Fingerprint Indexing for Paramodulation and Rewriting

Stephan Schulz

Institut für Informatik, Technische Universität München,
D-80290 München, Germany
`schulz@eprover.org`

Abstract. Indexing is critical for the performance of first-order theorem provers. We introduce *fingerprint indexing*, a non-perfect indexing technique that is based on short, constant length vectors of samples of term positions ("fingerprints") organized in a trie. Fingerprint indexing supports matching and unification as retrieval relations. The algorithms are simple, the indices are small, and performance is very good in practice.

We demonstrate the performance of the index both in relative and absolute terms using large-scale profiling.

1 Introduction

Saturating theorem provers like Vampire [3] and E [5] are among the most powerful ATP systems for first-order logic. These systems work in a refutational setting. The state of the proof search is represented by a set of clauses. It is manipulated using inference rules. The most important inferences (resolution and paramodulation/superposition) and simplifications (rewriting and subsumption) use two or more premises—usually a main premise and one or more additional side premises. This requires the system to find potential inference partners for a given clause in the potentially large set of clauses representing the search state.

The performance can be improved if inference partners are not found by sequential search, but via an *index*. An index, in this context, is a data structure with associated algorithms that allows the efficient retrieval of terms or clauses from the indexed set that are in a given *retrieval relation* with a query.

E has featured *(perfect) discrimination tree indexing* [2] for forward rewriting since version 0.1. It added *feature vector indexing* [6] for subsumption in version 0.8. In this paper, we present *fingerprint indexing*, a new, non-perfect term indexing technique that can be seen as a generalization of top symbol hashing. It shares the basic structure of feature vector indexing (indexed objects are represented by finite-length vectors organized in a trie), and combines it with ideas from coordinate and path indexing [7,2,1] (values in the index vectors represent the occurrence of symbols at certain positions in terms). The index can be used for retrieving candidate terms unifiable with a query term, matching a query term, or being matched by a query term. The index data structure has very low memory use, and all operations for maintenance and candidate retrieval are fast in practice. Variants of fingerprint indexing have been incorporated into E for backwards simplification and superposition, with generally positive results.

2 Background

We use standard terminology for first-order logic. A signature consists of a finite set F of function symbols with associated arities. We write $f|_n$ to indicate that f has arity $n \in \mathbb{N}_0$. We assume an enumerable set V of *variables* disjoint from F, typically denoted by $x, y, z, x_0, \ldots$, or by upper-case `X0`,`X1` if represented in TPTP syntax. Terms, subterms, literals and clauses are defined as usual.

A *substitution* is a mapping $\sigma : V \rightarrow \mathit{Term}(F, V)$ with the property that $\mathit{Dom}(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$ is finite. It can be extended to terms, atoms, literals and clauses. A *matcher* from a term s to another term t is a substitution σ such that $\sigma(s) \equiv t$. A *unifier* of two terms s and t a substitution σ such that $\sigma(s) \equiv \sigma(t)$. If s and t are unifiable, a *most general unifier (mgu)* for them exists, and is unique up to variable renaming.

A *(potential) position* in a term is a sequence $p \in \mathbb{N}^*$ over natural numbers. We use ϵ to denote the empty position. The set of positions in a term, $\mathrm{pos}(t)$ is defined as follows: If $t \equiv x \in V$, then $\mathrm{pos}(t) = \{\epsilon\}$. Otherwise $t \equiv f(t_1, \ldots, t_n)$. In this case $\mathrm{pos}\,(t) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n, p \in \mathrm{pos}(t_i)\}$. The subterm of t at position $p \in \mathrm{pos}(t)$ is defined recursively: if $p = \epsilon$, then $t|_p = t$. Otherwise, $p \equiv i.p'$ and $t \equiv f(t_1, \ldots, t_n)$. In that case, $t_p = t_i|_{p'}$. The top symbol of $x \in V$ is $\mathrm{top}(x) = x$ and the top symbol of $f(t_1, \ldots, t_n)$ is $\mathrm{top}(f(t_1, \ldots, t_n)) = f$.

Positions can be extended to literals (selecting a term in a literal) and clauses (selecting a term in a literal in a clause) easily if we assume an arbitrary, but fixed ordering of terms in literals and literals in clauses.

