Implementing the Davis–Putnam Method

Hantao Zhang *
Dept. of Computer Science, The University of Iowa, Iowa City, IA 52242, U.S.A.
email: hzhang@cs.uiowa.edu

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

Received: 30 August 1999

Abstract. The method proposed by Davis, Putnam, Logemann, and Loveland for propositional reasoning, often referred to as 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. A new technique of indexing only the first and last literals of clauses yields a unit propagation procedure whose complexity is sublinear to the number of occurrences of the variable in the input. We also show that the Davis–Putnam method can work better when unit subsumption is not used. We illustrate the performance of our programs on some quasigroup problems. The efficiency of our programs has enabled us to solve some open quasigroup problems.

Keywords: propositional satisfiability, Davis–Putnam method, trie data structure.

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 method proposed by Davis, Putnam, Logemann, and Loveland for propositional reasoning [3, 4], often referred as the Davis–Putnam method, 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

* Research supported by NSF under Grants CCR-9202838 and CCR-9357851.
† Research supported by NSF under Grant CCR-8922330.

© 2000 *Kluwer Academic Publishers. Printed in the Netherlands.*

choice of variable for splitting, and so forth. In this paper, we will concentrate on the use of tries (discrimination trees) and related refinements 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 [15, 16, 18]. 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 open problems in algebra concerning the existence of quasigroups satisfying certain constraints [1]. The usefulness of computer programs to attack these quasigroup problems has been demonstrated in [19, 8, 15]. We prefer these quasigroup problems as benchmarks over 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 important, 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 constraint) from $O(n^4)$ to $O(n^6)$ clauses, so large sets of clauses with hundreds of thousands of literals must be handled.

Recently, some incomplete methods based on local search have been proposed that can solve very large size SAT problems [10, 14]. 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 have no solutions, and incomplete methods cannot prove that no solution exists or count the number of solutions. Quasigroup completion problems (completion of partially filled in Latin squares, but without the constraints) are now a subject of research [9].

2. The Davis–Putnam Method

The Davis–Putnam method is based on three simple facts about truth table logic. First, 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

```
function Satisfiable ( clause set S ) return boolean
    /* unit propagation */
    repeat
        for each unit clause L in S do
            /* unit subsumption */
            delete from S every clause containing L
            /* unit resolution */
            delete L̄ from every clause of S in which it occurs
        od
        if S is empty then
            return true
        else if a clause becomes null 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 ∪ {L̄} ) then
        return true
    else
        return false
    fi
end function
```

Figure 1. A simple Davis–Putnam algorithm.

subsumption to any set of propositional clauses results in an equivalent set. Second, 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. Third, 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, to eliminate as much as possible other factors of the implementations, unless specified otherwise, all of our programs will use identical input and simply choose the minimally indexed variable not yet assigned a value for splitting. This literal selection strategy results in a substantial increase in the size of the search space (evident in Tables I and V), but ensures comparability of results; the focus of this paper is the effect of the trie data structure and other refinements on the speed of unit propagation, not the effect of splitting heuristics on search space size.

We have tested some other advanced SAT techniques, such as intelligent backjumping and lemma generation, with the trie data structure with good preliminary results [18]. However, for quasigroup problems, these techniques seem not to have much positive impact on performance. A thorough discussion is outside the scope of this paper.

3. 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. If two clauses (represented as lists of integers in ascending order by absolute value) have the same prefix of length n, then they 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 empty (nil),

[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.

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 implicitly marked by var; the edge from this node to neg is implicitly marked by $(-var)$; $rest$ is the next node in the linear list of nodes that have the same parent node in the trie; $rest$ does not contain any var or $(-var)$ edges.

A set S of (nontautologous) propositional clauses is 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:

$P = \{v \vee P_1, \ \ldots, \ v \vee P_n\}$ — the clauses that contain v positively.
$Q = \{\overline{v} \vee Q_1, \ \ldots, \ \overline{v} \vee Q_m\}$— 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 $S = \{x_1 \vee x_2, \overline{x}_1 \vee \overline{x}_2\}$, 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, or $rest$. 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 $rest$. We replace nodes by their simpler equivalents whenever possible in the *trie-merge* operation.

