Assignment Stack Shrinking

Alexander Nadel[1] and Vadim Ryvchin[1,2]

[1] Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
{alexander.nadel,vadim.ryvchin}@intel.com
[2] Information Systems Engineering, IE, Technion, Haifa, Israel

Abstract. Assignment stack shrinking is a technique that is intended to speed up the performance of modern complete SAT solvers. Shrinking was shown to be efficient in SAT'04 competition winners Jerusat and Chaff. However, existing studies lack the details of the shrinking algorithm. In addition, shrinking's performance was not tested in conjunction with the most modern techniques. This paper provides a detailed description of the shrinking algorithm and proposes two new heursitics for it. We show that using shrinking is critical for solving well-known industrial benchmark families with the latest versions of Minisat and Eureka.

1 Introduction

Modern SAT solvers are known to be extremely efficient on many industrial problems which may comprise up to millions of variables and clauses. Among the key features that enable the solvers to be so efficient, despite the apparent difficulty of solving huge instances of NP-complete problems, are *dynamic behavior* and *search locality*, that is, the ability to maintain the set of assigned variables and recorded clauses relevant to the currently explored space. This effect is achieved by applying various techniques, such as the VSIDS decision heuristic [1] (which gives preference to variables that participated in recent conflict clause derivations) and local restarts (such as [2]). Another important feature of modern SAT solvers is that they tend to pick *interrelated variables*, that is, variables whose joint assignment increases the chances of quickly reaching conflicts in unsatisfiable branches and satisfying clauses in satisfiable branches. Clause-based heuristics (such as CBH [3]), which prefer to pick variables from the same clause, increase the interrelation of the assigned variables.

Assignment stack shrinking (or, simply, shrinking) is a technique that seeks to boost the performance of modern SAT solvers by making their behavior more local and dynamic, as well as by improving the interrelation of the assigned variables.

Shrinking was introduced in [4] and implemented in the Jerusat SAT solver. After a conflict, Jerusat applies shrinking if its *shrinking condition* is satisfied. The shrinking condition of Jerusat is satisfied if the conflict clause contains no more than one variable from each decision level. The solver then sorts the conflict clause literals according to its *sorting scheme*. The sorting scheme of Jerusat sorts the clause by decision level from lowest to highest. Afterwards Jerusat backtracks to the *shrinking backtrack level*. The shrinking backtrack level for

Jerusat is the highest possible decision level where all the literals of the conflict clause become unassigned. Jerusat then guides the decision heuristic to select the literals of the conflict clause according to the sorted order and assign them the value false, whenever possible. As usual, Boolean Constraint Propagation (BCP) follows each assignment.

One can pick out three important components of the shrinking algorithm that can be tuned heuristically: the shrinking condition, the sorting scheme, and the determination of the shrinking backtrack level. Shrinking was implemented in the 2004 version of the Chaff SAT solver [5] with important modifications in each one of these components, as described below.

2 Algorithmic Details and New Heuristics

Chaff had two versions: *zchaff.2004.5.13* and *zchaff_rand*. We concentrate on *zchaff_rand*'s version of shrinking, since it was shown to be more useful in [5], and also performed better in the SAT'04 competition [6]. Suppose Chaff encounters a conflict. Chaff considers applying shrinking if the length of the conflict clause exceeds a certain threshold x. The clause is sorted according to decision levels. The algorithm finds the lowest decision level that is less than the next higher decision level by at least 2. (If no such decision level is found, shrinking is not performed.) The algorithm backtracks to this decision level, and the decision strategy starts reassigning the value false to the unassigned literals of the conflict clause, whenever possible. Chaff reassigns the variables in the reverse order, that is, in descending order of decision levels, since this sorting scheme was found to perform slightly better than Jerusat's in [5]. The threshold value x for applying shrinking is set dynamically using some measured statistics. More specifically, Algorithm 1 is used in Chaff for adjusting x after every y conflicts. Chaff measures the mean and standard deviation of the lengths of the recently learned conflict clauses and tries to adjust x to keep it at a value greater than the mean. The threshold on the number of conflicts y is 600 for Chaff.

Chaff's shrinking algorithm was implemented in Intel's SAT solver Eureka with two minor differences: (1) The threshold on the number of conflicts y is 2000; (2) Eureka forbids performing shrinking for two conflicts in a row.

An important detail for understanding the reasons for the efficiency of shrinking is that a conflict clause is recorded even when shrinking is applied. Hence the solver always explores a different subspace after performing shrinking. Previous works [4, 5, 7] claimed that a "similar" conflict must follow an application of shrinking, on the assumption that a conflict clause is not recorded when shrinking is applied, but this claim does not fit the actual way shrinking is implemented in Jerusat, Chaff, and Eureka.

