University of Potsdam
Faculty of Computer Science

Clause Learning in SAT

Seminar Automatic Problem Solving
WS 2005/06

Authors: Richard Tichy, Thomas Glase
Date: 25th April 2006

Contents

1 Introduction **1**

2 Motivation **1**

3 Definitions **2**

3.1 Conflict Clause 2
3.2 Decision Level 2
3.3 Implication Graph 3
3.4 Unique Implication Point 4
3.5 Cut 4

4 Learning Schemes **5**

4.1 1-UIP 6
4.2 Rel_Sat 9

5 Conflict Clause Maintenance **10**

5.1 Bounded Learning 11

6 Using Efficient Data Structures **11**

7 Clause Learning and Restarts **12**

8 Remarks on Complexity **13**

9 Benchmark Results and Comparison **13**

10 Conclusion **14**

Abstract

The development of clause learning has had a tremendous effect on the overall performance of SAT-Solvers. Clause learning has allowed SAT-Solvers to tackle industrial sized problems that formerly would have required impractical time scales. The development of techniques for efficient clause management and storage have also proved important in reducing some of the memory usage problems inherent in naive clause learning strategies. This paper attempts an introduction to some better known clause-learning strategies as a comparison among these strategies. A brief explanation of some of the techniques available to minimize memory usage when storing learned clauses in a database is also presented.

1 Introduction

The following paper is intended as both an introduction to some of the more important clause learning strategies common to modern SAT-Solvers and as a comparison between them. The first section discusses the general motivation behind clause learning. Following this is a discussion and definition of fundamental terms without which any further discussion or comparison of learning strategies would be incoherent. After this two important clause learning strategies are discussed. This discussion includes an explanation of how learned clauses are resolved in each strategy discussed. The following sections include brief discussions on techniques for clause maintenance as well as strategies for reducing the memory 'footprint' involved with storing learned causes in a usable database. The paper continues with a short discussion of the affect of clause learning on another powerful SAT-Solver strategy, namely restarts. After this, worst case complexity for clause learning in general is briefly discussed and benchmark results for each of the learning strategies discussed earlier are presented. The paper concludes with a summary of the more important ideas presented earlier.

2 Motivation

The performance of common SAT-Solvers relies primarily on two issues, namely unit propagation and backtracking. Unit propagation involves inferring information from clauses that have a single unassigned literal and assigning the literal a value such that the clause is then satisfied. Backtracking involves returning to the point in a search where a literal was assigned a value leading to a contradiction and reassigning that literal another value. In Boolean satisfiability problems this means assigning the opposite value of the previous assignment. Naive backtracking schemes simply jump back to the most recent choice variable assigned. This technique might lead to subtrees involving conflicts that were already encountered in a previously explored subtree. Such repetitions of conflict resolution, if possible, should be avoided in order to increase the performance of the search. Backjumping with learning focuses on this task and, through the use of learned clauses, prunes portions of the search tree that can never contain a solution.

Once clause learning is adapted as a strategy to improve solver performance, the classic tradeoff between runtime performance and memory usage is encountered. Unrestricted clause learning, where all learned clauses are stored in a database, is infeasible due to the memory required for storing clauses resolved at all conflicts. Furthermore, as the clause database becomes larger the time spent in unit propagation becomes longer as well. It becomes possible that any advantage gained by retaining learned causes is negated due to the increase in time spent in propagation. The motivation

behind techniques such as size bounded and relevance bounded learning rests in helping to overcome such problems at the expense of losing some of the information stored in learned clauses.

3 Definitions

In this section some essential definitions are given. These terms are needed to understand the different learning schemes described in chapter 3.

3.1 Conflict Clause

A *conflict clause* represents an assignment to a subset of variables from the problem that can never be part of a solution. By adding a conflict clause to a set of clauses, a kind of constraint is created, that excludes a corresponding variable assignment involving a conflict. Such a conflict is hard to detect in the worst case because participating variables might be scattered over multiple clauses. This situation is overcome by representing such a conflicting variable assignment with one clause. In addition, a conflict clause may imply a reason for the conflict to occur. This holds some importance especially in unsatisfiable problem cases, in which the question of why a problem is not satisfiable might be of interest. Learning in SAT involves finding and recording conflict clauses. A proper conflict clause must meet some properties, namely:

- the clause must be an Asserting Clause, thus containing exactly one literal assigned at the conflict level.
- the clause must be logically implied by the original set of formula, to ensure correctness.
- the clause must be made false by the variable assignment involving the corresponding conflict.