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

```
function trie-merge (trie t1, trie t2) return trie
    if t1 = □ or t2 = □ then
        return □
    else if t1 = nil then
        return t2
    else if t2 = nil then
        return t1
    fi
    let t1 = ⟨v1, p1, n1, r1⟩
    let t2 = ⟨v2, p2, n2, r2⟩
    if v1 = v2 then
        let n = trie-merge(n1, n2)
        let r = trie-merge(r1, r2)
        return ⟨v1, trie-merge(p1, p2), n, r⟩
    else if v1 < v2 then
        return ⟨v1, p1, n1, trie-merge(r1, t2)⟩
    else
        return ⟨v2, p2, n2, trie-merge(r2, t1)⟩
    fi
end function
```

Figure 2. The *trie-merge* procedure.

- Memory usage is reduced because of shared clause prefixes.
- Unit clauses can be found quickly.
- Tail-subsumed clauses are automatically eliminated when the trie is constructed. A clause is said to be *tail-subsumed* by another clause if its first portion of the literals is also a clause. For example, $x_1 \vee x_2 \vee x_3$ is tail-subsumed by $x_1 \vee x_2$.
- Tail-resolutions are automatically performed when the trie is constructed. If $c_1 \vee L$ and $c_1 \vee \overline{L} \vee c_2$ are two clauses, the *tail-resolution* is the resolution on the last literal of the first clause and the result is $c_1 \vee c_2$ which subsumes the second clause. So the second clause will be replaced by $c_1 \vee c_2$. In our implementation, such resolutions will be performed no matter which clauses enters the trie first.
- The *unit propagation* operation can be performed 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 variable assignments can be done in a single traversal of the trie.

4. Quasigroup Problems

The quasigroup problems are given by Fujita, Slaney, and Bennett in their award-winning IJCAI paper [8]. 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 $0, 1, \ldots, (v-1)$ — with certain constraints. Given $0 \leq i, j < v$, 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 $x, y, u, w \in S = \{0, \ldots, (v-1)\}$,

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

It was shown in [15] that the following two clauses are valid consequences of the clauses above and adding them reduces the search space.

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

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

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

[2] The QG7 constraint is the one used in [15], not [8].

the idempotency law, $x*x = x$, and a constraint such as $x*(v-1) \geq x-1$ to eliminate some isomorphic models are used for each problem here.

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 directly, but constraints QG3–QG7, must be transformed 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:

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

Experiments have shown that adding these "transposes" to the input can reduce the search space; this is also true of QG3, QG4, QG6, and QG7. More information on quasigroup problems can be found in [1], [8], [15], and [17].

5. DDPP: A Nondestructive Trie-based Implementation

DDPP (Discrimination-tree-based Davis–Putnam Prover) is a straightforward implementation of the Davis–Putnam method based on the *trie-merge* operation. It performs the operation nondestructively, and the result shares (nearly) maximal structure with the original trie to minimize the memory allocation. DDPP is written in Common Lisp. A detailed description can be found in [13].

Performance of DDPP (and LDPP) on some quasigroup problems is shown in Table I. Search was continued until the search space was exhausted and all models had been found. The number of branches is one plus the number of splittings in the Davis–Putnam method. The data was collected using CMU Common Lisp on a 200 MHz Pentium Pro processor with 128 MB memory. The same sets of clauses were used for all results in this paper.

Several open problems about quasigroups were first solved by DDPP [15]: QG5.13, QG5.14, and QG5.15 (negatively), and QG4.12 (positively), but DDPP was still 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

Table I. DDPP (nondestructive trie representation) vs. LDPP (destructive list representation); LDPP′ omits unit subsumption.

