Vampire

Alexandre Riazanov and Andrei Voronkov

Computer Science Department, University of Manchester

1 General description

Vampire is a resolution-based theorem prover for first-order classical logic. The current version implements ordered binary resolution with the set-of-support strategy and ordered hyperresolution. The competition version will have equality rules.

Vampire owes many of its design ideas to Otter [5]. It uses a saturation-based algorithm similar to that described in [3], but implemented more efficiently. Some planned (but not yet implemented) extensions of Vampire are directed toward the ideas underlying the design of SPASS [9].

2 Compiling and indexing

The efficiency of Vampire is due to a combination of compilation and indexing used to implement all costly operations: resolution, hyperresolution, superposition, backward and forward subsumption, and some simplification rules. All indexes used in Vampire implement so-called "perfect filtering" [2, 6]: they find exactly all elements in an index to which the operation is applicable. In addition, our algorithms for index-based retrieval compute the results of the operation on all these elements (for example, the most general unifiers when the operation is resolution), as proposed in [8].

We use code trees [8] for forward subsumption, path indexing [7] for backward subsumption and a variant of discrimination trees [4] for resolution. To give the reader a flavor of how we combine compilation and indexing, we describe in some detail how hyperresolution is implemented.

To implement hyperresolution efficiently in presence of indexing one has to solve the problem of simultaneous retrieval of unifiable terms. This problem can be formulated as follows: given terms $t_1, \ldots, t_n$ and indices $I_1, \ldots, I_n$, retrieve all possible tuples of terms $s_1, \ldots, s_n$ from the indices $I_1, \ldots, I_n$ respectively such that $\langle t_1, \ldots, t_n \rangle$ is unifiable with $\langle s_1, \ldots, s_n \rangle$, and for each such tuple return a most general unifier.

For solving the problem we use a very simple algorithm based on repetitive application of a retrieval operation *Retrieve* for *one term*: given an index I and a term t, $Retrieve(I, t)$ generates a sequence of all pairs s, σ such that the term s is in the index and σ is the most general unifier for s and t.

The algorithm *SimRetrieve* for *simultaneous* retrieval works as follows:

1. If the tuple of terms consists of one term t we simply apply the operation Retrieve to t and the corresponding index:

$$SimRetrieve(\langle I\rangle, \langle t\rangle) = \{\langle\langle s\rangle, \sigma\rangle \mid \langle\langle s\rangle, \sigma\rangle \in Retrieve(I, t)\}.$$

2. To compute $SimRetrieve(\langle I_1, \ldots, I_n\rangle, \langle t_1, \ldots, t_n\rangle)$, we first arbitrarily choose $t_i, i \in 1, \ldots, n$. Then we enumerate pairs from $Retrieve(\langle I_i, t_i\rangle)$ and for every such pair $\langle s, \sigma\rangle$ compute

$$SimRetrieve(\langle I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n\rangle, \langle t_1\sigma, \ldots, t_{i-1}\sigma, t_{i+1}\sigma, t_n\sigma\rangle).$$

   $SimRetrieve(\langle I_1, \ldots, I_n\rangle, \langle t_1, \ldots, t_n\rangle)$ will consists of all pairs

$$\langle \sigma \circ \theta, \langle s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n\rangle\rangle$$

   such that

$$\begin{array}{l}\langle\theta, \langle s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n\rangle\rangle \in \\ \quad SimRetrieve(\langle I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n\rangle, \langle t_1\sigma, \ldots, t_{i-1}\sigma, t_{i+1}\sigma, t_n\sigma\rangle).\end{array}$$

To improve efficiency of simultaneous retrieval we use a version of *Retrieve* which exploits advantages of *compilation*. A call of *Retrieve* taking as the argument a term of the form $t\theta$, where t is one of the initial terms t_i and θ is a substitution collected at the previous steps of retrieval, uses a compiled version of the retrieval algorithm *specialized* for this term $t\theta$.

