On the Evaluation of Indexing Techniques for Theorem Proving

Robert Nieuwenhuis[1⋆], Thomas Hillenbrand[2]
Alexandre Riazanov[3], and Andrei Voronkov[3]

[1] Technical University of Catalonia
roberto@lsi.upc.es
[2] Universität Kaiserslautern
hillen@informatik.uni-kl.de
[3] University of Manchester
{riazanov,voronkov}@cs.man.ac.uk

1 Introduction

The problem of *term indexing* can be formulated abstractly as follows (see [19]). Given a set L of *indexed terms*, a binary relation R over terms (called the *retrieval condition*) and a term t (called the *query term*), identify the subset M of L that consists of the terms l such that $R(l,t)$ holds. Terms in M will be called the *candidate terms*. Typical retrieval conditions used in first-order theorem proving are matching, generalization, unifiability, and syntactic equality. Such a retrieval of candidate terms in theorem proving is interleaved with insertion of terms to L, and deletion of them from L.

In order to support rapid retrieval of candidate terms, we need to process the indexed set into a data structure called the *index*. Indexing data structures are well-known to be crucial for the efficiency of the current state-of-the-art theorem provers. Term indexing is also used in logic and functional programming languages implementation, but indexing in theorem provers has several distinctive features:

1. Indexes in theorem provers frequently store 10^5–10^6 complex terms, unlike a typically small number of shallow terms in functional and logic programs.
2. In logic or functional language implementation the index is usually constructed during compilation. On the contrary, indexes in theorem proving are highly dynamic, since terms are frequently inserted in and deleted from indexes. *Index maintenance* operations start with an index for an initial set of terms L, and incrementally construct an index for another set L' that is obtained by insertion or deletion of terms to or from L.
3. In many applications it is desirable for several retrieval operations to work on the same index structure in order to share maintenance overhead and memory consumption.

⋆ Partially supported by the Spanish CICYT project HEMOSS ref. TIC98-0949-C02-01.

Therefore, along the last two decades a significant number of results on new indexing techniques for theorem proving have been published and successfully applied in different provers [19, 6, 8, 18, 24, 16, 11, 2, 4, 5, 26, 22, 21]. In spite of this, important improvements of the existing indexing techniques are still possible and needed, and other techniques for previously not considered retrieval operations need to be developed [14]. But implementors of provers need to know which indexing technique is likely to behave best for his/her applications, and developers of new indexing techniques need to be able to compare techniques in order to get intuition about where to search for improvements, and in order to provide scientific evidence of the superiority of new techniques over other previous ones[1][2].

Unfortunately, practice has revealed that an asymptotic worst-case or average-case complexity analysis of indexing techniques is not a very realistic enterprise. Even if such analysis was done, it would be hardly useful in practice. For example, very efficient linear- or almost linear-time algorithms exist for unification [17, 10] but in practice these algorithms proved to be inefficient *for typical applications*, and quadratic- or even exponential-time unification algorithms are used instead in the modern provers. Thus, theoretically worse (w.r.t. asymptotic complexity analysis) algorithms in this area frequently behave better in practice than other optimal ones. For many techniques one can design bad examples whose (worst-case) computational complexity is as bad as for completely naive methods. An average-case analysis is also very difficult to realize, among other reasons because in most applications no realistic predictions can be made about the distribution of the input data.

Similar phenomena take place when randomly generated data are used for benchmarking. For example, in propositional satisfiability attempts to find hard problems resulted in discovering random distributions of clauses which guarantee the existence of a *phase transition* [1, 13]. Experimentally it has been discovered that problems resulting from the random clause generation in the phase transition region are hard for all provers, but the provers best for these problems proved to be not very efficient for more structured problems coming from practical applications [7].

---

[1] In fact, in the recent past two of the authors recommended rejection for CADE of each other's papers on improvements of code tree and substitution tree indexing due to the lack of evidence for a better performance.

[2] Due to the lack of evidence of superiority of some indexing techniques over other ones, system implementors have to take decisions about implementing a particular indexing technique based on criteria not directly relevant to the efficiency of the technique. As an example, we cite [23]:

> Following the extremely impressive results of Waldmeister, we have chosen a *perfect discrimination tree* ... as the core data structure for our indexing algorithms.

Here the overall results of Waldmeister were considered enough to conclude that it implements the best indexing techniques.

Hence the only reasonable evaluation method is to apply a statistical analysis of the *empirical* behaviour of the different techniques on benchmarks corresponding to real runs of real systems on real problems. But one has to be careful because one should not draw too many conclusions about the efficiency of different *techniques* based on a comparison between different concrete *implementations* of them; many times these implementations are rather incomparable due to different degrees of optimization and refinement[3].

