An Empirical Study of Learning and Forgetting Constraints

Ian P. Gent, Ian Miguel and Neil C.A. Moore
{ipg,ianm,ncam}@cs.st-andrews.ac.uk

School of Computer Science, University of St Andrews, St Andrews, Scotland, UK.

Abstract. Conflict-driven constraint learning provides big gains on many CSP and SAT problems. However, time and space costs to propagate the learned constraints can grow very quickly, so constraints are often discarded (forgotten) to reduce overhead. We conduct a major empirical investigation into the overheads introduced by unbounded constraint learning in CSP. To the best of our knowledge, this is the first published study in either CSP or SAT. We obtain two significant results. The first is that a small percentage of learnt constraints do most propagation. While this is conventional wisdom, it has not previously been the subject of empirical study. Second, we show that even constraints that do no effective propagation can incur significant time overheads. Finally, by implementing forgetting, we confirm that it can significantly improve the performance of modern learning CSP solvers, contradicting some previous research.

1 Introduction

In this paper, we conduct an empirical investigation into the overheads introduced by unbounded constraint learning in CSP. To the best of our knowledge, this is the first published study in either CSP or SAT. We obtain two primary results. The first is that a small percentage of learnt constraints do most propagation. Although this is conventional wisdom, no published study exists. Second, we show that even constraints that do no effective propagation can incur significant time overheads. This clarifies conventional wisdom which suggests that watched literal propagators can have lower overheads when not in use. Finally, we show that forgetting can improve performance of modern learning CSP solvers by exhibiting a working implementation, contradicting some previous published research.

2 Background: Learning and Forgetting in SAT and CSP

Nogood learning is an important CSP search technique. In brief, when the solver reaches a dead-end, a new constraint is added to rule out future branches that

Proceedings of the 18^{th} RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2011).
In conjunction with IJCAI 2011, Barcelona, Spain, July 17-18, 2011.

will fail for the same reason. We experiment on a solver implementing Katsirelos *et al*'s [15–17] generalised nogood learning (g-nogood learning or, more succinctly, g-learning). The first step at a dead-end is to analyse the earlier decisions and propagation that contributed to the current failure. We seek a set of assignments and disassignments that, if repeated, lead directly to a failure. To analyse propagation, *explanations* are used:

Definition 1. *A* disassignment *is a pair of variable x and value a (denoted $x \nleftarrow a$) such that a has been ruled out as a possible value for x at the point in search under consideration. An* assignment *is a pair of variable x and value a (denoted $x \leftarrow a$) where variable x is set to a at the point in search under consideration. An* explanation for disassignment *$x \nleftarrow a$ (or* assignment *$x \leftarrow a$) is a set of assignments and disassignments that are sufficient for a propagator to infer $x \nleftarrow a$ (or $x \leftarrow a$).*

After explanations are processed to produce a new constraint, the solver backjumps and the constraint is added. It has the property that it will propagate immediately to mimic a right branch decision[1], guaranteeing completeness. The power of g-learning comes from learned constraints proceeding to propagate and being combined by iterative application of the above process into more powerful constraints that can remove subtrees of the search tree, as opposed to just providing a shortcut to propagation. While brief, the preceding description is sufficient for this paper. The reader is referred to [20, 15] for more detail. All of the experiments to follow in this paper are based on the minion solver [10] amended to do g-nogood learning.

G-learning is extremely effective on some types of benchmark, but its overheads can dominate on others. First, there is an overhead associated with instrumenting constraint propagators to store explanations, which are needed to produce the new constraints. This problem is mitigated by using *lazy explanations* [9], which reduce the overhead by producing explanations only when they are needed, thereby saving work. In the experiments to follow, lazy explanations are used. However the new constraints must still be propagated, slowing the solver down. Second, g-learning was originally described as *unrestricted learning* [16], where learned constraints are kept forever, resulting in worst case exponential memory usage. In our experience this causes g-learning solvers to run out of RAM on commonly available systems within an hour.

Forgetting in SAT and constraints. The fact that unrestricted learning is impractical has been understood for many years. One way to cope is to store constraints more efficiently, e.g. [22], but no technique can remove the fact storage space still grows unless the set of constraints can be generalised. A second way is to design algorithms to be fundamentally limited in the amount of space they can consume, e.g., dynamic backtracking [11]. A third method is to bound learning at the time constraints are created, by suppressing constraints that take up too much space. Bounding at creation time has been used by Dechter and

[1] note that the propagation is not necessarily the negative of failed left branch, but it simply an additional literal not previously set

Frost [5, 8] in the context of CSP learning solvers; and by Bayardo and Schrag [1], and Marques-Silva and Sakallah [18] for SAT solvers.

