Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval*

William McCune

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4801

November 6, 1990

Abstract

This article addresses the problem of indexing and retrieving first-order predicate calculus terms in the context of automated deduction programs. The four retrieval operations of concern are to find variants, generalizations, instances, and terms that unify with a given term. Discrimination-tree indexing is reviewed, and several variations are presented. The path-indexing method is also reviewed. Experiments were conducted on large sets of terms to determine how the properties of the terms affect the performance of the two indexing methods. Results of the experiments are presented.

1 Introduction

As automated deduction systems begin to emerge as useful vehicles for studying questions in mathematics and logic, the speed of those systems is becoming more and more important. Given the combinatorial explosion of typical searches, a large speedup in itself will usually not enable programs to find direct proofs of much more difficult theorems. However, reduced response time allows researchers to more easily interact with the program by evaluating failures and trying additional searches with different strategies or axiom sets. Improvement of speed by a factor of ten, five, or even two can easily cause a success.

Careful attention to term indexing in our automated deduction program OTTER [16] has caused substantial speedups. Two different indexing methods are used in OTTER:

---

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

discrimination-tree indexing and path indexing. Initial implementations and experiments showed that some of the indexing operations are usually better performed by discrimination-tree indexing and others by path indexing. In addition, some types of term are much better handled by one or the other of the indexing methods. Thus, no clear winner existed between the two methods. More extensive experiments were then conducted on sets of terms from several application areas. This article contains results of those more extensive experiments.

The problem of term indexing is to maintain a large set of first-order predicate calculus terms and at the same time to provide fast access to members of that set. Four retrieval operations are of interest—to find unifiable terms, generalizations, instances, and alphabetic variants of a query term. In the context of clause-based automated deduction, one wishes to find unifiable terms when applying resolution or paramodulation inference rules or when searching for unit conflict, one wishes to find generalizations when determining whether a new clause is subsumed by an existing clause or when applying rewrite rules, and one wishes to find instances when determining whether a new clause subsumes any existing clauses or when searching for applications of a new rewrite rule. Retrieval of variants can be useful when applying restricted forms of the preceding deduction operations.

Term indexing was used in the early days of automated deduction [17, 10], but it was undocumented or not emphasized in the literature, and the methods are not well known. As a result, it appears that several of the methods were "reinvented". In particular, the origins of the two indexing methods compared in this article remain uncertain. The roots of discrimination-tree indexing appear to be directly in formula manipulation systems, and the roots of path indexing appear to be in database technology.

Discrimination-tree indexing (also called discrimination-net indexing) and some of its variations have appeared in [13, 3, 8, 14, 15, 20, 2, 5, 6]. It is used to find demodulators in [13], it is presented from a Lisp point of view in [3], it is used with very large sets of terms in [15], it is compared to path indexing in [20], it is used in the context of very high-performance Knuth-Bendix completion in [5, 6], and it forms the basis for a high-performance deduction toolkit in [2].

The predecessors of the path-indexing method [20] are coordinate indexing [10] and FPA indexing [17, 12, 11]. Path indexing is a simple but substantial refinement of FPA indexing and coordinate indexing. A hybrid method similar to FPA indexing is presented in [1].

A third class of indexing methods is based on encoding terms into bit strings and using bit operations to aid retrieval [9, 22, 18]. A disadvantage of these methods is that although the bit operations are very fast, in most cases a linear search is required. These methods are not discussed here.

The remainder of the article is divided into preliminaries, including a more precise statement of the indexing problem (Section 2), presentations of discrimination-tree indexing and path indexing (Sections 3 and 4), experiments with the two methods (Section 5), and conclusions.

This article is based on a presentation I gave at the American Association for Artificial Intelligence Spring Series Symposium on High-Performance Theorem Proving at Stanford

University in March 1989.

2 Preliminaries

A term is a variable, a constant, or a complex term. A complex term is a fixed-arity function symbol applied to a sequence of terms. The methods in this article apply as well to atoms (an atom is a relation symbol applied to a sequence of terms). Constants and function symbols are collectively called rigid symbols. Variables are distinguished from constants by starting with a lower-case letter from u to z.

Two terms are unifiable if they have a common instance, in particular, if there exists a substitution of terms for variables that makes the two terms identical. Term t_1 is an instance of term t_2 (and t_2 is a generalization of t_1) if there exists a substitution of terms for variables in t_2 that makes it identical to t_1. Two terms are alphabetic variants if each can be renamed to the other by substituting variables for variables. For all of the unification and matching problems, I assume that the two terms do not share any variables.

When two terms fail to unify or match, I sometimes refer to the reason for the failure. Clashes occur when two rigid symbols cannot be unified or matched. A direct clash can be detected without considering any partial substitution. Indirect clashes and occurs-check failures are detected by considering a partial substitution. Examples of the three types of failure are given in Table 1.

Table 1: Types of Unification Failure

Direct clash	Indirect clash	Occurs-check
$f(a,b)$	$f(x,x)$	$f(x,x)$
$f(a,c)$	$f(a,c)$	$f(y,g(y))$

The problem is to maintain a set of terms and, given a query term that does not share variables with any term in the set, support the following four retrieval operations:

1. Find all terms that unify with the query term, and construct a most general unifying substitution for each.
2. Find all generalizations of the query term, and construct a matching substitution for each.
3. Find all instances of the query term, and construct a matching substitution for each.
4. Find all variants of the query term, and construct a matching substitution for each.

1. $f(x,x)$
2. $f(x,y)$
3. $f(x,b)$
4. $f(g(a),x)$
5. $f(g(a),b)$
6. $f(a,y)$
7. $g(x)$
8. $g(z)$
9. $g(b)$

Figure 1: A Set of Terms and Its Discrimination Tree

3 Discrimination-Tree Indexing

A discrimination-tree index is a tree that represents the structure of all the terms in the index. The terms in the index are stored in the leaves of the tree. The retrieval operation traverses and backtracks through the query term and the discrimination tree, finding the appropriate leaves. Basic discrimination-tree indexing and several variations are presented in this section.

