LINEAR-TIME ALGORITHMS FOR TESTING THE SATISFIABILITY OF PROPOSITIONAL HORN FORMULAE

WILLIAM F. DOWLING AND JEAN H. GALLIER

▷ New algorithms for deciding whether a (propositional) Horn formula is satisfiable are presented. If the Horn formula A contains K distinct propositional letters and if it is assumed that they are exactly $P_1, \ldots, P_K$, the two algorithms presented in this paper run in time $0(N)$, where N is the total number of occurrences of literals in A. By representing a Horn proposition as a graph, the satisfiability problem can be formulated as a data flow problem, a certain type of pebbling. The difference between the two algorithms presented here is the strategy used for pebbling the graph. The first algorithm is based on the principle used for finding the set of nonterminals of a context-free grammar from which the empty string can be derived. The second algorithm is a graph traversal and uses a "call-by-need" strategy. This algorithm uses an attribute grammar to translate a propositional Horn formula to its corresponding graph in linear time. Our formulation of the satisfiability problem as a data flow problem appears to be new and suggests the possibility of improving efficiency using parallel processors. ◁

The *satisfiability problem* for a class **C** of propositions is the problem of testing for any given formula A in **C**, whether some truth assignment v satisfies A. It is well known that the satisfiability problem is NP-complete for the class of all propositions [2, 8]. Therefore, if one is looking for a polynomial-time satisfiability test, one is led to consider subclasses of propositions. One such class is the class of propositional Horn formulae, which enjoys nice properties [1, 5, 6]. The class of propositional Horn formulae is obtained by restricting the form of the conjuncts in the conjunctive normal form of a proposition. If a proposition A has conjunctive normal form $C_1 \wedge \ldots \wedge C_m$, where each C_i is a disjunction of propositional letters (positive literal) or negations of propositional letters (negative literal), A is a Horn formula if and only if each C_i contains at most one positive literal.

Address correspondence to Jean H. Gallier, Department of Computer and Information Science, University of Pennsylvania, Moore School of Electrical Engineering D2, Philadelphia, PA 19104.

52 Vanderbilt Ave., New York, NY 10017 0743-1066/84/$03.00

From results of Jones and Laaser [6], it can be shown that testing the satisfiability of propositional Horn formulae is complete for the class **P** of problems solvable in polynomial time (in the size of the input) [2, 7, 8]. The method used in [6] to show that testing the satisfiability of Horn formulae is in **P** is to show that a polynomial-time algorithm can be obtained using unit-resolution [1, 5]. The complexity of this algorithm is $0(N^2)$, where N is the total number of occurrences of literals.

Alternatively, by observing that the satisfiability problem for Horn propositions reduces to the problem of determining whether the empty string belongs to the language generated by a context-free grammar $G = (N, T, P, S)$, a very simple algorithm running in time $0(N^2)$ can also be obtained (see Section 2).

In this paper, we present two linear-time algorithms for deciding whether a propositional Horn formula is satisfiable [1, 8], hence providing algorithms whose time complexity is optimal, since the input must be scanned at least once. Actually, these algorithms not only test whether a Horn formula A is satisfiable, but if so, find the least truth-assignment in the boolean algebra $\{\textbf{false}, \textbf{true}\}^K$ satisfying A (assuming that A contains K distinct positive literals, and that **false** < **true**).

The essence of these methods is to test whether sets of paths of a certain kind, called *pebblings*, exist in a graph associated with the Horn formula. In brief, the methods differ in the strategy used to find a pebbling.

The graph associated with a Horn proposition A describes the logical implications defined by the basic Horn propositions in it. The nodes of this graph are the distinct propositional symbols occurring in A plus two special nodes, one for **true** and one for **false**. The edges are labeled with basic Horn formulae. The fundamental property of the graph associated with the proposition A is that A is unsatisfiable if and only if there is a *pebbling* from **true** to **false**.

The first algorithm finds a pebbling in a breadth-first fashion and is a modification of the algorithm for finding the set of erasable nonterminals of a context-free grammar [4]. The second algorithm finds a pebbling by proceeding backward from **false**, using a "call-by-need" strategy.

One advantage of the second graph method is the fact that it proceeds from **false** in a "demand-driven fashion", and is therefore more oriented towards showing inconsistency.

Another advantage of our approach is that the representation of the problem leads to a data flow interpretation, which may lead to a very efficient algorithm if processors are used in parallel. We intend to investigate this question in a subsequent publication.

Since a proposition A is a tautology (satisfied by all possible truth assignments) if and only if $\neg A$ is not satisfiable, our algorithms can also be used as theorem provers for the class of negations of Horn formulae. In particular, our methods allow us to prove in linear-time theorems of the form $(C_1 \wedge \ldots \wedge C_m) \Rightarrow D$, where each C_i is a basic Horn formula, and D is a disjunction of conjunctions of literals in which at most one literal is negative. The second algorithm (the graph method) is particularly well suited to prove theorems of the above form, because it proceeds from **false** to **true** in a "call-by-need" fashion. This is even more interesting in the first-order case, since this generalizes PROLOG, in which D is typically a conjunction of positive literals. As a matter of fact, the second author has generalized the third algorithm of this paper to the first-order case, and built a prototype theorem-prover, HORNLOG

[3], which extends PROLOG in some respects. However, in order to keep this paper of reasonable length, we only present our algorithms for the propositional case.

1. PRELIMINARIES

Definition 1. A *literal* is either a propositional letter P (a positive literal) or the negation $\neg P$ of a propositional letter P (a negative literal). A *basic Horn formula* is a disjunction of literals, with at most one positive literal. A basic Horn formula will also be called a Horn clause, or simply a clause. A *Horn formula* is a conjunction of basic Horn formulae.