A fourth method of reducing overheads is to *forget* (i.e. remove) constraints some time after they were learnt by some heuristic method. Forgetting constraints after adding them is, to the best of our belief, used universally in conflict-driven clause learning (CDCL) SAT solvers, e.g. [1, 6, 12]. We believe that this is the first report of the successful use of forgetting in the core of a CSP solver since the advent of g-learning: relevance-bounding forgetting [1] was used in [8] but with s-nogoods which have been superseded in practice by g-nogoods, and relevance-bounding forgetting was tried unsuccessfully in [16]. In [20, 7], forgetting is an important part of a CSP solver, but via the external use of a SAT solver that itself implements forgetting.

3 Experiments on clause effectiveness

The following experiments analyse the overheads of unbounded constraint learning, showing that a small proportion of all learned constraints typically do the vast majority of all useful propagation and that they take a small proportion of overall time to do so.

3.1 Methodology

In the following experiments each instance was run once with a limit of 10 minutes search time. The reason why they were not run multiple times was that in this experiment the counts are important and variation in time is tolerated. They ran over three Linux machines with 8 Xeon E5430 cores @ 2.66GHz and 8GB memory. Lazy explanations [9] (mentioned in §2) and the conflict-directed dom/wdeg [4] variable ordering heuristic were used throughout.

We use a large, varied and inclusive set of 2,028 benchmark instances from 46 problem classes. The set has been chosen to include as many instances as possible, provided they are modelled using only all-different, table, negative table, disjunction, lexicographic ordering, (weighted) sum$\leq$, (weighted) sum$<$, $x \leq y + c$, $=$, $\neq$, $x \leftarrow c$, $x \not\leftarrow c$, $\lfloor x/y \rfloor = z$, $x \bmod y = z$ and $x \times y = z$ constraints. See [2] for definitions of these. Our sources are Lecoutre's XCSP repository (`http://tinyurl.com/lecoutre`) and our own stock of CSP instances. We include every extensional instance of the 2006 CSP solver competition, together with further instances from the random, industrial and academic spheres. Models runnable using minion can be found at `http://dl.dropbox.com/u/16721904/instances.zip`.

3.2 Small subset of clauses typically do most propagation

Received wisdom states that a small number of learned constraints do the majority of propagation in learning solvers, yet we are aware of no published evidence substantiating this view. The fact that constraint forgetting techniques are effective in learning solvers is consistent with the belief: if few constraints dominate collectively most can be thrown away without harming search. However constraint forgetting in some form is a positive necessity to avoid running out of memory, so it would still benefit the solver even if individual constraints were comparably effective. Irrespective, the effect must be quantified, and understanding the effect quantitatively might help to design effective forgetting strategies.

Procedure Measuring effectiveness of an individual constraint is more difficult in a learning solver than in a standard backtracking solver, because the learning procedure combines constraints together. Hence a constraint may do little propagation itself, but constraints derived from it during the learning process may do a lot. Hence the influence of a constraint may be wide. This is a subtle issue and we have not attempted to measure it. Rather we will be measuring only the direct effects of individual constraints, and not their "influence".

Therefore, in this section, the number of unit propagations is used as a measure of the effectiveness of a learnt constraint. This choice is not immediate, so we will now discuss why it was chosen. The problem is that propagations are not necessarily beneficial if they remove values but do not contribute to domain wipeouts or other failures. To get around this issue, as part of its clause forgetting system (see §4.1) minisat [6] measures the number of times a constraint has been identified as part of the reason for a failure. Hence, we did consider using the number of propagations that lead to failure as a measure of constraint effectiveness, rather than raw number of propagations. However, over our 2050 instances and 566,059 learned constraints, the correlation coefficient between propagation count and count of involvement in conflicts is 0.96. In other words each propagation is roughly equally likely to be involved in a conflict. Hence the following results should apply almost equally to propagations resulting in failure. The advantage of using the total number of propagations is that it is more easily defined and less coupled with learning.

For efficiency reasons, solvers do not collect this data by default. In order to carry out these experiments our solver was amended to print out a short message whenever a constraint propagated, giving the unique constraint number and the node at which the propagation occurred. These data were then analysed externally with the aid of a statistical package. Although this slows the solver down, the experiment is fair because counts are not affected.

Note that the later a constraint is posted, the less time is has to propagate. Hence the number of raw propagations carried out by each constraint are not directly comparable. To get around this, only constraints learned during the first 50% of nodes approximately are included, and for each constraint the number of propagations are counted only over the following 50% of nodes, so that every count is over the same number of nodes. For example, if the problem is solved in 9999 nodes, constraints learned between nodes 1 and 5000 are included, and the constraint learned at node 278 is counted from nodes 278 to 5277.

