2clseq and about it's scalability

Topi Pohjolainen
tpohjola@cc.hut.fi

26.03.2003

Abstract

Testing for satisfiability in proportional logic, abbreviated as SAT, represents a problem belonging to the NP-complete class. However, solutions to the problem are needed in increasing amount by various applications. Thus both the need for efficient solutions and the challenge imposed by the hardness of the problem have triggered plenty of research work on the subject. Recent attempts to realize the theories evolved have spawned a variety of tools for this purpose. More recent problem has been the analysis of the efficiency of the tools. The difficulty of analytic approach has shifted the emphasis on empirical methods. Even these tend to be problematic. All the characteristics contributing to the hardness of solving particular types of problems are not known. In this paper we experiment with a SAT-solver called **2clseq**. We have concentrated on a few chosen subclasses of the generel satisfiability problem in order to measure the behaviour of the tool against smaller set of problem characteristics. The results encourage this type of approach.

1 Introduction

The satisfiability problem of proportional logic has been studied and researched for quite some time. Recently there has been a fair amount of effort trying to efficiently realize the ideas developed. The techniques can be roughly divided into two classes, complete methods based on DPLL [1] and incomplete methods based on local search strategies. Both categories tend to have certain problem classes in which they perform well and others where they may not be able to find solution at all. This imposes quite a few problems in the analysis of the effiency of the tools not to mention when trying to make comparisions. One of the reasons for this is the lack of understanding of the problem characteristics. In order to overcome this at least partially, we have restricted our analysis on a few subclasses of the general problem. By this we hope to fix some of the characteristics unknown in order to examine effects of others that are known.

As mentioned before there are quite a few tools, both commercial and freely distributed, available for solving SAT-instances. One of the recent ones is of particular interest because despite of the relatively small development resources, it was

quite successful in SAT Competition 2002 [11]. This tool called **2clseq** belongs to the family of complete SAT-solvers based on DPLL algorithm. It combines some already experimented techniques with unique realization of method called hyper-resolution.
We experimented 2clseq in three different problem subclasses, random 3-SAT, factorisation of an integer and bounded model checking of IBM hardware test cases. In the two former the aim was to examine the effect of the size of the problem instance to the running time of the tool. In the third subclass 2clseq's running time was measured for each instance.
The paper is organized as follows. First the main techniques used in 2clseq are explained briefly. Then the methodology used in the experiments is outlined followed by description of the test setups for each of the subclasses. In the last section before the conclusions, we introduce the results and their analysis.

2 2clseq and DPLL

The DPLL (Davis, Putnam, Logemann, Loveland) algorithm[1] is widely used as the basis among complete SAT-solvers. It is a simple recursive method that takes as input the problem, denoted as set of clauses S, and a set of truth assignments, denoted by A, to some or all variables found in S. See Table 1. The DPLL routine first tries to simplify the problem, i.e, reduce the amount of clauses and their sizes in S.
Let S' denote the set of clauses gotten from S and A' the set of truth assignments gotten from A by simplification. If S' becomes empty it means that every clause in S was satisfied by the truth assignments in A and A is said to be the model for the set of clauses S. On the other hand if A' contains both truth assignments (true,false) for some variable in S, it means a conflict had been found. In this case the routine cannot do anything else than to return and to report it.
If neither model nor conflict is found, DPLL tries to find a variable in S' that isn't assigned by A' yet. Then both of the truth values for the variable are considered in turn by recurring DPLL for S' and A' amended with the corresponding truth assignment.
The basic DPLL uses technique called unit propagation to perform the simplification step. This means simply performing unit resolutions on the set of clauses S until closure.
In 2clseq this simplification method is enhanced with a form of hyper- resolution [2] and with a technique called equivalence reduction [2]. A flavour of clause learning [2, 3] with intelligent backtracking [9] is further added to the algorithm.