First, observe that every Horn formula A is equivalent to a conjunction of distinct basic Horn formulae by associativity, commutativity, and idempotence of "$\wedge$". Since "$\vee$" also has these properties, each basic Horn formula is equivalent to a clause of one of three types:

(i) Q, a propositional letter; or
(ii) $\neg P_1 \vee \ldots \vee \neg P_q$ where $q \geq 1$ and $P_1, \ldots, P_q$ are distinct propositional letters; or
(iii) $\neg P_1 \vee \ldots \vee \neg P_q \vee Q$ where $q \geq 1$, $P_1, \ldots, P_q$ are distinct propositional letters, and Q is a propositional letter.

For example, $(\neg P_1 \vee \neg P_2) \wedge (P_3) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_2) \wedge (\neg P_4 \vee P_5)$ is equivalent to $(\neg P_1 \vee \neg P_2) \wedge (P_3) \wedge (\neg P_4 \vee P_5)$. In the rest of this paper, it will be assumed that Horn formulae are in this "reduced" form, i.e., that there are no duplicate clauses and no duplicate literals within clauses.

Definition 2. A *directed edge-labeled graph* G is a triple (V, E, L), where V is a set of *nodes*, L is a set of *labels*, and E is a subset of $V \times L \times V$ of ordered triples called *edges*. Given an edge $e = (v_1, a, v_2)$, v_1 is the *source* of e, v_2 is the *target* of e, and a is the *label* of e.

Given an integer $n \geq 1$, let $[n]$ denote the finite set $\{1, 2, \ldots, n\}$.

2. A SIMPLE ALGORITHM RUNNING IN TIME $0(N^2)$

Assume that the Horn formula A is the conjunction of M basic Horn formulae, that the number of occurrences of literals in A is N, and the number of distinct propositional letters occurring in A is K.

We show that a context-free grammar GR_A can be constructed from A such that, if I is the start symbol of GR_A, A is unsatisfiable if and only if the empty string is derivable from I. As a consequence, we obtain a simple algorithm for testing the satisfiability of a Horn proposition, by adapting the well-known method for finding the set of erasable nonterminals of a context-free grammar [4].

Definition 3. Given a Horn proposition A, the context-free grammar $GR_A = (N, T, P, I)$ associated with A is defined as follows:

$N = \{P_1, \ldots, P_K, I\}$ where I is a new symbol;

$T = \varnothing$ (the empty set);

P is the set of productions defined as follows:

(i) For every basic Horn proposition of the form P_i, there is a production $P_i \to e$ (where e denotes the empty string);
(ii) For every basic Horn proposition of the form $\neg P_1 \vee \ldots \vee \neg P_q \vee Q$, there is a production $Q \to P_1 \ldots P_q$;
(iii) For every basic Horn proposition of the form $\neg P_1 \vee \ldots \vee \neg P_q$, there is a production of the form $I \to P_1 \ldots P_q$ (where I is the start symbol).

We now state the following Theorem reducing the satisfiability problem for Horn propositions to the well-known problem of finding the set of erasable nonterminals of a context-free grammar. However, instead of showing this Theorem immediately, we postpone its proof which can be obtained by introducing the concept of a *pebbling* which will be needed later.

Theorem 1. Given a Horn proposition A, A is unsatisfiable if and only if $I \Rightarrow^+ e$. Furthermore, if A is satisfiable, a letter Q in A must be **true** *if and only if $Q \Rightarrow^+ e$.*

Using Theorem 1, a simple algorithm is obtained by adapting the standard method for computing the set of nonterminals from which the empty string can be

ALGORITHM 1.

```
    Let V be a boolean array of size K, and consistent and change be
boolean flags.

  begin
        let S = {C1,...,CM}, where A = C1∧...∧CM
        consistent := true; change := true;
        for each propositional letter P in A do
                V(P) := false
        endfor;
        for each P such that (P) is a basic Horn formula in A do
                V(P) := true
        endfor;
        while change and consistent do
                    change := false;
                    for each basic Horn formula C in S
                            and consistent do
                       if C is of the form  ¬P1∨...∨¬Pq
                           and  V(P1)=...=V(Pq)=true then
                            consistent := false
                       else
                         if C is of the form  ¬P1∨...∨¬Pq∨P
                            and  V(P1)=...=V(Pq)=true
                            and  V(P)=false then
                            V(P) := true ; change := true;
                            S := S-{C}
                         endif
                       endif
                    endfor
        endwhile
  end
```

derived [4]. Recall that the set E of erasable nonterminals can be computed using the following sequence of sets:

$$E_0 = \{ A \in N \mid A \to e \in P \}$$

$$E_{k+1} = E_k \cup \{ A \in N \mid A \to B_1 \ldots B_n \in P \quad \text{and} \quad B_1, \ldots, B_n \in E_k \}.$$

Since the sets E_k are subsets of the finite set N of nonterminals, there is a least k, say k_0, for which $E_{k_0} = E_{k_0+1}$, and it can be shown that $E = E_{k_0}$. Algorithm 1 mimics the computation of the sets E_k.

If Algorithm 1 terminates with consistent = **true**, a satisfying assignment is given by V. The **while** loop can be executed at most $K + 1$ times, and the **for** loop at most N times (since it may be necessary to check every production). Hence, this algorithm is $0(N^2)$.

Note that the time complexity of Algorithm 1 can be improved if a more efficient way of checking the condition inside of the **for** loop can be found. Such a method will be presented in Sections 4 and 5.

3. THE GRAPH ASSOCIATED WITH A HORN PROPOSITION AND PEBBLINGS

The computation performed by Algorithm 1 can be clarified if we define a graph G_A associated with A. This graph implicitly represents all possible ways of checking the satisfiability of A, and is a powerful tool. Indeed, the satisfiability problem is expressible as a pebbling problem on G_A, and this provides intuition to the various strategies used by satisfiability testing algorithms.

