Clone: Solving Weighted Max-SAT in a Reduced Search Space

Knot Pipatsrisawat and Adnan Darwiche
{thammakn,darwiche}@cs.ucla.edu

Computer Science Department
University of California, Los Angeles

Abstract. We introduce a new branch-and-bound Max-SAT solver, Clone, which employs a novel approach for computing lower bounds. This approach allows Clone to search in a reduced space. Moreover, Clone is equipped with novel techniques for learning from soft conflicts. Experimental results show that Clone performs competitively with the leading Max-SAT solver in the broadest category of this year's Max-SAT evaluation and outperforms last year's leading solvers.

1 Introduction and Background

The maximum satisfiability problem (Max-SAT) is one of the optimization counterparts of the Boolean satisfiability problem (SAT). In Max-SAT, given a Boolean formula in conjunctive normal form (CNF), we want to determine the maximum number of clauses that can be satisfied by any complete assignment.[1]

Two important variations of the Max-SAT problem are the weighted Max-SAT and the partial Max-SAT problems. The *weighted* Max-SAT problem is the Max-SAT problem, in which each clause is assigned a positive weight. The objective of this problem is to maximize the sum of weights of satisfied clauses by any assignment. The *partial* Max-SAT problem is the Max-SAT problem, in which some clauses cannot be left falsified by any solution.[2] The combination to both variations is called the *weighted partial* Max-SAT problem. For the rest of this paper, we use the term Max-SAT to refer to any variation of the Max-SAT problem, while noting that our solver is meant for the broadest category: *weighted partial* Max-SAT.

There are two main approaches used by contemporary exact Max-SAT solvers: the satisfiability-based approach and the branch-and-bound approach. The former converts each Max-SAT problem with different hypothesized maximum weights into multiple SAT problems and uses a SAT solver to solve these SAT problems to determine the actual solution. Examples of this type of solver are ChaffBS, ChaffLS [1] and SAT4J-MaxSAT [2]. The second approach, which

[1] A clause is a disjunction of literals and a literal is simply a variable or its negation.
[2] In practice, a madatory clause is represented by a clause with a sufficiently large weight.

seems to dominate in terms of performance based on recent Max-SAT evaluations [3, 4], utilizes a depth-first branch-and-bound search in the space of possible assignments. An evaluation function which computes a bound is applied at each search node to determine any pruning opportunity.

The methods used to compute bounds vary among branch-and-bound solvers and often give rise to difference in performance. Toolbar utilizes local consistencies to aid bound computations [5, 6]. Lazy, MaxSatz, and MiniMaxSat compute bounds using some variations of unit propagation and disjoint component detection [7–10]. In this paper, we introduce a method of bound computation based on formula compilation. Our approach can be thought of as a new paradigm that combines search and compilation together. A recently developed solver, Sr(w), by Ramírez and Geffner [11] employs a very similar approach for computing bounds. Clone and Sr(w) were developed independently and both participated in this year's Max-SAT evaluation [4]. The performance of both solvers in the evaluation is available at http://www.maxsat07.udl.es/ms07.pdf, showing that Clone dominated Sr(w) in most of the evaluation categories. We note, however, that the version of Clone described here is an improved version whose performance is significantly better than the one that participated in the Max-SAT evaluation 2007.

For the remaining of this paper, we describe the components of our Max-SAT solver. Clone consists of two main components. The first is the preprocessor which takes a Max-SAT problem as an input and produces a data structure necessary for computing bounds. The second is the branch-and-bound search engine that takes advantage of the data structure and many inference techniques to aid the search. None of the techniques used by Clone are specific to any particular type of problem, making Clone applicable to any variation of Max-SAT.

In the next section, we discuss the preprocessor and our approach for computing bounds. In Section 3, we describe the search component and the inference techniques used. Experimental results are presented in Section 4 and we close with some remarks in Section 5.