3 Hyper-resolution in 2clseq

Hyper-resolution in general is about resolving more than two clauses the same time. The tool 2clseq uses a special case where the clauses taking part in the resolution are restricted by length, i.e, by the number of literals in them.

```
DPLL(S, A):
(S', A') ← simplify(S,A)
if A' contains a conflict
  return FALSE
else if S' = {}
  return TRUE
var ← select variable not in A'
if DPLL(S', A' ⋃ {var})
  return TRUE
else return DPLL(S', A' ⋃ {¬ var} )
```

Table 1: DPLL (Davis, Putnam, Logemann, Loveland) algorithm. S denotes set of clauses and A denotes set of truth assignments for the variables occuring in S.

Specifically, 2clseq resolves one n-ary clause with n-1 binary clauses.

Theorem 1:

Given an n-ary clause of the form $C = l_1 \vee l_2 \vee ... \vee l_n$ and a set of binary clauses of the form $S = \{\neg l_i \vee h \mid 1 \leq i \leq n-1\}$, then

$$\models C \wedge S \Rightarrow l_n \vee h$$

Proof: By resolving C with each of the binary clauses in S one by one the result follows.

The fact that the hyper-resolution step produces binary clauses can be exploited further. As the set of clauses contains one binary clause more after performing hyper-resolution, it may be possible to perform it again and produce another binary clause. This can be done until closure, i.e, until there aren't such clauses available that could be resolved anymore. And between every hyper-resolution steps 2clseq can try to perform unit resolutions which in turn might produce clauses that can be used to perform more resolutions. In 2clseq this is exactly what is done, unit resolution and hyper-resolution are interleaved and performed together until closure.

Hyper-resolution contributes indirectly to variable selection, recall DPLL algorithm in Table 1, too. In 2clseq the variable selection is based on ranking the variables occuring in the problem. The ranking in turn is based on an estimate of how many new literals each variable would implicate. This means assigning a truth value for the variable, then performing unit resolutions until closure on the set of clauses and finally counting the number of variables gotten assigned. But this would require extra work as for every variable both truth values had to be considered and then the unit resolutions performed. Hyper-resolution in turn does this ranking as a side-product. Every time a new binary clause has been created by the resolution steps (unit or hyper) the counters for the literals

equality_reduction(S):
1. **if** there exists literals a and b s.t. $(\neg a \vee b), (\neg b \vee a) \in$ S **then**
2. replace every occurence of literal a in S by literal b
3. remove every clause in S containing both literals b and $\neg b$
4. remove duplicate appearences of literal b in any clause of S

Table 2: Equality reduction algorithm. S denotes set of clauses.

in the binary clause just created are increased by one. Clearly these binary clauses would implicate one unit clause each if the other literal in them would be assigned.

Hyper-resolution is shown to resolve the same set of literals as would failed literal detection [2, 8].

4 Equality reduction

This technique makes use of the possible existence of pairs of binary clauses stating equivalence between two variables.

Theorem 2:

Let S denote a set clauses and let $S|_{a \leftarrow b}$ denote the set of clauses gotten from S by replacing every occurence of literal a with b. If there exists binary clauses of the form $(\neg a \vee b)$ and $(\neg b \vee a)$ in S then

$$\models S \quad \Leftrightarrow \quad \models S|_{a \leftarrow b}$$

Proof: As for any variables a and b,

$$\models ((\neg a \vee b) \wedge (\neg b \vee a)) \Leftrightarrow \models (a \Leftrightarrow b)$$

Then for every clause $C \in S$ is equivalent with clause C' gotten from C by replacing variable a with b or vice versa.

This is exploited in 2clseq in the form of equivalence reduction algorithm outlined in Table 2. In the worst case the algorithm at least reduces the number of variables in the problem by one, but it may be capable of producing more clauses that can be in turn further reduced by another round of equivalence reduction or resolved by hyper- and unit resolution steps. Thus it makes sence to interleave equivalence reduction with hyper- and unit resolutions and keep performing them all together until closure.

5 Intelligent backtracking and clause learning

If one recalls the way the DPLL algorithm works, outlined in Table 1, it can be noted that the variables occurring in the problem (in the set of clauses S)

are given truth assignments in two distinct ways. The first is the simplification prodecure consisting of hyper- and unit resolution routines accompanied with equivalence reduction. These routines set truth assignments on variables based on logical reasoning.
The second way of variable getting assigned is by DPLL splits. This is when DPLL recurrs. See lines 7 and 9 of the DPLL algorithm in Table 1.

