Using Rewarding Mechanisms for Improving Branching Heuristics

Elsa Carvalho[1] and João Marques-Silva[2] *

[1] Mathematics Department, Madeira University,
Madeira, Portugal
ecarvalho@uma.pt
[2] Technical University of Lisbon,
IST/INESC-ID, Lisbon, Portugal
jpms@sat.inesc-id.pt

Abstract. The variable branching heuristics used in the most recent and most effective SAT solvers, including zChaff and BerkMin, can be viewed as consisting of a simple mechanism for rewarding the variables participating in conflicts during the search process. In this paper we propose to extend the simple rewarding mechanism used in zChaff and BerkMin, and develop different rewarding mechanisms based on information provided by the search algorithm, namely the size of learned clauses and the size of the backjumps in the search tree. The results show that very significant gains can be obtained for real-world problem instances of SAT.

1 Introduction

The architecture of a modern backtrack search SAT solver is composed of three main engines [7, 8]: the decision engine, the deduction engine and the diagnosis engine. The decision engine selects the variables to branch on and the values to assign to those variables. The deduction engine identifies necessary assignments, and most often corresponds to Boolean Constraint Propagation. Finally, the diagnosis engine handles conflicts, and most often implements clause learning and non-chronological backtracking. Besides the three main engines, a SAT solver usually utilizes efficient data structures, for representing clauses. The overall search algorithm can also utilize a number of additional techniques, including applying search restarts, and applying clause deletion policies [2, 1].

Over the last years, backtrack search SAT solvers have evolved significantly, either by improving each of the basic engines, by developing new supporting data structures, and by integrating new additional techniques.

In terms of the decision engine, different variable branching heuristics have been proposed. These heuristics can be broadly categorized as *static*, *dynamic* or *semi-dynamic*. In static variable branching heuristics the order in which variables are selected is established before the search algorithm begins [3, 4]. This order of the variables is not affected by the subsequent search. Static variable branching heuristics have the advantage of having negligible cost during the search, but the key drawback of being ineffective for large practical problem instances. In dynamic variable branching heuristics, the variable that is selected at each step is decided given the current status of the search process. Dynamic variable branching heuristics range from simple literal counting procedures, applied to each literal in the formula [6], to more elaborate approaches based on probing assignments and evaluating the number of implied assignments [5]. Dynamic variable branching heuristics have the advantage of yielding smaller search trees, but the key drawback of requiring significant computational effort at each node in the search tree. In recent years, the most effective backtrack search SAT solvers proposed the utilization of semi-dynamic variable branching heuristics [8, 1]. In these heuristics, the information associated with variables is not static, but it is also not updated at each node in the search tree. In general, semi-dynamic variable branching heuristics update the heuristic information of a few variables after each conflict, and perform a

* This work is partially supported by the European research project IST-2001-34607 and by Fundação para a Ciência e Tecnologia under research projects POSI/CHS/34504/2000 and POSI/SRI/41926/2001.

global update after every N decisions, where N is a significantly large integer. There are a few key advantages to semi-dynamic variable branching heuristics. First, these heuristics introduce very little overhead. Second, the updates to the heuristic value of each variable are guided by information provided by conflicts identified during the search process, and so the search process effectively focuses on the variables that are relevant to the conflicts identified during the search process.

For recent semi-dynamic branching heuristics, and each time a conflict is identified, the heuristic value of each of a set of variables associated with the conflict is incremented by a fixed value, in general 1 [8, 1]. This increment can be viewed as a (constant) *reward*, given to a set of variables involved in conflicts.

In this paper we propose to extend this simple rewarding mechanism, independently of the actual semi-dynamic variable branching heuristic being utilized. The objective of extending the rewarding mechanism is to reward more the variables that yield significant savings in the search space, by contributing to large backjumps, or that are involved in creating *small* learned clauses. The results obtained on a representative class of real-world problem instances indicate that a more generalized rewarding mechanism can be very effective in reducing the search space.

