Diversifying a Parallel SAT Solver with Bayesian Moment Matching

Vincent Vallade[1](✉), Saeed Nejati[4][0000−0002−1473−3630], Julien Sopena[1,2], Souheib Baarir[1,3], and Vijay Ganesh[5][0000−0002−6029−2047]

[1] Sorbonne Université, CNRS F-75005 Paris, France
vincent.vallade@lip6.fr
[2] Sorbonne Université, Inria F-75005 Paris, France
[3] Paris Nanterre University, CNRS F-75005 Paris, France (Now at EPITA)
[4] Amazon Web Services, Seattle, USA
[5] University of Waterloo, Waterloo, Canada

Abstract. In this paper, we present a Bayesian Moment Matching (BMM) in-processing technique for Conflict-Driven Clause-Learning (CDCL) SAT solvers. BMM is a probabilistic algorithm which takes as input a Boolean formula in conjunctive normal form and a prior on a possible satisfying assignment, and outputs a posterior for a new assignment most likely to maximize the number of satisfied clauses. We invoke this BMM method, as an in-processing technique, with the goal of updating the polarity and branching activity scores. The key insight underpinning our method is that Bayesian reasoning is a powerful way to guide the CDCL search procedure away from fruitless parts of the search space of a satisfiable Boolean formula, and towards those regions that are likely to contain satisfying assignments.

1 Introduction

Modern Conflict-Driven Clause-Learning (CDCL) SAT solvers have been used successfully to solve a wide variety of real-world problems, coming from a variety of domains such as hardware and software verification/testing [3, 5], security [2], cryptography [11], and resolving mathematical conjectures [4].

In their paper [7], Duan et al. present a Bayesian Moment Matching (BMM) based probabilistic learning algorithm that was used as a pre-processor to a CDCL SAT solver with the aim of providing an initial assignment for the solver's search to start from (the problem of finding an optimal initial assignment to start a solver's search from is often referred to as the initialization problem). The BMM method for the Boolean SAT problem takes as input a Boolean formula in Conjunctive Normal Form (CNF) and a probability assignment $P(x = T)$ for every variable x that captures the likelihood of that variable x being true according to the method (where the corresponding joint probability distribution over the value assignment to the variables of the input formula is referred to as the *prior*), and outputs a joint probability distribution or posterior that is most likely to maximize the number of satisfied clauses. The BMM method repeatedly

applies Bayesian inference update rule on the input distribution and uses each clause in the input formula as evidence in order to compute the output posterior. The learned probabilities collectively represent an assignment that most likely satisfies most of the clauses (if not all). While Dual et al. report excellent results of using BMM as a pre-processor to solve the initialization problem, they didn't use it in any other way in their solver.

In this paper, we propose a BMM-based in-processing technique to update the polarity and branching scores of a CDCL solver. The choice of the prior distribution can affect the quality of the learned posterior distribution, and BMM uses Bayesian inference starting from a random prior. Therefore, running the same solver with different initial seeds could lead to different performance results. However, this behavior can be exploited in parallel portfolio settings, where the solvers are run with different priors. Therefore, in this paper we also evaluate the use of BMM approach for diversification of parallel portfolio solvers.

2 Algorithm Description

We refer the reader to [7] for a full description of the BMM method. We use the same methodology by applying the BMM method as a preprocessing mechanism to initialize the polarity and branching order (or activity) of the variables. In this paper, we go further and use the BMM method as an in-processor to update these same metrics. Algorithm 1 gives an overview of CDCL, along with the BMM update technique (see the shaded instructions).

We recall that the CDCL algorithm is based on a main loop that first applies `unitPropagation`[6] on the formula $\mathcal{F}$ simplified by the current assignment $\mathcal{A}$ (Line 6). If the formula is empty, the algorithm returns *true* (Line 9), and $\mathcal{A}$ is the model. If the formula implies an empty clause, then two scenarios are possible: (*i*) we are at level 0 and the algorithm returns $false$ (Line 13); (*ii*) otherwise we deduce the reasons for the empty clause and a backjump point is computed (Lines 15-18). Otherwise, a new literal is selected to make progress in the resolution of $\mathcal{F}$ (Lines 23-24). Note that *lvl* represents the number of decisions in the current branch, often called *decision level*.