Definition 1:

Let the S denote a set of clauses and v a variable occurring in S. If v gets assigned, i.e, given a truth value False or True, by one of the three reduction schemes, unit resolution, hyper resolution or equivalence reduction, then variable v is said to be assigned by reduction. Let A_R denote the set of assignments made by reductions.
Example: Let $A_R = \{a, b, \neg c\}$. This is interpreted as variables a and b been assigned True and variable c False. Thus the elements of A_R are literals.

Definition 2:

Let S and v be defined as in Definition 1, but assume v is a variable selected on line 6 of the DPLL algorithm. It gets assigned just before DPLL recurrs. The variable v is said to be assigned by DPLL split. Let A_{split} denote the set of assignments made by DPLL splits. Interpretation of the elements in A_{split} is the same as for A_R.

Intelligent backtracking is based on dividing all truth assignments made by the DPLL algorithm into A_R and A_{split}. Note that these two sets exhaust all assignments made. The essential part of intelligent backtracking is to associate a subset of A_{split} with each literal in A_R and to use these subsets to avoid searching, possibly large, areas of the search space.

Definition 3:

Let S be defined as in Definition 1. Consider a state of the search where $\{a, b, c\} \in A_{split}$. Now assume that S contains clause $C = \{\neg a \vee \neg b \vee \neg c \vee d\}$. Then application of unit resolution once for each of the literals a, b, c and for the clause C would produce unit clause d. Literal d is then appended to A_R. Now the subset formed by literals a, b and c is associated with literal d. This subset is said to be the reason for the variable d getting assigned True. The reason would have been exactly same if the clause C would have been $\{\neg a \vee \neg b \vee \neg c \vee \neg d\}$, the variable d would have only been assigned False.

The case in Definition three can be seen as a special case of the more general way of recording reasons for literals. In Definition three it was assumed that $\{a, b, c\} \in A_{split}$. Now suppose one of them, say b, belongs to A_R instead. Then by the definition there exists a subset of literals R_b being the reason for literal b. Let $R_b = \{f, \neg g\}$. Now instead of subset $\{a, b, c\}$, the algorithm has to lookup

R_b and associate subset $\{a, c\} \bigcup R_b = \{a, c, f, \neg g\}$ as the reason for d.
The reasons associated with literals in A_R become useful when a conflict is encountered.

Definition 4:

Let literal $\neg l_k$ be a conflicting literal and $l_k \in A_R$ in some point of the search. Let the reason associated with l_k be $R_k = \{h_1, h_2, ..., h_m\}$. And as conflicts can only rise when making reductions, there exists a subset of A_{split} representing the reason for $\neg l_k$ too. Let this be $R'_k = \{h_{m+1}, h_{m+2}, ..., h_n\}$. Now the union of R_k and R'_k is a set truth assignments that cannot all be made simultaneously. This set, $R_k \bigcup R'_k$, can be written in clausule form as $\{\neg h_1 \vee \neg h_2 \vee ... \vee \neg h_m \vee \neg h_{m+1} \vee ... \vee \neg h_n\}$ and is said to be a clause that the search has learned.

The straightforward use of the learned clause is to backtrack in the search until the clause can be satisfied. Using the symbols in Definition 4, this means undoing assignments found in A_{split} until $(A_{split} \bigcap (R_k \bigcup R'_k)) \neq (R_k \bigcup R'_k)$. This is called backtracking intelligently.

6 Backpruning

The basic case of intelligent backtracking depicted in last section can be taken further. The learned clause can be viewed as a constraint that the search wasn't aware of before. Intelligent backtracking uses it to backtrack until it can be satisfied, but doesn't use it any further. However, the clause could be added to the set of clauses representing the problem in order to assist the search not to violate it in any future assignments. This however requires complex bookkeeping and is generally implemented in more restricted forms.
Consider the case in Definition 4. In 2clseq's case the search adds to it's bookkeeping the fact that it cannot assign $\neg l_k$ before one of the assignments in $R_k = \{h_1, h_2, ..., h_m\}$ is canceled. In other words before clause $(\neg h_1 \vee \neg h_2 \vee ... \vee \neg h_m)$ can be satisfied, variable l_k cannot be assigned False. Thus the search can prune subsections of the search space that would violate this condition.

