An Efficient Algorithm for Unit Propagation

Hantao Zhang*
Computer Science Department
The University of Iowa
Iowa City, IA 52242
hzhang@cs.uiowa.edu

Mark E. Stickel†
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025
stickel@ai.sri.com

1 Introduction

In recent years, there has been considerable renewed interest in the satisfiability (SAT) problem of propositional logic. Many tools for the SAT problem use unit propagation. Given a set of ground clauses as constraints, unit propagation is the process of repeatedly applying unit-resolution and unit-subsumption operations to clauses in the set until no new unit clauses can be generated. It is well known that unit propagation is a complete decision procedure for the satisfiability of ground Horn theories. For Horn theories, Dowling and Gallier's linear time algorithm [4] is optimal because processing the input clauses takes linear time, too. Unit propagation is also used in testing the satisfiability of non-Horn theories, which have wider application in practice than Horn theories. We present a new unit-propagation algorithm that needs amortized linear time for unit resolution but amortized constant time for unit subsumption. The new algorithm has a simple data structure and performs constantly better than the best known algorithms in our experiments. Its efficiency allows us to solve many open problems in the study of quasigroups.

Traditionally, unit propagation is implemented as a series of applications of unit resolution and unit subsumption. A propositional clause is a disjunction of literals—each of which is either a propositional variable or its negation. A unit clause is a clause with a single literal. Let S be a set of propositional clauses and v be a literal. Then S can be divided into three sets:

- $P = \{v \vee P_1, \ \ldots, \ v \vee P_n\}$—the clauses that contain v.
- $Q = \{\overline{v} \vee Q_1, \ \ldots, \ \overline{v} \vee Q_m\}$—the clauses that contain v's complement.
- $R = \{R_1, \ \ldots, \ R_l\}$—the clauses that do not contain v or its complement.

*Partially supported by the National Science Foundation under Grants CCR-9504205 and CCR-9357851.

†Partially supported by the National Science Foundation under Grants CCR-9408630 and CCR-8922330.

Unit resolution of S with v will transform Q into $Q' = \{Q_1, \ \ldots, \ Q_m\}$ and *unit subsumption* of S by v will delete P from S. Altogether, S will be transformed into $S' = Q' \cup R$ by these two operations. As long as S' contains a unit clause $\{u\}$, the process is repeated with u, until either the empty clause is derived or no more new unit clauses can be generated.

Crawford and Auton [1] presented a very efficient implementation of a linear time algorithm that does not construct S' explicitly. In their implementation, for each propositional variable there are lists of all the clauses that contain the variable positively or negatively. Each clause has a counter that contains the number of propositional variables appearing in the clause that have not been assigned a truth value or, if the clause has been subsumed, the value `inactive`. An active clause is one whose counter value is not `inactive`. When a variable is assigned the value `true`, (a) the counter of all the active clauses that contain the variable negatively is decreased by one (this corresponds to unit resolution) and (b) the counter of all the clauses that contain the variable positively is set to `inactive` (this corresponds to unit subsumption). Assignments of `false` are done analogously. If the new counter value is zero, an empty clause is found; if it is one, a unit clause is found. Obviously, this algorithm takes linear time to do both unit resolution and unit subsumption. That is, to transform S into S' as given above, it takes $O(|Q| + |P|)$.

In the following, we present a new unit-propagation algorithm that takes only $O(|Q|)$ time for `true` assignments and $O(|P|)$ time for `false` assignments. That is, it takes (amortized) linear time for unit resolution and (amortized) constant time for unit subsumption.

The Davis-Putnam algorithm [2; 3] has long been a major practical method for solving SAT problems. The Davis-Putnam algorithm is based on unit propagation and case-splitting. Each case-splitting operation involves two unit propagations—one for the positive value of the variable split on and one for the negative value. For any moderate satisfiability problem, millions of unit propagations will be performed. Hence, the efficiency of unit propagation is crucial to the success of the Davis-

Putnam algorithm. Every bit of improvement on unit propagation is interesting. In fact, we have tested both Crawford and Auton's unit propagation algorithm and our new algorithm in the same implementation of the Davis-Putnam algorithm. Our algorithm has a speedup of two on average over Crawford and Auton's algorithm on various SAT problems.