Problem	Clauses	Models	Branches	DDPP Search (sec)	LDPP Search (sec)	LDPP′ Search (sec)
QG1.7	68083	8	958	5.8	13	6.1
.8	148957	16	590624	4011	11489	5101
QG2.7	68083	14	672	4.8	11	4.3
.8	148957	2	579624	4029	12282	5629
QG3.8	10469	18	1016	7.4	1.9	0.8
.9	16732	–	82405	717	122	71
QG4.8	9685	–	910	7.4	1.6	0.6
.9	15580	194	59514	610	98	61
QG5.9	28540	–	188	4.2	0.7	0.4
.10	43636	–	1454	63	10	4.2
.11	64054	5	12581	962	116	54
.12	90919	–	139582	18062	1807	853
.13	125464	–	1798177	370626	32704	15255
QG6.9	21844	4	52	1.7	0.3	0.1
.10	33466	–	314	14	2.0	0.7
.11	49204	–	2523	185	20	9.6
.12	69931	–	26848	3400	315	142
QG7.9	22060	4	42	1.1	0.3	0.1
.10	33736	–	817	28	4.5	1.8
.11	49534	–	11617	802	90	42
.12	70327	–	159908	20609	1942	913
.13	97072	64	2351662	552988	45573	19684

searching was still necessary. Moreover, the nondestructive *trie-merge* still required some storage allocation. Nevertheless, the approach remains appealing, particularly when destructive operations are undesirable or impractical as in logic programming.

6. LDPP: A Destructive Non-Trie-Based Implementation

LDPP (Linear-list-based Davis–Putnam prover) was written to avoid the cost of the nondestructive *trie-merge* operation in unit propagation. It uses reversible destructive operations on lists instead. Like DDPP, LDPP is written in Common Lisp.

LDPP is generally much faster than DDPP. It performs unit resolution and unit subsumption operations quickly by decrementing and setting fields. Resolvable or subsumable clauses need not be searched for because each variable contains pointers to all the clauses that contain the variable.

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

- `positive-literals`, `negative-literals`: List of pointers to variables occurring positively (resp. negatively) in this clause.
- `subsumed`: If the clause has been subsumed, this field contains a pointer to the variable whose assignment subsumed this clause; otherwise it is `nil`.
- `number-of-active-positive-literals`, `number-of-active-negative-literals`: When `subsumed` is `nil`, this is the number of variables in `positive-literals` (resp. `negative-literals`) that have not been assigned a value.

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` otherwise.
- `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`.
- Unit subsumption: Every clause in `contained-positively-clauses` has its `subsumed` field set to the variable, unless `subsumed` was already non-`nil`.
- Unit resolution: Every clause in `contained-negatively-clauses` has its `number-of-active-negative-literals` field decremented by one, unless `subsumed` 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.

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 `subsumed` field is `nil`; the literals still present in these clauses are those in `positive-literals` and `negative-literals` whose `value` is `nil`. An assignment can be undone during backtracking before trying an alternative assignment. Many other implementations of the Davis–Putnam method employ a similar approach, including Crawford and Auton's NTAB [2], Letz's SEMPROP, and McCune's MACE [12].

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 7.2.2).

7. SATO: A New Algorithm for Unit Propagation

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) [16] 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 based on the trie data structure.

SATO was used 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.

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 one of the following occurs: (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 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.

In Table II, results for two versions of SATO are given — SATO1 uses only the head list and SATO2 uses both the head and tail lists.

Table II. SATO1 (atoms point to clauses whose first literal contains it) vs. SATO2 (atoms point to clauses whose first or last literal contains it).

Problem	Branches	SATO1 (sec)	SATO2 (sec)
QG1.7	958	1.20	0.53
.8	590624	4328.93	283.10
QG2.7	672	1.05	0.49
.8	579624	3698.32	228.32
QG3.8	1016	1.46	0.28
.9	82405	121.39	12.02
QG4.8	910	0.97	0.20
.9	59514	93.31	12.08
QG5.9	188	0.81	0.24
.11	12581	211.03	18.32
.12	139582	6923.66	377.14
.13	1798177	137902.56	10099.64
QG6.9	52	0.77	0.15
.10	314	2.61	0.34
.11	2523	47.10	3.70
.12	26848	953.59	63.23
QG7.9	42	0.68	0.14
.10	817	2.44	0.47
.11	11617	32.57	10.25
.12	159908	2580.45	412.21
.13	2351662	160396.05	13908.68

The experimental results indicate that SATO2 is substantially faster than SATO1 on quasigroup problems. The times in Tables II and III were collected on an SGI Onyx R10000 processor (196 MHz) with 256 MB memory. SATO is written in C.

