Integrating symmetry breaking into a DLL procedure*

Chu Min Li and Bernard Jurkowiak
LaRIA, University of Picardie Jules Verne, 5 rue du moulin Neuf, 80000 Amiens, France
e-mail: {cli, jurkowia}@laria.u-picardie.fr

Paul W. Purdom Jr.
Computer Science Dept., Indiana University, 101 Lindly Hall, Bloomingon, In. 47405, USA
e-mail: pwp@cs.indiana.edu

1 Introduction

Many real-world problems have interesting symmetries. Encoding these problems into CNF formulas generally results in hard SAT problems. Examples includes the pigeon hole problem, which, although very simple, is nonpolynomial for any resolution-based method when encoded into SAT [6], and the n queen problem. In order to speed up search algorithms for these problems, intrinsic symmetries should be exploited to avoid repeated search of equivalent portions of search space.
The general strategy to exploit symmetries is to divide the objects of the search space into equivalence classes using symmetries, so that only one object in each class needs to be examined by a search algorithm. Brown, Finkelstein and Purdom [3] integrated symmetry-detection into a general-purpose backtracking search with dynamic variable search order, making intensive use of results from computational group theory. In the context of propositional reasoning, Benhamou and Sais [2] detected syntactic symmetries in a CNF formula and used these symmetries to reduce the search space. Crawford, Ginsberg and Luks [4] proposed a general schema to add symmetry-breaking axioms as CNF clauses into a CNF formula, which can be used as a pre-processor to any propositional reasoning method. In general case adding symmetry-breaking axioms appears to be intractable. Crawford, Ginsberg and Luks limited the number of added axioms to be polynomial. Shlyakhter [12] extended the approach of Crawford, Ginsberg and Luks by describing polynomial-size symmetry-breaking predicates for some common combinatorial objects.
In this paper we propose a generic symmetry-breaking schema in a DLL procedure [5], inspired from the approach of Brown, Finkelstein and Purdom. Like their approach, we exploit symmetries in every node of a search tree so that only one object in every equivalence class is searched by the DLL procedure. Unlike their approach, we are in propositional reasoning case and don't use results of computational group theory. We present our approach using two

*This work is supported by French CNRS under grant number SUB/2001/0111/DR16

hard search problems containing numerous symmetries. The experimental results show that even simple approaches to symmetry breaking can save a lot of time.
This paper is organized as follows. Section 2 presents the generic approach, which is illustrated by two examples discussed in Sections 3 and 4. Section 5 shows the experimental results. Section 6 concludes.

2 Generic approach

Let $\mathcal{F}$ be the CNF encoding of a problem $\mathcal{P}$. Our extended DLL is shown in figure 1.

procedure DLL($\mathcal{F}$)
Begin
```
if F is empty, return "satisfiable";

F:=ConstraintPropagation(F);
If F contains an empty clause, return "unsatisfiable".

F:=SymmetryBreaking(F);
If F contains an empty clause, return "unsatisfiable".

/* branching rule */
select a variable x in F according to a heuristic H, if the calling
of DLL(F ∪ {x}) returns "satisfiable" then return "satisfiable", oth-
erwise return the result of calling DLL(F ∪ {x̄}).
```
End.

Figure 1: A generic DLL Procedure with symmetry breaking

Constraint propagation in the extended DLL procedure includes the usual unit propagation and other look-ahead techniques such as equivalency reasoning [8].
The extended DLL procedure simultaneously works in the original problem representation $\mathcal{P}$ and its CNF encoding $\mathcal{F}$ with intensive interactions between the two spaces.
In fact, the successive branchings and constraint propagations give a partial assignment of boolean variables. The partial assignment corresponds to a partial object in the original problem represented by $\mathcal{P}$. The symmetry breaking procedure ensures that only the largest (or the smallest) object in each equivalence class is constructed and tested. As soon as the partial object is discovered impossible to be the largest, an empty clause is added into $\mathcal{F}$ and the extended DLL backtracks.
It is often necessary for the symmetry breaking procedure to fix some boolean variables to keep the possibility that the partial object is the largest, which considerably speeds up the extended DLL procedure.
We illustrate our approach by two examples.