Our main contribution here is the *design of the first method for comparing different implementations* based on a virtually unlimited supply of large real-world benchmarks for indexing. The basic requirements to such benchmarks are as follows. First, since different provers may impose different requirements on the indexing data structures, a general means should be given for obtaining realistic benchmarks for any given prover. Second, it should be possible to easily create benchmarks by running this prover on any problem, and do this for a significant number of different problems from different areas. Third, these benchmarks have to reproduce real-life sequences of operations on the index, where updates (deletions and insertions of terms) are interleaved with (in general far more frequent) retrieval operations.

The method we use for creating such benchmarks for a given prover is to add instructions making the prover write to a log file a trace each time an operation on the index takes place, and then run it on the given problem. For example, each time a term t is inserted (deleted, unified with), a trace like $+t$ (resp. $-t$, ut) is written to the file. Moreover, we require to store the traces along with information about the result of the operation (e.g., success/failure), which allows one to detect cases of incorrect behaviour of the indexing methods being tested. Ideally, there should be enough disk space to store all traces (possibly in a compressed form) of the whole run of the prover on the given problem (if the prover terminates on the problem; otherwise it should run at least for enough time to make the benchmark representative for a usual application of the prover).

The main part of the evaluation process is to test a given implementation of indexing on such a benchmark file. This given implementation is assumed to provide operations for querying and updating the indexing data structure, as well as a translation function for creating terms in its required format from the benchmark format. In order to avoid overheads and inexact time measurements due to translations and reading terms from the disk, the evaluation process first reads a large block of traces, storing them in main memory. After that, all terms read are translated into the required format. Then time measuring is switched on, and a loop is started which calls the corresponding sequence of operations, and time is turned off before reading the next block of traces from disk, and so on.

[3] Unfortunately, in the literature one frequently encounters papers where a very tightly coded implementation of the author's new method is compared with other relatively naive implementations of the previously existing methods, sometimes even run on machines with different characteristics that are difficult to compare.

This article is structured as follows. Section 2 discusses some design decisions taken after numerous discussions of the authors. Section 3 gives benchmarks for term retrieval and index maintenance for the problem of retrieval of generalizations (matching a query term by index terms), generated by running our provers Vampire [20], Fiesta [15] and Waldmeister [9] (three rather different, we believe quite representative, state-of-the-art provers) on a selection of carefully chosen problems from different domains of the TPTP library [25].

In Section 4 we describe the evaluation on these benchmarks of code trees, context trees, and discrimination trees as they are provided and integrated in the test package by their own implementors, and run under identical circumstances. As far as we know, this is the first time that different indexing data structures for deduction are compared under circumstances which, we believe, guarantee that experiments are not biased in any direction. Moreover, the implementations of code trees and discrimination trees we consider are the ones of the Vampire and Waldmeister provers, which we believe to be among the fastest (if not the fastest) current implementations for each one of these techniques. Hence for these two implementations it is unlikely that there are any differences in quality of coding. Although context trees are a new concept (see [3]) and the implementation used in the experiments (the only one that exists) was finished only one week before, the implementation is not naive since it is based on earlier, quite refined implementations of substitution trees.

All test programs, implementations and benchmarks mentioned in this paper are publicly available at `http://www.lsi.upc.es/~roberto`.

2 Some design decisions

In this section we discuss some decisions we had to take, since they may be helpful for the design of similar experiments for other term indexing techniques in the future.

We decided to concentrate in this paper on the measurement of three main aspects of indexing techniques (but the same methodology is applicable to other algorithms and data structures, see Section 6 for examples).

The first aspect we focus on is the time needed for queries where the retrieval condition is generalization: given a query term t, is there any indexed term l such that for some substitution σ we have $l\sigma = t$? This retrieval condition has at least two uses in first-order theorem provers: forward subsumption for unit clauses and forward demodulation (simplification by rewriting with unit equalities). It is also closely related (as an ingredient or prefilter) to general forward subsumption. It is well-known to be the main bottleneck in many provers (especially, but not only, in provers with built-in equality). Unlike some other retrieval conditions, search for generalizations is used in both general theorem provers and theorem provers for unit equalities. Some provers do not even index the other operations.