Basic discrimination-tree indexing serves as a pre-filter to unification or matching. The values of variables in the indexed terms and in the query term are ignored during the retrieval operation; in particular, variables are not bound during retrieval. There are no direct clashes between the retrieved terms and the query term, but there can be indirect clashes and occurs-check failures. The typical use of basic discrimination indexing is to retrieve terms from the index, then call a unify or match procedure to verify that each retrieved term unifies or matches with the query term and to bind variables and construct a substitution.

Let T be the set of terms indexed in a discrimination tree D, and let $T*$ be the set derived from T by replacing all variables with the special symbol $*$. Each node of D, except the root, is labeled with a function or constant symbol or the special symbol $*$. Each path from the root to a leaf corresponds to exactly one member of $T*$. In particular, for each path, the labels of the nodes is the preorder traversal of exactly one member of $T*$. In addition, the children of each node are unique. Each leaf of D contains the members of T that map to the member of $T*$ for that leaf. Figure 1 contains an example set of terms and its discrimination tree.

I present informal descriptions of backtracking algorithms for retrieving terms from a discrimination tree. Stickel's paper [20] contains formal recursive definitions of the various retrieval operations.

The operation of retrieving variants from a discrimination tree is the simplest. One starts at the root of the discrimination tree and walks through the query term in preorder. At each step of the walk, one branches to the child node in the discrimination tree that matches the current symbol in the query term, or fails if no matching child exists. A variable in the query term, regardless of its name, matches a $*$-node in the discrimination tree. If one reaches the end of a query term, one is always at a leaf of the tree, because symbol arities are fixed. No backtracking is required, because a variable cannot match with a nonvariable term; in particular, a symbol in the query term matches at most one child in the discrimination tree.

During retrieval of generalizations, subterms of the query term can match $*$-nodes in the discrimination tree as well as nonvariable nodes. Therefore, either a recursive set-oriented algorithm or a backtracking algorithm is required, because both branches must be explored. Figure 2 outlines a backtracking algorithm with the simplifying assumption that no variables exist in the query term.

If S is the current position in the query term, let next_symbol(S) be the next position in the preorder of the query term, and let skip_term(S) be the position following the subterm headed by S.

If N is the current node in the discrimination tree, let $*$child(N) be the $*$-node child of N (if any), and let matching_child(N,S) be the child of N (if any) that matches symbol S.

A stack of states <S,N> is maintained for backtracking.

```
S = start of query term;
N = root of discrimination tree;
while (S != NULL)
        if (a restore_state has not just occurred and *child(N) exists)
                save_state(<S,N>);
                N = *child(N);
                S = skip_term(S);
        else if (matching_child(N,S) exists)
                N = matching_child(N,S);
                S = next_symbol(S);
        else
                <S,N> = restore_state();
                if (<S,N> == NULL)
                        return(failure);
return(N);
```

Figure 2: Retrieving Generalizations from a Discrimination Tree

For retrieval of instances, we can make the simplifying assumption that all terms in the discrimination tree are ground—in particular, that no $*$-nodes occur in the discrimination tree. During retrieval of instances, a variable in the query term can match all children of a node in the discrimination tree. As in generalization retrieval, skipping must occur when a variable is matched; but in this case, the skipping is in the discrimination tree. When the current symbol in the query term is a variable, one must skip to all the descendant nodes in the discrimination tree that correspond to the ends of the terms that match the variable. Figure 3 outlines the procedure.

> Assume that S is the current symbol in the query term and that S is a variable. Let N be the current node in the discrimination tree.
>
> On the left-most branch of N not yet considered, skip to the descendant node that corresponds to the end of the term in the discrimination tree. Save S, N, and the current branch of N for backtracking.

Figure 3: Retrieving Instances from a Discrimination Tree

The standard way to skip to the end of the corresponding term and to save the position for backtracking is to access the arities of the symbols to identify the end of the term, then save the individual nodes in a stack for backtracking. See Variation 1, Section 3.1, for an optimization.

The operation to retrieve terms that unify with a query term has elements of both the generalization and the instance-retrieval operations. The procedure to find unifiable terms backtracks as in generalization retrieval when a $*$-node occurs in the discrimination tree, and it backtracks as in instance retrieval when a variable occurs in the query term.

3.1 Variation 1: Jump Lists

For instance and unifiable-term retrievals, the operation of skipping to the descendant nodes in the discrimination tree can be optimized by storing in each node an explicit list of pointers to the descendant nodes corresponding to the ends of the terms that start with the node. For example, the list for each child of the root points to all of the leaves for that subtree. These *jump lists* are updated whenever a term is inserted into or deleted from the discrimination tree. Retrieval time is saved, because the ends of the terms need not be computed and because backtracking is simplified. Jump lists need not include nodes for variables or constants, which would simply cause a jump to the current node of the discrimination tree.

The extra memory required for jump lists can be substantial. Let $g^3(a)$ abbreviate $g(g(g(a)))$. Consider the discrimination tree for the three terms $g^n(a)$, $g^n(b)$, and $g^n(c)$. The basic discrimination tree (excluding the root) has $n+3$ nodes, and the jump lists (excluding nodes for constants) total $3n$ nodes.

For Variations 2 and 3 which follow, it is useful to have an additional field in each member of a jump list. The additional field contains a pointer to the term being jumped.

3.2 Variation 2: Avoiding Rematching Structure

The typical use of discrimination indexing is to call a unify or match procedure with each retrieved term. This is somewhat wasteful, because the structure of the retrieved terms is already known to match in part the structure of the query term. The optimization is to record, while indexing, the variable/term pairs that must be bound. Then a special-purpose match or unify procedure can simply process the set of pairs. Our implementation of this variation requires jump lists (Variation 1) for instance and unifiable-term retrieval.

To apply the optimization to generalization retrievals, one can store, along with each indexed term, the list of variables in the term. During retrieval, a stack of terms is maintained. When a $*$-node is encountered, the corresponding subterm of the query term is pushed onto the stack; a term is popped from the stack when backtracking occurs. When a term is retrieved, its list of variables is matched (possibly failing) with the terms in the stack.