3 Matrix multiplication

The obvious method for 2×2 matrix multiplication requires 8 multiplications. A method due to Strassen [13] computes the product of two 2×2 matrices in 7 multiplications not making use of commutativity. The principle of strassen's method is to use 7 terms. Each term is a product of two factors. Each factor is made by adding up some elements from one matrix. Then each of the four elements of the product matrix is the sum of the some of the 7 terms. The essential problem is how to form the 7 terms and how to form the 4 elements of the product matrix using the 7 terms.

More generally, we wish to compute the product C of two $N \times N$ matrices A and B using L multiplications. We denote by A_{ij} (B_{ij}, C_{ij}) the elements of A (B, C) and by P_l the L terms. We define the following boolean variables:

$$A_{ijl} = \begin{cases} 1 \text{ if } A_{ij} \text{ is used in term } P_l \\ 0 \text{ otherwise} \end{cases}$$

$$B_{ijl} = \begin{cases} 1 \text{ if } B_{ij} \text{ is used in term } P_l \\ 0 \text{ otherwise} \end{cases}$$

$$C_{ijl} = \begin{cases} 1 \text{ if } C_{ij} \text{ uses term } P_l \\ 0 \text{ otherwise} \end{cases}$$

Then for each $1 \leq i, j, k, s, m, n \leq n$, we generate the CNF clauses corresponding to the following equation:

$$\sum_{1 \leq l \leq L} A_{ijl} B_{ksl} C_{mnl} = \begin{cases} 1 \text{ mod } 2 \text{ if } j = k \wedge s = n \wedge i = m \\ 0 \text{ mod } 2 \text{ otherwise} \end{cases} \tag{1}$$

The intuitive reason of the equations can be stated as follows. If two elements A_{ij} of A and B_{ks} of B should occur in the two factors forming a term contributing to the element C_{mn} of C, and that $j \neq k$, $s \neq m$ or $i \neq n$, the product of A_{ij} and B_{ks} should occur an even number of times over all the L terms, so that they can be eliminated each other. Otherwise the product should occur an odd number of times so that one occurrence remains. For better discussion of the reasons see [7].

We now have a CNF formula $\mathcal{F}$ to be solved by our extended DLL procedure. A solution of $\mathcal{F}$ tells which elements of A and B are added up to form each term and which terms are added up to form different elements of C. Nevertheless, the sign of every component in every sum remains to be determined.

The searching for solving matrix multiplication constructs L terms and checks if they satisfy equation (1). All possible L terms are tried until equation (1) is satisfied.

Given a set of L terms, we know that each term is symmetric to another, i.e. if one permutation of the L terms doesn't satisfy equation (1), then no permutation of these terms can be a solution. These $L!$ permutations are equivalent. Only one permutation should be tried.

For this purpose, we construct L boolean variable vectors corresponding to a permutation of a set of L terms.

$$\begin{aligned}
&V_1 = A_{111}A_{121}...A_{1n1}A_{211}...A_{2n1}...A_{nn1}B_{111}...B_{nn1} \\
&V_2 = A_{112}A_{122}...A_{1n2}A_{212}...A_{2n2}...A_{nn2}B_{112}...B_{nn2} \\
&V_3 = A_{113}A_{123}...A_{1n3}A_{213}...A_{2n3}...A_{nn3}B_{113}...B_{nn3} \\
&\vdots \\
&V_L = A_{11L}A_{12L}...A_{1nL}A_{21L}...A_{2nL}...A_{nnL}B_{11m}...B_{nnL}
\end{aligned}$$

Each vector corresponds to a term and becomes a binary number when the variables are fixed. The binary number uniquely determines a term. We impose an order among the binary numbers: $V_1 > V_2 > ... > V_L$. The symmetry breaking consists in checking the order before every branching. If there exist two (partially instantiated) variable vectors V_i and V_j such that $V_i < V_j$ and $i < j$, the DLL procedure should backtrack. Thus only one permutation of the L terms are tried.
Note that the order checking may also fix some variables. For example, if A_{121} is assigned 0, A_{122} is assigned 1, A_{111} and A_{112} are free, the order checking should assign 1 to A_{111} and 0 to A_{112}, since no other possibility can keep the order $V_1 > V_2$. The value of these variables are of course propagated to simplify the $\mathcal{F}$. The order checking continues until no variable can be fixed and no vector order is found violated.
The time spent for the order checking is negligible. Our approach would save a factor of L! in time, especially for unsatisfiable problems, since without symmetry-breaking, all search space would be explored.
Other symmetries can be broken in the same way, which awaits future research.