2 Bound Computation

In the literature, the Max-SAT problem is often viewed as the problem of minimizing the costs (weights) of falsified clauses of any assignment. We will follow this interpretation and use the term *cost* to refer to the sum of weights of clauses that cannot be satisfied. Moreover, we will use UB (upper bound) to denote the best cost of any complete assignment found so far and LB (lower bound) to denote the guaranteed cost of the current partial assignment. Branch-and-bound search algorithm can prune all children of a node whenever $LB \geq UB$.

To compute lower bounds, we take advantage of a tractable language called deterministic decomposable negation normal form (d-DNNF) [12, 13]. Many useful queries can be answered about sentences in d-DNNF in time linear in the size of these sentences. One of these queries is (weighted) minimum cardinality,

which is similar to Max-SAT, except that weights are associated with variables instead of clauses. Our approach is indeed based on reducing Max-SAT on the given CNF to minimum cardinality on a d-DNNF compilation of the CNF. If this compilation is successful, the Max-SAT problem is solved immediately since minimum cardinality can be solved in time linear in the d-DNNF size. Unfortunately, however, the compilation process is often difficult. Our solution to this problem is then to compile a relaxation of the original CNF, which is generated carefully to make the compilation process feasible. The price we pay for this relaxation is that solving minimum cardinality on the relaxed d-DNNF compilation will give lower bounds instead of exact solutions. Our approach is then to use these lower bounds for pruning in our brand-and-bound search.

We show how a Max-SAT problem can be reduced to a minimum cardinality problem in Section 2.1. We will then discuss problem relaxation in Section 2.2, followed by compilation in Section 2.3.

2.1 Reducing Max-SAT to minimum cardinality

Given a CNF formula and a cost for each literal of the formula, the weighted minimum cardinality problem asks for a satisfying assignment that costs the least. The cost of an assignment is the sum of costs of all literals that it sets to true. To reduce a Max-SAT problem into a minimum cardinality problem, we introduce a distinct selector variable to each clause of the Max-SAT problem and assign the clause's cost to the positive literal of the selector variable [14]. All other literals have zero cost. For example, the clause $C = (a \vee b \vee c)$ becomes $C' = (s \vee a \vee b \vee c)$ after the selector variable s is added. If C originally had cost w, then w is assigned to s and any assignment that set $s = \mathbf{true}$ will incur this cost. After this conversion, the formula will be trivially satisfiable, because every clause contains a distinct selector variable. Nevertheless, finding a satisfying assignment with the lowest cost is not easy. The minimum cardinality problem is NP-hard for CNF formulas. However, it can be solved efficiently once we have the formula in d-DNNF. Any solution to this problem can be converted back to a solution for the original Max-SAT problem by ignoring assignments of the selector variables.

At this point, we are almost ready to compile the CNF formula. The only remaining issue is the time complexity of the compilation, which is, in the worst case, exponential in the treewidth of the constraint graph of the CNF formula. In most cases, straight compilation will be impractical. As a result, we need to relax the formula to lower its treewidth.

2.2 Problem relaxation by variable splitting

The approach we use to relax the problem is called *variable splitting*, which was inspired by the work of Choi *et al* in [15]. In general, splitting a variable v involves introducing new variables for all but one occurrence of v in the original CNF

formula. [3] For example, splitting a in the CNF $(a \vee b) \wedge (\neg a \vee c) \wedge (a \vee d) \wedge (b \vee \neg c) \wedge (c \vee \neg d)$ results in the formula $(a \vee b) \wedge (\neg a_1 \vee c) \wedge (a_2 \vee d) \wedge (b \vee \neg c) \wedge (c \vee \neg d)$. In this case, a is called the *split variable*. The new variables (a_1 and a_2 in this case) are called the *clones* of the split variable. Figure 1 illustrates the constraint graph of the above CNF before and after the split.