We augment the aforementioned algorithm with the BMM procedure as a pre-processor, as well as an in-processing search re-initializer of variable polarities and activities (guiding the algorithm at key points of its progression). The shaded instructions in algorithm 1 implement both the *pre-processing* and *in-processing* steps. These steps are described in the following paragraphs.

Pre-processing step: BMM is called to initialize the polarity and activity of the variables (Line 4). During pre-processing, the input to BMM are the input clauses and a randomly-generated prior distribution for each variable. Because this step is executed once, we can afford more computational cost to get to a more accurate posterior distribution. Therefore the number of passes over the set of input clauses is set to $K = 100$.

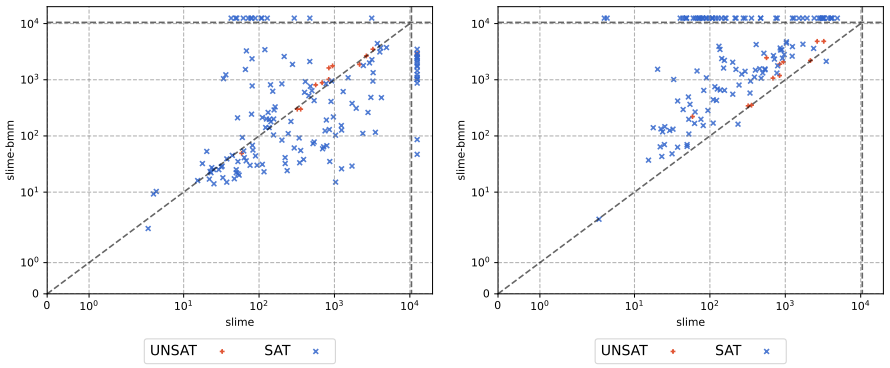
[6] The `unitPropagation` function implements the Boolean constraint propagation (BCP) procedure that forces (in cascade) the values of variables in unit clauses [6]

Algorithm 1: Conflict-driven clause learning algorithm with Bayesian Moment Matching (CDCL + BMM).

```
1  function CDCL(F: CNF formula)
       /* returns true if F is SAT else false (UNSAT)                    */
2      A ← ∅                                          // Current assignment
3      lvl ← 0                                      // Current decision level
4      bmmUpdate()                                   // Call BMM with K = 100
5      forever
6          (F′, A′) ← unitPropagation(F|_A)
7          A ← A ∪ A′                          // Add propagated literals in A
8          if F′ = ∅ then
9              return true                                        // F is SAT
10         end
11         if ∅ ∈ F′ then                // There is a conflict to be analysed
12             if lvl = 0 then
13                 return false                                   // F is UNSAT
14             end
15             C ← conflictAnalysis(F, A)
16             F ← F ∪ {C}
17             lvl ← backjumpAndRestart(lvl, C, ...)
18             A ← {ℓ ∈ A | δ(ℓ) ≤ lvl}
19         else
20             if threshold limit is reached then
21                 bmmUpdate()                        // Call BMM with K = 10
22             end
23             A ← A ∪ {pickBranching()}   // Pick a new decision literal
24             lvl ← lvl + 1
25         end
26     end
27 end
```

In-processing step: When the unit propagation reaches a fix point (Line 21), i.e. no unit clauses left to propagate and there is no conflict, BMM procedure is called to re-evaluate the probabilities for all variables. The difference here is that the prior distribution given to BMM is determined by the assignment trail (probability of variables on the trail is set to zero or one according to their polarity), and the rest of the variables are updated based on this prior. Moreover, a subset of the learnt clauses is considered as evidence by BMM (those learnt clauses that have a LBD [1] value less than or equal to 3). Unlike the pre-processing step, this step is executed several times during the search process, therefore we limit the number of passes over the set of clauses to $K = 10$.

The BMM update process has a considerable overhead, therefore we only call the update whenever a certain threshold is reached. This threshold is crossed when the solver has restarted 50 times and whenever the number of variables in the "current" assignment trail exceeds 40% of the total number of variables in



(a) Best runs (b) Worst runs

s

Fig. 1: Scatter plot showing the performance of the `slime-bmm` **vs** `slime` **sequential solvers** on the 2021 SAT Competition Crypto benchmark.

the input formula or the current trail size is larger than 90% of the largest trail seen so far [7]. The learned posterior distribution over all variables together with the trail are used in the `pickBranching()` of the CDCL to further extend the search tree.