7 Variable selection heuristic

When all simplifications possible are done, the search has to choose a variable to be split, i.e, to branch in two with the variable and it's negation.
The default heuristic in 2clseq is based on ranking of variables [2]. The ranking is based on an estimate of the number of new binary clauses the variable would implicate under unit propagation.

Definition 5:

Let p be the estimate of the number of new binary clauses a variable v would implicate under unit propagation and n be the corresponding

N	Size
1024	65kb
8192	4Mb
32768	64Mb
65536	256Mb
131072	1024Mb

Table 3: Memory requirements for some values of N (number of variables), when the word length W = 8 bit/byte

number for the variable's negation $\neg v$. Then in 2clseq the rank for the variable v is calculated as:

$$rank(v) = n + p + 1 + 1024 * n * p$$

If two or more variables score the same best score, then one of them is chosen randomly. Recall that the estimates were gotten as side-products of the reduction routines.

8 Relations between literals and binary clauses

In order to implement the reductions schemes outlined earlier, there is need for keeping record of relations between literals and binary clauses. In 2clseq this is achieved by associating each literal with the binary clauses the literal occurs in. In the initializing phase literals are ordered as they are read in. Then each of them is associated with an array that records every smaller literal the literal occurs in a binary clause with.
The required amount of memory (in number of bytes) can be estimated by:

$$Size = \lceil\frac{1}{W}\rceil + \lceil\frac{2}{W}\rceil + \lceil\frac{N-1}{W}\rceil \leq \frac{\frac{(N-1)N}{2} + N - 1}{W} = \frac{(N-1)(N+2)}{2W}$$

where W denotes the number of bits per byte and N the number of literals. And as each literal corresponds to a variable or it's negation, we have twice the number of literals then variables. By examining Table 3, it can be noted that this data structure alone limits the use of 2clseq in low memory environments in case of large problems.

9 Putting it all together

Consider the modified DPLL algorithm outlined in Table 4. The original simplification step is replaced by loop performing the reduction schemes explained in sections three and four. Conflict detection does the extra calculation and book-keeping needed for intelligent backtracking and backpruning. These data

```
DPLL_2clseq(S, A):
1.  do
2.      (S', A') ← do_hyper_resolution(S,A)
3.      (S', A') ← do_unit_resolution(S',A')
4.      (S', A') ← do_equiv_reduction(S',A')
6.      if A' contains a conflict
7.          calculate and store the reason for the conflict
8.          record backprune data for the conflicting literal
9.          return FALSE
10.     else if S' = {}
11.         return TRUE
12. while (S' ≠ S or A' ≠ A);
13. rank variables
14. var ← select variable not in A'
15. if DPLL(S', A' ⋃ {var})
16.     return TRUE
17. else return DPLL(S', A' ⋃ {¬ var} )
```

Table 4: DPLL algorithm with 2clseq modifications. Symbols S and S' denote sets of clauses, symbols A and A' denote sets of truth assignments for the variables occuring in S and S' respectively.

structures keep conflicts arising until the algorithm has returned from enough of recursions and thus has canceled assignments made when branching. Line 13 emphasizes the fact that variable selection is based on ranking of the variables as explained in section 7.

10 Test methodology used

As mentioned before, the tests conducted fall into three different classes. Each class has unique properties both in respect to difficulty to solve and test setup. Each of the classes, bounded model checking, integer factorisation and random 3-SAT, are described briefly. Test cases for factorisation and random 3-SAT were generated using the parameters listed in Table 6. Their meaning and the tools used are discussed in corresponding subsections.
In every class the quantity of interest was the processing time taken by 2clseq to come up with a solution. This means to decide whether or not the given instance was satisfiable or not. Processing time was measured using operating system's interface, specifically a tool called 'time'. This enabled us to get the processing time used by the tool alone, leaving out kernel space operations like page faults, I/O-operations and other support functions. The details of the test environment are listed in Table 5.

