Dynamic Strategy Priority: Empower the Strong and Abandon the Weak.

Michael Rawson and Giles Reger

University of Manchester, Manchester, UK

Abstract. Automated theorem provers are often used in other tools as black-boxes for discharging proof obligations. One example where this has enjoyed a lot of success is within interactive theorem proving. A key usability factor in such settings is the time taken for the provers to complete. Automated theorem provers typically run lists of proof strategies sequentially, which can cause proofs to be found more slowly than necessary if the ordering is suboptimal. We show that it is possible to predict which strategies are likely to succeed while they are running using an artificial neural network. We also implement a run-time strategy scheduler in the Vampire prover which utilises a trained neural network to improve average proof search time, and hence increases usability as a black-box prover.

1 Introduction

Modern automated theorem provers (e.g. E [30], iProver [12], Vampire [13], CVC4 [7]) for first-order logic rely on *portfolio modes* [25], which utilise tens to hundreds of distinct *strategies*. Of these strategies, only a few might solve a hard problem, often rapidly. Typically, a portfolio of strategies has a pre-defined execution order: the prover will process strategies in this order, running each until a winning strategy is found or the prover runs out of time or strategies.

Portfolio modes are important as, in practice, there is no best strategy. Furthermore, it is uncommon that two hard problems are efficiently solved by the same strategy. However, portfolio execution is not without problems: deciding the optimal ordering and time allocation is hard in general [24], and produces overly-rigid, brittle engineering when applied to specific domains, such as those found in the TPTP problem set [32]. Moreover, for any particular problem, some lengthy strategies that are successful on other problems are doomed to failure from the outset — as illustrated in Figure 1 — but are left to run unchecked by the prover, wasting time that could be spent on more productive strategies.

This work makes two contributions. We first demonstrate correlation between trends in dynamic properties of proof search, and the pending success or failure of a strategy (Sections 4, 5). We then utilise this to implement strategy scheduling, prioritising those strategies most likely to succeed (Section 6). This approach differs from previous work [30,20,16] which attempts to predict successful strategies *a priori* from static features of the input problem; instead

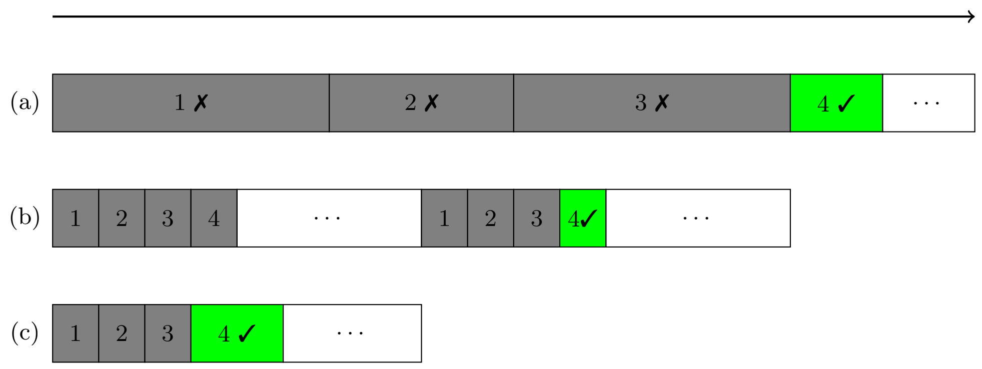

Fig. 1: Several strategy schedules and their effects. (a) Failing strategies might block a successful strategy. (b) Round-robin scheduling can help mitigate this problem. (c) This approach is improved if promising strategies run for longer.

we tip running strategies for success based on dynamic, run-time features and use this information to make decisions at runtime. Our experiments (Section 7) show that guiding scheduling in this way can significantly speed up portfolio-based approaches. Before introducing these contributions and results we first present necessary background in Sections 2 and 3.

2 Strategies in Vampire

Vampire is a first-order superposition theorem prover. This section reviews its basic structure and components relevant to the rest of this paper. We will use the word *strategy* to refer to a set of configuration parameter values that control proof search and *proof attempt* to refer to a run of the prover using such a strategy.

2.1 Input and preprocessing

Vampire accepts problems in first-order logic with equality and a pre-defined set of first-order theories (e.g. arithmetic, arrays, datatypes). Vampire typically works with problems in the TPTP format [32] but also accepts problems in SMT-LIB [3].