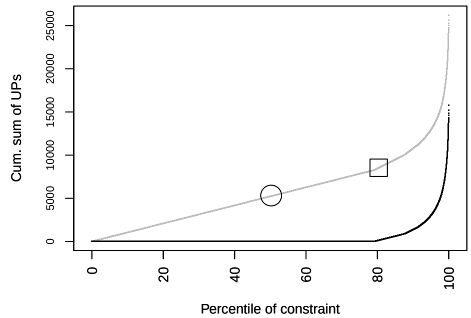


Fig. 1. What proportion of constraints are responsible for what propagation? – single instance

Results and analysis For instance `latinSquare-dg-8_all.xml.minion` we exhibit a graph that we will later show is representative of other instances. The upper curve[2] in Figure 1 shows what proportion of the best constraints are responsible for what proportion of all unit propagations (UPs). By "best" we mean doing the most propagations. Each point is an individual constraint and the constraints are sorted by increasing propagation count moving from the left to the right of the x-axis. The x-axis is the percentile of the constraint's propagation. The y-axis is the number of propagations accounted for by that constraint and those with a lower percentile. For example, the circled point on the x-axis is the median (50th percentile) constraint by propagation count: it is the 5223th constraint, out of 10446. The total propagation count for all 5223 constraints is exactly 5223 [sic] out of a total of 26220 for all constraints, i.e. 20% of the total. Hence the bottom 50% of constraints account for just 20% of all propagation. The slope is shallow until the 80th percentile constraint (marked by a small square), after which it steepens dramatically. Hence the top 20% of constraints do a lot more work than the rest. This agrees with the hypothesis that a minority of constraints do most propagation.

In §2 we noted that each constraint is guaranteed to propagate at least once. This first propagation has the effect of a right branch, so does not contribute effectively since the solver would have done this anyway. Hence we now report results with these ineffective propagations deleted. In the black (lower) curve in Figure 1 the same graph is shown with 1 subtracted from the propagation count of each constraint. Here the curve is zero until the 80% percentile, meaning that the worst 80% of constraints contribute no additional propagation after the right branch, i.e. just one propagation each: just 20% of constraints do *all* useful propagation and 10% do almost all.

The previous results focus on a specific instance, so we will now expand analysis to all 949 instances from the test set that cannot be solved within 1000 nodes of search. This is done to ensure that a trend has a chance to establish: to

[2] the points are close enough together to appear as a single curve, rather than distinct points

P	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1%	0.01	0.01	0.01	0.04	0.03	2.04
5%	0.01	0.02	0.04	0.09	0.09	2.04
10%	0.01	0.05	0.08	0.19	0.18	3.64
15%	0.01	0.09	0.13	0.31	0.31	3.91
20%	0.01	0.12	0.19	0.46	0.47	5.46
25%	0.01	0.17	0.27	0.64	0.68	6.80
30%	0.01	0.23	0.35	0.86	0.92	8.24
35%	0.01	0.30	0.46	1.11	1.22	9.69
40%	0.01	0.37	0.58	1.40	1.58	11.13
45%	0.01	0.47	0.72	1.73	1.99	12.57
50%	0.01	0.57	0.86	2.11	2.51	14.02
55%	0.02	0.67	1.00	2.56	3.22	16.33
60%	0.02	0.78	1.18	3.07	3.93	18.76
65%	0.02	0.89	1.34	3.65	4.86	21.27
70%	0.02	0.99	1.51	4.34	6.09	24.39
75%	0.02	1.09	1.70	5.15	7.56	27.51
80%	0.02	1.19	1.89	6.15	9.50	30.83
85%	0.02	1.32	2.08	7.40	11.75	37.07
90%	0.02	1.44	2.27	9.11	15.37	43.32
95%	0.02	1.55	2.48	11.68	21.88	50.00
100%	0.02	1.65	2.71	16.03	37.06	69.89

Table 1. What proportion of constraints are responsible for what propagation? – all instances

analyse only a few constraints might be less meaningful. In Table 1 for each chosen percentage P, we give what percentage of the best constraints are needed to account for P% of overall non-branching propagation[3]. These results show that usually a small proportion of the best constraints perform a disproportionate amount of propagation. For example 10% of all propagation is performed by a median of 0.08% and maximum of 3.64% of constraints, and 100% by a median of 2.71% and a maximum of 69.89%. Hence the behaviour described above for a single benchmark is robust over many instances: the best few constraints overwhelmingly perform most non-branching propagation. If anything, the above sample instance understates the effect, since it required about 20% instead of the median of 2.71% of constraints to do all propagations.

Conclusion We have shown empirically that the best constraints are responsible for much of the propagation and thus search space reduction.