3.2 Decision Level

During the process of problem solving each variable assignment takes place at certain decision levels starting from zero upwards. If the problem can be solved by unit propagation alone, then each variable is assigned at decision level zero. With each encountered decision variable the decision level is incremented by one. Implied variables have their values decided upon in unit propagation. Every implied variable gets the same decision level as the previous decision variable. Therefore in the worst case, where no values are assigned to variables during unit propagation, the maximum decision level is equal to the number of literals occurring in the set of clauses.

3.3 Implication Graph

An *Implication Graph*, furthermore referred to as an I-Graph, is a directed acyclic graph where each vertex represents a variable assignment,or in other words a literal. An incident edge to a vertex represents the reason leading to that assignment. These reasons are clauses that became unit and forced the variable assignment. For this reason, decision variables have no incident edges in contrast to implied variables that have assignments forced during unit propagation. As discussed earlier, each variable (decision or implied) has a decision level associated with it. If a graph contains a variable assigned both 1 and 0, that is both x and $\neg x$ exist in the graph, then the I-Graph contains a conflict.

An I-Graph is build according to the following formal steps.

Building the I-Graph G

1. Add a node for each decision labelled with the literal. These nodes have no incident edges.

2. While there exists a known clause $C = (l_i \vee ...l_k \vee l)$ such that $\neg l_i$,...,$\neg l_k$ are in G:

   a. Add a node labeled l if not already in G.

   b. Add edges (l_i, l) for $1 \leq i \leq k$ if not already extant in G.

   c. Add C to the label set of these edges to associate the edges as a group with clause C.

3. (optional) Add a node λ to the graph and add a directed edge from a variable occurring both positively and negatively to λ, which could be thus referred to as the conflict node.

Let's consider a small example with a partial assignment which makes x_1 the current decision variable and let's assume further that x_1 was decided to be false. The current decision level in the example is 4.

Clauses:

$w_1 = (x_1 \vee x_2)$

$w_2 = (x_1 \vee x_3 \vee x_7)$

$w_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$

$w_4 = (\neg x_4 \vee x_5 \vee x_8)$

$w_5 = (\neg x_4 \vee x_6 \vee x_9)$

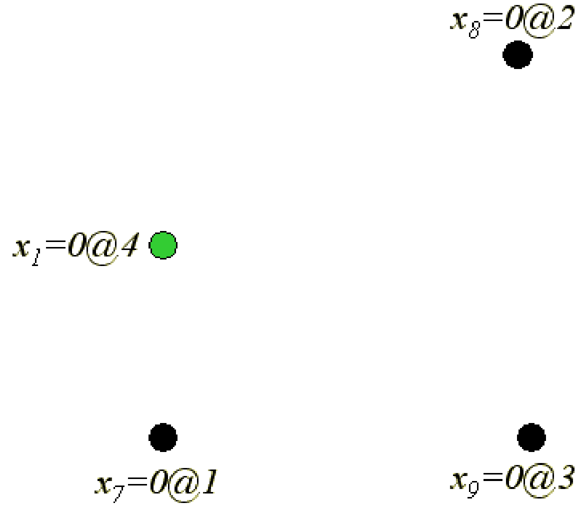


Figure 1: I-Graph after step 1

$w_6 = (\neg x_5 \vee \neg x_6)$

Current partial assignment: $\{x_7 = 0@1, x_8 = 0@2, x_9 = 0@3\}$

Current decision assignment: $\{x_1 = 0@4\}$

According to step 1 of the formalization given above of how to build an I-Graph, we get a temporary graph (see Figure 2) with no edges that contains only the nodes corresponding to decision variables x_7, x_8, x_9 and x_1.

Following the rules of step 2, the complete I-Graph (see Figure 3) is build step by step and finally depicts the conflict that arises through x_5 which is implied to be both true and false by the current assignment.

3.4 Unique Implication Point

A *Unique Implication Point* (UIP) is any node at the current decision level such that any path from the decision variable to the conflict node must pass through it.

