Literal-Independent Clause Processing for Unit Resolution

Trevor Standley and **Chu-Cheng Hsieh**
Computer Science Department
University of California, Los Angeles
{tstand, chucheng}@cs.ucla.edu

Abstract

We present a new and substantially different unit propagation technique based on the two literal watch scheme used in modern SAT solvers that allows for significantly shorter learned clauses. We have implemented our method on top of TiniSAT 0.22 (a lightweight solver written in C++). Benchmarks were run on instances from the 2005 SAT competition, and SAT race 2006, and results show that our method learns clauses an average of 26% shorter than unmodified TiniSAT, resulting in an average of 17% fewer decisions before a solution is found.

Introduction

Unit resolution is an important component in most modern SAT solvers. It allows a solver to prune the search space from an unruly to a manageable size, and its ability to detect dead-ends in the search tree can be improved by conflict clause learning [1].

The ability of a conflict clause to detect dead-ends depends in part on the size of the conflict clause. Smaller clauses make stronger statements, and place tighter constraints on possible variable assignments. The goal of this paper is to devise a method of deriving stronger conflict clauses, and to explore alternatives to the standard unit propagation algorithm.

We begin with a review of modern SAT solving methods focusing on the implementation details of TiniSAT, the solver on which we have made our modifications. We move on to where our approach differs from that of TiniSAT, and describe in detail the changes we have made. We then describe our experiments, the results, and our analysis of what happened. We conclude with some suggestions for future work in this area.

TiniSAT

TiniSAT[2] is a lightweight SAT solver written in C++ to be readable and easy to modify. For this reason, we found TiniSAT to be the ideal testbed of our ideas.

We will now give an introduction to the structure of TiniSAT. TiniSAT is arranged in a way that is quite typical of modern SAT solvers, and so the following can be seen as an introduction to SAT solving in general.

Algorithm TINISAT

```
1.    loop
2.      if (literal = selectLiteral()) == nil
3.        return SATISFIABLE
4.      if !propagate(literal)
5.        repeat
6.          learnClause()
7.          if assertionLevel() == 0 then
8.            return UNSATISFIABLE
9.          if restartPoint() then
10.           backtrack(1)
11.         else
12.           backtrack(assertionLevel())
13.       until assertLearnedClause()
```

Figure 1. Pseudo-code of TiniSAT from [8]

TiniSAT consists of a main loop (Figure 1), which contains each of the elements described below.

At every iteration, TiniSAT makes an assumption about the value of one of the variables (`selectLiteral()`). TiniSAT uses a decision heuristic to decide what assumption to make, i.e., which variable and what value, but it will always choose a variable the value of which unit propagation has not yet discovered under the current set of decisions. If no such literal exists then the values of all literals have either been decided on, or the values have been discovered without a contradiction, and therefore the instance is satisfied.

In order to decide which literal to assume next, TiniSAT uses a modification of the VSIDS heuristic. TiniSAT starts by giving a score to every literal. Specifically, the initial value of each literal is the number of clauses in which it occurs. Every time a clause is learned, the score of every literal in the clause is incremented. However, in order to encourage TiniSAT to choose literals that are active in recently learned clauses, the score of each literal is halved every 128 conflicts. The score of a variable is the sum of the two literals it can take on. The rationale is that the values of variables that are in many clauses are harder to determine, and variables that appear in recent conflict clauses are nearer to the "heart" of the problem. In order to choose a variable, TiniSAT goes through a list of recent conflict clauses and chooses the variable with the highest score. If an assumption about the value of the variable has been made before, TiniSAT picks the value that the variable was last assumed to have unless the score of one of the values (as in positive or negative) is

significantly higher than the score of the other. If there are no clauses to choose from, TiniSAT falls back on traditional VSIDS which can choose a variable even if it is not in a recently learned clause. This heuristic is proposed by Jinbo[2], which is essentially a combination of Chaff[5] and BerkMin[6].

TiniSAT then discovers as much as it can about the ramifications of the current decision by running unit propagation (`propagate(literal)`). Unit propagation may find a contradiction in the current decision sequence, and in this case the algorithm will do two things. First it will discover a clause that is implied by the original knowledge base and add it to the clause list in the hopes that it will make unit resolution more capable of finding a contradiction in the future. Second, the algorithm will backtrack.

