THE UNIVERSITY
of MANCHESTER

October 20, 2000

Computer Science
University of Manchester

Limited Resource Strategy in Resolution Theorem Proving

Alexandre Riazanov and Andrei Voronkov

Department of Computer Science
University of Manchester
Preprint Series
CSPP-7

Limited Resource Strategy in Resolution Theorem Proving

Alexandre Riazanov and Andrei Voronkov

October 20, 2000

Abstract

For most applications of first-order theorem provers a proof should be found within a fixed time limit. When the time limit is set, systems can perform much better by using algorithms other than the ordinary complete ones. In this report we describe the Limited Resource Strategy intended for reasoning with limited resources. 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.

Contents

1 Reasoning with limited resources

First-order theorem provers (in the sequel simply *provers*) have a number of applications. The main application areas for provers are software and hardware verification. In nearly all applications, provers are used in the following way. When a prover is run on a goal, a time limit is set. 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.

Since first-order logic is undecidable, any complete prover is potentially non-terminating. Therefore setting a time limit for processing a particular goal is a natural idea. Moreover, for most applications it is difficult to expect human users or systems ready to wait for an answer forever. This report addresses the problem of reasoning in limited time. It turns out that when the time is limited, systems can perform much better by using algorithms other than ordinary complete ones. In this report we describe the so-called Limited Resource Strategy (LRS) implemented in our system Vampire [Riazanov and Voronkov 1999], discuss its advantages and drawbacks, compare it with the strategies so far used in Vampire and other systems and describe experiments carried out over a number of first-order problems.

2 Saturation-based theorem proving

The fastest first-order theorem provers of the last two CASC competitions [Sutcliffe 2000] (with one exception of Setheo [Moser, Ibens, Letz, Steinbach, Goller, Schumann and Mayr 1997]) use saturation algorithms. There exist two main kinds of saturation algorithms, one was implemented in OTTER [McCune 1994] and its predecessors (an overview of these early provers can be found in [Lusk 1992]), another one was used for the first time in DISCOUNT [Avenhaus, Denzinger and Fuchs 1995]. In this section we describe these saturation algorithms. For simplicity, we will call them the OTTER and the DISCOUNT algorithms, respectively. The former algorithm is implemented at least in OTTER, Gandalf [Tammet 1997], SPASS [Weidenbach, Afshordel, Brahm, Cohrs, Engel, Keen, Theobalt and Topic 1999], and Vampire, and the latter one is implemented at least in DISCOUNT, E [Schulz 1999], SPASS and Vampire.

Unfortunately, the terminology related to saturation algorithms used in different research groups varies a lot, so no standard terminology exists. Let us develop some relevant terminology. 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*. All other kept clauses are *passive* (though as we will see passive clauses can also participate in inferences, but of special kind).

The two different saturation algorithms differ in their treatment of passive clauses. In the DISCOUNT algorithm passive clauses never participate in inferences or simplifications. In the OTTER algorithm passive clauses can participate in simplifications, for example rewriting by unit equalities or subsumption.

2.1 The OTTER saturation algorithm

Let us begin with the OTTER algorithm which is shown in Figure 1. It 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 performs inferences between the current clause *current* and the set of active clauses *active*. This function returns the set of clauses obtained by all such possible inferences. This function varies from system to system. Usually, *infer* applies inferences in some complete inference system of resolution with paramodulation.
- $\mathit{simplify}(\mathit{set}, \mathit{by})$ is a procedure that performs simplification. It deletes redundant clauses from *set* and simplifies some clauses in *set* using the clauses in *by*. To preserve completeness, the simplified clauses are always moved to *passive*. Typically, deleted clauses include tautologies and those clauses subsumed by clauses in *by*. A typical example of simplification is rewriting by unit equalities in *by*.
- Likewise, *inner_simplify* simplifies clauses in *new* using other clauses in *new*.

