Old or Heavy?
Decaying Gracefully with Age/Weight Shapes.

Michael Rawson and Giles Reger

University of Manchester, Manchester, UK

Abstract. Modern saturation theorem provers are based on the given-clause algorithm, which iteratively selects new clauses to process. This clause selection has a large impact on the performance of proof search and has been the subject of much folklore. The standard approach is to alternate between selecting the *oldest* clause and the *lightest* clause with a fixed, but configurable *age/weight ratio* (AWR). We show that an optimal fixed value of this ratio can produce proofs significantly more quickly on a given problem, and further that varying AWR during proof search can improve upon a fixed ratio. Based on these observations we develop several new modes for the Vampire prover which vary AWR according to a "shape" during proof search. The modes solve a number of new problems in the TPTP benchmark.

1 Introduction

Currently, the most successful theorem provers (such as Vampire [4], E [12], or SPASS [17]) for first-order logic are saturation-based, utilising the well-known *given-clause algorithm*. Simply, this algorithm *saturates* a set of clauses by iteratively selecting a clause and performing all non-redundant inferences with it until all clauses have been selected or the empty clause (witnessing inconsistency) has been found. Clearly, the order in which clauses are selected is key to the performance of the algorithm. Over the past few decades a certain amount of folklore has built up around the best methods for clause selection and recent work by Schulz and Möhrmann [13] systematically studied these. Our work extends this study with new results and also introduces the concept of a *variable* clause selection strategy (one that changes over time) which we instantiate with two simple patterns (or *shapes*) that prove to be pragmatically useful.

We restrict our attention to clause selection strategies that alternate between selecting clauses based on age (i.e. in a first-in-first-out manner) and weight (i.e. those with the fewest symbols first). It was confirmed by Schulz and Möhrmann that alternating these two heuristics outperforms either by itself. We refer to the ratio with which we use either heuristic as the *age/weight ratio* (AWR) as this is the terminology employed by the Vampire theorem prover, which we use as a vehicle for our study.

After covering relevant background material in Section 2 the remainder of the paper makes two main contributions. Firstly, in Section 3 we experimentally

confirm the folklore that (i) the choice of age/weight ratio often has a significant effect on the performance of proof search, and (ii) there is no 'best' age/weight ratio: indeed, a large range of pragmatically useful ratios exists. Secondly, we demonstrate (Section 4.1) that varying the age-weight ratio over time can achieve better performance than a fixed ratio and use this to motivate the addition of so-called *age/weight shapes* for varying the ratio over time. Experiments (Section 5) with these new options implemented in the Vampire theorem prover show a significant improvement, proving many new problems unsolvable by any current configuration of Vampire.

2 Background

This section introduces the relevant background for the rest of the paper.

First-Order Logic and Weight. Our setting is the standard first-order predicate logic with equality. A formal definition of this logic is not required for this paper but an important notion is that of the *weight* of a clause. In first-order logic, terms are built from function symbols and variables, literals are built from terms, and clauses are disjunctions of literals. The weight of a term is the number of function and variable symbols occurring in it and this notion is lifted to clauses.

Saturation-Based Proof Search. Saturation-based theorem provers *saturate* a set of clauses S with respect to an inference system $\mathbb{I}$: that is, computing a set of clauses S' by applying rules in $\mathbb{I}$ to clauses in S until no new clauses are generated. If the empty clause is generated then S is unsatisfiable. Calculi such as resolution and superposition have conditions that ensure *completeness*, which means that a saturated set S is satisfiable if it does not contain the empty clause as an element. As first-order logic is only semi-decidable, it is not necessarily the case that S has a finite saturation, and even if it does it may be unachievable in practice using the available resources. Therefore, much effort in saturation-based first-order theorem proving involves controlling proof search to make finding the empty clause more likely (within reasonable resource bounds). One important notion is that of *redundancy*, being able to remove clauses from the search space that are not required. Another important notion are literal selections that place restrictions on the inferences that can be performed. Both notions come with additional requirements for completeness. Vampire often gives up these requirements for pragmatic reasons (incomplete strategies have been found to be more efficient than complete ones in certain cases) and in such cases the satisfiability of S upon saturation is *unknown*.