3.5 Cut

In case of a conflict, the I-Graph can be split by a bipartition called a *cut*. The two sides of the partition can be referred to as the conflict side and the reason side of the implication graph. The conflict side contains

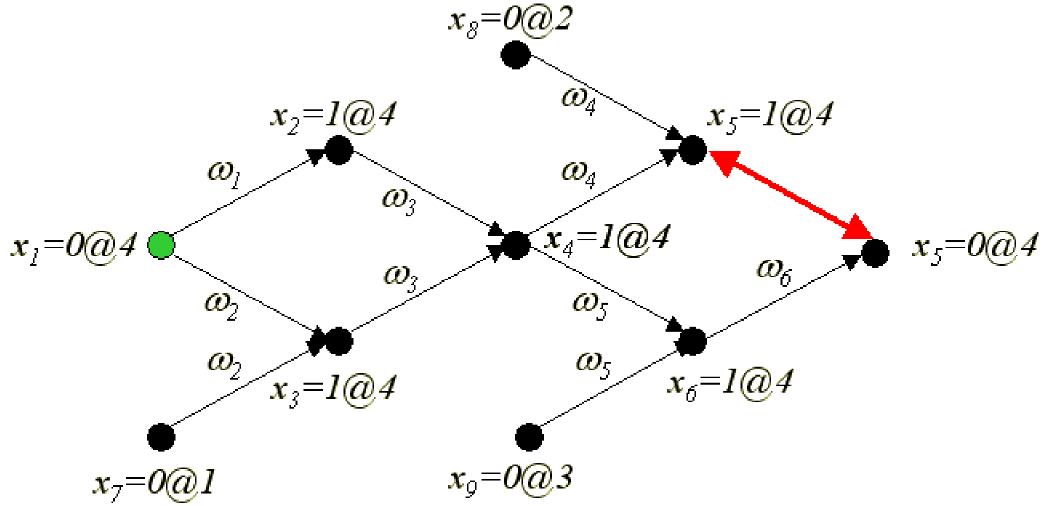
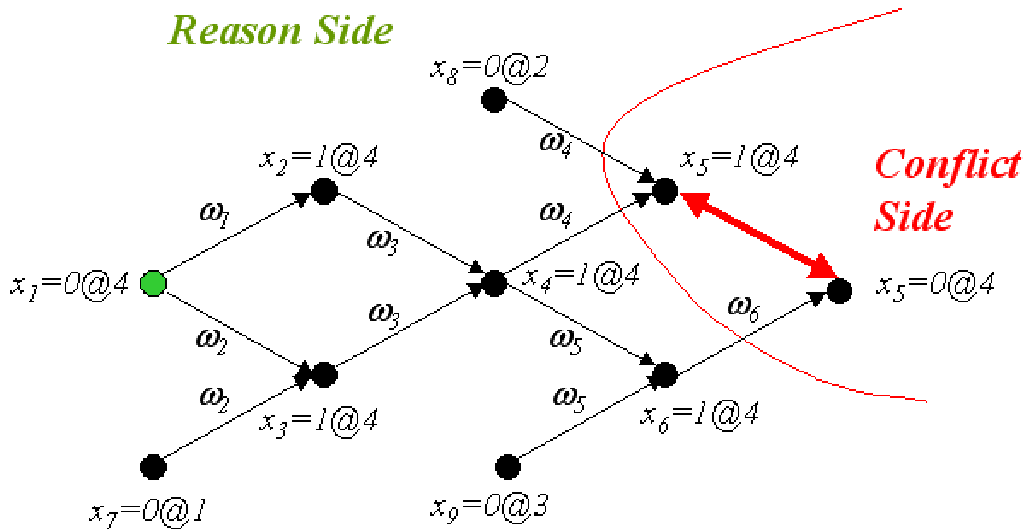


Figure 2: Complete I-Graph containing a conflict

the conflicting nodes. The reason side contains the nodes of the I-Graph bipartition not included in the conflict side. The conflict side can contain more nodes than just the conflict node λ and the conflicting nodes with edges leading to λ. There are various ways for creating such extensions of the conflict side and they are equivalent to various cuts. Different cuts of the I-Graph distinguish learning schemes from one another, because the conflict clause, and thus the knowledge gained from the conflict, is derived from the bipartition of the I-Graph. Two important techniques for producing cuts in the I-graph are described in the next section.