Problems are parsed and transformed into a set of input clauses used by the saturation algorithm. This process involves clausification and a number of preprocessing steps. These preprocessing steps can alter certain properties of the problem e.g. its size and distribution across the signature, sometimes significantly: for example, the E.T. system [10] implements a pre-processor for E which selects small sets of axioms from a larger set, for large-theory reasoning. This means that any prediction method that relies solely on the input problem (rather than dynamic characteristics such as post-preprocessing properties) will find it harder to predict the best parameters for proof search.

2.2 Saturation algorithms

Superposition provers such as Vampire use *saturation algorithms with redundancy elimination*. They work with a search space consisting of a set of clauses and use a collection of generating, simplifying and deleting inferences to explore this space. Generating inferences, such as superposition, extend this search space by adding new clauses obtained by applying inferences to existing clauses. Simplifying inferences, such as demodulation, replace a clause by a simpler one. Deleting inferences, such as subsumption, delete a clause, typically when it becomes redundant (see [2]). Simplifying and deleting inferences must satisfy this condition to preserve completeness.

The goal is to *saturate* the clause set with respect to the inference system. If the empty clause is derived then the input clauses are unsatisfiable. If no empty clause is derived and the search space is saturated then the input clauses are guaranteed to be satisfiable *only if* a complete strategy is used. A strategy is complete if it is guaranteed that all inferences between non-deleted clauses in the search space will be applied. Vampire includes many incomplete strategies as they can be very efficient at finding unsatisfiability.

All saturation algorithms implemented in Vampire belong to the family of *given clause algorithms*, which achieve completeness via a fair *clause selection* process that prevents the indefinite skipping of old clauses. These algorithms typically divide clauses into three sets, *unprocessed*, *passive* and *active*, and follow a simple *saturation loop*:

1. Add non-redundant *unprocessed* clauses to *passive*. Redundancy is checked by attempting to *forward simplify* the new clause using processed clauses.
2. Remove processed (passive and active) clauses made redundant by newly processed clauses, i.e. *backward simplify* existing clauses using these clauses.
3. Select a given clause from *passive*, move it to *active* and perform all generating inferences between the given clause and all other active clauses, adding generated clauses to *unprocessed*.

Later we will show how iterations of this saturation loop from different proof attempts can be interleaved. Vampire implements three saturation algorithms:

1. *Otter* uses both passive and active clauses for simplifications.
2. *Limited Resource Strategy (LRS)* [28] extends Otter with a heuristic that discards clauses that are unlikely to be used with the current resources, i.e. time and memory. This strategy is incomplete but also generally the most effective at proving unsatisfiability.
3. Discount uses only active clauses for simplifications.

A recent development in Vampire is AVATAR [37,26] which integrates with the saturation loop to perform clause splitting. The general idea is to use a SAT solver to select a subproblem by naming each clause sub-component by a propositional variable, running a SAT solver on these abstracted clauses, and using the subsequent propositional model to select components to include in proof search.

At the end of each iteration of the loop we check whether the underlying subproblem has changed. AVATAR can occasionally make loops run a little longer, but no more than other steps such as backward subsumption. Otherwise, the notion of saturation loop remains the same when using AVATAR.

There are also other proof strategies that fit into the above loop format and can be interleaved with superposition based proof attempts. For example, instance generation [8] saturates the set of clauses with respect to the instance generation rule and finite model finding [27] iteratively checks larger model sizes (these loops tend to be much longer than those from other algorithms).

2.3 Strategies in Vampire

Vampire includes more than 50 parameters. By only varying parameters and values used by Vampire at the last CASC competition, we obtain over 500 million strategies. These parameters control

- Preprocessing steps (24 different parameters)
- The saturation algorithm and related behaviour e.g. clause selection
- Inferences used (16 different kinds with variations)

Even restricting these parameters to a single saturation algorithm and straightforward preprocessing steps, the number of possible strategies is vast. For this reason, Vampire implements a portfolio *CASC mode* [13] that categorises problems based on syntactic features and attempts a sequence of approximately 50 strategies over a five minute period (this number can vary significantly). These strategies are the result of extensive benchmarking and have been shown, experimentally, to work well on unseen problems i.e. those not used for training.

