Linear Time Unit Resolution for Propositional Formulas—in Prolog, yet

Allen Van Gelder
Baskin Center for Computer Engineering and Information Sciences
University of California, Santa Cruz 95064
UCSC-CRL-95-32
avg@cs.ucsc.edu

April 19, 1995

Abstract

A procedure to analyze a propositional formula in clause form by unit resolution is described and illustrated with a Prolog implementation. It runs in worst-case time that is linear in the length of the formula. The main idea has been independently rediscovered by several implementers. Apparently, its first journal appearance was a sketch by Dalal and Etherington in 1992. However, there also had arisen a folkloric belief that unit resolution requires quadratic time.

This report shows that the implementation sketched for imperative languages, such as C, consumes quadratic time if translated to Prolog. This degradation occurs even if clause indexing permits the retrieval of an asserted clause in constant time. It is rather due to the way Prolog handles assertions involving data structures, such as lists. A modified Prolog implementation that restores linear time is described.

The time remains linear even if the procedure is run "on-line", meaning that new clauses appear in the input as processing proceeds. This property is useful in applications that have several mechanisms for deriving new clauses. The technique may have application in other problems that can be described as inductive closures on finite domains. *Ad hoc* solutions to such problems are error-prone and often inefficient.

Keywords: Propositional logic, satisfiability, validity, boolean formula, unit resolution, Prolog internals, algorithms, inductive closure.

1 Introduction

The decision problem of Boolean, or propositional, satisfiability is the "original" NP-hard problem. We assume the reader is generally familiar with it, and give definitions only when needed for clarity. We shall consider exclusively propositional formulas in *conjunctive normal form* (CNF), also called *clause form*. Each clause is a disjunction of literals, and clauses are joined conjunctively. A closely related problem is to determine *validity* of a formula in *disjunctive* normal form. As *language recognition* problems, satisfiability is in NP, while validity is in co-NP. However, as *decision* problems, they are essentially equivalent.

In the search for efficiency, certain special classes have been found to be decidable in polynomial time. The best known of these are:

- *2-CNF*, in which each clause has at most two literals, also called *Krom* formulas,
- *Horn clause* formulas, in which each clause has at most one positive literal.

On the other hand, certain procedures are guaranteed to terminate in polynomial time on all propositional formulas, but may do so without reaching a decision; such procedures are called *incomplete*. Unit resolution fits into both worlds, in a way, because it is known to terminate in polynomial time on all formulas, and is guaranteed to reach a decision on *Horn clause* formulas.

Recently, incomplete procedures have found application as subroutines within complete procedures. By drawing what inferences they can quickly, they assist the main procedure to avoid some unnecessary guessing and backtracking. In this application:

- It is important that such repeatedly used subroutines be as fast as possible; more than linear time is rarely practical.
- It is essential that the procedures deliver whatever information is derivable, even when a firm decision is not reached.
- Efficient on-line operation is desirable, as other parts of the overall program will discover or hypothesize new clauses as the processing proceeds.

As early examples, Gallo and Urbani embedded Horn SAT as a subroutine in a full-blown satisfiability checker [GU89], and Larrabee embedded 2-SAT [Lar92].

There has been a steady trickle of papers over the years that address how efficiently two problems concerning propositional Horn formulas can be solved:

- **Horn SAT:** Is a given Horn formula satisfiable?
- **Horn Renameability:** Can a given formula be "renamed" into a Horn formula by inverting the sense of some of its variables?

A brief historical perspective follows.

Horn SAT is easily seen to be equivalent to several problems in formal language theory and database theory for which linear time algorithms were known in the 1970s. A partial list is: identification of nullable (or eraseable) symbols of a context free grammar [Har78], emptiness of a context free grammar, and implication of functional dependencies [BB79]. This correspondence was observed by Dowling and Gallier [DG84], who adopted a bottom-up algorithm from Harrison [Har78], who in turn credited A. Yehudai. Dowling and Gallier also introduced a top-down linear-time algorithm. A correction to the latter was offered by Scutella [Scu90].