Modern saturating calculi are instantiated with a *term ordering*. This ordering is lifted to literals and clauses. Generating inferences can be restricted to (subterms of) maximal terms of maximal literals. Simplification allows the replacement of clauses with equivalent smaller clauses.

3 Fingerprint Indexing

We will now introduce *fingerprints* of terms, and show that the compatibility of the respective fingerprints is a required condition for the existence of a unifier (or matcher) between two terms. The basic idea is that the application of a substitution never removes an existing position from a term, nor will it change an existing function symbol in a term.

Consider a potential position p and a term t (as a running example, assume $t = g(f(x, a))$. Then the following cases are possible:

1. p is a position in t and $t|_p$ is a variable (e.g. $p = 1.1$)
2. p is a position in t and $t|_p$ is a non-variable term (e.g. $p = 1.2$ or $p = \epsilon$)
3. p is not a position in t, but there exists an instance $\sigma(t)$ with $p \in \mathrm{pos}(\sigma(t))$ (e.g. $p = 1.1.1.2$ with $\sigma = \{x \mapsto f(a, b))\}$
4. p is not a position in t or any of its instances (e.g. $p = 2.1$)

These four cases have different implications for unification. Two terms which, at the same position, have different function symbols, cannot be unified. Stated positively, if we search for terms unifiable with a query term t, and $\text{top}(t|_p) = f$, we only need to consider terms s where $\text{top}(s|_p)$ can potentially become f.

To formalize this, consider the following definition: Let $F' = F \uplus \{\mathbf{A}, \mathbf{B}, \mathbf{N}\}$ (the set of *fingerprint feature values* for F). The *general fingerprint feature function* is a function gfpf : $\mathit{Term}(F, V) \times \mathbb{N}^* \to F'$ defined by:

$$\text{gfpf}(t,p) = \begin{cases} \mathbf{A} & \text{if } p \in \text{pos}(t),\ t|_p \in V \\ \text{top}(t|_p) & \text{if } p \in \text{pos}(t),\ t|_p \notin V \\ \mathbf{B} & \text{if } p = q.r,\ q \in \text{pos}(t) \text{ and } t|_q \in V \text{ for some } q \\ \mathbf{N} & \text{otherwise} \end{cases}$$

A *fingerprint feature function* is a function fpf : $\mathit{Term}(F, V) \to F'$ defined by $\text{fpf}(t) = \text{gfpf}(t,p)$ for a fixed $p \in \mathbb{N}^*$.

Now assume two terms, s and t, and a fingerprint feature function fpf. Assume $u = \text{fpf}(s)$ and $v = \text{fpf}(t)$. The values u and v are *compatible for unification* if they are marked with a **Y** in the *Unification* table of Figure 1. They are *compatible for matching from s onto t*, if they are marked with a **Y** in the *Matching* table in Figure 1. It is easy to show by case distinction that compatibility of the fingerprint feature values is a necessary condition for unification or matching.

Unification

	f_1	f_2	**A**	**B**	**N**
f_1	**Y**	**N**	**Y**	**Y**	**N**
f_2	**N**	**Y**	**Y**	**Y**	**N**
A	**Y**	**Y**	**Y**	**Y**	**N**
B	**Y**	**Y**	**Y**	**Y**	**Y**
N	**N**	**N**	**N**	**Y**	**Y**

Matching

	f_1	f_2	**A**	**B**	**N**
f_1	**Y**	**N**	**N**	**N**	**N**
f_2	**N**	**Y**	**N**	**N**	**N**
A	**Y**	**Y**	**Y**	**N**	**N**
B	**Y**	**Y**	**Y**	**Y**	**Y**
N	**N**	**N**	**N**	**N**	**Y**

Fig. 1. Fingerprint feature compatibility for unification and matching (down onto across). f_1 and f_2 are arbitrary but distinct.

Now assume $n \in \mathbb{N}$. A *fingerprint function* is a function fp : $\mathit{Term}(F, V) \to (F')^n$ with the property that $\pi^i_n \circ \text{fp}$ (the projection onto the ith element of the result) is a fingerprint feature function for all $i \in \{1, \ldots, n\}$. A *fingerprint* is the result of the application of a fingerprint function to a term, i.e. a vector of n elements over F'. We will in the following assume a fixed fingerprint function fp.