Though time is considered to be the main factor for system comparison, memory consumption is also crucial, especially for long-running tests. During the last years processor speed has grown much faster than memory capacity and it is foreseen that this trend will continue in the coming years. Moreover, memory access speed is also becoming an important bottleneck. Excessive memory consumption leads to more cache faults, which become the dominant factor for time, instead of processor speed. In experiments described in [27], run on a relatively slow computer with 1Gbyte of RAM memory, it was observed that some provers consume 800Mbytes of memory in the first 30 minutes. The best modern computers are already about 10 times faster than the computer used in these experiments, which means that memory problems arise very quickly, a problem that will become more serious in the coming years. For all these reasons, memory consumption is the second object of measurement in this paper.

The third and last aspect we focus on here is frequency of updates (insertions and deletions) and the time needed for them. In this field there exists an amount of *folk* knowledge (but which, unfortunately, differs among researchers), about the following questions. How frequent are updates in real applications? Is it really true that the time needed for updates is negligible? Is it worth or feasible to restructure larger parts of the index at update time (i.e., more than what is currently done)?

Perfect filtering or not? Our definition of term indexing corresponds to *perfect filtering*. In some cases implementation of *imperfect filtering*, when a subset or a superset of candidate terms is retrieved, does not affect soundness and completeness of a prover. In particular, neither soundness nor completeness are affected if only a subset of candidate terms is retrieved, when the retrieved generalizations are only used for subsumption or forward demodulation.

It was decided that only perfect filtering techniques should be compared. Of course, this includes implementations based on imperfect filters retrieving a superset of candidate terms, combined with a final term-to-term correctness test. Indeed, comparing imperfect indexing techniques alone does not make much sense in our context, since the clear winner in terms of time would be the system reporting "substitution not found" without even considering the query term.

Should the computed substitution be constructed in an explicit form? When search for generalizations is used for forward subsumption, computing the substitution is unnecessary. When it is used for forward demodulation, the computed substitution is later used for rewriting the query term. We decided that explicit representation of the computed substitution is unnecessary since all indexing techniques build such a substitution in an implicit form, and this implicit representation of the substitution is enough to perform rewriting.

One or all generalizations? In term indexing, one can search for one, some, or all indexed candidate terms. When indexing is used for forward subsumption or forward demodulation, computing all candidates is unnecessary. In the case of subsumption, if any candidate term is found, the query term is subsumed and can be discarded. In the case of forward demodulation, the first substitution found will be used for rewriting the query term, and there is no need either for finding more of them. Hence we decided to only search for one candidate. Note that for other retrieval conditions computing all candidate terms can be more appropriate. Examples are unification, used for inference computation, and (backward) *matching*, used for backward subsumption and backward demodulation.

How should problems be selected? It was decided for this particular experiment that every participant contributes with an equal number of benchmarks, and that the problems should be selected by each participant individually. There was a common understanding that the problems should be as diverse as possible, and that benchmarks should be large enough. Among the three participants, Vampire is the only prover that can run on nonunit problems, so to achieve diversity benchmarks generated by Vampire were taken from nonunit problems only. Nonunit problems tend to have larger signatures than the unit ones. Quite unexpectedly, benchmarks generated by Fiesta and Waldmeister happened to be quite diverse too, since Fiesta generates large terms much more often than Waldmeister.

But even if one collects such a diverse set of benchmarks, one cannot expect that all practical situations are covered by (statistically speaking) sufficiently many of them. For example, if one wants to check how a particular indexing technique behaves on large signatures, our benchmarks suite would be inappropriate since it contains only two benchmarks in which signatures are relatively large. Our selection of 30 problems is also inadequate if one wants to check how a particular technique behaves when the index contains over 10^5 terms, since the greatest number of terms in our indices is considerably less.

But it is one of the advantages of the proposed methodology that a potentially unlimited number of new benchmarks with any given properties can easily be generated, provided that these properties correspond to those occurring in real problems.

Input file format. Input file format was important for a reason not having direct relation to the experiments. The benchmark files, written in any format, are huge. Two input formats have been proposed: one uses structure sharing and refers to previously generated terms by their numbers; the other one uses a stringterm representation of query terms. In the beginning it was believed that the first format would give more compact files, but in practice this happened to be not the case since a great majority of query terms proved to be small. In addition, files with structure sharing did not compress well, so a stringterm representation was chosen, see Figure 1. Finally, stringterms are easy to read for humans which helped us a lot when one indexing data structure produced strange results because an input file was mistakenly truncated.

But even compressed files with benchmarks occupy hundreds of megabytes, which means that difficulties can arise to only store them in a publicly accessible domain and transfer them over the Web. Fortunately, this problem was easy to solve, since these files have a relatively low Kolmogorov complexity, an observation that, this time, can also be used in practice: they have been produced by three much smaller generators (provers) from very short inputs, so for reproducing the files it is enough to store the generators and their input files.