Horn Renameability has been resolved since 1980 by Aspvall, in a much overlooked (but accessible) paper [Asp80]. However, *he* overlooked (and rediscovered) Lewis' elegant original solution [Lew78], which required quadratic time in the presence of long clauses.[1] Aspvall's extension was to convert the formula to 3-CNF first, then apply Lewis' transformation (which is linear-time on 3-CNF), and finally apply the recently discovered linear-time 2-SAT algorithm [APT79].

I hope the numerous later researchers who have published on Horn Renameability will forgive me for not citing them, and will forgive Aspvall for not giving his paper a better title.

More recently, Ausiello and Italiano showed how to make Dowling and Gallier's bottom-up algorithm run on-line without a loss of performance [AI91]. They also described another version, which also delivers additional information, and requires time kn, for k propositional variables and n atoms in the formula. The paper assumes k is known in advance for both versions; this assumption can be relaxed, as shown in this paper. The on-line property is useful in applications that have several mechanisms for deriving new clauses.

Dalal and Etherington sketched a linear-time algorithm for unit resolution [DE92, Sect. 4], which appears to have been independently rediscovered by several implementers. This paper describes that algorithm in detail for an imperative language, such as C, and then shows what modifications are needed to achieve linear time performance in Prolog.

The basic algorithm may be regarded as a generalization of widely known bottom-up algorithm mentioned [Har78, BB79, DG84, AI91]. Nevertheless, there are some aspects of it that might make a detailed presentation worthwhile:

1. It dispels the folkloric belief that unit resolution requires quadratic time. Dowling and Gallier mention that unit resolution takes quadratic time, and Gallo and Urbani make quite a point of it, including empirical evidence [GU89]. These remarks were not incorrect, being based on a particular implementation of unit resolution, but they gave the impression that improvement was not possible.

2. Unit resolution is a complete decision method for Horn clauses. Although a linear method for Horn clauses is already known, the present technique may be applicable to other special cases.

3. It has been implemented in Prolog to run in linear time, using commonly available Prolog features, such as dynamic clause indexing and local cut. The code is available electronically. This contrasts with most of the cited papers, which reported no implementation and used pseudo-code.

4. The implementation is straightforward; the inference engine worked correctly on the third try, after correcting a few "typos". (The I/O interface, actually about 70% of the code, took a few more passes.)

5. It serves as a model for re-implementation in a more efficient language, like C. In my experience it is usually much faster to develop a working high-level prototype, and re-implement into C if results warrant it, than to start from scratch in C. Of course, this is only successful if you avoid "clever" uses of the high level language that would be difficult to convert.

6. Unit resolution inference is an example of a class of problems that can be characterized as *inductive closure* with respect to a set of operators on a finite domain. Such problems arise in many applications, such as building parsing tables, determining deadlock, mode inference, etc. These problems are notoriously treacherous in Prolog (or any language) when approached by *ad hoc* methods. The idea of this algorithm may offer a systematic approach to such problems.

[1] Lewis showed that the problem could be transformed into a 2-SAT question in quadratic time; since the linear-time 2-SAT algorithm was not known when he published, he was not motivated to improve the transformation time.

7. Besides a decision (or lack thereof) on satisfiability, the data structures can deliver efficiently all inferences drawn during the procedure, in the form of inferred literals, identification of clauses subsumed by inferred literals, and strengthened clauses (whose original version contained complements of inferred literals).

8. The time remains linear even if the procedure is run "on-line".

9. Testing for Horn renameability becomes less important. The motivation is usually to see if the formula could be converted to a form on which a specialized Horn SAT algorithm could run. But unit resolution now can be run in linear time on the original formula. (We know of no implementations that perform Horn renaming.) Also, unit resolution may find a refutation when the formula is not Horn renameable, as shown by (a), $(\neg a)$, (b, c), $(b, \neg c)$, $(\neg b, c)$, $(\neg b, \neg c)$.