Starting with chaff[4], SAT solvers perform unit propagation in an efficient way. Namely, they use a two literal watch scheme. In order to tell when a clause has become unit, SAT solvers "watch" two free literals in each clause. Every variable has two watch lists (one corresponding to the positive, and another corresponding to the negative) containing each of the clauses for which it is a watch variable. When the value of a watch variable becomes set, the watch list for the literal opposite in sign is traversed, and each clause it contains is processed. When a clause is processed, at least one of its watch variables must have become negative (resolved) and so a new watch variable must be selected if possible. If it is not possible to find a suitable free literal in a clause then the clause has become unit (or empty if both watch variables are resolved). If a clause becomes unit, the literal it contains must be true, the variable's value is set, and it is put onto a stack so that its watch list may be processed in order. Implied variables are given a pointer to the clause that became unit for conflict clause learning. It is only necessary to watch two literals because a clause cannot become unit if two literals are not both false.

Upon detection of an inconsistency in the state of the solver via unit resolution, solvers perform a series of resolution steps to derive a clause implied by the knowledge base. This is called the conflict clause. Solvers resolve clauses that became unit (called reasons or antecedents) with the empty clause until the derived clause only contains one literal from the current decision level. Remember that shorter conflict clauses typically make stronger statements about the solution, and allow subsequent runs of unit resolution to detect inconsistency faster. For this reason it is important for the antecedents to be short. It is also important for the antecedents to share literals; this is because the resolvent will only contain one version of any shared literals. It is called merge resolution when the two clauses being resolved share a literal, and merge resolution is important because it is the only way for the conflict clause to become shorter.

TiniSAT, like most modern SAT solvers, processes the watched clauses on an implied literal's watch list in sequential order, and chooses watch lists to process arbitrarily. The order in which clauses are processed is important because it is only the first clause that implies a particular literal or becomes empty that has a chance to participate in the derivation of the conflict clause.

Backtracking involves unmaking some number of recent decisions that were discovered to be erroneous. Every learned clause contains one literal that was implied at the current decision level, the second highest level in which a literal in the conflict clause was implied is the backtrack level. It is safe to unmake all decisions made after this decision, i.e., every such decision must cause inconsistency. `backtrack()` simply sets the value of every variable implied after the backtrack level to free.

Intuitively, sometimes the decisions made by SAT solvers lead it to a place in the decision tree where the solver can wander aimlessly for a large period of time. In order to reduce the amount of time spent wandering, solvers sometimes induce a restart to the very beginning of the decision sequence. This is done using a restart schedule, or restart policy.

TiniSAT uses a well-known restart policy which was proposed by Luby et al.[7] They were able to prove optimality guarantees for their policy for many types of algorithms that have randomly distributed runtimes. They showed that (1) it approximates the optimal policy within a logarithmic factor, and (2) it determines the specific distribution of the runtime for different problem instances, and (3) no policy can be more than a constant factor faster. Under Luby's policy, the solver restarts as soon as some number of conflicts has occurred. In TiniSAT, the number is given by 512 multiplied by the following sequence: 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8… Although the solver forgets its decisions, it keeps the clauses it has learned.

Our Modifications

Ideally, every clause processed during the unit propagation algorithm would be processed independently of the watched literals, and in an order that takes into account our preference for short clauses that share literals. This could also be used to encourage a contradiction (if any) to be found more quickly.

```
def clause::unwatch(lit)
  otherWatch = otherWatch(lit)

  //i this clause's the index in lit's watchlist
  watchList(lit)[i] = watchList(lit).back()
  watchList(lit).pop_back()

  // clauses are stored as an array of literals
  // the first two literals in a clause are the
  // watched literals.
  swap out lit from watch position




def propagate(lit)
  lit.value = true


  stack.push(lit)

  while stack not empty:

    lit = -stack.pop()
    // process watch list for lit
    for clause in wachList(lit)
      // pick the second watch in the clause
      otherWatch = clause.otherWatch(lit)

      // clause subsumed
      if set(otherWatch) continue

      // find other watch
      newWatch = null
      for literal in clause.notWatched()
      if literal.value != false
        newWatch = literal

      if newWatch // suitable watch found
        clause.watch(newWatch)
        clause.unwatch(lit)
      else if free(otherWatch) // implication
        otherWatch.value = true
        otherWatch.antecedent = clause
        stack.push(otherWatch);
      else if otherWatch.value == false
        // contradiction

        // begins clause learning algorithm
        // with the clause that became empty
        learnClause(clause)
        return
```

