An abbreviated version of this paper appears in:
J.P. Jouannaud (ed.), *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, Vol. 845, pages 34-49, Springer-Verlag, September 1994.

Ordered Binary Decision Diagrams and the Davis-Putnam Procedure

Tomás E. Uribe
Computer Science Department
Stanford University
Stanford, CA 94305
uribe@cs.stanford.edu

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

June 23, 1994 (Draft)

Abstract

We compare two prominent decision procedures for propositional logic: Ordered Binary Decision Diagrams (OBDDs) and the Davis-Putnam procedure. Experimental results indicate that the Davis-Putnam procedure outperforms OBDDs in hard constraint-satisfaction problems, while OBDDs are clearly superior for Boolean functional equivalence problems from the circuit domain, and, in general, problems that require the schematization of a large number of solutions that share a common structure. The two methods illustrate the different and often complementary strengths of constraint-oriented and search-oriented procedures.

1 Introduction

We compare two of the most successful decision procedures for propositional logic: Ordered Binary Decision Diagrams [Bryant, 1986] and the Davis-Putnam procedure [Davis *et al.*, 1962, Davis and Putnam, 1960]. These methods were developed and used in different research communities with different applications in mind, and there has been a striking lack of comparison of their capabilities. Nevertheless, the topic is a matter of some curiosity. After the successful application of the Davis-Putnam procedure to previously open quasigroup existence problems [Slaney *et al.*, 1993], the question of how well OBDDs would do on the same problems arose repeatedly. Here we have tried to answer that and similar questions.

The satisfiability problems we consider are well beyond the reach of resolution-based theorem provers; while the strength of such systems lies in the full first-order setting, here we consider only the propositional case, for which the present two methods are much better suited. Hill-climbing procedures can find models for hard 3-SAT problems beyond the reach of the methods considered here [Selman *et al.*, 1992] (but have so far been found to be ineffective for finding models of satisfiable quasigroup problems [Walsh, 1993]); however, they suffer from being incomplete, and thus are not *decision* procedures that can be used in refutational theorem proving or to effectively enumerate all the solutions for a given problem.

This research was supported by the National Science Foundation under Grant CCR-8922330. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the National Science Foundation or the United States Government.

There are, of course, other methods for solving propositional problems, including integer linear programming and the many approaches used within the Constraint Logic Programming (CLP) framework (*e.g.*, [Simonis and Dincbas, 1993]). In particular, CLP systems—where binary decision diagrams themselves have been used to represent Boolean constraints—have proved to be effective for a wide range of problems, including encouraging preliminary results for the quasigroup ones [Wallace, 1993]. However, a comparison of the two methods described here and such others is beyond the scope and main intentions of this paper (see Section 5 for more on related work).

2 Ordered Binary Decision Diagrams

Ordered Binary Decision Diagrams (hereafter referred to as OBDDs) were introduced by Bryant [1986] as a tool for the efficient representation and manipulation of Boolean functions. Briefly, an OBDD is a directed acyclic graph where each node is labeled with a Boolean variable and has two outgoing edges, labeled as "*then*" or "*else*," except for two distinguished nodes, labeled *true* and *false*, which have no outgoing edges. A unique *root* node has no incoming edges. The ordering restriction stipulates that there can be an edge from a node labeled v_1 to a node labeled v_2 only if $v_1 \prec v_2$ in a given total ordering $\prec$ on the set of variables.

The Boolean function represented by an OBDD is recursively defined by a series of "if v then f_1 else f_2" tests, where the variable v is the root node label and f_1 and f_2 are the Boolean functions described by its descendants. The node *true* or *false* encountered at the end of a sequence of these tests indicates the value of the function for the corresponding truth-value assignment.

When isomorphic subgraphs are merged, the OBDD representation is *canonical* given a fixed ordering on the variables: two functions are equal if and only if they are represented by the same OBDD. As Moore [1992] points out, OBDD representations can be efficiently manipulated due mainly to three factors: computing Boolean operations over functions represented by OBDDs can be speeded up by taking into account the fixed variable ordering, structure can be shared among OBDDs, and, finally, the results of combination operations can be cached for future reference [Brace *et al.*, 1990].

Given the OBDD representation of functions f and g under a fixed variable ordering, the time needed to compute OBDDs representing the basic Boolean combinations ($\wedge$, $\vee$, etc.) of f and g is polynomial in the product of the sizes of their OBDDs [Bryant, 1986]. Thus, as Bryant [1992] points out, complex computations that are formulated as a sequence of operations on OBDDs remain tractable as long as the OBDD sizes remain relatively small.

It is well known that the ordering of the variables can drastically affect the size of the OBDD representation of different functions (and thus the time spent manipulating them). We will discuss this problem at length in Section 4; for now, we note that most work on variable ordering heuristics is in the circuit domain (*e.g.*, [Butler *et al.*, 1991]), and that the current best algorithm for computing an optimal ordering for a given formula is exponential in the number of variables [Friedman and Supowitz, 1990].

A basic feature of solving constraint-satisfaction problems using OBDDs alone is that *there is no search*: the problem-solving process consists entirely of constructing an OBDD representation for a Boolean function that satisfies a given set of constraints. Thus, the final result represents *all solutions*. Therefore, in fairness to OBDDs, we will compare them with exhaustive search algorithms, and not with those that find only one solution. However, this distinction is irrelevant in those cases where there are no solutions at all, since any procedure has to "exhaust" the space of possible solutions in one way or another.

Apart from the question of variable ordering, another basic difficulty in the search for an efficient way of constructing OBDD representations is problem subdivision. This question has two components: (1) "grainsize" considerations: what are the smallest "OBDD units" from which the final solution is to be computed? And (2) in what order should these constraints be combined? As we will see, these factors are not as crucial in the circuit domain as in the general case.

See [Bryant, 1992] for a survey of OBDDs and their applications, the most successful of which is the symbolic model checking of finite-state hardware specifications [Burch *et al.*, 1992, Long, 1993, McMillan, 1993]. For an overview of OBDDs directed to the automated deduction community, see [Moore, 1992]. The experiments reported here used the OBDD C library written by David E. Long [1993], which follows the guidelines described in [Brace *et al.*, 1990].

3 The Davis-Putnam Procedure

The Davis-Putnam procedure[1] [Davis *et al.*, 1962, Davis and Putnam, 1960] is an algorithm for deciding the satisfiability of formulas in the propositional calculus. With some bookkeeping, it can also return models of the formula, of which we are especially interested in the *minimal* ones (a model is identified with the set of atomic formulas that it assigns *true*, and is minimal if no other model makes fewer *true*).

Definition 1 (Minimal Models) *We identify a model M with the set of atomic formulas that are assigned true by M. A model M is* minimal *iff there is no other model M' such that $M' \subset M$.*

Restricting our attention to minimal models helps improve the efficiency of the Davis-Putnam procedure. For many highly constrained problems (such as the n-queens and n-rooks problems of Sections 4.1 and 4.2 and the quasigroup existence problems of Section 4.5) all models are both minimal and maximal: changing any atom to *true* or *false* produces a contradiction.

The Davis-Putnam procedure operates on formulas in *clause form*. Its input is a set of clauses, where each clause is a set of literals and each literal is either an atomic formula (a positive literal) or its negation (a negative literal). A clause is interpreted as the disjunction of its literals and a set of clauses as the conjunction of its clauses.

Definition 2 (Clauses) *A* positive clause *contains only positive literals; a* negative clause *contains only negative literals; a* Horn clause *contains at most one positive literal; the* empty clause *contains no literals and is unsatisfiable; a* unit clause *has a single literal.*

A version of the Davis-Putnam procedure is defined in Figure 1. Calling $DP(S, \{\})$ returns a set of models for the set of clauses S that includes all minimal models and possibly some nonminimal ones. The *assign* operation applies an atom's truth-value assignment to a set of clauses.

