Implementing the Davis-Putnam Algorithm by Tries

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, California 94025
stickel@ai.sri.com

Abstract

The Davis-Putnam method is one of the major practical methods for the satisfiability (SAT) problem of propositional logic. We show how to implement the Davis-Putnam method efficiently using the trie data structure for propositional clauses by presenting seven implementations of the method. We propose a new technique for implementing unit propagation whose complexity is sublinear to the number of occurrences of the variable in the input. We present the performance of our programs on some quasigroup problems. The efficiency of our programs allowed us to solve some open quasigroup problems.

1 Introduction

In recent years, there has been considerable renewed interest in the satisfiability (SAT) problem of propositional logic. The SAT problem is known to be difficult to solve—it is the first known NP-complete problem. Because the SAT problem is fundamental to many practical problems in mathematics, computer science, and electrical engineering, efficient methods that can solve a large subset of SAT problems are eagerly sought. Empirical research has been very fruitful for the development of efficient methods for SAT problems.

The Davis-Putnam method [3, 4] has long been a major practical method for solving SAT problems. It is based on unit propagation (i.e., unit resolution and unit subsumption) and case-splitting. It is known that many factors affect the performance of the method: the data structure for clauses, the choice of variable for splitting, etc. In this paper, we will concentrate on the use of tries (discrimination trees) for the Davis-Putnam method. In [5], de Kleer used tries to represent propositional clauses for efficient subsumption.

In the past, both of us have used tries to represent first-order terms and to implement efficient rewriting-based theorem provers. In Autumn 1992, we independently started using tries in the Davis-Putnam method. By using the trie data structure, our programs gain something in efficiency, and much in elegance. Some preliminary results of our experiments are presented in [14] and [15]. In this paper, we present in detail the data structures used in our programs.

One of the major motivations for developing our programs was to solve some open quasigroup problems in algebra [1]. The usefulness of computer programs to attack these

*Partially supported by the National Science Foundation under Grants CCR-9202838 and CCR-9357851.
†Research supported by the National Science Foundation under Grant CCR-8922330.

quasigroup problems has been demonstrated in [16, 7, 14]. We think these quasigroup problems are much better benchmarks than randomly generated SAT problems for testing constraint solving methods: the problems have fixed solutions; descriptions of the problems are simple and easy to communicate; most importantly, some cases of the problems remain open, offering challenge and opportunities for friendly competition as well as contributions to mathematical knowledge. Besides having large search spaces, quasigroup problems are demanding examples for the Davis-Putnam method because their propositional representations contain n^3 variables and (depending on the problem) from $O(n^4)$ to $O(n^6)$ clauses, so large sets of clauses with hundreds of thousands of literals must be handled.

We have not tested any randomly generated SAT problems in our experiments. Recently, the location of hard SAT problems has been identified by empirical research [11, 2]. However, different methods are still hard to compare based on their performance on randomly generated problems because some hard problems may occur in an area where most problems are easy and vice versa. Recently, some incomplete methods based on local search have been proposed which can solve very large size SAT problems [8, 12]. The usefulness of these methods for solving quasigroup problems remains to be seen. However, these methods cannot entirely replace the Davis-Putnam method because many quasigroup problems are known to have no solutions and incomplete methods cannot prove that no solution exists or count the number of solutions.

1.1 The Davis-Putnam Method

The Davis-Putnam method is based on three simple facts about truth table logic. Firstly, where A and B are any formulae, the conjunction $A \wedge (\overline{A} \vee B)$ is equivalent to $A \wedge B$ and the conjunction $A \wedge (A \vee B)$ is equivalent to A. It follows that the application of unit resolution and subsumption to any set of propositional clauses results in an equivalent set. Secondly, where X is any set of formulae and A any propositional formula, X has a model iff either $X \cup \{A\}$ has a model or $X \cup \{\overline{A}\}$ has a model. Thirdly, where X is any set of propositional clauses and A any propositional atomic formula, if A does not occur at least once positively in [some clause in] X and at least once negatively, then the result of deleting from X all clauses in which A occurs is a set which has a model iff X has a model.

A simple algorithm based on the first two of these facts[1] is shown in Figure 1. That it is sound and complete for propositional clause problems is well known. Naturally, one important place at which heuristics may be inserted is in the choice of a literal for splitting. In this paper, all of our programs simply choose the first literal in one of the shortest positive clauses. We can see potential virtue in using a more elaborate selection heuristic—for instance, giving some weight to the number of constraints in which a literal is involved—but that is not the focus of this paper. Our focus is on how to represent propositional clauses and implement unit propagation efficiently.