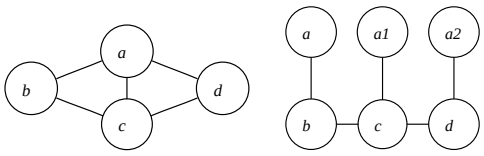

Fig. 1. (left) The constraint graph of $(a \vee b) \wedge (\neg a \vee c) \wedge (a \vee d) \wedge (b \vee \neg c) \wedge (c \vee \neg d)$. (right) The constraint graph after splitting a. The treewidth is reduced from 2 to 1.

After splitting, the resulting problem becomes a relaxation of the original problem because any assignment in the original problem has an assignment in the split problem with the same cost.[4] Therefore, the lowest cost of any split formula is a lower bound of the lowest cost of the original formula. The strategy we use for selecting split variables is the same as the one described in [15].

2.3 CNF to d-DNNF compilation

Once the treewidth of the problem is low enough, the problem can be practically compiled. The process of compiling a CNF formula into a d-DNNF formula is performed by a program called C2D [13, 16]. C2D takes a CNF formula as input and produces an equivalent sentence in d-DNNF. The output formula is fed to the search engine and will be used for later bound computations.

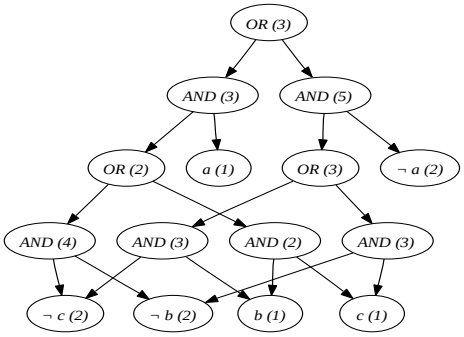

Fig. 2. The DAG of the d-DNNF formula $(a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))) \vee (\neg a \wedge ((b \vee \neg c) \vee (\neg b \vee c)))$. Each node in this graph is also labeled with the value used to compute the minimum cardinality of the root.

2.4 Computing bounds from d-DNNF

Every d-DNNF formula can be represented as a rooted DAG. Each node in the DAG is either a Boolean constant, a literal, or a logical operator (conjunction

[3] This type of splitting is called full splitting. While other degrees of splitting are possible, we focus our attention only to this method.

[4] This can be obtained by setting the value of every clone according to its split variable.

or disjunction). The root of the DAG corresponds to the formula. For example, consider the DAG of a d-DNNF formula $(a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))) \vee (\neg a \wedge ((b \vee \neg c) \vee (\neg b \vee c)))$ in Figure 2. Let the cost of every positive literal be 1 and the cost of every negative literal be 2. The minimum cardinality of the formula is simply the value of the root [14], which is defined recursively as:

1. The value of a literal node is the value of the literal
2. The value of an AND node is the sum of the values of all its children
3. The value of an OR node is the minimum of the values of its children

If the formula is a relaxed formula, then the computed minimum cardinality becomes a lower bound of the minimum cardinality of the formula before relaxation (hence a lower bound of the optimal cost of the original Max-SAT problem).

The d-DNNF formula can also be efficiently conditioned on any partial or complete assignment of its variables. Conditioning only affects the values of the nodes whose literals are set to false. The values of such nodes become ∞, which may in turn affect the values of their parents or ancestors. In practice, bound computation can be done incrementally. Only the nodes whose values have changed since the last evaluation need to be inspected. The resulting bound computed from the conditioned formula will be a lower bound of the optimal cost of any assignment that extends the conditioning assignment.

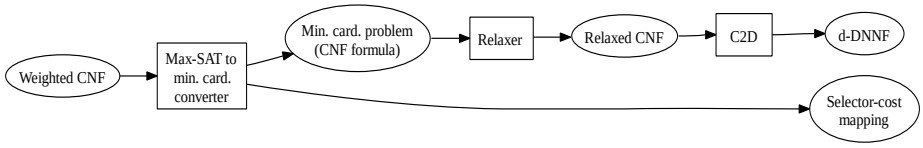

Fig. 3. A system diagram of Clone's preprocessor.