Applying shrinking contributes to search locality and makes the solver more dynamic, since the set of assigned variables becomes more relevant to the recently explored search space as irrelevant variables become unassigned. Also, since the variables on the assignment stack are precisely those that appeared in recent conflict clauses, conflict clauses are more likely to share common interrelated

Algorithm 1. Adjust Threshold for Shrinking (Threshold for shrinking x, Threshold for number of learned clauses y)

```
Require: $x$ is initialized with the value 95 at the beginning of SAT solving.
  $(mean, stdev)$ := mean and standard deviation of last $y$ learned clause lengths
  $center := mean + 0.5 * stdev$; $ulimit := mean + stdev$
  if $x \geq center$ then
    $x := x - 5$
  end if
  if $x < center$ then
    $x := x + 5$
  end if
  if $x > ulimit$ then
    $x := ulimit$
  end if
  if $x < 5$ then
    $x := 5$
  end if
  return $x$
```

variables. Shrinking often reduced the average length of learned conflict clauses and led to faster solving times, especially for the microprocessor verification benchmarks in Chaff [5].

We propose two new heuristics for shrinking. First, we propose generalizing the shrinking condition of Jerusat. We count the number of decision levels associated with a conflict clause's variables and perform shrinking if this number is greater than a threshold x. The threshold is calculated exactly like the conflict clause size threshold in Chaff in Algorithm 1, using the number of decision levels in the clauses instead of their lengths. We dub our proposal the *decision-level-based shrinking condition*. Interestingly, Jerusat's shrinking condition and its proposed generalization correspond to the recent observation that a "good" clause should contain as few decision levels as possible [8]. The clause deletion scheme of SAT'09 competition winner Glucose is based on this observation. Second, we propose using a new sorting scheme, called *activity ordering*. Our scheme sorts the variables of the conflict clause according to VSIDS's scores, from highest to lowest. Our proposal is intended to make the solver even more dynamic, since it reorders the relevant variables according to their contribution to the derivation of recent conflict clauses.

3 Experimental Results and Discussion

We used Eureka and Minisat for our experiments. Minisat was enhanced by a restart strategy that was found to be optimal for this solver in [2]. We used eight publicly available benchmark families: sat04-ind-goldberg03-hard_eq_check [6] (henceforth, abbreviated to ug), sat04-ind-maris03-gripper [6] (mm), sat04-ind-velev-vliw_unsat_2.0 [9] (uv2), SAT-Race_TS_1 [10] (ms1), SAT-Race_TS_2 [10] (ms2), velev_fvp-sat.3.0 [11] (sv3), velev_fvp-unsat.3.0 [11] (uv3), velev_vliw_unsat_4.0 [9] (uv4).

For each solver, we compared the following four versions, applying: (1) no shrinking; (2) the base version of shrinking, corresponding to Eureka's version of shrinking (recall from Section 2 that Eureka's shrinking algorithm is largely similar to Chaff's: its shrinking condition is based on clause length and the sorting scheme picks variables in descending order of decision levels); (3) the base version, modified by applying activity ordering; (4) the base version, modified by using the decision-level-based shrinking condition.

Table 1 provides some statistics regarding the benchmark families as well as Eureka's results. The first column of the table contains the family name, the second column specifies whether the instances are satisfiable, unsatisfiable, or mixed, and the third column contains the number of instances in the family. Each subsequent pair of columns shows the number of instances solved by Eureka within a three hour timeout and the overall run-time for the particular version in seconds (10800 seconds, that is, three hours, is added for an unresolved benchmark). Table 2 provides Minisat's results in the same format. (A table with all the details of the experimental results appears in [12].)

Compare the empirically best shrinking algorithm versus the version without shrinking for each solver. For Eureka, shrinking (the base version) is helpful for solving seven out of eight families, and critical for solving ug, uv2, uv3 and uv4.

Table 1. Shrinking within Eureka

			No Shr.		**Base Shr.**		**Act. Order**		**Dec. Cond.**	
Family	**SAT?**	**Inst.**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**
ug	UNS	13	10	67005	13	12041	13	14389	12	28457
mm	MIX	10	5	66602	7	39870	7	39426	8	44404
uv2	UNS	8	1	78870	8	12129	8	10283	8	10914
ms1	MIX	50	47	51117	49	27352	48	38208	50	16279
ms2	MIX	50	42	109899	44	92813	43	96564	42	99882
sv3	SAT	20	20	767	20	1119	20	788	20	1375
uv3	UNS	6	1	62038	6	10863	6	11761	6	11251
uv4	UNS	4	0	43200	4	10874	4	9018	4	10677
Sum		161	126	479498	151	207061	149	220437	150	223239

Table 2. Shrinking within Minisat

			No Shr.		**Base Shr.**		**Act. Order**		**Dec. Cond.**	
Family	**SAT?**	**Inst.**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**
ug	UNS	13	7	82310	10	43007	10	43686	11	44140
mm	MIX	10	0	108000	4	71234	0	108000	4	76680
uv2	UNS	8	1	85508	8	12235	8	10817	8	11743
ms1	MIX	50	48	36771	47	37771	49	26894	49	20557
ms2	MIX	50	44	82982	41	122233	42	107147	41	107780
sv3	SAT	20	16	53968	20	9330	20	10084	20	6954
uv3	UNS	6	0	64800	3	38056	0	64800	3	39652
uv4	UNS	4	1	33370	4	15230	4	9912	4	14798
Sum		161	117	547709	137	349096	133	381340	140	322304

For Minisat, shrinking (with the decision-level-based shrinking condition) is critical for solving seven out of eight families (ms2 is an exception). Overall, shrinking enables Eureka and Minisat to solve, respectively, 25 and 23 more benchmarks within the timeout. Hence employing shrinking is highly advantageous.