7.1. 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 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 $H_{\overline{x}}$. We need to perform $H_{\overline{x}}$ operations to add those literals that follow $\overline{x}$ in each of $H_{\overline{x}}$ 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 $O(k * H_{\overline{x}})$. 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, $H_{\overline{x}}$, 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.

7.2. Experimental Comparison of Two Algorithms

While the theoretical analysis shows that SATO's method has an advantage over LDPP's, it also appears to perform better in practice (see Tables I and II). 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) eliminating using unit subsumption.

7.2.1. *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. Table III lists the results for SATO1.2 and SATO1.3. SATO1.3 always takes less time than SATO1.2 to finish the job. This is in part because the trie data structure eliminates duplicate clauses. Actually, it automatically deletes a class of subsumed clauses, that is, those clauses one of whose prefixes (regarding a clause as a list) is also a clause in the system.

Table III. SATO1.2 (list representation) vs. SATO1.3 (trie representation).

Problem	Branches	SATO1.2 Create (sec)	SATO1.2 Total (sec)	SATO1.3 Create (sec)	SATO1.3 Total (sec)
QG1.7	958	0.38	1.29	0.38	0.68
.8	590624	0.85	2084.10	0.86	408.93
QG2.7	672	0.39	1.05	0.39	0.64
.8	579624	0.85	2293.32	0.91	328.45
QG3.8	1016	0.05	0.46	0.05	0.32
.9	82405	0.08	36.26	0.08	24.29
QG4.8	910	0.05	0.39	0.05	0.28
.9	59514	0.08	27.70	0.08	18.26
QG5.9	188	0.14	0.31	0.14	0.26
.10	1454	0.22	2.51	0.23	1.57
.11	12581	0.32	50.03	0.35	23.45
.12	139582	0.46	1317.66	0.54	490.96
.13	1798177	0.64	29901.56	0.79	11167.84
QG6.9	52	0.11	0.17	0.11	0.17
.10	314	0.17	0.61	0.17	0.51
.11	2523	0.25	6.59	0.27	5.05
.12	26848	0.35	163.88	0.41	113.37
QG7.9	42	0.11	0.15	0.11	0.17
.10	817	0.17	1.09	0.17	0.77
.11	11617	0.25	31.48	0.27	19.30
.12	159908	0.35	1241.95	0.42	571.83
.13	2351662	0.49	57294.93	0.63	18495.21

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 Table III. The creation times for SATO1 and SATO2 are the same as those of SATO1.3.

7.2.2. *Eliminating 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 a `subsumed` 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 the use of the `subsumed` 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 Table I, it is appreciably faster than LDPP on quasigroup problems.

8. Conclusions

In this paper, we have concentrated on the use of the trie data structure and other refinements 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, SATO1.3. Table 7.2.2 summarizes the characteristics of the different programs described here.

We conclude the following:

- 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., SATO1.3 vs. DDPP).
- Eliminating the unit subsumption operation can improve performance (e.g., LDPP′ vs. LDPP).

Table IV. Summary of program characteristics (R: representation; US: unit subsumption).

Program	R	How Resolvable Literals Are Found	US
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

– 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.

Table V. DDPP and LDPP′ with standard literal selection strategy: a literal of a shortest positive clause is chosen for splitting; search is halted after finding a model in satisfiable cases (marked by *).

	DDPP			LDPP′		
Problem	Branches	Search (sec)	Total (sec)	Branches	Search (sec)	Total (sec)
QG1.7*	11	0.2	7.5	11	0.1	7.0
.8*	57421	640	658	39349	309	327
QG2.7*	5	0.2	8.3	45	0.4	10
.8*	50721	541	560	4405	48	69
QG3.8*	379	2.8	3.8	403	0.3	0.9
.9	26847	280	281	24673	25	26
QG4.8	564	4.3	5.2	602	0.4	1.6
.9*	450	6.0	7.5	904	0.9	1.8
QG5.9	15	0.6	3.7	15	0.1	2.2
.10	50	3.1	9.1	38	0.2	4.1
.11*	94	12	21	80	0.5	6.5
.12	443	70	84	369	2.9	11
.13	16438	2802	2826	12686	123	139
QG6.9*	4	0.5	2.7	12	0.0	1.5
.10	65	2.5	7.0	59	0.1	3.2
.11	451	27	35	539	2.1	7.1
.12	5938	564	576	7288	38	43
QG7.9*	1	0.2	2.5	1	0.0	1.9
.10	40	1.6	5.7	40	0.1	4.2
.11	321	24	31	294	1.2	5.5
.12	2083	219	231	1592	7.6	13
.13*	30	7.2	26	205	1.5	11