[1] Eliminating "pure" variables that occur only positively or only negatively is not necessary for completeness. Moreover, in many types of problems, such as the quasigroup problems that we are especially interested in, the condition never occurs.

Figure 1: A Simple Davis-Putnam Algorithm

```
function Satisfiable ( clause set S ) return boolean
    /* unit propagation */
    repeat
        for each unit clause L in S do
            delete from S every clause containing L
            delete ¬L from every clause of S in which it occurs
        od
        if S is empty then
            return TRUE
        else if the null clause is in S then
            return FALSE
        fi
    until no further changes result
    /* splitting */
    choose a literal L occurring in S
    if Satisfiable ( S ∪ {L} ) then
        return TRUE
    else if Satisfiable ( S ∪ {¬L} ) then
        return TRUE
    else
        return FALSE
    fi
end function
```

1.2 Trie Data Structure for Propositional Clauses

Our programs gain something in efficiency, and much in elegance, from using the *trie* data structure, first used to represent sets of propositional clauses in [5].

We assume that each propositional variable has a unique index, which is a positive integer. The index of the negation of a variable is the negation of the index of that variable. A clause is represented by the list of indices of the literals in the clause.

Conceptually, the trie data structure for propositional clauses is very simple. It is a tree all of whose edges are marked by indices of literals and whose leaves are marked by a clause mark. A clause is represented in a trie as a path from the root to a leaf such that the edges of the path are marked by the literal indices of the clause. To save space, if two clauses (represented as lists of integers) have the same prefix of length n, then they should share a path of length n in the trie.

If all the nodes that have an edge of the same mark to the same parent node in a trie are made into a linear list, we may use a 3-ary tree to represent a trie as follows: Each node of the tree is either empty (*nil*), or a clause end-mark ($\Box$), or a 4-tuple $\langle var, pos, neg, rest \rangle$, where *var* is a variable index, *pos* is its positive child node, *neg* is its negative child node, and *rest* is its brother node. The interpretation is that the edge from this node to *pos* is marked by *var*; the edge from this node to *neg* is marked by $(-var)$; and *rest* is the next

Figure 2: The *trie-merge* procedure.

function *trie-merge* (trie t_1, trie t_2) **return** trie
    if $t_1 = \Box$ or $t_2 = \Box$ **then**
        return $\Box$
    else if $t_1 = nil$ **then**
        return t_2
    else if $t_2 = nil$ **then**
        return t_1
    fi
    let $t_1 = \langle v_1, p_1, n_1, r_1 \rangle$
    let $t_2 = \langle v_2, p_2, n_2, r_2 \rangle$
    if $v_1 = v_2$ **then**
        return $\langle v_1, \textit{trie-merge}(p_1, p_2), \textit{trie-merge}(n_1, n_2), \textit{trie-merge}(r_1, r_2) \rangle$
    else if $v_1 < v_2$ **then**
        return $\langle v_1, p_1, n_1, \textit{trie-merge}(r_1, t_2) \rangle$
    else
        return $\langle v_2, p_2, n_2, \textit{trie-merge}(r_2, t_1) \rangle$
    fi
end function

node in the linear list of the nodes that share the same parent node in the trie.

A set S of (nontautologous) propositional clauses can be easily represented by a trie T_S as follows: If S is empty, then $T_S = nil$; if S contains a null clause, then $T_S = \Box$; otherwise, choose any variable index v and divide S into three groups:

- $P = \{v \vee P_1,\ ...,\ v \vee P_n\}$—the clauses that contain v positively.
- $Q = \{\overline{v} \vee Q_1,\ ...,\ \overline{v} \vee Q_m\}$—the clauses that contain v negatively.
- $R = \{R_1,\ ...,\ R_l\}$—the clauses that do not contain v.

Let

- $P' = \{P_1, \ldots, P_n\}$—$P$ with occurrences of v removed.
- $Q' = \{Q_1, \ldots, Q_m\}$—$Q$ with occurrences of $\overline{v}$ removed.

Let $T_{P'}$, $T_{Q'}$, and T_R be the trie nodes recursively representing P', Q', and R, respectively. Then S can be represented by $T_S = \langle v, T_{P'}, T_{Q'}, T_R \rangle$. For example, if $S = \{x_1 \vee x_2, \overline{x}_1 \vee \overline{x}_2\}$, the indices of x_1 and x_2 are 1 and 2, respectively, then $T_S = \langle 1, \langle 2, \Box, nil, nil \rangle, \langle 2, nil, \Box, nil \rangle, nil \rangle$.

