CDCL SAT Solver

Weiming Zhang, Baihong Qian, Jialu Shen

Abstract

In homework 3 we implemented a basic SAT solver based on CDCL and choose the Variable State Independent Decaying Sum(VSIDS) as the branching heuristic to pick the next variable. In this paper, our group work further to implement Learning Rate Based(LRB) and Conflict History Based(CHB) as branching heuristics to realize possible acceleration. We also incorporate a restart policy to avoid difficult decisions and implemented UCB and EXP3 algorithms to select the best branching heuristic as mentioned before.

1 Introduction

Modern Boolean SAT solvers are a critical component of many innovative techniques in many fields and are widely used in a large variety of crafted and real-world problems. SAT problems have been proved to be NP-complete, but a great number of researchers are dedicated to exploring and improving algorithms that can accelerate solving instances involving a a huge number of variables and clauses. In particular, Conflict Driven Clause Learning(CDCL) SAT solvers perform outstandingly in solving large CNFs by learning new clauses during boolean constraint propagation.

A key element in CDCL SAT solvers is the branching heuristic, which plays the role of picking the next variable to branch on. How to choose the next variable to avoid overhead conflicts becomes significant in improving the overall efficiency. In our implementation of homework 3, the Variable State Independent Decaying Sum(VSIDS) was chosen to score the variables by their activity. In 2016, more efficient branching heuristics were proposed by researchers. They viewed online variable selection in SAT solvers as optimization problems and modeled the variable selection optimization as an online multi-armed bandit, which is a special-case of reinforcement learning. Our group implement two representative efficient branching heuristics, Learning Rate Based(LRB) and Conflict History Based(CHB) based on a well-known multi-armed bandit algorithm called exponential recency weighted average.

However, when solving large and complex CNFs, the CDCL solver might get stuck in a part of the search space that contains no solution. The solver can often get into such situation because some incorrect assignments were committed early on and unit resolution was unable to detect them. So our group implement a restart policy where our SAT solver can restart after a certain number of conflicts is detected to avoid being stuck. We view branching heuristics selection when restarting as a Multi-Armed Bandit model, and set up some reward and feedback evaluation for each branching heuristic. In this project we implement UCB, Epsilon-Greedy and EXP3 algorithm.

2 Preliminaries

2.1 Conflict Driven Clause Learning

The feature and advantage of CDCL SAT solvers are that they analyze every conflict it encounters to learn new clauses to block the same conflicts. The solver maintains an implication graph, a directed acyclic graph where the vertices are assigned variables and edges record the propagation between variables induced by Boolean constraint propagation. Whenever the solver comes through a conflict

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

analysis, the solver analyzes the implication graph and cuts the graph into the conflict side and the reason side. A learnt clause is generated on the variables from the reason side incident to the cut by negating the current assignments to those variables. In practice, the implication graph is typically cut at the first unique implication point (the First UIP). We achieve this by deducing from the conflict clause and resolute repeatedly by virtue of trails and finally we can get the first UIP when there is only one literal on the conflict level. After learning a new clause, remove all literals from trail with higher decision level and backtrack to an earlier state with no conflicts.

2.2 Variable State Independent Decaying Sums Branching Heuristic

VSIDS can be seen as a ranking function that maintains a floating point number for each Boolean variable in the input formula, which can be viewed as activity. In our SAT problems, this heuristic maintains a floating point score for each variable and initially set to 0. When a conflict occurs, the activity of some variables is bumped and increased by 1. Furthermore, the variable activities are multiplied by a decaying factor between 0 and 1. VSIDS ranks variables and selects the unassigned variable with the highest activity to branch on next, which is the decision stage of CDCL solver.

