JASS: Jolly Awesome Satisfiability Solver

Bart Grundeken (1096478)
bart.grundeken@gmail.com

Michael Henrichs (9327103)
mikehenrichs@gmail.com

Benny van Reeven (1100319)
bvanreeven@gmail.com

Robin van der Rijst (1100335)
rvdrijst@gmail.com

March 14, 2007

Abstract

Herein we describe our experiences constructing and testing a hybrid satisfiability solver (JASS), i.e., one that combines both the lookahead and conflict-driven architectures for solving satisfiability problems. We present our approach to creating the solver, give an insight in the implementation process and compare the execution of the hybrid solver on some problems to the execution of a pure lookahead solver on the same problems. In the end we determine whether or not combining the algorithms provides a useful optimization, if there is an increase in performance at all.

1 Introduction

Satisfiability solvers have traditionally been divided into two categories: conflict-driven solvers and lookahead solvers. Our project was to develop a solver that combined the best of both worlds: a "hybrid" solver that uses versions of both algorithms in unison. In this paper we discuss how we created this program, JASS (Jolly Awesome Satisfiability Solver): what steps we took and what obstacles we had to overcome; as well as the results of running our solver on a number of satisfiability problems and, more importantly, how these results compare to those of a "traditional" satisfiability solver. This paper is structured as follows: first, we present the process of creating the solver, in the next section. Then, we discuss the results of the solver and compare them to the results of a pure lookahead solver. Finally, we draw our conclusions.

2 Creating the Solver

In this section we discuss how we created our solver and the goals we set out to achieve. Our general approach is presented first, followed by how we created our own lookahead solver. Then we elaborate upon how we extended that solver with conflict analysis.

2.1 General Approach and Objectives

For all of us, this was more or less our first contact with satisfiability solvers, although we were familiar with the satisfiability problem, namely SAT-3, and

Algorithm 1 ITERATIVEUNITPROPAGATION($\mathcal{F}$)

```
while $\mathcal{F}$ does not contain an empty clause and unit clause $y$ exists do
    satisfy $y$ and simplify $\mathcal{F}$
end while
return $\mathcal{F}$
```

the fact that the problem is NP-complete and thus needs heuristics to be solved in a reasonable amount of time. We will not further elaborate on the theory behind SAT-3 and other satisfiability problems, however, we refer the reader to [6] for a more detailed discussion of NP-completeness and satisfiability problems. We studied a number of relevant papers and the source code of other solvers (like March [3] and MiniSAT [1]) to gain better understanding of existing satisfiability solvers and their inner workings, as well as the various algorithms the solving process is based upon.

Our initial approach to creating a hybrid solver was to create our own lookahead solver in C. This was also preferable since we needed to create a lookahead solver for testing purposes, in order to compare JASS to a normal lookahead solver that was in any other way as similar as possible to JASS. After we created this lookahead solver and had convinced ourselves of its correctness (and reasonable efficiency) we could extend this solver with conflict analysis. Once this solver was also tested and proved to work correctly, we could start the performance tests and, more precisely, compare the performance of the solvers.

2.2 Creating the Lookahead Solver

We created our lookahead solver more or less from scratch. The very first version was an implementation of the algorithms presented in [3]. We used feedback from our meetings with the teachers to optimize our solver in various ways. A functional lookahead solver was produced rather quickly, however, making that solver more efficient took quite some additional time. After we had created a reasonably efficient solver, we started to implement the conflict analysis part, as presented in the next section. Still, even then we found optimizations that were not restricted to just the hybrid solver. Issues regarding the creation of the lookahead solver were mostly issues of efficiency, not necessarily of correctness.

We had some problems with memory leaks, which caused major memory consumption for larger problems, resulting in out-of-memory errors. We also had to make modifications to our data structures now and then. Early on, we mostly had to modify these structures because these did not function correctly, while later on the modifications were mostly made to improve performance and efficiency.

During the development of the lookahead solver we learned a lot about the general structure of a satisfiability solver. We also learned a few tricks regarding the C language to make things faster. In the end, we were able to create a correct lookahead solver.