The Given Clause Algorithm and AWR Clause Selection. To achieve saturation, the *given clause algorithm* organises the set of clauses into two sets: the *active* clauses are those that have been active in inferences, whereas the *passive* clauses are those that have not. Typically, a further *unprocessed* set is required in order to manage the clauses produced during a single iteration of the loop. Realisations

of the given clause algorithm generally differ in how they organise simplifications. There are two main approaches (both implemented by Vampire, originally found in the eponymous theorem provers Otter [5] and DISCOUNT [1]): the Otter loop uses both *active* and *passive* for simplifications, whereas the Discount loop only uses *active*.

The algorithm is centred around the *clause selection* process. As previously mentioned, there are two main heuristics for this:

- *By Age (or First-in/First-out)* clause selection prefers the oldest clause (produced earlier in proof search), simulating a *breadth-first* search of the clause space. In Vampire the age of a clause is the number of inferences performed to produce it (e.g. input clauses have age 0).
- *By Weight (or symbol-counting)* clause selection prefers the youngest clause (with the fewest symbols). The intuition behind this approach is that the sought empty clause has zero symbols and lighter clauses are (in some sense) closer to this. Furthermore, lighter clauses are more general in terms of subsumption and tend to have fewer children, making them less explosive in terms of proof search.

Schulz and Möhrmann show that alternating these heuristics is beneficial and we consider this setting here. In Vampire this alternation is achieved by an age/weight ratio (AWR) implemented by a simple *balancing* algorithm. The balance is initialised to 0 and used as follows: a positive balance means that a clause should be selected by age, whereas a negative balance means that a clause should be selected by weight; given a ratio of $a : w$ the balance is incremented by a when selecting by age and decremented by w when selecting by weight. Figure 1 gives the Discount algorithm along with balance-based AWR clause selection. The lines relevant to clause selection are marked with ✓.

Portfolio Solvers. Vampire is a portfolio solver and is typically run in a mode that attempts multiple different *strategies* in quick succession, e.g. in a 30 second run it may attempt 10 or more different strategies, and may run these in parallel with different priorities [8]. These strategies employ many different options including different saturation algorithms (including Otter and Discount), preprocessing options, literal selection strategies, inference rules, and clause selection heuristics. The portfolio mode is a significant improvement on any single strategy.

We note here that Vampire's portfolio mode includes an additional option relevant to clause selection: the `--nongoal_weight_coefficient` option specifies a multiplier to apply to the weight of non-goal clauses, thus preferring clauses in or derived from the goal in clause selection. Use of this heuristic is orthogonal to the age/weight ratio and we do not investigate its impact any further here.

Related Work. We briefly review the clause selection approaches taken by other solvers. Otter 3.3 [6] selects *either* by age, by weight or manually. Prover9 [7] allows a configurable age/weight ratio. E [12] allows the user to specify an arbitrary number of priority queues and a weighted round-robin scheme that determines

```
    input: init: set of clauses;, a : w age-weight ratio
    var active, passive, unprocessed: set of clauses;
    var given, new: clause;
    active := ∅;
    unprocessed := init;
✓   balance := 0;
    loop
      while unprocessed ≠ ∅
        new:=pop(unprocessed);
        if new = □ then return unsatisfiable;
        if retained(new) then                                  (* retention test *)
          simplify new by clauses in active ∪ passive ;  (* forward simplification *)
          if new = □ then return unsatisfiable;
          if retained(new) then                        (* another retention test *)
            simplify active and passive using new ;  (* backward simplification *)
            move the simplified clauses to unprocessed;
            add new to passive
      if passive = ∅ then return satisfiable or unknown
✓     if balance > 0 then
✓         given := lightest clause in passive;
✓         balance:= balance − w;
✓     else
✓         given := oldest clause in passive;
✓         balance:= balance + a;
      move given from passive to active;
      unprocessed:=infer(given, active);                   (* generating inferences *)
```