2 The New Algorithm

There are two key ideas in the new algorithm:

- Delay testing for subsumption.
- Restrict resolution to the first and last active literals of a clause.

The first idea is to simply omit the unit-subsumption operation from Crawford and Auton's algorithm. The counter value is never `inactive`; it is always the number of unresolved literals in a clause that may have some subsumed literals as well as unassigned ones. Decrementing the counter value to one does not always signal the discovery of a new unit clause; the last unresolved variable may have already been assigned a subsuming value. To assign `true` to a variable v requires $O(|Q|)$ time for `true` assignments and $O(|P|)$ time for `false` assignments. The algorithm below incorporates this principle of delaying testing for subsumption in a different way.

The second idea follows from the observation that only resolution operations that reduce the clause to a unit or empty clause need to be recognized. To do this, it suffices to keep track of resolution operations with the first and last unresolved literals of a clause. A unit clause is found when the first and last unresolved literals finally coincide after a sequence of resolution operations.

No counters are associated with clauses. Instead of static lists of clauses that contain positive and negative occurrences of propositional variables, for each propositional variable there are four dynamically maintained lists of clauses such that the variable appears positively/negatively in the clause and all literals preceding/following it have been resolved.

The algorithm can be described informally as follows:

1. The literals in a clause are stored in consecutive cells and a clause is represented by two pointers which point to the first and the last literals of the clauses.
2. Associated with each propositional variable v, there are four lists of clauses:
   - `clauses_of_pos_head`(v): the clauses whose first literal is v.
   - `clauses_of_neg_head`(v): the clauses whose first literal is $\overline{v}$, the negation of v.
   - `clauses_of_pos_tail`(v): the clauses whose last literal is v.
   - `clauses_of_neg_tail`(v): the clauses whose last literal is $\overline{v}$.
3. When variable v becomes `true` (because v is in a unit clause), we ignore `clauses_of_pos_head`(v) and `clauses_of_pos_tail`(v), and check each clause in `clauses_of_neg_head`(v) and `clauses_of_neg_tail`(v). For a clause c in `clauses_of_neg_head`(v), we search for the first literal l in c that has no truth value and then add c into `clauses_of_pos_head`(l) if l is positive, or `clauses_of_neg_head`($\overline{l}$), otherwise. However, there are three exceptions:
   - if a literal whose value is `true` is found during the search process, c will not be added into any list since c was subsumed by a unit clause;
   - if every literal in c has value `false` then an empty clause has been found and that information is returned;
   - if l is the last literal of clause c, then a unit clause has been found and l is collected in a stack of "unit clauses".

   The handling of clauses in `clauses_of_neg_tail`(v) is analogous. The case when variable v is assigned `false` is analogous, too.

For the implementation, each propositional variable is represented by a unique identifier, which is a positive integer. The identifier of the negation of a variable is the negation of the identifier of that variable. For each variable v, `truth_value`(v) is one of `true, false` and `unknown`. We also extend the scope of `truth_value` to literals in the usual way.

A clause is represented by an array of identifiers of the literals in the clause. For clause c, `head_point`(c) and `tail_point`(c) are, respectively, the indices of the first and last literals of c. We also let `ith_literal`(c, i) return the literal of c indexed by i in the array, where `head_index`(c) $\leq i \leq$ `tail_index`(c).