2.3 Exponential Recency Weighted Average

Exponential recency weighted average (ERWA) is a technique to estimate a moving average incrementally by giving more weight to the more recent outcomes. Suppose a stream of outcomes $x_1, x_2, x_3, \cdots, x_n$. We can comput exponentially decaying weighted average of the outcomes:

$$\overline{x} = \sum_{i=1}^{n} \omega_i x_i, where\, \omega_i = \alpha(1-\alpha)^{n-i}$$

Here $\alpha \in [0, 1]$ is a factor that determines the rate at which the weights decay. To reduce computational overhead, the moving average can be computed incrementally by updating after each outcome:

$$\overline{x}_{n+1} = (1-\alpha)\overline{x}_n + \alpha x_{n+1}$$

ERWA is widely used in bandit problems where we need to select which arm to play at each time step in order to maximize its long-term reward. ERWA is a simple technique to estimate the empirical average of each arm. The LRB and CHB branching heuristics we implement are updated with the method of ERWA.

2.4 Multi-Armed Bandit Problem

A Multi-Armed Bandit is a reinforcement learning problem consisting of an agent and a set of candidate arms from which the agent has to choose while maximizing the expected gain. The agent relies on information in the form of rewards given to each arm and collected through a sequence of trials. An important dilemma in MAB is the tradeoff between exploitation and exploration as the agent needs to explore underused arms often enough to have a robust feedback while also exploiting good candidates which have best rewards. In our implementation, we adopt MAB strategy in restarting when selecting the next restart branching heuristic.

3 Branching Heuristics

3.1 LRB

LRB stands for Learning Rate Based Branching Heuristic. Its most significant contributions are introducing learning rate and using the learning rate in choosing variable to be assigned based on Multi-armed bandit.

The learning rate of a variable v at interval I is defined as

$$r = P(v, I)/L(I)$$

, in which P(v, I) is the number learnt clauses in which v participates during interval I and L(I) is the number of learnt clauses generated in interval I.

In implementation, LRB can be divided in three parts.

3.1.1 Exponential Recency Weighted Average

In this part, several containers are introduced. Learned counter is the number of learnt clauses generated by the solver. lrb_scores stores the EMA estimate of each variable. lrb_assigned stores when each variable was last assigned. lrb_participated stores the number of learnt clauses each variable participated in generating since it was assigned.

After a learnt clause is generated from conflict analysis, Learned counter and lrb_participated are updated according to their definition. When a variable transitions from assigned to unassigned, ERWA will calculate its learning-rate r and update the reward using the formula

$$Q = (1 - \alpha) * Q + \alpha * r$$

. When the solver requests the next branching variable, ERWA will select the variable with the highest reward.

3.1.2 Reason Side Rate

A variable reasons in generating the learnt clause if it appears in a reason clause of a variable in a learnt clause l, but does not occur in l. So lrb_reason is introduced to store the number of learnt clauses each variable reasoned in generating since assigned.

Reason Side Rate is defined as

$$rsr = A(v, I)/L(I)$$

, in which A(v, I) is the number of learnt clauses which v reasons in generating in interval I.

The formula updating reward becomes

$$Q = (1 - \alpha) * Q + \alpha * (r + rsr)$$

3.1.3 Locality

Inspired by the VSIDS decay, this extension multiplies the lrb_scores of every unassigned variable v by 0.95 after each conflict, that is

$$Q = Q * 0.95$$

The decay is used to restrict exploration supposing that variables with high learning rate also exhibit locality.

3.2 CHB

CHB stands for Conflict History Based Branching Heuristic, which is first used to solve the bandit problem, is a branching heuristic stemed from the Exponential Recency Weighted Average algorithm. Its principle is ranking the importance of variables and thus deciding which to be assigned by maintaining a reward value.