- **Bounded model checking**

Processor	L2 cache	Physical RAM	OS
AMD Athlon 1,4GHz	256kB	900 MB	Linux-2.4.18

Table 5: Test environment details

These tests consist of set of 15 files from IBM [7]. They represent verification instances of hardware design(s).

- **Integer factorisation**

  In general, given an integer n, the factorisation problem is to find primes $p_1, p_2, ..., p_k$ and integers $e_1, e_2, ..., e_k$ such that $n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$.
  The problems examined in our tests are special cases of this. Satisfiable instances represent factorisation of n as a product of two integers, i_1 and i_2, written as $n = i_1 * i_2$. Unsatisfiable instances in turn represent factorisations of primes. As primes are indivisible, the corresponding instances are unsatisfiable.
  Tests for this category were generated using a tool called genfacbm [4]. It takes the length of the integer to be factorised as input and returns a SAT instance representing the problem, i.e., the factorisation. The aim here was to establish an upper limit for the length of integers 2clseq is expected to factorize in reasonable time.

- **Random 3-SAT in phase transition region**

  This last category consists of randomly generated SAT instances having exactly three literals in each clause. In order to generate instances in the phase transition region the ratio between the total number of clauses and the total number of variables in the problem instance needed to be kept constant at 4.2 [5]. The idea was to then keep increasing the number of variables and observe the behaviour of 2clseq. Here too we wanted to establish an upper limit for the number of variables in instances that 2clseq is expected to solve in reasonable time. Tool called makewff [6] was used to generate testcases having the number of variables from 250 to 390. It should be noted that in this category the tool couldn't be controlled to produce specifically satisfiable (or unsatisfiable) instances.
  In addition to the generated instances we used a set of already existing instances. This set, called uniform-250-1065 [10], consists of 100 satisfiable and 100 unsatisfiable testcases each having exactly 250 variables and 1065 clauses.

11 Results

11.1 Integer factorisation using braun circuits

Both satisfiable and unsatisfiable cases, Figures one and two, seem to be fairly uniform for 2clseq as the variance of CPU time stays low except the case of

Class	Instances			
BMC	Filename			Runs/file
	bmc-ibm-{1-13}.cnf			1
	bmc-galileo-{1,2}.cnf			1
Factors	braun	Length	Seeds	Runs/length
	SAT	24 - 36	1,2,...,12	12
	SAT	37	1,2,3,4	4
	SAT	38 - 41	1,2	2
	UNSAT	24 - 36	1,2,...,12	12
	UNSAT	37	1,2	2
3-SAT		Var N		Runs/Var N
makewff	SAT+UNSAT	250,260,...,330		15
makewff	SAT+UNSAT	340,350,...,390		3
UNIFORM	SAT	250		100
UNIFORM	UNSAT	250		100

Table 6: Test setups for each class. Factors represent the factorisation tests. The circuit type used in the test generation tool was 'braun'. For each integer length the column Seeds lists the seed values used (input for genfacbm). The last class, 3-SAT, represents instances of random 3-SAT. In the instances column SAT denotes satisfiable and UNSAT unsatisfiable. The last column indicates how many instances where used per problem parameter.

satisfiable 34 bits. As can be noted from the figures, the processing time starts to increase rapidly as the length of the integer to be factorised approaches 34 bits. Unsatisfiable cases are clearly more difficult to solve than satisfiable ones. Timeouts started to appear after one CPU hour of processing for instances representing 36-bit integers. Based on these observations it is safe to say that 2clseq is able to factorise integers having length ≤ 34 bits in reasonable time.