Figure 3 summarizes the relationship between different parts of the preprocessor. The system's input is a Max-SAT problem (weighted CNF formula). The system produces a d-DNNF formula and information about the costs of selector variables. These outputs are passed on to the branch-and-bound search engine.

3 Search and Inference

The next component of Clone is the branch-and-bound search engine. The engine uses bounds computed from the d-DNNF formula for pruning. The search algorithm only branches on variables in the original Max-SAT problem (as opposed to clones or selectors). Every time a split variable is assigned a value, the algorithm ensures that all its clones are set to the same value. Otherwise, a solution in this search space may be meaningless in the original problem.

Apart from standard inference techniques such as unit propagation, non-chronological backtracking [17–19], and conflict clause learning [18, 20], the search engine of Clone employs many novel techniques to improve the efficiency of the

search. We describe them next. Some of these techniques require access to the clauses in the original Max-SAT problem. Hence, although we compile the CNF formula into a d-DNNF, the original formula is also given to the search engine.

3.1 Reducing the size of the search space

Given our method for computing bounds, it is possible to reduce the size of the search space that the algorithm needs to explore. Since the relaxed problem differs from the original problem only on split variables and their clones, as soon as every split variable (and its clones) is set to a value, the two problems become identical under the current assignment. Therefore, the bound computed from the d-DNNF formula at this point must be the exact Max-SAT optimal cost of the original problem under the current assignment.

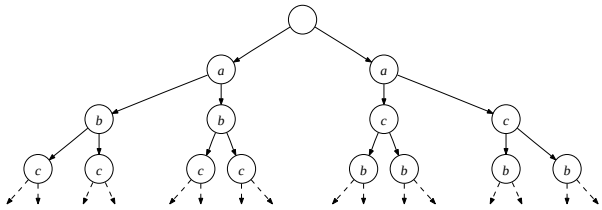

Fig. 4. An example search space with each node labeled with the last branch variable.

For example, consider the search space of a problem with 3 split variables– a, b, and c–in Figure 4. The bound computed at each node is a lower bound of the optimal cost of any complete assignment that extends the assignment at the node. However, once all three split variables are instantiated, the bound becomes exact. Therefore, there is no need to visit any node below depth 3 in this search tree. This realization suggests a natural way of reducing the size of the search space; we only need to search in the space of assignments of split variables.

3.2 Dealing with soft conflicts

Whenever $LB \geq UB$, the solver can prune the current branch of the search. This situation is called a *soft conflict*, because no hard clause is violated. A *hard clause* is one whose cost is greater than or equal to UB. A *soft clause* is a clause that is not hard. A hard clause cannot be falsified by any optimal solution.[5] Traditionally, branch-and-bound search algorithm backtracks chronologically (flip the most recent unflipped decision) when a soft conflict is found. The standard SAT techniques for dealing with conflicts do not apply directly to this type of conflicts, because of the lack of a violated hard clause. However, in some cases, we could efficiently construct a violated hard clause.

[5] Violation of a hard clause results in a hard conflict, which is handled by normal conflict analysis as in [20, 21].

Upon each soft conflict, clearly $LB \geq UB$. Moreover, some soft clauses may already be falsified by the current assignment. However, it is not necessary that the sum S of the costs of the violated soft clauses be greater than or equal to UB. This is simply because the method used for computing lower bounds may be able to implicitly identify certain clauses that can never be satisfied at the same time, even when their truth values are still in question. In any case, whenever $S \geq UB$, a violated hard clause can be artificially constructed. In particular, let $\{C_1, C_2, ..., C_k\}$ be a set of soft clauses whose sum of costs exceeds (or equal) UB. Then, $C = \vee_{i=1}^{k} C_i$ is a hard clause. The clause C simply makes explicit the fact that $C_1, ..., C_k$ cannot all be false at the same time.