We also reconsidered our initial ideas about how to store the terms internally before calling the indexing operations. We experimented with complicated shared internal representations, in order to keep the memory consumption of the test program small (compared with the memory used by the indexing data structures), thus avoiding noise in the experiments due to cache faults caused by our test driver, since the driver itself might heavily occupy the cache. But finally we found that it is better for minimising cache faults to simply read relatively small blocks (of 2MB) of input from disk at a time, and store terms without any sharing, but *contiguously* in the same order as they will be considered in the calls to the indexing data structure. Of course the number of terms (i.e., of operations on the index) read from disk at a time should not be too small, because otherwise timing is turned on and off too often, which also produces noise.

This is an extract from a benchmark file generated by Waldmeister from the TPTP problem LCL109-2. Comments have been added by the authors.

```
a/2                    #   each benchmark file starts with the
b/0                    #   signature symbols with respective arities
c/1                    #
...
?ab0                   #  query term a(b,x0), "?" signals failure
?b                     #  query term b
+ab0                   #  insert term a(b,x0) to the index
...
!ab5                   #  query term a(b,x5), "!" signals success
...
-accbb                 #  delete term a(c(c(b)),b) from index
...
```

Fig. 1. An example benchmark file

Is the query term creation time included in the results? Despite the simplicity of this question, it was not easy to answer. Initially, it was thought that time for doing this should be negligible and essentially equal for the different indexing implementations. However, in actual provers the query terms are already available in an appropriate format as a result of some previous operation, so there is no need to measure the time spent on copying the query term. In addition, some, but not all, participants use the flatterm representation for query terms and it was believed that creating flatterm query terms from similarly structured stringterms could be several times faster than creating terms in the tree form. Hence it was decided that the time for creating query terms should not be included in the results.

However, this decision immediately created another problem. An expensive single operation for matching the query term against a particular indexed term is the comparison of two subterms of the query term. For example, to check if the term $f(x, x)$ is a generalization of a query term $f(s, t)$, one has to check whether s and t are the same term. Such a check can be done in time linear in the size of s, t. All other operations used in term-to-term matching can be performed in constant time (for example, comparison of two function symbols or checking if the query term is a variable). Now suppose that everyone can create any representation of the query term at the expense of system time. Then one can create a perfectly shared representation in which equal subterms will be represented by the same pointer, and checking for subterm equality becomes a simple constant time pointer comparison. Clearly, in practice one has to pay for creating a perfectly shared representation, so doing this in system time would be hardly appropriate[4]. The solution we have agreed upon is this: the representation of the query term should not allow for a constant-time subterm comparison, and for each participating system the part of the code which transforms the stringterm into the query term should be clear and easy to localize (e.g., included in the test driver itself).

3 Generation of Benchmarks

We took 30 problems from TPTP (10 by each participant), which created 30 benchmarks. The table of problems and simple quantitative characteristics of the resulting benchmarks is given in Table 1. The first two columns contain the name of the problem and the system that generated the problem. The third column contains the number of symbols in the signature of the problem, for example 3+4 means that the signature contains 3 nonconstant function symbols plus 4 constants. In the following four columns we indicate the total number of operations (*Total* in the table), insertions in and deletions from the index (*Ins* and *Del* in the table), and the maximal size of the index in number of terms (*Max* in the table) during the experiment. In the last four columns we show the average size and depth of the indexed and query terms, respectively. Here by size we mean the number of symbols in the term, and the depth is measured so that the depth of a constant is 0.

4 Evaluation

We ran each indexing data structure on each benchmark, i.e., we did 90 experiments. The results are given in Table 2. We measured time spent and memory used by each system. In parentheses we put the time measured for index maintenance only (i.e. insertions and deletions). This time was measured in a second run of the 90 experiments, on the same benchmarks, but with all retrieval requests removed.

[4] Note that the (forward) matching operation is applied to formulae just after they have been generated, with the purpose of eliminating or simplifying them. Hence it is unlikely that they are in perfectly shared form, even in systems in which retained formulae are kept in a perfectly shared representation.

