — compilation time
$\beta = C/(Q_U - Q_C)$ — "break-even" point

The primary bases of comparison are the values α and β, which tell us, respectively, how much query time improvement we get from KC and on average after how many queries that improvement will allow us to amortize the cost of compilation. When comparing KC approaches, lower values of both α and β are better.

We take our critically constrained KBs at the same points (n variables and m clauses) in Random 3SAT as Kautz and Selman (1994), as shown in the following table. Also shown are the average times in seconds to evaluate 100 queries/KB without compilation.

n	75	100	150	200
m	322	430	645	860
Q_U	0.131	0.268	1.77	12.4

Our queries are ternary clauses over the same n variables, generated just as for Random 3SAT. We evaluated 100 KBs/data point and 100 queries/KB, so 10,000 queries/data point overall. We compare the described KC approaches in the table below. α is a dimensionless ratio; β is in units of queries; $|\mathcal{F}_C|$ is in literals. "Bootstrapped(x)" means that we used a time limit of x seconds for the DPPI phase and then fell back to bootstrapped KC for the approximate cases. "DPPI(x)" means that we ran DPPI with a time limit of x seconds and then based the statistics *only* on the KBs that could be compiled exactly given that limit (68 KBs for $n = 75$, 69 KBs for $n = 100$).

KC approach	n	α	β	$\lvert\mathcal{F}_C\rvert$
Brute-force	75	2.44e-1	1.80e2	1.10e2
Bootstrapped(3)	75	1.57e-2	1.05e1	4.37e3
DPPI(0.720)	75	3.88e-4	3.42e0	1.08e3
Brute-force	100	2.63e-1	3.27e2	1.46e2
Bootstrapped(5)	100	3.26e-2	1.90e1	9.73e3
DPPI(5.79)	100	3.70e-4	6.56e0	4.32e3
Brute-force	150	2.47e-1	5.85e2	2.17e2
Bootstrapped(50)	150	2.24e-2	2.90e1	3.71e4
Bootstrapped(300)	200	1.42e-2	3.41e1	1.24e5

As might be anticipated from the histograms in Figure 1, DPPI gives orders of magnitude improvement in α and β over brute-force KC when it is allowed to "skim off" the KBs which are easy for it. In the hybrid approach of bootstrapped KC, improvements in α and β relative to brute-force KC are dependent on the resources allocated to DPPI. With some minor tuning favoring improvements in β, we get improvement factors of an order of magnitude consistently across three different KB sizes. (We did not compile the 200-variable KBs using the brute-force approach.)

n	75	100	150
β improvement	17.2	17.2	20.2
α improvement	17.9	8.07	11.0

The compilation procedure used by Kautz and Selman (1994) is similar to our BRUTE-FORCE-KC.[2] Compared to the values that can be derived from their published data, for $n = 75$, BOOTSTRAPPED-KC's α and β are 12.4 and 171 times better, and DPPI's α and β on the skimmed-off, easy queries are 500 and 526 times better, respectively. For $n = 200$,

[1] Schrag and Crawford (1996) describe a pre-cursor to bootstrapped small prime implicate generation in which the lower bound is a single model. This idea is credited to Andrew Parkes.

[2] Personal communication.

BOOTSTRAPPED-KC's α and β are 13.3 times better and 2.40 times worse, respectively. Kautz and Selman did not report precise compilation times for the medium-sized KBs.

Conclusion

That DPPI alone works well on many of these Random 3SAT KBs depends on their being critically constrained. The problem of prime implicant generation is co-NP-hard; the total number of prime implicants can be $\Omega(3^n/\sqrt{n})$ (Chandra & Markowsky 1978); an irredundant cover can include as many as 2^n prime implicants. A full search by our DPPI algorithm must visit at least as many leaves as are contained in the union of irredundant covers of minimum cardinality for both prime implicants and prime implicates (Wegener 1988).