The syntactic features used by the current portfolio mode are coarse-grained and include the rough size of the problem (e.g. bucketed into tiny, small, medium, large, huge), the presence of features such as equality or certain theories, and whether the problem is effectively propositional or horn. Not all combinations of these are considered. Portfolio mode is created by considering a set of training data over a list of strategies and attempting to cover as much of it as possible by splitting the set of problems into smaller groups based on these features until all solutions fit into a given time limit. This process is greedy and places the strategy that solves the most problems during training first and then the next best strategy after removing the problems solved by the first solver and so on.

3 Machine Learning and Theorem Proving

In this section we review the necessary background in machine learning in the context of theorem proving, in particular neural networks. Other technologies exist which are capable of similar performance, but neural networks produced the best combination of convenience and performance for our purposes.

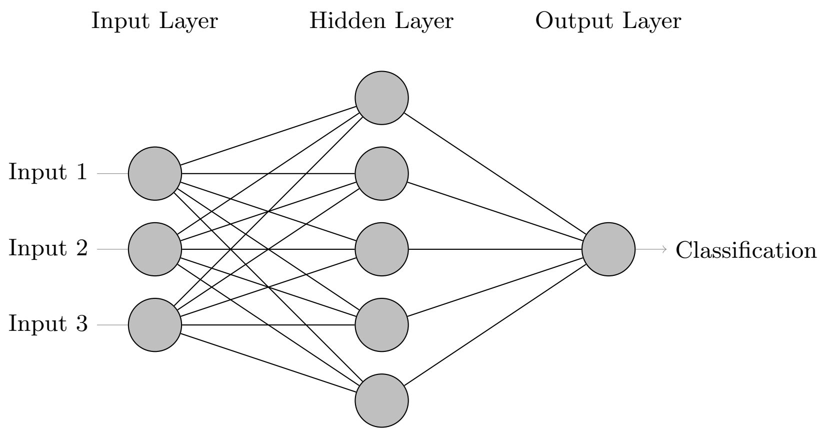

Fig. 2: A neural network with 3 inputs, a single densely-connected hidden layer, and a single output neuron. This is a type of network that might be used for simple binary classification tasks. A version of this architecture with more input and hidden neurons is used for this work.

3.1 Classification with neural networks

Binary classification problems can be heuristically solved by means of *supervised learning* methods. A binary classification problem is, informally, the statement "given input X, does X belong to class A or class B", where X is a member of the target domain $\mathcal{D}$, and A, B are disjoint subsets of $\mathcal{D}$ such that $\mathcal{D} = A \cup B$. Supervised learning methods attempt to solve binary classification problems by starting with a neutral, untrained heuristic for classifying X into A or B, then changing their internal state to improve the heuristic by looking at examples of A and B ("training"). Typically, supervised learning methods do not work on X directly, but from a set of "features" *derived* from X. Engineering features, or choosing which of a set of supplied features to use, is a field of study in itself.

A neural network consists of a collection of artificial "neurons", wired together with connections into a *neural network architecture* (see Figure 2 for an example). Each neuron n computes a real-valued function f_n, which may be tuned with real-valued parameters: every neuron has a *bias* b_n, and *weight vector* $\mathbf{w}_n$.

f_n is defined on an input vector $\mathbf{x}$ as

$$f_n(\mathbf{x}) = A\left(\mathbf{w}_n \cdot \mathbf{x} + b_n\right)$$

i.e. a weighted sum of all connected input neurons' activation functions, plus the bias of the neuron, all fed through an *activation function* $A(x)$. The choice of activation function is dependent on both the position of the neuron in the architecture, and on the target domain: a common choice is the sigmoid function $\sigma(x)$, as shown in Figure 3.

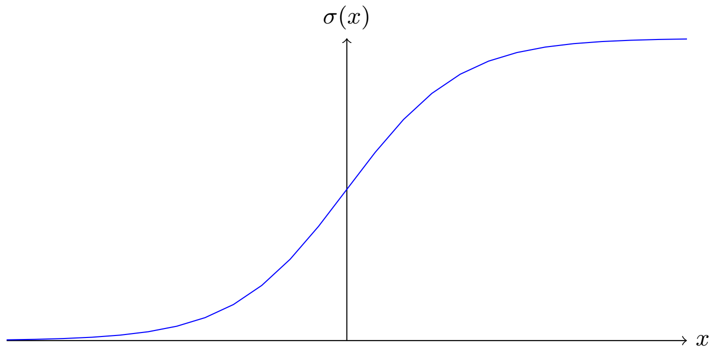

Fig. 3: The sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. This function "squeezes" the output to the range $[0, 1]$.