The Davis-Putnam procedure customarily uses the additional rule that if an atomic formula A occurs only positively in S, assign *true* to it, and if it occurs only negatively in S, assign *false* to it. We do not use this rule (except implicitly in Step 5) because detecting that an atomic formula occurs only positively or only negatively can be expensive and the condition does not occur in many types of problems. Moreover, use of this rule can result in not all minimal models being returned (although the procedure retains its validity as a satisfiability tester).

[1] Or, more accurately, the *Davis-Logemann-Loveland* procedure, after [Davis *et al.*, 1962].

$DP(S, M) \equiv$

1. If $S = \{\}$ (S is the satisfiable empty set of clauses), return $\{M\}$.
2. If $\{\} \in S$ (S contains the unsatisfiable empty clause), return $\{\}$.
3. If a positive unit clause $\{A\} \in S$, return $DP(assign(A, true, S), M \cup \{A\})$.
4. If a negative unit clause $\{\neg A\} \in S$, return $DP(assign(A, false, S), M)$.
5. If S contains no positive clauses, return $\{M\}$.
6. Let A be an atomic formula occurring in a positive clause with smallest size in S.

$$\text{Return : } DP(assign(A, true, S), M \cup \{A\}) \bigcup DP(assign(A, false, S), M).$$

$assign(A, true, S) \equiv$

1. Let $S' = \{C \mid C \in S \wedge A \notin C\}$ (do unit subsumption by A).
2. Return $\{C - \{\neg A\} \mid C \in S'\}$ (do unit resolution with A).

$assign(A, false, S) \equiv$

1. Let $S' = \{C \mid C \in S \wedge \neg A \notin C\}$ (do unit subsumption by $\neg A$).
2. Return $\{C - \{A\} \mid C \in S'\}$ (do unit resolution with $\neg A$).

Figure 1: The Davis-Putnam procedure

Different criteria for choosing a literal to split on in Step 6 could be used. Splitting on a literal in a positive clause reduces the number or size of nonunit positive clauses, pushing the problem closer to the tractable case of Horn clauses. This criterion is also consistent with the objective of finding all minimal models. If a literal of an all-negative clause were picked instead, and Step 5 checked for absence of negative clauses, the number of *false* assignments rather than the number of *true* assignments would be minimized.

LDPP (the List-based Davis-Putnam Prover) is a LISP implementation of the Davis-Putnam procedure written by Mark Stickel, and a successor of the earlier DDPP used to solve quasigroup problems in [Slaney *et al.*, 1993]. LDPP implements the Davis-Putnam procedure without logical refinements, and is fairly efficient because of the way it performs the crucial *assign* operation. LDPP uses reversible destructive list operations, similarly to Zhang's SATO [1993], Crawford and Auton's TABLEAU [1993], and Letz's SEMPROP theorem provers. All of the following problems were also run using SATO, a discrimination tree-based implementation written in `C`, with comparable results. See [Zhang and Stickel, 1994] for a detailed description of LDPP and SATO.

In LDPP, a set of clauses is represented by a list of structures for clauses and a list of structures for atomic formulas. The structure for a clause contains the fields:

- `positive-literals`, `negative-literals`: List of pointers to structures for atomic formulas 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 structure for the atomic formula whose assignment subsumed this clause.
- `number-of-active-positive-literals`, `number-of-active-negative-literals`: When `inactivated` is `nil`, this is the number of atomic formulas in `positive-literals` (resp. `negative-literals`) that have not been assigned a truth-value (*i.e.*, that have not been resolved away from this clause).

The structure for an atomic formula contains the fields:

- `value`: This is `true` if the atomic formula 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 structures for clauses that contain this atomic formula positively (resp. negatively).

To assign *true* to an atomic formula, its `value` field is set to `true`, and every active clause in `contained-positively-clauses` has its `inactivated` field set to the atomic formula. Every active clause in `contained-negatively-clauses` has its `number-of-active-negative-literals` field decremented by one. Note that we *don't* modify `negative-literals` itself. If the total number of active positive and negative literals is zero, the current truth assignment yields the unsatisfiable empty clause. If the total is one, a new unit clause has been produced. The newly derived unit clause can be identified by finding the only atom in the list of positive or negative literals whose value is `nil`. These are queued and assigned values before *assign* exits so that all unit propagation is done inside the *assign* procedure. These operations are undone to restore the set of clauses in order to test alternative assignments. Assignment of *false* to an atomic formula is done analogously.

4 Experimental Results

4.1 The n-Queens Puzzle

We consider first the classic *n-queens puzzle*—placing n queens on an $n \times n$ chess board so that no two queens attack each other—not because the puzzle has any intrinsic interest to us,[2] but because it is a simple and well-known constraint satisfaction benchmark that illustrates interesting issues concerning OBDDs.

The problem was encoded in OBDD form using a Boolean variable for each of the n^2 positions in the board; an assignment of *true* to a variable indicates a queen in the corresponding position. The general procedure for finding solutions using OBDDs is simple: we begin with the *true* OBDD and successively conjoin it with the constraints that define the puzzle. The final result is an OBDD that represents the function of n^2 variables that is *true* exactly for those assignments that encode queen positions that satisfy all the constraints. Thus, this final OBDD represents all the solutions to the original problem.

The OBDD variable ordering we used was the lexicographic ordering on the board positions: the $\langle x, y\rangle$ variable precedes the $\langle x', y'\rangle$ variable iff either $x < x'$, or else $x = x'$ and $y < y'$. Consider the Boolean function f_i that is *true* iff there is at least one queen in row i. Under this ordering, f_i can be represented with an OBDD of size linear in n. In general, we have the following (see [Bryant, 1992]):

[2] Slaney [1993] presents a thoughtful critique of this problem as a CSP benchmark.

Proposition 3 *A Boolean function is called* symmetric *if its value depends only on the* number *of true arguments. The size for an* OBDD *representation of a symmetric function of n arguments is independent of the variable ordering and has an upper bound of n^2.*

Now consider the conjunction of the functions f_i for n different rows, $f_{+R_n} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} f_i$. The size of the corresponding OBDD is the sum of the sizes of the n OBDDs: the graph for f_{+R_n} can be obtained by changing the edges that reached the *true* node in $f_{+R_{n-1}}$ to point to the root of f_n (the function that specifies that there is at least one queen in row n). Since each f_i has size n, this yields a final OBDD for f_{+R} of size n^2.

If we now consider the corresponding constraints over the *columns*, the situation is not so favorable: since all the tests for variables on row 1 must precede all tests for variables on row 2, all 2^n possible truth-value assignments for the variables on row 1 have to be considered before variables on row 2 can be branched upon. Each assignment corresponds to a distinct OBDD node, since the "subfunction" of the remaining variables is different for each. This immediately gives a lower bound of 2^n on the size of the OBDD under this ordering.[3]

Similarly, the constraint f_{-R} that specifies that there are no two queens in any row has an OBDD representation of size n^2, while that of the corresponding constraint f_{-C} on the columns is of (at least) exponential size in n, for this particular order on the variables. The following theorem establishes a lower bound for the size of the OBDD representation of the conjunction of the row and column constraints given *any* variable ordering:

Theorem 4 *Let f_{+RC} be the Boolean function $f_{+R} \wedge f_{+C}$ that is true iff there is at least one queen in each row and column of the $n \times n$ chess board. Then for any variable ordering, the size of the* OBDD *representing f_{+RC} is bounded from below by $O(2^{2\lceil\sqrt{n}-1\rceil})$.*

Proof: Consider the set $S = \left\{v_{\langle x_1,y_1\rangle}, \ldots, v_{\langle x_n,y_n\rangle}\right\}$ containing the first n variables in any given ordering. To each variable $v_{\langle x_i,y_i\rangle}$ in S we can assign either row x_i or column y_i such that at least $2\lceil\sqrt{n}-1\rceil$ variables are each mapped into a unique row or column: if we let k be the number of different y coordinates appearing in S, then there will be at least $\lceil\frac{n}{k}\rceil$ different x coordinates, so we will be able to assign a different row or column to at least $k + \lceil\frac{n}{k}\rceil - 1$ elements. This expression is minimized when $k = \lceil\sqrt{n}\rceil$, giving a lower bound of $2\lceil\sqrt{n}-1\rceil$.