Fig. 1. The Discount Saturation Algorithm with AWR clause selection

how many clauses are picked from each queue. The default is to use a combination of age and weight selection, although there is also a complex strategy developed by a genetic algorithm [11]. SPASS [17] uses symbol-counting based clause selection. iProver [3] follows in E in having a number of configurable queues but relies mainly on age and weight heuristics in those queues. The general idea in this paper of a *varying age/weight ratio over time* is applicable to any ratio-based clause selection strategy, and our specific results apply to those that take a ratio between age and weight.

3 Optimising Age/Weight Ratios

We set out to experimentally confirm two assumptions from folklore:

1. The choice of age/weight ratio often has a significant effect on the performance of proof search.
2. There is in general no single best age/weight ratio for a given set of problems.

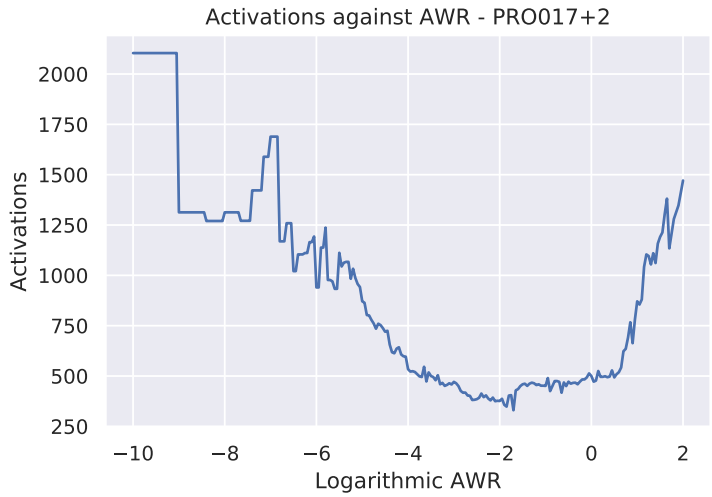

Fig. 2. The number of activations reported by Vampire after successful 1-second runs on a TPTP problem. In between the peaks on either side, the function of L is discontinuous with large peaks and troughs, but follows an approximate trend and settles toward the global optimum. This is typical for the observed set of problems.

These were supported by the work of Schulz and Möhrmann but here we focus on these points and explore them more deeply.

3.1 Logarithmic AWR

Optimising AWR values is more easily achieved if they have a continuous scale. AWR values are mathematically $\mathbb{Q}^+$, the positive rational numbers, but in practice are more easily seen logarithmically: 1:16 is the same distance in terms of the effect on proof search from 1:8 as it is from 1:32. Therefore, we introduce the *logarithmic* AWR L, defined in terms of age A and weight W as

$$L = \log_2 \left(\frac{A}{W} \right)$$

With this definition, AWR can now be seen continuously. As L tends to positive infinity, Vampire selects only by weight, whereas if L tends to negative infinity Vampire selects only by age. $L = 0$ represents the middle ground of a 1:1 age/weight ratio. Note that we have not updated the balancing algorithm used by Vampire to make use of this value (this still requires two numbers) but use the number in experiments when discussing AWR.

3.2 Experiments

As an initial illustrative example of how varying the AWR effects the number of clauses required to be processed before a proof is found consider Figure 2.

Table 1. Relative performance gain, showing the ratio in activations between the best AWR setting for a given problem and another base setting. We compare 1:1 (Vampire's default), 1:5 (the best-behaved from Schulz and Möhrmann), and the worst setting for the problem. Where the problem is not solved at all by the base setting, it is ignored.