The pre-processing step is similar to the pre-processing component in [7], however, the in-processing step has a novel design. In the original BMM paper, authors only update the BMM probabilities when a unary or binary clause is learned, and use those clauses as new evidence. Learning unary and binary clauses mean that solver can learn valuable information from the search subspace that it is exploring. However, it is equally important to help guide the search of the CDCL solver whenever the BCP has reached a fix-point (i.e., is not making progress) by modifying the polarity and activity prior to branching. Therefore, we designed this BMM update method to be called to guide the solver's search whenever the BCP is not making progress (i.e., there are no prospects of further learning without making decisions).

3 Evaluation of `slime` and `slime-bmm` Sequential Solvers

We chose to compare the efficacy of our BMM technique in the context of crypto benchmarks, given its previous success in this domain [11]. We chose the winner of the crypto track of the SAT competition 2021[8], called `slime` [12], as the base CDCL engine for our implementation[9], and evaluated it on the crypto track benchmarks from the same competition.

[7] These magic numbers are borrowed from the base solver that we used for our implementation

[8] https://satcompetition.github.io/2021/

[9] `https://github.com/lip6/painless/tree/bmm`

Table 1: This table shows performance of `p-slime` and `p-slime-bmm` on the 2021 Crypto Track.

Solvers	PAR2	UNSAT	SAT	TOTAL (200)
p-slime-bmm-50%	133H36	13	147	160
p-slime-bmm-75%	136H05	13	145	158
p-slime-bmm-90%	138H39	13	144	157
p-slime	142H03	13	142	155
p-slime-bmm-25%	147H42	13	140	153

It is worth noting that `slime` uses an in-processing approach similar to ours: a Stochastic Local Search (SLS) engine is used when the BCP reaches a fix point to guide the search. So, we replaced this component by our BMM procedure along with the removal of all heuristics related to the SLS sub-routine. It is this version that we refer to as `slime-bmm`.

Since the initialization of the BMM component induces randomness, we ran `slime-bmm` 10 times on each instance. As the `slime` configuration that won the competition was deterministic [12], a simple run of this latter was sufficient.

The scatter plots of fig. 1 show the results of our experiment. Plot of fig. 1a (respectively fig. 1b) highlights the scores of solvers with respect to the best (respectively worst) runs. The axis are running time in seconds on a logarithmic scale. If both solvers timeout for an instance, it is not represented in the figures. So we can focus on the instances where one of the two solvers stands out. Even though the timeout is fixed at 5000s, for readability, a timeout run is shown with a point beyond the grey dotted line, above 10^4 seconds.

Here we can observe that in the case of the best runs, `slime-bmm` is more effective than `slime`, solving 6 more instances. On the other hand, when looking at the worst case, `slime` is more stable and outperforms `slime-bmm`.
There are two main takeaways from the experimental results we observed in this section. First, BMM is a good candidate for an in-processing component that can help guide the CDCL search. Compared to other solving engines that are used for in-processing component (e.g. SLS), our experience was that BMM performs well right out of the box, without too much tuning of the heuristics. Second, running sequential solvers with different random priors given to BMM makes the solver explore different solutions (for satisfiable instances), which is a great opportunity for parallel portfolio solvers. Thus, in the next section we explored the possibility of using BMM with different priors as a means of diversifying the set of worker solvers in a parallel portfolio setting.

3.1 Architecture of the `p-slime-bmm` Parallel Portfolio Solver and Results

We named the resulting parallel solvers `p-slime` and `p-slime-bmm`. To implement these solvers, we used Painless infrastructure [9], a framework that eases the implementation of parallel SAT solvers for many-core environments. The parallelization and sharing strategies we implemented are the same as the one used

by the winner of the parallel track of the SAT competition 2021 `p-mcomsps` [13]. Both `p-slime` and `p-slime-bmm` are portfolio solvers [8]. Therefore each instance of a CDCL engine (thread) is launched on the entire formula. The sharing strategy is based on the Literal Block Distance (LBD) measure: the LBD of a clause is the number of decision levels represented in that clause [1]. Initially the thread responsible for sharing receives every clause with a LBD inferior or equal to 2. This distribution happens in an asynchronous manner between the solver threads and the sharing threads. After some predetermined round of sharing, if the sharing threads received too much/not enough clauses from a particular solver, it will dynamically decrease/increase the LBD limit for this solver. In `p-slime-bmm`, the cdcl engines are either `slime` or `slime-bmm` and we make the proportion of `slime-bmm` vary between the different versions of the solver from 25% to 90%. The aim here is to ensure a total collaboration between the solvers (of the portfolio) to get a maximum of solved instances. As a deterministic solver would not make sense in the parallel context, we use `slime` diversification mechanism, which consists in fixing a random polarity to each variable, for each `slime` in the portfolio.