Using the global variables OK, UNITS and CLAUSES,

- OK: boolean — false if an empty clause is found,
- UNITS: stack of literals — appearing in some unit clauses,
- CLAUSES: list of input clauses,

we can formally describe the `unit_propagation` algorithm by the following code:

```
proc propagate_true_value (v: variable)
    c: clause
    for c in clauses_of_neg_head(v) if OK do
            shorten_clause_from_head(c)
        end for
    for c in clauses_of_neg_tail(v) if OK do
            shorten_clause_from_tail(c)
        end for
```

```
end proc

proc shorten_clause_from_head(c: clause)
    /* this procedure implements unit resolution */
    i: integer
    for i from head_index(c) + 1 to tail_index(c) do
        L: literal := ith_literal(c, i)
        if (truth_value(L) = unknown)
            then if (tail_index(c) = i)
                    then stack_push(L, UNITS)
                    else insert_clause_head_list(c, L)
            else if (truth_value(L) = true)
            then return /* exit this function */
        end for
    /* all the literals in c are false. */
    OK := false
end proc

proc unit_propagation ()
    /* initiation */
    OK := true
    UNITS := empty
    for c: clause in CLAUSES
        if head_index(c) = tail_index(c) do
            L: literal := ith_literal(c,head_index(c))
            stack_push(L, UNITS)
        end for
    /* main loop */
    while (OK and not stack_empty?(UNITS)) do
        /* handle unit clauses */
        L: literal := stack_pop(UNITS)
        if truth_value(L) = true
            then continue
            else if truth_value(L) = false
            then OK := false
            else if positive(L)
            then { truth_value(L) := true
                    propagate_true_value(L) }
            else { truth_value(–L) := false
                    propagate_false_value(–L) }
        end while
    if (OK) then write("satisfiable if Horn")
        else write("unsatisfiable")
end proc
```

Remaining procedures are provided in the Appendix.

3 Analysis of the Algorithm

While it is easy to show the correctness of the algorithm, it is not straightforward to show its time complexity. In our algorithm, unit resolution and unit subsumption are done differently from conventional methods. At any moment in the execution of the algorithm, let C be a subset of CLAUSES, the input clauses, such that for each clause $c \in C$, both truth values of c's first literal (by `head_index(`c`)`) and last literal (by `tail_index(`c`)`) are unknown.

By assigning `true` to v, we have deleted from C in constant time the two sets, `clauses_of_pos_head(`v`)` and `clauses_of_pos_tail(`v`)`—this deletion corresponds to unit subsumption. However, it does not implement unit subsumption completely because: (a) if v appears in the middle of a clause, then that clause is not deleted. In other words, C contains trivial clauses—those subsumed by some unit clauses; (b) the clauses in `clauses_of_pos_head(`v`)` may be still accessible from their tail pointer by the algorithm, and similarly, `clauses_of_pos_tail(`v`)` may be still accessible from their head pointer.

By assigning `true` to v, we also have deleted from C clauses in `clauses_of_neg_head(`v`)` and `clauses_of_neg_tail(`v`)`. The procedure `propagate_true_value(`v`)` puts these clauses back into C by deleting $\overline{v}$ from these clauses—this corresponds to unit resolution. Again, this unit resolution is done partially since if v appears in the middle of a clause, then v is not deleted from the clause. However, it is guaranteed that no unit clauses will be missed in our algorithm because the first and last indices of a unit clause are identical and this is checked in `shorten_clause_from_head` and `shorten_clause_from_tail`.

For the time complexity of the algorithm, let m be the maximal length of clauses in CLAUSES and n_v be the number of clauses in CLAUSES containing v negatively, then the worst time complexity of `propagate_true_value(`v`)` is $O(m * n_v)$, because the worst time complexity of `shorten_clause_from_head` or `shorten_clause_from_tail` is $O(m)$. In the following, we show that the *amortized* time complexity of `propagate_true_value(`v`)` is actually $O(n_v)$. In other words, unit resolution is done in an amortized linear time while unit subsumption is done in amortized constant time.

It suffices to show that the amortized time complexity of `shorten_clause_from_head(`c`)` (the case of `shorten_clause_from_tail(`c`)` is analogous) is always $O(1)$ for every clause c, because it implies that the amortized complexity of `propagate_true_value(`v`)` is $O(n_v)$.

