Problem-Sensitive Restart Heuristics for the DPLL Procedure[⋆]

Carsten Sinz and Markus Iser

Research Group “Verification meets Algorithm Engineering”
Institute for Theoretical Computer Science
University of Karlsruhe, Germany
{sinz, iser}@ira.uka.de

Abstract. Search restarts have shown great potential in speeding up SAT solvers based on the DPLL procedure. However, most restart policies presented so far do not take the problem structure into account. In this paper we present several new problem-sensitive restart heuristics. They all observe different search parameters like conflict level or backtrack level over time and, based on their development, decide whether to perform a restart or not. We also present a Java tool to visualize these search parameters on a given SAT instance over time in order to analyze existing heuristics and develop new one.

1 Introduction

Randomization and restart policies have been incorporated into SAT solvers since the mid 90s [1, 2]. In a search procedure, a *restart* cancels the search for a solution after a certain number of steps, and then starts over again performing another solving attempt. To prevent the solver from generating the same search tree repeatedly, in early work on restarts randomization has been added to the decision heuristics of the DPLL procedure, thus modifying the search tree slightly on each run. With the advent of conflict driven clause learning (CDCL) SAT solvers [3, 4], new ways to modify the search tree (to prevent repeated searches without progress) emerged, as these solvers dynamically compute *variable activities*, which can be taken into consideration after a restart: variables with highest activity (i.e. those occurring most frequently in recently learned clauses) are branched on first. Using learned clauses also allows to carry over results from previously cancelled attempts, as learned clauses need not be dropped after a restart. So the time spent in a failed search is not wasted. The theoretical groundwork for restarts was laid down in the seminal paper by Gomes *et al.* [6], where they explained the success of restarts by *heavy-tailed distributions*, which occur in randomized searches.

Current DPLL SAT solvers implement different restart heuristics or *restart schemes*. MiniSat 2.0, the latest publicly available version of the well-known MiniSat [7] solver, implements the RGR strategy (*randomization and geometric restarts*) proposed by Walsh [8], i.e. the n-th restart is performed $k \cdot \alpha^{n-1}$ steps after the previous restart (where

[⋆] This work was supported in part by the “Concept for the Future” of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

$k = 100$ and $\alpha = 1.5$ by default, and steps refer to the number of conflicts). Other, more recent restart schemes are based on Luby sequences [9] or inner/outer restarts.

However, until recently, restart schemes for DPLL solvers have been static, in the sense that they are the same for each SAT instance, do not change during search, and thus are not *problem-sensitive*.[1] To the best of our knowledge, the first attempt to bring dynamic restart policies to DPLL solvers was made by Biere [12]. His adaptive restart strategy ANRFA (*average number of recently flipped assignments*) tries to estimate the *agility*of the on-going search process by taking the number of variable *flips* (determined with respect to the memorized phases) into account. A high number of flips indicates a high agility, which is considered good, whereas a low number of flips might indicate that the search process got stuck and a restart might be advantageous.

In this paper we present a set of new problem-sensitive restart policies. They are all based on observing a problem parameter during search and, depending on the development of this parameter, decide whether or not to perform a restart. We have also developed a Java tool for monitoring and visualizing such search parameters over time for a given SAT instance.

2 Problem-Sensitive Restart Heuristics

Accurately estimating the progress of a search process is a non-trivial task. However, some search parameters of a CDCL search—like length of learned clauses, search depth, or backtrack level—give hints whether the search is progressing fast or slowly. Based on these parameters, we have developed a set of new restart policies, which we will now describe in more detail. All our strategies are based on observing search parameters over time. Typically, we average over the last few parameter values that have occurred at a certain state during the search (in order to obtain more stable results). To put the most recent values of a search parameter into relation with "typical" values for the problem instance, we employ a short-term as well as a long-term memory for each parameter (storing the last L resp. S values for each). The short-term memory allows computing the current (smoothed) average value of the parameter, whereas the long-term memory is used to determine an average value over a longer period of time to compute, in a sense, a "typical" value for the parameter on this instance.