Let $S_1 \subseteq S$ be the set of $2\lceil\sqrt{n}-1\rceil$ assigned variables. Consider now the function f' obtained from f_{+RC} by fixing the variables in $S - S_1$ to be *false* (so that f' depends on the variables in S_1 and the last $n^2 - n$ variables in the ordering); the crucial observation now is that f' will be a *different function* for each of the $2^{|S_1|}$ combinations of truth assignments to the elements of S_1. Thus, the OBDD representing the original function f_{+RC} will contain at least $2^{2\lceil\sqrt{n}-1\rceil}$ different nodes below the branches for the variables in S. ■

The same result applies to the function $f_{-R} \wedge f_{-C}$ that is *true* iff there are no two queens on the same row or column, and to the function that is *true* iff there is exactly one queen on each row or column (*i.e.*, the function encoding the *n-rooks problem*—see Section 4.2). Tightening these bounds appears to be an interesting exercise in combinatorics, from which we refrain. For more general techniques for proving lower bounds on OBDD representations see [1991], where Bryant proves an exponential lower bound for integer multiplication. (Nonetheless, McMillan [1993] has characterized a large class of circuits with polynomial-size OBDD representations.)

It is possible for two sets of constraints that are "intractable" on their own to be combined efficiently into a polynomial-sized one. For a trivial example, consider any two functions with

[3] A similar argument applies to the variables at each row (modulo the truth assignments that have led to that subgraph of the OBDD), so this is very much a *lower bound* on the final OBDD size.

exponential OBDD representation, f_1 and f_2, extended by a new variable v. Then $f_1 \wedge v$ and $f_2 \wedge \neg v$ are both exponential in size, but their conjunction is the constant function *false* (which we will quickly detect if we happen to process the variable v first).

Back to the original problem, we have still another constraint to add: f_{-D}, which specifies that there are no two queens on the same diagonal. Thus, the n-queens problem is to find an OBDD representation of the function $f = (f_{-R}) \wedge (f_{-C}) \wedge (f_{+R}) \wedge (f_{+C}) \wedge (f_{-D})$. The problem of choosing the best way to combine these constraints remains. The above analysis only shows that it is probably not a good idea to begin with constraints that do not follow the variable ordering, and that $(f_{-R}) \wedge (f_{-C})$ and $(f_{+R}) \wedge (f_{+C})$ have a lower bound of $O(2^{\sqrt{n}})$.

The best strategy we found was an "incremental" one, which we call Strategy A, consisting of three main steps: (1) begin with f_{+R}, the positive constraints on the rows (of size linear in n^2); (2) introduce one OBDD variable at a time, adding *only* those negative constraints (row, column, and diagonal) between the new variable and those variables that have been previously introduced; and (3) add f_{+C}, the positive constraints on the columns. The order in which the variables are introduced follows the OBDD ordering; however, starting from the *last* variable in the ordering and proceeding toward the first proved better than the converse. The OBDD representing the constraints between each new variable and previously introduced variables was conjoined to the main OBDD directly; collecting sets of constraints into intermediate functions before adding them to the main OBDD was not much help.

Independently of the decomposition used, random variable orderings proved to be invariably bad. Many other variable orderings and decomposition schemes were tried, all yielding inferior performance. To illustrate the widely different performances obtained by different decompositions, we compare Strategy A with a suboptimal "large-grainsize" one, which we call Strategy B. This strategy composed constraints for the entire chess board in the following order: f_{+R}, f_{+C}, f_{-R}, f_{-C} and, finally, f_{-D}, the diagonal constraint, split into "positive slope" and "negative slope" constraints. Since the variable ordering used in both strategies was the same, they compute exactly the same final OBDD.

Table 1 shows n-queens results from LDPP and OBDDs. LDPP was run using the default strategy of choosing an atom from the shortest positive clause on which to branch. The number of LDPP branches is almost perfectly exponential, as is the maximum OBDD size. To stress the difference between finding one solution and finding all, we note that LDPP found single solutions for all of these problems (and up to $n = 15$) in under 2 seconds each.[4]

In practice, OBDDs manifest a serious drawback: exponential space requirements that, for instance, made it impossible for us to solve order 12. This is not the case with LDPP, where the practical limiting factor is time.[5]

Figure 2 shows how the OBDD size changes as more constraints are added for Strategies A and B, and clearly illustrates the difference in performance between the two. For Strategy A, range 0–10 corresponds to f_{+R}, range 10–110 to the 100 variables, introduced in the order described above, and range 110–120 to adding f_{+C}. For Strategy B, the ranges correspond to f_{+R} (0–20), f_{+C} (20–40), f_{-R} (40–60), f_{-C} (60–80), diagonals of negative slope (80–100), and, finally, diagonals of positive slope (80–120). If f_{+C} and f_{-R} are exchanged the first peak is removed, and if the diagonal constraints are added together incrementally the last section looks like that of Strategy A (though even in this case performance is inferior to that of Strategy A).

[4]All times reported here are in CPU seconds as measured on SPARC 2 workstations.

[5]In fact, the space required by LDPP is little more than that needed to store the initial set of clauses.

n	# sols.	LDPP time	LDPP branches	OBDD time A	OBDD time B	OBDD max size A	OBDD max size B	OBDD final size
5	10	0.11	14	0.13	0.28	195	634	169
6	4	0.12	40	0.18	0.88	468	2,036	131
7	40	0.24	108	0.64	3.13	1,732	5,996	1,101
8	92	0.56	384	1.66	10.07	6,223	16,610	2,543
9	352	1.88	1,478	6.16	43.65	22,950	56,530	9,559
10	724	7.33	5,716	23.85	220.64	91,216	235,381	25,947
11	2680	32.04	24,476	104.82	—	391,070	—	94,824
12	14,200	184.30	116,082					

Table 1: Results for the n-queens puzzle

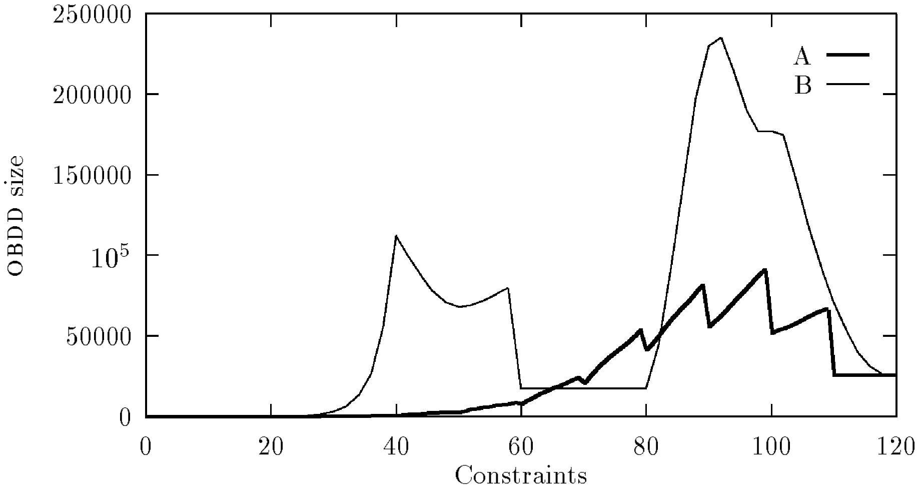

Figure 2: OBDD size for 10-queens, Strategies A and B

4.2 The n-Rooks Problem

Next we consider the *n-rooks problem*, namely, the n-queens puzzle without the diagonal constraints. Unlike the n-queens case, it is easy to find a closed form for the number of n-rooks solutions: $n!$. It is clear that the performance of LDPP, when asked to find all solutions, will necessarily have a lower bound of $O(n!)$; this is *not* the case for OBDDs, if an OBDD is considered an acceptable representation of all solutions. This gives OBDDs an advantage, as Figure 3 shows, and their performance is exponential rather than factorial. The best OBDD strategy we found was the equivalent to Strategy A described in the previous section, with the same variable ordering.