The optimization is more complex for instance retrievals, because one must collect a stack of subterms of the indexed terms to be matched with variables in the query term. Such terms do not exist in the proper form, because they are indexed with $*$-nodes instead of the correct variable. Our solution is to use a discrimination tree without $*$-nodes, in which variables have their true names, and to use jump lists with the additional term pointers. In such a case, a stack of variable/term pairs can be maintained while backtracking. When a term is retrieved, the pairs in the stack are matched (possibly failing).

As one might expect, the optimization for retrieving unifiable terms is a combination of the generalization and instance cases. With the optimization, the unification-retrieval operation returns two sets of pairs. One set contains pairs (query-variable,tree-subterm), and the other contains pairs (tree-variable,query-subterm). Each pair in the two sets must be unified (possibly involving secondary unifications and possibly failing). A further optimization is that some of the occurs-checks can be omitted. Arbitrarily choose one of the sets; the first occurrence of each variable in that set can be bound without an occurs-check.

3.3 Variation 3: Binding Variables During Indexing

The third variation is to bind variables and construct the substitution while traversing the discrimination tree. One advantage of this technique is that binding a variable in the discrimination tree can correspond to binding many variables in the indexed terms, thus saving binding operations (including occurs-checks). A second advantage is that conflicting bindings resulting from indirect clash and occurs-check failure can be discovered during indexing. A disadvantage is that this variation is not compatible with the use of $*$-nodes in the discrimination tree—terms must be indexed with respect to their correct variable names, a method that requires more memory for the discrimination tree. Variation 3 is incompatible with Variation 2.

Variation 3 applied to generalization retrieval is straightforward. A substitution environment (initially empty) is maintained during indexing. Assume that we are indexing, that the current subterm of the query term is T, and that the current node in the tree represents variable V. If V is not already bound, save the current state for backtracking, bind V to T, and continue. If V is bound to a term identical to T, save the current state for backtracking, and continue. Otherwise, backtrack.

Our implementation of Variation 3 for instance retrieval requires the use of jump lists with the additional term pointers. Assume that we are indexing, that the current subterm of the query term is a variable V, and that the current node in the tree is N with jump list J. If V is unbound, alternatively (by backtracking) process each member of J by binding V to the term and jumping to the descendant node. If V is bound to a term T, one must find a term, starting at N, that is identical to T. One method is simply to search the jump list for such a term; if one is found, jump over it and continue. A second method is to treat T as if it is the current query term and traverse the tree with T using a slightly different method that searches for exact matches rather than allowing the binding of variables; if T is successfully matched, the indexing reverts to its normal behavior.

Variation 3 applied to unification retrieval is a combination of its application to generalization and instance retrieval, with the following modifications. First, if one encounters a bound-variable node in the tree, the term to which it is bound is unified with the current subterm of the query term. Second, if the current query term is a bound variable, let the term to which it is bound become the query term. The following example shows an advantage of Variation 3 for unification retrieval. Consider query term $f(x,x)$. When the first occurrence of x is processed, it is bound to a term T. When the second occurrence of x is processed, T in effect replaces x for indexing purposes; discrimination occurs for T as if it is a subterm of the original query term.

3.4 Variation 4: Limiting Discrimination-Tree Depth

Variation 4 is to limit the depth of the discrimination tree to a fixed amount, say n. In the limited tree, each path from the root to a leaf is the first n symbols of one of the indexed terms. The leaves of the limited tree contain all terms with the corresponding initial substring. The effect of such a limit is to support indexing on at most the first n symbols of the indexed terms.

The motivation for this variation is simply to save memory. It is compatible with the use of jump lists, but it is not directly compatible with Variation 2 (avoiding rematching) or with Variation 3 (binding variables during indexing).

3.5 Implementations of Discrimination-Tree Indexing

We have several implementations of discrimination-tree indexing which are summarized in Table 2. All are written in C and use backtracking algorithms as outlined in the preceding paragraphs.

The program OTTER uses discrimination trees for generalization retrieval when finding left sides of rewrite rules during demodulation and when finding literals in potentially sub-

Table 2: Summary of Discrimination-Tree Implementations

Identifier	Type	Variations	Comments
DG-3	generalization	3	OTTER code
DI-1-3	instance	1,3	
DU-1-3	unify	1,3	
DG-F	generalization	none	flatterms
DG-2-F	generalization	2	flatterms
DG-3-F	generalization	3	flatterms
DI-1-2-F	instance	1,2	flatterms
DU-1-2-F	unify	1,2	flatterms

suming clauses (forward subsumption). Variation 3, binding variables during indexing, is used in the OTTER code.

Flatterms. Several of our other implementations use the *flatterm* representation introduced by Christian [5, 4] for the query term. In the flatterm form, a term is not stored as a tree; instead, it is stored as a doubly-linked list—one node per symbol—in which the nodes for the function symbols also have pointers to the ends of the corresponding terms (analogous to the jump pointers in a discrimination tree). See Figure 4.

$$g(f(f(a,g(b)),e))$$

Figure 4: A Term and Its Flatterm Representation

The use of flatterms as query terms simplifies and speeds the backtracking algorithms, because the maintenance of a position in a flatterm requires just one of its nodes, rather than the stack of nodes required for the tree form of a term. Our implementations use flatterms only for the query terms, not for the indexed terms or for constructing substitutions. (Christian uses flatterms exclusively, and he shows that their use speeds other term operations such as copying and symbol counting.)

Finding the Correct Child. One of the key operations in discrimination-tree indexing is to find the child node that matches the current (nonvariable) symbol in the query term. In all of our implementations, the children are stored as an ordered linked list, with $*$-nodes or variable nodes first. The lookup operation is a simple linear search of the list. The children of a node in Christian's discrimination trees are stored as an array, indexed by the symbol identifier, so that access is immediate. The use of arrays is certainly faster, but we have retained the use of linked lists. First, we must conserve memory, because we index very large sets of terms. Second, our applications usually have small numbers of distinct symbols, so that nodes have short lists of children. Finally, we must be able to efficiently handle applications with large numbers of distinct symbols.