The graph associated with a Horn proposition can be used to determine which propositional letters must be **true** in all truth assignments satisfying A, if any. A propositional letter Q is forced to be **true** iff either Q is a basic Horn formula in A, or there is some basic Horn formula $C_i = \neg P_1 \vee \ldots \vee \neg P_q \vee Q$ and it has already been established that $P_1, \ldots, P_q$ must all be **true**. If the above situation occurs and Q must also have the value **false** (which is the case if $\neg Q$ is a basic Horn formula in A), there is an inconsistency and A is not satisfiable. Our approach represents the proof process as a flow (of the truth value **true**) through a network of nodes that represents the implicational structure of a Horn formula. These nodes may be thought of or even implemented as individual processors that emit a boolean signal when their inputs surpass a certain threshold. The number of nodes of the network corresponding to A is only $K + 2$, and its total size including edges is approximately the size of A. Since it can be processed in linear time this is a fast and novel approach to the Horn formula satisfiability problem.

Definition 4. Given a Horn formula $A = C_1 \wedge \ldots \wedge C_M$, G_A is a labeled directed graph with $K + 2$ nodes (a node for each propositional letter occurring in A, a node for **true**, and a node for **false**) and set of labels $[M]$. It is constructed with i taking values in $[M]$ as follows:

(i) If the ith basic Horn formula in A is a positive literal Q, there is an edge from **true** to Q labeled i.

(ii) If the ith basic Horn formula in A is of the form $\neg P_1 \vee \ldots \vee \neg P_q$, there are q edges from $P_1, \ldots, P_q$ to **false** labeled i.

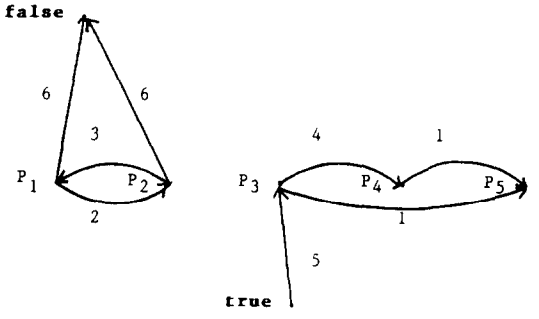
(iii) If the ith basic Horn formula in A is of the form $\neg P_1 \vee \ldots \vee \neg P_q \vee Q$, there are q edges from $P_1, \ldots, P_q$ to Q labeled i.

This graph is called the *graph corresponding to A*.

Example 1

$$A = (\neg P_3 \vee \neg P_4 \vee P_5) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_1) \wedge$$
$$(\neg P_3 \vee P_4) \wedge (P_3) \wedge (\neg P_1 \vee \neg P_2)$$

The graph G_A corresponding to A is the following:



We now present the theoretical basis for all graph-based satisfiability procedures.

Definition 5. Let $G = (V, E, L)$ be an edge-labeled directed graph. There is a *pebbling* of a node $Q \in V$ from a set $X \subseteq V$ if either Q belongs to X or, for some label i (corresponding to some basic Horn formula C_i), there are pebblings for $P_1, \ldots, P_q$ from X, where $P_1, \ldots, P_q$ are the sources of all incoming edges to Q labeled i.

Hence, Q can be pebbled from X if there is a sequence of "pebbling moves" such that, starting from nodes in X, a node is pebbled if and only if for some label i, all sources of incoming edges labeled i are pebbled.

Definition 6. The *length* d of a pebbling of Q from X is defined inductively as follows: if Q belongs to X, then $d = 0$. Otherwise, $d = 1 + \max\{d_1, \ldots, d_q\}$, where d_i is the length of the pebbling of P_i from X.

Theorem 2. Let A be a Horn formula, and $G_A = (V, E, [M])$ be the graph corresponding to A. If for some truth assignment v and some propositional letter Q, $v \models A$ and there is a pebbling of $Q \in V$ from {**true**}, *then* $v \models Q$.

PROOF. We proceed by induction on the length of pebblings. The case $d = 0$ is trivial. If there is a pebbling of length 1 from {**true**} to Q, there is an i such that (**true**, i, Q) $\in E$, which means that Q is a basic Horn formula in A. Since v satisfies A, v satisfies every basic Horn formula in A and so, $v \models Q$. If there is a pebbling of length $n > 1$ then, as above, there is an i such that the ith clause of A is $\neg P_1 \vee \ldots \vee \neg P_q \vee Q$, and there are pebblings of length less than n from {**true**} to each P_j $(1 \leq j \leq q)$. By induction, for each P_j, $v \models P_j$. Therefore, since $v \models \neg P_1 \vee \ldots \vee \neg P_q \vee Q$, we conclude that $v \models Q$. □

Corollary (Soundness). *A is unsatisfiable if there is a pebbling of* **false** *from* {**true**}.

This follows since if A were satisfiable, there would be some truth assignment v such that $v \vDash A$, and by virtue of the pebbling of **false** from {**true**}, we would have $v \vDash$ **false**, a contradiction.

Completeness is shown using the following theorem.

Theorem 3. Let $G_A = (V, E, [M])$ be the graph corresponding to a Horn formula A. If there is no pebbling of **false** *from* {**true**} *then A is satisfiable.*

PROOF. We will define a valuation v and then show that $v \vDash A$. Let $v \vDash P_i$ iff there is a pebbling of P_i from {**true**}. We show that v satisfies every basic Horn formula C_k in A. There are three cases depending on the form of C_k.

(i) If $C_k = Q$ is a propositional symbol in A, then (**true**, k, Q) is in E, there is a pebbling of length 1 from {**true**} to Q, and therefore $v \vDash Q$.

(ii) If $C_k = \neg P_1 \vee \ldots \vee \neg P_q \vee P_{q+1}$ is in A but v does not satisfy C_k, then $v \vDash P_i$ $(1 \leq i \leq q)$ and so there is a pebbling of each P_i from {**true**}. But then, there is a pebbling of P_{q+1} from {**true**} and $v \vDash P_{q+1}$, which implies that v satisfies C_k, a contradiction.

