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.

Abstract. Use of discrimination nets for many-to-one pattern matching has been shown to dramatically improve the performance of the Knuth-Bendix completion procedure used in rewriting. Many important applications of rewriting require associative-commutative (AC) function symbols and it is therefore quite natural to expect performance gains by using similar techniques for AC-completion. In this paper we propose such a technique, called AC-discrimination net, that is a natural generalization of the standard discrimination net in the sense that if no AC-symbols are present in the pattern, it specializes to the standard discrimination net. Moreover we show how AC-discrimination nets can be augmented so as to further improve the performance of AC-matching on problems that are typically seen in practice.

1 Introduction

Term matching is a fundamental operation in equational and functional programming and in various theorem proving methods, such as the many variants of the completion procedure [3]. In these applications the problem usually occurs in the form of many-to-one matching, where one has to determine for a given set of terms $t_1, \ldots, t_n$, also called *patterns*, if one matches a specified term s, called the *subject*. For instance, in a completion procedure the patterns to be considered are the left-hand sides of current rewrite rules, which may dynamically change.

The most efficient many-to-one matching algorithms use trie-like data structures, called *discrimination nets*, and corresponding tree automata, that allow one to factor out common expressions in a given set of patterns. Since completion procedures spend most of their time on normalization of terms (which consists of repeated steps of matching followed by term replacement), sophisticated matching algorithms based on discrimination nets may result in dramatic speedups, as has been demonstrated by Christian [1].

An important application of completion is to associative-commutative rewrite systems, which require a suitably modified operation, called associative-commutative matching (or AC-matching). One-to-one AC-matching is an NP-complete problem, but can be solved in polynomial time if patterns are restricted to linear terms (without multiple occurrences of variables) [2]. The essential component of the one-to-one AC-matching algorithm described by Benanav, Kapur, and Narendran [2] is the application of maximum bipartite graph matching. In fact, Verma and Ramakrishnan [13] showed that the two problems, associative-commutative matching and maximum bipartite graph matching are mutually

reducible, so that complexity bounds for one problem also apply to the other. The best currently known lower bound for AC-matching is $O(mn^{1.5})$, where m refers to the size of the pattern and n to the size of the subject.

In many applications, however, many-to-one matching is needed. Since AC-matching takes up most of the computation time in associative-commutative completion procedures—and discrimination nets had resulted in considerable speedups of standard completion—there have been attempts to adapt discrimination nets to AC-matching. These attempts have not been entirely satisfactory. In this paper, we demonstrate that discrimination nets, in combination with bipartite graph matching, can indeed be applied to do AC-matching efficiently. We propose associative-commutative discrimination nets, which are hierarchically structured collections of standard discrimination nets, with bipartite graph matching being used to combine the results from one level of the hierarchy so as to make them available to the next higher level. This approach applies to all different kinds of discrimination nets that have been proposed in the literature (deterministic or nondeterministic, adaptive or non-adaptive), and an associative-commutative net specializes to a standard discrimination net if applied to terms with no associative-commutative operators. The algorithm solves the many-to-one matching problem, but is of the same asymptotic complexity as the best current algorithm for the simpler problem of one-to-one matching.

The paper is organized as follows. In the next section we introduce some terminology and give a general definition of discrimination nets. In Section 3 we briefly discuss the reduction of associative-commutative matching to a matching problem on flattened terms. In Section 4 we introduce associative-commutative discrimination nets, design a corresponding matching algorithm, and analyse its complexity. In practical applications of associative-commutative matching, the bipartite graph matching problems that need to be solved are of a special form, which we exploit in Section 5 to improve the algorithm. Nonlinear matching is briefly discussed in Section 6. In the concluding section we summarize our results and compare them to other recent work.

2 Discrimination Nets

We shall consider terms built from a given finite set of function symbols $\mathcal{F}$ and a (countable) set of variables $\mathcal{V}$. We use the symbols s and t to denote terms, f and g to denote function symbols, and x, y, and z to denote variables. The *arity* of a function symbol f is denoted by $\alpha(f)$. If t is a term $f(t_1, \ldots, t_n)$, then $t_1, \ldots, t_n$ are called the *top-level arguments* of t.