Such averaging over last (temporal) values is also know as *moving average* (or running average) in statistics. To compute the moving average of a parameter x, we use an array to store its $L + 1$ most recent values $x_{n-L}, \ldots, x_n$. The same array can be used for both the long-term and short-term moving averages. It is used as a cyclic buffer with a pointer to the most recent entry. New entries move the pointer forward and overwrite the oldest value. Computing the moving average (A_n) over the last L values can be done efficiently (without summing over the whole array) in an iterative manner:

$$A_n = A_{n-1} + \frac{x_n - x_{n-L}}{L} \quad .$$

[1] For *local search* solvers, problem-sensitive restart policies have been known for some time [10, 11].

During the first L steps we just fill the array with values and, starting with step $L+1$, we use it to compute average values and to decide whether to restart or not. We record parameter values at each leaf of the search tree (i.e. on each conflict), such that the notion of step coincides with the number of conflicts. Parameters we found suitable for monitoring over time include

Conflict level: the height of the search tree when a conflict occurred.
Backtrack level: the height of the search tree to which the solver jumped back.
Length of learned clauses: the length of the currently learned clause.
Trail size: The total number of assigned variables when a conflict occurred (including variables assigned by unit propagation).

To determine whether we should perform a restart, we selected one of these parameters and tracked its evolution over time. It is possible to combine several parameters in a restart heuristic, but we have not made any experiments in this direction so far. We describe our restart heuristics exemplarily for the parameter **conflict level**.

Ratio of long term vs. short term average (R): This heuristic assumes that (relatively) low conflict levels are preferable to high conflict levels. The intuition is that uniformly low conflict levels span up a smaller search space. Moreover, the conflict clauses produced on lower conflict levels are potentially shorter and thus prune a larger fraction of the search space. Low conflict levels are determined in relation to the long-term average value: as soon as the short-term average conflict level (c_S) is much higher ($c_S/c_L \geq f_T$) than the long-term average conflict level (c_L), we perform a restart. Here f_T is a fixed threshold factor.
Avoidance of plateaus (P): When during search the same (high) conflict level occurs over and over again, this could indicate that the search got stuck. To avoid such a situation, we count the number S of minimal values of the conflict level over the last L steps. If the minimum occurs more often than a fixed number of times (given as a fraction of L) and the minimum is larger than a threshold value ($c_{T,\min}$), a restart is performed.
Preference for high variance (V): Similar to the last heuristic, and related to Biere's notion of *agility*, the intuition behind this restart scheme is to avoid situations where the conflict level is mainly constant (low variance). We compute the variance (resp. the standard deviation) over the last L conflict levels by $\sigma^2 = \frac{1}{L-1} \cdot \sum_{i=n-L+1}^{n} (x_i - \mu)^2$, where $\mu = \frac{1}{L} \cdot \sum_{i=n-L+1}^{n} x_i$ is the mean value of the last L conflict levels. If σ^2 is smaller than a threshold value $c_{T,\sigma}$, a restart is performed.

3 Experimental Evaluation

We have implemented the restart heuristics mentioned in the previous section on top of MiniSat 2.0.[2] The implementation required only minor modifications, including addition of the array storing recent versions of a search parameter.[3]

[2] http://minisat.se/downloads/minisat2-070721.zip
[3] In CDCL solvers, restarts are often closely tied to the deletion of learned clauses ("garbage collection"). In our extensions of MiniSat we left the clause deletion intervals unchanged. Also

We have made experiments with all search parameters mentioned in the previous section, but will report only on results based on the **backtrack level** as, without further tuning of the heuristics' threshold values, further investigation of this parameter seemed most promising. For heuristic R, we have used a threshold factor of $f_T = 3.5$, and values $L = 30$ and $S = 7$ for long and short term ranges. For restart scheme P, we have determined best parameters for L and S by a sequence of experiments (using the *manolios* benchmark set), which resulted in $L = 24$ and $S = 4$ as optimal values. The restart threshold $c_{T,\min}$ was set to $100 \cdot \frac{V}{C}$, where V and C denote the number of variables resp. clauses of the instance. For heuristic V, we have set L to 30 and $c_{T,\sigma}$ to $1.3^2 = 1.69$. We compared our heuristics with both the original MiniSat 2.0 version[4] (denoted by M in the tables) and a restart heuristics that makes (frequent) restarts after a constant number of 200 steps (denoted by C in the tables).

