Experiments with Associative-Commutative Discrimination Nets

Leo Bachmair
Ta Chen
I.V. Ramakrishnan
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794 (U.S.A)

Siva Anantharaman
Jacques Chabin
Departement d'Informatique
LIFO, Universite d'Orleans
45067-Orleans Cedex 02 (France)

Abstract

We recently proposed a data structure, called associative-commutative discrimination nets, that supports efficient algorithms for (many-to-one) term matching in the presence of associative-commutative functions. In this paper we discuss the integration of such discrimination nets into an actual equational theorem prover and report on corresponding experiments. The general associative-commutative matching problem is known to be NP-complete, but can be solved in polynomial time if the given terms are linear, i.e., do not contain multiple occurrences of the same variable. We therefore have implemented a two-stage matching procedure. First we check whether a match exists for the linearized versions of the given terms (where different occurrences of the same variable are replaced by different new variables). If a match for the linearized terms does exist, we then determine whether there is also a match for the original, non-linear terms (i.e., whether the proposed substitutions for different occurrences of the same variable are consistent). Our experimental results indicate that this approach works very well in theorem proving, where most matching attempts actually fail and are filtered out during the first stage, so that the second, more expensive stage of the algorithm is only needed in comparatively few cases.

1 Introduction

Matching and unification are two fundamental operations in theorem proving. Unification is part of deductive inference rules such as resolution, whereas matching is needed for subsumption, normalization by rewriting, and other mechanisms for simplifying formulas and eliminating redundancies. Experimental evidence indicates that the efficiency of resolution-type provers mainly depends on these simplification mechanisms. For instance, in rewrite-based equational theorem provers, most of the time (about 80 to 90%) is spent on term rewriting and normalization—operations that both require matching—and relatively little on deducing new formulas via unification. In resolution-type theorem provers the deletion of subsumed clauses is critical for performance. In short, efficient implementations of matching are indispensable for such provers, cf. [McCune, 1992J. We should also point out that pattern matching algorithms are a key component in many other applications, including functional and logic programming [Ramesh *et* a/., 1990] and rule-based expert systems [Forgy, 1982].

It is becoming increasingly clear that many applications of theorem proving require efficient methods for reasoning about associative-commutative functions. In this paper we discuss our work on extending the equational theorem prover REVEAL by efficient algorithms for matching in the presence of associative-commutative function symbols.

2 Preliminaries

We consider terms built from function symbols and variables. The letters s and *t* are used to denote terms, / and *g* to denote function symbols, and x, y, and *z* to denote variables. The expression $t|_p$ denotes the subterm off at position *p.* Positions may, for instance, be represented in Dewey decimal notation. The top-most position is denoted by Λ, and hence $t|_\Lambda = t$. The sequences 2 and 2.2 denote positions in $t = f(a, g(a, b))$,, with $t|_2 = g(a, b)$ and $t|_{2.2} = b$.

Let *AC* be a set of associativity and commutativity axioms

$$f(x, f(y, z)) = f(f(x, y), z)$$

$$f(x, y) = f(y, x)$$

for some function symbols /. We also write $f \in AC$ if / is such an associative-commutative symbol and write $s =_{AC} t$ to indicate that s and *t* are equivalent under associativity and commutativity. A term *t* is said to *AC-match* another term *s* (and *s* is called an *AC-instance* of t) if there exists a substitution , such that $t\sigma =_{AC} s$.

It is convenient to represent terms equivalent under *AC* by flattened terms. Let *L* be the set of all rewrite rules (called *flattening rules)*

$$f(X, f(Y), Z) \quad \rightarrow \quad f(X, Y, Z),$$

where $f \in AC$, and X, Y, and Z are sequences of terms with $|X| + |Z| \geq 1, |Y| \geq 2$. (By $|X|$ we denote the length

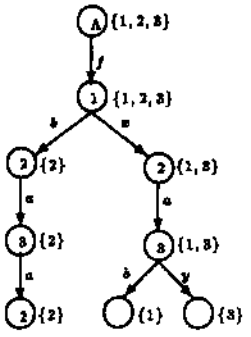

Figure 1: A standard discrimination net for $\{t_1 = f(x, a, b),\ t_2 = f(b, a, a),\ t_3 = f(x, a, y)\}$

of a sequence X.) Terms that can not be rewritten by L are said to be *flattened*. The flattened version of a term t is denoted by $\bar{t}$. By $\sim$ we denote the smallest symmetric rewrite relation (also called the *permutation congruence*) for which

$$f(X, u, Y, v, Z) \sim f(X, v, Y, u, Z)$$

if $f \in AC$. It is readily seen that s is an AC-instance of t if and only if $\bar{t}\sigma \sim \bar{s}$, for some substitution σ. (In the case of flattened terms, we have to allow $x\sigma$ to be a sequence of terms if x occurs as an argument of an AC-symbol in t.)