Figure 2. Pesudocode for TiniSAT's unit propagation subroutine.

```
 1  def enqueueWatchList(lit, priorityQueue)
 2    for i = 0 to wachList(lit).size()
 3      watchedClause c
 4      c.lit = lit
 5      c.index = i
 6      // find the score by summing scores of
 7      // literals contained in this clause
 8      for literal in watchlist(lit)[i]
 9        c.score += literal.score
10      // reduce score by 2^20 * size of clause
11      // to prioritize smaller clauses
12      c.score -= size(watchList(lit)[i]) << 20;
13      priorityQueue.push(c)
14    return
15
16  def propagate(lit)
17    lit.value = true
18    priorityQueue.empty()
19
20    enqueueWatchList(lit, priorityQueue)
21
22    while priorityQueue not empty:
23      c = priorityQueue.pop()
24      lit = c.lit
25      clause = watchList(lit)[c.index]
26
27      // pick the second watch in the clause
28      otherWatch = clause.otherWatch(lit)
29
30      // clause subsumed
31      if set(otherWatch) continue
32
33      // find other watch
34      newWatch = null
35      for literal in clause.notWatched()
36      if literal.value != false
37        newWatch = literal
38
39      if newWatch // suitable watch found
40        clause.watch(newWatch)
41        clause.unwatch(lit)
42      else if free(otherWatch) // implication
43        otherWatch.value = true
44        otherWatch.antecedent = clause
45        enqueueWatchList(otherWatch,priorityQueue)
46      else if otherWatch.value == false
47        // contradiction
48
49        // begins clause learning algorithm
50        // with the clause that became empty
51        learnClause(clause)
52        return
```

Figure 3. Pesudocode for the modification.

Figure 1 is the pseudocode for TiniSAT's unit propagation subroutine. Figure 2 is the pseudo-code of our approach.

Recall that when running the unit propagation algorithm, SAT solvers typically process the watch lists of implied literals completely before moving on to the watch list of another implied literal. In contrast, our approach involves using a priority queue of watched clauses.

We wish to process small clauses as quickly as possible so that the antecedents of derived literals are as small as possible. We also wish to encourage merge resolution to take place during the derivation of the conflict clause. Since we cannot tell a priori which literals the running derivation of the conflict clause will contain, we attempt to guess which literals it is likely to contain, and encourage those literals to appear in multiple antecedents.

To that end, clauses are ranked first on their size, and second on the sum of the scores of their contained literals, where a literal's score is the number of clauses that contain the literal.

We believe that if clauses are picked that contain literals that are common in all clauses, those literals will also be common in clauses that are used to derive the conflict clause. Also, by simply biasing probabilities with which literals are chosen, there will be literals (those with high probability) that will be likely to appear multiple times in the derivation of a conflict clause.

Every time a clause becomes unit, we add each clause in the appropriate watch list to the priority queue. Clauses on the priority queue are represented by a structure that contains the implied literal that is being watched, the position of the clause in the literal's watch list, and the clause's score. The initial literal's watch list is originally dumped into the priority queue before the main loop.

The main loop consists of popping the most promising watched clause from the priority queue, and processing that clause in pretty much the traditional way.

Complications

One complication involves removing a clause from a literal's watch list when another suitable watch has been found. The above method requires an index into the watch list of the watched variable. This can be made available by using an index rather than a pointer when defining a `watchedClause` (line 5); however, if we were to use the simple unwatching function above, the index of the clause in the back would be changed. Because it is in a priority queue, there is no way to find it and correct the error. Instead, we simply store `null` at the location in the unwatched variable's watch list, and store the variable in a purge list for later purging (not shown).

Also, while using a priority queue is simple, it is prohibitively expensive. Fortunately, the data we are ordering has some special structure. We know that clauses of lower size must be ejected by the queue first. We can therefore use an array of queues each of which only stores clauses of a specific size. For example, we may have an array of five priority queues. `queue[0]` stores clauses of size 2, `queue[1]` stores clauses of size 3, and so on until `queue[4]`, which stores all larger clauses. We can keep track of the smallest clause in the queue, and eject from the appropriate queue first. Early experiments showed that a maximum in efficiency is around 4 queues, and that this technique results in a greater than 40% speedup.

Since TiniSAT's clause learning requires the stack order for implied literals, we must keep track of this as well.

Experimental Results

	Decisions	Conflicts	Restarts	Merges	Size of CC	Time
TiniSAT	426021.5	59172.8	38.2	3.95	33.4	69.41
TiniSAT'	356775.7	43619.2	30.2	4.00	24.8	151.22
ratio	**0.83**	**0.73**	**0.79**	**1.01**	**0.74**	2.17

Figure 3. Average results for the 93 problems solved by both solvers

TiniSAT was run alongside our modification on 168 SAT problems from SAT Race 2006, and the industrial category of the 2005 SAT competition. Of these, TiniSAT solved 106, and our modification solved 96. There was a 20 minute time limit per instance. Although our method solved ten fewer problems than unmodified TiniSAT, the results are promising.