4 Learning Schemes

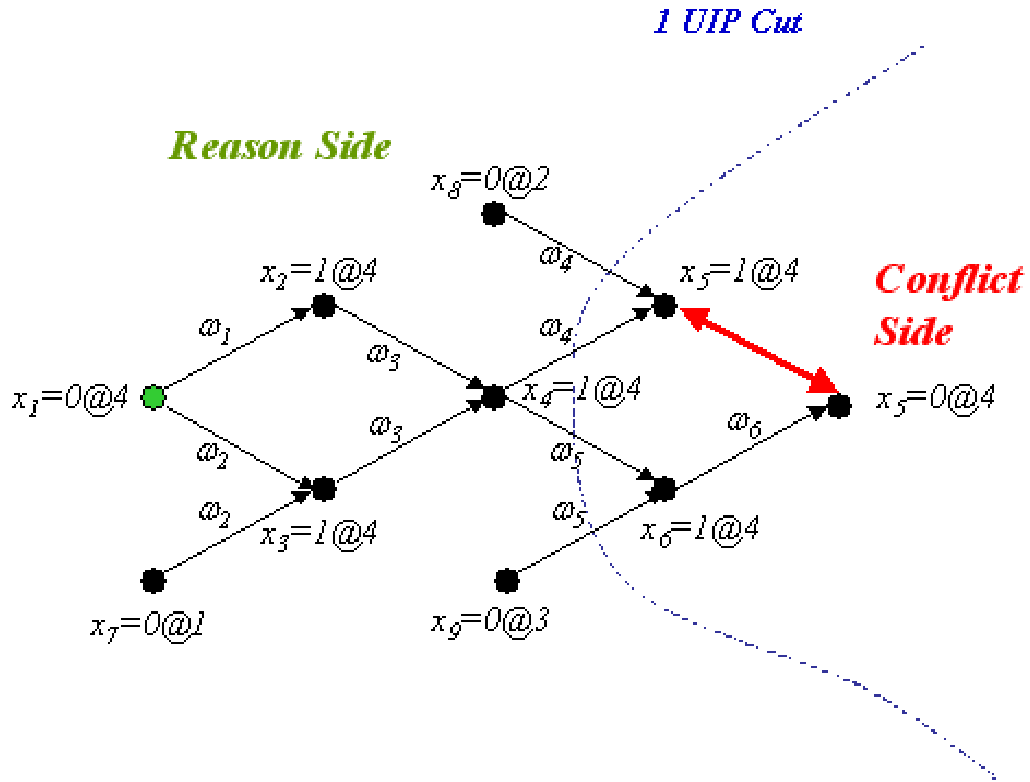
As already mentioned above, different learning schemes correspond to different cuts. In other words, different extensions of the conflict side of the I-Graph correspond to different cut techniques. Different cuts generate different conflict clauses which are then added to the database of clauses. This storage of these additional learned clauses represents the learned portion of the search.
Let's consider our small example, where a conflict occured through x_5. The conflicting nodes are on the conflict side, all other nodes on the reason side.



In the following, 2 learning schemes will be described. The 1-UIP scheme, applied by MiniSAT and zChaff [7, 8], and the technique used by ReLSat [1].

4.1 1-UIP

The 1-UIP scheme corresponds to the 1-UIP cut or First-UIP cut. Both terms describe the same cut which divides the I-Graph right before the first UIP encountered on the path leading from the conflict node back to the decision variable. That way, the cut is set close to the conflict by this extension. The notion behind which is that hopefully the knowledge gained is as relevant to the conflict as possible. In our example there are only two UIPs, x_4 and the decision variable x_1 itself. The first UIP is x_4.
All nodes following x_4 belong to the conflict side, all other nodes, including x_4, belong to the reason side.



Once the cut is decided upon the question of how to derive the conflict clause from the bipartioned I-Graph remains to be answered.

Deriving the Conflict Clause The conflict clause consists of all nodes, belonging to the reason side, that have edges leading into the conflict side. The variables represented by these nodes, in our example x_4, x_8, x_9 have to be negated in the conflict clause according to their current assignment, since the conflict clause should be made false by, and thus exclude, an assignment leading to the conflict. Therefore in our example above the derived conflict clause C would be:

Conflict Clause: $C = (\neg x_4 \vee x_8 \vee x_9)$

The updated database of clauses would be:

$w_1 = (x_1 \vee x_2)$

$w_2 = (x_1 \vee x_3 \vee x_7)$

$w_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$

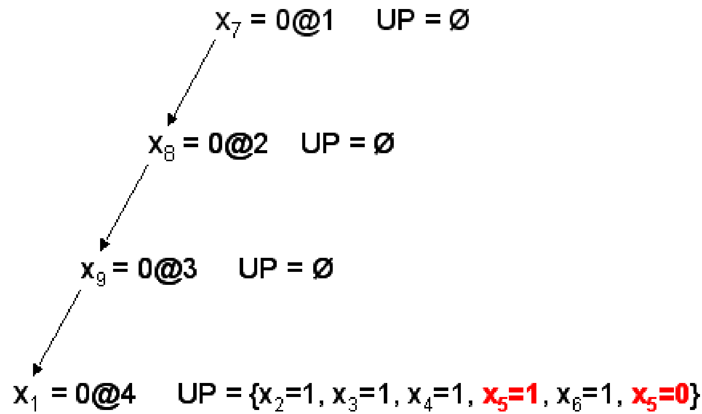
$w_4 = (\neg x_4 \vee x_5 \vee x_8)$

$w_5 = (\neg x_4 \vee x_6 \vee x_9)$

$w_6 = (\neg x_5 \vee \neg x_6)$

$C = (\neg x_4 \vee x_8 \vee x_9)$

This additional conflict clause affects the backtracking process in the following way. With naive backtracking the solver would just backtrack to the decision variable at the current decision level. That is in our example x_1, which would then be set to true. But this branch can lead exactly to the same conflict through x_5 as before.

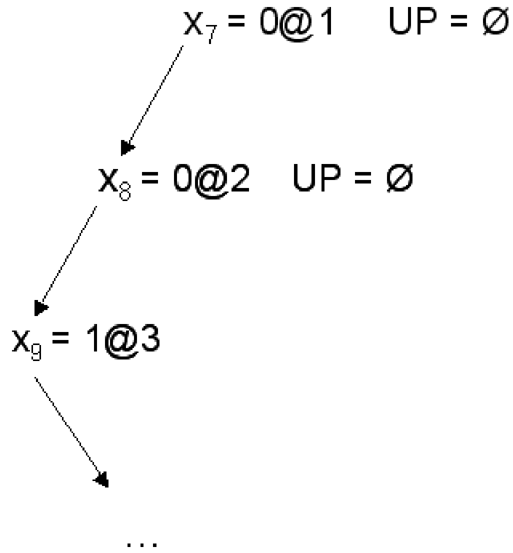


Effect on Backtracking With clause learning the backtrack level is determined by analyzing the conflict clause.

$$backtracklevel = max\,\{level\,(x) : x\ is\ an\ element\ of\ C - p\}$$

p is the one literal of C assigned at conflict level.

In our example $p = x_4$ and the maximum (decision) level of the other literals in C is 3. The solver backtracks to x_9 and flips it assignment to 1.



Current partial assignment: $\{x_7 = 0, x_8 = 0, x_9 = 1, x_4 = 1\}$

Take note that the former assignment of p, in our example x_4 is not discarded but kept in the new branch although the the backtrack level is smaller than the decision level at which p was assigned. Actually this is what forces the decision variable that was backtracked to to flip since the conflict clause is made unit.

$$C = (\neg x_4 \vee x_8 \vee x_9)$$

$x_8 = 0$ remains untouched because it was assigned at decision level 2, $x_4 = 1$ is kept and C becomes a unit clause which forces x_9 to flip to 1. If we pursue this new branch with the partial assignment we arrive at a solution for the problem which my be:

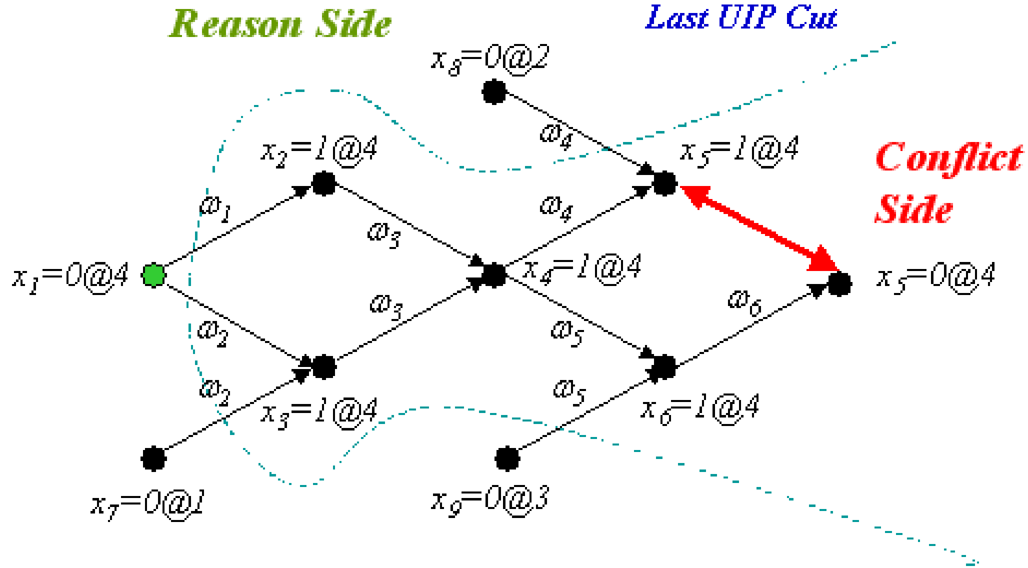
$$\{x_7 = 0, x_8 = 0, x_9 = 1, x_4 = 1, x_5 = 1, x_6 = 0, x_1 = 0, x_2 = 1, x_3 = 1\}$$