10. The *Prolog implementation* can be extended easily to permit backtracking, i.e., retracting added clauses in a LIFO manner. The reason is that `asserta` is used, and there is no `retract` in the basic on-line algorithm. Backtracking in an imperative language is a more difficult task, but the task can be guided by the Prolog method.

The famous satisfiability algorithm of Davis, Putnam, Logemann and Loveland [DP60, DLL62] relies heavily on a combination of unit resolution and backtracking. This algorithm should speed up the unit-resolution component.

2 The Algorithm

The main idea of the linear-time unit resolution algorithm is not to keep track exactly of the derived clauses, but just count the number of literals in them. When the count is reduced to one, a unit clause has been derived. Which literal remains, from those in the original clause, is then determined, and placed on the "agenda". When a literal is taken out of the agenda, if that literal is "new information" it is processed as described below, otherwise it is discarded. As seen in Figure 1, there is no essential difference between "batch" mode (`unitRes()`) and "on-line" mode (`unitRes()` followed by an arbitrary number of `updateClauses()` calls). In the discussion, we shall say the algorithm is in "on-line" mode if `updateClauses()` was the most recent top-level goal.

Excerpts of the code give details of the algorithm; the high level of Prolog makes pseudo-code unnecessary. For expressions involving *local cut*, `(`C `->` A `;` B`)` is usually readable as "if C, then A else B".

One key data structure is `occs`(x, L_p, L_n). Associated with each propositional variable x is a pair of lists: One list, L_p, contains the clause numbers in which x occurs positively; the other list L_n contains the clauses in which x occurs negatively. If either x or $\neg x$ is derived, it is placed on a list called the `Agenda`. (`Agenda` is not global.) When this literal is removed from the `Agenda`, if it turns out to be a *new* unit clause, `occs`(x, L_p, L_n) is processed and a new `minModel` fact for x is asserted.[2] (If x was already known, it is just discarded.) If positive x was derived, each clause in L_n (which contains $\neg x$) has its literal count reduced by one. If the negative $\neg x$ was derived, each clause in L_p has its literal count reduced by one. Literal counts are maintained in `clauseStatus`, discussed next. The total length of all lists in `occs` is proportional to the length of the formula, and each list is processed at most once, so all the processing requires only linear time. These time bounds hold whether the clauses appear "on-line", or in one batch, or in several batches. The time required to *construct* the `occs` structure involves Prolog internals, and is discussed later.

[2] The name `minModel` is imprecise: although these literals must be true in any model, their bindings are not necessarily sufficient to satisfy every clause.

```
unitRes(InFile, Result)  :-
        initDB,
        asserta(minModel(true,true)),
        asserta(minModel(false,false)),
        see(InFile), getClauses(ClauseList), seen,
        buildClauses(ClauseList, AgendaIn).
        unitReason(AgendaIn, _AgendaOut, Result).

updateClauses(UpdateFile, Result)  :-
        see(UpdateFile), getClauses(ClauseList), seen,
        buildClauses(ClauseList, AgendaIn).
        unitReason(AgendaIn, _AgendaOut, Result).

buildClauses([], []).
buildClauses([Lits | Clauses], AgendaOut)  :-
        genClauseNum(ClauseNum),
        processLits(Lits, ClauseNum, 0, Len, 0, LenOrig),
        asserta(clauseStatus(ClauseNum, Len, LenOrig)),
        asserta(clauseLits(ClauseNum, Lits)),
        ( Len == 0  ->
                AgendaOut = [false | AgendaRem]
        ; Len == 1  ->
                findLastLit(Lits, LastLit),
                AgendaOut = [LastLit | AgendaRem]
        ;
                AgendaOut =  AgendaRem
        ),
        buildClauses(Clauses, AgendaRem).
```

Figure 1: The main line of the algorithm: **`unitRes()`** begins a formula and processes it, while **`updateClauses()`** adds to a formula and processes it.