A trie is said to be *ordered* if for any node $\langle var, pos, neg, rest \rangle$, var is smaller than any variable index appearing in pos, neg and $rest$. The *trie-merge* operation, an extension of the merge-sort algorithm that merges two ordered tries into a single one, is shown in Figure 2. The insertion of one clause c into a trie T can be done using *trie-merge*, if we first create a trie T_c for c and then merge T_c and T.

If the value of var is true, $\langle var, pos, neg, rest \rangle$ is equivalent to *trie-merge*$(neg, rest)$. Similarly, $\langle var, pos, neg, rest \rangle$ is equivalent to *trie-merge*$(pos, rest)$ when the value of var

is false. The nodes $\langle var, pos, neg, \Box\rangle$ and $\langle var, \Box, \Box, rest\rangle$ are equivalent to $\Box$. The node $\langle var, nil, nil, rest\rangle$ is equivalent to $rest$. To save space, we also replace nodes by their simpler equivalents when possible in the trie-merge operation.

The major operation in the Satisfiable procedure, *unit propagation*, can be easily implemented on tries: when a variable var is forced to be true by a unit clause, we simply replace each node $\langle var, pos, neg, rest\rangle$ in the trie by *trie-merge*$(neg, rest)$; the case when var is forced to be false is handled similarly.

The trie representation of a set of propositional clauses has several advantages for the Davis-Putnam method:

- Duplicate clauses are automatically eliminated when the trie is constructed.
- The trie representation reduces memory requirements.
- Unit clauses can be found quickly in a trie.
- The *unit propagation* operation can be computed relatively efficiently. Because the subtrie $\langle var, pos, neg, rest\rangle$ does not contain any variable $var' < var$, it does not need to be searched or altered when assigning a value to var'. Multiple *unit propagation* operations can be done in a single traversal of the trie.

2 Quasigroup Problems

The quasigroup problems are given by Fujita, Slaney, and Bennett in their award-winning IJCAI paper [7]. Roughly speaking, the problems concern the existence of $v \times v$ Latin squares—each row and each column of a Latin square is a permutation of 0, 1, ..., $v-1$—with certain constraints. Given $0 \le i, j < v$, let $i * j$ denote the entry at the ith row and jth column of a square.

The following clauses specify a $v \times v$ Latin square: for all elements $x, y, u, w \in S = \{0, ..., (v-1)\}$,

$$x * u = y, x * w = y \Rightarrow u = w \quad : \text{the left-cancellation law} \tag{1}$$
$$u * x = y, w * x = y \Rightarrow u = w \quad : \text{the right-cancellation law} \tag{2}$$
$$x * y = u, x * y = w \Rightarrow u = w \quad : \text{the unique-image property} \tag{3}$$
$$(x * y = 0) \vee \cdots \vee (x * y = (v-1)) \quad : \text{the (right) closure property} \tag{4}$$

It was shown in [14] that the following two clauses are valid consequences of the above clauses and adding them into a prover reduces the search space.

$$(x * 0 = y) \vee \cdots \vee (x * (v-1) = y) \quad : \text{the middle closure property} \tag{5}$$
$$(0 * x = y) \vee \cdots \vee ((v-1) * x = y) \quad : \text{the left closure property} \tag{6}$$

For any x, y, z in $\{0, ..., (v-1)\}$, the following constraints are given in [7][2]:

[2]The QG7 constraint is the one used in [14], not [7].

Name	Constraint
QG1	$x*y=u, z*w=u, v*y=x, v*w=z \Rightarrow x=z, y=w$
QG2	$x*y=u, z*w=u, y*v=x, w*v=z \Rightarrow x=z, y=w$
QG3	$(x*y)*(y*x)=x$
QG4	$(x*y)*(y*x)=y$
QG5	$((x*y)*x)*x=y$
QG6	$(x*y)*y=x*(x*y)$
QG7	$((x*y)*x)*y=x$

In the following, problem QG$i.v$ denotes the problem represented by clauses (1)–(6) plus QGi for $S=\{0,...,(v-1)\}$. In addition, clauses for the idempotency law, $x*x=x$, and the constraint $x*(v-1)\geq x-1$, which eliminates some isomorphic models, are used for each problem here. Further details on these problems can be found in [7] and [14].