On the other hand, OBDDs still have a memory disadvantage: as was the case with the n-queens problem, LDPP can generate all the solutions to the n-rooks problem in polynomial *space* (obviously, provided the solutions are not stored, but rather consumed by some other process), while the size of the final OBDD has an exponential lower bound like that described in Theorem 4.

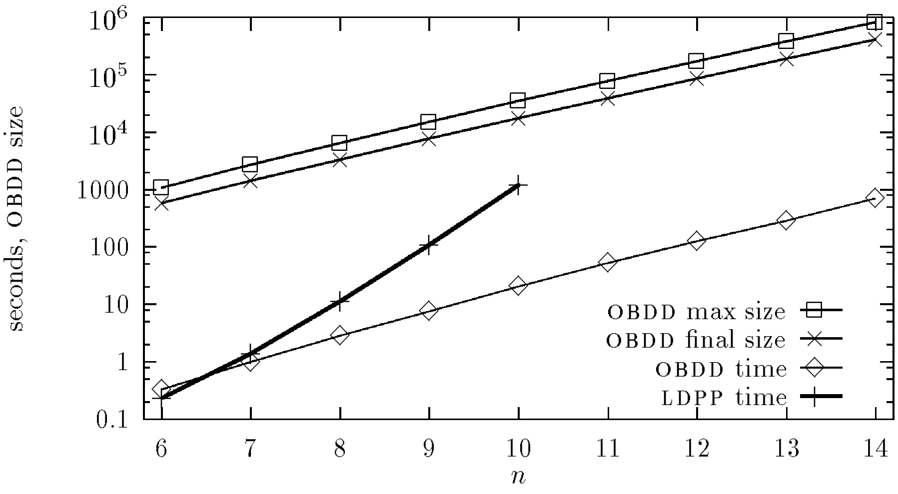

Figure 3: OBDD and LDPP performance on n-rooks

4.3 The Pigeonhole Problem

Another well-known theorem-proving benchmark is to prove the *Pigeonhole Principle*, which states that if n pigeons are placed into $n-1$ boxes, then at least two pigeons must go in the same box. This problem can be viewed as the n-rooks problem for n rows and $(n-1)$ columns, and the results are essentially the same. The number of LDPP branches is $(n-1)!$, and thus LDPP time grows factorially. However, OBDDs produce "only" exponential performance (again, using the equivalent of Strategy A of Section 4.1). This difference allowed OBDDs to solve the problem up to $n = 13$ in a time comparable to that taken by LDPP for $n = 11$.

Table 2 summarizes the results.

n	OBDD time	OBDD max size	clauses	LDPP time	LDPP branches
6	0.11	355	81	0.07	120
7	0.34	1,067	133	0.27	720
8	0.71	3,135	204	1.53	5,040
9	3.43	8,522	297	12.65	40,320
10	18.95	22,977	415	136.16	362,880
11	66.90	58,447	561	1,471.61	3,628,800
12	250.52	148,411	738		
13	2,162.82	361,047	949		

Table 2: Results for Pigeonhole problems

4.4 The Mutilated Checkerboard Problem

A classic problem in Artificial Intelligence is the *Mutilated Checkerboard Problem*: whether one can cover with dominoes an $n \times n$ checkerboard from which two squares from opposite corners have been removed [McCarthy, 1964]. Although there is an elegant proof of the impossibility of doing so in the general case,[6] here we focus only on the corresponding satisfiability problems for *particular* values of n.

The encoding we used assigns two propositions $H_{\langle x,y\rangle}$ and $V_{\langle x,y\rangle}$ to each board position $\langle x,y\rangle$, indicating whether there is a horizontal or vertical domino starting at position $\langle x,y\rangle$. It is clear that these two variables are directly constrained only by each other and corresponding variables for six neighboring positions. Thus, these constraints can be said to be *local*, unlike, for instance, those in the n-queens problem. The variable ordering was the usual lexicographic one, with the two variables for each board position being consecutive.

This "locality" makes it possible for relatively small OBDDs to represent large numbers of partial solutions to the covering problem. For instance, under the above ordering, an OBDD of size $5,681$ can represent the 1.29×10^7 different ways of covering the unmutilated 8×8 board. Indeed, OBDDs are remarkably superior to LDPP in this case, as Table 3 shows, even though time and space requirements are exponential. The OBDD strategy followed the variable ordering, consecutively adding all the constraints for each square. Most reasonable strategies yielded good performance; however, some that proved faster for $n \leq 12$ could not be completed for $n = 13$ because of OBDD exponential space requirements.

n	OBDD time	OBDD max size	clauses	LDPP branches	LDPP time
4	0.07	66	68	2	0.00
5	0.21	245	119	36	0.01
6	0.39	608	184	137	0.07
7	0.85	1,917	263	12,550	5.39
8	2.39	4,374	356	145,270	80.34
9	5.81	12,476	463	54,371,816	36,632.59
10	13.24	27,265	584		
11	33.75	72,800	719		
12	76.92	155,196	868		
13	293.58	395,752	1031		

Table 3: OBDD and LDPP results for the Mutilated Checkerboard problem

4.5 Quasigroup Existence Problems

The next class of problems we investigated is more substantial: quasigroup existence problems, some of which were open until recently settled with the assistance of automated reasoning programs, including the LDPP, DDPP and SATO implementations of the Davis-Putnam procedure (see [Fujita *et al.*, 1993, Slaney *et al.*, 1993]).

Briefly, the "QG5" quasigroup problem is finding an n by n multiplication table over the elements $\{1,\ldots,n\}$ that satisfies the equation $((ba)b)b = a$ for all elements a and b. An additional constraint is imposed that the table be *idempotent*, that is, that $a \cdot a = a$ for all a. These tables, when they exist, are also *Latin Squares*: there are no repeated entries in any row or column. Whether such quasigroups existed for $n \in \{9, 10, 12, 13, 14, 15\}$ were all open problems that have now been solved (in the negative) by LDPP and SATO (the case $n = 16$ has been solved by a program of R. Hasegawa and by SATO, and $n = 18$ is the next open one).

[6] Subramanian [1993] presents an interactive, mechanical verification of this theorem using NQTHM, the Boyer-Moore prover.

Two sets of extra constraints used to restrict the search in [Fujita *et al.*, 1993] and [Slaney *et al.*, 1993] were also of great help in finding solutions using OBDDs. The first reduces the number of isomorphic copies (but does not affect the existence of solutions) by assuming that $x \cdot n \neq z$ whenever $z + 1 < x$. The second set of constraints corresponds to the derived equations $b((ab)b) = a$ and $(b(ab))b = a$ for all a and b.

Building OBDDs from the clausal form of these problems results in very poor performance—at least in the absence of heuristics. Thus, we generated the OBDD constraints directly in a way analogous to that used for the n-queens puzzle of Section 4.1. Two encodings were used: in the first, n different Boolean variables represented the n possible values for each table entry; in the second, $\lceil \log_2 n \rceil$ variables were interpreted as the binary encoding of each entry. No variables were introduced for the diagonal elements, which have fixed values that, in turn, ruled out *a priori* two possible values for each nondiagonal entry. By reducing the number of OBDD variables, these encoding details improved performance (although the growth rates remained essentially the same).

Again, the main challenge in the OBDD case was to find a good variable ordering, a good decomposition of the constraints, and a good sequence for combining the pieces. Variable-ordering considerations similar to those discussed in Section 4.1 apply here—only now extended to three dimensions.[7] The variable ordering chosen was analogous to the one for the n-queens: we traverse the table row by row, from left to right; the Boolean variables that encode the value of each table entry are grouped together.