Neural network parameters may be *trained* by a variety of different techniques (randomly, neuro-evolution methods, backpropagation methods, *etc.*). We used a backpropagation method, which, given a training input X and correct output y, tunes the network parameters by propagating an error signal backwards through the network to obtain updated parameters for each neuron, hence reducing the error signal for that example [29]. Repeating this process for multiple training examples can produce very good results, but frequently only with the correct combination of learning algorithm, starting weights, network architecture, and training examples.

3.2 Related work

Machine learning has potential for significant effect in this area: despite the computational difficulties experienced by theorem provers, frequently proofs have human (or otherwise regular) structure, which may be exploited by the use of learning methods. The authors are aware of work in the fields of premise selection [9,15,35,38], static strategy selection [4,18,17], and more recently, direct proof guidance [36,11,19]. However, we are not aware of any previous work in the area of strategy selection at runtime for conventional theorem provers. The closest area of work is that of static strategy selection where work, with the exception of Bridge et al. [4] focusses on static properties of the input problem rather than dynamic properties of the proof search space. Bridge considered various dynamic features of the search space after 100 steps of the saturation algorithm in the default mode (of the E theorem prover [30]).

So far, machine learning techniques have typically used classical machine-learning techniques such as naïve Bayesian methods, SVMs [34], and decision trees [23]. However, some newer work has begun to utilise recently-developed deep-neural-network methods [9,38,19].

4 Feature collection

This section discusses which features are extracted from Vampire for prediction and how they are extracted. This information is firstly required to identify a correlation and will then be used to train a predictor.

Modifying Vampire to log execution data (such as memory usage, or more specific metrics such as the number of generated clauses) for different strategies obtained from its primary portfolio mode[1] is straightforward, but some data-collection decisions were made:

- Only numerical data immediately available in the prover was collected, but there is scope here for both non-numeric and derivative data sources, which may provide greater insight into the proof state in future work. Suppose that quantities A and B are measured directly, but are more often discussed in terms of categories C that all A, B pairs fall into. Data could then include C as well, or instead of, A and B, embedding C into the neural network via a one-hot encoding.
- Data was collected at intervals of a fixed number of internal resolution steps (in the experiments presented in this paper, this value was 10). This may not necessarily correspond to fixed time intervals, as each step may be more or less expensive, depending on the strategy and the problem.
- All available data was collected, even if it emerged to be constant or unhelpful. This allowed an agnostic approach to learning in which the neural network training procedure selected relevant features.

In all, 376 features are recorded, including the number of active clauses, passive clauses, and subsumptions, for instance. The execution traces produced are difficult to work with, however: some are short or non-existent, others are extremely lengthy. The mean execution trace length was 536 (standard deviation: 1208) with the longest trace being 9971 recorded steps long. Some feature values also have an extremely high variance. To deal with these problems, a post-processing step is applied. For each feature, the mean over the entire data set is then mapped to 0, and the dataset is scaled to unit variance. Data that are too short (fewer than 10 steps) are discarded (the strategy likely did not take very long in any case). The remaining data is then sliced into 10 evenly-sized "buckets", then an average of each bucket is taken to produce 10 values for every trace, and hence a fixed data size.

However, even now these data are not representative of the classification problem desired: these traces show *completed* runs of Vampire, whereas the classifier will be used to predict the success or otherwise of runs of Vampire that are still in progress. Hence, we take "snapshots" at various stages (in these experiments, at every quarter) of the trace, discarding the rest of the data, then post-process the remaining trace as described above. Conveniently, this also provides 4 times the original number of training examples. An example post-processed trace is shown in Figure 4.

[1] CASC-mode, a portfolio designed for the CASC competition [33].