Naturally occurring KBs may include some variables which are relatively unconstrained; compilation with respect to these variables may be unproductive. They also may include some sets of variables which are critically constrained — participating in short prime implicates which are relatively difficult to compute. If we can recognize candidate subsets of such variables from a problem's surface description, then we can try a bootstrapped KC approach on that subset. For example, the "mapworld" KB evaluated by Kautz and Selman (1994) includes a "maze" which is a highly constrained subgraph said to admit only 7 solutions. Using DPPI on this subgraph would produce a small prime implicant cover. Recognizing criticality generally without performing work much like exact KC may be impossible. See (Dechter & Pearl 1992) for a discussion of some related problems.

We show that compiling critically constrained KBs aggressively can make fast, exact KC possible and that the fallback approach of bootstrapped approximate unit clause KC also pays for itself after an average few tens of queries. We expect such a favorable amortization rate to be even more important for KC when dealing with KBs which admit transactions modifying their contents, in dynamic knowledge recompilation.

Acknowledgements

We thank Jimi Crawford and Matt Ginsberg for giving us the source code for their Lisp version of tableau. We thank Ron Rymon for his SE-tree software, which we used in the implementation of DPPI. We thank Roberto Bayardo, Pierre Marquis, Dan Miranker, Greg Provan, Bart Selman, and anonymous reviewers for their comments on earlier drafts.

References

Castell, T., and Cayrol, M. 1996. Une nouvelle méthode de calcul des impliquants et des impliqués premiers. In *IIème Conference sur la Resolution Pratique de Problèmes NP-complets.*

Chandra, A., and Markowsky, G. 1978. On the number of prime implicants. *Discrete Math.* 24:7–11.

Coudert, O. 1994. Two-level logic minimization: An overview. *Integration* 17:97–140.

Crawford, J., and Auton, L. 1993. Experimental results on the crossover point in satisfiability problems. In *Proc. AAAI-93,* 21–27.

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *JACM* 7:201–215.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. *CACM* 5:394–397.

Dechter, R., and Pearl, J. 1992. Structure identification in relational data. *Artificial Intelligence* 58:237–270.

del Val, A. 1994. Tractable databases: How to make propositional unit resolution complete through compilation. In *KR'94,* 551–561.

Dowling, W., and Gallier, J. 1984. Linear time algorithms for testing the satisfiability of propositional Horn formula. *J. Logic Programming* 3:267–284.

Jackson, P., and Pais, J. 1990. Computing prime implicants. In *CADE-10,* number 449 in Lecture Notes in AI, 543–557. Springer-Verlag.

Kautz, H., and Selman, B. 1994. An empirical evaluation of knowledge compilation by theory approximation. In *Proc. AAAI-94,* 155–161.

Madre, J., and Coudert, O. 1991. A logically complete reasoning maintenance system based on a logical constraint solver. In *Proc. IJCAI-91,* 294–299.

Rymon, R. 1994. An SE-tree-based prime implicant algorithm. *Annals of Math. and AI* 11.

Schrag, R., and Crawford, J. 1996. Implicates and prime implicates in Random 3SAT. *Artificial Intelligence* 81:199–222.

Selman, B., and Kautz, H. 1991. Knowledge compilation using Horn approximations. In *Proc. AAAI-91,* 904–909.

Selman, B., and Kautz, H. 1996. Knowledge compilation and theory approximation. *JACM.* To appear.

Selman, B.; Mitchell, D.; and Levesque, H. 1996. Generating hard satisfiability problems. *Artificial Intelligence* 81:17–29.

Slagle, J.; Chang, C.-L.; and Lee, R. 1970. A new algorithm for generating prime implicants. *IEEE Transactions on Computers* C-19(4):304–310.

Sochor, R. 1991. Optimizing the clausal normal form transformation. *J. Automated Reasoning* 7:325–336.

Wegener, I. 1988. On the complexity of branching programs and decision trees for clique functions. *JACM* 35(2):461–471.