4 Path Indexing

This section contains an informal presentation of Stickel's path-indexing method, a refinement of FPA indexing. See [20] for a more formal presentation. As in basic discrimination-tree indexing, the path-indexing method is a pre-filter to unification or matching: in addition to all appropriate terms, it may also retrieve terms that fail to unify or match because of indirect symbol clash or the occurs-check. In particular, path indexing retrieves exactly the same set of terms as basic discrimination tree indexing. As in basic discrimination-tree indexing, a unify or match procedure is called after indexing to verify unification or matching and to construct the substitution.

Every term has associated with it a set of *paths*—one path for each symbol (predicate, function, constant, or variable) in the term. In the standard tree form of a term, the paths are simply from the root to each node in the tree. All variables are replaced with the special symbol $*$. A path is written as an alternating sequence of symbols from the term and integers, starting and ending with symbols from the term. The integers give the positions of the children of each node. Figure 5 shows the paths for term $h(a, g(g(x)), f(b, y))$.

$$\begin{array}{l}
[h] \\
[h, 1, a] \\
[h, 2, g] \\
[h, 2, g, 1, g] \\
[h, 2, g, 1, g, 1, *] \\
[h, 3, f] \\
[h, 3, f, 1, b] \\
[h, 3, f, 2, *]
\end{array}$$

Figure 5: A Term and Its Paths

A *path list* is a list of pointers to terms that have a common path. To index a term so that it can later be found by one of the retrieval functions, one computes its paths and inserts a pointer to the term into the corresponding path lists. For example, to index the term $f(a, x)$, a pointer to the term is inserted into the path lists $[f]$, $[f, 1, a]$ and $[f, 2, *]$.

Path-indexing retrievals are performed by computing unions and intersections of path lists. Consider the query term $f(a, g(b), x)$. Every instance of the query term has paths $[f, 1, a]$ and $[f, 2, g, 1, b]$, so the retrieval operation to find instances of the query term is simply the intersection of those two path lists:

$$[f, 1, a] \cap [f, 2, g, 1, b].$$

The operation to find variants of $f(a, g(b), x)$ is similar to the instance retrieval, except that the third argument of f must be a variable:

$$[f, 1, a] \cap [f, 2, g, 1, b] \cap [f, 3, *].$$

To find terms that unify with $f(a, g(b), x)$, one must allow variables to occupy the positions corresponding to f, a, g, and b. The retrieval operation must include unions with paths ending with variables:

$$[*] \cup \left(\begin{array}{c} [f, 1, a] \cup [f, 1, *] \\ \cap \\ [f, 2, g, 1, b] \cup [f, 2, g, 1, *] \cup [f, 2, *] \end{array} \right).$$

Finally, the operation to find generalizations of $f(a, g(b), x)$ is similar to the unify retrieval, except that if the retrieved term is not a variable, it must have a variable as the third argument of f:

$$[*] \cup \left(\begin{array}{c} [f, 1, a] \cup [f, 1, *] \\ \cap \\ [f, 2, g, 1, b] \cup [f, 2, g, 1, *] \cup [f, 2, *] \\ \cap \\ [f, 3, *] \end{array} \right).$$

All appropriate terms are returned by the retrieval operations, but terms that fail to unify or match can be returned as well. The typical use of path indexing is to retrieve the set of terms, then call a match or unify routine with each member of the set to verify that it does unify or match and to construct the substitution.

4.1 A Variation on Path Indexing

Path indexing normally considers all paths in a term, which gives indexing at all depths of the query term. A variation is to limit the lengths of paths, which imposes a limit on the indexing depth. If the indexing depth is n, then the longest path that will be considered has length $2n + 1$. For example, to find terms that unify with $f(a, g(b), x)$ using indexing depth 0, one retrieves $[*] \cup [f]$; with indexing depth 1, one retrieves

$$[*] \cup \left(\begin{array}{c} [f, 1, a] \cup [f, 1, *] \\ \cap \\ [f, 2, g] \end{array} \right).$$

If the indexing depth is limited, more terms that fail to unify or match are retrieved, but the size of the index is smaller.

4.2 Implementation of Path Indexing

The path-indexing code in OTTER (written in C) was used for the experiments. The process of inserting a term into an index is straightforward. For each path (up to the indexing depth if applicable) in the term, one inserts a pointer to the term into the path list for the path (creating a new path list if necessary). The path lists are accessed by hashing, in which the path is the key. Each term has a unique integer identifier ID, and the path lists are kept ordered on the ID of the term so that the unions and intersections can be computed efficiently.

Given a query term, the retrieval operation consists of two steps. The first step is to construct a binary tree that specifies the unions and intersections to be performed. Each internal node is either a $\cup$-node or a $\cap$-node, and each leaf corresponds to a path and contains a pointer to the head of the path list. The second step, which is outlined in Figure 6, is a recursive procedure (due to Ross Overbeek) that performs the unions and intersections.

Note that each path list in a leaf is traversed at most once. The following optimization is applied in our implementation. When case intersection_node has a term from one child and it calls up a term from the other child, it can pass as an argument the term it already has as the minimum acceptable value. The procedure retrieve_next_term also receives min_term. The case leaf_node returns the next term $\geq$ min_term; The case union_node simply passes min_term through to subcalls.

Stickel's method [20] for computing the unions and intersections, which also traverses each path list at most once, does not require the lists to be ordered and does not use an explicit binary tree, but it requires an extra field in each member of the path lists. For example, to intersect three lists, each member of the first list is marked with a 1, then each member of the second list with mark 1 is marked with a 2, and finally the members of the third list that are marked with a 2 are retrieved. Although our method is probably somewhat slower because of all of the recursive calls, we retain it, because our shared-memory parallel implementations [19] would require a separate tag field for each processor.

5 Experiments

By analysis of the indexing algorithms, one can draw some general conclusions about their relative performances.