To perform conflict analysis on a soft conflict, we need to find a minimal set of violated soft clauses whose sum of costs is greater than UB. The disjunction of the literals of these clauses is a violated hard clause, which can be used by the standard conflict analysis algorithm to generate a conflict clause, which is a hard clause, and to compute a level to non-chronologically backtrack to.

A similar learning scheme for soft conflicts was proposed in the solver PMS [22]. This learning scheme utilizes Max-SAT resolution rule [23] and, according to [22], has not been coupled with non-chronological backtracking.

3.3 Avoiding work repetition

Both chronological and non-chronological backtracking are employed by Clone. The former is used when the solver encounters a soft conflict for which no violated hard clause can be easily constructed. In this case, no new clause is learned by the solver. Non-chronological backtracking is used for the other pruning opportunities and a new conflict clause is learned every time.

Combining these two types of backtracking naively may lead the solver to repeat a lot of work in some situations. For example, consider the search tree in Figure 5 (a). In this search tree, the variable a was flipped to $\neg a$ by a chronological backtrack, which indicates that the whole subtree under $a = \textbf{true}$ had been exhausted. After a few more decisions (b and c), the solver may run into a hard conflict and derive a conflict clause ($\neg c$). As a result, non-chronological backtracking will erase every assignment and set $c = \textbf{false}$ at the very top level (level 0). The search tree after backtracking is shown in Figure 5 (b). The assignment $a = \textbf{false}$ had been erased in the process and there is no way to recover the fact that the branch $a = \textbf{true}$ had been fully explored. MiniMaxSAT [9] is the only other solver employing both chronological and non-chronological backtracking that we are aware of and it may suffer from this inefficiency.[6]

To solve this problem, every time Clone non-chronologically backtracks past any chronologically flipped assignment, it records a *blocking clause*, which is a hard clause that captures that information. More formally, let ℓ be a current literal assignment that was flipped by a chronological backtrack at level k. If the solver non-chronologically backtracks past ℓ, it will learn $C = (\ell \vee \neg d_1 \vee \neg d_2 \vee$

[6] Based on the paper and communication with one of the authors.

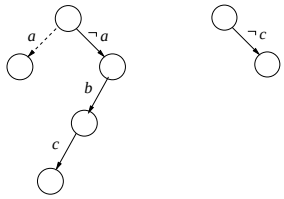

Fig. 5. (a) A search tree in which a has been flipped by chronological backtracking. (b) The search tree after non-chronological backtracking and learning conflict clause $(\neg c)$.

$... \vee \neg d_{k-1})$, where d_i is the decision assignment at level i. In the above example, $(\neg a)$ would be learned since it was the very first decision in the search tree.

4 Experimental Results

In this section, we present experimental results that compare Clone against other leading weighted Max-SAT solvers that are publicly available. The first solver is Toolbar (version 3.1), which was the best solver in the weighted category of the Max-SAT evaluation 2006 and also participated in the Max-SAT evaluation 2007. The second solver is Lazy, which was outperformed by only Toolbar in the same category in 2006. The last solver is MiniMaxSat, which outperformed others in the weighted partial Max-SAT category of the 2007 evaluation.

