A SAT solver using CDCL equipped with several advanced techniques

Min Wu, Yiwen Song, Chang Su
Department of Computer Science and Technology,
Shanghai Jiao Tong University, Shanghai, China
{qiemanqieman,phoebe_syw,suchang0912}@sjtu.edu.cn

Abstract

Applying advanced techniques in various aspects to CDCL solvers can significantly improve the performance of vanilla CDCL algorithm. In our project, for better branching heuristics, we implement VSIDS heuristic, LRB series of algorithms (including ERMA & RSR & LRB) and CHB algorithm. For restarting, we implement effective machine learning-based restart policy (MLR), and apply UCB algorithm to switch between candidate branching heuristics after restarting. We also equipped our CDCL solver with preprocessing techniques, including pure literal elimination and NiVER. At last, enough experiments were done to prove the effectiveness of our algorithm equipped with different techniques we implemented. Meanwhile, we also improved the data structures and applied algorithms with lower computational complexity.[1]

1 Introduction

Conflict-driven clause-learning (CDCL) SAT solvers are the dominant solvers in practice today, take formulas specified in conjunctive normal form (CNF) as input, and decide whether they are satisfiable. Applying several advanced techniques in various aspects to CDCL solvers can significantly boost the vanilla CDCL algorithm, including better branching heuristics, effective restart policy, and appropriate preprocessing technique.

In our project, we met the requirements from 1 to 4. As Section 2 shows, we first improved the data structures which could allow better performance, and encapsulated each module we were going to implement as a class. Section 3 presents the our implement for the requirements. We have improved the VSIDS heuristic, and then implemented LRB[1] series of algorithms (including ERMA & RSR & LRB) and CHB[2] algorithm for better branching heuristics, and the implementation details can be seen in Section 3.1. In Section 3.2, we equipped our CDCL with effective machine learning-based restart policy (MLR)[3], and applied classic bandit algorithm UCB to switch between candidate branching heuristics after restarting[4]. In Section 3.3, pure literal elimination[6] and non increasing variable elimination resolution (NiVER)[5], one of classic bounded variable elimination (BVE) algorithms, are implemented as preprocessing techniques to accelerate SAT solving. Meanwhile, during our implement, we used effective data structures and applied algorithms with lower computational complexity, to further speed up our CDCL solver.

Finally, in order to show our implementation more clearly and to make it easier to select desired parameters, we wrote a simple GUI for visualization. We also wrote a test script to evaluate our algorithm equipped with different advanced techniques we have implemented with enough experiments, and compared the effect of different techniques in Section 4. And according to the experimental results, we can see that our implementation significantly improves the performance of the vanilla CDCL algorithm.

[1] Our implementation is available at `https://github.com/qiemanqieman/advanced-sat-solver`.

2 Fundamental changes

2.1 Data structure

After performance analysis, we found that more than 90% of our SAT solver's time was spent on CRUD of the elements on list, which are quite time-consuming operations compared to the data structures of set, dict, and other hash-like data structures. So we redesigned the most important data structure based on set, dict, which is *AssignInfo* (defined in *ai.py*). Based on it, the time complexity of the major operations of the solver is reduced from O(N) to O(logN). This greatly improves the performance of the solver.

2.2 Modules

In order to make the program clearer and more structured, and facilitate the division of the workload and cooperation of the team, we divided the whole solver into different modules. In *heuristics*, we implemented several different branching heuristics. Restart policy is implemented in *restart*, bandit algorithm in *bandit*, and preprocessing techniques in *preprocess*.

3 Requirements

3.1 Better branching heuristics

Modern conflict-driven clause-learning SAT solvers routinely solve large real-world instances with millions of clauses and variables in them. Their success crucially depends on effective branching heuristics.In the past two decades, a large number of innovative heuristics have emerged, and we have mainly tried to implement *VSIDS-based, LRB-based* and *CHB-based* branching heuristics. In the beginning, we optimized the performance of *VSIDS* based on previous work. Then we implemented *LRB-based* branching heuristics—including *ERWA, RSR* and *LRB*. Finally, we implemented *CHB*. And we designed an abstract base class *Heuristic* in *heuristic.py*. all branching heuristics must inherit it and implement its abstract method(s).