The remainder of the paper is organized as follows. In the next section we review two well-known branching heuristics, VSIDS used in zChaff [8] and the heuristic used in BerkMin [1]. Afterwards, in Section 3 we detail the rewarding mechanisms proposed in the paper. Results are presented and analyzed in Section 4, and the paper concludes in Section 5.

2 Semi-Dynamic Variable Branching Heuristics

2.1 zChaff's VSIDS Heuristic

The branching heuristic proposed in zChaff implements a semi-dynamic decision strategy called *Variable State Independent Decaying Sum* (VSIDS). This heuristic has proved to be adequate for real-world problem instances, being also characterized by having a small computational cost.

The VSIDS heuristic considers variables that participate in conflicts to be more relevant to the search process than others. In the implementation of the VSIDS heuristic, all variables that are included in each learned clause are declared to be involved in the conflict. Besides this dynamic component, VSIDS also includes a static component that counts the number of occurrences of each literal in the initial formula. This value corresponds to the initial score of each literal when the search begins. Then the score is updated dynamically in the following way:

- Each time a conflict clause is created the score of each literal that participates in the clause is incremented by one.
- After every N conflicts the score of each literal is *aged*, by dividing it by a small constant. In this way the literals that do not participate in conflicts for a long time are penalized.

2.2 Berkmin's Heuristic

Given the promising results of the VSIDS heuristic, the authors of BerkMin [1] further extended this heuristic in two major ways:

- First, all clauses that participate in the implication sequence resulting in a conflict are declared to be involved in the conflict, and so their score is also incremented. This modification can be viewed as a way of rewarding *all* variables that are effectively involved in contributing to each conflict.
- Second, the heuristic of BerkMin gives preference to variables that occur in recently recorded clauses.

3 Rewarding Mechanisms

The main motivation of rewarding mechanisms is to extend one of the key aspects of zChaff's VSIDS heuristic, namely the reward of variables involved in conflicts, which become more likely

Reference	$1 \leq \#lits \leq 50$	$\#lits > 50$
RC	4	1

Table 1. Rewards based on clause size

Reference	$bjmp = 1$	$2 \leq bjmp \leq 20$	$bjmp > 20$
RJ	1	2	4

Table 2. Rewards based on backjump size

to be used by the variable branching heuristic. Whereas in zChaff and Berkmin the reward is uniform for every conflict, in this paper we propose to reward variables differently, depending on the potential impact of the conflict for the subsequent search. Existing data indicates that two measures of success of a SAT solver are the size of learned clauses, the smaller the better, and the size of backjumps, the larger the better [7]. As a result, we propose to reward variables accordingly. Variables involved in a conflict from which a small clause is learned or from which a large backjump is obtained are rewarded more than variables involved in a conflict from which a large clause is learned and the resulting obtained backjump is small. The rewards that are used in the proposed variable branching heuristic are shown in Tables 1 and 2. The numbers shown in the tables are intended to be indicative. We have used these rewards for the experiments shown in the paper, but other values could have been considered, allowing more aggressive rewards of large backjumps and very small clauses.

Given the proposed rewarding scheme, we can consider the sole application of rewards based on learned clauses, which we refer to as RC. We can consider the sole application of rewards based on backjump size, which we refer to as RJ. Finally, we can consider the integrated application of rewards based both on learned clauses and backjump size, which we refer to as RCJ.

Initially, variables occurring in smaller clauses are rewarded more than variables occurring in larger clauses. Whenever a clause is learned, if its size is small enough, then the variables associated with the learned clause are more rewarded. Finally, if the backjump that results from analyzing a conflict and learning a clause is large enough, then the variables associated with the learned clause are more rewarded.

4 Results

The rewarding mechanisms were integrated in the zChaff code [8]. In order to evaluate the effectiveness of utilizing rewarding mechanisms, a large set of real-world problem instances was considered. These instances represent practical formal verification examples. The classes of instances considered are shown in Table 3. For our evaluation we just considered classes of instances where the current version of zChaff can solve the majority of the existing problem instances. The evaluation of rewarding mechanisms is restricted to the VSIDS heuristic [3].