2.3 Implementation of the Lookahead Solver

We now present the algorithms on which our lookahead solver is based. Algorithm 1 shows the binary constraint propagation (BCP) algorithm, called

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

```
$\mathcal{F} \leftarrow$ IterativeUnitPropagation($\mathcal{F}$)
if $\mathcal{F} = \varnothing$ then
    return "satisfiable"
else if $\mathcal{F}$ contains empty clause then
    return "unsatisfiable"
end if
$x \leftarrow$ GetBranchVariable()
if DPLL($\mathcal{F} \cup \{x\}$) = "satisfiable" then
    return "satisfiable"
else
    return DPLL($\mathcal{F} \cup \{\neg x\}$)
end if
```

Algorithm 3 Lookahead( )

```
repeat
    for all free variables in $\mathcal{F}$ do
        $\mathcal{F}' \leftarrow$ IterativeUnitPropagation($\mathcal{F} \cup \{x_i\}$)
        $\mathcal{F}'' \leftarrow$ IterativeUnitPropagation($\mathcal{F} \cup \{\neg x_i\}$)
        if $\mathcal{F}'$ contains empty clause and $\mathcal{F}''$ contains empty clause then
            return "unsatisfiable"
        else if $\mathcal{F}'$ contains empty clause then
            $\mathcal{F} \leftarrow \mathcal{F}''$
        else if $\mathcal{F}''$ contains empty clause then
            $\mathcal{F} \leftarrow \mathcal{F}'$
        else
            H($x_i$) $\leftarrow$ MixDiff(Diff($\mathcal{F}$, $\mathcal{F}'$), Diff($\mathcal{F}$, $\mathcal{F}''$))
        end if
    end for
until NoNewFailedLiteralsDetected()
return $x_i$ with highest H($x_i$)
```

IterativeUnitPropagation, which is taken from [3] and used in all versions of our solver. Later extensions to this algorithm include the determining of reasons for literals (for later on calculating conflict clauses from them) and the tracking of clauses that provoke conflicts.

Our lookahead solver is based on the DPLL procedure shown in algorithm 2, where the GetBranchVariable procedure is implemented as the Lookahead procedure shown in algorithm 3. Both algorithms are taken from [3] and implemented as closely as possible in our lookahead solver.

Figure 1 shows the structure of the lookahead solver, as well as the conflict extension. The boxes represent modules, while the arrows represent the dependencies between them. In our case, a module consists of a C file and a header file, as is common in the C paradigm. Our solver consists of the following modules:

`global` Contains global constants, variables and data structures which are used throughout the entire solver and is used by all other modules.

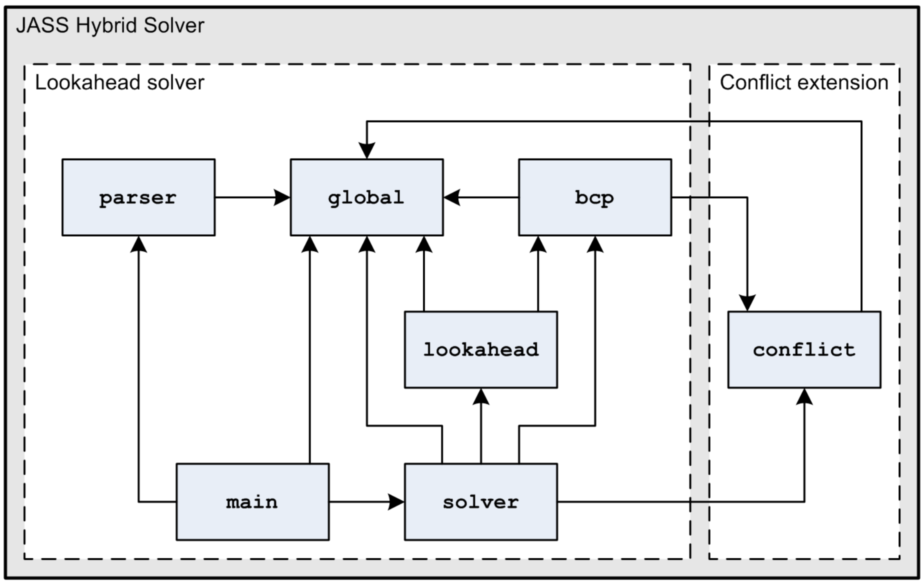

Figure 1: *Structure of the solver*

`main` Starting point of the program. Responsible for initiating the parsing and solving functions, as well as gathering statistical data.