3.2.1 Rewarding

In our learning of CDCL problem, it's an intuition that the variables involved in conflicts are key variables and need careful evaluation. Therefore, employing a function to calculate the involvement of variables in conflicts is proper and applicable. The reward is obtained after constraint propagation. The numerator is decided by its occurrence in the conflict. The denominator is decided by the number of new conflicts. After a new clause is learned by conflict analysis, all essential values are ready and the calculation is done at the beginning of every loop.

$$reward = \frac{multiplier}{numConflicts - lastConflict[v] + 1}$$

3.2.2 Updating

CHB maintains a Q score. Q score is updated as following:

$$Q = (1 - \alpha) * Q + \alpha * r_v$$

. It combines the history and the reward value mentioned before. With a slightly decreasing coefficient α, it lasts long for the propagation and conflict analysis process. Q values of variables involved are always updated in the loop. The algorithm branches on unassigned variables to select the one with the largest Q value and assign it a value. This course ensures the flexibility and validity of assignments and propagation.

$$v^* = argmax_{v \in unassigned} Q[v]$$

4 Restart

Though we've implemented two improved bandit-based branching heuristics, confronted with different SAT problem states, some of the branching heuristics perform very well while others perform inefficiently by being stuck in over-complicated conflicts. So we consider combining them with a restarting scheme, which helps to deal with the heavy-tailed phenomena in SAT, to switch between these heuristics thus ensuring a better and more diverse exploration of the search space. In a single propagation, when the number of conflicts reach a certain threshold, then we clear the assignments and select a new heuristic to assign new variables from the beginning.

Here we introduce Multi-Armed Bandit (MAB) strategies and treat branching heuristics as arms to decide on which heuristic to restart with. We will calculate and update reward for each arm from the performance in the unfinished CDCL propagation. In this way we can switch to a better and more diverse exploration of the search tree.

4.1 UCB

Upper-Confidence Bound(UCB) is a well-known strategy for MAB. We first implement UCB as our restart strategy.

Let $A = \{a_1, \cdots, a_K\}$ be the set of arms for MAB containing candidate heuristics and here in our project $K = 3$. The framework selects a heuristic a_i where $i \in \{1 \cdots K\}$ at each restart. To choose an arm, MAB strategies generally rely on a reward function calculated during each run to estimate the performance of the chosen arm. We adopt a reward function that estimates the ability of a heuristic to reach conflicts quickly and efficiently:

$$r_t = \frac{\log_2(decision_t)}{decidedVars_t}$$

Here t denotes the current run, and $decisions_t$ and $decidedVars_t$ respectively denote the number of decisions and the number of variables fixed by branching in the run t.

Then we construct the main framework of UCB strategy. For this, the following parameters are maintained for each candidate arm $a \in A$:

- $n_t(a)$ is the number of times the arm a is selected during the t-1 previous runs.
- $\hat{r}_t(a)$ is the empirical mean of rewards of arm a over the t-1 previous runs.

In our previous homework 5 for MAB problems, we at first enumerate all arms to ensure stability, and here we take the same idea. And in the later looping we need to compare each run's upper confidence bound to make a best selection. Here we define:

$$UCB(a) = \hat{r}_t(a) + \sqrt{\frac{4\ln(t)}{n_t(a)}}$$

4.2 Epsilon-Greedy

Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly. As epsilon refers to the probability of choosing to explore, this strategy exploits most of the time with a small chance of exploring. That is, with probability epsilon, an arm is chosen randomly, while in other case, the arm with highest sample mean is chosen. The sample mean is the same as reward defined in UCB.

4.3 EXP3

EXP3 is another well-known strategy for MAB. Its idea is to store and update each arm's weight and possibility. The framework randomly select an arm according to the possibility and update each arm's weight.

Then we construct the main framework of EXP3 strategy. For this, the following parameters are maintained for each candidate arm $a \in A$:

- $p_t(a)$ is the possibility of each arm after t-1 previous runs.
- $w_t(a)$ is the weight of each arm after t-1 previous runs.

We update possibilities and weights as follows:

$$p_t(a) = (1-\gamma)\frac{w_t(a)}{\sum\limits_{j=1}^{K} w_t(j)} + \frac{\gamma}{K}$$

$$w_{t+1}(a) = w_t(a)e^{\frac{\gamma r}{K}}$$

Here r represents the rewards, which has the same definition in UCB.