Most importantly, it is no longer possible to arrive at the same conflict involving x_5 again, which is what was to be expected.

4.2 Rel_Sat

In contrast to the 1-UIP scheme, The Rel_Sat scheme corresponds to what is referred to as the Last-UIP cut [1]. This cut divides the I-Graph right before the last UIP. The last UIP is the decision variable at the conflict level itself. In our example this corresponds to literal x_1. In this extension the conflict side contains all literals assigned at the conflict level except the

decision variable itself. Conversly, the reason side consists of all variables assigned at levels with a value less than that of the conflict level and the actual decision variable whose assignment was responsible for the confict.



The conflict clause derived would be:

$$C = (x_1 \vee x_7 \vee x_8 \vee x_9)$$

This conflict clause holds more information than the clause formed in the 1-UIP scheme. Nevertheless, in our example the solver would backtrack to level 3 and flip x_9 to 1 as well, which will lead to a possible solution in the best case. The difference is, besides the conflict clause, the p variable, in our example x_1, whose assignment is kept after backtracking.

5 Conflict Clause Maintenance

As mentioned earlier clause learning, if not somehow restricted, can quickly create memory related problems due to the potentially large number of learned clauses. In the learning schemes discussed above each conflict can potentially add at least one clause to the problem database for every confict discovered. Furthermore, the conflict clauses themselves can become very large as the search progresses. With no restrictions on the size of clauses added and no mechanism to discard irrelevant clauses, clause learning can add a prohibitive overhead in terms of the space needed for clause database storage. Also, the time spent for unit propagation increases with the number

of learned clauses added. Thus retaining all learned clauses is impractical. Bounded learning is one strategy that can help alleviate the problems of memory overhead and excessive time spent in propagation.

5.1 Bounded Learning

The different types of bounded-learning discussed here can be used with the learning schemes discussed in the previous section to restrict the recording of clauses based on some quality and to discard learned clauses that are no longer relevant. The two types of bounded learning discussed here are size-bounded learning and relevance-bounded learning.

Size-bounded learning constrains the size of the learned clauses to be less than some number i and is called i-order learning. Relevance-bounded learning, on the other hand, adds no constraints to the addition of clauses but instead discards learned clauses when they are no longer relevant according to some metric. For instance with i-relevance, a learned clause is not considered relevant if it differs by the current assignment by > i assignments or when > i literals in the clause are unassigned. The idea then is that i-order relevance maintains only the set of learned clauses that are i-relevant [6, 1].

Clauses can be removed from the database periodically based on one (or a combination) of the above bounding strategies. Consider, as an example, the conflict clause $(x_1 \vee x_7 \vee x_8 \vee x_9)$ and the literal assignment $x_7 = 0, x_2 = 1, x_4 = 0, x_6 = 1, x_5 = 0$. The preceding conflict clause would be removed from the database using i-relevance bounding because more than three literals of the clause do not currently have assignments or if more than three values for literals in the clause differed from those of the current literal assignment. In other words, when i > 3. The clause would also be deleted using size-bounded learning if the size of learned clauses were bounded at 4.

Bounded learning helps to reduce the potentially exponential addition of learned clauses by restricting the addition of arbitrarily large clauses. This is also helpful due to larger clauses being more expensive to process. Shorter clauses also tend to be more relevant to literal assignments higher in the search tree. Bounded learning also removes clauses which are less relevant to the locality of the current search. Ideally all learned clauses would be recorded, thus maximizing the information learned during the search. However the storage complexity and the added overhead of having to maintain and examine all recorded clauses when building the i-graph, encountering a conflict or during propagation make any such strategy too expensive to be useful.