Propositional clauses are obtained by simply instantiating the variables in clauses (1)–(6) by values in S and replacing each equality $x*y=z$ by a propositional variable $p_{x,y,z}$. The number of the propositional clauses is determined by the order of the quasigroup (i.e., v) and the number of distinct variables in a clause. Constraints QG1 and QG2 can be handled in the same way. For QG3–QG7, we have to transform them into "flat" form. For example, the flat form of QG5 is

$$(x*y=z),(z*x=w)\Rightarrow(w*x=y).$$

It can be shown that the two "transposes" of the above clause are also valid consequences of QG5:

$$(w*x=y),(x*y=z)\Rightarrow(z*x=w),$$
$$(z*x=w),(w*x=y)\Rightarrow(x*y=z).$$

It has been confirmed by experiments that adding these "transposes" to the input can reduce the search space. This is also true for QG3–QG7.

3 DDPP: A Straightforward Trie-based Implementation

The program DDPP (Discrimination-tree-based Davis-Putnam Prover) is a straightforward implementation of the Davis-Putnam method based on the *trie-merge* operation on tries. DDPP is written in Lucid Common Lisp. A detailed description of DDPP can be found in [10]. Several open problems about quasigroups were first solved by this simple program [14]. They are QG5.13, QG5.14 and QG5.15 (negatively), and QG4.12 (positively). Performance of DDPP on some quasigroup problems is shown in Figure 4. The number of branches is one plus the number of splittings in the Davis-Putnam method. The times were collected in Lucid Common Lisp on a 40MHz Sun SPARCserver 670 with 128MB memory.

Because the *trie-merge* operation needs to create new trie nodes and the *splitting* operation in the Davis-Putnam method uses a trie twice, it is expensive to implement *trie-merge* destructively (i.e., modifying the fields of the existing trie nodes). Instead, DDPP does the *trie-merge operation* nondestructively and the result of the *trie-merge* operation shares (nearly) maximal structure with the original tries to minimize the memory allocation.

4 LDPP: An Efficient Non-Trie-Based Implementation

DDPP was hindered by the speed of the crucial *unit propagation* operation. Although the trie representation eliminated searching subtries $\langle var, pos, neg, rest \rangle$ for variables $var' < var$, extensive searching was still necessary. Moreover, the nondestructive *trie-merge* required some storage allocation, which is relatively slow.

LDPP (Linear-list-based Davis-Putnam Prover) is an efficient implementation of the Davis-Putnam method that does not use a trie representation. Like DDPP, LDPP is written in Common Lisp. Experimental results for DDPP and LDPP are given in Figure 4, where identical sets of clauses were input to each prover. The disparity in number of branches in the search space is due to different orderings of clauses and literals that result in different literals being chosen for *splitting* operations. Results in later tables for SATO also use the same sets of clauses. In spite of the disparity in number of branches, it is meaningful to compare the search rates (i.e., branches per second) for DDPP, LDPP, and SATO.

LDPP is generally much faster than DDPP. LDPP performs unit resolution and unit subsumption operations very quickly by decrementing and setting fields. Resolvable or subsumable clauses need not be searched for since each variable contains pointers to all the clauses that contain the variable. Other recent implementations of the Davis-Putnam method that employ a similar approach include Crawford and Auton's TABLEAU [2], Letz's SEMPROP, and McCune's ANL-DP [9]. LDPP′ is a faster variant of LDPP that explores exactly the same search space as LDPP but does not perform the subsumption operation (see Section 6.2).

In LDPP, a set of clauses is represented by a list of clauses and a list of variables. Each clause contains the fields:

- `positive-literals`, `negative-literals`: List of pointers to variables occurring positively (resp. negatively) in this clause.
- `inactivated`: This is `nil` if the clause is still active (i.e., has not been subsumed). If the clause has been subsumed, this field contains a pointer to the variable whose assignment subsumed this clause.
- `number-of-active-positive-literals`, `number-of-active-negative-literals`: When `inactivated` is `nil`, this is the number of variables in `positive-literals` (resp. `negativeliterals` that have not been assigned a value (i.e., that have not been resolved away from this clause).

Each variable contains the fields:

- `value`: This is `true` if the variable has been assigned the value true, `false` if it has been assigned false, and `nil` if no value has been assigned.
- `contained-positively-clauses`, `contained-negatively-clauses`: List of pointers to clauses that contain this variable positively (resp. negatively).

To assign true to a variable:

- Its `value` field is set to `true`.

- Every clause in `contained-positively-clauses` has its `inactivated` field set to the variable, unless `inactivated` was already non-`nil`.
- Every clause in `contained-negatively-clauses` has its `number-of-active-negative-literals` field decremented by one, unless `inactivated` was already non-`nil`. Note that we *don't* modify `negative-literals` itself. If the sum of `number-of-active-negative-literals` and `number-of-active-positive-literals` reaches zero, the current truth assignment yields the unsatisfiable empty clause. If the sum reaches one, a new unit clause has been produced. The newly derived unit clause can be identified by finding the only atom in `positive-literals` or `negative-literals` whose value is `nil`. These are queued and assigned values before *unit propagation* finishes.