Large OBDD sizes occur when variables dependent on each other are far apart in the ordering (see [Hu and Dill, 1993]). In the variable ordering we used, constraints within rows will produce small OBDD sizes, while those within columns will generate large ones. Constraints that jump around the table will tend to produce large OBDDs. This poses a seemingly unavoidable problem in the case of QG5 constraints: placing one set of dependent variables together will separate another.

As in the n-queens case, the best OBDD strategy we found was incremental. Initially, an OBDD was created from the isomorphism-reducing constraint and the constraints that enforce a single value in each table entry. These two constraints follow the variable ordering, and are polynomial in n. Variables were then added one by one, together with all Latin Square constraints between them and previously introduced ones. The QG5 constraints, however, were treated specially: when each variable was introduced, all the QG5 constraints for that entry were added as well—adding only those that involved previously introduced variables did not work well. Table 4 shows the best OBDD results.

Strategy B was simpler: as before, we begin with the isomorphism-reducing constraint, and positive and negative constraints on the table entries; then the corresponding constraints were posted on the rows, and finally a nonredundant set of QG5 constraints was added (the constraint on columns is redundant after this). Table 5 compares the results for strategies A and B.

The clausal encoding for LDPP used the first encoding method above; using a bit-encoding is not useful in this case. In addition, the presence of literals for the diagonal values is harmless, since LDPP will effectively remove them at the start by branching first on the corresponding unit clauses. As with the n-queens problem, the growth rates are similar, but now OBDDs lag behind by a more significant amount. And again, memory limitations place OBDDs at a

[7] An argument similar to that in Theorem 4 gives exponential lower bounds on the OBDD size of some of the basic constraints. Note, however, that finding nontrivial lower bounds on the final OBDD size for these problems would imply answering the quasigroup existence questions that we are trying to solve in the first place.

disadvantage due to the exponential growth in the maximum OBDD size.

Figure 4 shows the OBDD size profile for Strategy A as applied to QG5.9. The same pattern is repeated for all smaller orders: at some point during the collection of the QG5 constraints, the OBDD size explodes exponentially.

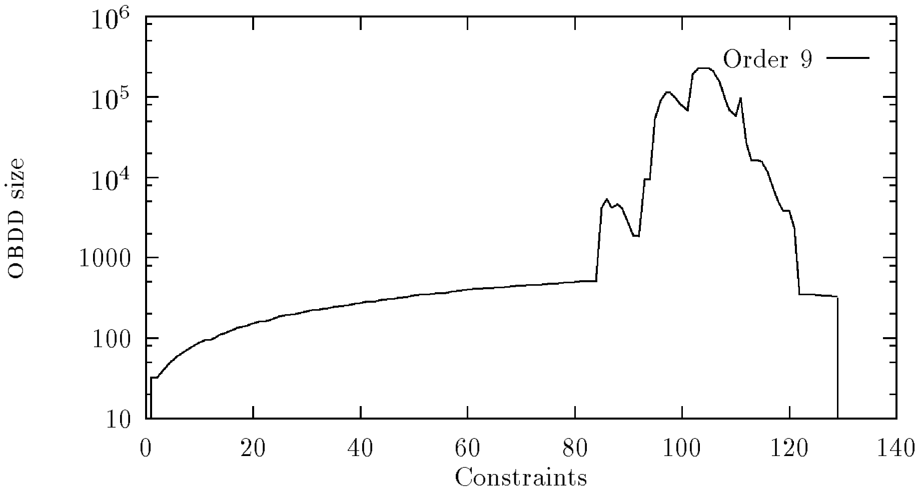

Figure 4: OBDD size for QG5.9 problem, Strategy A

Figure 5 shows the OBDD size for QG5.7 using Strategy B; the peaks correspond to the points when intermediate constraints are added to the main OBDD. The large peaks are for each of the 7 rows as the $((ba)b)b = a$ constraints for the elements of each row are added.

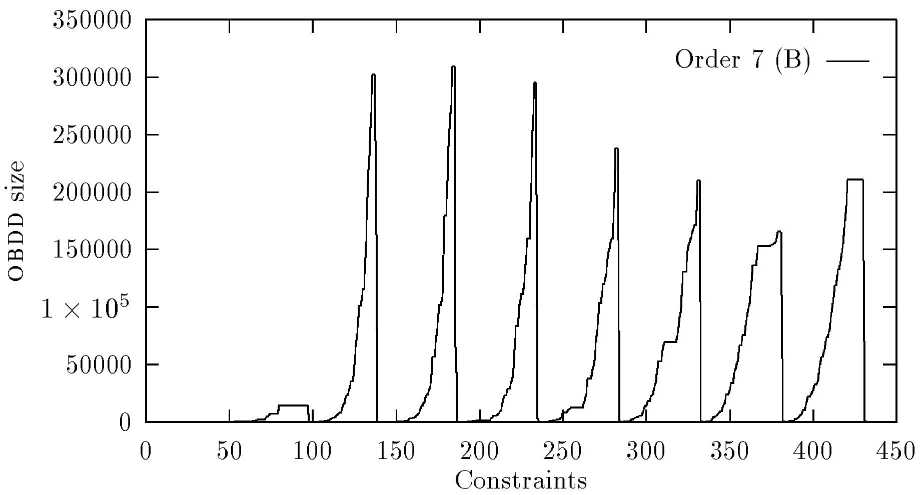

Figure 5: OBDD size for QG5.7 problem, Strategy B

We end this section by pointing out that OBDDs did not exhibit good performance on the

n	# solns.	OBDD final size	OBDD time	OBDD max size	clauses	LDPP search time	LDPP branches
4	0	1	0.27	48	583	0.01	1
5	1	62	2.19	140	1,436	0.02	1
6	0	1	2.82	433	3,004	0.03	1
7	3	314	21.59	1,225	5,608	0.13	5
8	1	226	108.46	16,678	9,629	0.23	11
9	0	1	716.64	228,596	15,508	0.52	19
10	0				23,746	1.77	62
11	5				34,904	9.01	230
12	0				49,603	42.89	1047
13	0				125,464 ⋆	2,094.56	14,784
14	0				169,030 ⋆	17,099.58	107,419

Table 4: Results for QG5 problems (a "⋆" indicates formulations that used extra, redundant clauses that reduce the Davis-Putnam search)

n	# solns.	OBDD final size	OBDD time A	OBDD max size A	OBDD time B	OBDD max size B
4	0	1	0.27	48	0.26	194
5	1	62	2.19	140	1.76	1,985
6	0	1	2.82	433	12.67	23,498
7	3	314	21.59	1,225	462.22	309,379
8	1	226	108.46	16,678		
9	0	1	716.64	228,596		

Table 5: Comparison of OBDD strategies for QG5 problems

ordered semigroup problems from [Slaney, 1993].

4.6 Boolean Equivalence Benchmarks

Finally, we compared the two approaches using the IFIP Boolean equivalence benchmarks from [Claesen, 1990], also discussed by Moore [1992]. These benchmarks can be described as lists of equations of the form $v = exp$, where v is a Boolean variable and exp is a Boolean expression. Expressions are built from the usual Boolean operators, previously defined variables, and a set of *input variables* that do not appear on the left-hand side of any equation.

The problem is to prove the equivalence of two sets of n variables, *i.e.*, show that $main = \neg((v_1 \leftrightarrow v_1') \wedge \ldots \wedge (v_n \leftrightarrow v_n'))$ is *false* for any input-variable truth assignment. Here, $\{v_1, \ldots, v_n\}$ and $\{v_1', \ldots, v_n'\}$ represent the outputs of two combinational circuits over the same inputs.