`parser` Responsible for parsing the DIMACS CNF input file into the global data structures.

`solver` Contains our implementation of the DPLL algorithm.

`lookahead` Contains our implementation of the LOOKAHEAD algorithm.

`bcp` Contains our implementation of the IUP algorithm.

`conflict` Contains our implementation of the conflict-driven extension.

Apart from these modules, there are a number of utility modules representing the different data structures used in our solver. Next, we will describe the solving process using the data structures summarized in Table 1.

Level Tracking In each phase of the process we keep track of our progress by incrementing and decrementing a global variable called `current_level`. This variable is initially set to 1, and is incremented after a chosen branch literal is assigned and all unit clauses have been propagated. Also, a level boundary is pushed onto the `assignment_stack` in order to separate the variable assignments made at different levels. When a conflict occurs, all variables that were assigned at the current level are unassigned, the `current_level` variable is decreased by one and a level boundary is removed from the `assignment_stack`.

Initialization and Parsing When the lookahead solver is executed, the main module first initializes all global data structures, as presented in Table 1, by calling the corresponding initialization functions. It then calls the parser, which

Variable	**Contents**
`all_clauses`	A list containing the clauses with their literals.
`initial_clause_lengths`	A list containing the initial length of each clause.
`current_clause_lengths`	A list containing the length of each clause during the solving process.
`occurrences`	A list containing for each literal the indexes of the clauses in which it occurs.
`unit_clauses`	A stack of unit clauses for the current stage in the solving process.
`assignment_stack`	A stack of variable assignments (true or false) separated by level boundaries.

Table 1: *Global data structures kept by the solver*

reads the specified DIMACS CNF file and fills the data structures accordingly. When the input file is parsed, the main module calls the solver, which subsequently starts the solving process.

Solving The solver first calls the lookahead module, which performs a full iterative lookahead to find the most efficient next branch literal. The solver then assigns this literal and calls the bcp module, which performs the IUP algorithm, after which the solver increases the `current_level` variable. However, when a conflict is encountered during the lookahead, the solver backtracks to the previous level and negates the current branch literal, the result of which is that the solver travels down the other branch. This process is repeated until either no free variables are left, which indicates that the problem is satisfiable and the current assignment is a solution to the problem, or the search space has been exhausted and no solution has been found.

Looking Ahead The lookahead module is responsible for testing all free variables and, if more than one free variable is available, selecting the best variable for branching by means of a DIFF heuristic, as described in [3]. The heuristic we use is based on the number of binary clauses generated by assigning a free variable.

Assigning and Propagating The bcp module is responsible for assigning variables and propagating the resulting unit clauses. The assigning of a variable is done by pushing the literal onto the `assignment_stack` and updating the clause lengths of all clauses in which the literal occurs. For false literals, 1 is subtracted from the length of the clause, while for true literals, the number of variables + 1 is subtracted. As a result, a clause with length greater than zero indicates a clause which is not yet satisfied, while a clause length smaller than zero indicates a satisfied clause. A clause of length zero indicates a conflict.

During the updating of the clause lengths, each clause of length one is pushed onto the `unit_clauses` stack. After the assignment is completed (and no conflicts are encountered), the unit propagation algorithm pops each unit clause from the stack and satisfies it by assigning the free variable. When no more unit clauses are left, control is returned to the caller.

Algorithm 4 ConflictDrivenDPLL($\mathcal{F}$)

```
while TRUE do
    $l_i \leftarrow$ GetBranchLiteral()
    if an $l_i$ is selected then
        $\mathcal{F} \leftarrow$ IterativeUnitPropagation($\mathcal{F} \cup \{l_i\}$)
        while $\mathcal{F}$ contains empty clause do
            $blevel \leftarrow$ AnalyzeConflicts()
            if $blevel = 0$ then
                return "unsatisfiable"
            else
                BackTrack($blevel$)
                $\mathcal{F} \leftarrow$ IterativeUnitPropagation($\mathcal{F}$)
            end if
        end while
    else
        return "satisfiable"
    end if
end while
```

