Introducing Pure Literal Elimination into CDCL Algorithm

Aulong Zha1* Miyuki Koshimura2 Hiroshi Fujita2

1,2 Graduate School and Faculty of Information Science and Electrical Engineering, Kyushu University

Abstract: Unlike unit propagation, pure literal elimination (PLE) has been adopt in few CDCL-type SAT solvers because it may not bring any significant efficiency. In this paper, we reconsider a role of pure literals in solving process. Although we introduce an algorithm that can extract the pure literals within linear time in number of variables, a high execution frequency of it will still lead to an excessive computing cost. For solving this problem, we give a dynamic evaluation and adjustment approach that can optimize the execution frequency of our algorithm.

1 Introduction

A literal $a$ is pure in a CNF formula $F$ if $\neg a$ does not occur in $F$. Pure literals can always be set to true without affecting satisfiability, which amounts to the same as removing classes containing them. Since this can lead to other literals be coming pure, the process needs to be iterated to obtain a satisfiability equivalent formula without any pure literals. This process is known as pure literal elimination (PLE) [4].

The elimination of pure literals is a common heuristic used in many satisfiability algorithms. It was part of the original DLL algorithm (Davis et al., 1962), and it is still employed by those DLL-type backtracking algorithms that achieve the best theoretical worst-case upper bounds for 3-SAT (Kullmann, 1999; Schierneyer, 1996) [2]. The currently most efficient implementations of CDCL-type SAT solvers use a data structure that is optimized for unit propagation, and therefore sacrifice the pure literal heuristic, while there are few modern solvers still use the heuristic only in their preprocessing phase.

In this paper, we present a pure literal extraction algorithm and evaluate it. We also give a dynamic evaluation and adjustment approach to optimize the execution frequency of the algorithm. Finally, we present the experimental evaluation.

2 Pure Literal Elimination

MiniSat [3] is a minimalistic, open-source CDCL-type SAT solver that has achieved numerous excellent performance in past SAT Competition. In the major solving procedure of MiniSat based solvers, variable assignment essentially happens in two different situations: the one is consists of the identification of unit clauses and the creation of the associated implications which carries out Boolean Constraint Propagation (BCP); the other one is decision assignment which picks a unassigned literal by a decision strategy, called Variable State Independent Decaying Sum (VSIDS) heuristic [5]. With defining $p()$ as the priority evaluation of variable assignment, we can obviously know that $p(BCP) > p(VSIDS)$. In this paper, we consider that PLE as the third situation of variable assignment which has a higher priority than decision assignment, but lower than BCP. In general, we have $p(BCP) > p(PLE) > p(VSIDS)$.

2.1 Pure Literal Extraction

With a standard structure of occurrence vector for each literal, which is recorded by the ID of clauses where the literal occurs, we introduce an algorithm that can be easily implemented to extract the pure literals within linear time in number of variables. The pseudo-code is described in Algorithm 1.

In this paper, we present a pure literal extraction algorithm and evaluate it. We also give a dynamic evaluation and adjustment approach to optimize the execution frequency of the algorithm. Finally, we present the experimental evaluation.

2.2 Evaluating Pure Literal Extraction Algorithm

We introduce our algorithm into MiniSat2.2.0. The extended solver carries out PLE by setting a new decision level for each of extracted pure literals, then skip the VSIDS heuristic and run the decision assignment. However, we are well aware of that pure literal extraction will keep a high execution frequency in solving process, which might reduce the efficiency of solving. Therefore, in order to find a balance of the execution frequency of our algorithm, we try to set several constant frequencies for it, and test them. The test questions are all of the 300 instances from the crafted benchmark of SAT Competition 2014. The
The pseudo-code of pure literal extraction algorithm