3 Discrimination Nets

Various data structures have been proposed for the elementary operations needed in theorem proving, e.g., [Ohlbach, 1990; Graf, 1994]. We illustrate the basic ideas by way of an example for one such data structure—discrimination nets—and its use for matching, or more specifically *many-to-one matching*, where one wishes to identify, given terms $t_1, \ldots, t_n$ (called the *patterns*) and s (called the *subject*), which of the terms t_i match s. In rewrite-based theorem proving, the terms t_i represent the left-hand sides of the current rewrite rules, whereas the subject term s is the term to be rewritten. This matching problem needs to be solved for several different subject terms, but with a fixed set of patterns. For instance, many terms may need to be rewritten using the same set of rewrite rules. Therefore, the pattern terms are represented by a special data structure similar to a trie and called a *discrimination net*, that allows one to quickly identify matching patterns for any given subject term. A key idea is that common parts of terms t_i be shared in the representation of the full pattern set.

For example, suppose we have three pattern terms, $f(x, a, b)$, $f(b, a, a)$, and $f(x, a, y)$. A discrimination net T for the set M of these three patterns is shown in Figure 1. The discrimination net is a tree, with edges labelled by function symbols and variables (occurring in terms in M) and nodes labelled by positions (of subterms of terms in M) and subsets of M. The labelling needs to satisfy the condition that for each node u, the associated set M_u consists of exactly those terms in M with occurrences of function symbols as prescribed by the path from the root of the net to u. (The sets M_u are called *match sets* for obvious reasons.) Interior nodes indicate *partial* matches, and leaves, complete matches.

For example, the left-most leaf in the net in Figure 1 has match set $\{2\}$, as $t_2 = f(b, a, a)$ is the only term in M which has an f at the root (position A), a b at position 1, and an a at positions 2 and 3.

To determine which patterns match a given terms s one simply traverses the discrimination net with the input term s as a guide. More specifically, an edge is traversed if its label is either identical to the corresponding symbol in s or else is a variable. For example, the term $s = f(a, a, a)$ causes traversal of the rightmost path in T, which yields match set $\{3\}$ and indicates that s is an instance of t_3, but not of t_1 or t_2. The traversal induced by the term $f(b, b, b)$ gets stuck at the second level, indicating that there is no match.

3.1 Associative-Commutative Discrimination Nets

Let us next consider how discrimination nets may be applied to terms with associative-commutative function symbols. By the *AC-nesting depth* of a position in a term we mean the number of AC-symbols that occur on the path from the position to the root. More formally, the AC-nesting depth at A is 0; and if n is the AC-nesting depth at position p in t, then the AC-nesting depth at a position $p.i$ in t is $n+1$, if t contains an AC-symbol at position p, and n, otherwise. The AC-nesting depth of a term is the maximum AC-nesting depth of any of its positions. The *top-layer* $\hat{t}$ of a term t is the expression obtained from t by removing all subterms at positions with non-zero AC-nesting depth. For example, if $f \in AC$ and $g \notin AC$, then the top-layer of $g(x, f(b, c))$ is $g(x, f)$. Obviously, if s is an AC-instance of t, then its top-layer must be an (ordinary) instance of the top-layer of t. For example, the term $g(a, g(a, a))$ cannot be an AC-instance of $g(x, f(b, c)$ as the respective top-layers do not match. If the top-layers of the given terms match, we need to consider the stripped-off subterms and recursively check for the existence of suitable AC-matches.