	Maximum Gain	Mean Gain (Standard Deviation)	
1:1	133.56	1.26	1.63
1:5	133.67	1.44	1.70
(worst)	222.01	3.95	7.60

This demonstrates the effect that varying AWR can have: a smaller number of activations means that fewer clauses were processed, which in general means that a proof was found faster[1]. On this problem, a good AWR value is over 400% better by this metric than the worst AWR value.

This experiment was repeated on the whole TPTP problem set, excluding problems Vampire does not currently support (e.g. higher-order problems). We ran Vampire for 1 second in default mode with the `discount` saturation algorithm[2] using a sensible set of AWR values (see Table 2) — these are the values used in Vampire's portfolio mode. These tend to favour weight-first over age-first as this has been experimentally shown to be preferable. We then filter out problems not solved by any of these or those solved trivially (e.g in preprocessing). The whole set yielded similar AWR data for 7,947 problems.

The first result is that choosing a good AWR value for a problem is well-rewarded. Table 1 summarises the impact that choosing the best AWR can have. Compared to the default we can perform, on average, 1.26 times fewer activations, which is modest but (as Table 2) shows we also lose just under 10% of problems by choosing the default. It is more relevant to note that there are cases where we can do *much* better by selecting a different AWR value. So choosing a better AWR value *can* go from no solution to a solution and *can* do so faster, but not necessarily. In the worst case (choosing the pessimal AWR value) we can perform almost 4 times as many activations.

The second result is that there is no "best" AWR across this full set of problems. Table 2 shows, for each AWR value, the % of problems solved, the number solved uniquely, the maximum and mean drop in performance. The drop in performance is how many times more activations were required for a proof in the worst/average case compared to the best. No AWR value solves all problems, with the best being 1:5. A ratio of 1:4 produces an unusually well-behaved maximum performance drop. Schulz and Möhrmann found that 1:5 had a similar property, but this might be explained by differences in prover and test

[1] It should be noted that if a small number of clauses are extremely expensive to process it may be slower than a larger number of less-expensive clauses, but in general this is a good heuristic measure for prover performance. It also avoids reproducibility issues involved with using system timing approaches.

[2] The default LRS [10] saturation algorithm can be non-deterministic so we avoid it for such experiments.

Table 2. Per-AWR value results on 1 second runs over 7,947 TPTP problems.

AWR	% Solved	Uniques	Maximum Drop	Mean Drop (Standard Deviation)	
8:1	85.25	16	150.67	1.37	1.98
5:1	86.10	1	122.22	1.33	1.64
4:1	86.93	1	101.44	1.32	1.42
3:1	87.63	2	105.00	1.29	1.41
2:1	88.62	3	112.67	1.27	1.45
3:2	89.83	2	119.89	1.27	1.51
5:4	89.98	4	125.00	1.26	1.55
1	90.56	4	133.56	1.26	1.63
2:3	91.20	9	147.67	1.28	1.79
2	91.68	0	162.67	1.31	1.97
3	91.81	5	190.56	1.37	2.30
4	91.85	3	**17.41**	1.38	0.67
5	92.00	2	133.67	1.44	1.70
6	91.57	1	106.44	1.47	1.46
7	91.49	1	104.89	1.49	1.44
8	91.09	2	101.33	1.53	1.45
10	90.52	1	101.78	1.60	1.53
12	90.00	0	101.67	1.65	1.62
14	89.29	4	103.00	1.70	1.75
16	89.42	5	101.33	1.74	1.76
20	88.61	3	100.89	1.82	1.94
24	88.26	2	101.33	1.89	2.08
28	87.57	2	99.22	1.96	2.24
32	87.01	1	100.00	1.99	2.36
40	86.23	4	98.78	2.09	2.64
50	84.93	1	98.78	2.17	2.88
64	84.17	2	101.22	2.28	3.19
128	81.34	3	107.44	2.57	4.16
1024	73.11	23	222.01	2.83	7.55

environment. It is interesting to note that the extreme AWR values solve fewer problems overall but solve the most uniquely. This is typical in saturation-based proof search: approaches that do not perform well in general may perform well in specific cases where the general approach does not.