Many aspects of the Davis–Putnam method are not addressed in this paper. To facilitate comparison, all of our programs use the same input and simply choose the minimally indexed variable not yet assigned a value for splitting in the Davis–Putnam method. Many selection heuristics can be efficiently implemented using the trie data structure; we ignore this because it is outside the scope of this paper. Our programs

Table VI. SATO and MACE with standard literal selection strategy: a literal of a shortest positive clause is chosen for splitting; search is halted after finding a model in satisfiable cases (marked by *).

	SATO			MACE [12]		
Problem	Branches	Search (sec)	Total (sec)	Branches	Search (sec)	Total (sec)
QG1.7*	85	0.1	0.6	16	0.2	2.0
.8*	40755	29	30	39357	63	67
QG2.7*	33	0.0	0.7	53	0.2	2.1
.8*	28155	29	30	4410	11	15
QG3.8*	239	0.1	0.2	408	0.2	0.4
.9	20340	10	10	24763	14	14
QG4.8	924	0.3	0.4	602	0.2	0.4
.9*	54	0.0	0.2	913	0.5	0.8
QG5.9	15	0.1	0.3	15	0.1	0.7
.10	38	0.1	0.5	38	0.1	1.1
.11*	41	0.2	0.7	82	0.4	1.8
.12	354	1.6	2.4	369	1.7	3.7
.13	12781	65	66	12686	79	82
QG6.9*	9	0.0	0.2	13	0.0	0.5
.10	65	0.1	0.4	59	0.1	0.8
.11	625	1.2	1.7	539	1.0	2.1
.12	7270	18	19	7288	20	22
QG7.9*	6	0.0	0.2	3	0.0	0.5
.10	42	0.1	0.4	40	0.1	0.8
.11	324	0.7	1.1	294	0.6	1.6
.12	1642	4.4	5	1592	3.7	5.2
.13*	22801	81	81	210	1.0	3.0

do not perform general 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 check for symmetries. Further research is needed to see how such operations can be done efficiently using the trie data structure.

Table VII. POSIT, SATZ, NTAB: default settings; search is halted after finding a model in satisfiable cases (marked by *).

Problem	POSIT [7] Branches	POSIT [7] Search (sec)	POSIT [7] Total (sec)	SATZ [11] Branches	SATZ [11] Total (sec)	NTAB [2] Branches	NTAB [2] Total (sec)
QG1.7*	6	0.0	3.0	7	33	75	3.4
.8*	5151	34	40	7770	218	2346	106
QG2.7*	21	0.1	3.0	10	53	12	2.5
.8*	5761	41	47	6975	270	4009	207
QG3.8*	49	0.2	0.6	221	1.2	6	0.3
.9	4803	19	19	8083	33	4585	59
QG4.8	113	0.3	0.7	255	1.5	104	1.0
.9*	63	0.2	0.9	585	3.6	134	2.0
QG5.9	1	0.1	1.2	3	6.2	4	0.9
.10	?	?	fault	29	12	19	2.8
.11*	?	?	fault	12	18	15	4.3
.12	?	?	fault	199	47	?	>13000
.13	6361	310	315	?	>18000	?	>2700
QG6.9*	?	?	fault	6	3.0	12	0.8
.10	55	0.7	2.1	60	6.0	77	3.3
.11	?	?	fault	462	22	1714	117
.12	?	?	fault	6623	318	?	>2700
QG7.9*	?	?	>2700	1	3.9	3	0.7
.10	15	0.2	1.6	3	6.6	16	1.4
.11	205	2.3	4.4	119	15	258	18
.12	?	?	fault	1116	70	1745	156
.13*	?	?	fault	2969	244	?	>2700