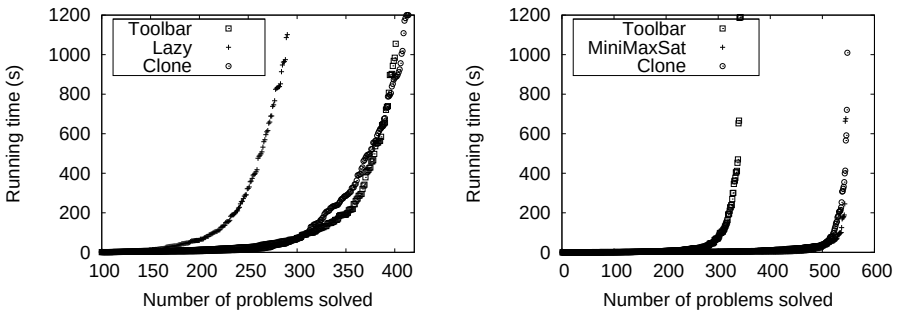

Fig. 6. Running time profile of solvers on Max-SAT problems from Max-SAT evaluation 2006 (left) and 2007 (weighted partial Max-SAT category) (right).

The version of Clone used here is an improved version of the one in the Max-SAT evaluation 2007. In addition to the techniques for handling soft conflicts described earlier, this version of Clone uses MaxWalkSat [24] to find initial seeds. Moreover, it utilizes an improved variable ordering heuristic which always starts with the Jeroslow-Wang heuristic [25] and dynamically switches to the VSIDS [26] heuristic if hard conflicts are encountered frequently enough. To make memory usage sustainable, Clone periodically deletes inactive learned clauses in the manner that maintains the completeness of the solver.

In all experiments, every Max-SAT problem is relaxed until its treewidth is less than or equal to 8, which we found to be empirically optimal. All experiments were run on two Intel Pentium 4 machines, each with 3.79GHz and

4GB of RAM. The time-out is 1,200 seconds per problem. We considered all non-random weighted Max-SAT problems from the 2006 evaluation and all non-random weighted partial Max-SAT problems from the 2007 evaluation.[7]

Suite (Max-SAT eval. 2006)	Pb. Ct.	Toolbar	Lazy	Clone
AUCTION (PATHS)	30	26	20	**28**
AUCTION (REGIONS)	30	30	30	30
AUCTION (SCHED.)	30	**30**	**30**	28
MAXONE	45	**44**	24	33
MAXCLIQUE	62	**33**	29	19
MAXCSP	180	120	53	**169**
QCP	25	10	6	**24**
RAMSEY	48	35	28	**36**
SPOT5	42	9	5	**12**
WMAXCUT	67	59	**60**	28
WQUEENS	7	5	5	5
Total	566	401	290	412

Suite (Max-SAT eval. 2007)	Pb. Ct.	Toolbar	MiniMaxSat	Clone
AUCTION (PATHS)	88	88	88	88
AUCTION (REGIONS)	84	84	84	84
AUCTION (SCHED.)	84	82	**84**	78
QCP	25	10	20	**24**
PLANNING	71	52	**71**	**71**
PSEUDO MIPLIB	16	**7**	5	5
PSEUDO FACTOR	186	8	**186**	**186**
SPOT5	42	9	7	**12**
Total	596	340	545	548

Table 1. Number of solved problems by suite (left) from the Max-SAT evaluation 2006 and (right) from the Max-SAT evaluation 2007

Figure 6 shows performance profiles of solvers on different set of benchmarks. In each plot, problems from each Max-SAT evaluation are used to compare Clone against solvers that participated in the evaluation. The left plot of Figure 6 shows that the performance of Clone is comparable to that of Toolbar and better than that of Lazy on problems from the 2006 evaluation. The right plot shows that Clone is comparable to MiniMaxSat, which does significantly better than Toolbar on problems from the 2007 evaluation. Table 1 reports the number of solved problems by suite. According to the left table, out of 566 problem, Clone solved 11 problems more than Toolbar and 122 problems more than Lazy. The right table shows that Clone solved roughly the same number of problems as MiniMaxSat on most suites and solved 3 more problems overall.

5 Conclusions

We presented in this paper a new weighed Max-SAT solver, Clone, which employs (1) a new approach for computing lower bounds, (2) a corresponding technique for reducing the search space, and (3) some novel techniques for handling soft conflicts. The performance of Clone on non-random problems used in the recent Max-SAT evaluations is competitive with those of other leading solvers, which are based on more mature approaches for Max-SAT. Our results demonstrate the potential of the new techniques adopted by Clone, some of which are quite different from the techniques adopted by classical Max-SAT solvers.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and for suggesting clarifications on the similarities of Clone and other solvers.