In summary, these results confirm the previous assumptions often made in folklore. It should be noted that this is a small experiment (1 second runs in default mode) and the relative performance of different AWR values cannot be generalised, but the general result that they are complementary can.

4 Variable AWR for Vampire

In this section we motivate and define a variable clause selection approach.

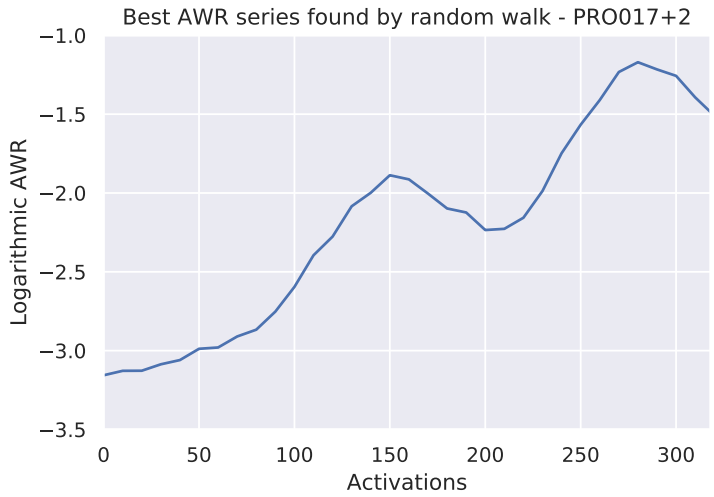

Fig. 3. The AWR series that produced the lowest number of activations on a particular problem, smoothed in order to show the actual effect on proof search. This is a search strategy that a single fixed AWR cannot reproduce.

4.1 The Optimal AWR Over Time

Although choosing a good AWR value is important, this is covered in part by the use of strategy scheduling in which many AWR values are tried in sequence (along with other prover options). Additionally, given that varying the AWR can have such a large impact, it seems likely that a constant AWR fixed for the entire proof search is unlikely to be optimal for any given problem. This can be shown by running Vampire with a randomised sequence of age/weight ratios given by a random walk repeatedly, then finding the best after a large number of repetitions. Applying this method with 10,000 repetitions to the problem seen earlier (`PRO017+2`) yields the example AWR trend shown in Figure 3, which reduces the best number of activations from 330 with a fixed AWR, to 287 with a varying AWR. Unsurprisingly, in ad-hoc experiments on other problems, the best shape is rarely constant. This suggests that implementing other shapes, such as an increasing or decreasing trend, might lead to quicker proofs in the Vampire theorem prover.

4.2 Varying AWR (in Vampire)

We describe how we vary AWR dynamically and how this is achieved in Vampire. In general we would like to describe any possible sequence that the AWR could follow during proof search. However, some details constrain the design space:

1. Changing the AWR too frequently or sharply has little effect, due to the "balancing" algorithm — see Section 1.

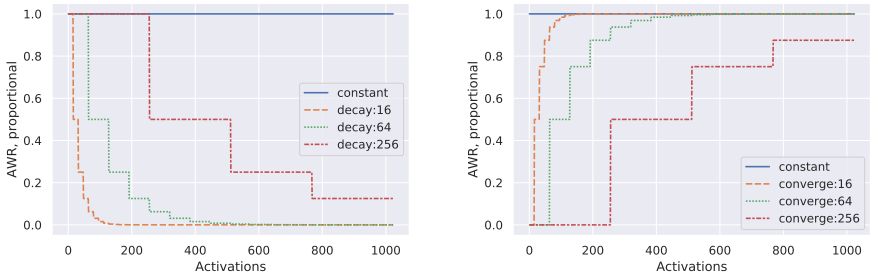

Fig. 4. The new *decay* and *converge* AWR shapes as implemented in Vampire. Different curves exhibit the effect of the AWR shape frequency setting.