To undo an assignment of true to a variable and thus restore the set of clauses to their state before the assignment so that alternative assignments can be tested:

- The `value` field for the variable is set to `nil`.
- Every clause in `contained-positively-clauses` has its `inactivated` field set to `nil`, provided `inactivated` had this variable as its value.
- Every clause in `contained-negatively-clauses` has its `number-of-active-negative-literals` field incremented by one, provided `inactivated` is `nil`.

Assignment of false to a variable is done analogously. The set of clauses after a sequence of assignments is represented by those clauses in the list whose `inactivated` field is still `nil`. The literals still present in these clauses are just those in `positive-literals` and `negative-literals` whose `value` is still `nil`.

5 SATO: Another Trie-based Implementation

The high cost of the *trie-merge* operation in DDPP, which motivated the abandonment of the trie representation in LDPP, does not imply that the trie data structure is ineffective for the Davis-Putnam method. Actually, the implementations of the Davis-Putnam method in the SATO program (SAtisfiability Testing Optimized) [15] did not use the *trie-merge* operation. In this section, we describe some ideas used in SATO to improve the performance of the Davis-Putnam method.

We used SATO to settle several open cases of quasigroup problems, including QG5.14 (without the idempotency law), QG6.15, QG7.15 (negatively), QG2.14, QG2.15 and QG7.16 (positively). Some of these problems required several weeks of CPU-time on a powerful workstation. The efficiency of SATO was indispensable for our success.

5.1 A New Algorithm for Unit Propagation

SATO possesses features of DDPP (trie representation) and LDPP (destructive operations and lists of pointers to atom occurrences to eliminate search). The major idea used in SATO is to keep two lists of literals: The *head list* is a collection of the occurrences of the first literal of each clause and the *tail list* is a collection of the occurrences of the last literal of each clause. If the first literal of a clause becomes true, that literal is simply removed

from the head list. If that literal becomes false, it is removed from the head list and we search for the next unassigned literal in the clause and add it to the head list unless: (a) if a literal with value true is found during the search process, no literal will be added to the head list since the clause was subsumed by a previous assignment; (b) if every literal in the clause has value false then a null clause has been found and that information is returned; (c) if the next unassigned literal of the clause is also in the tail list, then a unit clause has been found and that literal is collected in a list of "unit clauses". The handling of literals in the tail list is analogous.

The above idea implies that a clause should be represented as a double-linked list. When the trie data structure is used, a literal is represented by a trie node together with an edge from a node to its parent. That is, instead of the data structure $\langle var, pos, neg, rest \rangle$, we use $\langle var, pos, neg, rest, parent \rangle$, where *parent* is the parent of the current node.

For efficiency, both the head list and tail list are grouped according to the variable index of each node. That is, we use a variable table in which we not only record the value of each variable (true, false, or unknown), but also a head list and a tail list of nodes whose label is that variable. Initially, each head list contains at most one node. Whenever a variable's value goes from unknown to true (resp. false), we remove each node in the head list of this variable and try to add the negative (resp. positive) child node—together with its brothers—into the head list. We also remove each node in the tail list of this variable; if its negative (resp. positive) child is equivalent to $\Box$, we try to add its parent node into the tail list.

The first version of SATO does not use the tail list and thus does not need the parent link in the trie data structure. While this implementation does not need dynamic memory allocation, our experiments indicate that more than half of the total search time is spent on deciding whether a trie is equivalent to $\Box$— this operation is necessary to locate unit clauses and to select literals for splitting. This is because not every node with an interpreted variable is removed from the trie. For example, the clause $x \vee y$ is represented by $T_c = \langle x, \langle y, \Box, nil, nil \rangle, nil \rangle$, which is initially stored in the head list of variable x. Now suppose y has value false; then in our implementation, $\langle y, \Box, nil, nil \rangle$ will not be replaced by $\Box$. To decide that x is in a unit clause, we have to search the whole T_c.

SATO is written in C and the times were collected on a 40Mhz Sun SPARCstation 2 with 32MB memory. In Figure 5, results for two versions of SATO are given—SATO1 uses only the head list and SATO2 uses both the head and tail lists. The experimental results indicate that SATO2 is substantially faster than SATO1 on quasigroup problems.

5.2 Complexity Analysis

In the following, we show that the technique implemented in SATO2 for unit propagation is better, both theoretically and practically, than the method of LDPP.