The actual cost of `shorten_clause_from_head(`c`)` is the number of literals that have value `false` in the beginning of c; `shorten_clause_from_head(`c`)` has to pass by these literals before finding an unassigned literal or a true literal. However, if literal $\overline{u}$ of c is passed by because u was assigned `true` earlier, then clause c is not in `clauses_of_neg_head(`u`)` at the time when u was assigned `true`. Hence, we can distribute the cost of passing $\overline{u}$ to that of assigning u to `true`. After this kind of distribution, the cost of `shorten_clause_from_head(`c`)` is constant. This distribution does not affect the complexity of any other procedures because each literal in the input clauses is visited at most once in the execution of `unit_propagation`.

In short, the cost of `propagate_true_value(`v`)` is

$O(n_v)$, which consists of two parts: (i) the cost of visiting each clause in which $\overline{v}$ is the first or the last literal at the time when v is assigned `true`; and (ii) the cost prepaid for passing $\overline{v}$ which are in the middle of other clauses when v is assigned `true`. Note that not every $\overline{v}$ in the input clause set has to be visited in the execution of `unit_propagation`, i.e., $O(n_v)$ is a generous upper bound.

The above time complexity analysis also applies when v is assigned `false`. That is, the amortized complexity of `propagate_false_value`(v) is $O(p_v)$, where p_v is the number of clauses in CLAUSES containing v positively. It is easy to see that the space complexity of the algorithm is linear to the input clauses. Our experimental results (provided in the full version of this paper) show that our algorithm works better than any other previous algorithms.

The basic idea of our new algorithm is to check a linear list from both of its ends. This idea can be also used in other algorithms. For instance, it can be used for general constraint satisfaction. A constraint satisfaction problem (CSP) consists of a set of variables; a domain of values for each variable; and a collection of constraints. A solution to a CSP is an assignment of values to all the variables such that no constraint is violated. For each variable, we may use one pointer pointing to the first available value and another pointer pointing to the last available value of the domain. To search an available value for a variable, we need only constant time instead of linear time (with respect to the size of the domain). The same idea can be also extended to handle tree structures instead of linear lists. That is, instead of two end pointers, we have a pointer for each leaf node of the tree.

References

[1] Crawford, J. M. and L. D. Auton. Experimental results on the crossover point in satisfiability problems. *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93),* 1993, 21–27.

[2] Davis, M. and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery 7,* 3 (July 1960), 201–215.

[3] Davis, M., G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery 5,* 7 (July 1962), 394–397.

[4] Dowling, W.F. and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming 3* (1984), 267–284.

Appendix: Some Additional Procedures

```
proc propagate_false_value (v: variable)
    /* analogous to propagate_true_value */
    c: clause
    for c in clauses_of_pos_head(v) if OK do
            shorten_clause_from_head(c)
        end for
    for c in clauses_of_pos_tail(v) if OK do
            shorten_clause_from_tail(c)
        end for
end proc

proc shorten_clause_from_tail(c: clause)
    /* analogous to shorten_clause_from_head */
    i: integer
    for i from tail_index(c)−1 to head_index(c) do
        L: literal := ith_literal(c, i)
        if (truth_value(L) = unknown)
            then if (head_index(c) = i)
                    then stack_push(L, UNITS)
                    else insert_clause_tail_list(c, L)
            else if (truth_value(L) = true)
            then return /* exit this function */
        end for
    /* all the rest literals in c are false. */
    OK := false
end proc

proc insert_clause_head_list(c: clause, L: literal)
    /* insert clause c into the head list of variable |L| */
    if positive(L)
        then list_insert(c, clauses_of_pos_head(L))
        else List_insert(c, cLauses_of_neg_head(−L))
end proc

proc insert_clause_tail_list(c: clause, L: literal)
    /* insert clause c into the tail list of variable |L| */
    if positive(L)
        then list_insert(c, clauses_of_pos_tail(L))
        else list_insert(c, clauses_of_neg_tail(−L))
end proc
```