2.4 Extending the Solver with Conflict Analysis

Extending our lookahead solver with conflict analysis was the daunting task that now laid ahead of us. Our first approach was to study the conflict analysis procedure as found in MiniSAT, and somehow merge that into our own solver. However, MiniSAT's architecture was so different from ours that we would have to rewrite our entire solver in order to fit MiniSAT's conflict analysis procedures into our solver. Therefore, the idea of using MiniSAT for conflict analysis was abandoned, and we decided to write our own conflict analysis procedure, based on discussions with the teachers and on papers such as [7] and [8], which describe conflict-driven learning, among others. We have also used papers on two well-known conflict-driven satisfiability solvers, namely Chaff [4] and BerkMin [2].

After some iterations of feedback from the teachers and revising our conflict analysis procedures, we eventually reached a hybrid SAT solver that did both lookahead and conflict analysis correctly.

2.5 Implementation of the Conflict Extension

The ConflictDrivenDPLL procedure, which is shown in algorithm 4 and is also taken from [3], forms the basis for the conflict analysis in our hybrid solver. However, since our original solver has a lookahead architecture, we could not simply implement algorithm 4 as is. Therefore, we have modified the algorithm to make use of the Lookahead procedure, resulting in the HybridDPLL procedure, which is shown in algorithm 5.

Modifications with respect to ConflictDrivenDPLL are the following. First of all, since the Lookahead procedure assigns and propagates all free variables to determine the next branch literal, conflicts (both assigning x_i and assigning $\neg x_i$ result in an empty clause) are encountered there instead of in the DPLL algorithm. Therefore, in line 3 of algorithm 5, an extra **while** loop is added in order to backjump as long as the Lookahead encounters a con-

Algorithm 5 HybridDPLL($\mathcal{F}$)

```
while TRUE do
    $l_i \leftarrow$ Lookahead()
    while ConflictEncountered() do
        while $\mathcal{F}$ contains empty clause do
            $blevel \leftarrow$ AnalyzeConflicts()
            if $blevel = 0$ then
                return "unsatisfiable"
            else
                BackTrack($blevel$)
                $\mathcal{F} \leftarrow$ IterativeUnitPropagation($\mathcal{F}$)
            end if
        end while
        $l_i \leftarrow$ Lookahead()
    end while
    if an $l_i$ is selected then
        $\mathcal{F} \leftarrow$ IterativeUnitPropagation($\mathcal{F} \cup \{l_i\}$)
    else
        return "satisfiable"
    end if
end while
```

Variable	**Contents**
`reasons`	A list containing, for each variable, a list of branch literals which implied the setting of this variable.
`left_branch_levels`	A list containing, for each branch literal, the level on which it was set.

Table 2: *Global data structures used in the conflict extension*

flict. Furthermore, the non-chronological backtracking, which is represented by lines 4-12, is only needed after the Lookahead procedure, and not after the actual assigning and propagating of the chosen branch literal, as it is already verified within the Lookahead procedure that this can be done without conflict.

We now move on to the implementation of the modifications which we implemented to enable a conflict-driven architecture. Table 2 shows additional data structures used in the conflict extension.

Reason Tracking When conflict analysis is enabled, we keep track of branch literals that have caused a certain variable to be set. This is called the *reason* of the variable. From these reasons we can later construct our conflict clauses in the function `analyse_conflict` in the `conflict` module.

During BCP, all unit clauses are satisfied by assigning the free literal. When this is done, the union of the reasons of the other literals in the unit clause is set as the reason for this literal, the result of which is stored in the `reasons` list.

For performance reasons, reason tracking is disabled during lookahead. However, when a conflict is encountered during the assignment of a literal, we redo the BCP with reason tracking enabled in order to find the reason for the conflict. The literal is then forced into its negated state, and its reason is set to the reason of the conflict. When forcing the literal also fails, a conflict is encountered on the current branch, and the solver needs to analyze this conflict in order to continue the solving process. This analysis yields a reason for the conflict, which is used to determine the most recent left branch literal that is responsible for the conflict. Finally, we force this left branch literal to alleviate the conflict, and assign the conflict reason minus the literal itself as its reason. This process is repeated until no more conflicts are found.