6 Using Efficient Data Structures

As discussed in the previous section, naive clause recording can incur a heavy cost in terms of memory storage. This problem of memory consumption can

further be reduced using specialized data structures. The use of appropriate data structures can reduce the database memory usage associated with learned clauses.

One such strategy for reducing the memory required for clause learning is the use of cache aware implementations for learned clause storage. Cache aware implementations attempt to avoid on-chip cache misses by using arrays rather than pointer based data structures when storing clauses. Storing data this way results in more memory accesses involving contiguous memory locations. The idea is to minimize the number of main memory accesses which tends to be far less efficient than accessing cache memory.

Special data structures are also implemented for shorter stored clauses. These data structures keep a list for each literal p of all literals q for which there is a binary clause $\neg p \vee q$. The list is then scanned during unit propagation when assigning literal p the value true. This idea can also be extended to ternary clauses. All clauses can also be ordered by size, smallest to largest, during unit propagation. While there exist other strategies for reducing the time spent in unit propagation, such as the two-watched literal scheme, they are beyond the scope of this paper. Each of the forementioned strategies can have negligible decrease on the time spent in unit propagation and on memory consumption.

7 Clause Learning and Restarts

Restarts, as well as recording learned clauses, have proven to be a powerful strategy in improving the efficiency of SAT solvers. While a full explanation of restarting strategies in SAT solvers is beyond the scope of this paper, a brief description of the process is helpful to understand the impact of clause learning on SAT-Solver restart strategies.

During the search, the SAT solver may enter an area where useful conflict clauses are not produced, resulting in a phenomenon known as thrashing. Restarting the search attempts to solve this problem by throwing away the current literal assignment (except for those assignments created during the initial unit propagation) and then restarting the search from "scratch". Without clause learning there is a possibility the solver, after a restart, will follow a similar path through the search tree as the preceding iteration.

Clause learning reduces this potential for reproducing searches. Since most SAT solvers that utilize clause learning retain the cache of the clauses from the previous start, the newly started search will avoid the conflicts learned in the previous search iteration stored in the learned portion of the previous search. Such a "memory" from a previous restart will usually result in different variables being instantiated during unit propagation. This ultimately results in a different path through the search tree. Potentially duplicated searches are further reduced when using clause learning as the

sheer number of learned clauses typically tend to dominate the original input formula. Due to this the learned clauses, after time, have a greater affect on the choice of decision literal. This is especially true when the choice of decision literal itself is based upon some heuristic involving the clause database.

8 Remarks on Complexity

All learning schemes previously discussed can be implemented using a graph traversal algorithm of the implication graph. The worst case behaviour of this algorithm is O(V+E) where V is the number of vertices of the implication graph and E are the number of directed edges within the graph.

9 Benchmark Results and Comparison

The following section presents a comparison of runtimes for the two learning schemes discussed earlier, namely the First-UIP and Rel Sat learning schemes. Data for a third more complicated learning scheme (GRASP) [5]. In the comparison discussed here two distinct heuristics were used for choosing decision literals. The first and simplest method used is referred to as fixed branching and the second is the VSIDS (Variable State Independent Decaying Sum) heuristic. Three classes of benchmarks are used for comparison. The first is formal microprocessor verification, the second is bounded model checking and the third consists of a planning problem. The numeric value in brackets beside the name of each specific benchmark is the number of times the problem was run. The times presented are the average of all runs. The numeric value in red beside some average times indicates the number of times the problem completed. Times that exceeded the arbitrary cut-off are not factored into the average times discussed here. The literature surveyed did not elaborate on any of the specific benchmarks within each benchmark class. Nevertheless definite improvements are seen with some learning schemes over others and between the two different decision heurisitics used.