Thus, if s and t are flattened terms with $t\sigma \sim s$, then (i) $\hat{t}$ matches $\hat{s}$ and (ii) if p is a position of an AC-symbol in the top-layer of t, then $(t|_p)\sigma \sim s|_p$. Conversely, if (i) and (ii) are satisfied for some substitution σ, then $t\sigma \sim s$. In other words, AC-matching is completely characterized by conditions (i) and (ii). Condition (i) represents a standard matching problem, while condition (ii) leads to a bipartite graph matching problem. Suppose $t|_p = f(t_1, \ldots, t_m)$ and $s|_p = f(s_1, \ldots, s_n)$, where $f \in AC$. We may also assume, without loss of generality, that the terms $t_1, \ldots, t_k$ are non-variables, while all terms $t_{k+1}, \ldots, t_m$ are variables, for some k with $0 \leq k \leq m$. Define a bipartite graph $G = (V_1 \cup V_2, E)$, with $V_1 = \{s_1, \ldots, s_n\}$, $V_2 = \{t_1, \ldots, t_k\}$, and E consisting of all pairs (s_i, t_j), such that $t_j\sigma \sim s_i$, for some substitution σ. It can easily be seen that if (a) either $n = m$ or $n > m > k$, and (b) there is a maximum bipartite matching of size k in the bipartite graph G,

then $f(t_1, \ldots, t_m)\sigma \sim f(s_1, \ldots, s_n)$, for some substitution σ and hence condition (ii) is satisfied. In part (a), if $m > k$, then some of the terms t_i contain variables and, since we consider linear pattern terms only, a suitable matching substitution can be found, whenever all non-variable terms $t_1, \ldots, t_k$ are matched (and $n \geq m$, of course). For example, if $f \in AC$, then $s = f(h(a), a, b)$ is an AC-instance of $t = f(h(x), y)$: substitute a for x and the sequence a, b (or $f(a, b)$) for y.

These observations lead to the definition of an *AC-discrimination net* for a set of terms M as a hierarchically structured collection of standard discrimination nets: the top of the hierarchy is a standard discrimination net for the set of top-layers of terms in M; to each node with an incoming edge labelled by an AC-symbol (a so-called *AC-nodes*) another AC-discrimination net is attached that represents the subterms of terms in L_u, where L_u is the set of terms corresponding to the AC-node.

For example, an AC-discrimination net for the two terms $h(f(a, b), c, f(y, c))$ and $h(f(a, x), c, f(a, b))$ where f is the only AC-symbol, is shown in Figure 2. Both

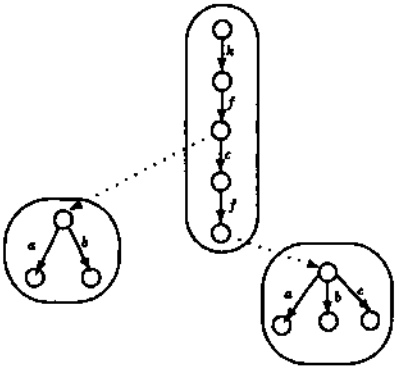

Figure 2: An AC-Discrimination Net

terms have the same top-layer, $h(f, c, f)$. The set of terms $\{f(a, b), f(a, x)\}$ is associated with the first AC-node; the set $\{f(y, c), f(a, b)\}$ with the second AC-node. The corresponding subnets represent $\{a, b\}$ and $\{a, b, c\}$, respectively.

To determine which terms AC-match a given term s, we traverse the individual standard nets in the hierarchy as usual, and use a bipartite graph matching algorithm to combine the results from different levels of the hierarchy. For example, suppose $h(f(b, a), c, f(a, b))$ is an input term for the above AC-net. The top-level net is successfully traversed to the first AC-node, call it v, at which point the two subterms b and a are provided as inputs to the first subnet (which represents the terms a and b). Traversal of the subnet yields two bipartite graphs, one for each element of the set $L_v = \{f(a, b), f(a, x)\}$. The size of each respective maximum matching indicates that the input term $f(b, a)$ is an AC-instance of both $f(a, b)$ and $f(a, x)$. Traversal of the top-level net resumes at node v and continues on to the second AC-node, call it v', with $L_{v'} = \{f(y, c), f(a, b)\}$. The input terms for

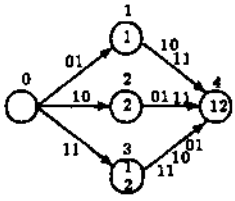

Figure 3: Secondary automaton of rank 2.

the second subnet are a and b. The subterm $f(a, b)$ of the input is found to be an AC-instance of $f(a, b)$, but not $f(y, c)$. Thus, only the pattern $h(f(a, x), c, f(a, b)$ is identified as an AC-match. A full description of the algorithm can be found in [Bachmair *et al.*, 1993].

