Adaptive saturation-based reasoning

Alexandre Riazanov and Andrei Voronkov

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

Abstract. We describe the Limited Resource Strategy intended to improve performance of resolution-based theorem provers when a fixed limit is imposed on the time of a run. We give experimental evidence that the Limited Resource Strategy gives a significant improvement over the algorithms not using passive clauses for simplification and the weight-based algorithms.

1 Reasoning with limited resources

In nearly all applications, provers for first-order logic are used in the following way. A time limit is set for every particular goal, and if neither proof nor countermodel could be found within the time limit, the prover is terminated. Then the goal can be reconsidered, for example by formulating intermediate statements (lemmas) or by providing some inference steps interactively, and the proof-search continues on the new goals or using the lemmas. Setting a time limit for processing a particular goal is a natural idea, for most applications it is difficult to expect human users or systems ready to wait for an answer forever. It turns out that when the time is limited, systems can perform much better by using algorithms other than ordinary complete ones. In this abstract we describe such an algorithm, the so-called Limited Resource Strategy (LRS), implemented in our system Vampire [6], discuss its advantages and drawbacks and compare it with the strategies so far used in Vampire and other systems.

2 Saturation-based theorem proving

The fastest first-order theorem provers of the last two CASC competitions [9] (with one exception of Setheo [5]) use saturation algorithms. There exist two main kinds of saturation algorithms, one was implemented in OTTER [4] and its predecessors (see [3]), another one was used for the first time in DISCOUNT [1]. Apart from OTTER, the former algorithm is implemented at least in Gandalf [10], SPASS [11], and Vampire, and the DISCOUNT algorithm has been adopted by E [7], SPASS and Vampire. Saturation algorithms used in first-order theorem provers 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 not on *all* kept clauses, but 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*. The two different saturation algorithms differ in their treatment of passive clauses. In the DISCOUNT algorithm passive clauses never participate in inferences or simplifications, while in the OTTER algorithm passive clauses can participate in simplifications, such as rewriting by unit equalities or subsumption.

2.1 The OTTER saturation algorithm

The OTTER algorithm shown in Figure 1 is parametrized by several procedures explained below:

- *select* is the clause selection function. It decides which clause should be selected for activation.

- *infer* is the function that returns the set of clauses obtained by all inferences between the current clause *current* and the set of active clauses *active*. Usually, *infer* applies inferences in some complete inference system of resolution with paramodulation.
- *simplify*(*set*, *by*) is a procedure that deletes redundant clauses from *set* (e. g. those subsumed by clauses in *by* and tautologies) and simplifies some clauses in *set* using the clauses in *by* (e. g. rewritten by unit equalities in *by*). The simplified clauses are always moved to *passive*.
- Likewise, *inner_simplify* simplifies clauses in *new* using other clauses in *new*.

```
input: init : set of clauses ;
var active, passive, new : sets of clauses ;
var current : clause ;
active := ∅ ;
passive := init ;
while passive ≠ ∅ do
  current := select(passive) ;
  passive := passive − {current} ;
  active := active ∪ {current} ;
  new := infer(current, active) ;
  if goal_found(new) then return provable ;
  inner_simplify(new) ;
  simplify(new, active ∪ passive) ;
  if goal_found(new) then return provable ;
  simplify(active, new) ;
  simplify(passive, new) ;
  if goal_found(active ∪ passive) then return provable ;
  passive := passive ∪ new
od ;
return unprovable
```

Fig. 1. The OTTER Saturation Algorithm

When we simplify *new* using the clauses in *active* $\cup$ *passive*, we speak of *forward simplification*, when we simplify *active* and *passive* using the clauses in *new*, we speak of *backward simplification.*

Typical behavior of this algorithm is quantitatively characterized by the following empirical observation: in a matter of seconds the total number of kept clauses gets very big, whereas the share of the active clauses is small and keeps decreasing. To illustrate this, we provide statistics on an unsuccessful run of Vampire with the time limit of 1 minute on the TPTP problem ANA003-1. During this run, 261,573 clauses were generated. The overall number of active clauses was 1,967, the overall number of passive clauses 236,389. The clauses generated in this run contain function symbols and equality, many of the clauses have a large number of literals and/or heavy terms. Even when the state-of-the-art term indexing techniques are used, it is very difficult to manage clause sets containing over 100,000 clauses efficiently. As a consequence, when theorem provers are used for practical applications, completeness is often compromised in favor of efficiency: the provers discard clauses that may be nonredundant.