Analyzing Conflicts Conflict analysis takes place when the assignment of a variable fails. The index of the clause that causes the conflict is stored into the global variable `conflicting_clause_index`. The function `analyse_conflict` uses this index to retrieve the conflicting clause, and calculates the conflict reason from this clause. It does this by taking the union of the reasons for each of the literals in this clause. The resulting reason is returned to the reason tracking process described above.

Non-Chronological Backtracking When a conflict occurs on the current branch, the solver needs to backtrack to a previous node in the search tree, in order to continue investigating another branch. A pure lookahead solver backtracks one level, while in a conflict-driven solver, a more elaborate scheme is used, known as non-chronological backtracking. This scheme uses the reason of a conflict, which is acquired from the function `analyse_conflict`, to determine a backtrack level. This level is always one level below the highest left branch literal in the reason, because that is the level where the conflict clause becomes unit. The solver then jumps back to this level and continues its search for a solution. If the determination of the backtrack level yields zero, the problem is unsatisfiable.

Learning Conflict Clauses Our implementation keeps a rotating array of learned conflict clauses, of which the maximum size can be specified using the global variable `MAX_LEARNT_CLAUSES`. When this maximum is set to zero, no clauses are learned, but the non-chronological backtracking is still applied. When the maximum number of learned clauses is greater than zero, the function `analyse_conflict` adds the reason of a conflict as a conflict clause to the library of clauses. In principle, these learned clauses should speed up the solving process because the problem becomes more constrained and thus the search tree is pruned.

We now move on to the results we obtained from benchmarking our solver and the comparison between the two versions of our solver: the pure lookahead and the hybrid solver.

3 Experimental Results

We tested our solvers on a variety of problem families, many of which can be downloaded from the Satisfiability Library [5]. Aside from the pure lookahead ("Lookahead") solver, we ran our hybrid solver in two ways: without learning additional clauses ("Hybrid (No Learning)") and with a maximum of 512 learned clauses ("Hybrid"). We have measured both the elapsed time ("Time") and the number of decision tree nodes ("# Nodes") that the solvers needed to solve the problems.

Problem family	Instances	Lookahead		Hybrid (No Learning)		Hybrid	
		Time	# Nodes	Time	# Nodes	Time	# Nodes
uf20	1000	0.000445	3	0.000415	3	0.000366	3
uf100	1000	0.0163	22	0.0158	22	0.0157	22
uf250	100	4.61	990	4.65	990	6.45	982
uuf50	1000	0.002436	4	0.002617	4	0.00259	4
uuf100	1000	0.0348	23	0.0351	23	0.038	22
uuf250	100	9.47	3453	9.54	3453	19.74	3426
rti100	500	0.015	22	0.014	22	0.016	22
bms100	500	0.053	54	0.054	54	0.073	54
hanoi	1	1024	50748	1048	50421	752	29427
barrel	3	0.207	6	0.191	6	0.16	6
longmult	9	258	1292	218	1280	289	1274
queuinvar	8	171	251042	255	251210	345	251160
qgbench	1	0.584	32	0.563	32	0.657	31
pyhala-braun SAT	4	792	2183	771	2183	809	2183
pyhala-braun UNSAT	3	1297	2603	1095	2603	1188	2590
ezfact	20	27.2	995	27.9	988	23.9	951
stanion	3	476	369269	497	369741	918	341776

Table 3: *Experimental Results*

Table 3 shows that the Lookahead solver often outperforms the Hybrid solver, when it comes to pure speed. When it comes to the number of nodes, the Hybrid sometimes uses less nodes than the Lookahead or Hybrid (No Learning) solvers. This would support the apparent conclusion that the speed loss is due to the overhead of the bookkeeping of learned clauses. We believe this can be mitigated with future work, however.

Some of the results defer from the above analysis, though. While the uf20, uf100 and uuf50 problem families may be too small to have any real impact, results in those areas do show that the Hybrid solver can be faster. Again, increased overhead in larger variants of those problem families make the Hybrid solver lag behind. This effect is apparent in the majority of the tested problem families, with the stanion family being apparently totally unsuited for the Hybrid solver, as even the number of nodes increases with regard to the Lookahead solver.