[7] In 2006, unweighted Max-SAT was the only other evaluation category. In 2007, there were 3 other categories. Here, we consider the broadest, most general category.

References

1. Fu, Z., Malik, S.: On solving the partial max-sat problem. In: Proc. of SAT'06
2. Le Berre, D.: Sat4j project homepage http://www.sat4j.org/.
3. Argelich, J., Li, C.M., Manya, F., Planes, J.: First evaluation of max-sat solvers http://www.iiia.csic.es/∼maxsat06/.
4. Argelich, J., Li, C.M., Manya, F., Planes, J.: Second evaluation of max-sat solvers http://www.maxsat07.udl.es/.
5. Larrosa, J., Heras, F., de Givry, S.: A Logical Approach to Efficient Max-SAT solving. ArXiv Computer Science e-prints (November 2006)
6. de Givry, S., Heras, F., Zytnicki, M., Larrosa, J.: Existential arc consistency: Getting closer to full arc consistency in weighted csps. In: IJCAI. (2005) 84–89
7. Alsinet, T., Manya, F., Planes, J.: A max-sat solver with lazy data structures. In: Proc. of 9th Ibero-American Conf. on Artificial Intelligence. (2004)
8. Li, C.M., Manya, F., Planes, J.: New inference rules for max-sat. JAIR (2007)
9. Viaga, F.H., Larrosa, J., Oliveras, A.: Minimaxsat: a new weighted max-sat solver. In: Proceedings of SAT'07. (2007)
10. Li, C.M., Manya, F., Planes, J.: Exploiting unit propagation to compute lower bounds in branch anb bound max-sat solvers. In: Proceedings of CP'2005. (2005)
11. Ramírez, M., Geffner, H.: Structural relaxations by variable renaming and their compilation for solving mincostsat. In: Proc. of CP-07.
12. Darwiche, A., Marquis, P.: A knowlege compilation map. Journal of Artificial Intelligence Research **17** (2002) 229–264
13. Darwiche, A.: New advances in compiling CNF to decomposable negational normal form. In: Proceedings of European Conference on Artificial Intelligence. (2004)
14. Darwiche, A., Marquis, P.: Compiling propositional weighted bases. Artificial Intelligence **157**(1-2) (2004) 81–113
15. Choi, A., Chavira, M., Darwiche, A.: Node splitting: A scheme for generating upper bounds in bayesian networks. In: Proceedings of UAI'07. (2007)
16. Darwiche, A.: The c2d compiler. Available at http://reasoning.cs.ucla.edu/c2d/.
17. Stallman, R., Sussman, G.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intel. **9** (1977)
18. Marques-Silva, J., Sakallah, K.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers (5) (1999) 506–521
19. Bayardo, R.J.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of AAAI'97. (1997) 203–208
20. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD. (2001) 279–285
21. Ryan, L.: Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University (2004)
22. Argelich, J., Manya., F.: Partial max-sat solvers with clause learning. In: Proceedings of SAT'07. (2007) 28–40
23. Larrosa, J., Heras, F.: Resolution in max-sat and its relation to local consistency in weighted csps. In: Proc. of the Intnl. Jnt. Conf. on Artfcl. Intel. (2005) 193–198
24. Selman, B., Kautz, H.: Walksat home page. http://www.cs.rochester.edu/u/kautz/walksat/.
25. Wang, J.: A branching heuristic for testing propositional satisfiability. In: Systems, Man and Cybernetics, 1995. 'Intelligent Systems for the 21st Century'., IEEE International Conference on. (1995) 4236–4238
26. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: 39th Design Automation Conference (DAC). (2001)