4.4 restart strategy

Restarting is a good choice when our SAT solver faces over-complicated conflicts, but if we set the conflict limits to a common value, it is possible that our SAT solver would be trapped into a never-ending loop. To avoid this problem, our group proposed two strategies as follows.

The first one is to set a certain iteration value for learning and updating. After certain numbers of runs, we consider that our SAT solver has learnt enough and has its own evaluation for each branching heuristic. And then we choose one best with the MAB and withdraw the limit for conflict limits.

The other strategy is to change the conflict limits by multiplying a constant each time we restart. This bears some similarities with the RR Strategy, but here we use the MAB selector, not by turns. Even we do not stop, the conflict limits grow in exponential speed, so it will neither be looping endlessly nor get stuck easily.

5 Further Implementation

5.1 Pre-Processing

Pre-processing SAT instances can reduce their size considerably. We implement subsumption and self-subsumption and show that these techniques not only shrink the formula, but also decrease runtime of SAT solvers substantially.

5.1.1 Subsumption

A clause C1 is said to (syntactically) subsume C2 if $C1 \subseteq C2$. A subsumed clause is redundant and can be discarded from the SAT problem. Particularly, a subsumed clause never needs to be part of a resolution proof of unsatisfiability.

In implementation, we introduced cl_sig as a container to simplify the process of comparing. For each clause a 64-bit signature is stored. The signature abstracts the set of literals of a clause in the following way: A hash function h maps literals to numbers 0 to 63, and the signature of a clause C is calculated as the bit-wise Or of $2^{h(p)}$ over its literals $p \in C$.

Then we check if a clause subsumes (as opposed to being subsumed by) some other clause in the database to eliminate redundant clauses.

5.1.2 Self-Subsumption

Also, similar clauses of a particular kind may occur: one clause C2 almost subsumes a clause C1, except for one literal x, which, occurs with the opposite sign in C2. For instance, let C1 = x, a, b, and C2 = x, a, then resolving on x will produce C3 = a, b, which subsumes C1. Thus after adding C3 to the CNF, we can remove C1, in essence eliminating one literal. In this case, we say that C1 is strengthened by self-subsumption using C2. This simplification rule is called self-subsuming resolution.

6 Results and Analysis

6.1 Experiments

Our group takes advantage of the framework of the solution of homework 3 and makes improvement on it. At first we test the CDCL solver on bmc-1.cnf, and it runs for about 800 seconds. We also tested that bmc-7.cnf needs 10 seconds.

After we implemented LRB and CHB branching heuristics, we also tested them on bmc-7.cnf and bmc-1.cnf:

- For LRB, bmc-1.cnf takes 1200 seconds and bmc-7.cnf takes 7.7 seconds.
- For CHB, bmc-1.cnf takes 580 seconds and bmc-7.cnf takes 4.6 seconds.

It is shown that our new heuristics perform well in solving large SAT problems and have some acceleration effects. CHB is faster than the other two in our implementation.

In the next stage we implemented a restart scheme, using the framework of Multi-Armed Bandit. We implemented relevant algorithms like UCB and EXP3. We find that using our strategy with exp-growing conflict limits, running examples of bmc cannot show the advantage of restart scheme.

At last we added a pre-processing method. Combine this with restart scheme, we get a faster outcome. But due to time limit, there exist some problems with the pre-processing scheme such as getting stuck.

