Project: Advanced SAT Solving

Haokun Zhu **zhuhaokun@sjtu.edu.cn** **Jiayi Zhang** **jiayizhang@sjtu.edu.cn**

Ning Li **lining01@sjtu.edu.cn**

Abstract

In this project, we improve the SAT solver in homework 3. First, we use better branching heuristics. We implement LRB and CHB to replace the VSIDS algorithm. Besides, we implement restart and apply bandit algorithm UCB to it. We also implement preprocess algorithms such as BVE and subsumption to speed up our SAT solver. And we optimize the data structure in our SAT solver. All of these changes have greatly improved the efficiency for solvering SAT problem.

1 The CHB Branching Heuristic

In this section, we describe our branching heuristic CHB in the context of a CDCL solver based on [3]. CHB maintains a floating point number for each Boolean variable called the Q score, initialized to 0 at the start of the search. Whenever a variable v is branched on, propagated, or asserted, the Q score is updated using Equation 1 where α is the step-size and r_v is the reward value.

$$Q[v] = (1 - \alpha)Q[v] + \alpha r_v \tag{1}$$

r_v is the reward value. A low (resp. high) reward value decreases (resp. increases) the likelihood of picking v to branch on. The reward value is based on how recently variable v appeared in conflict analysis. Let $numConflicts$ be an integer variable that keeps track of how many conflicts have occurred so far and $lastConflict$ be a mapping from each variable to an integer. Initially, $lastConflict[v] = 0$ for each variable v. Whenever a variable x is present in the clauses used by conflict analysis, $lastConflict$ is updated by $lastConflict[x] = numConflicts$. The reward value used by CHB is defined in Equation 2 .

$$r_v \leftarrow \frac{multiplier}{numConflicts - lastConflict[v] + 1} \tag{2}$$

If branching, propagating, or asserting the variable that triggered the update of Q encounters a conflict after propagation, then $multiplier = 1.0$. Otherwise, $multiplier = 0.9$ .The intuition of the reward value is to favor variables that appear recently in conflict analysis.

During branching, CHB selects the greediest play possible by branching on the unassigned variable v with the highest Q score. The algorithm always exploits and this does not appear to be an issue in practice since the problem itself forces exploration in two ways. First, if the algorithm greedily branches on variable v, then it cannot branch on v again until the solver undoes v through backtracking/restarting since v is now assigned and the algorithm is only allowed to branch on unassigned variables. Hence the algorithm is forced to branch on other variables. Second, the propagated variables also have their Q scores updated. Hence variables with low Q scores that will not be picked for branching can still have their Q scores updated.

2 Learning Rate Branching (LRB) Heuristic

In this section, we describe our branching heuristic LRB in the context of a CDCL solver based on [4].

2.1 Branching Heuristic as Learning Rate(LR) Optimization

To frame branching as an optimization problem, we need a metric to quantify the degree of contribution from an assigned variable to the progress of the solver, to serve as an objective to maximize. Since producing learnt clauses is a direct indication of progress, we define our metric to be the variable's propensity to produce learnt clauses. A variable v participates in generating a learnt clause l if either v appears in l or v is resolved during the conflict analysis that produces l. We define I as the interval of time between the assignment of v until v transitions back to being unassigned. Let $P(v, I)$ be the number learnt clauses in which v participates during interval I, and let $L(I)$ be the number of learnt clauses generated in interval I. The learning rate (LR) of variable v at interval I is defined as $\frac{P(v,I)}{L(I)}$.

2.2 Abstracting Online Variable Selection as a Multi-Armed Bandit (MAB) Problem

Given n Boolean variables, we will abstract online variable selection as an n-armed bandit optimization problem. A branching heuristic has n actions to choose from, corresponding to branching on any of the n Boolean variables. The expressions assigning a variable and playing an action will be used interchangeably. When a variable v is assigned, then v can begin to participate in generating learnt clauses. When v becomes unassigned, the LR r is computed and returned as the reward for playing the action v.

2.3 LRB

2.3.1 Exponential Recency Weighted Average (ERWA)

LRB maintains a floating point number for each Boolean variable called the Q score, initialized to 0 at the start of the search. Whenever a variable v is branched on, propagated, or asserted, the Q score is updated using Equation 1 where α is the step-size and r_v is the reward value.