4.6.1 Clausal Form

While OBDDs accept (and welcome) input in terms of the Boolean operators used in the benchmarks, the Davis-Putnam procedure requires clausal form. For this, Boolean functions were rewritten in terms of $\wedge$, $\vee$ and $\neg$, and each equation $l = r$ was interpreted as $l \leftrightarrow r$. We introduced new "intermediate" variables to keep the clausal forms small; this is a simple instance of the transformation proposed in [1990]. Otherwise, the standard transformation produced an impractically large number of clauses. (Note, for instance, that if no new propositions are introduced then equivalences such as the definition of *main* above will produce a number of clauses exponential in n.) Equations of the form $v_1 = v_2$ or $v_1 = \neg v_2$ were used to simplify all other equations by applying the substitutions $\{v_1 \mapsto v_2\}$ or $\{v_1 \mapsto \neg v_2\}$ respectively, and then eliminated from the set. However, the clausal representation is still not nearly as compact as the original.

On the other hand, OBDDs do not seem well-suited to clausal form, in the absence of heuristics. For instance, we tested unsatisfiable random 3-SAT problems with a 4.29 clause-to-variable ratio, experimentally determined to generate the "hardest" such problems [Crawford and Auton, 1993]. For problems with 172 clauses over 40 propositions, the average OBDD time was 173.92 seconds, for an average maximum size of 245,305 nodes, while the average LDPP time was 0.19 second, with an average of 64.75 branches. The main obstacle for OBDDs is again the size of intermediate graphs, despite the unit size of the final one. Clearly, heuristics are needed to find good variable orderings and to decide how clauses should be combined.[8]

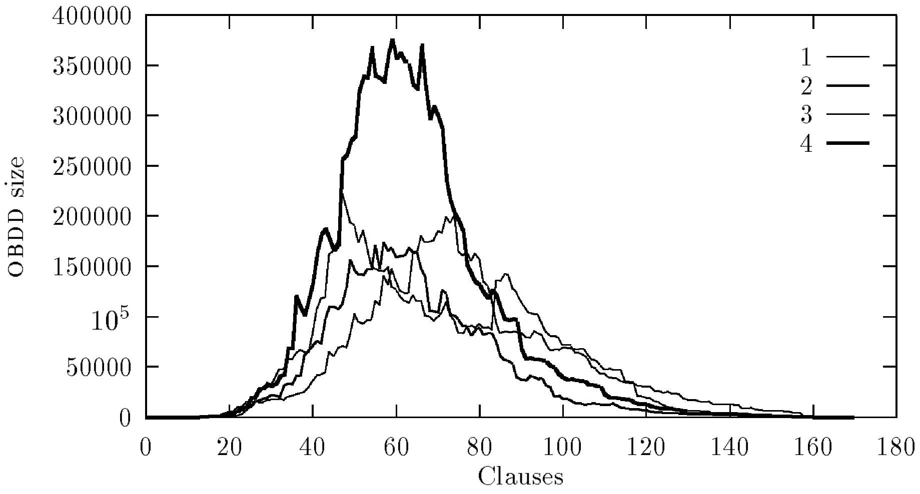

Figure 6: OBDD size for unsatisfiable random 40/172 3-SAT problems

Figure 6 shows the corresponding OBDD size profiles.

4.6.2 Problem Subdivision

Showing that the *main* variable defined above is always *false* is equivalent to n subproblems, each proving the equivalence of a pair $\langle v_i, v_i' \rangle$ of output variables. In the Davis-Putnam case, subdividing the main problem into these n subproblems and solving each one independently turned out to be more efficient: the sum of the times over all subproblems was usually significantly *smaller* than the time taken to solve all of them simultaneously. For each subproblem, only the relevant subsets of the complete circuits were used.

This was certainly *not* the case for OBDDs: the work done in showing the equivalence of one pair of outputs can be effectively reused in testing the equivalence of other pairs, through

[8]The OBDD strategy used, which conjoined clauses one by one to a main OBDD, did not fully exploit the main features of OBDDs, because of the small grainsize of clauses. A primitive trick that improved performance was to conjoin the OBDDs for each clause in a tree-like fashion, instead of adding each new clause into a single main OBDD; however, we did not investigate more elaborate decomposition schemes.

the sharing of subgraphs and cached results (after all, it is *one* pair of circuits that is being verified, not n independent ones). That is, the OBDD procedure is *incremental*. Also, as Moore [1992] points out, it is particularly easy to test the equivalence of two functions using OBDDs: simply determine if they have the same normal form, which can be done with a single pointer comparison in an efficient implementation.

Problem	OBDD time	OBDD max size	LDPP time	LDPP branches	LDPP all time	LDPP all branches
add1	0.16	55	0.07	294	0.22	711
add2	0.48	282	6.18	13,018	4.74	7,765
add3	1.54	1,622	11,405.60	1.23×10^7	—	—
add4	26.98	34,112	—	—	—	—
addsub	1.28	85	—	—	—	—
mul03	0.16	23	0.03	100	0.53	809
mul04	0.30	61	0.67	790	28.01	28,452
mul05	0.60	176	9.76	7,735	—	—
mul06	1.76	476	152.62	92,946	—	—
mul07	4.65	1,393	—	—	—	—
mul08	11.85	3,889	—	—	—	—
rip06	0.09	20	0.11	368	7.22	21,009
rip08	0.14	26	0.27	6,865	97.92	228,802

Table 6: IFIP Benchmark results

Table 6 summarizes the IFIP benchmark results. The "LDPP all" columns indicate LDPP performance when the equivalences were not split into subproblems. Clearly, these problems do not seem suitable for the Davis-Putnam procedure; even after subdividing the problems, most of them were beyond the reach of both SATO and LDPP, meaning that the problem of proving the equivalence of a particular pair of outputs proved too difficult.

No special OBDD variable orderings were used—the default was the order of appearance of variables in the input file—and with the relative exception of `add4`, OBDD sizes remained small. Note that the OBDD representation of intermediate functions remains always in terms of the input variables. The special status of input variables is lost in the transformation to clausal form; a good branching heuristic for Davis-Putnam would probably have to distinguish them again. [9]

In fairness to the Davis-Putnam procedure, we should note that LDPP was run using only its default branching heuristic (choosing a literal from a shortest positive clause) and simple variants thereof. We have not thoroughly investigated how changing this heuristic may improve Davis-Putnam performance—just as variable and constraint-ordering considerations improved the performance of OBDDs for the problems of Sections 4.1 through 4.5.

However, experience suggests that the Davis-Putnam procedure will often do no better on circuit problems than the naive approach of enumerating all 2^n combinations of the input variables and testing the equivalence of the output variables, but this is difficult to prove in general. As an example, though, it is simple to prove that the Davis-Putnam procedure, when testing equivalence of two definitions of the n-input even-parity function, must have at least

[9] Actually, when the clauses for these benchmarks are fed back to an OBDD procedure, the number of Boolean variables is now the number of atoms in the clausal form, and the OBDD performance on these sets of clauses is as bad as, if not worse than, that of LDPP.

2^n branches in its search space. Since parity is a symmetric function (see Prop. 3), this is trivial for OBDDs (if the circuits are represented in a reasonable way).

The n-input even-parity function is *true* iff an even number of its n arguments are *true*. It is easy to see that under any variable ordering, this function can be represented as an OBDD with $2n+1$ nodes [Bryant, 1986]. Let $e_1(I_1,\ldots,I_n)$ and $e_2(I_1,\ldots,I_n)$ be Boolean expressions over the propositions $\{I_1,\ldots,I_n\}$ that both compute the n-input even-parity function of $I_1,\ldots,I_n$ using no extra variables. Let O be a proposition distinct from $I_1,\ldots,I_n$. Let S_1 be a set of clauses equivalent to $O \equiv e_1(I_1,\ldots,I_n)$, let S_2 be a set of clauses equivalent to $\overline{O} \equiv e_2(I_1,\ldots,I_n)$, and let $S = S_1 \cup S_2$. Call S an *n-input e-test clause set*; proving that S is unsatisfiable demonstrates the equivalence of e_1 and e_2.