1. As the number of distinct symbols in the indexed terms *increases*, path indexing can be expected to improve, because although there are more path lists, the sizes of the path lists are smaller, and the number of unions and intersections is the same.

```
procedure retrieve_next_term(node) returns a term;

/* The boundary conditions have been omitted; in particular, */
/* the action to be taken when a recursive call returns NULL, */
/* and startup of the union_nodes. */

case leaf_node:
        /* The position in the path list is saved in the node. */
        return the next term in the path list;

case intersection_node:
        t1 = retrieve_next_term(node→left_child);
        t2 = retrieve_next_term(node→right_child);
        while (t1 ≠ t2)
                if (t1 < t2)
                        t1 = retrieve_next_term(node→left_child);
                else
                        t2 = retrieve_next_term(node→right_child);
        return t1;

case union_node:
        /* The current left_term and right_term were stored in */
        /* the node during the previous call with this node. */
        if (node→left_term ≤ node→right_term)
                node→left_term = retrieve_next_term(node→left_child);
        if (node→left_term ≥ node→right_term)
                node→right_term = retrieve_next_term(node→right_child);
        return minimum(node→left_term, node→right_term);
```

Figure 6: Computing Unions and Intersections in a Path Index

2. As the number of distinct symbols in the indexed terms *decreases*, discrimination indexing can be expected to improve, because the discrimination tree is smaller (more sharing of the initial substrings); in particular, there are fewer nodes to traverse.

3. Path indexing for instance retrieval can be expected to perform well, because the operations on the path lists are particularly simple; no unions are required.

4. Discrimination indexing for generalization retrieval can be expected to perform well, because there are at most two alternatives at each node of the tree: the $*$-child (if any) or the correct nonvariable child (if it exists).

5. Path indexing can be expected to perform well if there are many variables in the query term, because those variables represent "don't care" conditions and do not contribute union operations.

6. Discrimination indexing for instance and unification retrieval may be expected to perform poorly when the query term contains many variables, because a variable in the

query term matches all children of the current node in the discrimination tree.

7. Memory requirements for path indexing are more predictable than for discrimination indexing. The sum of the lengths of the path lists is the number of symbol occurrences in the indexed terms. The size of a discrimination tree depends on the sharing of the initial substrings of the indexed terms; it can be very sensitive to order of the arguments and subterms of the indexed terms.

However, worst-case analyses of the indexing algorithms are not particularly useful, and average-case analyses are difficult because of the wide variation in the type and structure of formulas that appear in real applications.

5.1 Term Sets

The term sets for experimentation were taken from typical OTTER applications. Experiments with the first two sets, Luka-5 and Robbins, are intended to simulate generalization retrieval during term rewriting (demodulation). The remaining six sets are paired: a set of positive literals and a set of negative literals in each pair. Generalization retrieval and instance retrieval in which the indexed set is the same as the query set simulate literal retrieval during forward and backward subsumption. Unification retrieval in which the indexed set is opposite in sign from the query set simulates unit conflict tests and, to a lesser extent, resolution inference rules. Unification-retrieval experiments were also performed in which the indexed set is the same as the query set.

Variables start with u–z.

Luka-5. The set Luka-5 is derived from the equality formulation of the fifth Lukasiewicz conjecture [7], a theorem in multivalued sentential calculus. A proof was found with OTTER by using Knuth-Bendix completion techniques. Luka-5 is the set of left sides of the first 2000 rewrite rules (demodulators) that were derived during the search. Most of those 2000 rewrite rules were still present (had not been simplified) at the time the proof was found. The indexed set is Luka-5, and the query set is the multiset of the 13,862 nonvariable subterms of the terms in Luka-5. A representative member of Luka-5 is

$$i(n(i(x,y)),i(n(i(i(z,y),n(z))),u)).$$

Robbins. The Robbins set is derived from a theorem on the relationship between Robbins and Boolean algebras: that the property $\exists x \exists y\ (x+y=y)$ makes a Robbins algebra Boolean [21]. The set Robbins is the left sides of the first 2000 rewrite rules derived in a Knuth-Bendix search with OTTER. (The operator + is associative and commutative, which we handle with axioms and rewrite rules rather than with AC-unification. No computer has yet proved the theorem.) The indexed set is Robbins, and the query set is the multiset of the

45,534 nonvariable subterms of the terms in Robbins. A representative member of Robbins is

$$n(+(n(+(y,D)),n(+(D,+(n(n(+(D,+(n(+(x,y)),n(+(y,n(x))))))),n(+(y,D))))))).$$

CL-pos, CL-neg. The sets CL-pos and CL-neg are derived from a theorem in combinatory logic (CL), that the fragment $\{B,N\}$, with $Bxyz = x(yz)$ and $Nxyz = xzyz$ contains fixed point combinators. The members of the sets are the first 1000 positive and negative literals, respectively, from a bidirectional paramodulation search (unsuccessful) with OTTER. Representative members of CL-pos and CL-neg are

$$\begin{aligned}&(a(x,a(a(a(a(a(a(B,y),z),u),a(a(a(B,y),z),u)),v),a(a(a(B,y),z),u))) =\\ &\quad a(a(a(a(a(B,B),a(B,a(a(a(B,a(B,x)),a(N,N)),v))),y),z),u))\end{aligned}$$

$$\begin{aligned}&(a(f(a(a(N,a(N,B)),a(B,a(a(N,x),y)))),a(a(a(a(N,B),f(a(a(N,a(N,B)),a(B,a(a(N,\\ &\quad x),y))))),a(B,a(a(N,x),y))),f(a(a(N,a(N,B)),a(B,a(a(N,x),y)))))) \neq\\ &\quad a(a(a(x,a(af(a(a(N,a(N,B)),a(B,a(a(N,x),y)))),a(B,a(a(N,x),y))),f(a(a(N,\\ &\quad a(N,B)),a(B,a(a(N,x),y)))))),y),a(a(f(a(a(N,a(N,B)),a(B,a(a(N,x),y)))),\\ &\quad a(B,a(a(N,x),y))),f(a(a(N,a(N,B)),a(B,a(a(N,x),y))))))).\end{aligned}$$