Processing clauses in an order that encourages small clauses to participate in the derivation of a conflict clause seems to drastically decrease the size of the conflict clauses eventually learned (to 74%). This in turn cuts down the number of decisions the solver has to make before arriving at a solution. Since merge resolutions are more prevalent early in the decision sequence and our solver doesn't make as many decisions, merge resolution has been encouraged even though it would appear that the average number of merge resolutions per conflict is somewhat constant.

The overhead involved in this implementation of the modification seems to be the reason for the large reduction of speed, but much of the overhead in our implementation is unnecessary. It should be possible to score a clause when it is created instead of on the fly, and this would lead to a substantial decrease in overhead. However, the data structures that TiniSAT uses to store clauses would make this optimization cumbersome.

TiniSAT also benefits from implication lists for binary clauses[10]. This feature was turned off in our modification to reduce the complexity of the code, but there is no reason conceptually that this optimization is incompatible with our modifications. Furthermore, this optimization seems to drastically improve the speed of TiniSAT.

There are also many applications in which speed is not the primary concern. For example, DPLL-like algorithms are used in the area of knowledge compilation, and in this area the trace of the algorithm is saved. Even though the time to complete may be longer, the size of the resulting trace is shorter, which could be important.

Because our modification does not make as many decisions, and the average size of the generated conflict clause is small, our modification also uses significantly less memory than the original TiniSAT. This can be important in situations when memory is a bottleneck in solving a problem.

Conclusion

We have introduced a new method for performing unit resolution in SAT solvers based on the two literal watch scheme that processes clauses independently. We have shown one way that this new freedom can be taken advantage of by introducing a natural heuristic to accompany this method, and empirically shown that the heuristic is effective at substantially reducing the average

size of the learned conflict clause when a contradiction is reached.

Future Work

In [5], Pipatsrisawat et al. suggest learning bi-asserting clauses, i.e., those that have two variables at the assertion level instead of only one. In order to make this worthwhile, it is necessary that at least one step in the resolution process used to derive the conflict clause be a merge resolution. Since our heuristic encourages merge resolution, it may be even more powerful when combined with this technique, however TiniSAT doesn't yet learn such clauses.

Having a strict ordering on clauses might not be necessary, so it might be possible to get similar benefits with less overhead by keeping clauses in some loose order.

It might not be necessary to order the clauses at all. One might be able to devise a way to simply update the antecedent of a literal with a better clause. The naïve way of going about this, however, often results in the implication graph containing cycles, which makes traditional conflict clause learning impossible.

Our modifications to the unit propagation algorithm add a significant amount of overhead. But as unit propagation is run after every decision, and most decisions do not result in a contradiction, it may be prudent to run our version of unit propagation only when a conflict has been discovered using a version of the traditional algorithm modified for speed rather than to support clause learning.

Prioritizing clauses as we do in this paper may lead to repeatedly using the same clauses to derive subsequent conflict clauses. However, it should be possible to count the number of times a clause is used in the derivation of the conflict clause, and use this information in the clause-ordering heuristic.

Many modern SAT solvers, TiniSAT excluded, delete learned clauses when they cease being useful to save processing time and space. They use a heuristic that tracks how often the clause is used to determine which clauses to delete. This information might be valuable in crafting stronger clause-ordering heuristics.

Acknowledgments

Knot Pipatsrisawat – for several hours of explaining the details of unit resolution and clause learning.
Dawn Chen – for support, help debugging the code, and proofreading.

References

[1] Marques Silva, J.P. and K.A. Sakallah. Conflict analysis in search algorithms for satisfiability. in *Proceedings of the Eighth IEEE International Conference on Tools with Artificial Intelligence.* 1996.

[2] Huang, J. A case for simple Sat solvers. Lecture notes in computer science, 2007 – Springer.

[3] SAT Race 2006. http://fmv.jku.at/sat-race-2006/

[4] Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. *In 39th Design Automation Conference.*

[5] Pipatsrisawat, K.; Darwiche A. 2008 A New Clause Learning Scheme for Efficient Unsatisfiability Proofs

[6] Eugene Goldberg, Yakov Novikov, BerkMin: A fast and robust Sat-solver, Discrete Applied Mathematics, Volume 155, Issue 12, SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing, 15 June 2007, Pages 1549-1561

[7] Luby, M.; Sinclair, A.; Zuckerman, D., "Optimal speedup of Las Vegas algorithms," Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the , vol., no., pp.128-133, 7-9 Jun 1993

[8] Huang, J. The effect of restarts on the efficiency of clause learning. In AAAI-06 Workshop on Learning for Search (2006).

[9] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In Proceedings of the International Conference on Computer Aided Design (ICCAD), pages 279 – 285, 2001.

[10] Kibria, R. MidiSAT - An extension of MiniSAT.