Theorem 5 *The Davis-Putnam procedure must case-split at least $2^n - 1$ times to prove the unsatisfiability of an n-input e-test clause set S.*

Proof: We use induction on n:
Basis: Let $n = 1$. Then S contains no unit clauses, since otherwise it would assign a truth-value to one of $I_1,\ldots,I_n,O$. Therefore, the Davis-Putnam procedure must case-split at least once (*i.e.*, $2^n - 1$ times).
Induction Step: Assume that $n > 1$ and that the theorem holds for values less than n. S is of the form $S_1 \cup S_2$ where S_1 is a set of clauses equivalent to $O \equiv e_1(I_1,\ldots,I_n)$ and S_2 is a set of clauses equivalent to $\overline{O} \equiv e_2(I_1,\ldots,I_n)$ for two n-input even-parity expressions e_1 and e_2. Because S contains no unit clauses, the Davis-Putnam procedure must case-split on some I_k or O.

- *Case $n > 1$, case-splitting on I_k*: on the case-splitting branch where I_k is assigned *true*,

$$\begin{aligned}
S_1(I_k \leftarrow \mathit{true}) &\equiv (O \equiv e_1(I_1,\ldots,I_{k-1},\mathit{true},I_{k+1},\ldots,I_n))\\
&\equiv (\overline{O} \equiv e_1'(I_1,\ldots,I_{k-1},I_{k+1},\ldots,I_n))\\
S_2(I_k \leftarrow \mathit{true}) &\equiv (\overline{O} \equiv e_2(I_1,\ldots,I_{k-1},\mathit{true},I_{k+1},\ldots,I_n))\\
&\equiv (O \equiv e_2'(I_1,\ldots,I_{k-1},I_{k+1},\ldots,I_n))
\end{aligned}$$

  where e_1' and e_2' are Boolean expressions for computing the $(n-1)$-input even-parity function. Thus, $S_1(I_k \leftarrow \mathit{true}) \cup S_2(I_k \leftarrow \mathit{true})$ is an $(n-1)$-input e-test clause set that requires, by induction, at least $2^{n-1} - 1$ case splits.

  The *false* branch likewise reduces S to an $(n-1)$-input e-test clause set. Therefore, the total number of case splits is at least $2^n - 1$.

- *Case $n > 1$, case-splitting on O*: on the case-splitting branch where O is assigned *true*,

$$\begin{aligned}
S_1(O \leftarrow \mathit{true}) &\equiv (\mathit{true} \equiv e_1(I_1,\ldots,I_n))\\
&\equiv (I_1 \equiv e_1(\mathit{true},I_2,\ldots,I_n))\\
&\equiv (\overline{I_1} \equiv e_1'(I_2,\ldots,I_n))\\
S_2(O \leftarrow \mathit{true}) &\equiv (\mathit{false} \equiv e_2(I_1,\ldots,I_n))\\
&\equiv (I_1 \equiv e_2(\mathit{false},I_2,\ldots,I_n))\\
&\equiv (I_1 \equiv e_2'(I_2,\ldots,I_n))
\end{aligned}$$

  where e_1' and e_2' are Boolean expressions for computing the $(n-1)$-input even-parity function. Thus, $S_1(O \leftarrow \mathit{true}) \cup S_2(O \leftarrow \mathit{true})$ is an $(n-1)$-input e-test clause set that requires, by induction, at least $2^{n-1} - 1$ case splits.

The *false* branch likewise reduces S to an $(n-1)$-input e-test clause set. Therefore, the total number of case splits is at least $2^n - 1$. ■

OBDDs are not *always* superior to Davis-Putnam in the circuit verification setting: consider, for instance, testing the equivalence of two *different* circuits that each have an exponential OBDD representation. A Davis-Putnam implementation may happen to detect the inequivalence of these circuits in less than exponential time (by finding a set of inputs for which a pair of corresponding outputs is different) while OBDDs are guaranteed to spend an exponential amount of time building the function representations before they can be compared. Also, the OBDD representation is not always more compact than clausal form: Devadas [1993] describes a class of functions with polynomial sum-of-products representation (so that their complements have polynomial clausal form) but with an exponential OBDD representation under any variable ordering.

5 Conclusions

In general, we can say that OBDDs are well-suited for representing large numbers of solutions that share a recursive structure (*e.g.*, the Mutilated Checkerboard, n-rooks and Pigeonhole problems), and for functional equivalence problems from the circuit domain. With the exception of the latter, the favorable performance of OBDDs often came at the expense of exponential memory usage that, for instance, crippled the ability of OBDDs to solve the n-queens problem (even though time performance was comparable to that of LDPP).

OBDDs proved impractical for solving highly constrained problems with few or no solutions, such as the quasigroup existence problems and hard random 3-SAT problems, that the Davis-Putnam procedure is good at. The Davis-Putnam procedure is also clearly superior if one wants only to find a single solution in a solution-rich search space. On the other hand, it was of practically no use in proving Boolean equivalences for which OBDDs are particularly adept.

The main obstacle to using OBDDs directly in constraint satisfaction problems is finding an adequate decomposition of the problem (including a good variable ordering). Our experimental results are inconclusive because of the large number of such decompositions—there is no guarantee that the OBDD strategies we used are the best possible.[10] Furthermore, obtaining bounds on OBDD performance appears to be a nontrivial problem in most interesting cases. Note that finding good decompositions is usually not a problem in the circuit domain—they are often already implicit in the circuit descriptions themselves.

In contrast to OBDDs, which need to be adapted to each particular problem, LDPP did quite well across many different problems using a single (and very simple) branching heuristic. Other Davis-Putnam heuristics, perhaps specific to the circuit domain, are yet to be explored; so are refinements that exploit symmetry or use more elaborate backtracking techniques.

5.1 Related and Future Work

Unfortunately, there seems to be little work directly related to that reported in this paper, which we hope will encourage more extensive experimental efforts and more detailed theoretical analyses in the future.

OBDDs have been successfully combined with interactive theorem-provers [Joyce and Seger, 1993], and have been found useful in Constraint Logic Programming languages that include Boolean constraints [Büttner and Simonis, 1987, Rauzy, 1994]. But except for the work of

[10] *Dynamic variable ordering* for OBDDs [Rudell, 1993] further expands the space of strategies. Experiments with the methods of Long's package [1993] yielded negative results.

A. Rauzy [1993], little comparative analysis of OBDDs has been done in this setting. Rauzy reports difficulties in the use of OBDDs for Boolean unification similar to those we encountered (namely, the problem of choosing a good variable ordering and the growth of intermediate OBDDs), and points out that enumerative methods can often be more efficient.[11]

Our results dramatize the distinction between search and constraint-solving that characterizes the CLP framework: the Davis-Putnam procedure can be described as all search and no constraint-solving, while OBDDs were used as all constraint-solving and no search. OBDDs do have many of the properties that have been found to be desirable in representing constraints: (1) unique normal forms that can concisely schematize a large number of solutions, and (2) incremental algorithms to compute normal forms for new constraints, including the caching of partial results for future use.

We have considered only *one* particular way of using OBDDs in classical automated deduction tasks, and there may be other, more fruitful uses. Similarly, numerous variants of the OBDD method have been recently proposed as more specialized alternatives for different tasks. In Minato's *zero-suppressed* OBDDs [1993], variables are assumed to have a default value of *false*. Multi-terminal binary decision diagrams (MTBDDs) are a generalization of OBDDs where terminal nodes correspond to the elements of an arbitrary finite domain (not just *true* and *false*) [Bahar *et al.*, 1993, Clarke *et al.*, 1993]. Multi-valued decision diagrams (MDDs) are a similar extension where branches can be more than binary [Kam and Brayton, 1990]. All these extensions seem well-suited, in particular, for the quasigroup problems, and such an application is left as another intriguing future experiment. So is the possibility of using OBDDs to replace clauses in a nonclausal version of the Davis-Putnam procedure, in the style of similar algorithms in [Jeong and Somenzi, 1992] and [Claesen, 1990]. Possega and Lüdascher [1992] present some preliminary results adapting the OBDD ideas to the first-order setting. Finally, we should mention the refinement of the OBDD idea recently proposed by Anuchitanukul and Manna [1994], which can achieve more compact representations by ignoring node labels when collapsing isomorphic subgraphs.