2.2 The DISCOUNT saturation algorithm

It was observed that usually the total number of active clauses is considerably less than the number of passive clauses. Therefore, processing the passive clauses consumes a significant amount of time. One can modify the OTTER saturation algorithm in such a way that passive clauses never participate in simplifications. Such a modified saturation algorithm will be called *the DISCOUNT algorithm* in this abstract. The algorithm is shown in Figure 2.

```
input: init : set of clauses ;
var active, passive, new : sets of clauses ;
var current : clause ;
active := ∅ ;
passive := init ;
while passive ≠ ∅ do
  current := select(passive) ;
  passive := passive − {current} ;
  simplify({current}, active) ;
  if current is simplified to a goal return provable
  if current is not simplified to a tautology
  then do
    simplify(active, {current}) ;
    if goal_found(active) then return provable ;
    active := active ∪ {current} ;
    new := infer(current, active) ;
    if goal_found(new) then return provable ;
    simplify(new, active) ;
    passive := passive ∪ new
  od ;
od ;
return unprovable
```

Fig. 2. The DISCOUNT Saturation Algorithm

Compared to the OTTER saturation algorithm, this algorithm has the following features:

- The new clauses are forward simplified by the active clauses only, the passive clauses do not take part in this.
- Neither active nor passive clauses are backward simplified by the retained new clauses.
- After selection of the current clause it is simplified again by the active clauses and then is itself used to simplify the active clauses only.

If we assume that the overall number of kept clauses is significantly larger than the number of used ones, this algorithm involves less computation for the same number of active clauses than the OTTER algorithm. However, this algorithm has a severe weakness: it performs a very limited amount of backward simplification steps compared to the OTTER algorithm. This sometimes results in the following effect: finding some proofs that are quickly found by the OTTER algorithm involving simplification, is now delayed significantly. For example, suppose that two unit clauses $t = a$ and $t = b$ are generated, where t is a heavy term, a and b are constants. If the prover using the DISCOUNT algorithm tries to select lighter clauses first, it may take a long time before these clauses become active. In the OTTER algorithm, rewriting $t = b$ by $t = a$ immediately gives a very light clause $a = b$ which is very likely to contribute to a derivation of the empty clause.

It was observed experimentally that the time spent for storing and retrieving passive clauses in the DISCOUNT algorithm is negligible compared to the overall runtime. Therefore, one cannot expect to improve considerably the performance of the DISCOUNT algorithm by, e.g. trying to discard some passive clauses when a time limit is set (though it can save a lot of memory).

3 Reasoning in limited time by the OTTER algorithm

Growth of the number of kept clauses in the OTTER algorithm causes fast deterioration of the rate of processing of active clauses. Thus, when a complete procedure based on the OTTER algorithm is used, even passive clauses with high selection priority often have to wait indefinitely long before they contribute to the search. In the provers based on the OTTER algorithm, all solutions to the completeness-versus-efficiency problem are based on the same idea: some nonredundant clauses are discarded from the clause sets *active*, *passive*, or *new*. In this section we explain several approaches to discarding clauses implemented in the state-of-the-art provers and analyze their main advantages and disadvantages.

3.1 Weight Limit Strategy

The Weight Limit Strategy was implemented already in the very first versions of OTTER. The idea is to set a limit W on the weight of clauses. The weight of a clause is a measure reflecting its complexity, for example the number of symbols in it. All new clauses with the weight greater than W are discarded. This *Weight Limit Strategy* may be helpful for interactive use and solving difficult problems when the user can analyse the output of an unsuccessful run to adjust the weight limit, but it is not very useful for completely automatic theorem proving since there is no general method for choosing appropriate weight limit. Too small a limit leads to loss of proofs, too high a limit does not improve performance significantly compared to the complete algorthm. Another problem with the weight limit was observed by the authors in case studies: for many problems heavy clauses are needed for a very short time in the beginning of the proof-search, and then only very light clauses suffice for finding a proof.

3.2 Incremental Weight Limit Strategy

Several provers, including Gandalf [10], Bliksem [2] and Fiesta adopted the *Incremental Weight Limit Strategy*. In this strategy the weight limit is initially set to a small value. If no proof is found with this small value, the weight limit is increased, and the proof search begins either from scratch or using the short clauses obtained during the previous run.

3.3 Memory Limit Strategy

This strategy was implemented for the first time in OTTER. The idea is as follows. Some memory limit is set in advance. When $\frac{1}{3}$ of the available memory has been filled, OTTER assigns new weight limit which is calculated in such a way that 5% of passive clauses have smaller weight than the limit. From then on, this recalculation of weight limit is performed after processing every 10 selected clauses.

The main problem with this strategy is that the use of memory and the time are loosely connected. Setting too low a memory limit makes the prover terminate before the time limit because all clauses needed for finding a proof have been discarded. Setting too high a limit results in considerable slowdown, since then the system behaves as poorly as based on a complete algorithm.