Since its introduction in 2001, *VSIDS* has remained one of the most effective and dominant branching heuristic despite intensive efforts by many researchers to replace it. In early 2016, researchers have provided the first branching heuristic that is more effective than *VSIDS* called the conflict history-based (*CHB*) branching heuristic. And another branching heuristic implemented in our solver, referred to as learning rate branching (*LRB*), significantly outperforms *CHB* and *VSIDS*.

Let's take them step by step.

3.1.1 Improved VSIDS

The development of various heuristics, notably the Variable State Independent Decaying Sum (*VSIDS*) branching heuristic (and its variants) and conflict analysis techniques, have dramatically pushed the limits of CDCL solver performance. So as a good start, we tried to implement it first and improved its performance as much as possible. In fact, we implemented *VSIDS* exactly as presented in the course, all we improved was the data structure used, which has been introduced in Section 2.1.

3.1.2 ERMA & RSR & LRB[1]

In learning rate branching heuristic, online variable selection in SAT solvers is viewed as an optimization problem. The objective to be maximized is called the *learning rate* (LR), a numerical characterization of a variable's propensity to generate learnt clauses. The goal is therefore to select branching variables that will maximize the cumulative LR during the run of the solver.

At first, the optimization problem is abstracted as a MAB problem. In order to compute a reasonable LR value, we first used the well-known Exponential Recency Weighted Average (ERWA) bandit algorithm as a branching heuristic. Then we improved it with two extensions—Reason Side Rate (RSR) and Locality. Putting these all together, then we obtains the Learning Rate Branching (LRB) Heuristic.

The pseudo code on which our implementation is based is as follows:

Algorithm 1: LRB as a branching heuristic based on ERWA, extended by RSR and Locality for maximizing LR

procedure INITIALIZE ▷ Called once at the start of the solver.
  $\alpha \leftarrow 0.4$
  $LearntCounter \leftarrow 0$
  for $v \in Vars$ **do**
    $Q_v \leftarrow 0$
    $Assigned_v \leftarrow 0$
    $Participated_v \leftarrow 0$
    $Reasoned_v \leftarrow 0$
  end
end procedure
procedure AFTERCONFLICTANALYSIS($learntClauseVars \subseteq Vars, conflictSide \subseteq Vars$)
▷ Called after a learnt clause is generated from conflict analysis.
  $LeartCounter \leftarrow LeartCounter + 1$
  for $v \in conflictSide \cup learntClauseVars$ **do**
    $Participated_v \leftarrow Participated_v + 1$
  end
  if $\alpha > 0.06$ **then** $\alpha \leftarrow \alpha - 10^{-6}$;
  for $v \in (\cup_{u \in learntClauseVars} reason(u)) \backslash learntClauseVars$ **do**
    $reasoned_v \leftarrow reasoned_v + 1$
  end
  for $v \in \{v \in Vars | isUnassigned(v)\}$ **do**
    $Q_v \leftarrow 0.95 \times Q_v$
  end
end procedure
procedure ONASSIGN($v \in Vars$) ▷ Called when v is assigned by branching or propagation.
  $Assigned_v \leftarrow LearntCounter$
  $Participated_v \leftarrow 0$
  $reasoned_v \leftarrow 0$
end procedure
procedure ONUNASSIGN($v \in Vars$)
▷ Called when v is unassigned by backtracking or restart.
  $Interval \leftarrow LearntCounter - Assigned_v$
  if $Interval > 0$ **then**
    $r \leftarrow Participated_v / Interval$
    $rsr \leftarrow Reasoned_v / Interval$
    $Q_v = (1 - \alpha) \cdot Q_v + \alpha \cdot (r + rsr)$
  end
end procedure
procedure PICKBRANCHLIT ▷ Called when the solver requests the next branching variable.
  $Interval \leftarrow \{v \in Vars | isUnassigned(v)\}$
  return $argmax_{v \in U} Q_v$
end procedure

3.1.3 CHB

As for *conflict history-based branching heuristic* (CHB), it is also based on exponential recency weight average, which is a popular algorithm used in non-stationary multi-armed bandit problems to estimate the average reward of different actions. Inspired by the bandit framework and reinforcement learning, the algorithm learns to choose good variables to branch based on past experience. The intuition of this algorithm is similar to the intuition of VSIDS, that is to favor variables that appear recently in conflict analysis. The main difference from LRB is that CHB tend to branch on variables which may cause conflict clause quicker, in which way, the solver will learn new clauses quicker than naive method. The pseudo code can be found from [2]

3.2 Restarting

Restarts are a critically important heuristic in most modern conflict-driven clause-learning SAT solvers. The precise reason as to why and how restarts enable CDCL solvers to scale efficiently remains obscure. Intuitively, restarting is meant to prevent the solver from getting 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. If the search space is large, it may take very long for these incorrect assignments to be fixed. Hence, in practice, SAT solvers usually restart after a certain number of conflicts is detected (indicating that the current search space is difficult), hoping that, with additional information, they can make better early assignments.