Table 1. Comparison of problem-sensitive restart strategies.

benchmark	# instances			# solved sat					# solved unsat					# solved total				
family	sat	unsat	total	P	R	V	C	M	P	R	V	C	M	P	R	V	C	M
manolios	0	210	210	–	–	–	–	–	**166**	145	161	158	151	**166**	145	161	158	151
velev-pipe	0	27	27	–	–	–	–	–	**16**	6	15	**16**	8	**16**	6	15	**16**	8
sat-race-2006	43	57	100	26	29	28	22	**36**	**50**	33	48	45	39	**76**	62	**76**	67	75
satcomp-07 indust.	68	107	175	36	37	**44**	40	37	**63**	51	55	57	53	**99**	88	**99**	97	90
satcomp-07 crafted	$\geq$34	$\geq$95	201	16	17	9	9	**22**	36	**48**	15	25	46	52	65	24	34	**68**
sat-race-2008	48	52	100	24	29	25	22	**33**	37	27	31	**41**	30	61	56	56	**63**	**63**

Table 1 shows the results of different restart heuristics on a number of benchmark families. All experiments were performed under SuSE Linux on machines equipped with an Intel Xeon E5430 processor running at 2.66 GHz and 16 GB of RAM. We set a time limit of 900 seconds per instance and solver for all of our experiments. The test set *manolios* is a parameterized benchmark suite consisting of hard pipelined-machine-verification problems. The *velev-pipe* family includes Velev's *pipe-unsat-1.0* and *pipe-unsat-1.1* problem sets.[5] The other benchmarks stem from previous SAT Competitions (2007, *industrial* and *crafted* category) and SAT-Races (2006 and 2008).[6] Table 1 shows the number of instances contained in each benchmark package, split into satisfiable and unsatisfiable instances. The following columns report on the number of instances that could be solved using the respective heuristics (P, R, V, C, or M), first only counting satisfiable, then unsatisfiable, and finally all instances. Best results for a benchmark set are indicated in boldface.

Whereas on satisfiable instances our restart heuristics did not perform better then the plain Minisat 2.0 RGR heuristic (in general even worse, with the exception of heuristic

note that only either the restart intervals or the clause deletion intervals have to be gradually increased to ascertain completeness of the search procedure.

[4] As mentioned in the introduction, MiniSat 2.0 implements a RGR strategy, where times between restarts grows exponentially over time.

[5] Available from http://www.miroslav-velev.com/sat_benchmarks.html.

[6] The SAT-Race 2006 and 2008 instances have been processed with the SatELite preprocessor.

V on the SAT Competition 2007 instances, industrial category), on unsatisfiable instances our heuristics were able to outperform the static Minisat scheme. Notable is a 100% increase in the number of solved instances for our heuristic P on the *velev-pipe* benchmark suite, as well as an increase of almost 10% on the *manolios* benchmarks. Considering both satisfiable and unsatisfiable instances, heuristic P still outperforms MiniSat in the number of solved instances in general.

On selected instances from the *manolios* and *velev-pipe* benchmark families, our restart scheme P considerably outperforms MiniSat, with speed-ups up to a factor of approximately 167. On average, heuristic P shows a speed-up of more than 38% over MiniSat 2.0 on these benchmark families. Noteworthy is also the achieved reduction in search space, measured in number of conflicts, which lies between 42.6% and 99.2%. In only one case (out of all 42 instances of the *manolios* benchmark that could be solved by both heuristics in between 2 and 15 minutes) the number of conflicts was higher for our strategy P. A reduction in number of conflicts is also typically connected with shorter proofs of unsatisfiability. This is especially important for algorithms that further process proofs generated by SAT solvers, like interpolation-based methods [13]. It is also striking that the number of restarts made by heuristic P is much higher than the number of restarts made by MiniSat.