Two fingerprints for s and t are unification-compatible (or compatible for matching from s onto t) if they are component-wise so compatible.

Theorem 1. *Assume an arbitrary fingerprint function* fp. *If* $\text{fp}(t_1)$ *and* $\text{fp}(t_2)$ *are not unification compatible, then t_1 and t_2 are not unifiable. If* $\text{fp}(t_1)$ *and* $\text{fp}(t_2)$ *are not compatible for matching t_1 onto t_2, then t_1 does not match t_2.*

A fingerprint function defines an equivalence on $\mathit{Term}(F, V)$, and we can use the fingerprints to organize any set of terms into disjoint subsets, each sharing a fingerprint. If we want to find terms in a given relation (unifiable or matchable)

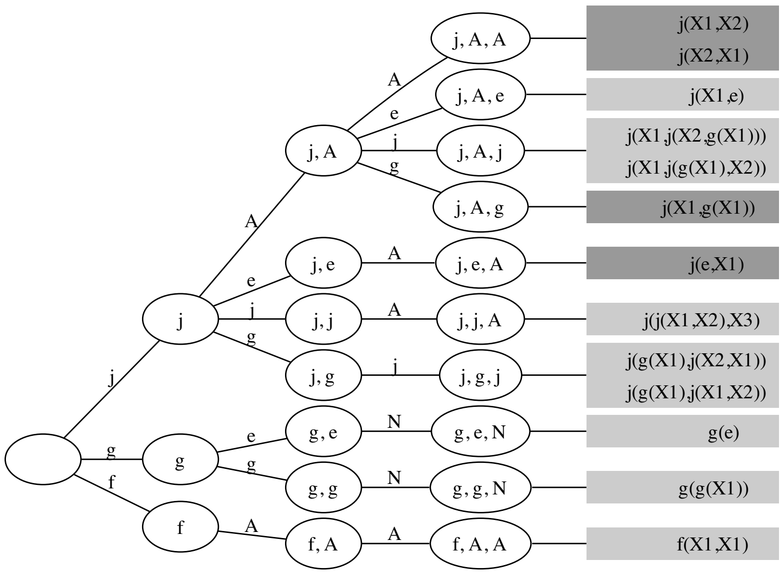

Fig. 2. Example fingerprint index

to a query term, we only need to consider terms from those subsets for which the query relation holds on the fingerprints.

However, we do not need to linearly compare fingerprints to find unification or matching candidates. A *fingerprint index* is a constant-depth *trie* over fingerprints that associates the indexed term sets with the leaves of the trie.

As an example, consider $F = \{j|_2, f|_2, g|_1, a|_0, b|_0, e|_0\}$ and fp : $\mathit{Term}(F, V) \to (F')^3$ defined by $\text{fp}(t) = \langle \text{gfpf}(t, \epsilon), \text{gfpf}(t, 1), \text{gfpf}(t, 2)\rangle$.

Figure 2 shows a fingerprint index for fp. When we query the index for terms unifiable with $t = j(e, g(X)))$, we first compute $\text{fp}(t) = \langle j, e, g\rangle$. At each node in the tree we follow all branches labeled with feature values unification-compatible with the corresponding fingerprint value of the query. At the root, only the branch labelled with j has to be considered. At the next node, branches e and $\mathbf{A}$ are compatible. Finally, three leaves (marked in darker gray), with a total of 4 terms, are unification-compatible with the query. In this case, all 4 terms found actually are unifiable with the query.

Even fairly short fingerprints are sufficient to achieve good performance of the index. Computing these small fingerprints is computationally cheap, and so is insertion and removal of fingerprints from the trie.

Since each term has a unique fingerprint, it is stored at exactly one leaf. To find all retrieval candidates, we traverse the trie recursively, collecting candidates from all leaves. Since all terms at a leaf are compatible with all fingerprints leading to it, and since all terms are represented at most once in the index, we only need to form the union of the candidate sets at all matching leaves. This is a major advantage compared to coordinate indexing, where it is necessary to compute the intersection of candidate sets for each coordinate. The same applies to path indexing, where e.g. Vampire goes to great lengths to optimize this bottleneck [4]. Moreover, since each term has a single fingerprint and is represented only once in the trie, there are at most as many fingerprints in an index as there are indexed terms. Thus, memory consumption of the index scales at worst linearly with the number of indexed terms.