The results of applying the different rewarding mechanisms are shown in Table 3. In this table, column TO means Time Out and represents the number of instances that exceeded the allowed running time (1800 seconds). All the results were obtained in a P IV @ 1.7GHz machine with 1GByte of RAM.

As can be concluded, the utilization of rewards is clearly useful, allowing significant gains, both in terms of the total CPU time and in terms of the number of aborted instances. For the branching heuristic using both clause and backjump rewards, the run times decrease by more than a third with respect to the plain VSIDS heuristic. The number of aborted instances decreases by two thirds with respect to the plain VSIDS heuristic.

Table 4 shows, for each class of instances, the number of instances for which the results are better, worse or equal when comparing the plain VSIDS heuristic with the VSIDS heuristic which

[3] Since the BerkMin code is not publicly available, it is not possible to modify its heuristic to also include rewarding mechanisms. An alternative implementation of the BerkMin's heuristic is not guaranteed to allow reproducing the same results.

Class	VSIDS	TO	RC	TO	RJ	TO	RCJ	TO
IBM 01 rule	7000	2	11204	2	5966	1	8318	1
IBM 02_1 rule 1	11213	3	8992	1	9446	2	7038	0
IBM 02_1 rule 2	11548	3	10214	2	10289	2	8776	1
IBM 02_1 rule 3	2021	0	1614	0	1464	0	1178	0
IBM 02_1 rule 4	1873	0	1347	0	1246	0	1098	0
IBM 02_1 rule 5	2404	0	2144	0	1761	0	1104	0
IBM 02_2 rule	4509	0	3750	0	3775	0	2848	0
IBM 02_3 rule 1	7216	1	6034	0	1838	0	4090	0
IBM 02_3 rule 2	14910	4	6100	0	7375	1	5441	1
IBM 02_3 rule 3	6520	1	7422	0	3361	0	3726	0
IBM 02_3 rule 4	11558	4	5887	0	11516	1	4390	0
IBM 02_3 rule 5	8491	2	5656	0	5931	1	4085	0
IBM 02_3 rule 6	14174	5	6799	0	8012	1	3744	0
IBM 03 rule	9713	4	9124	3	8620	2	8712	1
IBM 04 rule	10695	3	14381	6	13101	6	10560	4
IBM 05 rule	1308	0	1075	0	966	1	1239	0
IBM 06 rule	9879	1	8915	3	9238	2	6825	1
IBM 07 rule	363	0	249	0	278	0	231	0
IBM 09 rule	96	0	71	0	18	0	46	0
IBM 14_1 rule	3285	0	1292	0	1086	0	989	0
IBM 19 rule	1717	0	2723	0	2987	0	2019	0
IBM 21 rule	1035	0	912	0	763	0	1339	0
IBM 26 rule	481	0	787	0	697	0	1201	0
IBM 27 rule	1522	0	1892	0	1787	0	1749	0
IBM 28 rule	8988	1	7479	1	6263	1	7434	1
TOTAL	152519	34	126063	18	117784	21	98180	10

Table 3. Execution times for the different rewarding mechanisms, RC, RJ and RCJ

uses rewards on clauses and on backjumps. As can be observed, the number of instances for which the results are better with rewards is significantly larger than the number of instances for which the results are worse when rewards are used.

Finally, Figure 1 shows the percent variation in run times between the plain VSIDS heuristic and the utilization of rewarding mechanisms, both for clauses and for backjumps. As can be concluded, for the vast majority of problem instances, rewards introduce significant gains. Moreover, the number of classes for which reward mechanisms result in worse run times is small and negligible, with one exception.

5 Conclusions

This paper proposes the utilization of rewarding mechanisms as an effective improvement to existing variable branching heuristics. The experimental results indicate that the utilization of rewarding mechanisms can be extremely effective for specific classes of problem instances, namely instances associated with the formal verification of processors.