Finally, we conclude the paper with statistics comparing DDPP, LDPP′, and SATO with four other powerful SAT solvers: MACE [12], POSIT [7], SATZ [11], and NTAB [2] (Tables V, VI, and VII).[3] DDPP and LDPP are written in Common Lisp; all the others are written in C. We succeeded in solving all the problems only with our solvers and MACE. McCune's MACE uses data structures similar to those

[3] The data for these tables were collected on a 200 MHz Pentium Pro processor with 128 MB memory.

of LDPP, the same "choose an atom of a shortest positive clause" literal selection strategy as our solvers and has also been used to solve open quasigroup existence problems. We believe the other systems are competent to solve these problems with some adjustment, but the tables illustrate the nontriviality of the problems and that superior performance is not automatically and immediately transferrable between problem domains.[4] We welcome readers to try these quasigroup problems — and harder ones, many still open — using their systems.

Appendix

The CSPLib problem library (`http://csplib.cs.strath.ac.uk`) contains the Common Lisp quasigroup problem generator used for this paper and an ILOG Solver generator. The code for SATO is available from `http://cs.uiowa.edu/~hzhang/sato/`. SATO (full version) can be used to generate clauses for the quasigroup problems presented in this paper. For instance, the command `sato -Q5 -G9 -o` will generate QG5.9 in Lisp format, while `sato -Q3 -G10 -o2` will generate QG3.10 in DIMACS format.

References

1. Bennett, F. E. and Zhu, L.: 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 Auton, L. D.: Experimental results on the crossover point in satisfiability problems, *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93),* 1993, pp. 21–27.
3. Davis, M. and Putnam, H.: A computing procedure for quantification theory, *Journal of the Association for Computing Machinery* 7(3) (July 1960), 201–215.
4. Davis, M., Logemann, G., and Loveland, D.: 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, in *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92),* San Jose, California, July 1992, pp. 780–785.
6. Dowling, W. F. and Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae, *Journal of Logic Programming 3* (1984), 267–284.
7. J. W. Freeman, Improvements to propositional satisfiability search algorithms, Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, May 1995.

[4] Likewise, our solvers have not been adapted to random 3-SAT problems that benefit from features such as pure literal detection and different literal selection strategies.

8. Fujita, M., Slaney, J., and Bennett, F.: Automatic generation of some results in finite algebra, in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1993.
9. Gomes, C., Selman, B., and Crato, N.: Heavy-tailed distributions in combinatorial search, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP97)*, Linz, Austria, 1997.
10. Gu, J.: Local search for satisfiability (SAT) problem, *IEEE Transactions on Systems, Man, and Cybernetics* 23(4) (1993) 1108–1129.
11. Li, C.M., Anbulagan, M.: look-ahead versus look-back for satisfiability problems, in *Proceedings of International Conference on Principles and Practice of Constraint Programming*, 1997.
12. 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.
13. 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.
14. Selman, B., Levesque, H., and Mitchell, D.: A new method for solving hard satisfiability problems, in *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92),* San Jose, California, July 1992, pp. 440–446.
15. Slaney, J., Fujita, M., and Stickel, M.: Automated reasoning and exhaustive search: quasigroup existence problems, *Computers and Mathematics with Applications 29,* 2 (May 1995), 115–132.
16. Zhang, H.: SATO: a decision procedure for propositional logic, *Association for Automated Reasoning Newsletter,* No. 22 (March 1993), pp. 1–3.
17. Zhang, H.: Specifying Latin squares in propositional logic, in R. Veroff (ed.): *Automated Reasoning and Its Applications, Essays in Honor of Larry Wos,* Chapter 6, MIT Press, 1997.
18. Zhang, H.: SATO: An efficient propositional prover, in *Proc. of International Conference on Automated Deduction* (CADE-97), Lecture Notes in Artificial Intelligence 1104, Springer-Verlag, 1997, pp. 308–312.
19. Zhang, J.: Search for idempotent models of quasigroup identities, typescript, Institute of Software, Academia Sinica, Beijing, 1991.