2. A general (configurable) *shape* is more likely to be widely applicable than a specific series of data points.
3. The shape must extend naturally to an indefinitely-long proof search.

In this work we select two general trends to explore: a trend away from the original (fixed) AWR toward 1:1 ("decay"), and a trend from 1:1 toward the original setting ("converge"). In our investigations we found that even fluctuating sequences had a general trend and these two fixed trends are reasonable approximations of this. We have chosen to use an 'original' AWR value to move away from/towards as this original AWR has already been shown to be 'useful' in some sense.

Since a simple linear shape does not extend well to indefinite proof search (it is unclear what should happen after either 1:1 or the target AWR is reached), an exponential decay function is used instead. These exponential shapes are further parameterised by an integral *shape frequency* setting, which controls the rate of decay or convergence: every n steps, the difference between the current and the target AWR is halved, rounding where necessary. In future, this might allow the use of repeating patterns such as a sinusoid, hence *frequency*. Figure 4 illustrates rates at which the new configurations converge or decay from the fixed AWR setting for some indicative frequency settings.

Our approach here was restricted by the balancing algorithm used internally, as AWR steps must be discrete and do not take effect immediately. An alternative approach might use an age/weight probability, rather than a ratio, from which age or weight decisions would be pseudo-randomly (but reproducibly) taken with the use of a seeded PRNG. This would permit use of continuous age/weight functions, but would also introduce the possibility of "getting unlucky" in which a clause preferred by the age/weight probability is not selected due to an improbable-but-possible series of PRNG samples.

We introduce two new options: `--age_weight_ratio_shape` can take the values *constant*, *decay*, or *converge* and selects one of the above shapes; and `--age_weight_ratio_shape_frequency` specifies the frequency (rate) or conver-

Table 3. Number of problems solved (top) and unique problems solved (bottom) by various configurations varying AWR, AWR shape, and AWR frequency.

AWR	`constant`	`converge` Frequency					`decay` Frequency				
		1	10	100	1000	Union	1	10	100	1000	Union
Problems Solved											
10:1	7967	7972	7976	8050	8245	*8372*	8448	8441	8380	8169	*8614*
1:10	8575	8565	**8578**	8550	8489	*8778*	8458	8456	8484	8268	*8787*
1:100	8079	8084	8079	8039	8279	*8560*	8454	8484	8537	**8636**	*8907*
1:1000	7276	7279	7297	7418	8133	*8379*	8470	8492	8492	8473	*8873*
Union	*9019*	*9028*	*9016*	*8981*	*8871*	*9194*	*8572*	*8674*	*8725*	*8978*	*9048*
Uniquely Solved											
10:1	1	3	2	3	8	*16*	0	1	2	4	*7*
1:10	1	3	2	1	5	*11*	0	1	0	7	*8*
1:100	1	2	2	7	**9**	*20*	2	0	3	9	*14*
1:1000	1	2	0	0	3	*5*	2	0	0	**12**	*14*
Union	*4*	*10*	*6*	*11*	*25*	*52*	*4*	*2*	*5*	*32*	*43*

gence/decay (default is 100). These are used with the existing `--age_weight_ratio` option (default 1:1) to give a number of new option combinations, which can be used in conjunction with Vampire's portfolio mode pending integration into the strategy schedules. This version of the prover is currently in a separate branch in the main Vampire source repository[3].

5 Experimental Evaluation

We perform two experiments to evaluate the new techniques. The first compares the various options attempting to draw some conclusions about which option values work well together. The second looks at how useful the new options are in the context of portfolio solving. Both experiments use the TPTP (version 7.1.0) benchmark [15] and were run on StarExec [14].