Compare now our two variations of shrinking versus the base version. The effect of applying the decision-level-based shrinking condition in Minisat is clearly positive as it leads to better overall performance in terms of both the number of solved instances and the run-time. Although applying the decision-level-based ordering condition within Eureka does not lead to better results overall, the solver does perform better for four families (the gap is especially significant for ms1) than with the base version. While the impact of activity ordering is negative for Minisat overall, it performs better than best version (the version with the decision-level-based shrinking condition) for three families. Activity ordering is not helpful overall for Eureka, but is does help solve four families more quickly than the best version (the version with base shrinking). Hence it is recommended that shrinking be tuned for each specific solver and benchmark family.

An important question is whether the effect of shrinking can be achieved by applying other algorithms, proposed after shrinking. Consider the following three techniques: (1) Frequent restarts [13, 2]; (2) A clause-based heuristic, such as CBH [3]; and (3) RSAT's polarity selection heuristic [14], which assigns every decision variable the last value it was assigned. Observe that the combined effect of these three techniques seems to be similar to that of shrinking. First, restarting the search when a certain condition holds corresponds to backtracking when the shrinking condition is met. Second, applying a clause-based heuristic and RSAT's polarity selection heuristic results in selecting the last conflict clause and assigning its literals the value false, similar to what happens in shrinking. It was claimed in [13] that the impact of conflict clause minimization [15, 16] could be considered somewhat similar to the impact of shrinking, since minimization reduces the size of conflict clauses, as does shrinking, according to [5].

However, we have seen that shrinking is extremely useful within Eureka, which employs all the above-mentioned techniques, and Minisat with local restarts, which uses some of them. Thus empirically the effect of shrinking is not achieved by combining other techniques. Let us take a closer look at the differences between our basic version of shrinking and the combination of frequent restarts, CBH, and RSAT's polarity selection heuristic. First, the shrinking condition differs from the restart condition of any known restart strategy. Second, shrinking restarts the search only partially, in contrast to most modern restart strategies. Third, unlike clause-based heuristics, shrinking continues selecting variables from the last conflict clause, even if it is satisfied. Fourth, shrinking re-orders the variables in the last conflict clause. It is, therefore, the simultaneous effect of these features, achieved by carefully choosing the shrinking condition, the sorting scheme, and the shrinking backtrack level, that makes shrinking highly efficient.

4 Conclusion

Assignment stack shrinking is a technique that boosts the performance of modern complete SAT solvers by making them more dynamic and local, and by enhancing the interrelation of the assigned variables. We have described in detail different variations of the shrinking algorithm, including two new heuristics, one of which improves Minisat's overall performance. We have shown that shrinking is extremely efficient within Minisat and Eureka, and that its effects cannot be achieved by other modern algorithms. Shrinking is proving to be a useful concept (that is, a collective name for a family of algorithms) that can be enhanced independently of the other components of SAT solvers, such as restart strategies or decision heuristics.

Acknowledgment

The authors would like to thank Amit Palti for supporting this work and Paul Inbar for editing the paper.

References

1. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC, pp. 530–535. ACM, New York (2001)
2. Ryvchin, V., Strichman, O.: Local restarts. In: Büning, H.K., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 271–276. Springer, Heidelberg (2008)
3. Dershowitz, N., Hanna, Z., Nadel, A.: A clause-based heuristic for SAT solvers. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 46–60. Springer, Heidelberg (2005)
4. Nadel, A.: Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew Univeristy of Jerusalem, Jerusalem, Israel (November 2002)
5. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: An efficient SAT solver. In: [17], pp. 360–375
6. Berre, D.L., Simon, L.: Fifty-five solvers in Vancouver: The SAT 2004 competition. In: [17], pp. 321–344
7. Nadel, A.: Understanding and improving a modern SAT solver. PhD thesis, Tel Aviv University, Tel Aviv, Israel (August 2009)
8. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI, pp. 399–404 (2009)
9. Velev, M., Bryant, R.: Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In: Proceedings of the 38th Design Automation Conference (DAC 2001), pp. 226–231 (2001)
10. Sinz, C.: SAT-Race 2006 (2006), `http://fmv.jku.at/sat-race-2006/`
11. Velev, M.N.: Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In: Proc. Design, Automation and Test in Europe Conference and Exhibition, pp. 28–35 (2002)
12. Nadel, A., Ryvchin, V.: Experimental results for the SAT'10 paper Assignment stack shrinking, `http://www.cs.tau.ac.il/research/alexander.nadel/sat10_ass_res.xlsx`

13. Biere, A.: PicoSAT essentials. JSAT 4(2-4), 75–97 (2008)
14. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
15. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res. (JAIR) 22, 319–351 (2004)
16. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 237–243. Springer, Heidelberg (2009)
17. Hoos, H.H., Mitchell, D.G. (eds.): SAT 2004. LNCS, vol. 3542. Springer, Heidelberg (2005)