The second key data structure is **`clauseStatus`**(k, *Len*, *LenOrig*). As each clause is read in the input, it is assigned an integer index k, and its literal count *LenOrig* is determined. The field *Len* is the count of literals that are "live" in the sense that they are not contradicted by a known unit clause. Initially, this can be different from *LenOrig* only in on-line mode. Also in on-line mode, if some literal in the clause *agrees with* a known unit clause, then the clause is "subsumed", which is denoted by a negative *Len*.

Each input clause is converted into a list of literals, *Lits*, from whatever input format the program uses. (In an imperative language, external propositional variable names are mapped to integers via a symbol table during this step.[3]) A list of literal lists is the input to **`buildClauses`**, which among other things, asserts each clause as **`clauseLits`**(k, *Lits*).

As mentioned before, if *Lits* contains the complement of a literal that has just been removed from the **`Agenda`**, then the associated value of *Len* is reduced by one. Whenever *Len* $= 1$, the procedure **`findLastLit`**

[3]Dowling and Gallier unnecessarily shied away from claiming that the symbol table could be built and accessed in linear time. Assuming a fixed input alphabet, when there are many variable names, some names will require multiple letters, and the symbol table can indeed be built in time proportional to the number of *letters* in the input. This can be guaranteed with a *TRIE* structure, which is found in many algorithms texts. However, most implementors use a hash table and take their chances.

```
unitReason(Agenda0, AgendaOut, Result)  :-
        ( Agenda0 = []  ->
                AgendaOut = Agenda0,
                Result = dontknow
        ;
          Agenda0 = [false | AgendaRem]  ->
                asserta(minModel(false, true)),
                AgendaOut = AgendaRem,
                Result = unsat
        ;
          Agenda0 = [+(PropVar) | AgendaRem]  ->
                processUnit(PropVar, true, AgendaRem, AgendaNew),
                unitReason(AgendaNew, AgendaOut, Result)
        ;
          Agenda0 = [~(PropVar) | AgendaRem]  ->
                processUnit(PropVar, false, AgendaRem, AgendaNew),
                unitReason(AgendaNew, AgendaOut, Result)
        ;
                write('% should never get here'), nl, fail
        ).

processUnit(PropVar, Tval, AgendaRem, AgendaNew) :-
        ( minModel(PropVar, V)  ->
                ( V = Tval  ->
                        AgendaNew = AgendaRem
                ;
                        asserta(minModel(PropVar, Tval)),
                        AgendaNew = [false | AgendaRem]
                )
        ;
                asserta(minModel(PropVar, Tval)),
                getOccs(PropVar, ClauseListP, ClauseListN),
                ( Tval = true  ->
                        reduceClauses(ClauseListN, AgendaRem, AgendaNew)
                ;
                        reduceClauses(ClauseListP, AgendaRem, AgendaNew)
                )
        ).
```

Figure 2: The unit resolution engine. Other figures show `reduceClauses()` and two versions of `getOccs()`.

determines the one remaining "live" literal, which is placed on the `Agenda`. Procedure `findLastLit` is executed at most once per clause and can be done in time proportional to the clause's original length.

2.1 Prolog Implementation Issues

The code to maintain the `occs` data structure appears in Figure 4 in a form suitable for translation into an imperative language, such as C. Procedure `processLits()`, not shown, simply calls `processLit_C()` for each

```
reduceClauses([], AgendaRem, AgendaRem).
reduceClauses([ClNum | ClNums], AgendaRem, AgendaNew)  :-
        getClauseStatus(ClNum, Len, LenOrig),
        Len1 is Len - 1,
        pushClauseStatus(ClNum, Len1, LenOrig),
        ( Len1 = 1  ->
                getClauseLits(ClNum, Lits),
                findLastLit(Lits, LastLit),
                Agenda1 = [LastLit | AgendaRem]
        ;
                Agenda1 = AgendaRem
        ),
        reduceClauses(ClNums, Agenda1, AgendaNew).
```

Figure 3: Processing of clauses containing the *complement* of a newly derived unit clause. Clauses lengths are reduced, and any new unit clauses are placed on the agenda.

