march_ks

Marijn Heule and Hans van Maaren
{m.j.h.heule, h.vanmaaren}@tudelft.nl

1 introduction

The march_ks SAT solver is an upgraded version of the successful march_dl and march_eq SAT solvers, which won several awards at the SAT 2004 and SAT 2005 competitions. For the latest detailed description, we refer to [2]. Like its predecessors, march_ks integrates equivalence reasoning into a DPLL architecture and uses look-ahead heuristics to determine the branch variable in all nodes of the DPLL search-tree. The main improvements in march_ks are:

- renewed pre-processing techniques: Removal of the 3-SAT translator and therefore a new procedure for the addition of resolvents.
- an improved adaptive algorithm to trigger the DOUBLELOOK procedure - inspired by the one used int satz by Li [1].
- a guided jumping strategy: Instead of the conventional depth-first search, march_ks uses a jumping strategy based on the distribution of solutions measured on random 3-SAT instances [3].

2 pre-processing

The pre-processor of march_dl, reduces the formula at hand prior to calling the main solving (DPLL) procedure. Earlier versions already contained unit-clause and binary equivalence propagation, as well as equivalence reasoning, a 3-SAT translator, and finally a full - using all free variables - iterative root look-ahead.

However, march_ks is the first version of march which does *not* use a 3-SAT translator by default (although it is still optional). The motivation for its removal is to examine the effect of (not) using a 3-SAT translator on the performance. Because the addition of resolvents was only based on the ternary clauses in the formula (after the translation) we developed a new algorithm for this addition which uses all clauses with at least three literals.

3 the architecture

As a look-ahead SAT solver, the branch rule of march_ks is based on a look-ahead evaluation function (DIFF). The applied DIFF measures the reduction of CNF- and equivalence-clauses between two formulas $\mathcal{F}$ and $\mathcal{F}'$ in a weighted manner. The solver differs from the straightforward look-ahead architecture in two aspects: (1) look-ahead is performed on a subset of the free (unfixed) variables, and (2) if a certain look-ahead significantly reduces the formula, the DOUBLELOOKHEAD procedure is called to check whether this look-ahead will eventually result in a conflict. Algorithms below show the pseudocode of this architecture.

Algorithm 1 PARTIALLOOKAHEAD( )

for each variable x_i in $\mathcal{P}$ **do**
  $\mathcal{F}'$:= ITERATIVEUNITPROPAGATION($\mathcal{F} \cup \{x_i\}$)
  $\mathcal{F}''$:= ITERATIVEUNITPROPAGATION($\mathcal{F} \cup \{\neg x_i\}$)
  if $\mathcal{F}' \ll \mathcal{F}$ **and** $\emptyset \notin \mathcal{F}'$ **then**
    $\mathcal{F}'$:= DOUBLELOOKAHEAD($\mathcal{F}'$)
  else if $\mathcal{F}'' \ll \mathcal{F}$ **and** $\emptyset \notin \mathcal{F}''$ **then**
    $\mathcal{F}''$:= DOUBLELOOKAHEAD($\mathcal{F}''$)
  end if
  if $\emptyset \in \mathcal{F}'$ **and** $\emptyset \in \mathcal{F}''$ **then**
    return "unsatisfiable"
  else if $\emptyset \in \mathcal{F}'$ **then**
    $\mathcal{F} := \mathcal{F}''$
  else if $\emptyset \in \mathcal{F}''$ **then**
    $\mathcal{F} := \mathcal{F}'$
  else
    $\mathrm{H}(x_i) := 1024 \times \mathrm{DIFF}(\mathcal{F}, \mathcal{F}') \times \mathrm{DIFF}(\mathcal{F}, \mathcal{F}'') + \mathrm{DIFF}(\mathcal{F}, \mathcal{F}') + \mathrm{DIFF}(\mathcal{F}, \mathcal{F}'')$
  end if
end for
return x_i with highest $\mathrm{H}(x_i)$ to branch on