[3] It may seem anomalous that some entries exceed P%, since the best P% constraints must do *at least* P% of propagations. This apparent anomaly is because there may be no integer number of constraints doing P% of propagation, so it is necessary to overcount.

3.3 Clauses have high time as well as space costs

Unit propagation by watched literals [19] is designed to reduce the amount of time spent propagating infrequently propagating constraints, by the possibility of watches migrating to inactive literals that do not trigger and cost nothing to propagate. Before describing the experiment, we will first briefly outline how watched literal propagation works.

Unit propagation (UP) is a way of propagating clauses. Watched literals are an efficient implementation of UP, first described in [19]. The idea is to *watch* a pair of variables, that are not set to false. Provided that such variables exist, a clause must be satisfiable, and unit propagation needn't happen yet. Suppose that one of these variables is set to false: if another non-false variable can be found then the propagation watches it instead, otherwise the single non-false variable has to be unit propagated to true immediately to avoid the constraint being unsatisfied. The empirical evidence suggests that since the propagator only cares about assignments to two variables it is efficient compared to other unit propagators that watch all assignments (e.g. ones that count false assignments). If the watched variables are set to 1 early in search then the clause will essentially be zero cost until the solver backtracks beyond that point, because it will never be triggered on those variables.

Hence, perhaps weakly propagating constraints do not cost much time, if space is available to store them, since there is a possibility of infrequently propagating constraints doing little work? Hence the next question is: do constraints which do not propagate a lot cost significant time as well as space?

Procedure The minimum amount of time to process a single domain event with a watched literal propagator can be on the order of a handful of machine instructions, taking nanoseconds to run, during which time the system clock may not tick. Hence, to obtain nano-scale timings, the solver keeps a running total of the number of *processor* clock ticks as recorded by the RDTSC register specific to Intel processors [13]. Each of these occupies $1/(2.66 \times 10^9)$ seconds, since we used a 2.66 GHz Xeon E5430. The overhead of collecting data is very low, taking only one assembly instruction to get the number, and a few more cycles to add it to the running total.

At the end of search, all the cycle counts are printed out and analysed externally with the aid of a statistical package.

Results and analysis How does time spent correlate with unit propagations performed? Figure 2 is a scatterplot for the single instance used in §3.2. Each point represents a single constraint. The x-axis gives the number of unit propagations (including the right-branching initial one), and the y-axis the total number of processor cycles used to propagate it during the entire search. First, and unsurprisingly, as an individual constraint propagates more, it often requires more time to do so. What may be surprising is that the *worst case* for constraints is roughly constant, and independent of the number of propagations. That is,

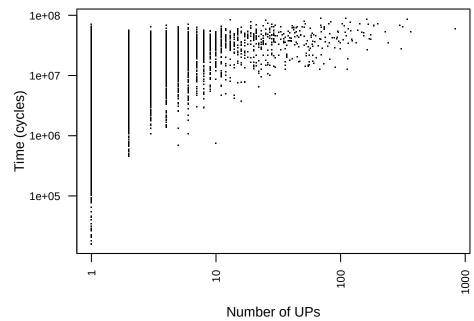


Fig. 2. How much time does propagation take?

constraints which do no effective propagation can take a similar amount of time to propagate as constraints which propagate almost 1000 times. For this sample instance, 74% of propagation time is occupied with constraints that never propagate again after the first time. This suggests that learnt constraints can lead to significant time overhead without doing any useful propagation.

Table 2 extends the study to the 1,923 instances out of the full set of 2,028 where at least one constraint is learned. Each row is a chosen percentage $R\%$ of the total non-branching propagations, and the columns are summary statistics for what % of the overall propagation time the best constraints take to achieve $R\%$ of all propagation. A constraint is "better" than another if it does more propagations per second of time spent propagating. For example, the third row says that the median over all instances is that 10% of all non-branching propagation can be done in just 0.62% of the time taken by the best available constraints. Using the most efficient constraints, all non-branching propagation can be achieved in a mean of less than a quarter of the time of using all constraints. All other time spent is completely wasted since it leads to no effective propagation.

Conclusion The results on all instances confirm the result from the single instance, and shows that learnt constraints which do no propagation contribute significantly to the time overhead of the solver.

The design of watched literal propagators make it possible that constraints that do not propagate will cost the solver very little in time. This is because the watches could potentially migrate to "silent literals" that do not trigger often. Hence, we feel it significant that we have shown that this is often not the case, and useless clauses can be very costly on an individual basis.

4 Clause forgetting

The above results suggest that, if picked carefully, the solver can often remove constraints to save a lot of time at only a small cost in search size. As described in §2, this is a well known and well used technique in both CSP and SAT.