---

literal in a new clause, and combines the results to determine `Len` and `LenOrig`. The procedures `getOccs_C()` and `pushOccs_C()` simulate array access; however, `retract()` is avoided. Clearly, `processLit_C()` can run in constant time in an imperative language, as it simply inserts one element at the front of a linked list.

What may be surprising is that `processLit_C()` will not run in constant time in Prolog, even though both `asserta()` and `getOccs_C()` use constant-time clause indexing. The reason involves Prolog internals. The `asserta` predicate must rename, or "standardize apart", any free variables that may occur in the term being asserted. Therefore, it copies the entire structure. (Even if the implementation were optimized to use structure sharing on variable-free terms, it would still have to inspect the entire structure.) It follows that the time to build the `occs` lists of any one variable is quadratic in their final length. Indeed, if one variable occurs in some constant fraction, say 10%, of the clauses, then running time becomes quadratic.

The title promised linear time in Prolog. The solution is to recognize that `occs` is just a convenient representation of two binary relations `occP` and `occN`. Tuple `occP`(x, p) denotes that variable x occurs positively in clause number p. Tuple `occN`(x, n) means that x occurs negatively in n. In Prolog, these relations can simply be asserted directly. Clause indexing permits constant time per insertion. Each tuple needs to be *retrieved* at most once, when x or $\neg x$ becomes a new unit clause. Again, through clause indexing, `findall()` can retrieve all tuples matching a specified x on their first argument in time proportional to the number of such tuples. Thus linear time is achieved. The code just described appears in Figure 5.

Is this implementation issue a theoretical nicety that makes no difference on "practical problems"? To answer this question, we tested both implementations on a formula from a real application, which had 7652 atoms in 3296 clauses, 1355 variables, an average clause length of 2.32, and a maximum clause length of 6. The average size of both lists combined in `occs` is 5.6 items. The formula had 4 unit clauses initially and 48 more were derived. The Prolog-oriented implementation was about 14% faster. It required about 10 CPU seconds on a Sun sparc2. In a formula in which variables occur more frequently on average, the difference would be greater.

To verify linear time performance in online-mode, we divided the formula mentioned above into four approximately equal batches of clauses, and checked the CPU times spent in each batch. Each batch of clauses after the first was added on top of the previous ones. The program was run under Quintus Prolog and SICStus Prolog, both of which support dynamic clause indexing. Figure 6 shows the results. We report

```
processLit_C(+(PropVar), ClauseNum, Incr) :-
        ( minModel(PropVar, true)  ->
                Incr = -1
        ; minModel(PropVar, false)  ->
                Incr = 0
        ;
                getOccs_C(PropVar, OldOccsP, OldOccsN),
                pushOccs_C(PropVar, [ClauseNum | OldOccsP], OldOccsN),
                Incr = 1
        ).
processLit_C(~(PropVar), ClauseNum, Incr) :-
        ( minModel(PropVar, false)  ->
                Incr = -1
        ; minModel(PropVar, true)  ->
                Incr = 0
        ;
                getOccs_C(PropVar, OldOccsP, OldOccsN),
                pushOccs_C(PropVar, OldOccsP, [ClauseNum | OldOccsN]),
                Incr = 1
        ).

getOccs(PropVar, OldOccsP, OldOccsN)  :-
        getOccs_C(PropVar, OldOccsP, OldOccsN).

getOccs_C(PropVar, OccsP, OccsN)  :-
        occs(PropVar, OccsP, OccsN)  -> true.

pushOccs_C(PropVar, NewOccsP, NewOccsN)  :-
        asserta(occs(PropVar, NewOccsP, NewOccsN)).
```

Figure 4: Updating the `occs` data structure, with code appropriate for translation to an imperative language, in which `getOccs_C()` and `pushOccs_C()` would be array accesses to `occs`.

relative, rather than absolute, times, because different machines were used. Most inferences occurred in the second batch, explaining why `unitReason()` took longer there.

