Better Caching for Better Model Counting

Jeroen G. Rook[1]*, Anna L. D. Latour[1], Holger H. Hoos[1,2], and Siegfried Nijssen[3]

[1] Leiden University, Leiden, The Netherlands
[2] University of British Columbia (UBC), Vancouver, Canada
[3] UCLouvain, Louvain-la-Neuve, Belgium

Abstract

State-of-the-art model counters provide a variety of branching heuristics, aiding users in configuring these solvers so they perform well on specific types of problem instances. However, they provide a limited choice of cache management strategies. We argue that the state of the art in model counting could benefit from more sophisticated heuristics for cache management. We motivate this with preliminary results and propose to use machine learning to develop new cache management heuristics.

Summary

Over the last two decades, exact model counters have become increasingly fast, due to *conflict-driven clause learning* (CDCL) [4], component caching [4] and new variable branching heuristics [6, 8].

Model counters store counts of subformulae (components) in a cache. If a previously processed subformula is encountered later in the search, its model count can be retrieved from the cache, instead of recomputed. Innovation in component encodings made their memory usage more compact, making it possible to store more partial model counts in the same amount of memory [8]. However, memory is a limited resource, and even solvers that use compactly encoded components will eventually fill their cache. Modern model counters typically clean up in a first-in-first-out manner.

We observe that (1) this strategy could be improved by developing heuristics that clean up components based on criteria other than age, such as predicted number of cache hits during the rest of the search or predicted computational costs of recomputing model counts, and (2) model counters store any component in the cache, regardless of their potential utility for the rest of the search process. We also observe that the use of component caching with a limited amount of memory can facilitate a favourable time-space trade-off [1]. This is important, because memory is ultimately a hard constraint, while time is less so.

Therefore, we aim to develop new storage and deletion heuristics for more efficient cache management. Our ultimate goal is to solve model counting problems faster, without increasing the available memory for caching. Here, we present first steps towards achieving that goal. First, we empirically determine the running time reduction we can obtain by automatically configuring existing solver parameters, motivating our belief that innovation in cache management can further reduce running time. We then present our ideas on how to create cache management schemes for faster model counting.

To gauge how much model counters might benefit from better cache management heuristics, we took the non-probabilistic version of state-of-the-art model counter GANAK [6] and exposed many hard-coded design choices, in the form of configurable parameters.

We also added three simple cache clean-up schemes: one based on the number of total and recent cache hits [8], one based on the time since the last hit, and one that deletes components uniformly at random. This resulted in a highly configurable model counter. We then constrained the amount of memory available for caching component counts to 100 MB and applied *automated algorithm configuration* (AAC) [2] to the (newly exposed) parameters of GANAK, to find optimised configurations for two specific types of problem instances.

We ran our experiments on instances from

*Corresponding author j.g.rook@umail.leidenuniv.nl

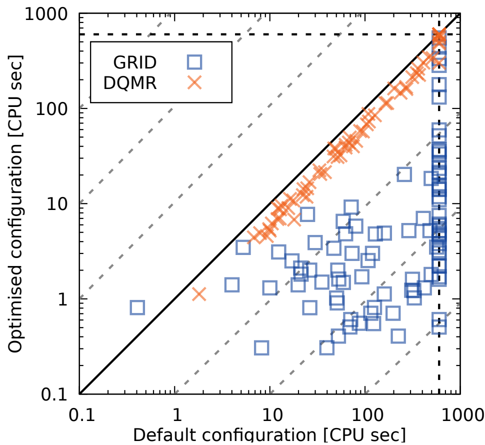

Figure 1: Running time comparison between default and optimised configurations of GANAK for instances of two benchmarks. Cutoff time: 600 CPU sec; maximum cache size: 100 MB.

Table 1: PAR10 values in CPU sec, number of time outs and total number of instances for default and optimised configurations for instances of two benchmarks.

	GRID		**DQMR**	
	PAR10	t/o	PAR10	t/o
Default	2483	36/90	949	9/64
Optimised	28	0/90	481	4/64

unweighted versions of the GRID and DQMR benchmark sets [5], as these contain sufficiently many instances whose solving times with GANAK are within a reasonable range. We selected 180 and 127 instances, respectively.

We randomly split each dataset up into two equally-sized subsets: one for configuration, and one for evaluation. The cache size is limited to 100 MB, to ensure the cache is filled and cleaned up during search. Using SMAC [3], we then automatically configured GANAK on the training set of each of the benchmarks individually. Figure 1 and Table 1 compare the running times and PAR10[1] values of GANAK's default configuration to those of the optimised configuration, measured on the test set instances of the two different benchmarks.

[1] Penalised Average Runtime: counting every instance not solved within the cutoff time as ten times.

First consider the results on the GRID examples. We see that the optimised configuration solves all instances within the cutoff time and that the PAR10 value improves by two orders of magnitude after configuration. Figure 1 shows speedups of up to a factor of 550 for the 54 examples that were already solved within cutoff time by the default configuration.

Now consider the DQMR results. Table 1 shows that the PAR10 value decreased by a factor 2 after configuration; however, some instances were still not solved within the cutoff; Figure 1 shows that running time decreased quite uniformly across our instance set.

Overall, automated configuration of GANAK's current design space produces much less pronounced improvements over GANAK's default parameters for these instances than on the GRID instances. It will be interesting to investigate further why this is the case.

The variable branching heuristics available in GANAK come in many different flavours: following fail-first strategies, aiming to maximise the number of cache hits, or simply adding randomness. These strategies have their own configurable parameters. On the other hand, even our extended version of GANAK only provides a few (very crude) choices for cache clean-up, and only one heuristic for filling the cache: it simply stores every count it computes. Neither optimised configuration chooses a cache clean-up strategy that differs from GANAK's default, which uses a first-in-first-out heuristic.

We therefore believe that we can achieve further performance improvements by expanding the cache-related design space explored by AAC, through the addition of new, more refined cache management heuristics. The results above motivate the need for these improvements. To find these heuristics, we take inspiration from CrystalBall [7], where data mining is used to predict how long a SAT solver should keep a learnt clause. We plan to apply the same principle to two prediction tasks: (1) given a newly computed component count, should the solver store it in the cache, and (2) given a stored component count, should the solver delete it from the cache?

References

[1] Bacchus, F., Dalmao, S., Pitassi, T.: DPLL with caching: A new algorithm for #SAT and Bayesian inference. ECCC (2003)

[2] Hoos, H.H.: Programming by optimization. Commun. ACM **55**(2), 70–80 (2012)

[3] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: LION (2011)

[4] Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT (2004)

[5] Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI (2005)

[6] Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: A scalable probabilistic exact model counter. In: IJCAI (2019)

[7] Soos, M., Kulkarni, R., Meel, K.S.: CrystalBall: Gazing in the black box of SAT solving. In: SAT (2019)

[8] Thurley, M.: sharpSAT — Counting models with advanced component caching and implicit BCP. In: SAT (2006)