A few lines of additional research can be outlined. First, the utilization of more aggressive rewards, that reward more larger backjumps or very small learned clauses. Second, the utilization of more rewarding mechanisms, e.g. the number of implied assignments and the depth at which each conflict occurs. Finally, the utilization of rewarding mechanisms with other variable branching heuristics and with other representative classes of instances may provide further insights into the practical usefulness of this technique.

References

1. E. Goldberg and Y. Novikov. BerkMin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.

Class	#Better	#Worse	#Equal
IBM 01 rule	4	11	5
IBM 02_1 rule 1	14	1	5
IBM 02_1 rule 2	15	0	5
IBM 02_1 rule 3	10	4	6
IBM 02_1 rule 4	8	6	6
IBM 02_1 rule 5	12	4	4
IBM 02_2 rule	12	4	4
IBM 02_3 rule 1	9	8	3
IBM 02_3 rule 2	17	1	2
IBM 02_3 rule 3	12	5	3
IBM 02_3 rule 4	17	1	2
IBM 02_3 rule 5	11	5	4
IBM 02_3 rule 6	18	1	1
IBM 03 rule	8	4	8
IBM 04 rule	6	8	6
IBM 05 rule	8	4	8
IBM 06 rule	11	3	6
IBM 07 rule	19	0	1
IBM 09 rule	7	3	10
IBM 14_1 rule	14	1	5
IBM 19 rule	5	9	6
IBM 21 rule	6	9	5
IBM 26 rule	0	17	3
IBM 27 rule	8	7	5
IBM 28 rule	12	4	4
TOTAL	263	120	117

Table 4. Class by class analysis of the number of instances with improved, worse and equal results

2. C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence*, pages 431–437, July 1998.
3. J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.
4. R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
5. C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 341–355, October 1997.
6. J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In P. Barahona and J. Alferes, editors, *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, volume 1695 of *Lecture Notes in Artificial Intelligence*, pages 62–74. Springer-Verlag, September 1999.
7. J. P. Marques-Silva and K. A. Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
8. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.

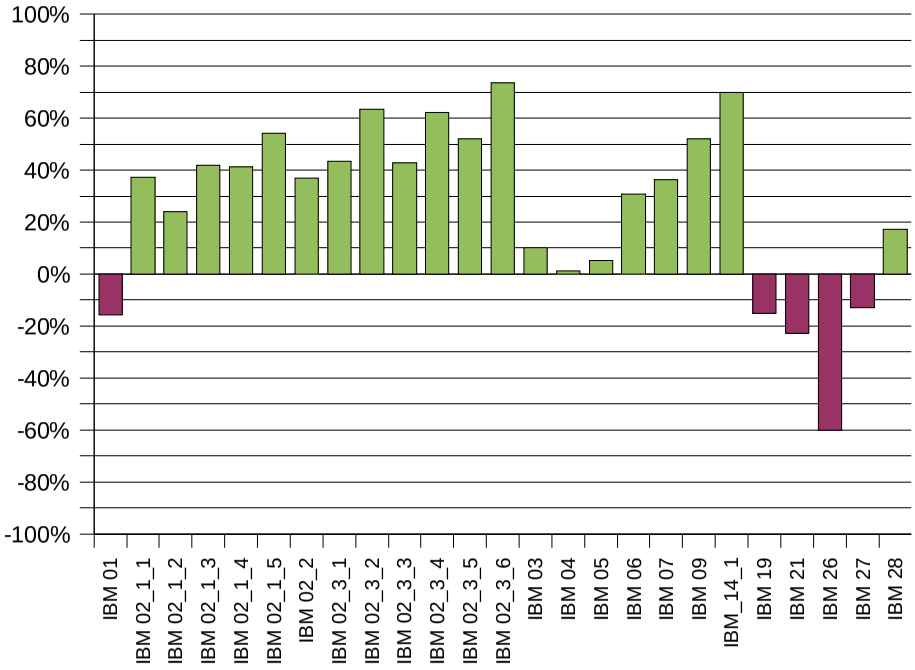

Fig. 1. Percent variation between VSIDS and RCJ