problem and generator		sig	operations				indexed terms		query terms	
			Total	Ins	Del	Max	Size	Depth	Size	Depth
BOO015-4	wal	3+4	275038	228	228	179	13.8	5.8	5.1	2.2
CAT001-4	vam	3+4	2203880	18271	3023	16938	26.0	9.9	7.1	2.9
CAT002-3	vam	4+4	2209777	12934	4181	11191	29.2	10.4	7.1	3.1
CAT003-4	vam	3+4	2151408	18159	4476	16606	28.3	10.6	7.1	2.9
CIV003-1	vam	6+14	3095080	70324	22757	47567	14.1	5.1	4.3	1.6
COL002-5	fie	2+7	940127	13399	5329	8353	25.4	9.3	7.5	2.8
COL004-3	fie	2+5	1176507	765	28	737	18.1	6.7	9.3	3.2
COL079-2	vam	3+2	2143156	14236	4619	9633	38.8	11.8	7.3	2.7
GRP024-5	wal	3+4	2686810	506	506	296	16.3	6.6	7.7	2.9
GRP164-1	fie	5+4	11069073	53934	3871	50063	16.4	6.1	5.9	2.4
GRP179-2	fie	5+2	10770018	52825	2955	49870	16.8	6.1	6.1	2.4
GRP187-1	wal	4+3	9999990	2714	1327	1387	12.0	5.3	5.0	2.0
GRP196-1	fie	2+2	18977144	3	0	3	26.3	7.0	10.8	4.8
HEN011-2	vam	1+10	4313282	4408	439	3969	10.7	4.1	2.7	0.8
LAT002-1	vam	2+6	2646466	26095	1789	24306	17.5	6.2	5.6	2.0
LAT009-1	wal	2+3	2514005	596	596	291	19.2	7.4	7.8	2.8
LAT020-1	wal	2+3	9999992	910	493	417	14.8	5.6	9.3	3.2
LAT023-1	fie	3+4	822539	4088	2065	2499	18.4	6.6	6.0	2.3
LAT026-1	fie	3+4	772413	6162	3509	4770	20.8	7.1	5.6	2.0
LCL109-2	fie	3+4	312992	4465	519	3947	16.7	6.1	6.0	2.6
LCL109-2	wal	2+1	463493	196	196	165	19.2	7.9	5.7	2.3
LCL109-4	vam	4+3	1944335	40949	3135	37817	20.0	5.7	8.1	2.7
RNG020-6	fie	6+6	2107343	4872	960	3912	17.4	5.4	6.3	2.4
RNG028-5	wal	5+4	3221510	304	304	218	31.4	9.0	14.0	3.8
RNG034-1	vam	4+4	2465088	15068	4589	11685	26.4	7.0	6.1	2.4
RNG035-7	wal	3+5	5108975	482	482	360	21.7	9.0	13.9	4.7
ROB006-2	wal	3+4	9999990	1182	34	1148	19.1	7.7	12.4	4.9
ROB022-1	fie	3+4	922806	2166	826	1341	21.4	9.3	6.5	2.8
ROB026-1	wal	2+4	9999991	648	15	633	16.9	7.9	12.6	5.0
SET015-4	vam	4+6	3664777	3256	995	2261	16.3	5.8	3.6	1.4

Table 1. Benchmark characteristics

Time. Vampire's implementation of code trees is on the average 1.39 times than Waldmeister's implementation of discrimination trees and 1.91 times faster than the current implementation of context trees[5]. These factors roughly hold for almost all problems, with a few exceptions, for example on problem CIV003-1 Waldmeister is slightly faster than Vampire. The time spent for index maintenance is in all cases negligible compared to the retrieval time, and there are essentially no fluctuations from the average figures: Waldmeister spends on the index maintenance 1.18 times less than Vampire and 1.49 times less than the context trees implementation.

Memory. In memory consumption, differences are more important. On average, the implementation of context trees used 1.18 times less memory than Vampire's implementation of code trees and 5.39 times less memory than Waldmeister's implementation of discrimination trees.

5 A short interpretation of the results

Although the main aim of this work was to design a general-purpose technique for measuring and comparing the efficiency of indexing techniques in time and space requirements, in this section we very briefly and globally describe why we believe these concrete experiments have produced these concrete results.

5.1 Waldmeister's discrimination trees

Waldmeister uses *discrimination trees*, which are like tries where terms are seen as strings and common prefixes are shared, in its so-called *perfect* variant. Nodes are arrays of pointers and can have different sizes: each node can have a child for each one of the function symbols and for each one of the variables that already occurred along the path in the tree, plus an additional child for a possible new variable. During matching one can index the array by the currently treated query symbol f, and directly jump to the child for f, if it exists, or to one of the variable children. Note that the case where children for both f and some variable exist (or for more than one variable), is the only situation where backtracking points are created. Usually, queries are represented as the so-called *flatterms* of [2], which are linked lists with additional pointers to jump over subterms t when a variable of the index gets instantiated by t.