R	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1%	0.00	0.02	0.17	6.12	3.32	100.00
5%	0.00	0.05	0.33	6.17	3.32	100.00
10%	0.00	0.11	0.62	6.30	3.52	100.00
15%	0.00	0.18	0.95	6.50	3.82	100.00
20%	0.00	0.26	1.38	6.79	4.38	100.00
25%	0.00	0.35	1.88	7.12	5.11	100.00
30%	0.00	0.45	2.31	7.52	5.82	100.00
35%	0.00	0.54	2.85	8.07	6.82	100.00
40%	0.00	0.63	3.38	8.46	7.75	100.00
45%	0.00	0.71	4.03	9.01	9.10	100.00
50%	0.00	0.79	4.54	9.46	9.97	100.00
55%	0.00	0.91	5.38	10.50	11.67	100.00
60%	0.00	1.04	6.08	11.16	13.32	100.00
65%	0.00	1.20	6.87	11.97	15.10	100.00
70%	0.00	1.38	7.99	13.06	17.73	100.00
75%	0.00	1.58	9.06	14.00	19.62	100.00
80%	0.00	1.78	10.07	15.27	22.59	100.00
85%	0.00	2.03	11.35	16.78	25.91	100.00
90%	0.00	2.29	12.56	18.55	30.03	100.00
95%	0.00	2.59	14.31	20.76	34.05	100.00
100%	0.00	2.89	15.23	24.01	41.02	100.00

Table 2. How much time does propagation take?–all instances

Indeed Katsirelos and Bacchus have implemented relevance bounded learning for a g-learning solver in [16]. They report poor results showing that relevance bounding with $k = 3$ leads to more timeouts and slower solution time. However a very small number of similar problems are tried so results are inconclusive.

In this section, we try a range of well-known existing strategies for forgetting learned constraints.

4.1 Context

For size-bounded and relevance-bounded learning [5, 8] the solver must respectively not learn the constraint if it has more than k literals in it or remove the constraint once k literals become unset for the first time. Both have been applied successfully to the CSP in the past, but using a s-learning solver. Since they were last tried, algorithms for propagating disjunctions have progressed significantly with the introduction of watched literal propagation [19], meaning that learned constraints are faster to propagate. Hence the techniques may no longer be useful and, if they are useful, the optimal choice of parameters will probably have changed as long clauses become less burdensome. Also, the learning algorithms applied have fundamentally changed with the advent of g-nogood learning. Katsirelos has shown [15] that the properties of clauses change as a result of g-learning, for example the average clause length can reduce. This also motivates the re-evaluation of existing forgetting strategies. Finally, theoretical results [14, 3] from SAT show that there is an exponential separation between solvers using size-bounded learning and learning unrestricted on length, meaning that the former may need exponentially more search than the latter on particular problems. This means that size-bounded learning is theoretically discredited, but it remains to see how it performs in practice.

Recently there have been a collection of new forgetting heuristics in SAT solvers, which are based on activity. Using activity-based heuristics the clauses that are least used for conflict analysis are removed when the solver needs to free space to learn new clauses. As well as guessing which clauses are least beneficial, new strategies also decide *how many* to keep. This is a difficult trade off, because keeping more increases propagation time, but throwing them away reduces inference power. The best choice is problem dependent. We will experiment on what we will call the *minisat* strategy after the solver it originated in [6].

The strategy has 3 main components:

activity each clause has an activity score, which is incremented by 1 each time it is used as an explanation in the firstUIP procedure

decay periodically, activities are reduced, so that clauses that have been active recently are prioritised

forgetting just before the scores are decayed each time, half of all constraints are removed with a couple of exceptions:
- those that have unit propagated in the current branch of search are kept,
- those with scores below a fixed threshold are removed first even if the target of removing half has already been reached, and
- binary and unary clauses are always kept.

In order to implement this algorithm the frequency of decay & forgetting and the divisor for decay must be supplied. The threshold below which all clauses are removed is simply 1 over the size of the clause database.

4.2 Experimental evaluation

We will describe an experiment to test the effectiveness of the forgetting strategies from the literature described above.

Implementing constraint forgetting As mentioned in §3.2 each learned constraint propagates at least once and this is necessary for the completeness of g-learning. Hence when implementing bounded learning, our solver propagates it once anyway even if the constraint is going to be discarded immediately.

In our implementation, currently unit clauses, a.k.a. *locked* clauses[4], can be slated for deletion meaning that they are not propagated any more, but the memory cannot be freed until it is no longer unit. In our solver, restarts are not

[4] nomenclature due to [6]

used. It is possible to prove that deleting clauses is safe (i.e. the solver is still complete), provided that they are not locked.

For k relevance bounding, recall that the solver must remove the constraint when k literals become unset for the first time. Our implementation works as follows: when the constraint is created the literals are sorted by descending depth at which they became false[5] and the k'th depth is selected. When the solver backtracks beyond depth k, exactly k literals will have become unset and the constraint can be deleted. There is little runtime overhead using this implementation.

