npSolver – A SAT Based Solver for Optimization Problems

Norbert Manthey and Peter Steinke

Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
peter@janeway.inf.tu-dresden.de

Abstract. The pseudo-Boolean (PB) solver npSolver encodes PB into SAT and solves the optimization instances by calling a SAT solver iteratively. The system supports MaxSAT, PB and WBO. Optimization instances are tackled by a greedy lower bound mechanism first. The solver can translate PB to SAT based on a portfolio of different encodings, based on the number of clauses. As back end of the system any SAT solver can be used, even incremental solvers for the optimization function. By using GLUCOSE as back end and the SAT simplifier Co-processor, npSolver shows an outstanding performance on decision instances and can compete on optimization instances.

1 Introduction

The performance of SAT solvers improved impressively in the last decade. To utilize this power, many different encodings for PB constraints have been developed. In 2006, MINISAT+ [6] combined three of these encodings to a PB solver. Since that time, new encodings have been proposed. In this paper we present npSolver, a pseudo-Boolean (PB) solver that translates PB to SAT similar to MINISAT+, but which incorporates novel techniques. The used SAT solver can be exchanged, enabling the usage of most recent and even parallel systems. To keep learned clauses and avoid re-encodings, we also provide incremental solving of optimization instances. The search for the optimum supports simple bounding, binary search and a novel approximation technique. Besides the encodings for general PB constraints we furthermore provide special encodings for cardinality constraints.

2 Preliminaries

Let $V$ be a finite set of Boolean variables. The set of literals is $V \cup \{ \overline{x} \mid x \in V \}$. A clause is a disjunction of literals and a formula is a conjunction of clauses. An assignment $\sigma$ is a set of literals. In the following, an assignment cannot contain both $x$ and $\overline{x}$ for any variable $x$. An assignment $\sigma$ satisfies a clause $C$ if $C$ contains a literal in $\sigma$, and satisfies a formula if it satisfies all its clauses. The SAT problem asks the question whether a given formula $\phi$ is satisfiable. Details how SAT is solved in state-of-the-art are given in [4].
Now we briefly introduce the pseudo-Boolean (PB) decision problem and its extension to an optimization problem. For more details we point the reader to [9]. In PB constraints, each literal $l_i$ is always assigned a integer weight $w_i$. A linear pseudo-Boolean constraint has the form: $\sum w_i l_i \triangleright k$, where $w_i$ and $k$ are integer constants, $l_i$ are literals and $\triangleright$ is one of the following classical relational operators: $=, >, <, \leq$ or $\geq$. The right side of the constraint is called degree. Each PB constraint can be transformed into equivalent PB constraints with the operator $\leq$ [6,9].

A PB constraint is satisfied with respect to a truth assignment, if the left side evaluates to an integer that satisfies the relational operator. The pseudo-Boolean decision problem asks whether for a set of PB constraints there exists a truth assignment such that all PB constraints are satisfied. The pseudo-Boolean optimization problem is the task for finding the optimal assignment for a set of PB constraints with respect to an optimization function.

Special cases of a linear PB constraint are cardinality constraints. In a cardinality constraint $\sum l_i \leq k$, the weight $w_i$ of each literal $l_i$ is $w_i = 1$. The at-most-one constraint is a specialization for $k = 1$. The at-least-one constraints with the form $\sum l_i \geq 1$ represents SAT clauses.

In the following we have analyzed the set of PB instances from recent PB competitions and have extracted the frequency of occurrence of special cases to stress that translating PB into SAT, and using special encodings, is useful. Table 1 shows the distribution of PB constraints in the instances of the PB benchmark 2010/2011 [1]. The first column gives the kind of the constraint that has been analyzed. The second column gives the absolute number of occurrences of this kind and is followed by a column that gives the relative frequency. The last three columns give the relation between the degree and the number of literals in the constraint. The last row cumulates all other rows. The table clearly motivates that translating PB to SAT is a good idea: 88.4% of the constraints can be encoded as a single SAT clause. Another 1.8% of the constraints encode cardinality constraint for which special SAT encodings are present. When the exactly-k constraint is also considered as cardinality constraint, another 1% is encoded with these encodings. Finally, only 8.5% of all constraints encode a general PB constraint.

3 The Solver Design

Like MiniSat+ we encode each PB constraint individually, by using the best encoding in terms of a expected fast SAT solver run time. In addition, we detect cardinality constraints and use specially designed encodings. See section 4 for more information about the different encodings.