Another area for future work is to compare OBDDs with efficient implementations of other Boolean manipulation methods. For instance, Hsiang [1985] presents a system of rewrite rules that yields a canonical form, built using $\wedge$ and *xor* operations, for any Boolean expression. These were already used in [Chandrasekhar *et al.*, 1987] together with the Knuth-Bendix completion procedure to prove Boolean equivalences for circuit verification. To yield competitive performance, adequate caching and structure-sharing techniques could also be used.

Acknowledgements

Thanks to Alan Hu, Sanjeev Khanna, Liz Wolf and Jerry Yang, who provided many useful comments. Thanks also to Richard Waldinger, who suggested trying out the Mutilated Checkerboard problem.

References

[Anuchitanukul and Manna, 1994] Anuchit Anuchitanukul and Zohar Manna. Differential BDDs. Technical Report STAN-CS-TR-94-1525, Computer Science Department, Stanford University, Stanford, CA, September 1994.

[11] See [Benhamou and Colmerauer, 1993] for this and many other descriptions of Boolean constraint solving within the CLP scheme.

[Bahar *et al.*, 1993] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *IEEE International Conf. on Computer-Aided Design*, pages 188–191, November 1993.

[Benhamou and Colmerauer, 1993] Frédéric Benhamou and Alain Colmerauer, editors. *Constraint Logic Programming: Selected Research.* MIT Press, 1993.

[Boy de la Tour and Chaminade, 1990] Thierry Boy de la Tour and Gilles Chaminade. The use of renaming to improve the efficiency of clausal theorem proving. In Ph. Jorrand and V. Sgurev, editors, *Artificial Intelligence IV: Methodology, Systems, Applications (AIMSA'90)*, pages 3–12, Sofia, Bulgaria, 1990. Elsevier Science Pub. (North Holland).

[Brace *et al.*, 1990] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of the* 27^{th} *ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[Bryant, 1986] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Bryant, 1991] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[Bryant, 1992] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Burch *et al.*, 1992] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic modelchecking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Butler *et al.*, 1991] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M. Ray Mercer. Heuristics to compute variable ordering for efficient manipulation of ordered binary decision diagrams. In *Proc. of the* 28^{th} *ACM/IEEE Design Automation Conference*, pages 417–420, 1991.

[Büttner and Simonis, 1987] Wolfram Büttner and Helmut Simonis. Embedding Boolean expressions into logic programming. *Journal of Symbolic Computation*, 4(2):191–205, October 1987.

[Chandrasekhar *et al.*, 1987] Mandalagiri S. Chandrasekhar, John P. Privitera, and Kenneth W. Conradt. Application of term rewriting techniques to hardware design verification. In *Proc. of the* 24^{th} *ACM/IEEE Design Automation Conference*, pages 277–282, 1987.

[Claesen, 1990] Luc J.M. Claesen, editor. *Formal VLSI Correctness Verification—VLSI Design Methods*, volume II. Elsevier Science Pub. (North Holland), 1990. Chap. 2, *Efficient Tautology Checking Algorithms*.

[Clarke *et al.*, 1993] Edmund M. Clarke, Masahiro Fujita, Patrick C. McGeer, Kenneth McMillan, Jerry Chih-Yuan Yang, and Xudong Zhao. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. In *Proc. Intl. Workshop on Logic Synthesis*, Lake Tahoe, CA, May 1993.

[Crawford and Auton, 1993] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. of the* 11^{th} *National Conference on Artificial Intelligence*, pages 21–27, 1993.

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[Devadas, 1993] Srinivas Devadas. Comparing two-level and ordered binary decision diagram representations of logic functions. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):722–723, May 1993.

[Friedman and Supowitz, 1990] Steven J. Friedman and Kenneth J. Supowitz. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, May 1990.

[Fujita *et al.*, 1993] Masayuki Fujita, John Slaney, and Frank Bennett. Automatic generation of some results in finite algebra. In *Proc. of the* 13^{th} *International Joint Conference on Artificial Intelligence*, pages 52–57, 1993.

[Hsiang, 1985] Jieh Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 3(25):255–300, March 1985.

[Hu and Dill, 1993] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *Proc. of the* 30^{th} *ACM/IEEE Design Automation Conference*, pages 266–271, 1993.

[Jeong and Somenzi, 1992] Seh-Woong Jeong and Fabio Somenzi. A new algorithm for the binate covering problem and its application to the minimization of Boolean relations. In *IEEE International Conf. on Computer-Aided Design*, pages 417–420, 1992.

[Joyce and Seger, 1993] Jeffrey Joyce and Carl-Johan Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Proc. of the* 30^{th} *ACM/IEEE Design Automation Conference*, pages 469–474, 1993.

[Kam and Brayton, 1990] Timothy Y. K. Kam and Robert K. Brayton. Multi-valued decision diagrams. Technical Report UCB/ERL M90/125, University of California, Berkeley, December 1990.

[Long, 1993] David E. Long. *Model Checking, Abstraction, and Compositional Verification.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, July 1993.

[McCarthy, 1964] John McCarthy. A tough nut for proof procedures. AI Memo 16, Stanford University, July 1964.

[McMillan, 1993] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Minato, 1993] Shin-Ichi Minato. Zero-supressed BDDs for set manipulation in combinatorial problems. In *Proc. of the* 30^{th} *ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[Moore, 1992] J. Strother Moore. Introduction to the OBDD algorithm for the ATP community. Technical Report 84, Computational Logic, Inc., Austin, Texas, October 1992.

[Possega and Ludäscher, 1992] Joachim Possega and Bertram Ludäscher. Towards first-order deduction based on Shannon graphs. In *Proc.* 16^{th} *German Conf. on AI (GWAI-92)*, volume 671 of *Lecture Notes in Artificial Intelligence*, pages 67–76. Springer-Verlag, 1992.

[Rauzy, 1993] Antoine Rauzy. Using enumerative methods for Boolean unification. In [Benhamou and Colmerauer, 1993], chapter 13, pages 237–251. MIT Press, 1993.

[Rauzy, 1994] Antoine Rauzy. Notes on the design of an open Boolean solver. In *Proc. Intl. Conf. on Logic Programming*, 1994. To appear.

[Rudell, 1993] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE International Conf. on Computer-Aided Design*, pages 42–47, November 1993.

[Selman *et al.*, 1992] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proc. of the* 10^{th} *National Conference on Artificial Intelligence*, pages 440–446, San Jose, CA, July 1992.

[Simonis and Dincbas, 1993] Helmut Simonis and Mehmet Dincbas. Propositional calculus problems in CHIP. In [Benhamou and Colmerauer, 1993], chapter 15, pages 269–285. MIT Press, 1993.

[Slaney *et al.*, 1993] John Slaney, Masayuki Fujita, and Mark Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, 1993. To appear.

[Slaney, 1993] John Slaney. FINDER version 3.0 — notes and guide. Technical report, Centre for Information Science Research, Australian National University, 1993.

[Subramanian, 1993] Sakthi Subramanian. *A Mechanized Framework for Specifying Problem Domains and Verifying Plans.* Ph.D. Dissertation, Dept. of Comp. Science, U. of Texas, Austin, 1993.

[Wallace, 1993] Mark Wallace. Personal communication. ECRC, Munich, Germany, July 1993.

[Walsh, 1993] Toby Walsh. Personal communication. University of Edinburgh, October 1993.

[Zhang and Stickel, 1994] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, Dept. of Computer Science, University of Iowa, August 1994.

[Zhang, 1993] Hantao Zhang. SATO: A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, (22):1–3, March 1993.