The results of our experiments for Waldmeister's discrimination trees are not unexpected. The implementation is very tightly coded, and in spite of the lower

[5] Note that in these average times problems with large absolute run-times predominate. Furthermore, context trees are new (see [3]) and several important optimizations for them have not yet been implemented. According to the first author, they are at least 3 times faster than the substitution trees previously used in Fiesta. We welcome anyone who has an efficient implementation of substitution trees (or any other indexing data structure) to compare her or his technique with ours on the same benchmarks.

		time (in seconds)			memory (in Kbytes)		
problem	from	Vam	Wal	Con	Vam	Wal	Con
BOO015-4	wal	0.25 (0.00)	0.31 (0.01)	0.46 (0.01)	11	575	11
CAT001-4	vam	3.28 (0.34)	5.74 (0.31)	7.11 (0.41)	3859	13786	3109
CAT002-3	vam	2.90 (0.22)	5.51 (0.31)	6.67 (0.33)	2483	9281	2021
CAT003-4	vam	3.21 (0.34)	5.82 (0.34)	6.90 (0.46)	3826	13595	3086
CIV003-1	vam	7.57 (0.93)	7.13 (0.65)	15.57 (1.16)	3754	22664	3081
COL002-5	fie	1.30 (0.17)	1.55 (0.21)	2.61 (0.28)	925	6090	922
COL004-3	fie	0.96 (0.00)	1.22 (0.00)	2.39 (0.02)	80	727	86
COL079-2	vam	5.46 (0.29)	8.41 (0.26)	7.24 (0.43)	2769	9158	2138
GRP024-5	wal	3.54 (0.01)	4.82 (0.00)	7.44 (0.01)	19	591	22
GRP164-1	fie	17.60 (0.72)	24.60 (0.61)	32.06 (0.89)	5823	28682	5352
GRP179-2	fie	18.34 (0.71)	24.25 (0.60)	32.40 (0.87)	5597	29181	5207
GRP187-1	wal	10.44 (0.02)	11.68 (0.03)	17.64 (0.02)	96	903	97
GRP196-1	fie	6.96 (0.00)	11.92 (0.00)	15.45 (0.00)	1	543	1
HEN011-2	vam	3.36 (0.03)	3.39 (0.02)	5.18 (0.04)	221	2069	211
LAT002-1	vam	5.83 (0.32)	7.72 (0.29)	9.48 (0.44)	3164	14603	2554
LAT009-1	wal	3.78 (0.01)	4.97 (0.01)	5.97 (0.01)	19	591	20
LAT020-1	wal	17.73 (0.01)	24.97 (0.01)	29.87 (0.01)	30	631	31
LAT023-1	fie	1.10 (0.04)	1.49 (0.03)	1.92 (0.07)	198	1646	210
LAT026-1	fie	1.11 (0.09)	1.49 (0.07)	1.79 (0.12)	373	2813	371
LCL109-2	fie	0.47 (0.04)	0.65 (0.06)	0.80 (0.05)	508	2285	466
LCL109-2	wal	0.49 (0.00)	0.66 (0.00)	0.82 (0.00)	16	591	15
LCL109-4	vam	5.62 (0.70)	7.65 (0.46)	13.02 (0.72)	6703	24403	4986
RNG020-6	fie	2.25 (0.07)	3.19 (0.05)	5.33 (0.08)	544	2435	517
RNG028-5	wal	4.19 (0.01)	6.66 (0.01)	9.08 (0.01)	28	607	29
RNG034-1	vam	3.27 (0.33)	4.95 (0.21)	6.86 (0.34)	2545	8330	2125
RNG035-7	wal	8.19 (0.01)	12.10 (0.01)	18.55 (0.01)	36	647	37
ROB006-2	wal	9.88 (0.01)	14.31 (0.02)	21.60 (0.02)	128	1142	116
ROB022-1	fie	0.92 (0.03)	1.20 (0.03)	2.23 (0.03)	119	1086	101
ROB026-1	wal	8.52 (0.01)	13.35 (0.01)	17.34 (0.01)	69	807	68
SET015-4	vam	2.54 (0.02)	2.69 (0.02)	4.53 (0.05)	314	1373	258
		total			total		
		161.09 (5.48)	224.39 (4.64)	308.31 (6.90)	44258	201835	37248

In this table, we list the benchmarks, named after the TPTP problem they come from, along with the prover run on the given problem. Vam stands for Vampire's code tree implementation, Wal for Waldmeister's discrimination trees, and Con for the context trees implementation. Between parentheses, times without retrievals, i.e., only insertions and deletions.