(iii) If $C_k = \neg P_1 \vee \ldots \vee \neg P_q$, since for each P_i there is an edge (P_i, k, **false**) from P_i to **false** and there is no pebbling of **false** from {**true**}, for some i, say i_0, there is no pebbling of P_{i_0} from {**true**}. Hence, v assigns the value **false** to P_{i_0} and $v \vDash C_k$. Since v satisfies every basic Horn formula in A, v satisfies A.

If we view {**false**, **true**} as a boolean algebra in which **false** < **true**, the K-fold Cartesian product {**false**, **true**}^{K} is also a boolean algebra. Then, we have the following corollary.

Corollary. Given a Horn formula A, let $G_A = (V, E, [M])$ be its corresponding graph.

(1) *A is satisfiable if and only if there is no pebbling of* **false** *from* {**true**}.

(2) *If A is satisfiable, the truth assignment $(v(P_1), \ldots, v(P_K))$ such that $v(P_i) =$* **true** *if and only if there is a pebbling of P_i from* {**true**} *and $v(P_i) =$* **false** *otherwise, is the least truth assignment in the boolean algebra* {**false**, **true**}^{K} *satisfying A.*

Theorem 1 can now be proved by showing the following lemma whose simple proof is omitted.

Lemma 1. Given a Horn proposition A, its grammar GR_A, and its graph G_A, there is a pebbling of Q from {**true**} *if and only if $Q \Rightarrow^+ e$.*

The above lemma indicates that there is a duality between pebblings in the graph G_A and derivations in the grammar GR_A. This duality helps in understanding how satisfiability methods actually work. The method given in Section 2 and unit resolution attempt to find a pebbling, starting from **true**. On the other hand, the graph method presented in Section 6 attempts to find a derivation of the empty string from **false** (or any other propositional letter).

In the next two sections, we present linear-time algorithms for deciding the satisfiability of a Horn proposition. The key to linear-time complexity is to store and

propagate information which indicates when a positive literal in a basic Horn proposition is "ready to be pebbled". Such a method is presented in the next two sections.

4. A LINEAR-TIME ALGORITHM REFINING ALGORITHM 1

Before presenting a linear-time refinement of Algorithm 1, we discuss the representation of Horn formulae.

4.1. Representation of Horn formulae

Since we are concerned with the complexity of an algorithm for testing the satisfiability of a Horn formula, it is important that the actual representation of Horn formulae be absolutely clear since, as we shall see later, this affects the complexity of the algorithm.

If A is a Horn formula containing K distinct propositional letters, we will assume that it is represented as a string in the language defined by the context-free grammar given below, and that if A contains K distinct propositional letters, they are exactly the letters $P_1, \ldots, P_K$. This seemingly innocuous assumption actually affects the complexity of the algorithm as we shall see later. However, we do not feel that it is an unreasonable assumption, since the problem of interest is to test the satisfiability of Horn formulae, and not to find the set of distinct propositional letters in it.

BNF Defining the Syntax of Horn Formulae

⟨S⟩ → ⟨Horn-clause⟩
⟨Horn-clause⟩ → ⟨Basic-Horn⟩|⟨Basic-Horn⟩ ∧ ⟨Horn-clause⟩
⟨Basic-Horn⟩ → (⟨neg-lit-list⟩)
|(⟨neg-lit-list⟩ ∨ ⟨pos-lit⟩)|
(⟨pos-lit⟩)
⟨neg-lit-list⟩ → ⟨neg-lit⟩|⟨neg-lit-list⟩ ∨ ⟨neg-lit⟩
⟨neg-lit⟩ → ¬ID
⟨pos-lit⟩ → ID

In this BNF, ID is treated as a terminal. In an implementation, ID would be decoded by the lexical analyzer.

In order to speed up the selection of the basic Horn clause in the **for** loop of Algorithm 1, we shall compute for each positive literal P, the list *clauselist* [P] of all basic Horn propositions in which P occurs as a negative literal. We also compute the arrays *numargs* and *poslitlist* of dimension M (the number of basic Horn propositions) such that, numargs[n] is the number of negative literals in clause number n that have current truth value **false**, and poslitlist[n] is the positive literal occurring in clause n, if any. If clause n does not contain a positive literal, poslitlist[n] = 0 (0 corresponds to **false**). Then, a basic Horn clause C_n is ready to be processed if numargs[n] = 0, meaning that all negative literals in C_n have been evaluated to **true**.

We keep the basic Horn clauses ready to be processed inside of the **for** loop in a queue which is updated whenever a new positive literal is evaluated to **true**. Initially, the queue contains the basic Horn propositions consisting of a single positive literal.

Upon entry to the **while** loop, the queue contains the basic Horn clauses which have just been processed, that is, such that the positive literal in them has been evaluated to be **true**. The size of the queue is held in oldnumclause. During the **for** loop, each clause on the queue is popped and processed as follows. Let clause1 be the current head of the queue, and let nextpos = poslitlist[clause1] be the positive literal in clause1. Since the basic Horn clause clause1 was entered into the queue because all of its negative literals are **true**, nextpos is set to **true** if it is not already **true**. Then, for each basic Horn clause clause2 on the list clauselist[nextpos] of clauses containing nextpos negatively, numargs[clause2] is decremented by one. If numargs[clause2] = 0, all negative literals in clause2 are **true**, and clause2 is ready to be processed. If clause2 contains a positive literal n = poslitlist[clause2], clause2 is entered into the queue to be processed at the next round. Otherwise, clause2 only contains negative literals (which is indicated by poslitlist[clause2] = 0). If the queue is empty, the Horn clause is consistent. Otherwise the clause on the top of the stack is popped and the **while** loop is reentered. The number of basic Horn clauses entered in the queue during the **for** loop is newnumclause. At the end of the **for** loop, oldnumclause is reset to newnumclause and the **while** loop is reentered if some new literal has been found **true**, which is indicated by the fact that the queue is nonempty.

Each propositional symbol P_i is represented as a record containing a value field "val" and a pointer field "clauselist" to the list of clauses containing P_i negatively. See Algorithm 2.