The expression $t|_p$ denotes the subterm of t at position p. Positions may, for instance, be represented as sequences of integers. So, if $p' = pq'$ (that is, p is a *prefix* of p'), then $t|_{p'}$ is a subterm of $t|_p$. We use Λ for the top-most position ($t|_\Lambda = t$). By $t(p)$ we denote the symbol at position p in t. The symbol $t(\Lambda)$ is also called the top-most, or root, symbol of t. For example, 2 is a position in $t = f(a, g(a, b))$ and $t|_2 = g(a, b)$, while $t(2) = g$.

The application of a substitution σ to a term t is written $t\sigma$. A term t is said to *match* another term s (and s is called an *instance* of t) if there is a substitution σ such that $s = t\sigma$.

In term rewriting, efficient matching algorithms have been designed that employ an indexing technique similar to tries based on so-called *discrimination nets*. In defining discrimination nets, we have to refer to "partially constructed" terms. By a *skeleton* we mean a set S of pairs (p, f_p) of positions and function symbols, where (i) S contains a pair with first component p whenever it contains one with first component $p.i$, and (ii) if $p.i$ is a first component of some pair in S, then $i \leq \alpha(f_p)$. Given a (non-empty) skeleton S, we define its *fringe* to be the set of all positions $p.i$, such that S contains a pair (p, f_p), where $1 \leq i \leq \alpha(f_p)$, but contains no pair with first component $p.i$. The fringe of the empty skeleton is defined to be $\{\Lambda\}$. Skeletons and corresponding fringes can conveniently be represented as terms built from function symbols and some special constant $\Box$ not contained in $\mathcal{F}$: the fringe consists of all positions at which $\Box$ occurs, while the remaining positions, with their corresponding function symbols, determine the skeleton. For example, the term $f(\Box, g(a, \Box))$ represents the skeleton $\{(\Lambda, f), (2, g), (2.1, a)\}$ with corresponding fringe $\{1, 2.2\}$.

A term t is said to be *compatible* with a skeleton S if for every pair (p, f_p) in S either p is a position in t and $t(p) = f_p$, or else some prefix of p is a variable position in t. A set M of terms is said to be compatible with S if each term in M is. If M is compatible with S, we say that p is a *discrimination position* (for M and S) if it is a position in the fringe of S and $t(p)$ is a function symbol (not a variable), for some term t in M.

Let ω be a new constant not contained in $\mathcal{F}$. A *matching tree* is a tree where the edges are labelled by symbols from $\mathcal{F} \cup \{\omega\}$ and with each node u is associated a non-empty set of terms M_u (the match set), a skeleton S_u (the partial match), and, if u is not a leaf, a position p_u (the discrimination position), such that (i) the empty skeleton is associated with the root of the tree; (ii) p_u is a discrimination position for M_u and S_u, for each non-leaf node u; and (iii) for each edge (u, v), $S_v = S_u \cup \{(p_u, f)\}$, where f is the label of (u, v), and M_v is the set of all terms in M_u that are compatible with S_v. Note that, for each node u, the match set M_u is compatible with the skeleton S_u.

A *discrimination net* is a maximal matching tree that contains no duplicate nodes. (A node v' is a duplicate of another node v, if there are two edges (u, v) and (u, v') that are labelled by the same symbol.) In other words, a discrimination net is a matching tree to which no edges can be added. Thus, in a discrimination net there is no discrimination position for M_u and S_u, for any leaf u. We also say that D is a discrimination net for the set of terms M associated with its root. Observe that, if D is a discrimination net for M, all discrimination positions p_u must be positions in some term t of M. Furthermore the discrimination positions along any branch in D are distinct, so that any discrimination net for a finite set of terms M has to be finite.