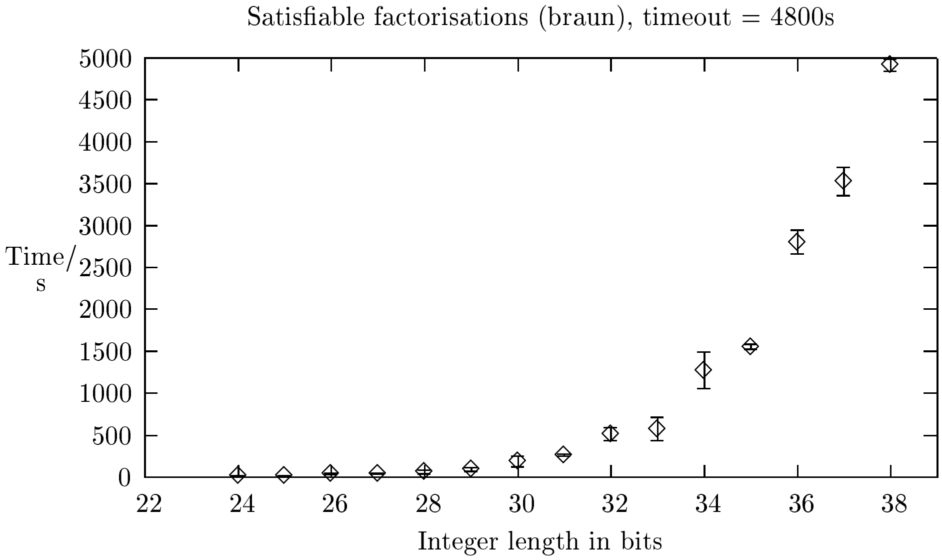

Figure 1: Results for satisfiable integer factorisations. Errorbars denote variance calculated from the measurements.

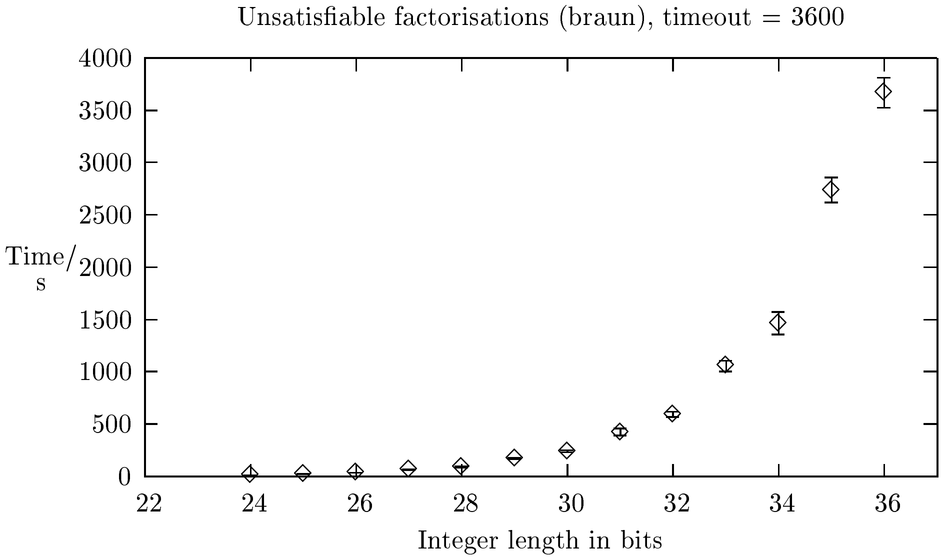

Figure 2: Results for unsatisfiable integer factorisations. Errorbars denote variance calculated from the measurements.

Variable N	SAT	UNSAT	?	ALL
250	7	8	0	15
260	10	5	0	15
270	13	2	0	15
280	12	3	0	15
290	11	4	0	15
300	5	10	0	15
310	12	3	0	15
320	9	3	3	15
330	12	1	2	15
340	1	0	2	3
350	3	0	0	3
360	2	0	1	3
370	1	0	2	3
380	0	0	3	3
390	1	0	2	3

Table 7: Random 3-SAT instances solved. Second column contains number of satisfiable instances solved, third indicates the number of unsatisfiable ones and fourth column shows the number of instances left unsolved. These were timed out after one CPU hour. The fifth column contains the number of testcases in total. Average processing times can be found in Figures three and four.

11.2 3-SAT

The most notable phenomenon in this category was the wide difference in computation time between instances having same test parameters. See Figures three and four. The fact there weren't that many test cases per fixed number of variables might explain the large variances to certain extent. But if one consults Table 8 representing large amount testcases having constant number if variables, one can observe the large variance there too. This suggests that the problem has characteristics contributing to hardness to solve other than the number of variables or the clause/variable ratio discussed earlier. However it is still possible to estimate the limit we are after. By examining Table 7 we can observe that timeouts after one hour of processing started to emerge for instances having 320 variables. Table 7 also reveals that for 310 variables we got out of 15 only three unsatisfiable instances. As unsatisfiable instances were even two or three times harder to solve than satisfiable ones, this particular category needs to be considered with care. But testcases for 300 variables impose much safer grounds to draw conclusions as there were 10 out of 15 cases unsatisfiable and they were all solved within an hour of CPU time. Based on this, one can safely say that if the instances have variable number equal or less than 300, 2clseq can be expected to solve them in reasonable time.