2.3.2 Extension: Reason Side Rate (RSR)

Recall that LR measures the participation rate of variables in generating learnt clauses. If a variable appears on the reason side near the learnt clause, then these variables just missed the mark. We show that accounting for these close proximity variables, in conjunction with the ERWA heuristic, optimizes the LR further.

More precisely, if a variable v appears in a reason clause of a variable in a learnt clause l, but does not occur in l, then we say that v reasons in generating the learnt clause l. We define I as the interval of time between the assignment of v until v transitions back to being unassigned. Let $A(v, I)$ be the number of learnt clauses which v reasons in generating in interval I and let $L(I)$ be the number of learnt clauses generated in interval I. The reason side rate (RSR) of variable v at interval I is defined as $\frac{A(v,I)}{L(I)}$.

This extension modifies the update to $Q_v \leftarrow (1 - \alpha) \cdot Q_v + \alpha \cdot (r + \frac{A(v,I)}{L(I)})$ where $\frac{A(v,I)}{L(I)}$ is the RSR of v . The extension simply encourages the algorithm to select variables with high RSR when deciding to branch. We hypothesize that variables observed to have high RSR are likely to have high LR as well.

2.3.3 Extension: Locality

If the solver is currently working within a community, it is best to continue focusing on the same community rather than exploring another. We hypothesize that high LR variables also exhibit locality. Inspired by the VSIDS decay, this extension multiplies the Q_v of every unassigned variable v by

0.95 after each conflict. Again, we did not change the definition of the reward. The extension simply discourages the algorithm from exploring inactive variables.

Algorithm 1 Pseudocode for ERWA as a branching heuristic using our MAB abstraction for maximizing LR with the RSR extension and the locality extension

procedure INITIALIZE
  $\alpha \leftarrow 0.4$ ▷ The step-size
  $LearntCounter \leftarrow 0$ ▷ The number of learnt clauses generated by the solver
  for $v \in Vars$ **do** ▷ $Vars$ is the set of Boolean variables in the input CNF
    $Q_v \leftarrow 0$ ▷ The EMA estimate of v
    $Assigned_v \leftarrow 0$ ▷ When v was last assigned
    $Participated_v \leftarrow 0$ ▷ The number of learnt clauses v participated in generating
    $Reasoned_v \leftarrow 0$ ▷ The number of learnt clauses v reasoned in generating
  end for
end procedure

procedure AFTERCONFLICTANALYSIS($learntClauseVars \subseteq Vars, conflictSide \subseteq Vars$)
  ▷ Called after a learnt clause is generated from conflict analysis
  $LearntCounter \leftarrow LearntCounter + 1$
  for $v \in conflictSide \cup learntClauseVars$ **do**
    $Participated_v \leftarrow Participated_v + 1$
  end for
  for $v \in (\bigcup_{u \in learntClauseVars} reason(u)) \setminus learntClauseVars$ **do**
    $Reasoned_v \leftarrow Reasoned_v + 1$
  end for
  if $\alpha > 0.06$ **then**
    $\alpha \leftarrow \alpha - 10^{-6}$
  end if
  $U \leftarrow \{v \in Vars \mid isUnassigned(v)\}$
  for $v \in U$ **do**
    $Q_v \leftarrow 0.95 \times Q_v$
  end for
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** ▷ $Interval = 0$ is possible due to restarts
    $r \leftarrow Participated_v / Interval$ ▷ r is the LR
    $rsr \leftarrow Reasoned_v / Interval$ ▷ rsr is the RSR
    $Q_v \leftarrow (1 - \alpha) \cdot Q_v + \alpha \cdot (r + rsr)$ ▷ Update the EMA incrementally
  end if
end procedure

function PICKBRANCHLIT ▷ Called when the solver requests the next branching variable
  $U \leftarrow \{v \in Vars \mid isUnassigned(v)\}$
  return $\arg\max_{v \in U} Q_v$ ▷ Use a priority queue for better performance
end function

3 Restarting

Restarting is a solution for the heavy-tailed distribution of running time often found in combinatorial search. The basic idea of restarting is that we reset the state of the solver after a certain number of conflicts, by this way we can avoid wasting time with some early incorrect assignments which

cannot be detected immediately. Combining with classic bandit algorithms, the solver can also switch between different heuristics for a more diverse exploration of the search space.

