

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. This is the first such 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

Conflict-driven constraint learning (CDCL) is a technique well-used in both SAT and CSP solvers. However little is known about the costs and benefits of the constraints added, usually only that certain algorithms are more effective than others at reducing search time. We give the first detailed empirical analysis of how much inference can be expected from constraints, and how much these inferences cost the solver. These results confirm that constraint forgetting is essential and is likely to be successful. We demonstrate its effectiveness in a g-nogood learning framework, differing from certain earlier published results.

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 will fail for the same reason. We focus on Katsirelos *et al*'s [1–3] g-nogood learning (g-learning) adapted to use lazy explanations [4]. 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. *An explanation for disassignment $x \leftarrow a$ (or assignment $x \leftarrow a$) is a set of assignments and disassignments that are sufficient for a propagator to infer $x \leftarrow a$ (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, 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 [5, 1] for more detail.

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 *lazy learning* [4], which reduces the overhead by producing explanations more efficiently. However the new constraints must still be propagated and this slows the solver down. Second, g-learning was originally described as *unrestricted learning* [2], where learned constraints are kept forever, but in an exponential search tree this results in 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 with this is to store constraints more efficiently, e.g. [6], but this does not 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 [7]. 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 Frost [8, 9] in the context of CSP learning solvers; and by Bayardo and Schrag [10], and Marques-Silva and Sakallah [11] 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 CDCL SAT solvers, e.g. [10, 12, 13]. 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 [10] was used in [9] but with s-nogoods which have been superseded in practice by g-nogoods, and relevance-bounding forgetting was tried unsuccessfully in [2]. In [5, 14], forgetting is an important part of a CSP solver, but via the external use of a SAT solver that itself implements forgetting.

3 Empirical Study of Clause Learning

As stated in [15], there exist few empirical studies into the effectiveness of modern CDCL solvers. Although there exist many *techniques* that undoubtedly speed up such solvers, there is a lack of concrete knowledge about what underlies their success. We will now investigate the costs and benefits of CDCL in a solver *without* forgetting, in order to see where forgetting is likely to help.

Experimental setup. In the following experiments each instance was run 5 times with a limit of 10 minutes search time, and the instance with median

runtime was taken as representative. They ran over three Linux machines with 8 Xeon E5430 cores @ 2.66GHz and 8GB memory. We used a version of minion[16] containing a faithful implementation of g-learning [1] with lazy learning and used the dom/wdeg [17] variable ordering heuristic throughout.

Test instances. We use a large, varied and inclusive set of 2,050 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 , $x \leftarrow c$, $x \leftarrow c$, $\lfloor x/y \rfloor = z$, $x \bmod y = z$ and $x \times y = z$ constraints. See [18] 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.

3.1 Some constraints 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 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, we cannot be sure until the effect is quantified, and understanding the effect quantitatively might help to design effective forgetting strategies.

We use number of propagations as a measure of the effectiveness of a learnt constraint. More propagations are not necessarily beneficial if the propagations remove values but do not contribute to domain wipeouts or other failures. Hence, we did consider using the number of propagations that lead to failure. The correlation coefficient between propagation count and count of involvement in conflicts is 0.95 over 4,933,170 constraints from 938 instances, hence the values are almost linearly dependent, so our results should apply almost equally to propagations resulting in failure. From here we report only the total number of propagations since it is more easily defined and less coupled with learning.

We first exhibit a graph for a single instance that we will later show is representative of most other instances. The grey (upper) curve in Figure 1(left) shows what proportion of the best constraints are responsible for what proportion of all propagations. By “best” we mean doing the most propagations. Each point is an individual constraint. 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 adjacent to 5925 on the y -axis showing that the bottom 50% of constraints account for just 19% 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 our hypothesis that a minority of constraints do most propagation.

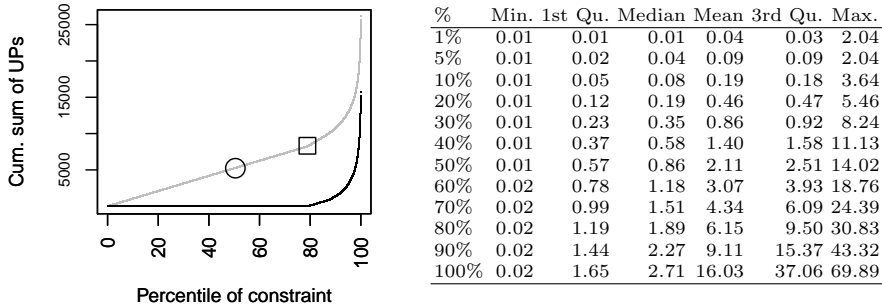


Fig. 1: What proportion of constraints are responsible for what propagation?

We noted in §2 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. We report results with these ineffective propagations deleted. In the black (lower) curve in Figure 1(left) 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: just 20% of constraints do *all* useful propagation and 10% do almost all.