The construction of a discrimination net for a given (finite) set of terms T is straightforward. An initial matching tree consists of a single node u labelled

by $M_u = M$ and $S_u = \emptyset$. If for some leaf v in a given matching tree there exists a discrimination position p_v for M_v and S_v, then the tree can be expanded correspondingly. If there are no further discrimination positions, the process terminates with a discrimination net. Since for any given leaf, there may be different discrimination positions to choose from, different discrimination nets may be obtained from a given initial set of terms M. In this sense our definition characterizes *adaptive* discrimination nets. Discrimination nets have typically been constructed according to some fixed order in which positions are chosen, e.g., left-to-right preorder as in [4]. The construction of discrimination nets can be optimized via a suitable choice of discrimination positions, in order to improve the matching time and decrease the size of the discrimination net, see [11]. A discrimination net for the set of terms $\{f(a,a), f(a,x)\}$ is shown in Figure 1(a).

Let T be a set of *linear* terms (that is, terms without multiple occurrences of the same variable) and D be a discrimination net for T. We define a corresponding *matching automaton* A as follows. The nodes of D are the states of A, with the root being the initial state and the leaves being final states. On a given input term s, the automaton makes a transition $u \rightarrow v$, if D contains an edge (u,v) labelled by the symbol f and either $s(p_u) = f$ or else $s(p_u)$ is a variable and $f = \omega$. The automaton is said to *succeed*, if it reaches a final state, and is said to *fail*, otherwise.

We emphasize that these matching automata are deterministic (at any given moment an automaton may make at most one transition) and inspect each input symbol at most once. It can be proved that if an automaton reaches a final state v, then M_v contains exactly those terms of T which match s. If the automaton fails, then the input term s is matched by no term in T. There are obvious optimizations of matching automata, such as merging equivalent states (which may be expressed in the discrimination net by labelling edges by sets of symbols); for further details see [11]. For example, Figure 1(b) shows an optimized version of the above net.

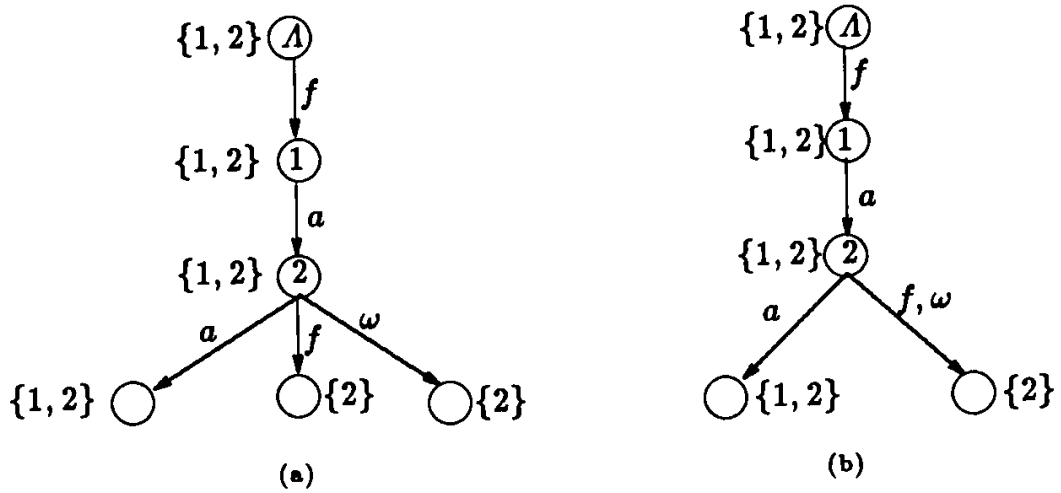

Fig. 1. (a) A discrimination net for $\{1 : f(a,a), 2 : f(a,x)\}$ and (b) its optimization.

In dealing with nonlinear patterns one would first apply discrimination nets to linearized versions of the given terms, where different occurrences of the same variable are considered different, and in a second phase check whether the proposed instantiations for all occurrences of the same variable are consistent; see also Section 6.

It is possible to slightly change the definition of discrimination nets. We say that a term t is *strongly compatible* with a skeleton S if for every pair (p, f_p) in S, p is a position in t and either $t(p) = f_p$ or else $t(p)$ is a variable and $f_p = \omega$. If t is compatible with S, then some instance of t, but not necessarily t itself, is strongly compatible with S. For example, both $f(x, a, b)$ and $f(b, a, a)$ are compatible with the skeleton $\{(\Lambda, f), (1, b)\}$, but only the second term is strongly compatible.