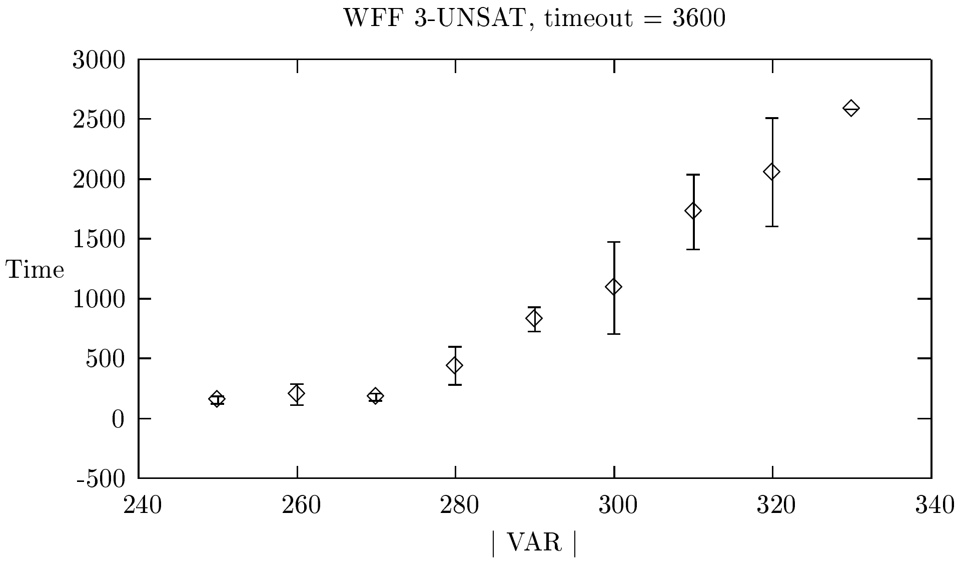

Figure 3: Results for unsatisfiable random 3-SAT instances. Errorbars represent variance calculated from the measurements.

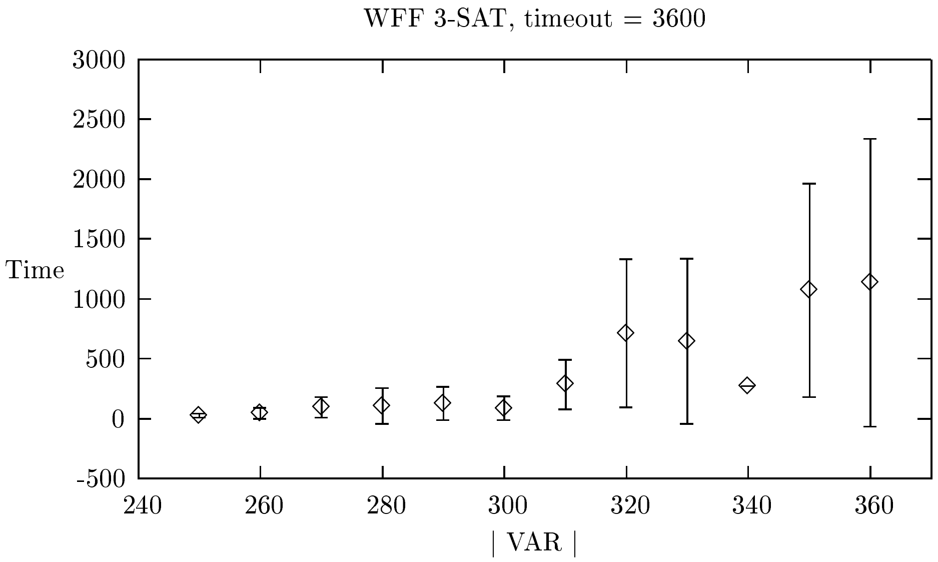

Figure 4: Results for satisfiable random 3-SAT instances. Errorbars represent variance calculated from the measurements. It should be noted from Table 7 that as there was only one solved instance for 340 bits, the variance for it is undefined.