3.1 Choosing Heuristics with UCB Algorithms

Referring to [6], we can transform the heuristics choosing problem into a multi-armed bandit(MAB) problem. We consider each time running boolean constraint propagation as pulling the arm of a certain heuristic once, and we can define the reward as $r_t(a) = \frac{\log_2(decisions_t)}{decidedVars_t}$, where t is restarting times, $decisions_t$ is the number of decisions and $decidedVars_t$ is the number of variables fixed by branching in the run t. Each time the boolean constraint propagation function finished, the expected reward will be updated with an incremental method: $\hat{r}_t(a) = \hat{r}_{t-1}(a) + \frac{1}{n_t(a)}[r_t - \hat{r}_{t-1}(a)]$, where $n_t(a)$ is number of times the arm a is selected. Each time the solver meets the conflict times limitation, we will update the upper confidence bound(UCB) of each arm(which represents a heuristic method), and choose the arm with the highest UCB value for following search. Below are two classic method to equation we use to estimate the UCB values(K is the number of arms):

1. **UCB1:** $UCB1(a) = \hat{r_t}(a) + \sqrt{\frac{4.\ln(t)}{n_t(a)}}$
2. **MOSS:** $MOSS(a) = \hat{r_t}(a) + \sqrt{\frac{4}{n_t(a)}\ln(\max(\frac{t}{K.n_t(a)}, 1))}$

3.2 Machine Learning-based Restart Policy

Referring to [5], we build a dynamic restart policy for our CDCl solver based on literal block distance(LBD).

As we have mentioned above, the restart policy can prevent the solver from getting stuck in a time-wasting part caused by some early incorrect assignments. However, restarting too often without a correct policy can lead to bad performance. This is because each time the solver restarts, it erases the search tree which is built during last search and rebuilds a new search tree from scratch, and this procedure may become the bottle neck of performance. In order to design a effective restart policy, we need to figure out why restarts enable CDCL solvers to scale efficiently remains obscure.

According to reference materials we have, restart policy gives CDCL solvers a chance to get out of the local minima. Without restart policy, solvers cannot evaluate the new clause it has learnt and determine whether the new clause is effective for finding a solution of not. This lack of function may lead to a great amount of time wasted since the solver may stuck in a situation where it keeps learning useless clauses. In a word, the key of the restart policy is the evaluation for the newly learned and future learned clauses.

To design a restart policy considering the quality of learned clauses, we apply a machine learning-based method using literal block distance(LBD) to evaluate clauses. Literal block distance(LBD) is defined as the number of distinct decision levels of the variables, so LBD values can indicate a clause's effectiveness for pruning the search space. A clause with low LBD value can prune more search space than a clause with higher LBD. Alg.2 shows how to calculate LBD value.

To determine whether the solver should restart or not, we have to predict the next LBD value we will have. Some experimental evidence supports that recent LBD values are good features to predict the coming value, and we can approximately estimate the next LBD value we will get with function below:

$$f(l_{-1}, l_{-2}, l_{-3}, l_{-1} \times l_{-2}, l_{-1} \times l_{-3}, l_{-2} \times l_{-3}) =$$
$$\theta_0 + \theta_1 \times l_{-1} + \theta_2 \times l_{-2} + \theta_3 \times l_{-3} + \theta_4 \times l_{-1} \times l_{-2} + \theta_5 \times l_{-1} \times l_{-3} + \theta_6 \times l_{-2} \times l_{-3}$$

l_{-i} are the LBD values of the learned clause from i conflicts ago, and θ_i are coefficients to be trained by the machine learning algorithm Alg.3 embedded in the solver.

We need to know the distribution of LBD value so we can figure out the next LBD value is relatively high or low. According to our reference, they figure out that their LBD distribution is close to normal or a right-skewed one, so we can use the normal distribution to estimate the high percentiles. The judgment algorithm is Alg.4.