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

input: *init* : set of clauses ;
var *active*, *passive*, *new* : sets of clauses ;
var *current* : clause ;
active := $\emptyset$;
passive := *init* ;
while *passive* $\neq \emptyset$ **do**
  current := *select*(*passive*) ;
  passive := *passive* $-$ {*current*} ;
  active := *active* $\cup$ {*current*} ;
  new := *infer*(*current*, *active*) ;
  if *goal_found*(*new*) **then** **return** *provable* ;
  inner_simplify(*new*) ;
  simplify(*new*, *active* $\cup$ *passive*) ;
  if *goal_found*(*new*) **then** **return** *provable* ;
  simplify(*active*, *new*) ;
  simplify(*passive*, *new*) ;
  if *goal_found*(*active* $\cup$ *passive*) **then** **return** *provable* ;
  passive := *passive* $\cup$ *new*
od ;
return *unprovable*

Figure 1: The OTTER Saturation Algorithm

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. It was possible to process such a large number of clauses in one minute due to success of the modern indexing techniques [Graf 1996, Ramakrishnan, Sekar and Voronkov 2000]. Even when the state-of-the-art 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 report. The algorithm is shown in Figure 2.

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 don

```
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
```

Figure 2: The DISCOUNT Saturation Algorithm

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. Again, the passive clauses are not affected.

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 backward simplification, is now delayed significantly. To illustrate this, consider a simple example. Suppose that a, b are constants and two unit clauses $t = a$ and $t = b$ with a heavy term t are generated. Since most provers try to select lighter clauses (with fewer function symbols), it may take a long time before these clause both appear among the active ones. Rewriting the latter clause by the former one gives a very light clause $a = b$ which may be very likely to contribute to a derivation of the empty clause. This simplification from $t = a$ into $t = b$ will be discovered by the OTTER algorithm as soon as both $t = a$ and $t = b$ have been generated, but it may take a long time before they get selected by the DISCOUNT algorithm. The same applies for other selection criteria, for example when older clauses are preferred to younger ones, since $t = a$ and $t = b$ can as well be younger clauses, so their selection will be delayed.

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* is especially helpful for interactive use and solving difficult problems. The user specifies an arbitrary weight limit, runs the prover on a goal, and studies the output. If a proof was not found, the weight limit can be changed. Weight

Limit Strategy is not very useful for fully automatic theorem proving. Since problems submitted to provers are of very diverse nature, the useful weight limits can vary. When the weight limit is too high, there is essentially no difference between a complete algorithm and a weight-limit based one. When the weight limit is too small, a proof with the given weight limit may not exist at all, even when a complete algorithm will easily find a proof. Another problem with the weight limit was observed 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

However, since light clauses experimentally proved to be very useful, the Weight Limit Strategy is very appealing. Several provers, including Gandalf [Tammet 1997], Bliksem [de Nivelle 2000] and Fiesta adopted the *Incremental Weight Limit Strategy*. The idea of this strategy is that 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. This strategy is very efficient on quite a number of problems, but also very inefficient on many problems. There are two kind of problems on which the strategy behaves poorly.

1. Suppose that W is the minimal weight sufficient to find a proof of a problem by the prover's inference system. If the proof search for the weight $W - 1$ is too costly, the prover will not increment the weight to W, spending all his time on smaller weights.

2. When heavy clauses are only needed in the beginning of proof-search, the strategy is inefficient, since it will generate and store heavy clauses when it is no more necessary.

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 memory allocated by the prover for storing clauses and indexing structures exceeds the memory limit, the prover discards a clause, usually the heaviest one, thus releasing some amount of memory. This strategy has its own disadvantages. The main problem is that the use of memory and the time are loosely connected. If a complete algorithm is run by Vampire, the memory used in 1 minute can vary as much as between 20 and 400 megabytes on a computer with the 400MHz Pentium processor. Setting too a low memory limit will make a prover terminate before the time limit because all clauses needed for finding a proof will have been discarded. This effect was indeed observed in some previous versions of OTTER intended for the CASC competitions. For example, in the experiments presented in [Voronkov 2000] OTTER terminates before the time limit without finding a proof on a number of problems, in the worst case in less than 2 minutes when the time limit was 30 minutes. Setting too a high limit will result in considerable slowdown, since then the system will behave 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* have no chance to be processed by the time limit at all, and discards these clauses. Such clauses will be called *unreachable*. Note that the notion of unreachable clauses is fundamentally different from the notions 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 a proof search on a problem, it is also given a time limit t as an argument. The system keeps track of statistics on the use of resources during the proof search. The main resource measure is the average time spent by processing each clause selected as *current*. Note that usually this time increases because the sets *active* and *passive* are growing (*new* is usually relatively small), so the operations *infer* and *simplify* 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. This means roughly the following. 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.