Table 2. Results

amount of sharing than in other techniques, the retrieval speed is very high because the low-level operations (essentially, symbol comparison and variable instantiation) are very simple. It is clear that, especially for larger signatures and deep terms, the kind of nodes used leads to high memory consumption. Memory consumption is even higher because the backtracking stack needed for retrieval is inscribed into the tree nodes, enlarging them even more. This speeds up retrieval at the cost of space.

5.2 Context trees

As we have mentioned, in discrimination trees common prefixes are shared. In *substitution trees* [6], terms keep their tree structure and all common *contexts* can be shared (note that this includes the comon prefixes of the terms seen as strings). *Context trees* are a new indexing data structure, where, by means of a limited kind of context variables, also common subterms can be shared, even if they occur below different function symbols (see [3] for all details). More sharing allows one to avoid repeated work (this is of course the key to all indexing techniques).

The basic idea is the following. Assume one has three terms $h(x, f(t))$, $h(x, g(t))$, and $h(b, f(t))$ in the index, and let the query be $h(b, f(s))$ where the terms s and t are large and s is not an instance of t. In a discrimination tree (and in a substitution tree) t will occur three times, and two repeated attempts will be made for matching s against t. In a context tree, the root contains $h(x_1, F(t))$ (where F can be instantiated by a single function symbol of any arity) and an immediate failure will occur without exploring any further nodes.

These experiments were run on a first implementation (which does not yet include several important enhancements, see [3]), based on curried terms and on an extension of substitution trees with equality constraints and where one does not distinguish between *internal* and *external* variables. Due to the high amount of sharing, context trees need little space, while still being suitable for compiling (see below).

5.3 Vampire's code trees

On one hand, Vampire's code trees can be seen as a form of compiled discrimination trees. One of the important conclusions that can be drawn from the experiments is that it pays off to compile an index into a form of interpreted abstract instructions (similar findings have also been obtained in the field of logic programming). Consider for instance a typical algorithm for term-to-term matching `TermMatch(query,set)`. It is clear that the concrete term `set` is known at compile (i.e. index update) time, and that a specialized algorithm `TermMatchWithSet(query)` can be much more efficient. If the whole index tree is compiled, then also all control instructions become unnecessary (asking whether a child exists, or whether a sibling exists, or whether the node is leaf, etc.). These advantages give far more speedup than the overhead for interpreting the operation code of the abstract instructions (which can be a constant time computed `switch` statement).

But code trees are more than compiled discrimination trees. Let us first consider *standard* discrimination trees, where all variables are represented as a general variable symbol $*$, e.g., different terms like $f(x, y)$ and $f(x, x)$ are both seen as $f(*, *)$, and the corresponding path in the tree is common to both. This increases the amount of sharing, and also the retrieval speed, because the low-level operations are simpler. But it is only a *prefilter*: once a possible match has been found, additional equality tests have to be performed between the query subterms by which the variables of terms like $f(x, x)$ have been instantiated. One important optimization is to perform the equality tests not always in the leaves, but as high up as possible in the tree *without decreasing the amount of sharing*. This is how it is done in partially adaptive code trees [21]. Doing the equality tests in the leaves would frequently lead to repeated work during retrieval time. Also, according to the first-fail principle (which is often beneficial in indexing techniques), it is important to impose strong restrictions like the equality of two whole subterms as soon as possible. Due to the fact that it is conceptually easier to move instructions than to restructure a discrimination tree, code trees are very adequate for this purpose. Finally, code is also compact and hence space-efficient, because one complex instruction can encode (matching with) a relatively specific term structure.

6 Conclusions and Related Work

In Graf's book on term indexing [6], the benchmarks are a small number of sets of terms (each set having between 500 and 10000 terms), coming from runs with Otter [12]. In other experiments also randomly generated terms are used. In all of Graf's experiments first the index is built from one set and then retrieval operations are done using as queries all terms of another set over the same signature. As said, the drawback of such an analysis is that it is unclear how representative the sets of Otter terms are and how frequent updates are in relation with retrieval operations. In addition, in real provers updates are interleaved with queries, which makes it difficult to construct optimal indexes, especially in the case of highly adaptive structures such as substitution trees or context trees. Furthermore, in such experiments it is usually unclear whether the quality of coding of the different techniques is comparable. Although the quality of code in our provers can also be questioned, at least the authors of the discrimination tree and code tree implementations believe that their code is close to optimal. Note that, unlike all previously published papers, the systems participating in the experiments are competitors, and benchmarks were generated by each of them independently, so it is unlikely that the results are biased toward a particular technique.

We expect more researchers to use our benchmarks and programs, which are available at `http://www.lsi.upc.es/~roberto`, and report on the behaviour of new indexing techniques. A table of results will be maintained as well at that web site.