4 Implementation

We have implemented fingerprint indexing in our theorem prover E to speed up superposition and backwards rewriting. For this purpose, we have added three global indices to E, called the *backwards-rewriting index*, the *paramodulation-from index*, and the *paramodulation-into* index.

The *backwards-rewriting index* contains all (potentially rewritable) subterms of processed clauses. Each term is associated with the set of all processed clauses it occurs in. Given a new unit clause $l \simeq r$, we find all rewritable clauses by finding all leaves compatible with the fingerprint of l, try to match l onto each of the terms t stored at the leaf, and, in the case of success, verify if $\sigma(l) > \sigma(r)$. If and only if this is the case, all clauses associated with the term are rewritable with the new unit clause (and hence are removed from the set of processed clauses). Note that in this implementation the (potentially expensive) ordering check only has to be made once for every t, not once per occurrence of t.

For the paramodulation indices, we use a somewhat more complex structure. The FP-Trie indexes sets of terms with the same fingerprint. For each term, we store a set of clauses in which this term occurs (at a position potentially compatible with the superposition restrictions). Finally, with each of these clauses, we store the positions in which the term occurs. The paramodulation-into index is organized analogously.

5 Experimental Results

To measure the performance of fingerprint indexing, we performed a series of experiments. All tests use problems from the set of 15386 untyped first-order problems in the TPTP problem library [8], Version 5.2.0. The full test data and the version of the prover used for the test runs are archived at `http://www.eprover.eu/E-eu/FPIndexing.html`. All tests were run with a time limit of 300 seconds on 2.4 GHz Intel Xeon CPUs under the Linux 2.6.18-164.el5 SMP Kernel in 64 bit mode.

We have instrumented the prover by adding profiling code to measure the time spent in parts of the program without the overhead of a standard profiler. For quantitative analysis of the run times, we only use cases where the proof search followed very similar lines for the indexed and non-indexed case (as evidenced by clause counts). This resulted in 5824 problems used for comparison.

Table 1. CPU times for different parts of the proof process (in seconds)

Index	Run time	Sat time	PM time	PMI time	MGU time	BR time	BRI time
NoIdx	16062.392	14078.300	8980.320	0.000	2545.080	2280.250	0.000
FP0	16644.127	14835.130	9904.120	26.380	4360.330	1846.280	41.440
FP0FP	9581.606	8211.010	3633.590	27.950	1322.530	1071.210	42.030
FP1	7006.758	6145.870	1816.100	25.710	450.760	379.570	40.150
FP2	6200.043	5556.330	1345.440	28.900	199.600	104.340	43.300
FP3D	6107.780	5463.240	1266.820	31.410	150.880	91.430	46.040
FP4M	6050.617	5423.820	1197.720	33.640	109.870	64.740	49.620
FP5M	6088.364	5455.180	1203.240	38.250	107.860	65.630	53.520
FP6M	6000.177	5385.810	1181.710	38.240	99.110	39.010	55.660
FP7	6022.196	5404.150	1179.250	41.880	95.880	38.400	57.610
FP8X2	6066.482	5429.390	1193.820	56.430	88.580	37.710	77.400
NPDT	6082.246	5434.760	1184.750	64.910	83.110	33.200	79.910

We include results for a number of different versions: NoIdx (no indexing), FP0 (pseudo-fingerprint of lengths 0), FP0FP (pseudo-fingerprint emulating optimizations in the unindexed version), FP1 (sampling at ϵ, equivalent to top-symbol hashing, FP2 (ϵ, 1), FP3D (ϵ, 1, 1.1), FP4M (ϵ, 1, 2, 1.1), FP5M ($\epsilon, 1, 2, 3, 1.1$), FP6M ($\epsilon, 1, 2, 3, 1.1, 1.2$), FP7 ($\epsilon$, 1, 2, 1.1, 1.2, 2.1, 2.2), FP8X2 (ϵ, 1, 2, 3, 4, 1.1, 1.2, 1.3, 2.1, 2.2, 2.3, 3.1, 3.2, 3.3, 1.1.1, 2.1.1), NPDT (non-perfect discrimination trees).