The fixed branching heuristic chooses the first unassigned variable with smallest index based upon some variable preordering. The VSIDS heuristic is more complicated [7, 8]. For each literal l in the original formula keep a score $s(l)$ which, initially, is the number of occurrences in the formula of the literal. The idea is then to increment $s(l)$ each time a clause with l is added to the database of clauses. After N number of decisions the score $s(l)$ is recomputed so that $s(l) = r(l) + s(l)/2$ where $r(l)$ is the number of occurrences of l in conflict clauses since the last update. The heuristic would choose the literal l with te highest $s(l)$ value. The idea is that this decsion

	Microprocessor Formal Verification			Bounded Model Checking			Planning
	fvp-unsat.1.0 (4)	sss.1.0 (48)	sss.1.0a (9)	barrel (8)	longmult (16)	queueinvar (10)	satplan (20)
1UIP fixed	11.36 (3)	15307.13 (3)	2997.37	281.48 (1)	3141.8	817.07 (5)	18 (2)
1UIP VSIDS	532.8	24.56	10.63	1012.62	2887.11	6.58	39.34
grasp fixed	22.51 (3)	6497.99 (8)	3382.47	32646 (1)	5597.1	792.12 (5)	149.8 (2)
grasp VSIDS	2222.44	94.64	33.99	654.54	6196.82	97.82	309.03
rel_sat fixed	80.94 (3)	3114.83 (16)	4261.25 (4)	293.09 (1)	4396.73	478.37 (6)	3323.71 (3)
rel_sat VSIDS	2034.09	193.93	82.51	292.33 (1)	5719.73	14.4	96.61

Figure 3: Experimental results for 1UIP, Grasp and Rel Sat.

would be relevant to the most clauses in the database and thus be more likely to generate the most unit clauses during propagation.

The benchmark results clearly show that while the VSIDS decision heuristic may not always be faster, it is certainly more robust. 1UIP, considered the simplest strategy to implement, outperformed Rel Sat and GRASP in all but 2 benchmarks. In every instance but one all runs were completed before the cut-off using the VSIDS decision heuristic, regardless of the learning scheme. In most cases the VSIDS heurisitic was faster or at least competitive. Exceptions to this are the first verification benchmark, fvp-unsat.1.0, and the barrel model checking problem instance. The learning scheme that proved to be the fastest in all instances was the simplest, namely the 1UIP learning scheme. Grasp appears to have the advantage in microprocessor verification and Rel Sat outperformed Grasp in the model checking and planning instances.

Interesting to note is that the SAT-Competition Winner for 2005, SATElite incorporated both the 1UIP learning scheme and VSIDS decision heuristic. The competition consists of what are considered industrial sized SAT problems with potentially millions of literals and clauses.

10 Conclusion

In the previous discussion it is clear that clause learning has the potential to speed up the solving process by pruning the search space of subtrees that cannot contain solutions. This pruning is accomplished through the recording of the reasons for conflicts that occur during the search. These clauses are recorded in a database that includes the original clauses of the

problem instance as well. The learned clauses can be used to choose decision literals (VSIDS) and are used during unit propagation to avoid decisions made earlier in the search that led to conflicts.

Some form of conflict clause maintenance is necessary for practical SAT applications due to the potential of an exponential number of conflict clauses being discoverd. Bounding learned clauses based on either size or relevancy helps reduce the size of the clause database. Unbounded learning is simply not feasible for larger problem instances due to exponential storage requirements and the prohibitive time spent in propagating when the database becomes to large. This is to say there is a trade off between the number of clauses stored, the speed of the resulting search and the amount of memory used. Efficient data structures can also reduce the memory needed by SAT solvers and can also speed up unit propagation. Worth noting is the power that clause learning adds to the restart strategy. Erroneous paths through the search space can be "memorized" if the database of learned clauses is retained from restart to restart.

Experimental results clearly showed that the 1UIP learning scheme outperformed the Grasp and Rel Sat schemes when used with the VSIDS branching heuristic. Even when not using VSIDS, IUIP failed to outperform Grasp and Rel Sat in only one benchmark instance. Regardless of the learning scheme involved, it is commonly agreed upon that clause learning is a major step forward in the efficiency of SAT-Solvers.

References

[1] R. Bayardo and R. Shrag. Using csp look-back techniques to solve real-world sat instances. *Proc. of the Internationall Conf. on Automate Deduction*, 1997.

[2] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Understanding the power of clause learning. *IJCAI*, pages 1194–1201, 2003.

[3] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

[4] Vladimir Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, pages 69–129, 1996.

[5] J.P. Marques-Silva and K.A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, 48(5):506–521, 1999.

[6] David G. Mitchell. A sat solver primer. *EATCS Bulletin (The Logic in Computer Science Column)*, 85:112–133, 2005.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. *Proc. Design Automation Conf.*, pages 530–535, 2001.

[8] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. *Proc. ICCAD*, pages 279–285, 2001.