A Propagation Rate based Splitting Heuristic for Divide-and-Conquer Solvers

Saeed Nejati, Zack Newsham, Joseph Scott, Jia Hui Liang,
Catherine Gebotys, Pascal Poupart, and Vijay Ganesh

University of Waterloo, Waterloo, ON, Canada

Abstract. In this paper, we present a divide-and-conquer SAT solver, MapleAmpharos, that uses a novel *propagation-rate* (PR) based splitting heuristic. The key idea is that we rank variables based on the ratio of how many propagations they cause during the run of the worker conflict-driven clause-learning solvers to the number of times they are branched on, with the variable that causes the most propagations ranked first. The intuition here is that, in the context of divide-and-conquer solvers, it is most profitable to split on variables that maximize the propagation rate. Our implementation MapleAmpharos uses the AMPHAROS solver as its base. We performed extensive evaluation of MapleAmpharos against other competitive parallel solvers such as Treengeling, Plingeling, Parallel CryptoMiniSat5, and Glucose-Syrup. We show that on the SAT 2016 competition Application benchmark and a set of cryptographic instances, our solver MapleAmpharos is competitive with respect to these top parallel solvers. What is surprising that we obtain this result primarily by modifying the splitting heuristic.

1 Introduction

Over the last two decades, sequential Boolean SAT solvers have had a transformative impact on many areas of software engineering, security, and AI [10, 8, 25]. Parallel SAT algorithms constitute a natural next step in SAT solver research, and as a consequence there has been considerable amount of research in parallel solvers in recent years [21]. Unfortunately, developing practically efficient parallel SAT solvers has proven to be a much harder challenge than anticipated [13, 27, 12]. Nonetheless, there are a few solver architectures that have proven to be effective on industrial instances. The two most widely used architectures are portfolio-based and variants of divide-and-conquer approach.

Portfolio solvers [5, 6, 1, 15] are based on the idea that a collection of solvers each using a different set of heuristics are likely to succeed in solving instances obtained from diverse applications. These portfolio solvers are further enhanced with clause sharing techniques. By contrast, divide-and-conquer techniques are based on the idea of dividing the search space of the input formula F and solving the resulting partitions using distinct processors. Partitions are defined as the original formula F restricted with a set of assumptions P (i.e., $F \wedge P$). These sets of assumptions cover the whole search space and each pair of them has at least one clashing literal [11, 16–18, 26, 29].

Different strategies for choosing assumptions to split the search space (known as splitting heuristics in parallel SAT solver parlance) and the relative speedup obtained by splitting some or all of the variables have been studied in detail in the literature [21]. It is well-known that the choice of splitting heuristic can have a huge impact on the performance of divide-and-conquer solvers. One successful approach is to use *look-ahead* to split the input instance into sub-problems, and solve these sub-problems using CDCL solvers. Such methods are generally referred to as cube-and-conquer solvers [16], where the term cube refers to a conjunction of literals. In their work, Heule et al. [16] split the input instances (cube phase) and solve the partitions (conquer phase) sequentially, one after another. A potential problem with this approach is that it might result in sub-problems of *unbalanced hardness* and might lead to a high solving time for a few sub-problems. By contrast, in concurrent-cube-and-conquer [29] these two phases are run concurrently.

In this paper, we propose a new propagation rate-based splitting heuristic to improve the performance of divide-and-conquer parallel SAT solvers. We implemented our technique as part of the AMPHAROS solver [2], and showed significant improvements vis-a-vis AMPHAROS on instances from the SAT 2016 Application benchmark. Our key hypothesis was that variables that are likely to maximize propagation are good candidates for splitting in the context of divide-and-conquer solvers because the resultant sub-problems are often simpler. An additional advantage of ranking splitting variables based on their *propensity to cause propagations* is that it can be cheaply computed using conflict-driven clause-learning (CDCL) solvers that are used as workers in most modern divide-and-conquer parallel SAT solvers.

