HaifaSat: a new robust SAT solver

Roman Gershman

Computer Science, Technion, Haifa, Israel.
Email: gershman@cs.technion.ac.il

Abstract. HaifaSat is a new SAT solver that is based on a new decision heuristic called Clause-Move-To-Front (CMTF), a new resolution-based scoring strategy and many other improvements. Experiments on hundreds of industrial benchmarks show that HaifaSat is faster and more robust than other well known solvers like zChaff 2004 and Berkmin.

1 Introduction

HaifaSat is a state-of-art SAT solver that extends the DPLL framework with a few algorithmic improvements.

2 Some of HaifaSat improvements

In modern SAT solving on industrial problems the resolution algorithm (which we will call here fuip-Alg) is a key component that allows to create new conflict clauses and thus refute unsatisfiable space efficiently (without enumerating all assignments). Recall that the core of fuip-Alg is a loop that marks all variables from the resolved clauses in order to resolve them further. Each variable can be marked at most once, therefore each resolved clause contributes only those variables that were not marked before during the resolution. For further information we refer the reader to [4, 2].

Here we provide several extensions to the original algorithm which tend to create better conflict clauses.

1. **Lazy Antecedent** zChaff 2004 chooses the shortest antecedent between possible antecedent clauses immediately during the BCP process. HaifaSat, on the other hand, stores up to K candidates for each propagated literal during BCP without determining the antecedent clause. Thus, instead of working on a single implication graph, fuip-Alg in HaifaSat works on a set of such graphs, which are represented by the cross-product of the antecedent sets of size at most K. The antecedent clause of each variable is determined when this variable is resolved. Since the resolved clause contributes only unmarked variables, it is possible to choose the antecedent that contributes the smallest number of unmarked variables. This technique has the same goal as the original method, but attempts to do it better.

2. **Clause Minimization** was proposed originally in [1] (originally as a tool to ease the analysis of resolution rather than as an optimization). Suppose that a new conflict clause C is derived. For any literal $v \in C$ if $Antecedent(v) \setminus \{\overline{v}\} \subseteq C$ then $C \setminus \{v\}$ is also a valid conflict clause. In order to implement clause minimization efficiently we incorporated it only for the binary antecedents, i.e. if $v \in C$ has antecedent clause $(u, \overline{v})$ s.t. $u \in C$ then C is minimized to $C \setminus \{v\}$. Efficient bookkeeping enables us to find all such minimizations in a single pass on C.

2.1 Decision Heuristic and Abstraction-Refinement

A good decision heuristic, as observed by [4], indirectly guides the SAT solver to create effective conflict clauses through resolution. Berkmin [3] stores all conflict clauses in a stack and chooses variables from the last unsatisfied conflict clause in this stack. Siege [4] uses the Variable-Move-To-Front (VMTF) heuristic that stores all the variables in a list, and after each conflict it moves some of the variables in the conflict clause to the front. The decision is made on the first undefined variable from the list.

A heuristic like the one implemented in Berkmin causes the SAT solver to follow an *abstraction-refinement* process internally. Indeed, each conflict clause is an abstraction of the clauses that participated in its derivation, and therefore trying to satisfy it first corresponds to trying to satisfy the abstract model. Refinement is done when the solver succeeds to satisfy the clause C, but encounters a conflict later on and creates a new conflict clause that necessarily prunes space that is not covered by C (since the latter is already satisfied). However, chronological order does not necessary reflect the actual connections between the clauses. In fact, the Resolve Graph (a directed graph whose vertices are clauses, and (a, b) is an edge iff b participated in the resolution of the conflict clause a) represents such connections. But instead of storing the entire resolve graph, the following Clause-Move-To-Front (CMTF) heuristic is implemented.

All the conflict clauses are stored in the list and every new conflict clause is added to the front of the list. During the resolution, a bounded number of clauses that participated in resolving the conflict clause are also moved to the front. Thus if some clause participates in the resolution, it can be positioned after its resolvent (until the next time it participates in a resolution, a case in which it can be moved to a new location). When the solver looks for the next unsatisfied clause, it starts from the top of the list and continues towards its bottom. If all the conflict clauses are satisfied then the original VMTF strategy (from Siege) is applied. Our experiments on industrial benchmarks show that the decision heuristic in Berkmin is more effective than VMTF and CMTF appears to be more effective than Berkmin.

2.2 Resolution-Based Scoring

The following definitions refer to the resolution algorithm, and will later be used to explain Resolution-Based-Scoring (RBS). We call a variable *Flipped-Var* if

it is implied by a newly created conflict clause. When such a variable becomes undefined again due to backtracking, it stops being Flipped-Var. We denote by *R-set*(x) the set of variables from the *current* (conflict) decision level that were resolved *before* x. For each variable we define *Flipped-Weight* $w(x)$ which will be defined later. We also define *R-sum*(X) for a set X as *R-sum*$(X) := \sum_{x \in X} w(x)$. $w(x)$ is defined recursively as 0 if x is not Flipped-Var and *R-sum*$(Rset(x)) + 1$ otherwise. In the latter case, $w(x)$ is set when x becomes Flipped-Var and set back to 0 when x becomes once more undefined. The intuition is the following: among variables in the resolve set there are Flipped-vars which were implied by the previous conflicts. These variables connect the current conflict with previous ones (can be associated with edges in the resolve-graph). In practice, every such Flipped-var can represent contribution of many conflicts (all descendants of the conflict clause that implied this variable). Flipped-weight represent scores of such contributions recursively, i.e. it *roughly* measures the importance of a particular resolution according to how many conflicts were needed in order to imply the resolved variable using *Regular Resolution*.

Flipped-Weights are used to update activity and sign-score of the variable. Activity is used to choose the most active variable in the unsatisfied clause from CMTF. Sign-score is used to determine the value of this variable: if sign-score is positive then the value is set to TRUE. Every time a variable x is resolved during FUIP-ALG, we increase its activity by *R-sum*(*R-set*$(x)) + 1$ and change its sign-score by the same value to the direction opposed to the one which is associated with the current value of x, i.e. down for TRUE and up for FALSE. Every constant number of conflicts both activity and sign-score are divided by 2 as in VSIDS. HAIFASAT *never* resets its scores to the literal count scores after the first initialization. This score system already works at least as good as the Zchaff 2004 scoring system on many industrial problems. But we can improve it further by noticing that it is updated only due to conflicts, i.e. there is no 'positive' learning from 'quite' decisions that do not participate in the conflicts. HAIFASAT, therefore, adds a small constant factor to a variable's sign-score each time it is implied, i.e., if this variable is implied to TRUE then sign-score is increased, and it is decreased otherwise.

Consider a satisfiable set of clauses with a satisfying assignment $a(x) \in \{0, 1\}$ for each variable x in these clauses. Suppose we start to decide on all variables according to this assignment. Some of the decisions cause additional implications which must be consistent with the given assignment a. A perfect decision heuristic could extend this assignment without contradictions and positively correlate with implications in the following manner: if x is implied to FALSE then sign-score of x would be negative. In fact, this perfect heuristic could choose variables in *any order* and its sign-score value would be consistent with each implication. We try to converge to such a heuristic by using the simple method described above.

References

1. Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *IJCAI*, pages 1194–1201, 2003.
2. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.
3. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.
4. L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.