Making your prover meet the deadlines

Alexandre Riazanov and Andrei Voronkov

University of Manchester
{riazanov,voronkov}@cs.man.ac.uk

1 Saturation-based theorem proving with limited time

For most applications it is difficult to expect from human users or systems to be ready to wait for an answer forever. When the time is limited, systems can perform much better by using algorithms other than ordinary complete ones. Here we describe such an algorithm, the so-called Limited Resource Strategy (LRS), implemented in our system Vampire [4].

The fastest contemporary first-order theorem provers (with one exception of Setheo [3]) use saturation algorithms. There exist two main kinds of saturation algorithms, one was implemented in OTTER [2] and its predecessors, another one was used for the first time in DISCOUNT [1]. These algorithms operate on *clauses*. For each new clause generated by an inference the prover decides whether this clause should be *kept* or discarded. The set of kept clauses may be huge, so most of the systems perform inferences only on a subset of them. The clauses in this subset, i.e. those used for inferences will be called *active*, and all other kept clauses *passive*. In the DISCOUNT algorithm passive clauses never participate in inferences or simplifications, while in the OTTER algorithm passive clauses can participate in simplifications.

2 Limited Resource Strategy

The strategy tries to identify which clauses in *passive* and *new* in the OTTER algorithm are *unreachable*, i.e. have no chance to be processed by the time limit at all. To do this, the system keeps track of statistics on the average time spent by processing each clause selected as *current*. From time to time the system tries to estimate how the proof search statistics would develop towards the time limit and, based on this estimation, identify unreachable clauses. Suppose that p clauses have been processed as *current* in t seconds, i.e. p/t clauses per second, and l is the current time limit. Assuming that the proof search will develop at the same pace, in total $p \cdot l/t$ clauses will be processed by the end of the time limit. If the number of currently kept clauses k is greater than $p \cdot l/t$, then $k - p \cdot l/t$ clauses can be discarded. Of course this estimation may be inaccurate, because the time for processing one clause as *current* will most likely increase.

The next question is which $k - p \cdot l/t$ clauses should be discarded. Let us look at the clause selection function. Many theorem provers maintain two priority queues: the *age priority queue* gives higher priority to older clauses, the *weight priority queue* to lighter clauses. The rational behind this strategy is based on the following observation: light clauses are easy to process and most likely to contribute to a derivation of the empty clause, but discarding an old clause is more likely to turn an unsatisfiable set of clauses into a satisfiable one than for a younger clause. The system uses *age-weight ratio* ratio (aka *pick-given* ratio [2]) to decide how often the first clause in each queue should be selected. If the ratio is $a : w$, then out of any $a + w$ selected clauses a will be taken from the age priority queue and w from the weight priority queue. We have decided that $p \cdot (l/t - 1) = p \cdot (l - t)/t$ currently passive clauses can still be processed within the time limit. Of these clauses $a \cdot p \cdot (l - t)/(t \cdot (a + w))$ will be selected from the age priority queue and $w \cdot p \cdot (l - t)/(t \cdot (a + w))$ from the weight priority queue, all clauses that are not in this parts of the queues are unreachable.

Now suppose that the strategy discarded some clauses, and the maximal weight of the remaining clauses is W. If a new clause C obtained by an inference has a weight $W' \geq W$, it is unreachable since it cannot be inserted in the reachable part of the weight priority queue and is younger than any clause in *passive*. Thus, any future clause with the weight $\geq W$ can be discarded, so we can set the limit on the weight to be $W - 1$. We can also use the weight limit to discard any *kept* clause if any inference with this clause as a parent gives a clause with a weight exceeding W. Suppose that we perform a resolution inference with the clause $A \vee C$ where A is the only selected literal. If the weight of C is greater than the weight limit, then any clause inferred from $A \vee C$ would be too heavy. Therefore, $A \vee C$ can be discarded from the search space because it cannot produce a reachable clause. When the weight of C does not exceed W we can sometimes simply estimate the weight of $C\theta$, it is often sufficient to have θ constructed only partially. For example, when retrieval of literals unifiable with $\neg A$ is being performed in an index, we can identify a branch in the index that does not have to be inspected.

3 Comparison of LRS with other approaches

Setting a fixed weight limit in the beginning of proof-search can hardly be helpful since the limit needed to solve an unknown problem can not be calculated a priori. The Incremental Weight Limit Strategy, when the prover is restarted with a bigger weight limit after an unsuccessful run, also has several pitfalls. Suppose that the strategy is applied to a problem for which the minimal weight limit sufficient to solve the problem is W. For some problems, the proof-search with weight limits smaller than W can consume more time than the time limit, while setting the limit to W would solve the problem almost immediately. For some problems, clauses with the weight W are only needed very early in the proof-search. If too many clauses with the weights between W' and W are generated, the strategy can spend too much time on processing these clauses. For the first kind of problems, LRS will behave like a complete algorithm, since early in the proof-search LRS behaves like a complete strategy. For the second kind of problems, when LRS discovers that clauses of the weights between W' and W are unreachable, it will discard these clauses. The DISCOUNT algorithm behaves poorly for problems which require many backward simplification steps. Backward simplification steps are performed only when the simplifying clause becomes active.

The main feature of the LRS is the possibility to adapt to a particular problem based on the runtime information about the proof-search process. No previous knowledge about the problem is needed. Ideally, the LRS should not lose proofs found by a complete strategy. In reality, mistakes in calculating unreachable clauses are unavoidable, so in practice on some problems the complete algorithm beats LRS. For example, when an LRS-based algorithm discards clauses beyond the dynamically set weight limit, it is possible that a simplification of a discarded clause would result in a short proof. However, our experiments carried out over a large number of problems demonstrate that on the average the performance of the LRS-based algorithm is superior to other algorithms.

References

1. J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: a system for distributed equational deduction. In J. Hsiang, editor, *Proc. RTA-95*, volume 914 of *LNCS*, pages 397–402, 1995.
2. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.
3. M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO—the CADE-13 systems. *Journal of Automated Reasoning*, 18:237–246, 1997.
4. A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proc. CADE-16*, volume 1632 of *LNAI*, pages 292–296, 1999.