These requirements can be easily satisfied when no clause is identified as unreachable, i.e. by using a complete algorithm. So we have a third requirement

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.

To give the reader an idea how an estimation of unreachable clauses can be done, we give a simple example. 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.

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. Clauses are picked from the priority queues 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 [McCune 1994], 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, including at least OTTER, Vampire, and Waldmeister [Hillenbrand, Buch, Vogt and Löchner 1997].

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. We will now show that the strategy can have drastic influence on the way the sets *new* and *active* are processed. This effect is due to the following observation. 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 $W' \geq W$. We claim this clause is unreachable. Indeed, C cannot be inserted in the reachable part of the weight priority queue. In addition, C is younger than any clause in *passive*. Thus, it cannot be in the reachable part of the age priority queue. So C is unreachable. 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 $W - 1$. (Note that this limit can go down as soon as the inference process slows down or more clauses with weights less than W are kept).

Apart from using the dynamically changing weight limit W for discarding new clauses, we can also use the weight limit to discard some *kept* clauses. In order to discard a kept clause we have to be sure that 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 clause $(C \vee D)\theta$ is called a *resolvent* of the clauses $A \vee C$ and $\neg B \vee D$.

When this rule is applied, the nonmaximal part of $A \vee C$ will always be part of the clause $C \vee D$. Likewise, the non-selected part of $\neg B \vee D$ will be part of $C \vee D$. It follows that $C \vee D$ is always at least as heavy as the nonmaximal part of $A \vee C$ or the nonselected part of $\neg B \vee D$. 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 *passive* $\cup$ *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 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 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 hardly 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 Experimental comparison of the DISCOUNT algorithm and the LRS-based algorithm

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 [Schumann and Fischer 1997]). 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. The time limit for each run was set to 60 seconds. We used a computer with the 450 MHz Pentium processor. Therefore, altogether we compared the two algorithms on 15,528 tests. The comparison is fair for the following reason. Both algorithm used the same computer, algorithms, datastructures, and memory management. The only difference was on the top-level loop.

To summarize the results, we consider only so-called *interesting* tests. Intuitively, a test is interesting if it is neither too easy nor too difficult. Formally, we consider the following criterion. A test is interesting if one of the following conditions is satisfied:

1. exactly one of the two algorithms solved the problem; or
2. both algorithms solved the problem, and at least one of them spent more than 10 seconds on it.

The results are presented in Figure 3. In this figure we use the following notation:

$++$: the problem was solved only by the LRS-based algorithm;

$+$: the problem was solved by both algorithms, but the LRS-based one was at least 10 seconds faster;

$=$: the problem was solved by both algorithms, and the difference in the solution times is less than 10 seconds;

$-$: the problem was solved by both algorithms, but the LRS-based one was at least 10 slower faster;

$--$: the problem was solved only by the discount algorithm.

Of 1726 interesting benchmarks, 1492 were solved by the LRS-based algorithm, and 1045 by the DISCOUNT algorithm. If these experiments were carried out as part of an interactive verification proof, a significant amount of time would have been saved by using the LRS-based algorithm.

7 Experimental comparison of the OTTER algorithm with and without LRS

We compared the OTTER algorithm with and without LRS using the same benchmark suites as in the previous section. The results are shown in Figure 4. Of 1318 interesting benchmarks, 1267 were solved by the LRS-based algorithm, and 589 by the standard OTTER algorithm.

category	++	+	=	−	−−	interesting
ALG	0	0	3	0	0	3
ANA	6	4	1	0	0	11
BOO	27	19	11	0	0	57
CAT	10	5	2	0	0	17
CID	3	0	0	0	0	3
CIV	0	3	0	0	0	3
COL	86	1	2	16	3	108
FLD	43	35	7	3	37	125
GEO	25	5	21	18	3	72
GRP	54	21	17	11	6	109
HEN	0	10	0	0	0	10
KRS	0	0	0	0	1	1
LAT	14	6	0	0	0	20
LCL	60	23	80	16	11	190
LDA	3	8	12	2	8	33
MSC	1	1	3	0	0	5
NUM	8	2	7	1	0	18
PLA	0	1	0	0	0	1
PUZ	1	1	2	0	1	5
RNG	16	5	1	0	0	22
ROB	17	0	0	0	3	20
SET	31	18	70	10	17	146
SYN	31	6	24	2	7	70
TOP	2	0	0	0	0	2
LRU	243	67	146	82	137	675
total	681	241	409	161	234	1726