Algorithm 2 Literal Block Distance

function LBD($learned\_clause$)
  $Lbd \leftarrow 0$
  $level\_list \leftarrow \{\}$
  for all $lit \in learned\_clause$ **do**
    $level \leftarrow get\_level(lit)$
    if $level \notin level\_list$ **then**
      $level\_list$ append $level$
      $Lbd+ = 1$
    end if
  end for
  return Lbd
end function

Algorithm 3 LBD Estimate

function INITIALIZE
  $\alpha \leftarrow 0.001$, $\epsilon \leftarrow 0.00000001$, $\beta_1 \leftarrow 0.9$, $\beta_2 \leftarrow 0.999$
  $conflicts \leftarrow 0$, $conflictsSinceLasrRestart \leftarrow 0$
  $t \leftarrow 0$
  $prevLbd_3 \leftarrow 0$, $prevLbd_2 \leftarrow 0$, $prevLbd_1 \leftarrow 0$
  $\mu \leftarrow 0$, $m2 \leftarrow 0$
  for $v \in \{0 \dots |FeatureVector()| - 1\}$ **do**
    $\theta_i \leftarrow 0$, $m_i \leftarrow 0$, $v_i \leftarrow 0$
  end for
end function
function FEATUREVECTOR
  return $[1, prevLbd_1, prevLbd_2, prevLbd_3, prevLbd_1 \times prevLbd_2, prevLbd_1 \times prevLbd_3, prevLbd_2 \times prevLbd_3]$
end function
function AFTERCONFLICT($learned\_clause$)
  $conflicts \leftarrow conflicts + 1$
  $conflictsSinceLastRestart \leftarrow conflictsSinceLastRestart + 1$
  $nextLbd \leftarrow LBD(learned\_clause)$
  $\delta \leftarrow nextLbd - \mu$, $\mu \leftarrow \mu + \delta / conflicts$, $\Delta \leftarrow nextLbd - \mu$, $m2 \leftarrow m2 + \delta \times \Delta$
  if $conflicts > 3$ **then**
    $t \leftarrow t + 1$
    $feature \leftarrow FeatureVector()$
    $predict \leftarrow \theta \cdot features$
    $error \leftarrow predict - nextLbd$
    $g \leftarrow error \times features$
    $m \leftarrow \beta_1 \times m + (1 - \beta_1) \times g$, $v \leftarrow \beta_2 \times v + (1 - \beta_2) \times g \times g$
    $\hat{m} \leftarrow m / (1 - \beta_1^t)$, $\hat{v} \leftarrow v / (1 - \beta_2^t)$
    $\theta \leftarrow \theta - \alpha \times \hat{m} / (\sqrt{\hat{v}} + \epsilon)$
  end if
  $prevLbd_3 \leftarrow prevLbd_2$, $prevLbd_2 \leftarrow prevLbd_1$, $prevLbd_1 \leftarrow nextLbd$
end function

Algorithm 4 AfterBCP

function AFTERBCP($IsConflict$)
  if $\neg IsConflict \wedge conflicts > 3 \wedge conflictsSinceLastRestart > 0$ **then**
    $\sigma \leftarrow \sqrt{m2 / (conflicts - 1)}$
    if $\theta \cdot FeatureVector() > \mu + 3.08\sigma$ **then**
      $conflictSinceLastRestart \leftarrow 0$
      $Restart()$
    end if
  end if
end function

4 Preprocessing

Preprocessing has become a key component of the Boolean satisfiability(SAT) solving workflow. In practice, preprocessing is situated between the encoding phase and the solving phase, with the aim of decreasing the total solving time by applying efficient simplification techniques on SAT instances to speed up the search subsequently performed by a SAT solver.[7] Here we introduce two key preprocessing techniques and implement them in our SAT solver.

4.1 Bounded Variable Elimination

Given two clauses $C_1 = \{x, a_1, \ldots, a_n\}$ and $C_2 = \{\overline{x}, b_1, \ldots, b_m\}$, the implied clause $C = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ is called the resolvent of the two original clauses by performing resolution on the variable x. We write $C = C_1 \otimes C_2$. This notion can be lifted to sets of clauses. Let S_x be a set of clauses which all contain x, $S_{\overline{x}}$ a set of clauses which all contain $\overline{x}$. Then $S_x \otimes S_{\overline{x}}$ is defined as $S_x \otimes S_{\overline{x}} = \{C_1 \otimes C_2 | C_1 \in S_x, C_2 \in S_{\overline{x}}\}$.