3 Conclusion and Future Work

An old algorithmic paradigm has been applied to perform propositional unit resolution in time linear in the length of the formula. The algorithm can run on-line without modification, and without noticable degradation. Prolog implementation issues had to be addressed to achieve linear time in that language. The use of Prolog also simplified the implementation by providing built-in symbol table services, data structure operations, and dynamic storage allocation. The implementation is quite flexible in that it is not necessary to know in advance how many variables or clauses will be presented, nor what the variable names will be.

The implementation described here has been used as a basis for other algorithms, which incorporate

```
processLit_PL(+(PropVar), ClauseNum, Incr) :-
        ( minModel(PropVar, true)  ->
                Incr = -1
        ; minModel(PropVar, false)  ->
                Incr = 0
        ;
                asserta(occP(PropVar, ClauseNum)),
                Incr = 1
        ).
processLit_PL(~(PropVar), ClauseNum, Incr) :-
        ( minModel(PropVar, false)  ->
                Incr = -1
        ; minModel(PropVar, true)  ->
                Incr = 0
        ;
                asserta(occN(PropVar, ClauseNum)),
                Incr = 1
        ).


getOccs(PropVar, OldOccsP, OldOccsN)  :-
        findall(ClNumP, occP(PropVar, ClNumP), OldOccsP),
        findall(ClNumN, occN(PropVar, ClNumN), OldOccsN).
```

Figure 5: Building the data structures with code that runs in linear time in Prolog. Relations `occP` and `occN` replace `occs`.

Pct. of Formula	25	25	25	25
Pct. Inferences	6	79	3	12
SICStus CPU %	24	25	26	25
– buildClauses	24	23	26	24
– unitReason	0	2	0	1
Quintus CPU %	23	27	24	26
– buildClauses	22	22	23	25
– unitReason	1	5	1	1

Figure 6: Processing time for four equal batches of clauses, as percent of total, under two Prolog systems.

backtracking, and as a basis for re-implementation into C. The groundwork for simple and efficient backtracking in the Prolog implementation has been laid by the consistent use of `asserta()` and the avoidance of `retract()`. The idea has yielded faster implementations of model searching algorithms, such as that of Davis, Putnam, Logemann and Loveland [DP60, DLL62].

Acknowledgements

This work was supported in part by NSF grant CCR-8958590, by equipment donations from Sun Microsystems, Inc., and software donations from Quintus Computer Systems, Inc.

References

[AI91] G. Ausiello and G.F. Italiano. Online algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming*, 10(1):69–90, 1991.

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A Linear-time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Information Processing Letters*, 8(3):121–123, March 1979.

[Asp80] B. Aspvall. Recognizing Disguised NR(1) Instances of the Satisfiability Problem. *Journal of Algorithms*, 1:97–103, 1980.

[BB79] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):30–59, 1979.

[DE92] M. Dalal and D. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44:173–180, December 1992.

[DG84] W. Dowling and J. Gallier. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[GU89] G. Gallo and G. Urbani. Algorithms for testing the satisfiability of propositional formulae. *Journal of Logic Programming*, 7(1):45–61, 1989.

[Har78] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, 1978.

[Lar92] T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, January 1992.

[Lew78] H. R. Lewis. Renaming a Set of Clauses as a Horn Set. *Journal of the Association for Computing Machinery*, 25(1):134–135, January 1978.

[Scu90] M. G. Scutella. A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *Journal of Logic Programming*, 8(3):265–273, 1990.