These results focus on a single instance, so we will now summarise over all 948 instances from our test set that cannot be solved within 1000 nodes of search. In Figure 1(right) for each chosen percentage P we summarise what percentage of the best constraints are needed to account for $P\%$ of overall non-branching propagation¹. 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 just 0.08% of constraints, and 100% by a median of just 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, our sample instance understates the effect, since it required about 20% instead of the median of 2.71% of constraints to do all propagations.

We have shown that the best constraints are responsible for much of the propagation and thus search space reduction. 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. Perhaps these weak constraints do not cost much time, if space is available to store them. Hence we ask: do constraints which do not propagate cost significant time as well as space?

3.2 Constraints have time as well as space overheads

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,

¹ 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 we need to overcount.

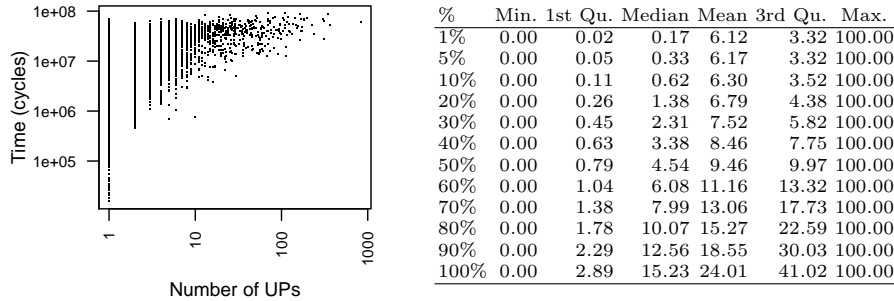


Fig. 2: How much time does propagation take?

taking nanoseconds to run. To obtain nano-scale timings, we total the number of processor clock ticks recorded by the RDTSC register. Each of these occupies $1/(2.66 \times 10^9)$ seconds, since we used 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.

How does time spent correlate with unit propagations performed? Figure 2(left) is a scatterplot for the single instance used in §3.1. 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, 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, we found that 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.

Figure 2(right) extends the study to all 2,050 instances. 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. 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 using all available constraints. All non-branching propagation can be achieved in a mean of less than a quarter of the time of using all constraints. Note that all other time spent is completely wasted since it leads to no effective propagation. This confirms the result from the single instance, and shows that learnt constraints which do no propagation contribute significantly to the time overhead of the solver.

4 Constraint Forgetting in g-learning

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 not a new idea in either constraints or SAT. Indeed Katsirelos and

Bacchus have implemented relevance bounded learning for a g-learning solver in [2]. However, on a single problem class, they reported poor results showing that relevance bounding with $k = 3$ leads to more timeouts and slower solution time.

We try the following forgetting strategies:- *unbounded* [2]: never throw away; *size bounded(k)* [8]: learn only constraints of arity k or less; *relevance bounded(k)* [10]: throw constraint away once k literals become unset; and *last(k)*: keep the last k constraints added. The last strategy performed only slightly better than unbounded and worse than the other strategies, and we omit its results.

Implementing Constraint Forgetting. As mentioned in §3.1 each learned constraint propagates at least once and this is necessary for the completeness of g-learning. Hence in our implementation of size-bounded learning each constraint propagates once even if it is too long. Lazy explanations [4] require that constraints are available for reference as long as any pruning they have done remains current. In our system, such *locked clauses* [12] can be slated for deletion meaning that they are not propagated any more, but the memory cannot be freed until all explanations associated with the constraint are no longer needed.

Recall that for k relevance bounding, 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 false² and the k 'th depth is selected. When the solver backtracks beyond this depth k literals will have become unset and the constraint can be deleted. There is little runtime overhead using this implementation.

Experimental Results on Constraint Forgetting. We tried each strategy with a wide range of parameters and in Table 1 report the best parameter for each. The “Beauty Contest” columns give both the number of instances solved and the total amount of time spent. Hence a timeout 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. Finally first and third quartiles and median nodes per second are given, to show the increase in search speed gained. The ‘Search measures’ columns give measures of what effect each strategy has compared to unbounded learning: the number of instances both variants complete; what factor additional nodes the strategy needs on those instances; and the speedup factor on those instances. When the former is smaller less propagation is lost and the search space is closer to unbounded³.

Table 1 show that the best forgetting strategies increase the nodes only modestly but obtain a very worthwhile reduction in search time due to increased nodes per second. Our best strategy is relevance bounded learning with $k = 14$. With this strategy the search space is only increased by about a third, but the speedup is over 5 times. It solves 158 more instances than unbounded learning in 30% less time. The effect is greatest on instances that take a learning solver the longest: unbounded learning exceeded the time limit 691 times, whilst relevance(14) did so 402 times (and non-learning 364 times). This is expected as the time and space overheads of learning are largest on instances that have run

² this information is available from the learning subsystem

³ constraint forgetting could occasionally lead to *less* search, as in backjumping [20]

Strategy	Beauty contest					Search measures					
	Instances	Time	1st Q	NPS	Median	NPS	3st Q	NPS	Instances	Nodes inc.	Speedup
stock	994	230,614.7	414.6	1375.0	11020.0	804	8.6	6.1			
relevance.14	971	269,991.0	154.3	449.0	1125.0	809	1.3	5.4			
size.12	971	283,492.0	207.0	541.6	1419.0	810	1.6	5.5			
unbounded	813	387,867.5	25.8	92.6	843.1	813	1.0	1.0			

Table 1: Comparison of various strategies for forgetting constraints

for most nodes. Stock Minion (i.e. without any learning method at all) beat unbounded by a factor of five or more in runtime 422 times and lost by this margin 158 times, whereas stock beat relevance.14 fivefold or better only 129 times and lost on 59. Hence bounding seems to make learning more robust.