The Davis-Putnam method as given in Figure 1 consists of two major operations: *unit propagation* and *splitting*. The *unit propagation* consists of a sequence of *unit propagation* operations. For any moderate satisfiability problem, thousands of the *splitting* operations will be performed, and each *splitting* will invoke *unit propagation*. Other things being equal, the complexity of the *Satisfiable* procedure depends on that of *unit propagation*. In the following, we concentrate on the complexity of *unit propagation*.

Given a set S of input clauses and any variable v, let P_v be the number of clauses of S in which v appears positively, and let N_v be the number of clauses in which v appears

negatively. It is easy to see that for LDPP, the complexity of the *unit propagation* operation is $O(P_v + N_v)$ when v receives a value, either true or false.

We show below that the *unit propagation* operation in SATO takes an amortized time of $O(N_v)$ when v is assigned to true and $O(P_v)$ when v is assigned to false. To facilitate the understanding, we may assume that each clause is represented by a double-linked list, even though the trie data structure gives better results. We also assume that a literal cannot be in both the head list and the tail list: if this is the case, we remove it from both lists and add it to the unit-clause list.

Suppose x is assigned true and the number of $\overline{x}$ literals in the head list is HN_x. We need to perform HN_x operations to add those literals that follow $\overline{x}$ in each of HN_x clauses to the head list. Recall that in our implementation, not every node with an interpreted variable is removed from the trie. Because of this, when the first literal (i.e., $\overline{x}$) of a clause becomes false, adding the next (unassigned) literal of the clause to the head list does not always take constant time: If the next literal has value false, we have to pass by this literal and so on, until we find a literal whose value is true or unassigned, or until no literal is left in the current clause.

That is, if k literals are passed by in the adding process, the complexity of adding the next literal to the head list will be $O(k)$ and the worst complexity would be $O(k * HN_x)$. However, if $\overline{y}$ is passed by because y was assigned true earlier, then this $\overline{y}$ is not in the head list at the time when y was assigned true. Hence, we can distribute the cost of passing $\overline{y}$ to that of assigning y to true (i.e., $O(N_y)$). After this kind of distribution, the cost of adding the next literal following each $\overline{x}$ in the head list is constant.

The case when $\overline{x}$ appears in the tail list is handled similarly. In short, the cost of assigning x to true, $O(N_x)$, consists of two parts: the cost of visiting each $\overline{x}$ literal in the head and tail lists and the cost prepaid for passing $\overline{x}$ literals neither in the head list nor in the tail list. Note that not every $\overline{x}$ in the clause set has to be visited when assigning x to true, i.e., $O(N_x)$ is a generous upper bound.

The above complexity analysis also applies when x is assigned false. When the set of clauses is represented by a trie, the number of $\overline{x}$ literals in the head list, HN_x, in the above analysis can be replaced by the number of the corresponding trie nodes (which is usually smaller than HN_v). However, we cannot say that the amortized complexity of assigning x to true is bounded by the number of the trie nodes representing $\overline{x}$ because, when assigning x to true, a trie node representing $\overline{x}$ may be visited more than once while an occurrence of $\overline{x}$ is visited at most once when clauses are represented by double-linked lists.

6 Comparison of LDPP and SATO

While the theoretical analysis shows that SATO's method has an advantage over LDPP's, it also appears to perform better in practice (see Figures 4 and 5). Because it is difficult to make accurate comparisons across different implementations in different languages, for purposes of comparison, we also carefully implemented LDPP's algorithm, which is similar to Crawford and Auton's method, in SATO. We discuss below the two key ideas that can be borrowed from SATO to improve on LDPP: (a) using a trie for clauses; and (b) avoiding using unit subsumption.

6.1 Using Trie for Clauses

Using tries can automatically remove some subsumed clauses (including duplicates). LDPP and Crawford and Auton's method can take advantage of this. To test this idea, we implemented two versions of LDPP's algorithm in SATO: in SATO1.2, each clause is represented by a single linked list of integers; in SATO1.3, the trie data structure is used and each clause is represented by a path from a leaf to the root of a trie. Figure 6 lists the results for SATO1.2 and SATO1.3. In terms of branches per second, SATO1.3 is always better than SATO1.2 because the discrimination tree method automatically eliminates duplicate clauses. Actually, it automatically deletes a class of subsumed clauses, i.e., those clauses one of whose prefixes (regarding a clause as a list) is also a clause in the system.

It might appear that creating a trie for a set of clauses would be appreciably more expensive than creating lists of lists of literals. In fact, both operations have the same theoretical complexity. In practice, creating a list is slightly faster than creating a trie, as indicated by the creation times in Figure 6. The creation times for SATO1 and SATO2 are the same as those of SATO1.3.