Contributions. We present a new splitting heuristic based on propagation rate, where a formula is broken into two smaller sub-formulas by setting the highest propagating variable to True and False. We evaluate the improved solver against the top parallel solvers from the SAT 2016 competition on the Application benchmark and a benchmark of cryptographic instances obtained from encoding of preimage attacks on the SHA-1 cryptographic hash function. Our solver, called MapleAmpharos, outperforms the baseline AMPHAROS and is competitive against Glucose, parallel CryptoMiniSat5, Treengeling and Plingeling on the SAT 2016 Application benchmark. Additionally, MapleAmpharos has better solving time compared to all of the solvers on our crypto benchmark.

2 Background on Divide-and-Conquer Solvers

We assume that the reader is familiar with the sequential conflict-driven clause-learning (CDCL) solvers. For further details we refer the reader to the Handbook of Satisfiability [9].

All divide-and-conquer SAT solvers take as input a Boolean formula which is partitioned into many smaller sub-formulas that are subsequently solved by sequential CDCL solvers. Usually the architecture is of a master-slave type. The master maintains the partitioning using a tree, where each node is a variable and can have at most two children. Left branch represents setting parent variable to

False and right branch is for setting it to True. It means that if the original formula is F, and variable x is used to split the formula, the two sub-formulas would be $F_1 = F \wedge \neg x$ and $F_2 = F \wedge x$. We refer to the variable x as the *splitting variable*. Each leaf in this tree is called a cube (conjunction of literals). The path from root to each of the leaves represents different assumption sets that will be conjuncted to the original formula to generate partitions. The slave processes are usually CDCL solvers and they solve the sub-formulas simplified against the respective cubes, and report back to the master the results of completely or partially solving these sub-formulas.

Depending on the input problem, there could be redundant work among the workers when the formula is split. When the workers are directed to work on the same problem (with different policies), it is called *intensification*, and when the search space of workers are less overlapping, we call it *diversification*. Maintaining a balance between these two usually leads to a better performance [14].

3 Propagation Rate-based Splitting Heuristic

In this Section we describe our propagation rate based splitting heuristic, starting with a brief description of AMPHAROS that we use as our base solver [2]. We made our improvements in three steps: 1) We used Maplesat [19] as the worker or backend solver. This change gave us a small improvement over the base AMPHAROS. 2) MapleAmpharos-PR: We used a propagation-rate based splitting heuristic on top of using Maplesat as worker solver. 3) MapleAmpharos: We applied different restart policies at worker solvers of MapleAmpharos-PR.

3.1 The AMPHAROS Solver

AMPHAROS is a divide-and-conquer solver wherein each worker is a CDCL SAT solver. The input to each worker is the original formula together with assumptions corresponding to the path (from the root of the splitting tree to the leaf) assigned to the worker. The workers can switch from one leaf to another for the sake of load balancing and intensification/diversification. Each worker searches for a solution to the input formula until it reaches a predefined limit or upper bound on the number of conflicts. We call this the *conflict limit*. Once the conflict limit is reached, the worker declares that the cube[1] is hard and reports the "best variable" for splitting the formula to the master. A variable is deemed "best" by a worker if it has the highest VSIDS activity over all the variables when the conflict limit is reached. The Master then uses a load balancing policy to decide whether to split the problem into two by creating False and True branches over the reported variable.

[1] While the term cube refers to a conjunction of literals, we sometimes use this term to also refer to a sub-problem created by simplifying a formula with a cube.

3.2 Propagation Rate Splitting Heuristic

As mentioned earlier, the key innovation in MapleAmpharos is a propagation rate-based splitting heuristic. Picking variables to split on such that the resultant sub-problems are collectively easier to solve plays a crucial role in the performance of divide-and-conquer solvers. Picking the optimum sequence of splitting variables such that the overall running time is minimized is in general an intractable optimization problem.

For our splitting heuristic, we use a dynamic metric inspired by the measures that look-ahead solvers compute as part of their "look-ahead policy". In a look-ahead solver, candidate variables for splitting are assigned values (True and False) one at a time, and the formula is simplified against this assigned variable. A measure proportional to the number of simplified clauses in the resultant formula is used to rank all the candidate variables in decreasing order, and the highest ranked variable is used as a split. However, look-ahead heuristics are computationally expensive, especially when the number of variables is large. Propagation rate-based splitting on the other hand is very cheap to compute.