We have implemented MLR algorithm to decide when to restart and UCB algorithm to decide which heuristic algorithm to use when restart.

3.2.1 MLR (machine learning-based restart)

In machine learning-based restart, one of the key concept is literal block distance (LRB), which is the number of distinct decision levels of the variables in a clause. Intuitively, a clause with low LBD prunes more search space than a clause with higher LBD, thus is better. LBD is an important quality metric of learnt clause. we will often look at LBDs as a sequence, and we suppose that current newly learnt clause's LBD is determined by some previously learnt clauses' ones. Machine learning is then used to approximately model this relationship. When we run the solver, we will learn the parameters of the model, and at the same time, use what we learnt till now to predict next learnt clause's LBD value. If the predicted value exceeds a predefined threshold, it's time to restart. Readers can find full introduction and pseudo code from [3].

3.2.2 UCB applied after restarting

It is commonly understood that a heuristic may be highly effective for solving a group of problems, but may perform poorly when applied to another group. So we can follow the MAB framework, using UCB algorithm to switch between different heuristics each restart, hoping to find the best one.

In this case, the arms $\{a_1, ..., a_K\}$ represent different heuristics. The reward function, which is the measurement of exploitation, is defined as follows:

$$r_t(a) = \frac{\log_2 decisions_t}{decidedVars_t}$$

decisionst and decidedVarst respectively denote the number of decisions and the number of variables fixed by branching in the run t. The reward function estimates the ability of a heuristic to reach conflicts quickly and efficiently.

$\sqrt{\frac{4.\ln(t)}{n_t(a)}}$ is a measurement of exploration, with $n_t(a)$ representing the number of times the arm a is selected during the $t-1$ previous runs.

The UCB policy select the arm $a = \arg\max_a(\hat{r}_t(a) + \sqrt{\frac{4.\ln(t)}{n_t(a)}})$,where $\hat{r}_t(a)$ is the empirical mean of the rewards of arm a over the $t-1$ previous runs. Within this formula, the left-side term aims to put emphasis on arms that received the highest rewards. Conversely, the right-side terms ensure the exploration of underused arms.

3.3 Preprocessing techniques

Bounded variable elimination (BVE) is a powerful technique for preprocessing, which aims at simplifying a SAT problem by reducing its size. Variable elimination is applied greedily until no more improvement can be made to the clause database by a single elimination. Different notions of "improvement" can be used, including minimizing the number of literal occurrences, or minimizing the number of clauses. In our implement, we chose the former one, and used non increasing variable elimination resolution (NiVER) algorithm to implement it concretely. Pure literal elimination is also a well known simplification method used in most SAT solvers, and we also applied it to our solver.

Table 1: An example for resolution in NiVER.

Clauses with literal l_1	Clauses with literal $-l_1$
$(l_1, l_2, -l_3)$	$(-l_1, l_2, l_3)$
$(l_1, -l_2, l_4)$	$(-l_1, -l_2, -l_4)$
$(l_1, -l_5, -l_6)$	$(-l_1, -l_5, l_6)$

Old_Num_Lit = 18, Number of Clauses deleted = 6

Added resolvents	Discarded resolvents(Tautologies)
$(l_2, -l_3, -l_5, l_6)$	$(l_2, -l_3, l_2, l_3)$
$(-l_2, l_4, -l_5, l_6)$	$(l_2, -l_3, -l_2, -l_4)$
$(l_2, l_3, -l_5, -l_6)$	$(-l_2, l_4, l_2, l_3)$
$(-l_2, -l_4, -l_5, -l_6)$	$(-l_2, l_4, , -l_2, -l_4)$
	$(-l_5, -l_6, -l_5, l_6)$

New_Num_Lit = 16, Number of Clauses added = 4

3.3.1 NiVER (non increasing variable elimination resolution)

For each variable, NiVER checks whether it can be removed by variable elimination resolution, without increasing the total number of occurrences of all literals. If so it eliminates the variable by variable elimination resolution, and then the resolvents that are not tautologies will be added to the formula and all clauses containing that variable will be deleted from the formula. Variables are checked in the sequence of their numbering in the original formula. Some variable removals cause other variables to be removable, and NiVER will iterates until no more variables can be removed. The Alg.2 presents the pseudo code for NiVER preprocessor, and Table 1 is an example for resolvent for Var 1.