A heuristic is used to choose the next term t_i at Step 2 of the algorithm above. We choose the term which seems to have less unifiers in the index, to avoid retrieval of unifiers which will be rejected at the subsequent steps.

3 Other features

Another interesting feature of Vampire is a strategy intended for the best performance when the time and/or memory are limited. We call it the *limited resource strategy*. This strategy is inspired by Otter's `max_memory` option, but unlike Otter's strategy with `max_memory` fixed in advance, we change the size limits on clauses based on periodical *runtime* evaluation of resources consumed so far.

The idea is as follows. When a time limit is given, Vampire makes estimations on what clauses (called *currently eligible clauses*) can be processed by the end of the time limit at all. For each newly generated clause it is checked whether it is currently eligible. If it is not, Vampire discards the clause. The estimation principle takes into consideration the current strategy and tries to calculate how the process of clause generation will develop by the time limit.

It is very difficult to define the right formulas for estimation. Inaccurate estimations can lead to two possible results:

1. *Too strong eligibility criteria.* Too many clauses are discarded by the strategy. In this case the prover terminates before the time limit expires because no new clauses can be generated.

2. *Too weak eligibility criteria.* Too few clauses are discarded. In this case when the prover terminates, the set of kept clauses contains many clauses to which no inference rules have been applied. Such clauses may have been kept for a long time, thus slowing down processing of new clauses (for example, because subsumption is applied to all kept clauses).

Our first implementations of the limited resource strategy often resulted in incorrect estimations of both kinds. For example, with the time limit of 300 seconds, Vampire sometimes terminated after 250 seconds because too many clauses have been discarded. The current version has better estimations: too strong eligibility criteria only result in loss of less than 1 second. Too weak eligibility criteria are more difficult to avoid: we still have cases when 40% of kept clauses were not used by the time limit. Correct estimations are especially difficult for clauses with many literals.

4 Future

The competition version will have equality rules: superposition and simplification. We are now making experiments on a non-standard splitting rule, different from the one used in SPASS, but we cannot yet report the results. Finally, we plan to implement some forms of constraints.

5 Performance

For the previous competitions it was always the case that several provers exposed better performance than the previous year winner. Therefore, it is difficult to predict the comparative performance of a prover based on the previous year results. However, newer versions of other provers are not yet released, so we can only present a comparison of Vampire with the previous year systems.

Since equality is not yet implemented in Vampire, we only show the results for problems without equality. The performance of other systems is taken from the Web page
`http://www.cs.jcu.edu.au/~tptp/CASC-15/CASC-15FullResults.html`.

We used a SPARC station with a performance similar to computers used at the competition. Our station has more memory, but we put the limit of 64M to simulate the conditions of CASC-15. We would like to note that Vampire's performance is not linear in the speed of a computer, because of the limited resource strategy.

Tables 1 and 2 show the performance of Vampire respectively on Horn and non-Horn problems as compared to the best systems of the 1998 competition CASC-15. Table 3 gives the summary for all 30 problems without equality.

For Horn problems, Vampires is able to solve all problems selected for the competition. For non-Horn problems, the only problem not solved by Vampire (`SYN067-3`) has not been solved by any other prover. In total, Vampire solves 29 problems out of 30, while SPASS that was the best on nonequality problems solved 24 problems.