EC-pos, EC-neg. The sets EC-pos and EC-neg are derived from a theorem in equivalential calculus (EC), that the formula XGK=$e(x,e(e(y,e(z,x)),e(z,y)))$ implies the formula PYO=$e(e(e(x,e(y,z)),z),e(y,x))$ by condensed detachment. The members of the sets are the first 500 positive and negative literals, respectively, from a UR-resolution (unit-resulting resolution) search with OTTER. The literals all have the property that each variable has exactly two occurrences; the positive literals have the additional property that they are composed entirely of variables and the function symbol e. Representative members of EC-pos and EC-neg are

$$P(e(e(e(e(x,e(y,y)),e(x,z)),z),e(u,e(u,e(v,v)))))$$

$$\begin{aligned}&\neg P(e(e(x,e(e(y,e(z,x)),e(z,y))),e(e(u,e(e(v,e(w,u)),e(w,v))),e(e(e(e(e(e(v6,e(e(v7,\\ &\quad e(v8,v6)),e(v8,v7))),e(e(e(a,e(b,c)),c),e(b,a))),e(e(v9,e(e(v10,e(v11,v9)),\\ &\quad e(v11,v10))),v12)),v12),e(e(v13,e(e(v14,e(v15,v13)),e(v15,v14))),v16)),v16)))).\end{aligned}$$

Bool-pos, Bool-neg. The sets Bool-pos and Bool-neg are derived from a theorem in the relational formulation of Boolean algebra. The theorem is that associativity of + is a consequence of an axiomatization that does not include that fact. The members of the sets are the first 6000 positive and negative literals, respectively, from a UR-resolution search with OTTER. These sets differ from the preceding sets in that they contain more distinct symbols and the literals are less deeply nested. Representative members of Bool-pos and Bool-neg are

$$SUM(p(s(c3,a),x),p(s(c3,a),n(x)),c4)$$

$$\neg SUM(s(p(c2,n(x)),p(c2,x)),s(p(c2,n(x)),p(c2,x)),c4).$$

5.2 Results of Experiments

The experimental results for term retrieval are summarized in Tables 3–5. All of the experiments were run on a Sun Microsystems SPARCstation 1+ computer (about 12 million instructions/second). All of the times (given in seconds) include unification/matching as well as retrieval from the index, because Variation 3 unifies/matches during indexing. The times also include (where applicable) transforming the query terms into the flatterm representation. The times do not include construction of the indices. The best time(s) for each comparison is (are) enclosed in a box. There are no results for retrieval of alphabetic variants, because none of our current applications uses that operation.

Table 3: Generalization Retrieval

Indexed set	Query set	Path	Path-6	Path-3	DG-3	DG-F	DG-2-F	DG-3-F	Notes
Luka-5	13862	63.7	64.5	69.6	1.4	2.8	1.5	$\boxed{1.2}$	A
Robbins	45534	27.0	35.4	93.9	$\boxed{1.9}$	4.7	3.7	3.8	A,B
CL-pos	CL-pos	16.1	16.3	39.0	1.2	1.2	$\boxed{0.8}$	$\boxed{0.8}$	
CL-neg	CL-neg	22.3	17.6	26.4	$\boxed{1.2}$	2.0	$\boxed{1.2}$	$\boxed{1.2}$	C
EC-pos	EC-pos	5.0	5.0	6.5	$\boxed{0.2}$	0.6	$\boxed{0.2}$	$\boxed{0.2}$	
EC-neg	EC-neg	10.1	26.0	32.7	0.5	0.6	$\boxed{0.4}$	$\boxed{0.4}$	
Bool-pos	Bool-pos	28.7	28.7	28.7	1.2	1.7	1.2	$\boxed{0.9}$	
Bool-neg	Bool-neg	14.5	14.5	14.2	$\boxed{0.6}$	0.9	0.7	0.7	

Table 4: Instance Retrieval

Indexed set	Query set	Path	Path-6	Path-3	DI-1-3	DI-1-2-F	Notes
CL-pos	CL-pos	2.3	2.8	26.4	3.9	$\boxed{2.2}$	D
CL-neg	CL-neg	5.7	4.9	19.9	$\boxed{1.2}$	1.6	
EC-pos	EC-pos	$\boxed{1.0}$	1.1	4.3	1.2	1.2	E
EC-neg	EC-neg	$\boxed{1.4}$	16.6	30.4	6.1	3.1	E,D
Bool-pos	Bool-pos	7.0	7.1	7.1	3.8	$\boxed{3.2}$	
Bool-neg	Bool-neg	8.8	8.8	9.2	$\boxed{0.7}$	0.8	

The designation "Path-n" indicates path indexing to depth n. The designations "DG-", "DI-", and "DU-" are from Table 2 and indicate which variations were in use and whether query terms were flattened. Recall that Variation 1 is the use of jump lists, Variation 2 is to avoid rematching some of the structure of the terms, and Variation 3 is the binding of variables during retrieval.

Notes on the experiments appear below. Each refers to one or more lines in Tables 3–5.

Table 5: Unifiable Term Retrieval

Indexed set	Query set	Path	Path-6	Path-3	DU-1-3	DU-1-2-F	Notes
CL-pos	CL-pos	30.2	40.2	127.8	27.2	$\boxed{12.5}$	F
CL-pos	CL-neg	42.3	42.5	94.2	39.5	$\boxed{19.8}$	F
CL-neg	CL-neg	26.3	24.2	41.7	6.2	$\boxed{6.0}$	
CL-neg	CL-pos	49.1	49.0	104.3	99.3	$\boxed{26.3}$	F,G
EC-pos	EC-pos	45.8	45.6	42.7	$\boxed{18.4}$	26.5	H
EC-pos	EC-neg	61.0	60.5	52.1	$\boxed{16.8}$	35.8	H
EC-neg	EC-neg	223.3	220.5	220.1	$\boxed{134.2}$	164.1	H
EC-neg	EC-pos	57.1	56.8	55.5	$\boxed{27.0}$	35.0	H,G
Bool-pos	Bool-pos	33.1	33.1	33.1	8.8	$\boxed{8.1}$	
Bool-pos	Bool-neg	26.9	26.9	26.7	$\boxed{2.7}$	$\boxed{2.7}$	
Bool-neg	Bool-neg	14.6	14.5	14.3	$\boxed{0.9}$	$\boxed{0.9}$	
Bool-neg	Bool-pos	6.0	6.0	5.9	12.1	$\boxed{5.4}$	F,G

Key to Notes in Tables 3–5

A. In the Luka-5 and Robbins experiments, the query set is the multiset of nonvariable subterms of the indexed set. The size of the multiset is given in the column "Query set".