Algorithm 1. Pure literal extraction algorithm in Solver.cc

```
1: vec<int> pureLiteralList; //declared in Solver.h
2: vec<int> pureLiteralExtraction() {
3:     initialize indexCid2Lit;
4:     eliminatedClauseList = new bool[nClaused];
5:     eliminatedClauseList.clear;
6:     for (int i=0; i<trail.size; i++) {
7:         for (int j=0; j<indexCid2Lit[trail[i].size]; j++) {
8:             eliminatedClauseList[indexCid2Lit[trail[i][j]]] = true;
9:         }
10:     }
11:     for (int i=0; i<nVars; i++) {
12:         bool noPositiveLit = false;
13:         bool noNegativeLit = false;
14:         if (assigns[i]==l_Undef) {
15:             for (int j=0; j<indexCid2Lit[2*i].size; j++) {
16:                 if (!eliminatedClauseList[indexCid2Lit[2*i][j]]) break;
17:             }
18:             for (int j=0; j<indexCid2Lit[2*i+1].size; j++) {
19:                 if (!eliminatedClauseList[indexCid2Lit[2*i+1][j]]) break;
20:             }
21:             if ((noPositiveLit && !noNegativeLit) || (!noPositiveLit && noNegativeLit)) {
22:                 if (noPositiveLit) pureLiteralList.push(2*i+1);
23:                 else pureLiteralList.push(2*i);
24:             }
25:         }
26:     }
27:     return pureLiteralList;
28: }
```

The structure and initialization of occurrence vector for each literal

```
1: indexCid2Lit = new vec<int>[2*nVars];
2: for (int i=0; i<clauses.size; i++) {
3:     for (int j=0; j<ca[clauses[i].size]; j++) {
4:         indexCid2Lit[ca[clauses[i][j]].push(i);
5:     }
6: }
```

Table 1: The result of comparison test

<table>
<thead>
<tr>
<th></th>
<th>org</th>
<th>org+ple</th>
<th>org+ple@frq≤10</th>
<th>org+ple@frq≤100</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>70</td>
<td>43</td>
<td>57</td>
<td>56</td>
</tr>
<tr>
<td>UNSAT</td>
<td>28</td>
<td>3</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>time out</td>
<td>202</td>
<td>254</td>
<td>225</td>
<td>224</td>
</tr>
</tbody>
</table>

MiniSat2.2.0 \(^2\) [3] is not suitable for PLE. Therefore, we decided to use ROKK solver [6] which adds a new learnt clause management strategies to MiniSat2.2.0. ROKK uses Periodic ReduceDB strategy to manage number of learnt clause and compare learnt clauses using Linear Activity and TrueLBD which is a kind of Literal Blocks Distance (LBD) [1]. This management is suitable for PLE.

2.3 Dynamic Evaluation and Adjustment Approach

As has been noticed in test result, a high execution frequency of pure literal extraction algorithm still lead

---

\(^1\)The frequency ∈ \{5, 10, 50, 100\} has been tested respectively, though the best two results are listed in Table 1.

\(^2\)Remove half of the learnt clauses minus some locked clauses. A locked clause is a clause that is reason to current assignation. Clauses below a certain lower bound activity are also be removed.
to an excessive computing cost, especially burdened with the instances that have a huge number of variables and clauses. If we find an approach that can effectively utilize the features of instance and current states to set an optimal frequency, the PLE solvers should get a better performance.

Assume that in unit time (Δtime) the Decision Level increases (ΔDecisionLvl) by 100, without considering the circumstances under backtracking [2], we should make the extraction frequency is less or equal to 100. We let the Difficulty Coefficient be (nVars * nClauses), and let the Computing Coefficient be (Δpropagation / Δtime). Furthermore, we give a penalty in order to dynamically adjust extraction frequency in current states, which will freeze pure literal extraction in a period of time, according to three different situations of ΔprocessEstimate. The dynamic evaluation and adjustment approach sets an optimal frequency as follows: (ΔnPureLits is the increasing number of extracted pure literals within a Δtime)

\[
\text{frequency} = \frac{\text{Computing Coefficient}}{\text{Difficulty Coefficient}}
\]

\[
\times (\Delta \text{DecisionLvl} - \Delta n\text{PureLits})
\]

\[
+ \text{penalty}^{-1}
\]

\[
= \frac{(\Delta \text{propagation} / \Delta \text{time})}{(n\text{Vars} \times n\text{Clauses})}
\]

\[
\times (\Delta \text{DecisionLvl} - \Delta n\text{PureLits})
\]

\[
+ \text{penalty}^{-1}
\]