Algorithm 2 DoubleLookahead($\mathcal{F}$)

for each literal l_i in $\mathcal{P}$ **do**
  $\mathcal{F}'$:= IterativeUnitPropagation($\mathcal{F} \cup \{l_i\}$)
  if $\emptyset \in \mathcal{F}'$ **then**
    $\mathcal{F}$:= IterativeUnitPropagation($\mathcal{F} \cup \{\neg l_i\}$)
    if $\emptyset \in \mathcal{F}$ **then**
      break
    end if
  end if
end for
return $\mathcal{F}$

Algorithm 3 IterativeUnitPropagation($\mathcal{F}$)

while unit clause $y \in \mathcal{F}$ **and** $\emptyset \notin \mathcal{F}$ **do**
  satisfy y and simplify $\mathcal{F}$
end while
return $\mathcal{F}$

The partial behavior of the look-ahead procedure is implemented by performing look-ahead only on variables in set $\mathcal{P}$: At the beginning of each node, this set is filled by pre-selection heuristics based on an approximation function of Diff. In contrast to earlier versions of march, the size of $\mathcal{P}$ is not fixed to a percentage of the original number of variables. Currently, its size equals a constant times the average number of detected failed literals.

The DoubleLook procedure is called when $\mathcal{F}' \ll \mathcal{F}$ (see algorithm 1). We denote by $\mathcal{F}' \ll \mathcal{F}$ that many clauses in $\mathcal{F}'$ are reduced clauses of $\mathcal{F}$. More specific, clauses in $\mathcal{F}$ that are satisfied in $\mathcal{F}'$ are not counted. If $\mathcal{F}' \ll \mathcal{F}$, then there is a relatively high probability that during DoubleLook($\mathcal{F}'$) $\mathcal{F}'$ will contain the empty clause ($\emptyset$), forcing $\mathcal{F}$ to be satisfiability equivalent to $\mathcal{F}''$.

However, how can we implement $\mathcal{F}' \ll \mathcal{F}$ in such a way that it results in optimal performance? In [4], we propose an adaptive algorithm to determine when to call the DoubleLook procedure. This algorithm does not only improve the performance of march, but also of satz and kcnfs.

In [3], we observed that the solutions of hard random 3-Sat formulas are not distributed uniformly, but in a biased manner. Based on this distribution, we developed a jumping strategy which visits the subtrees of the DPLL search tree in descending order of the observed likelihood of containing a solution. This jumping strategy is also a new feature of march_ks.

4 additional features

- cache optimizations: Two alternative data-structures are used to store the binary and n-ary clauses. Both are designed to decrease the number of cache misses in the PartialLookahead procedure.

- tree-based look-ahead: Before the actual look-ahead operations are performed, various implication trees are built from the binary clauses of which both literals occur in $\mathcal{P}$. These implications trees are used to decrease the number of unit propagations.

- necessary assignments: If both $x_i \rightarrow x_j$ and $\neg x_i \rightarrow x_j$ are detected during the look-ahead on x_i and $\neg x_i$, x_j is assigned to true because it is a necessary assignment.

- resolvents: Several binary resolvents are added during the solving phase. All these resolvents have the property that they are easily detected during the look-ahead phase and that they could increase the number of detected failed literals.

- restructuring: Before calling procedure PartialLookahead, all satisfied n-ary clauses of the prior node are removed from the active data-structure to speed-up the look-ahead.

References

[1] Chu Min Li. *A constrained-based approach to narrow search trees for satisfiability.* Information processing letters **71** (1999), 75–80.

[2] Marijn J.H. Heule and Hans van Maaren. *March_dl: Adding Adaptive Heuristics and a New Branching Strategy.* Journal on Satisfiability, Boolean Modeling and Computation **2** (2006), pp. 47-59.

[3] Marijn J.H. Heule and Hans van Maaren. *Whose side are you on? Finding solutions in a biased search-tree.* Proceedings of Guangzhou Symposium on Satisfiability In Logic-Based Modeling (2006), pp. 82-89.

[4] Marijn J.H. Heule and Hans van Maaren. *Effective Incorporation of Double Look-Ahead Procedures.* Submitted to Sat 2007.