3.2 Secondary Automata

The AC-matching algorithm has to construct bipartite graphs of the form $G = (V_1 \cup V_2, E)$, where $V_1 = \{s_1, \ldots, s_n\}$ is a set of input subterms, $V_2 = \{t_1, \ldots, t_k\}$ is a set of (non-variable) pattern subterms, and E contains an edge (s_i, t_j) if and only if t_j AC-matches s_i, so as to extract subsequently a maximum matching of size k. The set V_2 depends only on the patterns; it remains fixed for a given AC-net and node therein and, at least for applications in theorem proving, is usually small with at most five elements. The set of vertices V_1 and the set of edges E depend on the input. The graph G is computed stepwise, via a sequence of bipartite graphs $G_1, \ldots, G_n$. Traversal of the subnet for input s_1 determines the edges incident on that node, thereby defining G_1. Traversal of the subnet for s_2 determines the edges incident on s_2, which are added to G_1 to yield G_2; and so on.

For example, consider $V_2 = \{t_1, t_2\}$. Each traversal of the subnet for a term s_i yields a bitstring b_i of length two: the bitstring 10 is obtained if s_i is an AC-instance of t_1 but not of t_2; and the strings 00, 01 and 11 are interpreted correspondingly. The size of a maximum matching on the bipartite graph G_i can be readily determined from b_i and the size (and kind) of a maximum matching on G_{i-1}. The computation can be conveniently represented in a finite automaton, which we call a *secondary automaton*. More specifically, we speak of a *secondary automaton of rank k* (denoted by S_k) if V_2 contains k elements.

The secondary automaton S_2 is shown in Figure 3.2. The initial state of the automaton represents graphs with a maximum matching of size 0; its three successor states represent three different types of graphs with maximum matchings of size 1;[1] and the final state represents graphs with a maximum matching of size 2. In our algorithm we use secondary automata up to rank 5, which is sufficient for theorem proving applications.

[1] The three cases are: (i) t_1 can be matched, but not t_2; (ii) t_2 can be matched, but not t_1; (iii) both t_1 and t_2 can be matched, but not at the same time.

4 Implementation of AC-Matching

The -AC-matching problem is an NP-complete problem, but can be solved in polynomial time if pattern terms are linear [Benanav *et* a/., 1987]. We have therefore designed AC-matching as a two-phase process. In the first phase, an AC-discrimination net for the linearized versions of patterns is used as a filter: if no AC-match exists for the linearized patterns, then there certainly exists none for the original patterns. If in the first phase some terms are identified as AC-matches, we need to check in a second phase whether any of these matching terms can be extended to AC-matches in the presence of non-linearity. A straightforward AC-matching algorithm is used in this second phase. Our experimental results provide strong evidence that in typical theorem proving problems most of the patterns can be eliminated during the first phase, so that the expensive second phase is invoked only sparingly and the two-phase approach results in substantial performance gains.

4.1 Term representation

We adopt a variant of the data structure for flat terms used in HIPER, see [Christian, 1989] for details. Flattened versions of terms equivalent under *AC* are unique only up to permutation congruence. For instance, the two terms /(a, *b, c)* and /(c, a, *b)* are AC-equivalent if $f \in AC$.. We define a *unique* representative for each *AC*-equivalence dass by using a total ordering on function symbols and variables and lexicographically extending it to a total ordering on terms. The minimal term in an equivalence dass serves as the unique representative. For example, if $a < b < c$, then $f(a, b, c) < f(c, a, b)$.

4.2 Standard Discrimination Nets

Let us next discuss the design and implementation of discrimination nets. We first consider standard discrimination nets, which provide the basis for AC-discrimination nets. In theorem proving, the set of (pattern) terms that need to be represented in a discrimination net change dynamically. We have chosen a non-deterministic variant of such nets in which insertion and deletion of patterns can be performed easily. (The main difference between deterministic and non-deterministic nets is that the latter require backtracking in order to identify all matches. Deterministic nets require no backtracking, but may use more space.)

Pattern terms are stored at the root of the net. Each leaf maintains a list of pointers to the patterns in the matchset associated with the leaf node. With each node we maintain a linked-list representing the (labelled) edges from that node. This list is kept sorted (according to the edge labels), so that the traversal of the net can be implemented by a linear search through this list.