The implementation of size-bounded learning and the minisat strategy follow straightforwardly from the definitions given above.

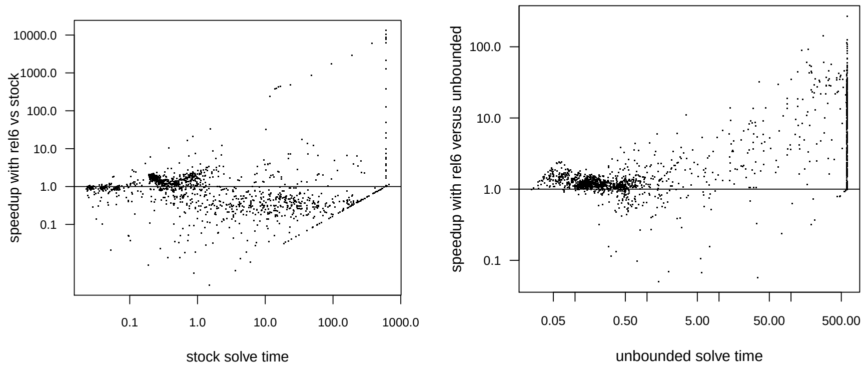
Experimental methodology Each of the 2028 instances was executed four times with a 10 minute timeout, over 3 Linux machines each with 2 Intel Xeon cores at 2.4 GHz and 2GB of memory each, running kernel version 2.6.18 SMP. Parameters to each run were identical, and the minimum time for each is used in the analysis, in order to approximate the run time in perfect conditions (i.e. with no system noise) as closely as possible. Each instance was run on its own core, each with 1GB of memory. Minion was compiled statically (`-static`) using g++ version 4.4.3 with flag `-O3`.

Beauty contest We tried each strategy with a wide range of parameters and in Table 3 report a selection of the best parameters for each. The best parameters were found by testing a wide interval of possible parameters, and finding a local optimum. Close to the local optimum more parameters were tried to locate the best single value where possible (e.g. for discrete parameters). minion with no learning at all is also included in the comparison under name "stock.undefined". In the table, the strategies are abbreviated to name.parameter, except minisat which is abbreviated to minisat.interval.decayfactor.

The "Beauty Contest" columns give both the number of instances solved and the total amount of time spent. Hence an instance that times out does not count towards instances solved and costs 600 seconds. The best strategy is that which solved the most instances, taking into account overall time to break ties. In the table the best strategies are listed first. Finally first and third quartiles and median nodes per second are given. These statistics show the increase or decrease in search speed. A solver with forgetting should have a higher search speed because it has fewer constraints to propagate. The 'Search measures' columns give measures of what effect each strategy has compared to unbounded learning. This is a measure of how effective search is compared to unbounded learning, as opposed to how fast. The columns are as follows:

Instances means the number of instances the variant and unbounded both complete. The number of instances being compared in the following two statistics.

[5] this information is available from the learning subsystem



(a) No learning (b) Unbounded learning

Fig. 3. Graph comparing the best strategy (relevance-bounded $k = 6$) against other strategies

Nodes inc. means what factor additional nodes the strategy needs on those instances. The smaller the number[6], the less propagation is lost as a result of forgetting.

Speedup means speedup factor, e.g. speedup factor of 2 means that the strategy takes half the time to solve the all instances together. Note that because only instances completed by both are included, there are no timeouts in the total.

The aim is to maximise nodes per second, while keeping the node increase as little as possible.

Analysis of results In these results, most of the strategies for forgetting clauses improve over unbounded learning (none.undefined in Table 3) in terms of both instances solved and overall time. There is an overall increase in the number of instances solved: provided that the increased node rate compensates for the increase in the number of nodes searched, there will be a net win. There is an apparent paradox because for some strategies that beat unbounded learning, e.g. size.2, the number of nodes increases more than the node rate in the "search measures" section. However this is not a problem, because "beauty contest" is based on all instances, whereas "search measures" is based only on instances that didn't timeout. Hence the paradox is because for these strategies, the instances that timed out were the most improved in terms of nodes and node rate. This makes sense when the instances that run the longest with unbounded learning are the most encumbered by useless clauses.