5.1 Comparing New Options

We ran Vampire in default mode (with `discount` saturation algorithm) for 10s whilst varying `age_weight_ratio` and `age_weight_ratio_shape_frequency` for both new values of `age_weight_ratio_shape`. We also ran the constant default AWR of 1:1 as a baseline.

The results are given in Table 3. The results for the different shapes are grouped into columns and then by frequency with rows giving results per AWR value. We separately report the total number of problems solved and those solved uniquely (by a single strategy across all experiments, including the baseline).

[3] `https://github.com/vprover/vampire/tree/awr-shapes`

Table 4. Results for the tested configurations. *Proved* refers to the total number of problems a configuration solved. *Fresh* is the number of problems a configuration solved which were not solved by the baseline. *Uniques* is the number of problems a configuration solved which were not solved by any other configuration. *u-score* is a refined unique score which correlates to a configuration's utility in solving new problems, as used in Hoder *et al.* [2].

Configuration	Frequency	Proved	Fresh	Uniques	u-score
baseline	–	**13,057**	0	1	714.2
converge	1	13,039	24	3	714.3
converge	5	13,029	27	1	709.5
converge	10	13,028	35	5	714.3
converge	50	13,015	45	5	712.8
converge	100	12,976	51	1	705.9
converge	500	12,895	63	4	698.3
converge	1000	12,837	52	0	688.6
converge	5000	12,775	53	1	682.4
converge	10000	12,751	53	0	678.7
decay	1	12,698	48	1	673.6
decay	5	12,702	51	1	674.9
decay	10	12,698	48	1	674.2
decay	50	12,712	49	2	679.1
decay	100	12,726	46	1	678.8
decay	500	12,795	29	1	685.5
decay	1000	12,860	29	2	692.6
decay	5000	12,982	16	2	707.1
decay	10000	13,002	7	0	706.3
converge	(combined)	13,167	117	41	–
decay	(combined)	13,106	93	17	–

In all cases, for a given start AWR value, varying the AWR over time outperformed keeping it constant. As the new strategies always include some reasoning using `1:1` AWR this suggests that all benefit from *some* fair reasoning during proof search and some biased reasoning. The best combination of options overall was decaying from an initial age/weight of 1:100 with frequency 1000. However, in general, strategies that converged performed better than those that decayed, both in terms of total problems solved and in terms of unique solutions. Longer frequencies tended to do better, suggesting that more time at the intermediate AWR values is preferable. In total, 28 out of a total of 36 strategies contributed some unique solutions. This shows that these new options are complementary in general.

5.2 Contribution to Portfolio

Our next experiment aims to answer the question "How much can the portfolio mode of Vampire improve using these new options?". To address this we use the portfolio mode used in the most recent CASC competition [16] as a baseline

Table 5. Total number of problems solved compared to other solvers.

Solver	Total solved	Uniquely solved	
		All	Excluding Vampire (old)
Vampire (old)	13,057	0	-
Vampire (new)	13,191	54	1030
E	10,845	190	190
iProver	8,143	215	215
CVC4	9,354	501	502

and run the new options on top of this. Note that this portfolio mode contains techniques completely unrelated to the age/weight ratio, e.g. finite model building [9], as well as other options related to clause selection, e.g. non-goal weight coefficient and set-of-support.

Vampire first ran to establish baseline performance in the given portfolio mode on all problems in TPTP, with a wallclock time limit of 300 seconds. We then forced the new options on top of the portfolio mode, using the existing AWR values in the various strategies as the starting point. The purpose is to gauge what impact adding such options to a new portfolio mode could have. In this experiment we are mainly hoping to find new solved problems and identify new strategies that could be added to a portfolio mode. Therefore, it makes sense to consider the union of all experiments ran.

Overall, the baseline solved the most problems (13,057). No experimental configuration improved on this figure, but some problems not solved by baseline were solved by the new configurations, and some entirely new problems were solved. The union of all *converge* and *decay* configurations improved on the baseline, with 13,167 and 13,106 solved problems respectively.