Algorithm 2: NiVER CNF Preprocessor

input : Origin CNF sentence F.
output : Sentence after NiVER, and removed clauses containing variables eliminated.

repeat
  $entry = FALSE$
  forall $V \in Var(F)$ **do**
    $P_C = \{C | C \in F, l_V \in C\}$
    $N_C = \{C | C \in F, l_{\neg V} \in C\}$
    $R = \{\}$
    forall $P \in P_C$ **do**
      forall $N \in N_C$ **do**
        $R = R \cup Resolve(P, N)$
      end
    end
    $Old\_Num\_Lits = Numbers\ of\ Literals\ in\ (P_C \cup N_C)$
    $New\_Num\_Lits = Number\ of\ Literals\ in\ R$
    if $Old\_Num_Lits \geq New\_Num\_Lits$ **then**
      $F = F - (P_C \cup N_C)$
      $F = F + R$
      $entry = TRUE$
      $removed\_list.append(var, P_C \cup N_C)$
    end
  end
until $\neg entry$;
return $F, removed\_list$

Then the sentence obtained by preprocessor will be used as the input for the CDCL solver. After the CDCL's assignment, we will then assign for the variables eliminated during preprocessing. And the assignment method is shown by the Alg.3 below.

Algorithm 3: Assignment for variables eliminated during NiVER preprocessor after CDCL

input : Assignment after CDCL, removed_list in NiVER
output : Final assignment.

```
forall (Var, clauses) = removed_list.pop() do
    for c ∈ clauses do
        assigned_vars = Var(clause) − Var
        if assigned_vars are all not in assignment then
            Var = var's value in c
            add Var to assignment
            break
        end
    end
end
return assignment
```

3.3.2 A lighter NiVER

Preprocessing in SAT is a trade-off between the amount of reduction achieved and invested time[7]. Light weight approaches can produce fast preprocessing. So in our implement, besides the above original NiVER algorithm, we also allow a lighter NiVER to be selected for preprocessing.

In lighter-NiVER, the preprocessor will not iterates until no more variables can be removed, but instead it only traverses once. This setup is based on the fact that the number of variables that can be eliminated per round decreases dramatically as the number of iteration rounds increases, and it means later iterations will probably be of little help to the final performance. And we can see this in experiments. The lighter NiVER is the same as the original algorithm presented in Alg.2, and the only slight difference is that the value of $entry$ will not be changed during the loop, which leads to the end of the iteration after one traverse.

3.3.3 Pure literal elimination

A pure literal is a literal whose negation does not occur in the formula. The corresponding technique of pure-literal elimination removes all clauses that contain a pure literal, because these clauses can be trivially satisfied by making the pure literal true without falsifying any other clauses[6]. Pure literal elimination and unit propagation are the two best known simplification methods used in most SAT solvers[5].

In our implement, we first find out all the pure literals, and eliminate them and remove the clauses contained them. Then we will run the NiVER proprocessor. The NiVER algorithm may produce new pure literals, so we will do pure literal elimination after the preprocessing (before CDCL).

4 Results

4.1 GUI and Test script

In order to present our implementation more clearly and to make it easier to select desired parameters, we wrote a simple GUI for visualization. From the GUI you can select the CNF file you want to test, and choose assignment algorithm (VSIDS, ERWA, RSR, LRB, CHB), restart policy, restart bandit, and preprocess policy that you want to use, and set corresponding parameters. The assignment result and the total running time will also be shown by the GUI.

We also wrote a test script to help ourselves conveniently test the effects of different techniques on multiple test files. The test script will help you automatically test the running effect of all techniques' combinations on the files you selected.