Then there is the class of problem families where the Hybrid (No Learning) solver is faster than the Lookahead solver, yet the Hybrid is slower. We see this effect with rti100, longmult, qgbench and pyhala-braun-SAT. While rti100 may be once again too small to draw real conclusions from, the other families show that the Hybrid (No Learning) solver's non-chronological backtracking is apparently suited for those problem families, creating an increase in performance. In longmult the number of nodes even decreases. Once again, since performance seems to degenerate as we move from the Hybrid with no learned clauses to that with a maximum of 512, we can only conclude that the additional overhead of keeping track of all the learned clauses has a negative impact. The apparent conclusion is that there is a break-even point in maximum learned clauses where the overhead of learned clauses compensates for the speed gain due to non-chronological backtracking. With pyhala-braun-UNSAT, this break-even point lies higher than 512 learned clauses, as we see the time increase between the Hybrid (No Learning) and Hybrid solvers, yet the Hybrid is still faster than the Lookahead solver.

Finally, we come to the group of problem families where the Hybrid outperforms its peers. We see this in the hanoi, barrel and ezfact families. The table shows no difference in the number of nodes with the barrel family; yet the ezfact and, most noteworthy, the hanoi family shows a marked decrease in node count. Granted, we tested the hanoi family only with the hanoi4 instance, but the structure seems to suit the Hybrid solver. Apparently it is so that the Hybrid has such advantages from its conflict analysis, that result in such decreases in node counts, that the overhead from bookkeeping is mitigated. Here, we have yet to test the Hybrid performance against that of a pure conflict-based solver, however.

Concluding this section, we can see that a marked increase in node count has a positive effect on overall performance, yet the bookkeeping of conflict clauses may mitigate this effect. Future work is needed to improve this bookkeeping, yet may, arguably, result in increased memory usage. We have presented a number of noteworthy results where the advantages of the Hybrid are clear. Another interesting point is that a solver that just performs lookahead yet uses non-chronological backtracking, beats both the Lookahead and Hybrid solvers in terms of performance.

4 Conclusion

In this paper we have presented our hybrid satisfiability solver JASS, which combines both the lookahead and conflict-driven architectures. We have also presented our experimental results, from which we will draw conclusions below. Having created, tested and benchmarked our solver in various phases, we have seen both the difficulty in creating such a program as well as the many opportunities there always are to improve performance.

The main conclusion we can draw is that the hybrid solver leads to lower node counts, and thus (in principle) to faster problem solving. However, improved bookkeeping of conflict clauses is necessary, such that the overhead (which cannot be fully avoided) is such, that it no longer overshadows the performance gain due to the decrease in node count.

Future work would include such improvements. There are also more difficult improvements that may be made; for instance, some way to detect the kind of problem family we are dealing with so the solver can adapt its maximum learned clauses as needed. This improvement might very well be worthy of a entirely separate research project however, and we are even then left with the question if such a pre-selection algorithm will itself not take so much time as to override its effects on the solver.

References

[1] N. Één and N. Sörensson, *An extensible SAT solver.* 6th International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919, p 502-518, 2003.

[2] E. Goldberg and Y. Novikov, *BerkMin: A fast and robust SAT-solver.* In Design, Automation, and Test in Europe (DATE 02), pages 142149, March 2002.

[3] M.J.H. Heule, *March: Towards A Lookahead Sat Solver For General Purposes.* M.Sc. Thesis, Delft University of Technology, Delft, The Netherlands, March 2003.

[4] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an Efficient SAT Solver.* In Proceeding of the 38th Design Automation Conference (DAC01), 2001.

[5] SATLIB – The Satisfiability Library. `http://www.satlib.org`

[6] M. Sipser, *Introduction to The Theory of Computation*, Second Edition. Thomson, 2005.

[7] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, *Efficient Conflict Driven Learning in a Boolean Satisfiability Solver.* Presented at International Conference on Computer Aided Design (ICCAD), San Jose, CA, 2001.

[8] L. Zhang and S. Malik, *The Quest for Efficient Boolean Satisfiability Solvers.* In Ed Brinksma and Kim Guldstrand Larsen, editors, Proc. Computer Aided Verification, volume 2404 of LNCS, pages 1736, Copenhagen, Denmark, 2002.