B. There is a substantial bonus for Variation 3 for generalization retrieval with the Luka-5 terms.

C. In generalization retrieval on CL-neg with path indexing, limiting the indexing depth to 6 is substantially better than no limit. Indexing beyond depth 6 is apparently wasteful in this case. Note that the CL-neg terms are very deep.

D. The depth of path indexing makes a large difference in the CL-pos and EC-neg instance-retrieval comparisons. In general, the penalty for limiting the depth of path indexing seems to be greater for instance retrieval than for the other operations.

E. Path indexing wins in the two EC instance-retrieval cases, and only in those cases.

F. There is a penalty for Variation 3 (binding while indexing) in the CL unifiable-term retrievals. Also see note H.

G. Path indexing performs relatively well in these three unification comparisons. The reason may be that the query terms have many variables and the indexed terms have fewer variables, making the behavior similar to instance retrieval on which path indexing performs well.

H. There is a bonus for Variation 3 (binding while indexing) in the EC unifiable-term retrievals. When compared with note F, the reason for the behavior of Variation 3 is not clear, given the similarity in structure between the CL terms and the EC terms.

5.3 Memory Requirements

Memory requirements for the indices were calculated for the sets of terms used in the retrieval experiments. Tables 6 and 7 give data in nodes rather than in bytes so that they can apply to implementations other than our own.

Table 6: Discrimination-Tree Memory (in nodes)

Set	Terms	$*$-tree	var-tree	jump(*complex*+*const*+*var*)	Notes
Luka-5	2000	5462	8113	(9858+85+5780)	a
Robbins	2000	17054	19083	(28274+5622+2835)	
CL-pos	1000	17669	18884	(13154+3464+8553)	
CL-neg	1000	65767	65767	(37771+17964+15511)	
EC-pos	500	1722	3263	(3563+0+2273)	a,b
EC-neg	500	16721	16721	(12596+3000+7349)	
Bool-pos	6000	19441	19915	(16538+13536+2713)	
Bool-neg	6000	28437	28437	(24183+17813+1360)	

Table 7: Path Index Memory (in nodes)

		Path	Path-6	Path-3	
Set	Terms	(*nodes on lists*, *lists*, *path total*)			Notes
Luka-5	2000	(25763,911,10971)	(25217,653,7009)	(17011,91,541)	c
Robbins	2000	(51145,4242,104518)	(25899,177,1829)	(11492,36,188)	
CL-pos	1000	(36844,1049,18507)	(28410,207,2325)	(12234,20,116)	
CL-neg	1000	(89181,5416,110630)	(47775,568,6590)	(13087,33,203)	
EC-pos	500	(9256,291,4131)	(8340,120,1330)	(3738,14,80)	d
EC-neg	500	(34212,6111,184007)	(12980,123,1383)	(3750,11,61)	d
Bool-pos	6000	(55968,633,3661)	(55968,633,3661)	(55968,633,3661)	e
Bool-neg	6000	(79668,3437,31205)	(79668,3437,31205)	(66371,895,5727)	

For discrimination-tree indexing, the column $*$-tree indicates the number of nodes in the basic discrimination tree, and column var-tree is the size of the tree in which variable nodes are labeled with the variable name rather than $*$. The triple (*complex*+*const*+*var*) gives the the number jump-list members for complex terms, constants, and variables, respectively, for the var-tree. They are given separately because some implementations require members for constants and variables, while others do not. Our implementation consumes 12 bytes for each node of a discrimination tree, and 12 bytes for each member of a jump list. In addition, each leaf has a list of pointers to terms for that leaf: each member of that list uses 8 bytes.

The memory required for a Path index is given as a triple (*nodes on lists*, *lists*, *path total*). The third, *path total*, is the sum of the lengths of the paths. (I assume that a copy of the path is stored at the head of each path list.) Our implementation uses 8 bytes for each member of a path list, 16 bytes for the head of each path list, 1 byte for each symbol in a path, and 2000 bytes for the hash table.

Notes on memory requirements appear below. Each refers to one or more lines in Tables 6 and 7.

Key to Notes in Tables 6 and 7

a. For the Luka-5 and EC-pos discrimination indexes, the var-trees are much larger than the $*$-trees. For each of the other sets, there is little or no difference.

b. There are no jump-list members for constants, because EC-pos has no constants.

c. In the Luka-5 path indices, Path has 911 path lists, and Path-6 has 91, but the time for generalization retrieval (Table 3) is similar.

d. In the EC path indices, there is great variation in the number of path lists, but the time for unification retrieval (Table 5) is similar.

e. All terms in the set Bool-pos have depth 3 or less, so the counts for Path, Path-6, and Path-3 are identical.

Table 8 shows the approximate memory usage in kilobytes for our implementations. The column var-tree-jump is the total memory for the var-tree with jump nodes for complex terms only.

Table 8: Memory Usage for Our Implementations (Kbytes)

Set	Path	Path-6	Path 3	$*$-tree	var-tree	var-tree-jump
Luka-5	234	221	140	82	113	232
Robbins	584	214	95	221	245	584
CL-pos	332	235	100	220	235	392
CL-neg	913	400	107	797	797	1250
EC-pos	85	72	32	25	43	86
EC-neg	557	109	32	205	205	356
Bool-pos	464	464	464	281	287	485
Bool-neg	726	726	553	389	389	679

6 Conclusion

Our codes for the two indexing methods are written in straightforward, portable C. They have not been highly tuned or optimized. Some important optimizations might have been missed in the code or algorithms, so the timing results in Tables 3–5 should be used simply as guidelines.

The strongest conclusion from the experiments (Table 3) is that discrimination indexing is a clear winner over path indexing for generalization retrieval on the types of term with which I experimented. This result fits very well with the needs of our theorem prover OTTER, because, in many applications, the most time-consuming operation is generalization retrieval for forward subsumption and for demodulation (term rewriting). For retrieval of instances, Table 4 does not show a clear winner with respect to time. We use path indexing with an optional depth limit for instance retrieval in OTTER, because it usually requires less memory. Table 5 indicates that, for unifiable-term retrieval, discrimination indexing is somewhat faster; however, OTTER retains path indexing (with an optional depth limit) for unifiable-term retrieval, because the operation is usually not a bottleneck, and it usually requires less memory.