The elimination of a variable x in the whole CNF can be computed by pairwise resolving each clause in S_x with every clause in $S_{\overline{x}}$. The produced resolvents $S = S_x \otimes S_{\overline{x}}$ replace the original clauses S_x and $S_{\overline{x}}$, resulting in a satisfiability equivalent problem. The algorithm is shown in Alg.5. In order to limit the increase of literals, which may result in a worse situation for solve the CNF, we add a bound(where *bounded* in BVE from). If the number of literals in S is larger than that in S_x and $S_{\overline{x}}$, we won't replace S_x and $S_{\overline{x}}$ with S.[1] Besides, in order to speed up the BVE process, we add a hyperparameter MAX. Only variable x with $|S_X| \leq MAX$ and $|S_{\overline{x}}| \leq MAX$ can have a chance to be eliminated.

Algorithm 5 Bounded Variable Elimination

```
function BVE(Sentence, MAX)
    Removed_Vars ← {}
    R ← {}
    for all x ∈ Var(Sentence) do
        if |S_x| ≤ MAX and |S_x̄| ≤ MAX then
            for all C_1 ∈ S_x do
                for all C_2 ∈ S_x̄ do
                    C ← Resolve(C_1, C_2)
                    if C is not a tautology then
                        R ← R ∪ C
                    end if
                end for
            end for
            if NumOfLiterals(S_x + S_x̄) ≥ NumOfLiterals(R) then
                Sentence ← Sentence − S_x − S_x̄ + R
                Removed_Vars ← RemovedVars + (x, S_x)
            end if
        end if
    end for
    return Sentence, Removed_Vars
end function
```

After elimating variables and finally solve the CNF, we need recover the elimated variables and assign values to them. Alg.6 shows the process. We reverse the order of elimated variables and assign value one by one.

4.2 Subsumption

We observed that often similar clauses of a particular kind occur: one clause C_2 almost subsumes a clause C_1, except for one literal x, which, occurs with the opposite sign in C_2. For instance, let $C_1 = \{x, a, b\}$, and $C_2 = \{x, a\}$, then resolving on x will produce $C_1' = \{a, b\}$, which subsumes C_1. Thus after adding C_1' to the CNF, we can remove C_1, in essence eliminating one literal. In this

case, we say that C_1 is strengthened by self-subsumption using C_2. This simplification rule is called self-subsuming resolution.[2] Alg.7 shows the process.

Algorithm 6 Postprocess

```
function POSTPROCESS(Res, Removed_vars)
    for all (x, S_x) ∈ Reversed(Removed_vars) do
        Res ← Res − x − x̄
        for all C ∈ S_x do
            if Unsatisfy(C − x) then
                Res ← Res + x
                break
            end if
        end for
        if x ∉ Res then
            Res ← Res + x̄
        end if
    end for
    return Res
end function
```

Algorithm 7 Subsumption

```
function SUBSUMPTION(Sentence)
    for all C ∈ Sentence do
        for all l ∈ C do
            for all C_l̄ ∈ S_l̄ do
                if (C_l̄ − l̄) ⊆ C and C_l̄ ≠ C then
                    C ← C − l
                end if
            end for
        end for
    end for
    return Sentence
end function
```

5 Improving CDCL Solver

5.1 Improving data structure in LRB

We notice that there are many assignment operations in the LRB algorithm. If we use **ndarray** in numpy, we can do these assignment operations in parallel which will greatly accelerate the CDCL Solver.

Moreover, because the program stores the assignment in a list where each element is a tuple, every time we need to fetch a literal in the assignment, we have to convert the assignment list into a new list. In order to avoid this repeating computation, we store the assigned literal and unassigned literal in two sets.

6 Experiment

6.1 Different Heuristics

In table 1, we can see that in most cases, LRB achieves the best performance and in the rest cases, LRB achieves the second best performance and is only a bit slower than the best one.