After encoding all PB constraints to SAT, the optimization function (if one exists) is transformed into a PB constraint. The first value of $k$ is received by running the encoded SAT instance without the encoded optimization function.
Table 1. Distribution of special PB constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>absolute</th>
<th>relative</th>
<th>$k &gt; n$</th>
<th>$n^2$</th>
<th>$n^2 &gt; k &gt; n$</th>
<th>$n &gt; k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>at-least-one</td>
<td>226,678,359</td>
<td>88.4 %</td>
<td>0 %</td>
<td>0 %</td>
<td>88.43 %</td>
<td></td>
</tr>
<tr>
<td>at-most-one</td>
<td>3,913,720</td>
<td>1.5 %</td>
<td>0 %</td>
<td>0 %</td>
<td>1.52 %</td>
<td></td>
</tr>
<tr>
<td>at-most-k</td>
<td>997,132</td>
<td>0.3 %</td>
<td>0 %</td>
<td>0 %</td>
<td>0.38 %</td>
<td></td>
</tr>
<tr>
<td>exactly-k</td>
<td>2,729,545</td>
<td>1.0 %</td>
<td>0 %</td>
<td>0 %</td>
<td>1.06 %</td>
<td></td>
</tr>
<tr>
<td>other</td>
<td>22,014,154</td>
<td>8.5 %</td>
<td>0.04 %</td>
<td>0.02 %</td>
<td>8.51 %</td>
<td></td>
</tr>
<tr>
<td>equality</td>
<td>793</td>
<td>0.0 %</td>
<td>0 %</td>
<td>0 %</td>
<td>0.00 %</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>256,333,703</td>
<td>100 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

and applying the assignment of the variables to the optimization function. To find the optimum we incrementally reduce $k$ until we find the optimal value of the function. These steps are repeated until the formula becomes unsatisfiable. The last upper bound is returned as the optimum of the PB instance. In addition to MiniSat+ our solver is also able to perform binary search and use an incremental SAT solver, to avoid the loss of learned clauses. However, the incremental solver has to be restarted whenever an unsatisfiable formula has been encoded such that for binary search using the incremental solver does not improve the system significantly. The components of our solver are visualized in Figure 1. Since most of the time the optimization constraint is larger than the other PB constraints in a PB instance, we try to approximate the objective function. Therefore we divide each weight $w_i$ and $k$ of the constraint by an heuristically chosen factor $c$, resulting in a PB constraint that can be encoded with less clauses. To avoid invalid solutions we overestimate the approximation by rounding up new weights: $w'_i := \lceil \frac{w_i}{c} \rceil$ and rounding down the new degree: $k' := \lfloor \frac{k}{c} \rfloor$. We call this technique quickbound. As long as the encoded formula is satisfiable, we use this approximation of the objective function to search for the optimum. If the SAT formula is unsatisfiable, we leave quickbound and perform the usual algorithm, i.e. without the approximation. Note that we have to encode the last bound twice – one time with quickbound and one time without. Still, the benefit is to encode the optimization function with less clauses to reduce the fraction of clauses for that function for example in MaxSAT instances.

4 Encoding PB into SAT

The algorithm decides for each PB constraint which encoding to use. Here we try to use only encodings that maintain general arc consistency (GAC) by unit propagation, since this is important for the performance of a SAT solver. We choose among the following known encodings for PB: sorting networks \cite{[2]} for cardinality constraints, a nested encoding for at-most-one constraints and BDDs, adder networks \cite{[6]} for general PB constraints. We treat the at-most-one $x_1 + \cdots + x_n \leq 1$ constraints specially by introducing a fresh variable $y$ if $n > 4$ and encode:

$$ (y + \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} x_i \leq 1) \land (\overline{y} + \sum_{i=\lfloor \frac{n}{2} \rfloor+1}^{n} x_i \leq 1). $$

\footnote{The tool is available at tools.computational-logic.org}
Fig. 1. Solver Design: The PB instance $PB$ is encoded to the SAT formula $F$. The preprocessor Copyprocessor2 [8] is used to simplify $F$ to $F'$. Optimization problems are solved iteratively by encoding the objective function with different bounds $i$ until the optimum is found. The resulting formulas $F'_i$ are solved by a SAT solver, calculating the models $J'_i$.

There are other encodings for the at-most-one constraint, e.g. [5], but for small $n$ the above method produces less clauses. Finally, we encode BDDs without any auxiliary variables (by adding a clause for each path that leads to a false node), if this results in fewer clauses.

Among all these encodings (except for the adder network) we use the best encoding for each individual PB constraint, in terms of less clauses. Only if the number of clauses becomes higher than 1 000 000 we switch to adder networks, that need the fewest number of clauses but do not maintain GAC.

Equality constraints are translated into two $\leq$-constraints, resulting in twice the number of clauses and auxiliary variables, except for the sorting networks, where just the number of clauses are duplicated while using the same amount of auxiliary variables.

Table 2 shows the encodings our solver used during the benchmark presented in section 6. Since the global watchdog encoding presented in [11] is rarely useful, we decided to not implement this encoding and only approximate the number of clauses for the table (with $n^3 \log(k) \log(w_{max})$, where $w_{max}$ is the biggest weight $w_i$ in a constraint). This encoding might be useful for constraints with $k \gg n$, but our benchmark lacks of such instances.