The pseudo-code of updating penalty

1: initialize double penaltyTimeLimit;
2: double penalty = ΔprocessEstimate;
3: if (penalty < 0)
4: penalty = penaltyTimeLimit;
5: if else (penalty > penaltyTimeLimit)
6: penalty = 0;
7: else
8: penalty = penaltyTimeLimit − penalty;

### 3 Experiment

To demonstrate and assess the effectiveness of our improvement, we introduce PLE into ROKK1.0.1 and initialize penaltyTimeLimit to two different values: 0.1, 0.01 respectively, which become two versions of experimented solver. Including the original ROKK, we use these three solvers to solve all of the 300 instances from the main track benchmarks of SAT Race 2015. The time to solve each instance was limited to 3600 seconds. The experiment environment is described in Table 2.

Experimental results indicate that three solvers show almost the same performance though the version ‘org+ple@p=0.01’ solved one more instance than the other two solvers. The PLE row in Table 3 shows that we extracted pure literals form only about 51% of 300 instances (e.g., in the experiment of ‘org+ple@p=0.1’, there are 152 instances from which the pure literals were extracted, including 62 SAT results, 49 UNSAT results and 41 time out results).

### Conclusion

In this paper, we initially tried to introduce PLE into MiniSat2.0 solver by using an algorithm that can extract the pure literals within linear time in number of variables. However, it suffers from a problem of the unsuitable learnt clause management. Therefore, we decided to use ROKK solver replaced MiniSat. In addition, we presented a dynamic evaluation and adjustment approach instead of setting a constant frequency, which can optimize the execution frequency of our algorithm.

With comparing the experimental results, one version of our extended solvers achieved the best result by a slight advantage. Although this is a small improvement, the good performance of oracle may provide a perspective of parallel algorithms.

---

**Table 2: The experiment environment**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core i7-2860QM @ 2.50GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>7.7 GB</td>
</tr>
<tr>
<td>OS</td>
<td>ubuntu 12.04 LTS 64-bit</td>
</tr>
<tr>
<td>GCC</td>
<td>gcc version 5.0.0</td>
</tr>
</tbody>
</table>

**Table 3: The comparison of experimental results**

<table>
<thead>
<tr>
<th></th>
<th>org</th>
<th>org+ple@p=0.01</th>
<th>org+ple@p=0.1</th>
<th>oracle(^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>136</td>
<td>135 (61)</td>
<td>135 (62)</td>
<td>140</td>
</tr>
<tr>
<td>UNSAT</td>
<td>95</td>
<td>97 (47)</td>
<td>96 (49)</td>
<td>100</td>
</tr>
<tr>
<td>time out</td>
<td>69</td>
<td>68 (46)</td>
<td>69 (41)</td>
<td>60</td>
</tr>
</tbody>
</table>

\(^3\)Oracle provides an upper bound on the performance that could be achieved by org(+ple@p∈\{0.01, 0.1\}): its runtime is that of a hypothetical version that makes every selection in an optimal fashion. Essentially, oracle thus indicates the performance that would be achieved by only running the most suitable version for each single instance.
Figure 1: The comparison of the runtime of each instance between org and org+ple@p=0.01

Figure 2: The comparison of the runtime of each instance between org and org+ple@p=0.1

Figure 3: The comparison of the runtime of each instance between org+ple@p=0.1 and org+ple@p=0.01

Figure 4: The comparison of the runtime cumulative distribution function (CDF)
References