In our solver MapleAmpharos when a worker reaches its conflict limit, it picks the variable that has caused the highest number of propagations per decision (the propagation rate) and reports it back to the Master node. More precisely, whenever a variable v is branched on, we sum up the number other variables propagated by that decision. The propagation rate of a variable v is computed as the ratio of the total number of propagations caused whenever v is chosen as a decision variable divided by the total number of times v is branched on during the run of the solver. Variables that have never been branched on during the search get a value of zero as their propagation rate.

When a worker solver stops working on a sub-problem due to it reaching the conflict limit, or proving the cube to be UNSAT, it could move to work on a completely different sub-problem which has a different set of assumptions. Even through this node switching we do not reset the propagation rate counters.

The computational overhead of our propagation rate heuristic is minimal, since all the worker solvers do is maintain counters for the number of propagations caused by each decision variable during their runs. An important feature of our heuristic is that the number of propagation per decision variable is deeply influenced by the branching heuristic used by the worker solver. Also, the search path taken by the worker solver determines the number of propagations per variable. For example, a variable v when set to the value True might cause lots of propagation, and when set to the value False may cause none at all. Despite these peculiarities, our results show that the picking splitting variables based on the propagation rate-based heuristic is competitive for Application and cryptographic instances.

3.3 Worker Diversification

Inspired by the idea of using different heuristics in a competitive solver setting [15], we experimented with the idea of using different restart policies in worker CDCL solvers. We configured one third of the workers to use Luby

Table 1: Solving time details of MapleAmpharos and competing parallel solvers on SAT 2016 Application benchmark

	MapleAmpharos	AMPHAROS	Ampharos-Maplesat	MapleAmpharos-PR
Avg. Time (s)	979.396	392.933	310.94	1035.73
# of solved	182	104	107	171
SAT	77	42	44	72
UNSAT	105	62	63	99
	CryptoMiniSat	Syrup	Plingeling	Treengeling
Avg. Time (s)	942.894	898.767	965.167	969.467
# of solved	180	180	192	184
SAT	72	74	76	77
UNSAT	108	106	116	107

restarts [20], another third to use geometric restarts, and the last third to use MABR restarts. MABR is a Multi-Armed Bandit Restart policy [22], which adaptively switches between 4 different restart policies of linear, uniform, geometric and Luby. We note that while we get some benefit from worker diversification, the bulk of the improvement in the performance of MapleAmpharos over AMPHAROS and other solvers is due to the propagation rate splitting heuristic.

4 Experimental Results

In our experiments we compared MapleAmpharos against 5 other top-performing parallel SAT solvers over the SAT 2016 Application benchmark and a set of cryptographic instances obtained from encoding of SHA-1 preimage attacks as Boolean formulas.

4.1 Experimental Setup

We used the Application benchmark of the SAT competition 2016 which has 300 industrial instances obtained from diverse set of applications. Timeout for each instance was set at 2 hours wall clock time. All jobs were run on 8 core Intel Xeon CPUs with 2.53 GHz and 8GB RAM. We compared our solver MapleAmpharos against the top parallel solvers from the SAT 2016 competition, namely, Treengeling and Plingeling [7], CryptoMiniSat5 [28], Glucose-Syrup [4] and also baseline version of AMPHAROS solver [2]. Our parallel solver MapleAmpharos uses Maplesat [19] as its worker CDCL solver.

4.2 Case Study 1: SAT 2016 Application Benchmark

Figure 1, shows the cactus plot comparing the performance of MapleAmpharos against the other top parallel SAT solvers we considered in our experiments. In this version of the MapleAmpharos solver we used the best version of worker