Figure 3: Comparison of the LRS-based algorithm with the DISCOUNT algorithm

category	++	+	=	−	−−	interesting
ALG	0	0	3	0	0	3
ANA	11	0	1	0	0	12
BOO	0	0	12	0	0	12
CAT	12	0	0	0	0	12
COL	20	25	32	0	3	80
FLD	88	14	2	0	7	111
GEO	62	4	9	0	0	75
GRP	25	18	26	0	6	75
LAT	6	0	4	0	0	10
LCL	152	41	25	0	5	223
LDA	12	10	2	0	0	24
MSC	0	2	2	0	0	4
NUM	17	12	3	0	1	33
PLA	1	0	0	0	0	1
PUZ	0	1	0	0	0	1
RNG	5	2	3	0	0	10
ROB	5	0	0	0	6	11
SET	67	18	40	0	5	130
SYN	3	0	63	0	0	66
TOP	2	0	0	0	0	2
LRU	241	81	59	24	18	423
total	729	228	286	24	51	1318

Figure 4: Comparison of the LRS-based algorithm with the OTTER algorithm without LRS

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.

8 Comparing LRS with weight-limit based approaches

Here we present results of experiments on the 75 problems from the last CASC competition in the mixed category. The problems were run with the time limit of 5 minutes on a computer with a SPARC 233 Mhz processor. 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). We present time (in seconds) spent for finding a proof. If a proof was not found, we use "-". Only those problems for which a proof was found by at least one strategy are presented. We give the total number of problems solved by each strategy. If a particular strategy solves a problem considerably faster than all other strategies or it is the only strategy that solves the problem, the corresponding entry is given in boldface.

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, and for 3 more problems the time obtained by this strategy is 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

AVENHAUS J., DENZINGER J. AND FUCHS M. [1995], DISCOUNT: a system for distributed equational deduction, *in* J. Hsiang, ed., 'Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA-95)', Vol. 914 of *Lecture Notes in Computer Science*, Kaiserslautern, pp. 397–402.

DE NIVELLE H. [2000], *Bliksem 1.10 User's Manual*, MPI für Informatik, Saarbrücken.

GRAF P. [1996], *Term Indexing*, Vol. 1053 of *Lecture Notes in Computer Science*, Springer Verlag.

HILLENBRAND T., BUCH A., VOGT R. AND LÖCHNER B. [1997], 'Waldmeister: High-performance equational deduction', *Journal of Automated Reasoning* **18**(2), 265–270.

problem	LRS	50	40	30	20	winc
ana002-4	**160.6**	-	-	-	-	-
cat005-3	0.1	0.1	0.1	0.1	0.1	0.2
col003-10	46.7	**36.2**	-	-	-	-
col003-7	46.9	**36.2**	-	-	-	-
col003-8	46.3	**35.9**	-	-	-	-
col003-9	47.0	**36.8**	-	-	-	-
col042-2	11.6	10.4	-	-	-	-
col042-3	11.5	10.1	-	-	-	-
col081-1	-	-	-	-	**253.0**	-
geo010-1	32.7	96.4	30.2	**4.0**	-	15.9
geo065-2	0.0	0.0	0.0	0.0	0.0	0.3
grp050-1	3.5	2.7	1.0	-	-	**0.5**
grp051-1	3.7	36.8	-	-	-	27.8
grp052-1	25.1	2.7	1.2	-	-	**0.9**
grp056-1	1.4	0.0	-	-	-	0.9
grp062-1	13.1	12.2	5.6	-	-	**2.6**
grp071-1	62.6	90.9	79.5	51.7	-	**2.3**
grp072-1	58.7	113.7	-	19.9	-	**11.5**
grp077-1	0.1	0.1	0.1	0.1	0.0	0.3
grp078-1	0.1	0.1	0.1	0.1	0.1	0.2
grp083-1	14.1	13.0	8.3	**5.0**	-	90.2
grp105-1	**57.7**	177.8	178.4	-	-	-
lcl002-1	-	-	-	227.4	-	**192.3**
lcl017-1	**43.9**	100.1	94.2	-	-	-
lcl021-1	156.5	263.6	164.0	-	-	-
lcl064-1	**49.9**	71.2	69.1	68.7	221.7	-