I have not addressed the important issue of indexing with respect to special unification algorithms—in particular, commutative, permutative, and associative-commutative unification. Others have addressed this issue in part. It appears that path indexing can be extended to effectively handle commutative functions by simply deleting the appropriate argument positions from the paths [20]. Christian's discrimination indexing [5] handles permutative terms by making multiple indexing calls with permuted variations of the query term.

Ross Overbeek and Ralph Butler have designed and constructed Formula Data Base (FDB) [2], a package of C subroutines for building high-performance automated deduction systems. At the heart of FDB is basic discrimination-tree indexing for retrieval of unifiable terms, instances, and generalizations. Christian's flatterm representation [5] is used throughout FDB. When FDB finds a leaf of a discrimination tree (a set of candidates for unification or matching) it uses novel techniques based on structure sharing and permutations for constructing the unifying substitutions. The performance of FDB's discrimination indexing is usually slightly better than mine, but factors of up to 2 in both directions have been observed. For information on the status and availability of FDB, contact Ross Overbeek, MCS-221, Argonne National Laboratory, Argonne, IL, 60439-4844, e-mail overbeek@mcs.anl.gov.

OTTER [16] is a resolution/paramodulation deduction system for first-order logic with equality. OTTER is available for free by anonymous FTP and is also available through several other sources; for information write to me or send e-mail to mccune@mcs.anl.gov. If there is any demand for the term sets used for the experiments, I shall make those available by anonymous FTP as well.

Acknowledgments

Ross Overbeek has been truly creative in his many approaches to term indexing; I have had valuable discussions on the topic with him. Work on applications of automated deduction with Larry Wos and other members of the Argonne group led to the term sets I used for the experiments. Mark Stickel read an early draft of this article and provided many useful comments.

References

[1] R. Butler, E. Lusk, W. McCune, and R. Overbeek. Paths to high-performance automated theorem proving. In J. Siekmann, editor, *Proceedings of the 8th Conference on Automated Deduction, Lecture Notes in Computer Science, Vol. 230*, pages 588–597, New York, 1986. Springer-Verlag. An early version appears as Argonne National Laboratory Tech. Memo ANL/MCS-TM-71.

[2] R. Butler and R. Overbeek. A tutorial on the construction of high-performance resolution/paramodulation systems. Tech. report ANL-90/30, Argonne National Laboratory, Argonne, IL, 1990.

[3] E. Charniak, C. Reisbeck, and D. McDermott. *Artificial Intelligence Programming.* Lawrence Earlbaum Assoc., Hillside, NJ, 1980.

[4] J. Christian. Fast Knuth-Bendix completion: A summary. In N. Dershowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science, Vol. 355*, pages 551–555, New York, 1989. Springer-Verlag.

[5] J. Christian. *High-performance permutative completion.* PhD thesis, The University of Texas at Austin, 1989.

[6] J. Christian. Flatterms, discrimination nets, and fast term rewriting. Submitted, 1990.

[7] J. M. Font, A. J. Rodriguez, and A. Torrens. Wajsberg algebras. *Stochastica*, 8(1):5–31, 1984.

[8] S. Greenbaum. *Input transformations and resolution implementation techniques for theorem proving in first-order logic.* PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[9] L. Henschen and S. Naqvi. An improved filter for literal indexing in resolution systems. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 528–530, 1981.

[10] C. Hewitt. *Description and theoretical analysis (using schemata) of Planner: A language for proving theorems and manipulating models in a robot.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, January 1971.

[11] E. Lusk, W. McCune, and R. Overbeek. Logic Machine Architecture: Kernel functions. In D. Loveland, editor, *Proceedings of the 6th Conference on Automated Deduction, Lecture Notes in Computer Science, Vol. 138*, pages 70–84, New York, 1982. Springer-Verlag.

[12] E. Lusk and R. Overbeek. Data structures and control architecture for the implementation of theorem-proving programs. In R. Kowalski and W. Bibel, editors, *Proceedings of the 5th Conference on Automated Deduction, Lecture Notes in Computer Science, Vol. 87*, pages 232–249, New York, 1980. Springer-Verlag.

[13] J. McCharen, R. Overbeek, and L. Wos. Complexity and related enhancements for automated theorem-proving programs. *Computers and Mathematics with Applications*, 2:1–16, 1976.

[14] W. McCune. An indexing method for finding more general formulas. *Association for Automated Reasoning Newsletter*, 1(9):7–8, January 1988.

[15] W. McCune. Discrimination tree indexing for large sets of formulas: Experiments and the structure of terms. Notes on a talk given at the AAAI symposium on high-performance theorem proving, Stanford, CA, March 1989.

[16] W. McCune. OTTER 2.0 users guide. Tech. Report ANL-90/9, Argonne National Laboratory, Argonne, IL, March 1990.

[17] R. Overbeek. *A new class of automated theorem-proving algorithms.* PhD thesis, Pennsylvania State University, 1971.

[18] K. Ramamohanarao and J. Shepard. A superimposed codeword indexing scheme for very large Prolog databases. In *Third International Conference on Logic Programming, Lecture Notes in Computer Science, Vol. 225*, pages 569–576, New York, 1986. Springer-Verlag.

[19] J. Slaney and E. Lusk. Parallelizing the closure computation in automated deduction. In M. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Vol. 449*, pages 28–39, New York, July 1990. Springer-Verlag. Extended abstract.

[20] M. Stickel. The path-indexing method for indexing terms. Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.

[21] S. Winker. Absorption and idempotency criteria for a problem in near-boolean algebras. Preprint MCS-P177-0990, Argonne National Laboratory, Argonne, IL, August 1990.

[22] M. Wise and D. Powers. Indexing Prolog clauses via superimposed codewords and field encoded words. In *IEEE Conference on Logic Programming*, pages 203–210, Atlantic City, 1984.