	N	$\overline{T}$	$\sqrt{\frac{\sum(T_i-\overline{T})^2}{N-1}}$
SAT	100	40.6393	46.3994
UNSAT	100	123.91	45.4127

Table 8: Average CPU times and their variances for Uniform-250-1065 testcases. N denotes the number of test instances and T denotes computation time.

Test	Time	Status
bmc-ibm-1.cnf	14.55	SAT
bmc-ibm-2.cnf	0.18	SAT
bmc-ibm-3.cnf	45.63	SAT
bmc-ibm-4.cnf	14.36	SAT
bmc-ibm-5.cnf	1.89	SAT
bmc-ibm-6.cnf	-	OOM
bmc-ibm-7.cnf	1.35	SAT
bmc-ibm-10.cnf	-	OOM
bmc-ibm-11.cnf	487.82	SAT
bmc-ibm-12.cnf	248.48	SAT
bmc-ibm-13.cnf	3598.89	TIMEOUT
bmc-galileo-8.cnf	-	OOM
bmc-galileo-9.cnf	-	OOM

Table 9: Results for BMC (Bounded Model Checking) instances. OOM denotes out of memory.

11.3 Bounded model checking

This was a simple and straightforward test category, but it didn't reveal much information either. Table 9 lists the computation times for the solved instances and short descriptions for the failures.
Every test case was run only once as the only random element in 2clseq is the tie-break of best ranking literals in variable selection. As in random 3-SAT's case here too the difference in computation time was quite large. The fact that 2clseq's memory requirements limited experiments in this category decreased the significance of the test. If nothing else, together with results for random 3-SAT, one can see potential for further study. There seem to be other problem characteristics that we are not aware of yet.

12 Conclusions

From the three problem classes examined, the behaviour of 2clseq was consistent between testcases only in integer factorisation. Even as the memory requirements posed a limit for 2clseq in BMC class, the general performance there, as in case of random 3-SAT, was far from being uniform. Finding explanations for the wide difference in the performance would propably benefit the overall development of the tool itself not to mention widening the general understanding of what properties contribute to the hardness of a problem. However the goal was to establish the upper limits for parameters in integer factorisation and in random 3-SAT classes that instances with smaller parameter values are solved by 2clseq in reasonable time. For the factorisation of an integer the parameter of interest was the length of the integer having value of 34 bits and in random 3-SAT's case the number of variables having upper bound of 300.

References

[1] M. Davis, G. Logemann, and D. Loveland. A Machine Program For Theorem Proving.*In CACM 5, pages 394-397*, 1962.

[2] F. Bacchus. Enhancing Davis Putnam with Extended Binary Clause Reasoning. *National Conference on Artificial Intelligence (AAAI-2002) to appear*, 2002.

[3] F. Bacchus. Extending Forward Checking. *Principles and Practice of Constraint Programming-CP2000, number 1984 in Lecture Notes in Computer Science, pages 35-51.* Springer-Verlag, New York.

[4] T. Pyhälä. *http://www.tcs.hut.fi/~tpyhala/genfacbm.tgz*

[5] J. M. Crawford and L. D. Auton. Experimental Results on the Crossover Point in Random SAT. *In Artificial Intelligence, pages 31-57*, 1996.

[6] H. Kautz and B. Selman *http://www.cs.washington.edu/homes/kautz/walksat/*

[7] http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/BMC/description.html

[8] J. W. Freeman. Improvements to Proportional Satisfiability Search Algorithms.*Ph.D. Dissertation, University of Pennsylvania*, 1995.

[9] R. J. Bayardo, R. C. Schrag. Using CSP Lookback Techniques to Solve Real-World SAT Instances. *In proceedings of the AAAI National Conference (AAAI), pages 203-208..*

[10] http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html

[11] http://gauss.ececs.uc.edu/Conferences/SAT2002/