6.2 Avoiding Unit Subsumption

The Davis-Putnam method performs unit resolution and unit subsumption operations. These are done in LDPP by decrementing literal counts for unit resolutions and setting an inactivated flag for unit subsumptions. Every assignment to a variable requires examining and possibly modifying every clause that contain the variable. A key issue in SATO is that only unit resolutions are performed, so only occurrences with one polarity or the other are examined.

This idea can be applied to LDPP by eliminating use of the `inactivated` field. There are some extra costs: counts for resolved literals are decremented for subsumed clauses as well as unsubsumed ones, derived units might already be assigned a (subsuming) value and must be ignored, and subsumed clauses must be ignored by the process for selecting literals to split on. Despite these extra costs, there is substantial benefit to omitting the subsumption operation. LDPP′ is LDPP modified to eliminate the subsumption operations. As can be seen from Figure 4, it is appreciably faster than LDPP on quasigroup problems.

7 Conclusions

In this paper, we have concentrated on the use of the trie data structure for implementing the Davis-Putnam method. Seven implementations of the Davis-Putnam method with their performance results on some quasigroup problems are presented: DDPP, LDPP, LDPP′, SATO1, SATO2, SATO1.2 and SATO1.3. Figure 3 summarizes the characteristics of the different programs described here.

We conclude that:

- The trie representation for clause sets is more efficient than the list representation (e.g., SATO1.3 vs. SATO1.2).
- Using lists of pointers to occurrences of atoms in clauses and reversible, destructive updating is faster than using the DDPP's nondestructive trie-merge operation (e.g.,

Figure 3: Summary of Program Characteristics

Program	Representation	How Resolvable Literals Are Found	Unit Subsumption
DDPP	trie	search trie; uses trie-merge operation instead of destructive operations like the other programs	yes
SATO1.3	trie	atom points to clauses that contain it	yes
SATO1.2 & LDPP	list	atom points to clauses that contain it	yes
LDPP′	list	atom points to clauses that contain it	no
SATO1	trie	atom points to clauses whose first active literal contains it	no
SATO2	trie	atom points to clauses whose first or last active literal contains it	no

SATO1.3 vs. DDPP).

- Eliminating the unit subsumption operation can improve performance (e.g., LDPP′ vs. LDPP).
- Restricting the unit resolution operation to operate only on clauses whose first or last literal contains the atom can improve performance (no single factor comparison was made, but see SATO2 vs. LDPP′ and SATO1.3; SATO2 vs. SATO1 shows the benefit of considering first and last literals instead of just first literals).

We have proposed a new method for efficiently implementing *unit propagation* in the Davis-Putnam method. We showed that this new method, used in SATO2, is better, both theoretically and practically, than the approach used in LDPP and many other systems. That approach appeared earlier in Dowling and Gallier's linear algorithm for satisfiability of Horn clauses [6]. We think our ideas can be used to design a new sublinear algorithm for the satisfiability of Horn clauses.

Many aspects of the Davis-Putnam method are not addressed in this paper. All of our programs simply choose the first literal in one of the shortest positive clauses for splitting in the Davis-Putnam method. How to efficiently implement other selection heuristics using the trie data structure remains a research problem. Our programs do not perform subsumption checking; it would be interesting to see how de Kleer's subsumption algorithm on tries could be integrated into the Davis-Putnam method. We did little or no checking

for pure literals, and did not perform intelligent backtracking or check for symmetries.[3] Further research is needed to see how such operations can be done efficiently using the trie data structure.

References

[1] Bennett, F. E. and L. Zhu. Conjugate-orthogonal Latin squares and related structures. In J. H. Dinitz and D. R. Stinson (eds.). *Contemporary Design Theory: A Collection of Surveys.* Wiley, New York, 1992.

[2] 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.

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

[4] 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.

[5] de Kleer, J. An improved incremental algorithm for generating prime implicates. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92),* San Jose, California, July 1992, 780–785.

[6] 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.

[7] Fujita, M., J. Slaney and F. Bennett. Automatic generation of some results in finite algebra. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1993.

[8] Gu, J. Local search for satisfiability (SAT) problem. *IEEE Transactions on Systems, Man, and Cybernetics 23,* 4 (1993), 1108–1129.

[9] McCune, W. A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problems. Draft manuscript, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, May 1994.

[10] Meglicki, G. Stickel's Davis-Putnam engineered reversely. Available by anonymous FTP from arp.anu.edu.au, Automated Reasoning Project, Australian National University, Canberra, Australia, 1993.

[11] Mitchell, D., B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92),* San Jose, California, July 1992, 459–465.