Example 1 *Figure 4 shows the discrimination net for the set {f(x,* a, 6), /(b, a, a), /(x, a, y)}, *where the underlying ordering on symbols is x<y<a<b<f.*

This variant of discrimination nets essentially corresponds to the third variant described in [McCune, 1992]. In contrast to HIPER and also our earlier description of discrimination nets in [Bachmair *et al.,* 1993], we distinguish between different (non-AC) variables. Although

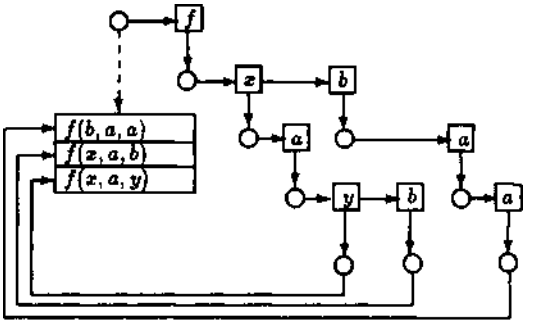

Figure 4: Standard discrimination net

such a distinction may potentially result in more backtracking and consume more memory, it has the advantage that checking for the consistency of substitutions for (non-AC) variables can be done during traversal of the net, so that failures caused by inconsistencies can be discovered earlier. To allow for consistency checking we maintain a global substitution table and, whenever descending to a branch corresponding to a (non-AC) variable x, compare the entry for *x* (if present) in the substitution table with the current subterm of the input term.

4.3 AC-Discrimination Nets

Recall that *AC-nets* require additional operations such as bipartite matching using secondary automata, intersection of matchsets produced by AC-subnets and reporting the result from one level of the net to the next higher level. We first describe data structures that are needed for these operations. With each *AC-node v* we associate a set L_v of AC-subterms. These subterms are stored in a linked list; each element in the list is called a *control block* and corresponds to one AC-subterm. In the control block for a term $t \in L_v$ we maintain information relevant to bipartite matching and secondary automata transitions. When a pattern term is deleted, all control blocks corresponding to its AC-subterms must also be deleted. To facilitate deletion, each pattern is assigned an unique ID number and all control blocks derived from the pattern inherit this ID, which is then used as a search key during deletion. The search is implemented as a simple linear search in the linked list associated with L_v.

For each pattern, there is a pointer from the control block of each AC-subterm *t* at AC-nesting depth / to the control block of an AC-superterm[2] of *t* at AC-nesting depth / — 1. Similarly, if a term *t* is in the matchset of a leaf, there is a pointer from the leaf to the control block of the closest AC-subterm enclosing this *t.*

Example 2 *Suppose k*(a,c, f(y,c)) *and k(a,c, f*(a,b)), $f \in AC$, *are two patterns. We have* L_2 — {/(y, c),/(a,b)} *represented by the list of control blocks C and D. The enclosing AC-subterm of* /(y, c) *and*

[2] An AC-superterm of *t* is a super term of *t* with *AC*-function symbol at its root.

$f(a,b)$ are the patterns themselves. So there is a pointer from control block C to A and D to B.

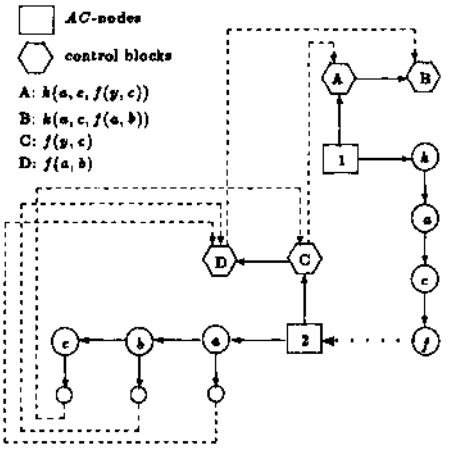

Figure 5: AC-discrimination net

4.4 Secondary Automata

A secondary automaton is constructed for each desired rank and stored during initialization of the system. Each state of a secondary automaton is implemented as an array of pointers. Recall that transition symbols are bitstrings. We use the integers representing the bitstrings as indices with the arrays. For example, Figure 6 is the secondary automaton of rank 2 (also see Figure 3 for its abstract representation) where each state is represented as a four-element array. On bitstring 01, whose integer value is 1, the automaton changes from state 0 to state 1.