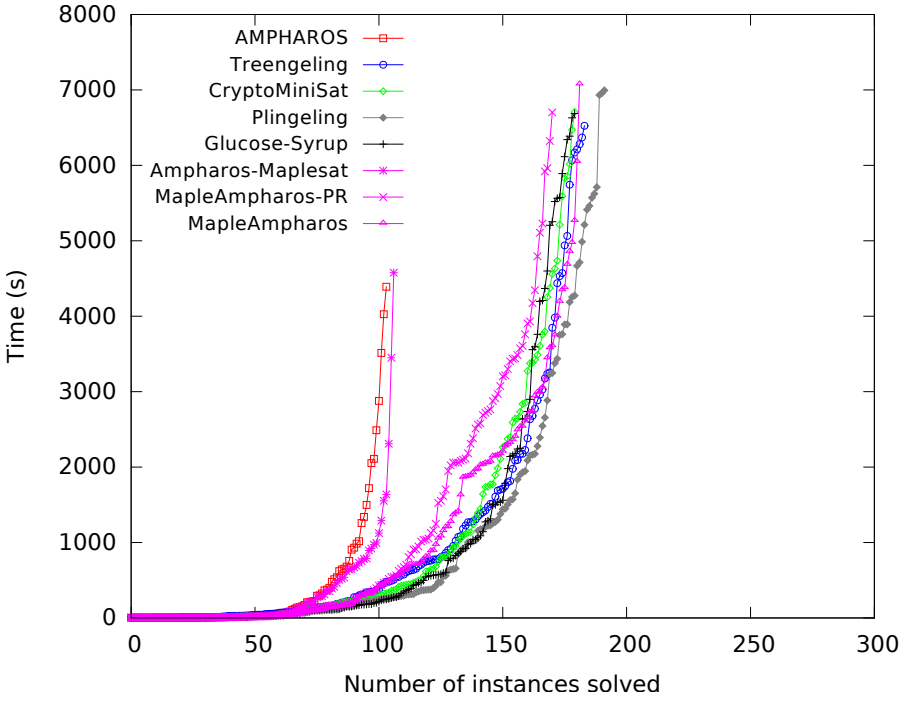

Fig. 1: Performance of MapleAmpharos vs. competing parallel solvers over the SAT 2016 Application benchmark

diversification (which is a combination of Luby, Geometric and MAB-restart referred to section 3.3). As can be seen from the cactus plot in Figure 1 and the Table 1, MapleAmpharos outperforms the baseline AMPHAROS, and is competitive vis-a-vis Parallel CryptoMiniSat, Glucose-Syrup, Plingeling and Treengeling. However, MapleAmpharos performs the best compared to the other solvers when it comes to solving cryptographic instances.

4.3 Case Study 2: Cryptographic Hash Instances

We also evaluated the performance of our solver against these parallel SAT solvers on instances that encode preimage attacks on the SHA-1 cryptographic hash function. These instances are known to be hard for CDCL solvers. The best solvers to-date can only invert at most 23 rounds automatically (out of maximum of 80 rounds in SHA-1) [22, 23]. Our benchmark consists of instances corresponding to a SHA-1 function reduced to 21, 22 and 23 rounds, and for each number of rounds, we generate 20 different random hash targets. The solution to these instances are preimages that when hashed using SHA-1, generate the same hash targets. The instances were generated using the tool used for generating these type of instances in SAT competition [24]. The timeout for each instance was