ALGORITHM 2.

```
program algorithm2(infile,outfile);
   {k = number of distinct positive literals in A
    m = number of basic Horn clauses in A}
   constant nodefalse = 0;
   type  clause = record
                        clauseno: 1..maxclause;
                        next: ^clause
                  end;
   type  literal = record
                         val: boolean;
                         clauselist: ^clause
                   end;
   type  Hornclause =  array[1..maxliteral] of literal;
   type  count = array[1..maxclause] of nodefalse..maxliteral;
   var A: Hornclause;
            numargs, poslitlist: count;
            queue: queuetype;
            numpos: 0..maxclause; {number of positive unit clauses}
            consistent: boolean;
begin
    input(A);
    initialize(clauselist, numargs, poslitlist);
    let queue = list of basic Horn clauses consisting of a single
                    positive literal, and numpos be their number.
    consistent := true;
    satisfiable(A,queue,consistent);
    if consistent then
         print('Satisfiable Horn Clause');
         printassignment
    else
         print('Unsatisfiable Horn Clause')
    endif
end
```

```
procedure satisfiable(var A: Hornclause, queue: queuetype,
                                              consistent: boolean);
  var clause1, clause2: 1..maxclause;
        n: nodefalse..maxliteral;
        nextpos: 1..maxliteral;
        oldnumclause, newnumclause: 0..maxclause;
  begin
     oldnumclause := numpos; {number of positive unit clauses}

     {Propagate true as long as new literals become true
      and no inconsistency}

     while queue <> nil and consistent do
          newnumclause := 0;

     {propagate true for every clause in the clauselist for the
      positive literal nextpos in clause1, the head of the queue}

        for i := 1 to oldnumclause and consistent do
              clause1 := pop(queue);
              nextpos := poslitlist(clause1);

     {for every clause clause2 on the clauselist for nextpos,
      decrement the number of negative literals and check
      whether the positive literal n in clause2 can be computed}

              for clause2 in A[nextpos].clauselist do
                    numargs[clause2] := numargs[clause2] -1 ;

      {If all negative literals in clause2 are true and the
       the positive literal is not already computed, then compute}

                  if numargs[clause2] = 0 then
                        n := poslitlist[clause2];
                        if not A[n].val then

       {If n is a positive literal, then evaluate and enter clause2
        into the queue. Otherwise, n corresponds to false and
        A is inconsistent}

                                  if n <> nodefalse then
                                        A[n].val := true;
                                        queue :=push(clause2,queue);
                                        newnumclause := newnumclause+1
                                  else
                                        consistent := false
                                  endif
                               endif
                        endif
                  endfor
        endfor;
        oldnumclause := newnumclause
     endwhile
  end
```

ALGORITHM 2 (Continued)

4.2. Complexity of Algorithm 2

Assuming that the distinct positive literals in A are $P_1, \ldots, P_K$, it is easy to initialize the arrays numargs and poslitlist and the lists clauselist in linear time. However, if we allowed arbitrary identifiers for the propositional letters, we would have to build a symbol table to uniquely index the distinct identifiers, and this would require $N \log(N)$ steps. Since the problem of interest is to test satisfiability and not a parsing problem, we do not feel that the above assumption is unreasonable.

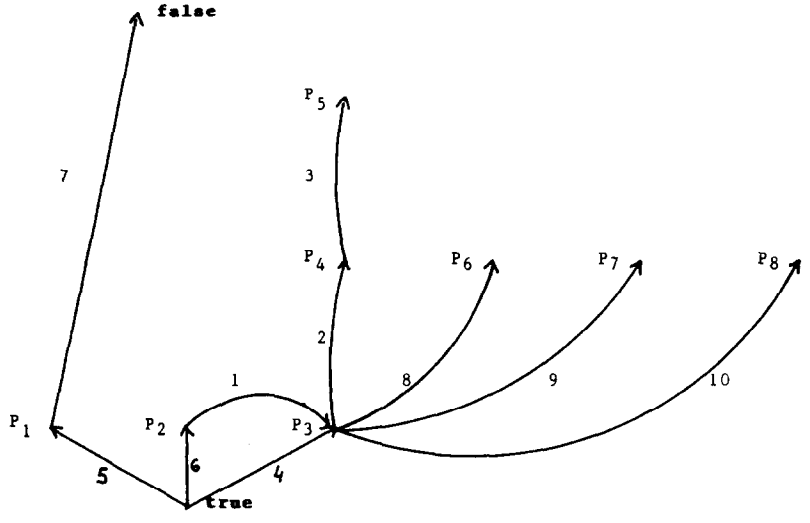
Note that every basic Horn clause in A is entered *at most once* into the queue. Indeed, a basic Horn clause clause2 is entered into the queue if and only if all the negative literals in it are **true** and the positive literal in it is not already **true**. As soon as clause2 is entered, the positive literal in it is set to **true**, thus preventing reentry. Whenever a clause clause1 is removed from the queue upon entrance to the **while** loop, all clauses clause2 in the clauselist for the positive literal nextpos in clause1 are considered. Notice that this corresponds to the deletion of negative occurrences of nextpos in A, and that these occurrences are disjoint for each round through the **while** loop. Hence, the contribution of the **while** loop is proportional to the number of negative occurrences of literals in A, which is linear in N, the total number of occurrences in A.

Note that Algorithm 2 finds pebblings in the graph G_A by moving from **true** to **false** in a *breadth-first* fashion.

Example 2

$A = (\neg P_2 \vee P_3) \wedge (\neg P_3 \vee P_4) \wedge (\neg P_4 \vee P_5) \wedge$
$(P_3) \wedge (P_1) \wedge (P_2) \wedge (\neg P_1) \wedge$
$(\neg P_3 \vee P_6) \wedge (\neg P_3 \vee P_7) \wedge (\neg P_3 \vee P_8)$

Graph associated with A:



Initially, the queue contains the clauses (4, 5, 6) with 4 the head element, and since clause 4 consists of the positive literal P_3, Algorithm 2 will compute P_4, P_6, P_7, P_8, and find the inconsistency in computing P_1.