These results contradict those of [2] where relevance bounding was found inferior to unrestricted learning. We have used a larger benchmark set (including the instances they tried) and our solver is not restricted to FC consistency. In [21], relevance bounded learning for SAT solvers was investigated and was little different to unbounded; our study differs in being focussed on a learning CSP solver. In [9], relevance bounded learning with $k = 3$ was found inferior to g-nogoods [3]. The advent of watched literal propagators [22] to propagate learning constraints very efficiently may explain why the optimal value for k has increased.

Table 1 shows that stock Minion (with no learning) is the dominant single strategy: the best learning method is still 21 instances behind. We believe this is impressive on a benchmark set containing one quarter random instances, which learning solvers are known to be bad at [1]. Figure 3 gives further evidence that learning is valuable and promising. Each point is an instance, with the x -axis the runtime taken by stock Minion and the y -axis is stock runtime over relevance.14 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. Hence whether to use learning remains a modelling decision, where big wins are sometimes available but sometimes it is better turned off. In related work, we have used machine learning to decide whether or not to use lazy learning (without forgetting) [23]. We were able to create a decision procedure that runs all instances in less cumulative time than either solver individually. Speedups such as forgetting should make a combined solver even better.

5 Conclusions

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

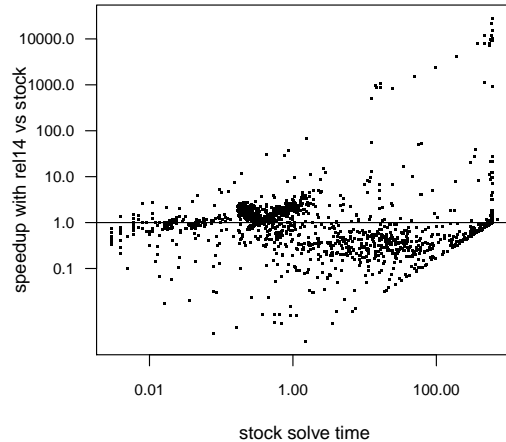


Fig. 3: Comparison of strategies

very small minority of constraints contribute most of the propagation added by learning. Furthermore, these best constraints cost only a small fraction of the runtime cost. 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.

References

1. Katsirelos, G.: Nogood Processing in CSPs. PhD thesis, University of Toronto (Jan 2009) <http://hdl.handle.net/1807/16737>.
2. Katsirelos, G., Bacchus, F.: Unrestricted nogood recording in CSP search. In: CP. (2003) 873–877
3. Katsirelos, G., Bacchus, F.: Generalized nogoods in CSPs. In: AAAI05. 390–396
4. Gent, I., Miguel, I., Moore, N.: Lazy explanations for constraint propagators. In: PADL 2010. LNCS (January 2010)
5. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. *Constraints* **14**(3) (2009) 357–391
6. Richaud, G., Cambazard, H., Jussien, N.: Automata for nogood recording in constraint satisfaction problems. In: In CP06 Workshop on the Integration of SAT and CP techniques. (2006)
7. Ginsberg, M.L.: Dynamic backtracking. *JAIR* **1** (1993) 25–46
8. Dechter, R.: Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.* **41**(3) (1990) 273–312
9. Frost, D., Dechter, R.: Dead-end driven learning. In: AAAI94. 294–300
10. Bayardo, R.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world sat instances, AAAI Press (1997) 203–208
11. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A new search algorithm for satisfiability. In: ICCAD-96. (November 1996) 220–227
12. En, N., Srensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS. (2003) 502–518
13. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics* **155**(12) (2001) 1549 – 1561 SAT.
14. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In Gent, I.P., ed.: CP. Volume 5732 of LNCS., Springer (2009) 352–366
15. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: IJCAI. (2009) 399–404
16. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. (2006) 98–102
17. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI 04. (August 2004) 482–486
18. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog. Technical Report 08, Swedish Institute of Computer Science (2005)
19. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 01. (2001)
20. Prosser, P.: Domain filtering can degrade intelligent backtracking search. In: IJCAI, Morgan Kaufmann (1993)
21. Lynce, I., Marques-Silva, J.P.: The effect of nogood recording in DPLL-CBJ SAT algorithms. In: Recent Advances in Constraints. Volume 2627 of LNCS., Springer (2003) 144–158
22. Gent, I.P., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in minion. In: CP. (2006) 182–197
23. Gent, I.P., Kotthoff, L., Miguel, I., Moore, N.C., Nightingale, P., Petrie, K.: Learning when to use lazy learning in constraint solving. Technical Report 2010/2, CIRCA, University of St Andrews (2010) <http://tinyurl.com/circapreprint>.