Table 2. Best encodings in terms of less clauses, where AMO is the presented encoding for at-most-one encoding and 2-product is the encoding of [5], resulting in fewer clauses if $n > 47$.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>AMO</th>
<th>2-product</th>
<th>sorting networks</th>
<th>BDDs</th>
<th>WatchDog</th>
<th>adder networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>611859</td>
<td>19227</td>
<td>112253</td>
<td>22967061</td>
<td>517</td>
<td>567</td>
</tr>
<tr>
<td>relative</td>
<td>2.58 %</td>
<td>0.08 %</td>
<td>0.47 %</td>
<td>96.86 %</td>
<td>0.00 %</td>
<td>0.00 %</td>
</tr>
</tbody>
</table>
4.1 Encoding MaxSAT into SAT

To solve MaxSAT, for each clause $C_i$ with the weight $w_i$ NP Solver introduces a relaxation variable $r_i$ that is added to $C_i$ (compare to [3]). The optimization that is performed afterwards it to minimize the sum $\sum_i w_i r_i$.

4.2 Encoding Weighted PBO into SAT

Weighted boolean optimization (WBO) could be transformed as MaxSAT: For each constraint $C_i$ with a weight $w_i$ add a relaxation variable $r_i$ to the constraint and minimize the sum $\sum_i w_i r_i$. For a PB constraint with $n$ variables and degree $k$, the an encoding would require the clauses for the original constraint, and additional clauses if the relaxation variable is added. To avoid these additional clauses, we decided to add the relaxation variable not to the constraint $C_i$ itself, but to all the clauses that are generated to encode this constraint. The optimization function is created as in MaxSAT: $\sum_i w_i r_i$.

5 Experimental Results

We compare the available methods of our solver with the native domain solver bsolo [7], because of its high performance in recent competitions. MINISAT+ has not been chosen as reference, because the internally used SAT solver is too old for a fair comparison, hence we have to use the SAT formula output MINISAT+ with a current state-of-the-art SAT solver, but then the solver can only solve decision problems, and we found an unsatisfiable PB instance where the SAT translation resulted in a satisfiable SAT formula. Table 3 shows the number of solved instances and the corresponding mean run time per solver on all 492 instances of the last two PB competitions. We divided the instances into two sets: decision instances (PB) and optimization instances (PBO). For decision instances the translation to SAT clearly has a higher performance than bsolo. Almost 40 instances more can be solved, and the mean run time for these instances is not significantly larger although more instances can be solved.

For the optimization instances the picture is different. BSOLO can solve more instances than any of the other approaches. However, notice that BSOLO has a higher performance on a particular instance family, namely on random instances (rand.biglist and random/rand.newlist). NP Solver can solve only 12 of these instances where BSOLO solves 29. In total, BSOLO solves 217 instances and NP SOLVER can solve 245 instances. Surprisingly, for the optimization instances the two search algorithms binary search and top down search do not differ significantly. At the current point we cannot explain this behavior.

With the default settings, the current state of the solver already shows a high performance. Adding incremental search, the quick bound method or a

---

Table 3. Solving PB instances from the PB competition

<table>
<thead>
<tr>
<th>Instance</th>
<th>bsolo</th>
<th>npSolver+bin</th>
<th>npSolver+topdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB</td>
<td>98</td>
<td>136 s</td>
<td>141</td>
</tr>
<tr>
<td>PBO</td>
<td>119</td>
<td>206 s</td>
<td>127</td>
</tr>
</tbody>
</table>

---

3 Such an example is the instance PB10/normalized-PB10/DEC-SMALLINT-LIN/oliveras/j60/normalized-j6013_2-unsat.opb
SAT preprocessor to the system does not improve the results. Still, we believe that we can tweak the parameters of npSolver and improve the incremental search, resulting in an even more powerful tool. Furthermore, with new SAT solvers available, the performance of the solver will increase automatically.

6 Conclusion

We presented the PB solver npSolver that is based on a translation to SAT. Based on a parameterized implementation this tool can use various encodings to achieve a good SAT formula. With this formula, a SAT solver is called to solve the PB instance. For optimization instances, the solver is called multiple times. Besides PB, the tool can also transform MaxSAT and WBO instances into SAT instances and solve these optimization problems. We showed in a brief evaluation that the current early stage of work results already in a powerful tool. By tuning the parameters of the system and choosing an appropriate SAT solver, npSolver can be adapted easily to applications.

Future development of npSolver includes optimizing and parallelizing the system. A first step would be to evaluate parallel SAT solving approaches. Next, more encodings can be integrated into the system. Afterwards, the limits for each encoding will be determined by intensive testing. Finally, we might also adopt the translation into SAT to other optimization problems, resulting in a wider range of problems that could be solved by SAT solvers.

References

8. N. Manthey. Coprocessor 2.0 – A flexible CNF Simplifier (Tool Presentation), 2012. Accepted for SAT 2012.