Table 1 shows the result of the time measurements, summed over all 5824 problems. "Run time" is the total run time of the prover. "Sat time" is spent in the main saturation loop, "PM time" is for paramodulation/superposition, "PMI time" is for paramodulation index maintenance. "MGU time" is the time for unification. "BR time" and "BRI time" are the times used for backwards-rewriting and backward-rewrite index maintenance.

Comparing the unindexed version with the FP6M index, total run time decreases by more than 60%. The time spent for unification itself has been reduced by a factor of about 25. The total time for unification-related code (i.e. index maintenance and unification) amounts to less than 2.5% of the total run time.

Comparing the times for FP6M with the times for discrimination tree indexing, we see that overall fingerprint indexing outperforms discrimination tree indexing, if not by a large margin. Time for actual unification is slightly lower for discrimination tree indexing, but index maintenance is slightly more expensive.

We see an even stronger improvement for backwards rewriting. The time for the operation itself drops more than 58-fold. Time for index maintenance is of the same order of magnitude. Taking index maintenance into account, the total time for backward rewriting improved by a factor of about 25.

Figure 3(a) shows a scatter plot of run times for NoIdx and FP6M. Please note the double logarithmic scale. For the vast majority of problems, the indexed version is significantly, and often dramatically, faster, while there are no problems for which the conventional version is more than marginally faster. Figure 3(b) compares FP6M and discrimination tree indexing. For most problems, performance is nearly identical.

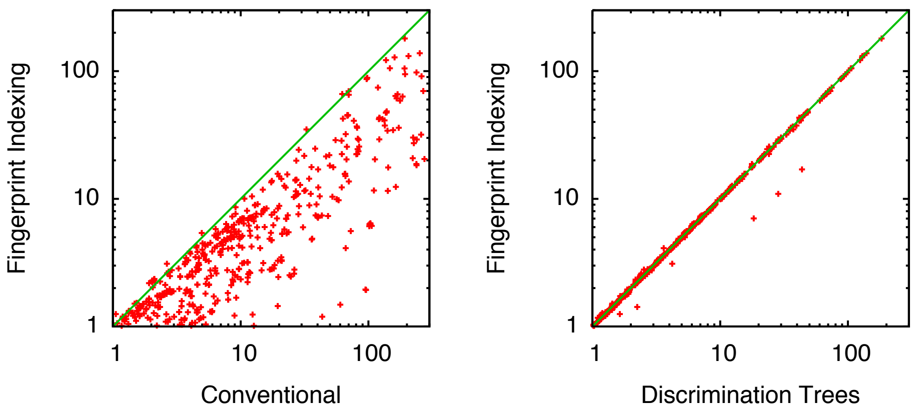

Fig. 3. Scatter plots of run times (in seconds) for FP6M over (a) non-indexed and (b) non-perfect discrimination tree implementations

6 Conclusion

In this paper, we have introduced *fingerprint indexing*, a lightweight indexing technique that is easy to implement, has a small memory footprint, and shows excellent performance in practice.

In the future, we will further investigate the influence of different fingerprint functions, and evaluate if further gains can be made by automatically generating a good fingerprint function based on the signature.

Acknowledgements. I thank the University of Miami's Center for Computational Science HPC team for making their cluster available for the experimental evaluation.

References

1. Graf, P.: Term Indexing. LNCS (LNAI), vol. 1053. Springer, Heidelberg (1996)
2. McCune, W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Journal of Automated Reasoning 9(2), 147–167 (1992)
3. Riazanov, A., Voronkov, A.: The Design and Implementation of VAMPIRE. Journal of AI Communications 15(2/3), 91–110 (2002)
4. Riazanov, A., Voronkov, A.: Efficient Instance Retrieval with Standard and Relational Path Indexing. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 380–396. Springer, Heidelberg (2003)
5. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications 15(2/3), 111–126 (2002)
6. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: Proc. of the IJCAR 2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland (2004)
7. Stickel, M.E.: The Path-Indexing Method for Indexing Terms. Technical Note 473, AI Center, SRI International, Menlo Park, California, USA (October 1989)
8. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)