5. A "CALL-BY-NEED" GRAPH ALGORITHM

The algorithm given below checks whether for every propositional letter Q, the empty string can be derived from Q (in the grammar GR_A associated with A). Actually, the algorithm starts by checking whether $I \Rightarrow^+ e$, thus checking for

consistency first. This algorithm proceeds in a "call-by-need" fashion, in the sense that to determine whether $Q \Rightarrow {}^{+}e$, it determines whether for some basic Horn clause $C_i = (\neg P_1 \vee \ldots \vee \neg P_q \vee Q)$, $P_1 \Rightarrow {}^{+}e, \ldots, P_q \Rightarrow {}^{+}e$. Hence, this algorithm proceeds from **false** to **true**, contrary to the previous one.

The algorithm is conveniently implemented as a recursive procedure which, given a basic Horn formula $C_i = \neg P_1 \vee \ldots \vee \neg P_q \vee Q$, finds recursively whether all P_i must be **true** in order to set Q to **true**. Observe that in order to find whether Q should be **true**, it is sufficient to visit all the nodes reachable from Q in the graph G'_A obtained from G_A by *reversing* the direction of the edges. For instance, in Example 1, in order to know whether P_5 should be **true**, since there are edges from P_3 and P_4 to P_5, we must find whether both P_3 and P_4 are **true**. Since there is an edge from **true** to P_3 and an edge from P_3 to P_4, all of P_3, P_4, P_5 are indeed **true**.

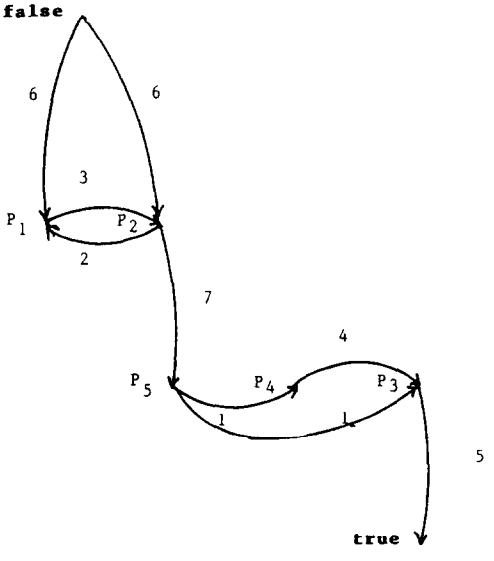
Hence, in writing this algorithm, it is convenient to consider the graph G'_A obtained from G_A by reversing the direction of the edges.

Since the graph may have cycles (as the cycle P_1, P_2, P_1 in Example 1), it is necessary to use a marking technique to prevent the procedure from looping. Choosing the right kind of marking is actually rather subtle, as illustrated by the following example.

Example 3

$$A = (\neg P_3 \vee \neg P_4 \vee P_5) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_1) \wedge$$
$$(\neg P_3 \vee P_4) \wedge (P_3) \wedge (\neg P_1 \vee \neg P_2) \wedge (\neg P_5 \vee P_2)$$

The graph G'_A corresponding to A is the following:



The difficulty is that we want to minimize both the number of visits to nodes, and the number of truth computations (that is, determining whether a positive literal in a clause has the value **true**). The first solution that comes to mind is to mark the nodes as they are visited, and only visit unmarked nodes. Unfortunately, this does not work. Indeed, if the algorithm visits the path beginning with **false**, P_2, P_1, P_5, even

though P_2 will eventually get the value **true**, P_1 will not since it has been marked and therefore, will not be revisited. The problem is that there may be different ways of entering a node and multiple visits must be allowed.

The solution is to mark the edges and allow a visit to a node provided that *either there is some unmarked incoming edge to it, or one of its immediate successors has some unmarked outgoing edge.* To implement the above strategy, each edge of the graph has a field *visited*, and each node has a field *marked*. The *marked* field is a counter holding the number of nonvisited outgoing edges from a node, and it is decremented every time such an edge is visited. In order to perform truth evaluations only when necessary, we use the lists clauselist and the array numargs. As in the previous algorithms, whenever it is found that a positive literal P has the value **true**, the counters corresponding to all the clauses on the clauselist corresponding to P are updated (decremented by 1). To avoid recomputing the value of a positive literal, a field *computed* is then set to **true**. In this way, every positive literal is computed at most once.

The graph G'_A is implemented as an array of linked lists, each entry in the array being a record corresponding to a node of the graph, and each linked list being the list representing all edges having that node as source. In order to speed up the algorithm, for every node P (positive literal), we create a list *successors* consisting of records (one for each label in the set of all outgoing edges with source P). Each record contains a label number i and a pointer to the list of target nodes of all edges with source P labeled i.

The graph is initialized in such a way that, for every basic Horn clause consisting of a single positive literal, the *val* field of the corresponding node is set to **true**, and it is set to **false** for other nodes. The *visited* field of every edge is set to **false** and the *computed* field of every node P is set to the sum of the number of negative literals in all basic clauses containing P.

5.1. *Algorithm Buildgraph*

The algorithm *buildgraph* builds the graph G'_A associated with a Horn formula A and initializes the fields. Since we have checked that the BNF given above is SLR(1), we can use a syntax-directed translation scheme for building the graph. The abstract translation of a Horn clause to its associated graph can be rigorously and elegantly specified by an attribute grammar. Using an attribute grammar to specify such a translation scheme is not a major innovation, but it is one of the distinctive features of this paper. Indeed, the translation of a Horn clause into its associated graph specified by the attribute grammar given below is independent of the evaluation scheme used. Hence, this specification is truly denotational, which is an elegant feature of this approach.

The following attribute grammar specifies the translation.