4.2 Experimental results

Here are some details for our experiment:

1. Examples: We used the examples given in homework3 and ten samples selected from SAT Competition 2018 Random Track [2] for our experiment.
2. Test: We tested the running time of our CDCL solver equipped with different techniques by running the test script *test.py*. The full test results can be seen in the *results/timeTestResult.csv*, and the data in the table below is selected from the file. TIMEOUT means the running time is over 200 seconds, and was automatically skipped by the script.
3. Baseline: We use the CDCL solver provided by the *homework3_solution* as our baseline.

4.2.1 Comparison of the effect of different branching heuristics

Do not use any restart policy or preprocessing techniques, and just compare the running time of our CDCL solver with different branching heuristics. The corresponding test results are shown in Table.2.

Table 2: Running time (second) of different branching heuristics.

	baseline	our solver				
		VSIDS	ERWA	RSR	LRB	CHB
bmc-1	1095.44	8.42	10.81	4.45	22.57	40.23
bmc-2	1.903	0.094	0.101	0.133	0.062	0.421
bmc-7	24.043	0.274	0.230	0.236	0.233	1.556
test1	$\geq$30min	0.0099	0.0100	0.0049	0.0040	0.0130
test2	4.08	5.28	TIMEOUT	9.75	12.61	1.71
test3	5.76	7.06	TIMEOUT	14.08	24.73	2.97
test4	62.65	5.12	TIMEOUT	6.60	4.09	1.09
test5	18.00	116.07	TIMEOUT	39.16	97.18	14.41
test6	17.33	164.83	TIMEOUT	81.85	189.34	30.69
test7	50.91	128.63	TIMEOUT	105.11	TIMEOUT	38.18
test8	0.248	0.0289	0.0265	0.0319	0.0325	0.0291
test9	1.07	14.02	TIMEOUT	8.76	11.78	7.83
test10	$\geq$30min	12.45	TIMEOUT	11.95	19.33	3.65

From the table we can see that our better branching heuristics have already allowed our solver to perform much better than the baseline in most cases. We can also see that each branching heuristic has different performance on different examples. In these examples, the performance of RSR heuristic is relatively stable and good, while CHB usually performs best in the competition samples.

4.2.2 Effect of restart policy

To demonstrate the effectiveness of our MLR restart policy, we compared the running time of several examples when using MLR policy (without UCB) and when not using any restart policy, under different branching heuristics. The result is shown in Table.3

Meanwhile, to prove that the application of the bandit algorithm UCB is effective, we also compared the performance of the MLR when using UCB algorithm and not using in Table.4.

[2] `http://www.satcompetition.org/`

Table 3: Running time (second) with and without MLR restart under different heuristics.

	VSIDS		RSR		LRB		CHB	
	\	MLR	\	MLR	\	MLR	\	MLR
test3	7.06	1.75	14.08	11.12	24.73	12.55	2.97	0.27
test4	5.12	1.78	6.60	4.27	4.09	35.02	1.09	1.19
test5	116.07	9.88	39.16	1.86	97.18	35.03	14.41	1.07
test6	164.83	101.30	81.85	33.74	189.34	124.01	30.69	33.61
test7	128.63	16.06	105.11	54.4	TIMEOUT	TIMEOUT	38.18	33.05

Table 4: Running time (second) with and without UCB when we use MLR start policy.

	Without UCB					With UCB
	VSIDS	ERWA	RSR	LRB	CHB	
bmc-1	57.50	50.66	46.47	33.15	59.20	36.51
test2	3.46	TIMEOUT	18.14	9.54	6.14	3.69
test3	1.75	TIMEOUT	11.12	12.55	0.27	4.96
test4	1.78	TIMEOUT	4.27	35.02	1.19	1.35
test5	9.88	TIMEOUT	1.86	35.03	1.07	8.96
test6	101.30	TIMEOUT	33.74	124.01	33.61	54.07
test7	16.06	TIMEOUT	54.43	TIMEOUT	33.05	16.52
test9	46.00	31.20	TIMEOUT	TIMEOUT	TIMEOUT	24.36
test10	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	3.15	64.22

From the two tables, we can see that in several cases, MLR restart can accelerate the solver under most branching heuristic. And the UCB can prevents the solver from getting stuck in a bad heuristic that takes a long time.