[12] Selman, B., H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92),* San Jose, California, July 1992, 440–446.

[3] LDPP′ optionally uses intelligent backtracking and some versions of SATO can identify symmetries in a preprocessing step.

[13] Slaney, J. FINDER, Finite Domain Enumerator: version 2.0 notes and guide. Technical Report TR-ARP-1/92, Automated Reasoning Project, Australian National University, Canberra, Australia, 1992.

[14] Slaney, J., M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: quasigroup existence problems. To appear in *Computers and Mathematics with Applications.*

[15] Zhang, H. SATO: a decision procedure for propositional logic. *Association for Automated Reasoning Newsletter,* No. 22 (March 1993), 1–3.

[16] Zhang, J. Search for idempotent models of quasigroup identities. Typescript, Institute of Software, Academia Sinica, Beijing.

Figure 4: Quasigroup Problems: DDPP and LDPP

Problems	Models	DDPP		LDPP		LDPP′
		Branches	Search (sec)	Branches	Search (sec)	Search (sec)
QG1.7	8	353	27	389	34	26
.8	16	97521	9547	101129	15191	3463
QG2.7	14	217	19	205	20	8
.8	2	53410	5275	33835	5591	1358
QG3.8	18	542	35	573	8	5
.9	–	26847	2415	24763	445	208
QG4.8	–	564	36	602	7	4
.9	194	22491	2188	27479	448	228
QG5.9	–	15	8	15	.9	.4
.10	–	50	31	38	2	.9
.11	5	136	147	125	10	5
.12	–	443	660	369	39	15
.13	–	16438	27559	12686	1466	639
QG6.9	4	17	5	18	.8	.4
.10	–	65	23	59	2	.8
.11	–	451	238	539	30	11
.12	–	5938	4997	7288	501	177
QG7.9	4	9	4	8	.6	.3
.10	–	40	17	40	2	.7
.11	–	321	220	294	17	6
.12	–	2083	2047	1592	110	38
.13	64	61612	86870	34726	2780	1050

Figure 5: Quasigroup Problems: SATO1 and SATO2

Problems	Models	Clauses	SATO1		SATO2	
			Branches	Search (sec)	Branches	Search (sec)
QG1.7	8	11131	413	5	384	1.2
.8	16	28781	136778	4328	102626	387.6
QG2.7	14	11131	598	6	282	1.0
.8	2	28781	153755	3698	37639	187.4
QG3.8	18	17885	514	5	611	2.4
.9	–	28711	20557	321	25300	129.3
QG4.8	–	17885	668	7	557	2.3
.9	178	28711	38733	653	32129	168.3
QG5.9	–	28711	19	1	15	0.2
.10	–	43846	61	3	38	0.6
.11	5	64307	113	11	116	2.3
.12	–	91219	476	69	369	6.9
.13	–	125815	17424	3179	10764	228.4
QG6.9	4	28711	25	1	16	0.3
.10	–	43846	147	6	59	0.7
.11	–	64307	583	47	519	6.5
.12	–	91219	7161	953	5728	94.7
QG7.9	4	28711	9	.4	9	0.2
.10	–	43846	115	5	93	0.4
.11	–	64307	312	30	254	3.2
.12	–	91219	1643	258	1281	22.4
.13	64	125815	27111	6039	27988	592.5

Figure 6: Quasigroup Problems: SATO1.2 and SATO1.3

Problems	Models	SATO1.2			SATO1.3		
		Branches	Create (sec)	Search (sec)	Branches	Create (sec)	Search (sec)
QG1.7	8	461	1	16	376	1	4
.8	16	97521	3	7524	102610	3	1895
QG2.7	14	759	1	22	340	1	4
.8	2	130690	3	8911	80245	3	1724
QG3.8	18	488	0	3	1072	.1	4
.9	–	18474	.2	103	48545	.3	286
QG4.8	–	522	0	3	925	.1	4
.9	178	35801	.1	201	52826	.2	316
QG5.9	–	18	.1	0	19	.2	.2
.10	–	66	.2	1	62	.3	1
.11	5	117	1	5	102	1	4
.12	–	398	1	17	383	2	15
.13	–	13704	2	794	10764	3	604
QG6.9	4	22	.1	.3	24	.2	.2
.10	–	144	.2	1	150	.4	1
.11	–	533	1	16	519	1	6
.12	–	6839	1	298	5728	1	239
QG7.9	4	9	.1	.3	7	.2	.2
.10	–	75	.3	1	54	.4	1
.11	–	295	1	10	254	1	8
.12	–	1749	1	80	1281	2	56
.13	64	27206	2	1696	27989	2	1587