4 Limited Resource Strategy

The main idea of LRS is the following. The system tries to identify which clauses in *passive* and *new* are *unreachable*, i.e. have no chance to be processed by the time limit at all, and discards these clauses. The notion of unreachable clauses is fundamentally different from the notion of redundant ones: redundant clauses are those that can be discarded without compromising completeness at all, the notion of unreachable clauses makes sense only in the context of reasoning with limited resources. How can one identify unreachable clauses? When the system starts solving a problem, it is given a time limit t as an argument. The system keeps track of statistics on the average time spent by processing each clause selected as *current*. Usually this time increases because the sets *active* and *passive* are growing, so the operations with them take more and more time. 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.

The main requirements we imposed on the implementation are the following.

Requirement 1. Vampire with LRS should be at least as fast as Vampire using the complete algorithm.

Requirement 2. A proof should not be lost when the time limit is set to an acceptable value. Suppose that Vampire with a time limit t_1 has found a proof in time $t_2 < t_1$. Then Vampire with the time limit t_2 should find a proof as well. In other terms, when the time limit is set to t_2 no reachable clause should be lost.

Requirement 3. As many unreachable clauses as possible should be identified as unreachable by Vampire.

Requirement 3 is in conflict with Requirement 2, because the exact estimation of which clauses are unreachable is essentially impossible.

The following example gives the reader an idea how an estimation of unreachable clauses can be done. 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. If we assume 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. So 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. To avoid too big errors, the estimation of p/t must be done frequently enough. In our experiments the estimation was performed after every 500 inferences were produced.

The next question is which $k-p\cdot l/t$ clauses should be discarded. The answer can be obtained by applying Requirement 2. One of the consequences of this principle is that no clause processed by the time limit by the complete strategy should be discarded. But the clause selection in the complete algorithm is controlled by the function *select*, so to identify potentially unreachable clauses let us look deeper at the clause selection function.

All modern theorem provers maintain one or more priority queues from wich clauses are picked using some *ratios*. By far the most popular design is based on 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 a ratio to decide how often the first clause in each queue should be selected. This ratio is called the *pick-given* ratio in OTTER's manual [4], we will call it the *age-weight ratio*. For example, if the age-weight ratio is 1:4, then out of each 5 selected clauses 1 will be taken from the age priority queue and 4 from the weight priority queue. This strategy was introduced for the first time in OTTER and then used by a number of systems.

Assume that our clause selection function is based on the age-weight queue design with age-weight ratio $a : w$, which means that out of any $a + w$ clauses a will be selected from the age queue and w from the weight 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. So Vampire implements a deletion algorithm that discards clauses according to these formulas.

This example shows that LRS can delete many unreachable clauses from *passive*, but it does not demonstrate the full power of the strategy. Suppose that the strategy discarded some clauses, and the maximal weight of the remaining clauses is W. Suppose a new clause C obtained by an inference has a weight $W' \geq W$. This clause is unreachable since it cannot be inserted in the reachable part of the weight priority queue and is younger than any clause in *passive*. This means that any future clause with the weight $\geq W$ can be discarded, so we can set the limit on the weight to be $W - 1$.

Apart from using the dynamically changing weight limit W for discarding new clauses, 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. Resolution-based provers use calculi based on ordered resolution with negative selection. A typical inference rule in such a calculus is ordered resolution with negative selection:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\theta},$$

where θ is a most general unifier of A and B, the atom $A\theta$ is maximal in the clause $(A \vee C)\theta$ and $\neg B$ is a literal selected in the clause $\neg B \vee D$. The nonmaximal part of $A \vee C$ will always be part of the clause $C \vee D$. The application of the substitution θ to $C \vee D$ yields a clause at least as heavy as $C \vee D$ (unless we factor equal literals). Suppose now that we perform a resolution inference with the clause $A \vee C$. 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. To implement this, when LRS reduces the weight limit, we can search through the whole set $\mathit{passive} \cup \mathit{active}$ for clauses whose nonmaximal (nonselected) part has a weight greater than W and discard them.

When the weight of C does not exceed W we can sometimes simply compute the weight of $C\theta$. For example, if we have found the substitution θ together with a sufficiently big set of clauses containing the literal $\neg A\theta$, it still might be useful to compute the weight of $C\theta$ and compare it with W in an attempt to avoid building all the inferences. Moreover, to estimate weight of $C\theta$ it is often sufficient to have θ constructed only partially. This can be done, for example, when retrieval of literals unifiable with $\neg A$ is being performed in an index, in which case we can identify a branch in the index that does not have to be inspected.

5 Comparison of LRS with other approaches

The main feature of the LRS over other algorithms 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. In this section we briefly explain some advantages of LRS as compared to other existing approaches.

Weight-limit based approaches. Setting a particular weight limit in the beginning of proof-search can hardly be helpful since the weight limit needed to solve an unknown problem can not be calculated a priori. So we only compare LRS with the Incremental Weight Limit Strategy. This strategy has several well-known 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 such problems the Incremental Weight Limit Strategy is likely to be much less efficient than the complete strategy.
- For some problems, clauses with the weight W are only needed very early in the proof-search, and then clauses with weights less than or equal to some $W' < W$ will suffice. If too many clauses with the weights between W' and W are generated, the strategy can spend too much time on processing these clauses and will fail to find a proof.