⟨S⟩ → ⟨Horn-clause⟩
newG(0) = newG(1)
Num-basic(1) = 1

⟨Horn-clause⟩ → ⟨Basic-Horn⟩
Num-basic(1) = Num-basic(0)
newG(0) = newG(1)

⟨Horn-clause⟩ → ⟨Basic-Horn⟩ ∧ ⟨Horn-clause⟩
Num-basic(1) = Num-basic(0)
Num-basic(3) = Num-basic(0) + 1
newG(0) = union(newG(1), newG(3))
⟨Basic-Horn⟩ → (⟨neg-lit-list⟩)
newG(0) = makegraph(**false**, list-node(2), Num-basic(0))

⟨Basic-Horn⟩ → (⟨neg-lit-list⟩ ∨ ⟨pos-lit⟩)
newG(0) = makegraph(node(4), list-node(2), Num-basic(0))

⟨Basic-Horn⟩ → (⟨pos-lit⟩)
newG(0) = makegraph(node(2), **true**, Num-basic(0))

⟨neg-lit-list⟩ → ⟨neg-lit⟩
list-node(0) = append(node(1), NIL)

⟨neg-lit-list⟩ → ⟨neg-lit-list⟩ ∨ ⟨neg-lit⟩
list-node(0) = append(node(3), list-node(1))

⟨neg-lit⟩ → ¬ID
node(0) = lexval(2)

⟨pos-lit⟩ → ID
node(0) = lexval(1)

Following the usual conventions, symbols occurring in a production are indexed from left to right, starting with 0. The attributes newG, node, list-node, and lexval are all synthesized attributes. The attribute Num-basic is inherited.

For this particular attribute grammar, a bottom-up syntax-directed translation can be used, and it is enough to initialize Num-basic to one and increment it whenever a reduction by production ⟨Horn-clause⟩ → ⟨Basic-Horn⟩ ∧ ⟨Horn-clause⟩ is made. The attribute lexval returns the integer code assigned to a propositional letter assigned by the lexical analyzer (n is assigned to P_n). All other attributes and functions are self-explanatory. It is obvious that the fields val, computed, and visited can be initialized during the construction of the graph.

Once the graph G'_A is constructed, the Algorithm 3 is used to test the satisfiability of A.

The procedure *update* updates numargs[n] for every clause n in the clauselist corresponding to the positive literal current.

5.2. *Complexity of Algorithm 3*

5.2.1. ALGORITHM BUILDGRAPH

Assuming that the input Horn formula A has K distinct propositional letters and that they are exactly $P_1, \ldots, P_K$, neither union nor makegraph has to make comparisons to find out which nodes are identical. Indeed, the nodes of the graph are represented as records in a linear array, and each node is identified by its integer index. Hence, during the parse using the shift/reduce algorithm, each reduction has a cost which is proportional to the number of symbols in the right-hand side of the production involved, and the total number of steps is proportional to the number in symbols in A, which is $0(N)$, where N is the number of occurrences of literals in A.

```
program algorithm3(infile,outfile);
    constant nodefalse = 0;
             maxclause = 500;
             maxnode = 500;
  type edge = record
                    target: nodefalse..maxnode;
                    visited: boolean;
                    next: ^edge
                  end;
  type clause = record
                      clauseno: 1..maxclause;
                      next: Clause
                    end;
  type pairitem = record
                       clauseno: 1..maxclause;
                       edgelist: ^edge;
                       next: ^pairitem
                     end;
  type succptr = ^pairitem;
  type count = array[1..maxclause] of 0..maxnode
  type node = record
                    marked: 0..maxnode;
                    computed: boolean;
                    val: boolean;
                    clauselist: ^clause;
                    successors: succptr
                  end;
  type graph = record
                     m: 1..maxclause;
                     k: nodefalse..maxnode;
                     nodes: array [nodefalse..maxnode] of node
                   end
  type nodeindex = nodefalse..maxnode;
  type labelindex = 1..maxclause;
  var g: graph;
        numargs: count; {number of negative literals in each clause}
        poslitlist: count; {positive literal in each clause}
        current: nodeindex;
        numpos: 0..maxclause; {number of positive unit clauses}

   begin
         {build and initialize graph}
         buildgraph(g);
         {call traverse from false to check for unsatisfiability}
         if numpos = 0 then {no positive unit clauses, satisfiable}
               print('Satisfiable Horn Clause')
         else
            traverse(nodefalse,g);
            if g.nodes[nodefalse].val then
            {Clause is unsatisfiable}
                  print('Unsatisfiable Horn Clause')
            else
            {Clause is satisfiable, compute truth assignment}
               print('Satisfiable Horn Clause');
               for current := 1 to g.k do
                  if not g.nodes[current].computed then
                              traverse(current,g)
                  endif
               endfor
            endif
         endif;
         {print satisfying assignment}
         print-assignment
   end
```

ALGORITHM 3.

```
procedure traverse(current: nodeindex; g: graph);
var  arc: ^edge;
         tagset: succptr;
         j: labelindex;
        begin

           {If val of current is not already computed,
            call traverse recursively}

          if not g.nodes[current].computed then

             {Take care of nodes initialized to true}

             if g.nodes[current].val then
                g.nodes[current].computed := true;
                update(current,numargs)
             else

             {For every clause number j, compute the value of
              the targets of all edges with source current labeled j,
              as long as current.val is not true}

                tagset := g.nodes[current].successors;
                for each j in tagset and not g.nodes[current].val do
                    arc := tagset.edgelist;

                    {traverse recursively for every arc labeled j}

                    while arc <> NIL do

                    {If arc not visited then call traverse}

                      if not arc^.visited then
                        g.nodes[current].marked := g.nodes[current].marked-1;
                        arc^.visited := true;
                        traverse(arc^.target,g)

                    {If all arcs visited and target node has some unmarked
                     outgoing edge, then call traverse}

                      else
                         if (g.nodes[arc^.target].marked <> 0) and
                                 (g.nodes[current].marked = 0)       then
                               traverse(arc^.target,g)
                         endif
                      endif;
                      arc := arc^.next
                    endwhile; {while arc <> NIL do}

                      {If not already computed and
                       all arguments for clause j are available,
                       compute the truth value of current}

                      if not g.nodes[current].computed then
                         if  numargs[j] = 0  then

                      {update counter for every clause in the clauselist
                       corresponding to current and set to true}

                           update(current,numargs);
                           g.nodes[current].val := true
                         endif {if numargs[j] = 0 then)}
                      endif
                    endfor;
                    g.nodes[current].computed := true
                 endif {if g.nodes[current].val }
              endif {if not g.nodes[current].computed ...}
            end; {traverse}
```

ALGORITHM 3 (Continued)

5.2.2. ALGORITHM SATISFIABLE

Observe that the graph G'_A has $K+2$ nodes and $N-P$ edges, where N is the number of occurrences of literals in A and P the number of basic Horn formulae containing both a positive and a negative literal.