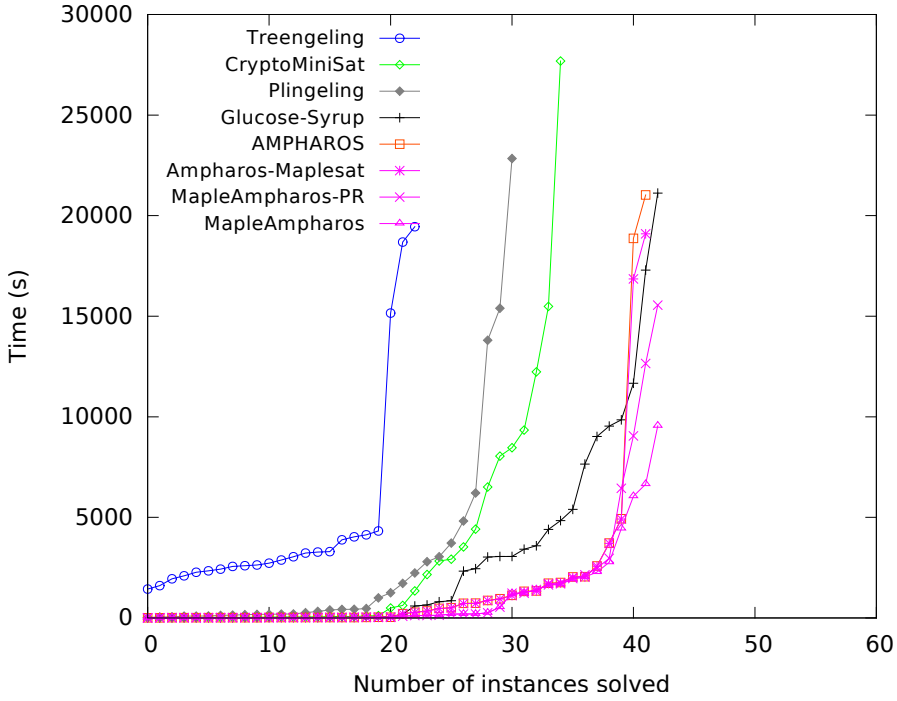

Fig. 2: Performance of MapleAmpharos vs. competing parallel solvers on SHA-1 instances

set to 8 hours. Figure 2 shows the performance comparison and Table 2 shows details of the average solving times on this benchmark. We compute the average for each solver only over the instances for which the resp. solvers finish. As can be seen from these results, MapleAmpharos performs the best compared to all of the other solvers. In particular, for the hardest instances in this benchmark (encoding of preimage attacks on 23 rounds of SHA-1), only Glucose-Syrup, AMPHAROS, and MapleAmpharos are able to invert some of the targets. Further, MapleAmpharos generally solves these SHA-1 instances much faster.

5 Related Work

Treengeling [7], one of the most competitive parallel SAT solvers to-date, uses a concurrent-cube-and-conquer architecture wherein a look-ahead procedure is used to split the formula, and sequential CDCL worker solvers are used to solve the sub-formulas thus generated. Treengeling is multi-threaded, and uses Lingeling as the backend sequential solver.

AMPHAROS [2] is a divide-and-conquer solver that uses VSIDS scoring as its splitting heuristic. The unit literals and low-LBD learnt clauses in each of

Table 2: Average solving time comparison on SHA-1 benchmark

	MapleAmpharos	AMPHAROS	Ampharos-Maplesat	MapleAmpharos-PR
Avg. Time (s)	1048.53	1619.1	1518.76	1457.14
# of solved	43	42	42	43
	CryptoMiniSat	Syrup	Plingeling	Treengeling
Avg. Time (s)	3056.31	2912.84	2668.48	4783.35
# of solved	35	43	31	23

the workers are shared with other workers through the master node. It also adaptively balances between intensification and diversification by changing the number of shared clauses between the workers as well as number of cubes. AMPHAROS uses MPI for communication between master and workers. It uses MiniSat and Glucose as worker solvers, and other sequential solvers can be easily retrofitted as worker solvers.

The parallel version of CryptoMiniSat5 [28] that participated in the SAT 2016 competition implements various in-processing techniques that can run in parallel. Unlike AMPHAROS that shares clauses based on LBD, it shares only unary and binary clauses.

Unlike AMPHAROS, Glucose-Syrup [3] uses a shared memory architecture for communication. Additionally, it uses a lazy exchange policy for learnt clauses. They share clauses between cores when they are deemed to be useful locally rather than right after they are learnt. Furthermore, it implements a strategy for importing shared clauses into each of the workers. It checks whether the clauses are in the UNSAT proof of a cube and if not, they will be flagged as useless. This approach reduces the burden on propagation by reducing the sharing of not useful clauses.

Plingeling [7] is a portfolio solver which uses Lingeling as the CDCL solver. It launches workers with different configurations and shares clauses among the workers. The workers may also exchange facts derived using simplifications (e.g., the equivalence between variables) applied to their copy of the formula.