We also made experiments comparing heuristic P with the latest, non-public version of MiniSat (2.1), as used in SAT-Race 2008. MiniSat 2.1 adds a whole set of improvements to version 2.0, including phase memorization, special handling for binary and blocked clauses, an improved memory manager, and a Luby restart strategy. MiniSat 2.1 has shown to perform considerably better than MiniSat 2.0 in SAT-Race 2008 (81 vs. 59 solved instances). This even holds when MiniSat 2.0 is extended with our new problem-sensitive restart strategy P. However, on the *manolios* benchmark family, our heuristic performs better, even without the other improvements implemented in MiniSat 2.1. Considering only unsatisfiable instances, strategy P is also better on the SAT-Race 2006 instances.

We performed further experiments with a modified version of MiniSat 2.0 implementing the same Luby strategy as in MiniSat 2.1. However, on all benchmark sets, we obtained better results with our strategy P. In another set of experiments we added phase memorization to MiniSat 2.0 with Luby strategy and heuristics P. Again, the results with strategy P were better (in number of solved instances), with the only exception of the *velev* benchmark set, where the Luby strategy was able to solve one instance more.

RViewer: A Tool for Monitoring Search Parameters. To experiment with different restart heuristics and to visualize search parameters over time we have implemented a Java tool called *RViewer*[7]. RViewer reads a dump file generated by a CDCL SAT solver, which contains the sequence of search parameters over time, one for each conflict. RViewer visualizes the development of the contained parameters during search. It allows to select one or multiple search parameters for display, computation of the moving average, zooming in and out, as well as moving through the dump file. RViewer was of great help in setting up restart policy P and to detect the phenomenon of plateaus for the backtrack level.

[7] RViewer is available for download at http://baldur.iti.uka.de/software/RViewer.

4 Related Work & Conclusion

Huang [14] experimentally compares different restart policies, including a whole set of different RGR strategies, and a strategy based on Luby sequences. All policies examined in his survey are static ones, but he suggests that "substantial performance gains may be possible by using appropriate dynamic restart policies." Kautz *et al.* [15] give theoretical and empirical results on context-sensitive restart policies for randomized search procedures. The notion of context-sensitivity they use differs from our notion of problem-sensitivity in that their strategies are selected per instance, whereas our strategies are "more dynamic" in that they can also vary during a solver's run. Related to our work is that of Haim and Walsh [16], where they estimate the runtime of a SAT solver based on problem parameters observed during the initial phase of the search.

We have presented different dynamic, problem-sensitive restart heuristics that we implemented on top of MiniSat 2.0. We obtained good results with our policy P, which is based on avoidance of plateaus. The phenomenon of plateaus, which we observed in almost all SAT instances using our tool RViewer, also seems to be new. Directions for future research include further refinement of dynamic restart policies, especially by employing a combination of several problem parameters. It would also be interesting to find a theoretical underpinning why unsatisfiable instances seem to profit most from dynamic restart policies. A larger set of different restart policies might also be of help in implementing parallel SAT solvers based on competition parallelism.

References

1. J.M. Crawford and A.B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI*, 1994.
2. R.J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI/IAAI*, 1997.
3. J.P. Marques Silva and K.A. Sakallah. GRASP – a new search algorithm for satisfiability. In *ICCAD*, 1996.
4. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, 2001.
5. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, 2007.
6. C.P. Gomes, B. Selman, and H.A. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, 1998.
7. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, 2003.
8. T. Walsh. Search in a small world. In *IJCAI*, 1996.
9. M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47(4), 1993.
10. H.H. Hoos. An adaptive noise mechanism for WalkSAT. In *AAAI/IAAI*, 2002.
11. W. Wei, C. M. Li, and H. Zhang. A switching criterion for intensification and diversification in local search for SAT. *J. Satisfiability, Boolean Modeling and Comput.*, 4:219–237, 2008.
12. A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *SAT*, 2008.
13. K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 2003*, 2003.
14. J. Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, 2007.
15. H.A. Kautz, E. Horvitz, Y. Ruan, C.P. Gomes, and B. Selman. Dynamic restart policies. In *AAAI/IAAI*, 2002.
16. S. Haim and T. Walsh. Online estimation of SAT solving runtime. In *SAT*, 2008.