CRUCIAL OBSERVATION. Due to the marking, only edges reachable from current are visited, and each such edge is visited exactly once (edges are marked using the field "visited"). This implies that the total number of calls to traverse is bounded by $N+1$, where N is the number of occurrences of literals in A.

Indeed, in the worse case, for every basic Horn clause in A, the positive literal (if any) in it and the targets of the edges which correspond to the negative literals in the basic Horn clause are visited once. This accounts for N visits, plus the starting node **false**. Also, the truth value of every node (current) is computed exactly once, since it is marked when it is computed (using the "computed" field). Since the contributions of the calls to update are disjoint and correspond to the deletion of occurrences of negative literals in A, the cost of the truth computations is also line in N. Hence, the complexity of *traverse* is linear in N. Since the construction of the graph has complexity $0(N)$, the complexity of Algorithm 3 is $0(N)$.

REMARK. The assumption that if the Horn formula A contains K propositional letters, they are exactly $P_1,\ldots,P_K$ affects the complexity of the algorithm *build-graph*. Indeed, if the letters occurring in A are $P_{i_1},\ldots,P_{i_K}$ where $\{i_1,\ldots,i_K\}$ is different from $\{1,\ldots,K\}$, in building the graph G_A it is necessary to build a symbol table, which amounts to sorting $P_{i_1},\ldots,P_{i_K}$. Since the complexity of sorting is $0(N\log(N))$, the construction of the graph would have complexity $0(N\log(N))$. However, this assumption does not affect the complexity of the algorithm *traverse*, since the input is the graph, in which the nodes have already been sorted.

6. CONCLUSION

We have presented two linear-time algorithms for testing the satisfiability of propositional Horn formulae. We have shown that given a Horn proposition A, a context-free grammar GR_A and a graph G_A can be constructed and that the satisfiability problem for A is equivalent to two dual problems:

(1) Whether the node **false** can be pebbled from **true** in the graph G_A.
(2) Whether the empty string can be generated by GR_A.

The difference between these algorithms is in the strategy used for pebbling. Algorithm 2 proceeds from **true** to **false** in a breadth-first fashion. On the other hand, Algorithm 3 proceeds from **false** to **true** in a depth-first fashion, trying to detect whether the empty string can be derived from **false**. Hence, Algorithm 2 will do more work on clauses whose graph is very wide, and Algorithm 3 will work harder on clauses whose graph has many long paths from **false**. For instance, Algorithm 3 is very fast on the following clause generalizing Example 2, but Algorithm 2 does a lot of redundant work.

Example 4

$$A = (\neg P_2 \vee P_3) \wedge (\neg P_3 \vee P_4) \wedge \ldots \wedge (\neg P_{n-1} \vee P_n) \wedge$$
$$(P_3) \wedge (P_1) \wedge (P_2) \wedge (\neg P_1) \wedge$$
$$(\neg P_3 \vee P_{n+1}) \wedge (\neg P_3 \vee P_{n+2}) \wedge \ldots \wedge (\neg P_3 \vee P_{2n})$$

There are $n + 2$ nodes at height 2, which hurts algorithm 2. However, Algorithm 3 finds immediately the path from **false** to **true**. This analysis suggests that an algorithm which proceeds simultaneously from **true** to **false** and from **false** to **true**, in a dovetailing fashion, might be more efficient. We leave the design of such an algorithm as a topic for further research.

A nice feature of the algorithms of this paper is that they can take advantage of parallelism. Furthermore, because Algorithm 3 performs a "lazy unsatisfiability check", it has an interesting generalization to the first-order case. This generalization has been worked out and implemented, see Gallier [3].

It is also possible to design an algorithm searching the graph from bottom-up in a depth-first fashion, and such an algorithm happens to correspond to positive unit resolution [5]. However, this algorithm does not appear to be as easily amenable to parallelism, and for this reason, is omitted.

The formulation of the satisfiability problem as a *data flow* problem is also interesting, in the sense that it suggests a solution using processors attached to the nodes of the graph and running in parallel. We intend to investigate this problem in a subsequent paper.

REFERENCES

1. Chang, C. and Lee, R., *Symbolic Logic and Mechanical Theorem-Proving*, Academic Press, New York, 1973.
2. Cook, S. A., The Complexity of Theorem-proving Procedures, *Proc. Third ACM Symp. on Theory of Computing*, 151–158 (1971).
3. Gallier, J. H., HORNLOG: A First-Order Theorem Prover for Negations of Horn Clauses based on Graph-Rewriting, Technical Report, Department of Computer and Information Science, University of Pennsylvania, 1984.
4. Harrison, M. A., *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.
5. Henschen, L. and Wos, L., Unit Refutations and Horn Sets, *J. ACM* 21:590–605 (1974).
6. Jones, N. D. and Laaser, W. T., Complete Problems for Deterministic Polynomial Time. *Theor. Comp. Sci.* 3:107–117 (1977).
7. Karp, R. M., Reducibility Among Combinatorial Problems, in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85–103.
8. Lewis, H. R. and Papadimitriou, C. H., *Elements of the Theory of Computation*, Prentice-Hall, New York, 1982.