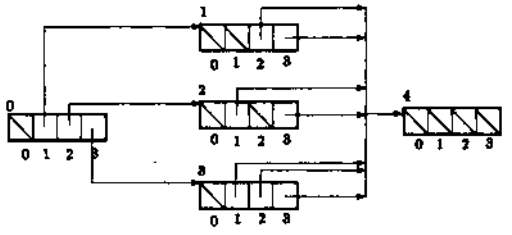

Figure 6: Secondary automaton of rank 2

Bitstrings are obtained as follows. Note that for a pattern $f(t_1,\ldots,t_k,x_{k+1},\ldots,x_m)$, $f \in AC$, and a subject $f(s_1,\ldots,s_n)$, each s_i, $1 \le i \le n$, can AC-match more than one t_j, $1 \le j \le k$. Since nondeterministic nets are used, t_j may appear in the matchsets of different leaf nodes, all of which will be visited when s_i is being matched. We assign each t_j a bitstring of length k such that the bits for the bitstring for t_j are all zeros except the j^{th} bit. The input bitstring to the secondary automaton for s_i is then obtained by performing an inclusive-OR operation between the bitstrings of t_j, where t_j is in the matchset of a leaf node visited. For instance, for the pattern $f(h(a,x),h(a,c),e)$, $f \in AC$, the arguments $h(a,x)$, $h(a,c)$ and e are in matchsets of different leaf nodes, say node 1, 2 and 3, and the bitstrings associated with them are 100, 010 and 001, respectively. On the subject $f(h(a,c),e,e)$, since $h(a,c)$ AC-matches with both $h(a,x)$ and $h(a,c)$, leaf nodes 1 and 2 will be visited and so the bitstring obtained will be $100 \vee 010 = 110$, indicating that $h(a,c)$ matches the first and second arguments of the pattern.

Optimization to reduce rank: The number of nonvariable arguments of an AC-subterm in a pattern determines the rank of the associated secondary automaton. The size $|S_k|$ of a secondary automaton grows rapidly with increasing k. For instance, $|S_1| = 2$, $|S_2| = 5$, $|S_3| = 16$, $|S_4| = 68$ and $|S_5| = 406$. We only construct secondary automata up to rank 5, but this has not been a limitation in practice. In particular, we use optimization techniques that have enabled us to run all the benchmarks using automata of rank ≤ 5. First observe that multiple occurrences of the same term can be merged for the purpose of constructing secondary automata. We only need to keep track of the multiplicity of each distinct argument. For instance, to deal with the subterms in $f(a,a,b,c,c,c)$ we only need an automaton of rank 3, not rank 6. Also, we may partition terms according to their top function symbol. For example, suppose $t_1 = h(t_1')$, $t_2 = k(t_2')$, $t_3 = k(t_3')$, $t_4 = h(t_4')$, $t_5 = h(t_5')$, and $t_6 = h(t_6')$. Evidently, an instance of one of the first three terms can not be an instance of one of the last three terms, and vice versa. Consequently, instead of using an automaton of rank 6 for all six terms, we may use two automata of rank 3 for each of the two subgroups. If necessary, we can partition further by looking at more function symbols other than the top symbol.

4.5 Putting it all together

We are now ready to explain the operation of an indiscrimination net based upon the above representation. At each *AC-node,* we do the following steps. We check whether (i) the number of arguments in the pattern exceeds those in the subject; or (ii) if the pattern has no variable arguments and has fewer non-variable argument than the subject. In such cases, the pattern is marked unmatchable and no further action is done on behalf of it. For patterns which pass the above test, secondary automata are used to determine the existence of a maximum bipartite matching for the corresponding bipartite graph. Secondary automata are used for small bipartite graphs (deriving from pattern terms with few nonvariable AC-arguments). If the number of non-variable arguments is large, we use a general bipartite matching algorithm [Papadimitriou and Steiglitz, 1982]. In the experiments reported below, we never had to resort to a general bipartite matching algorithm.

On successful completion of the above steps, we include the AC-subterm in the matchset for the *AC-node*. This matchset is intersected with those computed at other AC-nodes at the same level of the net. We use Stickel's algorithm [McCune, 1992] to do list intersection. Specifically, suppose L_1, L_2 and L_3 are three lists.

We first mark the elements of L1 with 1. Then those elements of L2 that are marked with 1 are marked with 2. Finally, elements of L3 that are marked with 2 are the results of L1 n L2 n L3. Since an intersection has to be done at each AC-node, we can use the number of AC-nodes visited for each level as the current value of the mark. Finally, when a leaf node r is reached, the matchset Mr associated with r is also intersected with the outcome of earlier intersections at this level.

Example 3 With the AC-discrimination net in Figure 5, suppose the subject is k(f(a, c),c,/(a, 6)) and we have just entered node 2. Upon processing /(a, c) through the subnet at node 2, only B is marked with 1, since f(a,x) of the second pattern is matched, but not f(a,b) of the first pattern. Then, although both terms /(a, b) and /(a, b) in L3 are in the matchset computed at node 3, only B gets marked with 2 because it is marked currently with 1. Therefore, the second pattern is selected by the net.