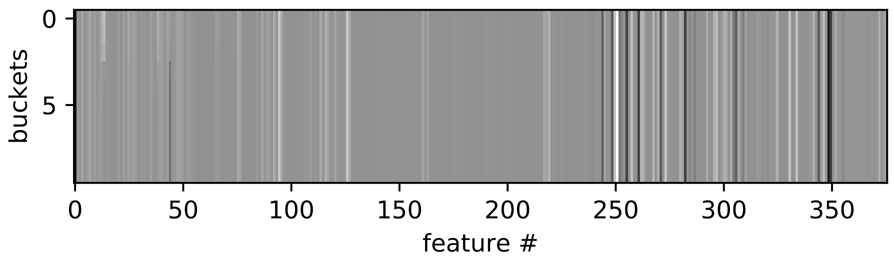

Fig. 4: An example trace, displayed as a colourmap, after post-processing.

5 Predicting successful strategies

Being able to predict which proof search attempts will succeed in time, and which will fail, based solely on information in the execution trace may seem unlikely. However, it is known that the "slowly-growing search space" maxim, which states that strategies which minimise the number of derived clauses over time are more likely to succeed, is an effective heuristic for finding good strategies in saturation-based theorem proving [24]. Since the data we use includes the number of derived clauses, among many other features, it appears more plausible that this approach might work at least as well as the slow-growth heuristic alone. Engineering a prediction algorithm that attempts to partition traces into "succeeding" and "failing" classes is possible with the use of modern machine-learning techniques. Conveniently, these methods do not usually produce a binary output, but instead some $f(\mathbf{X}) \in [0, 1]$ which might be seen as the "level of confidence" in success of the trace, $\mathbf{X}$. This success score can be used to apply an ordering to executing strategies, allowing "smart" dynamic scheduling of strategies.

In particular, we evaluated (using the *Keras* [5] backend-agnostic neural network suite and the *scikit-learn* [22] utilities)

1. A simple neural network with one input for each datum in the trace and a single hidden layer.
2. A convolutional network [14] which performs a 1-dimensional convolutional pass along the time axis for each feature before the hidden layer.
3. A recurrent network [6], feeding the time series into a gated recurrent unit before processing.

Results for this classification task are shown in Figure 5. The consistent level of accuracy achieved is encouraging. Both of the more-advanced classifiers performed better than the simple neural network. However, the simple network was chosen for integration into Vampire for implementation simplicity, and for performance reasons — it is "good enough".

Method	Mean Accuracy	Standard Deviation
Simple neural network	81.5%	2.0%
Convolutional network	82.4%	3.1%
Recurrent network	83.9%	1.9%

Fig. 5: k-fold cross-validation classification accuracy on a balanced dataset of around 10,000 (total) succeeding and failing execution traces. $k = 5$.

6 Intelligent scheduling for Vampire

We show that this abstract predictor can be used in a concrete implementation for the Vampire prover. In the modified prover, it is used to run several strategies from Vampire's portfolio in a modified scheduler: strategies self-report their own execution data to the supervisor process and halt for re-evaluation at regular intervals. When a strategy halts, the scheduler then decides whether to re-schedule the strategy for some more time, or to swap it out for a different strategy. The algorithm used is as follows, taking as input a set of strategies to run:

1. Initialize an empty "run pool" of processes, with a maximum size equal to the desired number of workers (e.g. CPU cores available). Also initialize an empty priority queue of paused processes.
2. Whenever the pool is under capacity, take the following steps:
   (a) If the best process in the paused queue has a priority greater than the static priority p_{static}, wake it and move it into the running pool. In tests, a static priority of around 0.5 appeared to work best.
   (b) Otherwise, take a new strategy from the input set and start a new process to run that strategy.
   (c) If the input set is empty, take the process from the queue regardless of its priority.
3. When a strategy pauses:
   (a) Remove the process from the pool.
   (b) Re-evaluate the process priority using the neural network and the data it provided.
   (c) Insert the process into the queue with the computed priority.
4. When a strategy terminates, check if it succeeded. Otherwise, remove it from the pool.
5. If the pool, the queue, and the set of input strategies are all depleted, all strategies have failed. Exit.

To embed the neural network into Vampire, we took the trained network weights from our Python-based experiments, and generated a C source file with these weights included as a large array. The neural network's architecture was then re-implemented manually in C++, using the network weights compiled into the new program. This approach had several advantages: while perhaps not as efficient as an architecture such as TensorFlow [1] (which may use graphics

Table 1: Results for the three variations.