4 Polynomial multiplication

The straitforward method of multiplying two polynomials of degree-bound N should compute N^2 products. Using the same principle as Strassen's method for matrix multiplication, we can reduce the number of products. For example, the multiplication of $(A_0+A_1x)\times(B_0+B_1x)$ can be computed using three products $A_0 * B_0$, $A_1 * B_1$ and $(A_0 + A_1) * (B_0 + B_1)$. We wish to compute the product of two polynomials of degree-bound N A and B using L multiplications. We denote by A_i (B_j, C_k) the coefficient of x^i (x^j, x^k) in A (B, C) and by P_l the L terms. We define the following boolean variables:

$$A_{il} = \begin{cases} 1 \text{ if } A_i \text{ is used in term } P_l \\ 0 \text{ otherwise} \end{cases}$$

$$B_{jl} = \begin{cases} 1 \text{ if } B_j \text{ is used in term } P_l \\ 0 \text{ otherwise} \end{cases}$$

$$C_{kl} = \begin{cases} 1 \text{ if } P_l \text{ is used in term } C_k \\ 0 \text{ otherwise} \end{cases}$$

For the convenience of this paper, we assume that A_{il}=B_{il} for all $0 \leq i < n$ and $0 \leq l < L$. Then for each $0 \leq i, j < n$ and $0 \leq k < 2 * n - 1$ we generate CNF clauses corresponding to the equation:

$$\sum_{1 \leq l \leq L} A_{il} B_{jl} C_{kl} = \begin{cases} 1 \bmod 2 \text{ if } i + j = k \\ 0 \bmod 2 \text{ otherwise} \end{cases} \quad (2)$$

We put all clauses into a formula $\mathcal{F}$ to be solved by our extended DLL procedure. The L terms are symmetric as in matrix multiplication. Only one permutation of a set of L terms should be tried. The L variable vectors corresponding to the L terms are

$$V_1 = A_{01}A_{11}...A_{N-1,1}$$
$$V_2 = A_{02}A_{12}...A_{N-1,2}$$
$$V_3 = A_{03}A_{13}...A_{N-1,3}$$
$$\vdots$$
$$V_L = A_{0L}A_{1L}...A_{N-1,L}$$

The order checking for the L vectors is similar as in matrix multiplication case. A factor of $L!$ in time would be saved.
Recall that the multiplication of two polynomials can be done in $O(nlog(n))$ using FFT, if all coefficients are real or complex numbers. Otherwise, minimizing the number of products for the multiplication using the above approach might be interesting.

5 Experimental results

The CNF encoding of matrix and polynomial multiplication involves modulo 2 arithmetics and contains a lot of equivalency clauses. So we integrate our symmetry-breaking into the DLL procedure $EqSatz$ [8] which is specialized in solving instances containing equivalency clauses. We denote the extended DLL procedure by sEqSatz.
We compare *sEqSatz* with others efficient solvers such as Satz [9], EqSatz, Relsat [1], zChaff [10] and Walksat [11]. The experimental results are obtained on a Pentium IV 1800 MHZ with 256 Mb of RAM under linux. We use default options for all solveurs. All times are in seconds. We use "-" to say that Walksat is not executed for unsatisfiable instances and $\diamond$ to say that Walksat doesn't find a solution for a satisfiable instance within 100000 flips and 10 tries. "?" is reported when zChaff returns "Out of memory".