problem	LRS	50	40	30	20	winc
lcl105-1	-	-	-	-	192.9	222.2
lcl121-1	37.1	38.9	39.8	3.8	-	**3.1**
lcl128-1	36.1	39.5	39.4	**12.6**	-	13.7
lcl191-1	28.3	50.9	50.7	49.5	16.0	**7.3**
lcl222-1	136.1	155.4	152.3	148.0	**61.1**	77.9
lcl223-1	92.2	94.2	92.5	89.0	**31.9**	64.9
pla009-1	0.4	0.4	0.3	-	-	-
pla015-1	1.2	1.2	1.2	-	-	-
pla018-1	0.4	0.4	0.4	-	-	-
pla023-1	0.7	0.6	0.7	-	-	-
puz017-1	0.1	0.1	0.0	0.1	-	-
rng041-1	0.1	0.1	0.1	0.1	0.1	0.3
set095-6	0.2	0.2	0.2	0.2	0.2	1.4
set130-6	75.8	-	-	-	-	**19.7**
set135-6	75.4	-	-	-	-	**19.9**
set236-6	0.5	0.5	0.5	0.5	0.4	-
syn015-1	0.3	0.3	0.3	-	-	0.8
syn436-1	68.5	71.1	66.9	69.2	69.3	59.2
syn448-1	15.6	14.4	14.1	14.2	14.1	13.8
syn461-1	8.4	9.0	8.4	8.8	8.4	**6.0**
syn469-1	62.7	65.5	61.7	65.5	64.0	**40.4**
syn477-1	6.5	6.6	6.5	6.5	6.6	7.0
syn486-1	8.8	8.8	8.8	9.2	8.7	19.6
syn489-1	30.5	32.7	30.6	32.1	30.9	**7.7**
syn500-1	18.7	18.8	18.6	18.8	18.7	72.1
total	48	45	36	27	21	33

Figure 5: LRS vs. weight-limit based strategies

LUSK E. [1992], Controlling redundancy in large search spaces: Argonne-style theorem proving through the years, *in* A. Voronkov, ed., 'Logic Programming and Automated Reasoning. International Conference LPAR'92.', Vol. 624 of *Lecture Notes in Artificial Intelligence*, St.Petersburg, Russia, pp. 96–106.

MCCUNE W. [1994], OTTER 3.0 reference manual and guide, Technical Report ANL-94/6, Argonne National Laboratory.

MOSER M., IBENS O., LETZ R., STEINBACH J., GOLLER C., SCHUMANN J. AND MAYR K. [1997], 'SETHEO and E-SETHEO—the CADE-13 systems', *Journal of Automated Reasoning* **18**, 237–246.

RAMAKRISHNAN I., SEKAR R. AND VORONKOV A. [2000], Term indexing, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Elsevier Science and MIT Press. To appear.

RIAZANOV A. AND VORONKOV A. [1999], Vampire, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Vol. 1632 of *Lecture Notes in Artificial Intelligence*, Trento, Italy, pp. 292–296.

SCHULZ S. [1999], System abstract: E 0.3, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Lecture Notes in Artificial Intelligence, Trento, Italy, pp. 297–301.

SCHUMANN J. AND FISCHER B. [1997], NORA/HAMMR: Making deduction-based software component retrieval practical, *in* 'Proc. Automated Software Engineering (ASE-97)', IEEE Computer Society Press, Lake Tahoe, pp. 246–254.

SUTCLIFFE G. [2000], 'The CADE-16 ATP system competition', *Journal of Automated Reasoning* . to appear.

TAMMET T. [1997], 'Gandalf', *Journal of Automated Reasoning* **18**(2), 199–204.

VORONKOV A. [2000], CASC $16\frac{1}{2}$, Preprint CSPP-4, Department of Computer Science, University of Manchester.
URL: *http://www.cs.man.ac.uk/preprints/index.html*

WEIDENBACH C., AFSHORDEL B., BRAHM U., COHRS C., ENGEL T., KEEN E., THEOBALT C. AND TOPIC D. [1999], System description: SPASS version 1.0.0, *in* H. Ganzinger, ed., 'Automated Deduction—CADE-16. 16th International Conference on Automated Deduction', Vol. 1632 of *Lecture Notes in Artificial Intelligence*, Trento, Italy, pp. 378–382.