	Solved	New	Unique	Average time (wall)	Average time (cpu)
baseline	12,899	-	420	3.03s $\pm$ 9.91s	10.53s $\pm$ 33.60s
no-prediction	11,827	56	33	2.86s $\pm$ 8.33s	10.38s $\pm$ 32.24s
prediction	12,425	111	88	2.59s $\pm$ 9.55s	8.74s $\pm$ 31.03s

hardware to accelerate computations), our approach is *reasonably* efficient, but is also low-latency, does not incur any additional dependencies, and does not add significantly to program start-up time.

7 Experimental results

We evaluate whether our prediction results carry through to improved performance in Vampire.

Experimental setup We take all relevant problems for Vampire from TPTP 6.4.0 (17,281 problems in total) and run three variations of Vampire 4.2.1's CASC portfolio mode:

– *baseline* was the standard portfolio mode
– *no-prediction* uses the dynamic scheduling architecture without prediction. This effectively produces a round-robin scheduling of strategies.
– *prediction* uses the trained neural network to predict whether a strategy will be successful as previously described.

The default 3GB memory limit and 300s time limit from the standard portfolio mode were kept but in addition each variation was run in a *multicore* setup using 4 cores to distribute strategies over.Experiments were run on the StarExec cluster [31] where each node contains an Intel Xeon 2.4GHz processor.

Results Figure 6 illustrates the overall results by plotting the number of problems solved against time taken. Table 1 gives some raw numbers. In this table *New* refers to the number of problems solved by this variation that were not solved by *baseline*, *Unique* refers to the number of problems solved by this variation and not by the other two, and the average solution times are given for wall-clock and cpu-time with standard deviation.

Table 2 gives further statistics comparing the two scheduling variations with the baseline. For the purposes of this table we only compare the results on problems that both the given variation and baseline solve and where the difference between solution times is greater than 1 second. This second part is important as it allows for small variations in solution times due to natural non-determinism[2].

[2] This allowance is often not made but it can heavily skew results. Without it all variations look almost identical as the cases where variations behave in the same way dominate. This does not indicate that the variations are not an improvement,

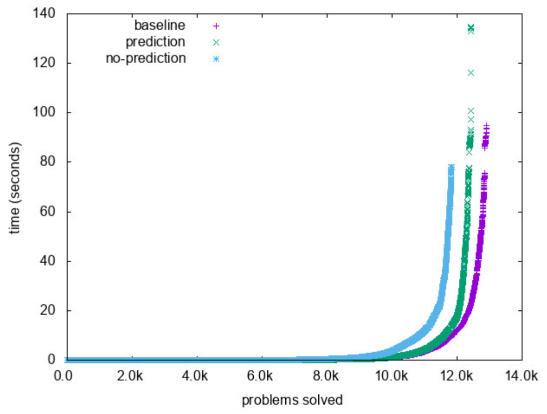

Fig. 6: Plot of problems solved against time for the three variations.

Table 2: Amount of speedup or slowdown compared to the baseline solver.

	no-prediction		prediction	
	wall	cpu	wall	cpu
Average Speedup	1.54	6.90	1.94	6.24
Number of times better	451	788	650	1,297
Number of times worse	1,895	2,518	642	940
Average Speedup (when better)	7.02	28.19	3.47	10.46
Average Slowdown (when worse)	29.02	78.82	18.83	44.52

Speedup is a multiplication factor (a speedup of 2 means a proof was found in half the time), similarly for slowdown. Better means that the given variation was faster, whereas worse means that the variation was slower.

Discussion An immediate observation is that the overall number of problems solved is slightly worse for the variations performing scheduling. An immediate explanation for this could be that the strategies in the portfolio mode might still be quite fragile i.e. their performance might be degrading with the small overhead of context-switching and additional memory contention. Further experiments could explore this by using more generic strategies with longer individual time limits (many strategies in CASC mode run for less than 1 second). However, the overall number of problems solved is still high enough to make the result useful. The new problems solved by the scheduling variations could also be attributed to this non-determinism, suggesting again that more effort should be spent to provide a broader set of complementary strategies to choose from. We note that one explanation for this second effect could have been if the strategy schedule

but there are many cases where there is no difference. The most likely explanation for this is that solutions are quick and no scheduling is required e.g. if the problem is solved during preprocessing.

was longer than the time given and prediction moved a strategy into the allowed time. This is a behaviour we might expect for short time limits but for this experiment all strategies were run.