Let discrimination nets be defined as before, but with condition (iii) replaced by: (iii') for each edge (u, v), $S_v = S_u \cup \{(p_u, f)\}$, where f is the label of (u, v), and M_v is the set of all terms in M_u that are *strongly* compatible with S_v. Such discrimination nets can be used as *nondeterministic* automata, as follows. There is a transition $u \rightarrow v$ if D contains an edge (u, v) labelled either by the symbol ω or else by the symbol f, where $s(p_u) = f$. Thus, there may be different transitions from a state on the same input. Whenever a final state is reached, the corresponding match set contains only patterns that match the input term, but may not contain *all* such patterns. For further details see Christian [1]. Figure 2 illustrates the differences between the deterministic and nondeterministic versions of a discrimination net.

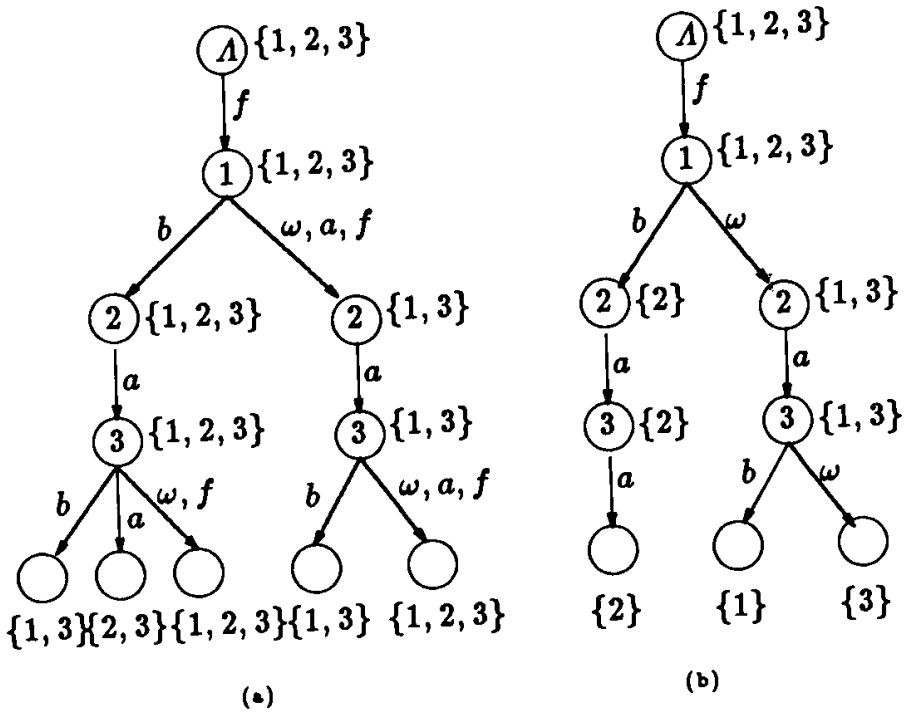

Fig. 2. Deterministic (a) and nondeterministic (b) discrimination nets for $\{1 : f(x, a, b),\ 2 : f(b, a, a),\ 3 : f(x, a, y)\}$

Nondeterministic automata may require that all possible computation sequences be enumerated, so that the same symbol of the input term s may have to be inspected repeatedly, as often as once for every pattern. Therefore they may be less efficient than deterministic automata in practice. On the other hand, they are smaller in size and can often be more easily updated, as patterns can readily be added to or deleted from T. Thus, in applications where the set of terms to match with keeps changing (such as in completion procedures), nondeterministic automata may be preferable, while for fixed sets T deterministic automata are more efficient.

The associative-commutative matching algorithm proposed in this paper is based on the use of discrimination nets, but applies to any version thereof, whether deterministic or nondeterministic, adaptive or non-adaptive. For reasons of simplicity, our exposition is given in terms of deterministic automata.

3 Associative-Commutative Matching

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 f. We write $f \in AC$ to indicate that f is such an associative-commutative symbol, and denote by $=_{AC}$ the equational theory induced by the set of equations AC. Thus, $s =_{AC} t$ if and only if s and t are equivalent under associativity and commutativity.