Figure 4 shows the performance in terms of solved problems of all the configurations tested. These data show that configurations which are more similar to the baseline (i.e. slow decay or fast convergence) achieve more similar performance, as expected. In total, 134 problems were solved by the new configurations that were not solved by the baseline. This is an impressive result — it is rare to be able to improve portfolio mode by this many new problems with a single new proof search option.

We also report the *u-score* computed by giving $1/n$ points per problem solved where n is the number of strategies solving a problem [2]. This gives a measure of contribution per strategy. To consider new options for extending the portfolio mode we will consider options from largest u-score to smallest (and only those with unique solutions overall).

Finally, two problems were solved which were marked with an "Unknown" status in the TPTP headers. Only converging with frequency 50 solved `SET345-6` and only decaying with frequency 1 solved `LAT320+3`.

5.3 Comparison with other Solvers

To place these results in context, we compare the overall number of problems solved by our new strategies with the results of other solvers, using their 2018 CASC competition mode. These results are from 300-second runs in identical conditions and are given in Table 5. In this table *Vampire (old)* stands for the 2018 competition version whilst *Vampire (new)* stands for the union of all problems solved by new options in the previous section. Between them, the two versions of Vampire solve 1,030 problems uniquely. 54 unique problems found in the previous section remain unique when compared to other competitive solvers.

6 Conclusions and Future Work

Clause selection is a key part of any saturation-based theorem prover and age/weight ratios have a significant effect on the performance of proof search in the Vampire theorem prover. We have supported the known folklore that there is no clear optimal age/weight ratio. Further, we have demonstrated that varying the age/weight ratio over time *during* proof search can improve further on an optimal, but fixed age/weight ratio in terms of the number of activations. Experiments within Vampire on the TPTP benchmark suggest that these *age/weight shapes* show promise for future developments in this novel approach to proof search. Indeed, including our relatively simple shapes already leads to significant performance gains.

Future directions for research include trying a greater number of "shapes" (such as repeating patterns), other approaches for parameterising these shapes, a pseudo-random approach to age/weight instead of the balancing algorithm, and integration of the new approaches into existing strategy schedules.

References

1. Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. Discount-a distributed and learning equational prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997.
2. Kryštof Hoder, Giles Reger, Martin Suda, and Andrei Voronkov. Selecting the selection. In *International Joint Conference on Automated Reasoning*, pages 313–329. Springer, 2016.
3. K. Korovin. iProver—an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298, 2008.
4. Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.
5. William McCune. Otter 2.0. In *International Conference on Automated Deduction*, pages 663–664. Springer, 1990.
6. William McCune. Otter 3.3 reference manual. *arXiv preprint cs/0310056*, 2003.

7. William McCune. Release of prover9. In *Mile High Conference on Quasigroups, Loops and Nonassociative Systems, Denver, Colorado*, 2005.
8. Michael Rawson and Giles Reger. Dynamic strategy priority: Empower the strong and abandon the weak. In *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning co-located with Federated Logic Conference 2018 (FLoC 2018), Oxford, UK, July 19th, 2018.*, pages 58–71, 2018.
9. Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 323–341, 2016.
10. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. 36(1-2):101–115, 2003.
11. Simon Schäfer and Stephan Schulz. Breeding theorem proving heuristics with genetic algorithms. In *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, pages 263–274, 2015.
12. S. Schulz. E — a brainiac theorem prover. 15(2-3):111–126, 2002.
13. Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 330–345, 2016.
14. Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec, a cross community logic solving service. https://www.starexec.org, 2012.
15. G. Sutcliffe. The TPTP problem library and associated infrastructure, from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
16. Geoff Sutcliffe and Christian B. Suttner. The CADE-18 ATP system competition. *Journal of Automated Reasoning*, 31(1):23–32, 2003.
17. C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.