Both solvers were run on a cluster of 12-core Intel Xeon CPU E5645, with 64 GB of RAM, a timeout of 5000s, setting the framework to launch 10 sequential solvers. In this performance study, we use the following success metrics: penalized average runtime (PAR-2) sums the execution time of a solver and penalizes the executions that exceed the timeout with a factor 2; the number of instances solved. As observed in table 1, it seems that having both algorithms in equal proportion is the sweet spot, as `p-slime-bmm-50%` solves 5 more SAT instances than `p-slime`, resulting in a much better PAR-2. Hence, the new proposed solver proves to be more efficient than the state-of-the-art `p-slime`.

4 Conclusion

In this paper, we presented a Bayesian Moment Matching (BMM) in-processing technique for CDCL SAT solvers. We invoked this BMM method, as an in-processing technique after Boolean Constraint Propagation and before branching is called in a CDCL SAT solver, with the goal of updating the polarity and branching activity scores. Bayesian reasoning has proven to be a powerful way to guide the CDCL search procedure away from fruitless parts of the search space of a satisfiable Boolean formula. We experimented massively our approach on cryptographic instances and under sequential settings. The outputs were positives for some random seeds and not others (given the probabilistic nature of BMM). This led us to develop a portfolio parallel solver using our new algorithm as a back-end engine along with the standard engines. The resulting derived hybrid parallel solver showed good performances with respect to the vanilla solver. Using an incremental approach, we found that 50% of `slime-bmm` is a good proportion but it may not be the most optimal solution. The next step will be to fine-tune the proportion of `slime-bmm` in the portfolio based on the work done in [10] using a Multi-Armed Bandit approach.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of the 21st International Joint Conferences on Artifical Intelligence (IJCAI). pp. 399–404. AAAI Press (2009)
2. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: Aeg: Automatic exploit generation. In: Network and Distributed System Security Symposium (Feb 2011)
3. Bradley, A.R.: Sat-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer (2011)
4. Bright, C., Kotsireas, I., Ganesh, V.: The science of less-than-brute force: When satisfiability solving meets symbolic computation. In: Communications of the ACM (CACM). ACM (2022)
5. 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)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commununications of the ACM **5**(7), 394–397 (1962)
7. Duan, H., Nejati, S., Trimponias, G., Poupart, P., Ganesh, V.: Online bayesian moment matching based sat solver heuristics. In: International Conference on Machine Learning. pp. 2710–2719. PMLR (2020)
8. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel sat solver. Journal on Satisfiability, Boolean Modeling and Computation **6**(4), 245–262 (2009)
9. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Painless: a framework for parallel sat solving. In: Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 233–250. Springer (2017)
10. Liang, J.H., Oh, C., Mathew, M., Thomas, C., Li, C., Ganesh, V.: Machine learning-based restart policy for CDCL SAT solvers. In: Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. pp. 94–110 (2018). https://doi.org/10.1007/978-3-319-94144-8_6, `https://doi.org/10.1007/978-3-319-94144-8_6`
11. Nejati, S., Ganesh, V.: Cdcl (crypto) sat solvers for cryptanalysis. arXiv preprint arXiv:2005.13415 (2020)
12. Riveros, O.: Slime sat solver. In: Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. p. 37. Department of Computer Science, University of Helsinki, Finland (2021)
13. Vallade, V., Le Frioux, L., Oanea, R., Baarir, S., Sopena, J., Kordon, F., Nejati, S., Ganesh, V.: New concurrent and distributed painless solvers: P-mcomsps, p-mcomsps-com,p-mcomsps-mpi, and p-mcomsps-com-mpi. In: Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. p. 40. Department of Computer Science, University of Helsinki, Finland (2021)