6 Conclusion

We present a propagation rate-based splitting heuristic for divide-and-conquer solvers, implemented on top of the AMPHAROS solver, that is competitive with respect to 5 other parallel SAT solvers on industrial and cryptographic instances. Many of these competing solvers were top performers in the SAT 2016 competition. Our key insight is that attempting to maximize propagations is a good strategy for splitting in divide-and-conquer solvers because the resultant sub-problems are significantly simplified and hence easier to solve. Finally, a crucial advantage of our approach is that the computational overhead associated with propagation rate-based splitting heuristic is minimal.

References

1. Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.M., Piette, C.: Revisiting clause exchange in parallel sat solving. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 200–213. Springer (2012)
2. Audemard, G., Lagniez, J.M., Szczepanski, N., Tabary, S.: An adaptive parallel sat solver. In: International Conference on Principles and Practice of Constraint Programming. pp. 30–48. Springer (2016)
3. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel sat solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 197–205. Springer (2014)
4. Audemard, G., Simon, L.: Glucose and syrup in the sat'16. SAT COMPETITION 2016 pp. 40–41 (2016)
5. Balyo, T., Sanders, P., Sinz, C.: Hordesat: a massively parallel portfolio sat solver. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 156–172. Springer (2015)
6. Biere, A.: Yet another local search solver and lingeling and friends entering the sat competition 2014. SAT Competition 2014(2), 65 (2014)
7. Biere, A.: Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. SAT COMPETITION 2016 p. 44 (2016)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in computers 58, 117–148 (2003)
9. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically Generating Inputs of Death. ACM Transactions on Information and System Security (TISSEC) 12(2), 10 (2008)
11. Chu, G., Stuckey, P.J., Harwood, A.: Pminisat: a parallelization of minisat 2.0. SAT race (2008)
12. Fujii, H., Fujimoto, N.: Gpu acceleration of bcp procedure for sat algorithms. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). p. 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) (2012)
13. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: Limits to parallel computation: P-completeness theory. Oxford University Press on Demand (1995)
14. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and intensification in parallel sat solving. In: International conference on principles and practice of constraint programming. pp. 252–265. Springer (2010)
15. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel sat solver. Journal on Satisfiability, Boolean Modeling and Computation 6, 245–262 (2008)
16. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding cdcl sat solvers by lookaheads. In: Haifa Verification Conference. pp. 50–65. Springer (2011)
17. Hyvärinen, A.E., Junttila, T., Niemelä, I.: Partitioning sat instances for distributed solving. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 372–386. Springer (2010)
18. Hyvärinen, A.E., Manthey, N.: Designing scalable parallel sat solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 214–227. Springer (2012)

19. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for sat solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 123–140. Springer International Publishing (2016)
20. Luby, M., Sinclair, A., Zuckerman, D.: Optimal Speedup of Las Vegas Algorithms. In: Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the. pp. 128–133. IEEE (1993)
21. Manthey, N.: Towards next generation sequential and parallel sat solvers. KI-Künstliche Intelligenz 30(3-4), 339–342 (2016)
22. Nejati, S., Liang, J.H., Ganesh, V., Gebotys, C., Czarnecki, K.: Adaptive restart and cegar-based solver for inverting cryptographic hash functions. arXiv preprint arXiv:1608.04720 (2016)
23. Nossum, V.: SAT-based preimage attacks on SHA-1. Master's thesis (2012)
24. Nossum, V.: Instance Generator for Encoding Preimage, Second-Preimage, and Collision Attacks on SHA-1. Proceedings of the SAT competition pp. 119–120 (2013)
25. Rintanen, J.: Planning and SAT. Handbook of Satisfiability 185, 483–504 (2009)
26. Semenov, A., Zaikin, O.: Using monte carlo method for searching partitionings of hard variants of boolean satisfiability problem. In: International Conference on Parallel Computing Technologies. pp. 222–230. Springer (2015)
27. Sohanghpurwala, A.A., Hassan, M.W., Athanas, P.: Hardware accelerated sat solversa survey. Journal of Parallel and Distributed Computing (2016)
28. Soos, M.: The cryptominisat 5 set of solvers at sat competition 2016. SAT COMPETITION 2016 p. 28 (2016)
29. Van Der Tak, P., Heule, M.J., Biere, A.: Concurrent cube-and-conquer. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 475–476. Springer (2012)