	Gandalf c-1.1	SPASS 1.0.0a	p-SETHEO C-15	Otter 3.0.5	AI-SETHEO C-15	Vampire
LCL131-1	26.00	timeout	17.20	68.40	no_soln	33.15
LCL207-1	59.80	2.30	1.50	0.30	4.00	0.29
PLA014-1	35.50	0.30	2.50	timeout	5.90	0.06
LCL010-1	0.10	0.30	1.60	0.50	4.00	5.31
LCL211-1	0.30	13.40	1.40	4.70	4.90	0.70
PLA013-1	34.50	0.20	2.00	timeout	6.00	0.05
LCL115-1	1.70	18.90	no_soln	1.40	163.40	26.43
LCL093-1	5.10	timeout	no_soln	43.20	no_soln	38.26
LCL075-1	0.50	1.90	10.40	2.20	15.40	30.57
LCL018-1	timeout	timeout	no_soln	125.70	no_soln	37.43
LCL033-1	0.00	0.20	1.90	0.00	131.90	0.07
LCL129-1	timeout	262.90	no_soln	timeout	no_soln	247.10
SYN192-1	1.60	0.70	5.50	0.50	19.00	0.12
PLA018-1	37.20	1.00	2.60	timeout	5.10	0.42
LCL011-1	0.20	1.40	1.80	39.30	7.30	31.42
PLA012-1	38.70	2.60	3.30	timeout	5.50	0.71
LCL009-1	1.00	0.40	2.30	0.20	5.30	0.71
SYN310-1	0.00	0.20	197.80	0.00	113.50	1.25
LCL051-1	3.10	timeout	no_soln	7.90	no_soln	32.03
SYN190-1	2.10	1.40	5.70	0.50	22.80	0.23
Attempted	20	20	20	20	20	20
Solved	18	16	15	15	15	20
Av.Time	13.74	19.26	17.17	19.65	34.27	24.32

Vampire did not participate in the competition (and does not like light), so we put Vampire's column in gray shade. We also grayed the hard problems: those not solved by at least two of the best provers.

Table 1. Performance for Horn clause non-equality problems

	SPASS c-1.1	p-SETHEO 1.0.0a	AI-SETHEO C-15	Gandalf 3.0.5	Otter C-15	Vampire
SYN476-1	timeout	38.80	55.20	unknown	timeout	175.03
SYN480-1	129.00	38.00	62.80	unknown	timeout	120.16
CIV006-2	0.90	no_soln	no_soln	0.50	0.30	0.09
SYN510-1	149.80	25.10	61.40	unknown	timeout	190.94
PUZ005-1	0.50	2.50	14.40	0.40	0.10	0.07
SYN067-3	timeout	no_soln	no_soln	unknown	timeout	timeout
PUZ032-1	0.20	1.40	4.60	0.00	0.00	0.02
PUZ028-6	2.00	3.00	5.20	unknown	timeout	9.03
SYN036-1	2.40	no_soln	no_soln	106.60	timeout	0.23
SYN486-1	153.00	36.20	69.50	unknown	timeout	85.25
Attempted	10	10	10	10	10	10
Solved	8	7	7	4	3	9
Av.Time	54.73	20.71	39.01	26.88	0.13	58.08

Table 2. Performance for non-Horn non-equality problems

	SPASS c-1.1	Gandalf 1.0.0a	p-SETHEO C-15	AI-SETHEO 3.0.5	Otter C-15	Vampire
Attempted	30	30	30	30	30	30
Solved	24	22	22	22	18	29
Av.Time	31.08	16.13	18.66	35.78	16.40	36.80

Table 3. Overall performance for non-equality problems

Acknowledgments

In our previous life in Uppsala, we have been supported by TFR grants. We thank Christoph Weidenbach for his comments about the implementation of SPASS and Max-Planck Institut für Informatik for supporting a visit of the second author. We thank anonymous referees for their comments, among which our favorite is *"after all, Vampire is not a glass of Tuborg"*.

References

1. H. de Nivelle. An algorithm for the retrieval of unifiers from discrimination trees. *Journal of Automated Reasoning*, 20(1/2):5–25, January 1998.
2. P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.
3. E.L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In A. Voronkov, editor, *Logic Programming and Automated Reasoning. International Conference LPAR'92.*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 96–106, St.Petersburg, Russia, July 1992.
4. William W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
5. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.
6. I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science and MIT Press, 1999. To appear.
7. M. Stickel. The path indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.
8. A. Voronkov. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.
9. C. Weidenbach, B. Gaede, and G. Rock. Spass & flotter. version 0.42. In M.A. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 141–145, New Brunswick, NJ, USA, 1996.