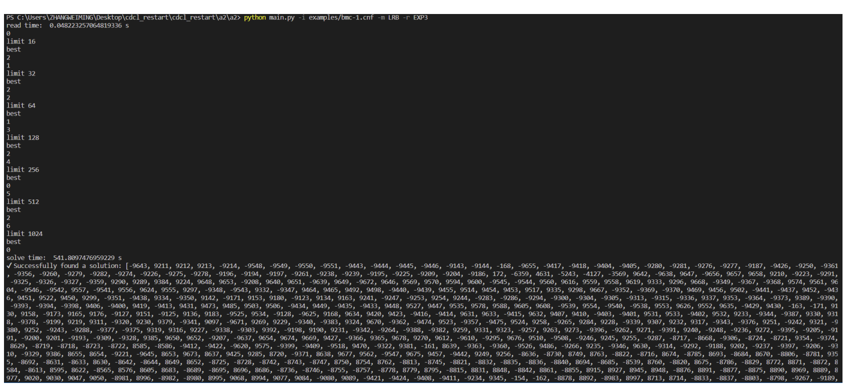

Figure 1: Preprocessing with restart to solve bmc-1

6.2 Obstacles and intuition

The core task of the project is to implement more comprehensive and efficient strategies for our SAT solver. Our group began our attempts on the implementation of new branching heuristics at first. Our group adopted a different phasing method from our original VSIDS heuristics and encountered some problems that the solver ran too slowly. So we changed our method, referred to the pseudocode provided by the paper and finally achieved acceleration. Meanwhile, a hard problem troubled us for a while, which is how to get the "reason" set. Then we solved it by adding all relevant variables during conflict analysis, and do some exclusion when computing.

More heuristics increased the complexity of the code framework. So we use a class to store the whole SAT solver, and each branching heuristic corresponds to a solving function. We initialize all variables when initializing the class. Similarly, we construct the restart scheme with another class for convenience.

When experimenting on the performance of restarting, we found that it was generally slower because sometimes restart could cause wasting. But we did not compare on large examples given by the teacher and on larger problems restart scheme would prove its advantage. To avoid being stuck, we combine our restart with the round-robin restart scheme and take advantage of it in the form of MAB. We set the conflict limits double itself when restarting so that it wouldn't be stuck or loop endlessly.

During our whole experimenting process, we find our CDCL solver not fast due to the fact that bmc-1.cnf would take several minutes, though through our improvement it does have acceleration effect. We find that the time killer might lie in the original structure of our CDCL implementation in homework 3. The data structure used by us is not so optimal so it results in more time on boolean constraint propagation.

Furthermore, we did some additional improvement on the preprocessing of the clauses.

Due to time limit and the state of our group members, we failed to do more work on accelerating our CDCL and there exist some problems with our pre-processing scheme, but I think there is a big promising point to make our CDCL solver faster, which lies in the data structure and BCP function in our original framework.

References

[1] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Exponential recency weighted average branching heuristic for SAT solvers. In: Proceedings of AAAI 2016 (2016)

[2] Armin Biere. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In Hans Kleine Büning and Xishun Zhao, editors, Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings, volume 4996 of Lecture Notes in Computer Science, pages 28–33. Springer, 2008. doi: 10.1007/978-3-540-79719-7_4.

[3] 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)

[4] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Kissat_MAB: Combining VSIDS and CHB through Multi-Armed Bandit. In Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions, volume B-2021-1 of Department of Computer Science Series of Publications B, pages 15–16. University of Helsinki, 2021.

[5] Jia Hui Liang, Chanseok, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. Maple COMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions, page 20–21, 2017.

[6] Audemard, G., and Simon, L. 2012. Refining restarts strategies for SAT and UNSAT. In Principles and Practice of Constraint Programming, 118–126. Springer.

[7] Balint, A.; Belov, A.; Heule, M. J. H.; and J¨arvisalo, M. 2013. Solver and benchmark descriptions. In Proceedings of SAT Competition 2013, volume B-2013-1. University of Helsinki.

A Appendix

A.1 Group Contribution

Weiming Zhang: Help implement two branching heuristics and solve some problems. Conduct the construction of restart scheme and implement UCB and EXP3 algorithms. Adjust the whole code framework. Contribution:40%

Baihong Qian: Implement the first version of LRB. Implement Priortity Queue, Luby heuristic in restart and pre-processing, trying to optimize. Implement Epsilon-Greedy algorithm. Contribution:40%

Jialu Shen: Implement the first version of CHB. Make the PPT for the presentation. Contribution:20%