5 Experimental Results

We next present experimental results we obtained with our implementation on a Sparc 10 with 32 MB of memory. We first compare AC-discrimination nets with straightforward AC-matching, and then show the effect of the resulting savings in the context of a theorem prover. Finally, we show that the use of AC-discrimination nets does no compromise on the performance of a theorem prover on problems without associativity and commutativity.

5.1 AC-Matching

The first data set provides an indication of the effectiveness of AC-discrimination nets. We took a set of 1,000 terms from a typical theorem proving application in OTTER [McCune, 1992], and built an AC-net for it. Then each one of these terms is used as a subject (resulting in a successful AC-match, of course). The resulting times are compared with a straightforward AC-matching algorithm that uses no discrimination net. We ran the algorithms in two different modes: (i) to identify the first matching term (and then stop), or (ii) to find all matching terms. We also repeated these tests for a smaller subset of 100 terms.3 The results are summarized in Table 1. It can be seen that AC-discrimination nets result in a three- to four-fold speedup for finding the first matching term (which is the variant needed for rewriting). All times are in seconds.

5.2 AC-Completion

We then ran a number of experiments with our equational theorem prover REVEAL. All problems contained associative-commutative function symbols. They include the following:

1. grobner: The Grobner base of a certain ideal in a polynomial ring in two variables over integers is computed via completion;

3 We should note that the terms in the original OTTER-examples do not contain AC-symbols; but we declared one of the (binary) symbols to be AC.

Problems	Net	No Net	With Net	Speedup
CL_pos_100				
First	0.02	8.38	1.05	7.98
All		8.83	1.08	8.18
CL_pos_1000				
First	0.15	43.74	11.48	3.81
All		87.62	11.44	7.65
CL_neg_100				
First	0.03	0.48	0.15	3.20
All		0.93	0.12	7.75
CL_neg_1000				
First	0.47	45.45	12.74	3.57
All		89.63	12.20	7.35

Table 1: Performance of *AC*-Discrimination Nets

2. grpfini30: A canonical rewrite system is computed for an abelian group of order 30, specified with three generators;
3. jacobson: Commutativity is proved for rings satisfying X* = X ;
4. moufang: The sesquilinearity of the associator is proved for (non-associative) alternative rings;
5. robbinh : It is proved that any Robbins algebra is a Huntington algebra (and hence a Boolean algebra) if there exists an element c, such that c + c = c ;
6. robbinh2 : Similar to robbinh, but under the assumption that there exist elements c, d such that c -f d — c ;
7. uqsl2 : Certain properties of some 'unitary quantum groups' are proved

Table 2 summarizes timings for these and several other AC-problems. We compare the performance of the prover with AC-discrimination nets (under columns 'ACN' in the table) with a more naive strategy 'NaT without AC-nets. One of the key components of our prover is the normalization of terms by rewriting, which requires (AC-)matching. Given a set T of rewrite rules and a term s to be normalized, the prover takes one rule from T and tries to match it at the root of s. In case of failure, the next rule in T is tried, and so on, until either a match is found or else all rules in T are exhausted. In the latter case, the process is restarted at the next position of s. We measured (a) the time to normalize terms with respect to the current rewrite system and (b) the total time leading to proofs. The speedups are given for both the normalization and for total computing times. The speedups for normalization, naturally, are significantly greater than those for total times.

Since the prover needs to find only one matching pattern, from Table 1 we may expect an average speedup of about 4 for finding the first matching pattern. Since in normalization about half the time is spent in finding a match, we can expect to improve normalization by a factor of 2 on the average. This is consistent with the speedup figures for normalization in Table 2. Finally, about 70% of the total computing time is spent in normalization and the rest on other operations, such as unification (for deduction of so-called critical pairs), etc. So we can expect a speedup of about 35% on the

	Time				Speedup	
	Normalization		Total			
Prob.	Nai	ACN	Nai	ACN		
balukas	1.82	1.40	3.15	2.75	1.30	1.15
boolston	1.35	0.79	2.89	2.36	1.71	1.22
grobner	18.18	6.66	30.05	18.95	2.73	1.59
grpfini30	2.69	1.88	7.77	7.10	1.43	1.09
hunt2acm	91.63	95.72	118.01	121.58	0.96	0.97
jacobson	137.34	61.06	169.94	89.75	2.25	1.89
midlaw	0.29	0.20	0.99	0.90	1.45	1.10
moufang	45.74	12.76	58.97	26.12	3.58	2.26
newpb	5.26	4.00	8.19	7.16	1.32	1.14
newpb2	7.07	5.67	10.99	9.63	1.25	1.14
robbinh	3.11	1.18	6.47	4.50	2.64	1.44
uqs12	10.15	1.38	15.35	6.74	7.36	2.27

Table 2: Performance of *AC*-Discrimination Nets

average for completion. This is again consistent with the speedups in Table 2 we obtained for our examples.