The Limited Resource strategy is immune to both kinds of problems. 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. 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. As a consequence, sometimes the algorithm cannot find proofs easily found by other strategies, especially when the proofs contain simplification steps between heavy clauses.

Shortcomings of LRS. Ideally, the requirements for LRS guarantee that it 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.

The main reason for miscalculating reachability of clauses is backward simplifications. 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.

6 Experiments

To compare the LRS-based algorithm with the DISCOUNT algorithm, we implemented the DISCOUNT algorithm in Vampire and made a number of experiments on two benchmarks suites: (i) all 3340 clausal problems in TPTP, (ii) the 1836 problems from the list software reuse application (see [8]). On each problem, Vampire was run with 3 different literal selection functions using the DISCOUNT algorithm, and with the same 3 different selection functions using the LRS-based algorithm. Therefore, altogether we compared the two algorithms on 15,528 tests. To summarize the results, we consider only tests satisfying the following conditions: (i) exactly one of the two algorithms solved the problem, or (ii) both algorithms solved the problem, and at least one of them spent more than 10 seconds on it. Of 1726 benchmarks, 1492 were solved by the LRS-based algorithm, and 1045 by the DISCOUNT algorithm. The DISCOUNT algorithm was not able to solve 681 problems solved by the LRS, compared to 234 problems not solved by the LRS while solved by the DISCOUNT algorithm. The LRS was faster on 241 of the problems solved by both algorithms, while the DISCOUNT algorithm was faster on 161 problems.

We also compared the OTTER algorithm with and without LRS using the same benchmark suites as in the previous section. Of 1318 benchmarks selected according to the same criterion as above, 1267 were solved by the LRS-based algorithm, and 589 by the standard OTTER algorithm.

To illustrate how LRS influences the proof-search statistics in terms of the share of active clauses in the kept clauses, consider the TPTP problem ANA003-1. This problem was solved by no algorithm. The following table summarizes the total number of used and kept clauses.

	DISCOUNT	OTTER	LRS
used	8,191	1,967	42,050
kept	1,473,106	236,389	51,751

The DISCOUNT algorithm could process about 4 times more active clauses than the OTTER algorithm. However, it comes at a price of not performing some simplification steps. Also, the DISCOUNT algorithm kept about 6 times more clauses than the OTTER algorithm since it could not recognize that some of them are redundant w.r.t. passive clauses. The LRS-based algorithm could process about 21 times more active clauses than the OTTER algorithm and about 4 times more than the DISCOUNT algorithm. The small difference between the numbers of the kept and the active clauses shows that the calculations of reachable clauses made by LRS were quite precise.

To compare the LRS with weight-limit based approaches, we experimented with the 75 problems from the CASC-16 competition in the mixed category. We compare the results obtained by Vampire using the following strategies based on the OTTER algorithm: LRS, four different fixed weight limits (50, 40, 30, 20) and Incremental Weight Limit (winc). Only those 51 problems for which a proof was found by at least one strategy are considered. In the table below we give the total number of problems solved by each strategy.

strategy	LRS	50	40	30	20	winc
solved	48	45	36	27	21	33

As it can be seen from the results, the Limited Resource Strategy solves more problems than the Incremental Weight Limit Strategy or any strategy using fixed weight limit, even when the values of the weight limit are optimal for this benchmark suite. There is a problem that could only be solved by LRS (ANA002-4), and for 3 more problems the time obtained by this strategy was considerably better than by any other strategy. LRS also gives better average time than the strategy with weight limit 50, when they are compared on problems solved by both of them.

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. H. de Nivelle. *Bliksem 1.10 User's Manual.* MPI für Informatik, Saarbrücken, 2000.
3. E.L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In A. Voronkov, editor, *Proc. LPAR'92.*, volume 624 of *LNAI*, pages 96–106, St.Petersburg, Russia, 1992.
4. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.
5. 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.
6. A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proc. CADE-16*, volume 1632 of *LNAI*, pages 292–296, 1999.
7. S. Schulz. System abstract: E 0.3. In H. Ganzinger, editor, *Proc. CADE-16*, LNAI, pages 297–301, 1999.
8. J. Schumann and B. Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. ASE-97*, pages 246–254. IEEE Computer Society Press, 1997.
9. G. Sutcliffe. The CADE-16 ATP system competition. *Journal of Automated Reasoning*, 24(3):371–396, 2000.
10. T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.
11. C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description: SPASS version 1.0.0. In H. Ganzinger, editor, *Proc. CADE-16*, volume 1632 of *LNAI*, pages 378–382, 1999.