5.1 Experimental results for matrix multiplication

Table 1 shows the performance of different solveurs on matrix multiplication. The name $MatNL$ denotes a CNF instance encoding the multiplication of two matrices $N \times N$ using L products. We also give the number of variables (#var), the number of clauses (#clause) and the satisfiability (sat) of each instance.

Instances	#var	#clause	sat	Satz	EqSatz	Relsat	zChaff	Walksat	sEqSatz
$Mat25$	588	1968	N	1511	657	19720	12	-	34
$Mat26$	744	2464	N	>36000	>36000	>36000	?	-	1462
$Mat27$	900	2960	Y	>36000	>36000	>36000	<1	$\diamond$	15419

Table 1: *Run time of Satz, EqSatz, Relsat, zChaff, Walksat and sEqSatz on matrix multiplication*

These are relatively small instances. Note that though zChaff is faster than sEqSatz on $Mat25$, sEqSatz performs better when the symmetry group becomes larger, although zChaff impressively finds a solution for $Mat27$.

5.2 Experimental results for polynomial multiplication

Similar to table 1, table 2 shows the performance of different solveurs on polynomial multiplication. The name $PolNL$ denotes a CNF instance encoding the multiplication of two polynomials of degree-bound N using L products.

Instances	#var	#clause	sat	Satz	EqSatz	Relsat	zChaff	Walksat	*sEqSatz*
$Pol23$	69	192	Y	<1	<1	<1	<1	<1	<1
$Pol35$	415	1350	N	<1	1	6	<1	-	<1
$Pol36$	525	1692	Y	<1	<1	8	<1	1	<1
$Pol48$	1704	5760	N	>7200	>7200	>7200	?	-	<1
$Pol49$	1959	6592	Y	>7200	>7200	>7200	<1	◇	<1
$Pol512$	5253	18000	N	>7200	>7200	>7200	?	-	1298
$Pol513$	5747	19650	Y	>7200	>7200	>7200	?	◇	28

Table 2: *Run time of Satz, EqSatz, Relsat, zChaff, Walksat and sEqSatz on polynomial multiplication.*

6 Conclusion

We propose a generic symmetry-breaking schema into a DLL procedure. The extended DLL procedure works simultaneously in the original problem representation and in its CNF encoding, with intensive communication between two search spaces. We impose an order among symmetric objects represented by boolean variable vectors. The symmetries are then broken by dynamically checking the order before every branching.
In the future, we will study the use of more symmetries, which would save a lot more time.

Acknowledgement: We thank G. Richomme for fruitful discussions.

References

[1] R.J. Bayardo and R.C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *proceedings AAAI'97*, pages 203–208, 1997.

[2] B. Benhamou and L. Sais. Tractability through symmetries in prositional calculus. In *Journal of automated reasoning,*, pages 89–102, 1994.

[3] C.A. Brown, L. Finkelstein, and P.W. Purdom. Backtrack searching in the presence of symmetry. In *Applied algebra, algebraic algorithms and error correcting codes, 6th international conference*, pages 99–110, 1988.

[4] J.M. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-Breaking Predicates for Search Problems. In *proceedings Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of ACM*, pages 394–397, 1962.

[6] A. Haken. The intractability of resolution. In *Theoretical Computer Science 39*, pages 297–308, 1985.

[7] J.E. Hopcroft and L.R. Kerr. On minimizing the number of multiplications necessary for matrix multiplication. In *Appl Math*, 1971.

[8] C.M. Li. Integrating Equivalency Reasoning into Davis-Putman Procedure. In *Proceedings of the AIII-2000*, pages 291–296, Austin, USA, August 2000.

[9] C.M. Li and Anbulagan. Heuristic based on unit propagation for satisfiability problems. In *Proceedings IJCAI'97*, pages 366–371, Nagoya, Japan, August 1997.

[10] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Sat'2001*, pages 278–283, Boston, USA, June 2001.

[11] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings AAAI-94*, 1994.

[12] L. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Sat'2001*, pages 34–45, Boston, USA, June 2001.

[13] V. Strassen. Gaussian elimination is not optimal. In *Numerische Mathematik*, pages 354–356, 1969.