These results are interesting because contrary to [16], relevance- and size-bounded learning work well for certain choices of k. However, the results in this paper were based on a larger set of benchmarks and a larger range of parameters were tried. Also, different implementation decisions in our solver will result

[6] constraint forgetting could occasionally lead to *less* search, as in backjumping [21], so a number under 1 is possible in principle

Strategy	Beauty contest					Search measures		
	Instances	Time	1st Q NPS	Median NPS	3st Q NPS	Instances	Nodes inc.	Speedup
stock.undefined	1667	248598.9	403.9	1353.0	10390.0	1312	129.6	6.7
relevance.6	1641	278203.7	205.3	502.4	1257.0	1336	2.4	4.2
relevance.5	1639	277357.3	217.6	541.6	1433.0	1336	2.8	4.7
relevance.4	1639	280652.1	222.5	533.4	1549.0	1333	3.6	4.3
relevance.7	1637	278973.3	201.7	482.9	1184.0	1336	1.9	4.4
size.10	1637	280804.7	196.7	534.4	1225.0	1336	4.1	5.1
relevance.10	1636	279244.4	178.1	454.1	1021.0	1335	1.6	5.2
relevance.3	1635	280366.6	242.1	566.2	1728.0	1336	5.5	3.4
size.8	1635	281008.0	214.6	566.2	1383.0	1335	5.2	4.5
size.5	1634	283213.5	235.9	595.7	1574.0	1335	7.5	3.9
relevance.14	1631	281037.3	141.7	409.5	874.6	1334	1.3	5.6
size.12	1631	282370.3	187.6	504.2	1143.0	1335	2.1	5.5
size.13	1631	282911.4	180.1	485.7	1081.0	1335	1.8	5.5
size.14	1631	283324.7	180.1	469.2	1044.0	1335	1.6	5.7
relevance.15	1629	282680.8	136.6	404.9	865.1	1335	1.3	5.9
size.9	1629	283146.9	205.9	541.2	1298.0	1334	4.5	5.0
size.11	1629	283882.0	193.7	516.0	1170.0	1333	3.0	5.3
relevance.16	1629	284854.4	134.5	406.7	860.9	1335	1.3	5.6
size.15	1628	287587.7	176.5	463.9	1007.0	1333	1.7	4.7
relevance.13	1627	281439.7	155.0	427.0	928.2	1335	1.4	5.3
relevance.2	1625	287833.7	250.6	580.3	2006.0	1329	61.3	3.2
relevance.12	1623	284866.5	159.0	420.5	928.9	1334	1.4	5.3
size.2	1621	289421.7	257.4	604.3	2088.0	1327	21.6	3.7
relevance.17	1620	288246.0	126.1	402.2	830.4	1335	1.3	5.1
size.20	1619	295401.9	155.1	413.9	907.9	1335	1.3	4.9
relevance.20	1618	293226.9	119.2	361.1	783.1	1334	1.2	5.3
size.1	1616	294566.6	262.4	611.1	2192.0	1323	61.6	3.1
mostrecent.1	1600	302325.7	227.2	544.0	2102.0	1319	65.8	3.1
mostrecent.2	1600	305267.5	206.9	500.7	2008.0	1323	37.0	2.8
mostrecent.10	1569	326114.8	155.6	381.5	1683.0	1323	34.8	2.6
relevance.30	1555	333292.2	98.4	255.6	686.2	1335	1.2	4.1
size.30	1554	330743.5	124.0	359.9	786.2	1335	1.2	4.2
minisat.1.1	1517	349391.3	112.9	278.1	1164.0	1326	8.0	2.1
relevance.40	1501	360096.1	70.5	166.2	635.5	1335	1.1	3.3
size.40	1498	354322.2	108.1	260.1	720.8	1334	1.1	3.9
mostrecent.100	1475	386555.2	77.2	217.8	1002.0	1326	6.1	2.2
minisat.201.501	1440	410767.3	60.8	173.3	810.8	1321	2.0	2.0
minisat.201.1001	1439	411044.4	60.9	170.6	800.4	1321	2.0	2.0
minisat.201.1	1438	410130.1	60.9	174.2	805.6	1321	2.0	2.1
minisat.401.501	1419	431958.5	46.4	152.4	698.8	1319	1.8	1.9
minisat.401.1001	1417	438939.3	45.6	146.5	676.0	1320	1.8	1.7
minisat.401.1	1413	444863.3	43.8	143.5	660.1	1319	1.8	1.6
relevance.100	1404	406542.4	31.4	99.2	564.3	1330	1.0	2.0
size.100	1397	406529.6	40.5	110.5	581.3	1330	1.1	1.9
minisat.601.1001	1373	500036.1	36.8	127.9	586.7	1319	1.6	1.4
minisat.601.501	1371	502484.1	36.1	121.2	583.9	1318	1.5	1.4
mostrecent.1000	1371	559058.3	31.6	106.3	566.1	1330	1.3	1.6
minisat.601.1	1367	510004.5	35.8	126.0	581.4	1316	1.4	1.5
minisat.1.1001	1344	440553.2	22.7	100.7	585.6	1322	3.0	0.9
none.undefined	1343	440552.2	22.2	76.4	510.0	1343	1.0	1.0
minisat.1.501	1343	442209.0	22.6	97.6	574.2	1321	3.0	0.9