Table 1: Different Heuristics

	bmc-1	bmc-2	bmc-3	bmc-4	bmc-5	bmc-6	bmc-7	bmc-8	bmc-9	bmc-10
vsids	146.92	0.54	47.16	657.46	4.99	506.23	4.94	696.68	792.23	3268.77
CHB	338.83	0.3657	1118.94	3942.16	27.90	22659.90	0.84	/	/	/
LRB	36.03	0.15	100.84	762.78	10.82	661.06	0.42	289.03	183.17	1177.97

6.2 Restarting

In table 2, we can see that in some case like `bmc-3`, a solver with restarting may provide much better performance than another solver without this method because the performance improvement with restarting is significant. In other case like `bmc-1`, restarting solver may be defeated by solver without restarting because the performance improvement is not enough to cover the cost for each restarting. In case like `bmc-2`, the difference between solve with and without restarting is so slight that we can ignore it because the problem is too small or the restarting is too infrequent.

Table 2: Restarting

	bmc-1	bmc-2	bmc-3	bmc-4	bmc-5	bmc-6	bmc-7
vsids+restart	87.93	0.60	10.90	781.60	6.83	858.28	7.00
lrb+restart	292.10	0.26	120.87	144.29	16.28	12777.32	0.72
chb+restart	1537.30	0.48	430.59	250.78	20.07	/	1.33
MOSS+restart	220.03	0.59	15.85	338.29	6.87	11294.88	6.66
UCB1+restart	171.37	0.63	13.92	337.45	7.19	7495.98	6.76

6.3 Preprocessing

We test the preprocess algorithm Bounded Variable Elimination and Subsumption based on the heuristic algorithm LRB and VSIDS. The result is shown in Table 3. We can see that preprocess can actually decrease the calculating time in some cases, no matter the heuristic algorithm is. And preprocess performs better on large cases than small cases. This is just what we want, because for small cases, the basic SAT solver is enough.

Table 3: Preprocessing

	bmc-1	bmc-2	bmc-3	bmc-4	bmc-5	bmc-6	bmc-7
LRB+BVE	79.37	0.98	82.21	338.66	15.07	605.52	3.51
LRB+Subsumption	97.51	0.29	95.15	511.53	10.98	833.48	1.02
vsids+BVE	94.53	2.23	100.33	500.42	22.84	1297.03	12.01
vsids+Subsumption	117.83	0.67	53.13	581.28	7.96	346.79	3.87

7 Conclution

In this project, we successfully implement different heuristics, restarting and preprocessing in our SAT solver and improve our data structure to achieve better performance. All these methods are helpful for improving our SAT solver. To be honest, there is still something to improve. We will work on it in the future.

References

[1] Sathiamoorthy Subbarayan and Dhiraj K Pradhan. "NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances". In: *International conference on theory and applications of satisfiability testing*. Springer. 2004, pp. 276–291.

[2] Niklas Eén and Armin Biere. "Effective preprocessing in SAT through variable and clause elimination". In: *International conference on theory and applications of satisfiability testing*. Springer. 2005, pp. 61–75.

[3] Jia Liang et al. "Exponential recency weighted average branching heuristic for SAT solvers". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1. 2016.

[4] Jia Hui Liang et al. "Learning rate based branching heuristic for SAT solvers". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2016, pp. 123–140.

[5] Jia Hui Liang et al. "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*. Vol. 10929. 2018, pp. 94–110.

[6] Cherif Mohamed Sami, Djamal Habet, and Cyril Terrioux. "Combining VSIDS and CHB using restarts in SAT". In: *27th International Conference on Principles and Practice of Constraint Programming*. Vol. 210. 2021, 20:1–20:19.

[7] Seshia3 Cesare Tinelli. "Handbook of Satisfiability". In: (), p. 391.

A Appendix

- Haokun Zhu(33.3%): Implement the CHB and LRB heuristic in the CDCL Solver and make some improvement to the data structure in the CDCL Solver.
- Jiayi Zhang(33.3%): Implement the ML-based restarting policy and the UCB-based heuristics switching policy for the solver.
- Ning Li(33.3%): Finish the preprocessing part. Implement Bounded Variable Elimination and Subsumption for preprocessing.