4.2.3 Effect of preprocessing

We tested the effectiveness of our preprocessor lighter-NiVER with example *bmc-1* and *bmc-7* in Table.5. The value $x+y$ in column "with preprocess" means it needs y seconds to preprocess and then another x seconds to run CDCL. And we can see that preprocessing speeds up the solver significantly on the *bmc-1*. When solving *bmc-7*, after preprocessing the speed of the CDCL can be much faster, but the trade-off is that preprocessing will take some time.

4.3 Summary and analysis of experimental results

By comparing the performance of the solver we implemented with that of MiniSAT, we can find that the language implementation has a great influence on the running speed. The fastest combination of techniques to be used is different for different CNF examples, and no techniques work best in all cases. But compared to the baseline, our CDCL solver wins in most cases by a significant margin.

Table 5: Running time (second) with and without preprocessor lighter-NiVER.

	Other techniques			preprocessor(lighter-NiVER)	
	heuristic	restart policy	bandit	without preprocess	with preprocess
bmc-1	/	MLR	UCB	36.51	15.95+3.92
	VSIDS	MLR	None	57.50	10.62+3.91
	LRB	MLR	None	33.15	18.22+3.99
	CHB	MLR	None	59.20	14.73+4.03
	VSIDS	None	None	8.42	10.04+3.89
	CHB	None	None	40.23	14.78+4.01
bmc-7	/	MLR	UCB	0.275	0.034+1.916
	VSIDS	MLR	None	0.289	0.035+1.968
	CHB	MLR	None	1.521	0.242+2.031

5 Conclusion and future work

In our project, we equipped our CDCL solver with several advanced techniques, which meet the requirements 1 to 4. The experimental results proved that the techniques we implemented are effective, and the performance of our solver is much better than that of the baseline.

For future work, to further accelerate the solver, we can try more bandit algorithms to switch between candidate branching heuristics after restarting, for example EXP3 as mentioned in the document, and try to apply other preprocessing techniques like subsumption to preprocessor. Variables Or literals, which one used as the basic unit of the solver can gain better performance? We can try and compare. We can even add parallel techniques to our solver, and using numba and taichi libraries to optimize performance of Python may speed up the solver a lot, although that's not what AI course cares about. All in all, there is a lot of future work can be done to further improve the performance.

References

[1] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K. (2016). *Learning Rate Based Branching Heuristic for SAT Solvers.* In: Creignou, N., Le Berre, D. (eds) Theory and Applications of Satisfiability Testing – SAT 2016. SAT 2016. Lecture Notes in Computer Science(), vol 9710. Springer, Cham. https://doi.org/10.1007/978-3-319-40970-2_9

[2] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. 2016. *Exponential recency weighted average branching heuristic for SAT solvers.* In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16). AAAI Press, 3434–3440.

[3] Liang, J.H., Oh, C., Mathew, M., Thomas, C., Li, C., Ganesh, V. (2018). *Machine Learning-Based Restart Policy for CDCL SAT Solvers.* In: Beyersdorff, O., Wintersteiger, C. (eds) Theory and Applications of Satisfiability Testing – SAT 2018. SAT 2018. Lecture Notes in Computer Science(), vol 10929. Springer, Cham. https://doi.org/10.1007/978-3-319-94144-8_6

[4] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. *Combining VSIDS and CHB using restarts in SAT.* In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021*, Montpellier, France (Virtual Conference), October 25-29, 2021, volume 210 of LIPIcs, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik, 2021.

[5] Subbarayan, S., Pradhan, D.K. (2005). *NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances.* In: Hoos, H.H., Mitchell, D.G. (eds) Theory and Applications of Satisfiability Testing. SAT 2004. Lecture Notes in Computer Science, vol 3542. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11527695_22

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition,* volume 336 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2021.

[7] Eén, N., Biere, A. (2005). *Effective Preprocessing in SAT Through Variable and Clause Elimination.* In: Bacchus, F., Walsh, T. (eds) *Theory and Applications of Satisfiability Testing. SAT 2005.* Lecture Notes in Computer Science, vol 3569. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11499107_5

A Division of labor within the group

Min Wu(33.3%): optimized code framework, implemented branching heuristics, MLR, wrote GUI.

Yiwen Song(33.3%): implemented UCB, wrote test script, and PPT&presentation.

Chang Su(33.3%): implemented preprocessors, and main part of the report.