Finally, we like to mention that memory consumption of the AC-discrimination nets for the problems in Table 2 never exceeded the memory required for storing the patterns, i.e., the extra memory needed at most doubled.

5.3 Non-AC Problems

As mentioned earlier, standard discrimination nets have been used quite successfully to solve many-to-one non-AC-matching problems. It is critical that the performance of our AC-matching algorithm do not degrade when used for non-AC problems. Table 3 shows that this has been accomplished in our implementation. All the problems in there are non-AC The second column are the timings obtained by standard discrimination net while the third column pertains to *AC*-discrimination nets. Observe that the overhead of using AC-discrimination net for non-AC problems is negligible.

Problems	SN	ACN	Ratio
assoc	0.54	0.55	0.98
groupl	0.74	0.76	0.97
groupr	0.59	0.60	0.98
furtin	0.83	0.85	0.98
hunting2c	14.84	15.05	0.99
ternary	4.09	4.36	0.94

Table 3: Timings for some Non-AC Problems Using Standard Net and AC-Net

6 Summary

In this paper, we have presented the design, implementation and experimental results of an AC-discrimination net based AC-matching algorithm. This algorithm has been integrated into the equational theorem prover REVEAL, which we used for our experiments. Our implementation exploits the fact that although AC-matching is iVP-complete, it can be solved in polynomial time if patterns are restricted to linear terms. It solves the many-to-one AC-matching in two phases. In the first phase only those patterns are selected that AC-match under the assumption that all *(AC)* variables are different (i.e., linear). The consistency of the substitutions computed for *(AC)* variables are checked in the second phase. The experimental results show that a high percentage of patterns are filtered out in the first phase, leaving very few patterns for the more complex consistency checking in the second phase. For instance, for our benchmark problem uqsl2, no more than one pattern was selected by the AC-discrimination in 98% of the cases (no pattern was selected in 88% and one pattern in 10% of the cases). We also ran experiments with different search strategies and obtained similar results. In particular, we want to mention that on robbhin2 we were able bring down the total time from 213.5 minutes to 175.5 minutes using AC-discrimination nets. All these* results and the speedups we obtained provide strong evidence that AC-discrimination nets and secondary automata are indeed useful tools for significantly improving the performance of theorem provers for *AC* problems.

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments. This research was supported in part by the NSF under grants CDA-9303181, INT-9314412 and CCR-9404921.

References

[Bachmair *et al,* 1993] L. Bachmair, T. Chen, and Iv Ramakrishnan. Associative-commutative discrimination nets. In *Proceedings of the 4 th International Joint Conference on Theory and Practice of Software Development, CAPP/FASE,* pages 61-74, Orsay, France, April 1993. Springer-Verlag LNCS 668.

[Benanav *et al.,* 1987] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation,* 3:203-216, 1987.

[Christian, 1989] J. Christian. *High-Performance Permutative Completion.* PhD thesis, The University of Texas at Austin, August 1989.

[Forgy, 1982] C. Forgy. Rete, a fast algorithm for the many patterns many objects Match problem. *Artificial Intelligence,* 19:17-37, 1982.

[Graf, 1994] P. Graf. Extended path-indexing. In *12th Conference on Automated Deduction,* pages 514-528. Springer-Verlag LNCS 814, 1994.

[McCune, 1992] W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning,* 9:147-167, 1992.

[Ohlbach, 1990] H.J. Ohlbach. Abstraction tree indexing for terms. In *Proceedings of the 9th European Conference on Artificial Intelligence,* pages 479-484, London, August 1990. Pitman Publishing.

[Papadimitriou and Steiglitz, 1982] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982.

[Ramesh *et al,* 1990] R. Ramesh, I.V. Ramakrishnan, and D.S. Warren. Automata-driven indexing of prolog clauses. In *Seventh Annual A CM Symposium on Principles of Programming Languages,* pages 281-290, San Francisco, 1990.