Other algorithms and data structures for indexing could be compared using our framework; let us mention but a few:

1. retrieval of instances (used in backward subsumption by unit clause);
2. retrieval of instances on the level of subterms (used in backward demodulation);
3. retrieval of unifiable terms;
4. forward subsumption on multiliteral clauses;
5. backward subsumption on multiliteral clauses.

Other interesting algorithms are related to the use of orderings:

1. comparison of terms or literals in the lexicographic path order or the Knuth-Bendix order.
2. retrieval of generalizations together with checking that some ordering conditions are satisfied.

We suggest anyone interested to contact the authors on the design of benchmark suites for these problems within the framework of this paper.

Acknowledgments

We thank Jürgen Avenhaus for valuable remarks to yesterday's version of this paper which was as a result substantially rewritten today.

References

1. P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In R. Reiter J. Mylopoulos, editor, *IJCAI 1991. Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 331–340, Sydney, Australia, 1991. Morgan Kaufmann.
2. J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, February 1993.
3. H. Ganzinger, R. Nieuwenhuis, and P. Nivela. Context trees. In *IJCAR 2001, Proceedings of the International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, Siena, Italy, June 2001. Springer Verlag.
4. P. Graf. Extended path-indexing. In A. Bundy, editor, *CADE-12. 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 514–528, Nancy, France, June/July 1994.
5. P. Graf. Substitution tree indexing. In J. Hsiang, editor, *Procs. 6th International Conference on Rewriting Techniques and Applications (RTA-95)*, volume 914 of *Lecture Notes in Computer Science*, pages 117–131, Kaiserslautern, 1995.
6. P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.
7. J. Gu, P.W. Purdom, J.Franco, and B.W. Wah. *Algorithms for the Satisfiability Problems*. Cambridge University Press, 2001.

8. C. Hewitt. *Description and theoretical analysis of Planner: a language for proving theorems and manipulating models in a robot.* PhD thesis, Department of Mathematics, MIT, Cambridge, Mass., January 1971.
9. T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. Waldmeister: High-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.
10. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
11. W.W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
12. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.
13. D.G. Mitchell, B. Selman, and H.J. Levesque. Hard and easy distributions of SAT problems. In W.R. Swartout, editor, *Procs. 10th National Conference on Artificial Intelligence*, pages 459–465, San Jose, CA, January 1992. AAAI Press/MIT Press.
14. R. Nieuwenhuis. Rewrite-based deduction and symbolic constraints. In H. Ganzinger, editor, *CADE-16. 16th Int. Conf. on Automated Deduction*, Lecture Notes in Artificial Intelligence, pages 302–313, Trento, Italy, July 1999.
15. R. Nieuwenhuis, J.M. Rivero, and M.Á. Vallejo. The Barcelona prover. *Journal of Automated Reasoning*, 18(2):171–176, 1997.
16. H.J. Ohlbach. Abstraction tree indexing for terms. In H.-J. Bürkert and W. Nutt, editors, *Extended Abstracts of the Third International Workshop on Unification*, pages 131–135. Universität Kaiserslautern, 1989. SEKI-Report SR 89-17.
17. M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.
18. P.W. Purdom and C.A. Brown. Fast many-to-one matching algorithms. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, First International Conference, RTA-85*, volume 202 of *Lecture Notes in Computer Science*, pages 407–416, Dijon, France, 1985. Springer Verlag.
19. I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1–97. Elsevier Science and MIT Press, 2001. To appear.
20. A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *CADE-16. 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, July 1999.
21. A. Riazanov and A. Voronkov. Partially adaptive code trees. In M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka, and L.M. Pereira, editors, *Logics in Artificial Intelligence. European Workshop, JELIA 2000*, volume 1919 of *Lecture Notes in Artificial Intelligence*, pages 209–223, Málaga, Spain, 2000. Springer Verlag.
22. J.M.A. Rivero. *Data Structures and Algorithms for Automated Deduction with Equality.* Phd thesis, Universitat Politècnica de Catalunya, Barcelona, May 2000.
23. S. Schulz. *Learning Search Control Knowledge for Equational Deduction*, volume 230 of *Dissertationen zur künstliche Intelligenz.* Akademische Verlagsgesellschaft Aka GmmH, 2000.
24. M. Stickel. The path indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.
25. G. Sutcliffe and C. Suttner. The TPTP problem library — CNF release v. 1.2.1. *Journal of Automated Reasoning*, 21(2), 1998.
26. A. Voronkov. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.
27. A. Voronkov. CASC $16\frac{1}{2}$. Preprint CSPP-4, Department of Computer Science, University of Manchester, February 2000.