Table 3. Comparison of various strategies for forgetting constraints

in a different time-space trade off. In fact, the best strategy solves 298 more instances than unbounded learning in about 45 hours less runtime. However it still trails stock minion by 26 instances and about 8 hours of runtime. In spite of this, Figure 3(a) gives evidence that learning is still valuable and promising in specific cases. Each point is an instance, with the x-axis the runtime taken by stock Minion and the y-axis is stock runtime over rel.6 runtime; points above the line are speedups and points below are slowdowns. Whilst many instances are slowed down, speedups of up to 5 orders of magnitude are available on some types of problem. Apart from the best strategy, various parameters for relevance-bounded learning perform similarly to $k = 6$, as well as some size-bounded learning parameters. It seems clear that they are significantly better than unbounded learning, but not much different to each other.

The minisat strategy is not effective for any choice of parameters that we tried. However there is reason to believe that a better implementation might improve matters. Notice that the increase in nodes for the better strategies ($200.X$) is relatively small. Using a profiler, we have discovered that the reason for slowness is the amount of time taken to maintain and process the scores, and to process the constraints periodically. Hence perhaps a better implementation would turn out to perform competitively overall.

Now we will analyse the best forgetting strategy more carefully. Figure 3(b) depicts the speedup on each instance for relevance-bounded $k = 6$ compared to unbounded. It shows that most individual instances are speeded up, sometimes by two orders of magnitude, although a few are slowed down by up to an order of magnitude.

In conclusion, whether to use learning remains a modelling decision, where big wins are sometimes available but sometimes it is better turned off.

5 Conclusions

In this paper, we have carried out the first detailed empirical study of the effectiveness and costs of individual constraints in a CDCL solver. We found that, typically, a very small minority of constraints contribute most of the propagation added by learning. While this is conventional wisdom, it has not previously been the subject of empirical study. It is important to verify and make precise folklore results, for until evidence exists and is published it is unverifiable and acts as a barrier for entry to new researchers, who may not yet be aware of folk knowledge.

Furthermore, these best constraints cost only a small fraction of the runtime cost. Conversely, constraints that do no effective propagation can incur significant time overheads. This contradicts conventional wisdom which suggests that watched literal propagators have lower overheads when not in use. This result shows why it is important to experiment on "known" results, because they are not always entirely correct.

Together, these results explain why forgetting can work so well. It is obvious that forgetting is a positive necessity due to memory constraints, but this research shows that forgetting is not only necessary but also fortuitously effective because of the disparity in propagation between constraints.

Finally, we performed an empirical survey of several simple techniques for forgetting constraints in g-learning, and found that they are extremely effective in making the learning solver more robust and efficient, contrary to some previously published evidence.

References

1. R. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. pages 203–208. AAAI Press, 1997.
2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report 08, Swedish Institute of Computer Science, 2005.
3. E. Ben-Sasson and J. Johannsen. Lower bounds for width-restricted clause learning on small width formulas. In *SAT*, volume 6175 of *LNCS*, pages 16–29, 2010.
4. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI 04*, pages 482–486, August 2004.
5. R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.
6. N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
7. T. Feydy and P. J. Stuckey. Lazy clause generation reengineered. In I. P. Gent, editor, *CP*, volume 5732 of *LNCS*, pages 352–366. Springer, 2009.
8. D. Frost and R. Dechter. Dead-end driven learning. In *AAAI-94*, volume 1, pages 294–300. AAAI Press, 1994.
9. I. Gent, I. Miguel, and N. Moore. Lazy explanations for constraint propagators. In *PADL 2010*, number 5937 in LNCS, January 2010.
10. I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.
11. M. L. Ginsberg. Dynamic backtracking. *JAIR*, 1:25–46, 1993.
12. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549 – 1561, 2007.
13. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.
14. J. Johannsen. An exponential lower bound for width-restricted clause learning. In O. Kullmann, editor, *SAT*, volume 5584 of *LNCS*, pages 128–140, 2010.
15. G. Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, Jan 2009. `http://hdl.handle.net/1807/16737`.
16. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *CP*, pages 873–877, 2003.
17. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In M. M. Veloso and S. Kambhampati, editors, *AAAI*, pages 390–396, 2005.
18. J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.
19. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC 01*, 2001.
20. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
21. P. Prosser. Domain filtering can degrade intelligent backtracking search. In *13th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.
22. G. Richaud, H. Cambazard, and N. Jussien. Automata for nogood recording in constraint satisfaction problems. In *In CP06 Workshop on the Integration of SAT and CP techniques*, 2006.