The average solution times are improved with the two variations and the variation using prediction achieves the best solution time on average. When looking at the more detailed results of Table 2 we see that in the majority of cases the prediction variant was faster than the baseline, which was not the case when no prediction was applied. Furthermore, the impact of getting it wrong was larger without prediction (i.e. the slowdown is bigger).

8 Conclusions and future work

The aim of these experiments was to improve Vampire's overall performance, if not in the number of total theorems proved, but in the average time taken to prove problems. This approach has been shown to produce a significant increase in speed without an excessive penalty in the number of problems solved.

There are several routes that could be explored in order to further improve performance. As well as improving predictor performance by use of more sophisticated data curation, processing, and machine-learning techniques, it may also be possible to improve the naïve scheduling algorithm. Further research might include designing scheduling algorithms which keep predictions as up-to-date as possible, maximise processor utilisation, minimise memory usage/swapping, reduce context-switching overhead, or even minimise the number of required calls to the prediction algorithm.

This form of optimisation for Vampire is relatively novel: historically the aim of the team has been to prove as many theorems as possible, rather than to improve the speed of moderately-hard problem solving. As immediate impact, these developments may be useful in improving Vampire's performance in the new SLH division in the CASC competition, as well as improving the overall usability of ITP via quicker "hammer" results, such as those reported via Sledgehammer [21].

References

1. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.
2. L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.
3. Clark Barrett, Cesare Tinelli, et al. The smt-lib standard: Version 2.0.
4. James P Bridge, Sean B Holden, and Lawrence C Paulson. Machine learning for first-order theorem proving. *Journal of automated reasoning*, 53(2):141–172, 2014.
5. François Chollet et al. Keras. `https://keras.io`, 2015.

6. Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. In *International Conference on Machine Learning*, pages 2067–2075, 2015.
7. Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A tour of CVC4: how it works, and how to use it. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, page 7, 2014.
8. H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Proc. LICS'03*, pages 55–64, 2003.
9. Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, Francois Chollet, and Josef Urban. DeepMath — deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.
10. Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System description: Et 0.1. In *International Conference on Automated Deduction*, pages 389–398. Springer, 2015.
11. Cezary Kaliszyk and Josef Urban. FEMaLeCoP: Fairly efficient machine learning connection prover. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 88–96. Springer, 2015.
12. K. Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.
13. Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.
14. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
15. Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: machine learning for sledgehammer. In *International Conference on Interactive Theorem Proving*, pages 35–50. Springer, 2013.
16. Daniel Kühlwein, Stephan Schulz, and Josef Urban. E-MaLeS 1.1. In *International Conference on Automated Deduction*, pages 407–413. Springer, 2013.
17. Daniel Kühlwein, Stephan Schulz, and Josef Urban. E-MaLeS 1.1. In *International Conference on Automated Deduction*, pages 407–413. Springer, 2013.
18. Daniel Kühlwein and Josef Urban. Males: A framework for automatic tuning of automated theorem provers. *Journal of Automated Reasoning*, 55(2):91–116, 2015.
19. Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.
20. William McCune and Larry Wos. Otter — the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.
21. Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 1, 2010.
22. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
23. J Ross Quinlan. Learning efficient classification procedures and their application to chess end games. In *Machine Learning, Volume I*, pages 463–482. Elsevier, 1983.

24. Giles Reger and Martin Suda. Measuring progress to predict success: Can a good proof strategy be evolved? *AITP 2017*, 2017.
25. Giles Reger, Martin Suda, and Andrei Voronkov. The challenges of evaluating a new feature in Vampire. In *Vampire Workshop*, pages 70–74, 2014.
26. Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In P. Amy Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 399–415, Cham, 2015. Springer International Publishing.
27. Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 323–341. Springer International Publishing, 2016.
28. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comp.*, 36(1-2):101–115, 2003.
29. David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.
30. Stephan Schulz. E — a brainiac theorem prover. *AI Communications*, 15(2, 3):111–126, 2002.
31. Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec, a cross community logic solving service. `https://www.starexec.org`, 2012.
32. 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.
33. Geoff Sutcliffe and Christian Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.
34. Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.
35. Josef Urban. MaLARea: a metasystem for automated reasoning in large theories. *ESARLT*, 257, 2007.
36. Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP machine learning connection prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 263–277. Springer, 2011.
37. Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer International Publishing, 2014.
38. Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2783–2793, 2017.