A term t is said to *AC-match* another term s if there exists a substitution σ, such that $t\sigma =_{AC} s$. We shall first consider the problem of deciding, for given linear terms $t_1, \ldots, t_n$, which ones AC-match a specified term s.

In dealing with associativity and commutativity it is of advantage to "flatten" terms and allow varying arity for AC-symbols. More precisely, if f is an AC-symbol, then $f(X)$ is a syntactically valid term, if the sequence of terms X consists of at least two terms. (We denote the length of a sequence X by $|X|$.)

By L we denote the set of all rewrite rules (also called *flattening rules*) of the form

$$f(X, f(Y), Z) \to f(X, Y, Z), \qquad f \in AC,\ |X| + |Z| \geq 1,\ |Y| \geq 2.$$

Terms that can not be rewritten by L are said to be *flat*. The normal form of a term t under the flattening rules is denoted by $\bar{t}$. Let us denote by $\sim$ the smallest symmetric rewrite relation (also called the *permutation congruence*) for which

$$f(X, u, Y, v, Z) \sim f(X, v, Y, u, Z), \qquad \text{if } f \in AC.$$

It is well-known that the flattened versions of terms equivalent under AC are unique up to the permutation congruence. Consequently, a term t AC-matches a term s if and only if there exists a substitution σ, such that $\overline{t\sigma} \sim \bar{s}$. If we assume that $x\sigma$ is a flat term, for all variables x, then $\bar{t}\sigma$ is almost a flat term as

well. The only subterms to which a flattening rule can be applied are of the form $f(Y, x\sigma, Y')$, where $f \in AC$ and $x\sigma = f(Z)$, for some AC-symbol f, in which case $f(Y, f(Z), Y')$ will be flattened to $f(Y, Z, Y')$. The same effect is achieved by substituting for x, not the term $f(Z)$, but instead the *sequence* of terms Z. Denoting by $\overline{\sigma}$ this substitution, with sequences of terms substituted for variable arguments of AC-symbols, we find that $\overline{t\sigma} \sim \overline{s}$ is equivalent to $\overline{t}\,\overline{\sigma} \sim \overline{s}$.

In sum, the AC-matching problem can essentially be reduced to ordinary matching up to permutation of arguments of AC-symbols, provided terms are flattened first. Henceforth, we shall consider only flattened linear terms and speak of AC-matching to refer matching up to permutation. In the next section we apply discrimination nets to this problem.

4 Associative-Commutative Discrimination Nets

The *AC-nesting depth* at a position in a term is recursively defined as follows. The AC-nesting depth at Λ is 0. 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 $t(p)$ is an AC-symbol, and n, otherwise. The AC-nesting depth of a term is the maximum AC-nesting depth of any of its positions. By the *top-layer* $\hat{t}$ of a term t we mean 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 $g(a, f(b, c))$ is $g(a, f)$. Observe that the top-layer of a term is a syntactically valid term if AC-symbols are regarded as constants. If t contains no AC-symbols, then $\hat{t} = t$. Furthermore, if $s \sim t$, then $\hat{s} = \hat{t}$.

Now suppose s and t are flat terms, such that $t\sigma \sim s$, where $x\sigma$ may be a sequence of terms, if x occurs as argument of an AC-symbol in t. Then (i) $\hat{t}$ matches $\hat{s}$ and (ii) for all positions p of AC-nesting depth 0 in t, if $t(p)$ is an AC-symbol, 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 can be characterized in terms of conditions (i) and (ii).

Condition (i) represents a standard matching problem. Let us consider condition (ii). Suppose $t|_p = f(t_1, \ldots, t_m)$ and $s|_p = f(s_1, \ldots, s_n)$, where $f \in AC$. Let us also assume, without loss of generality, that for some k, $0 \leq k \leq m$, no term $t_1, \ldots, t_k$ is a variable, while all terms $t_{k+1}, \ldots, t_m$ are variables. 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 $s_i\sigma \sim t_j$, for some substitution σ. It can easily be seen that if (a) either $n = m$ or $n > m > k$, and (b) there is a 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. (Recall that we consider only linear terms.)

In sum, a flat term t AC-matches another flat term s if and only if (i) the top-layer of t matches the top-layer of s and (ii) maximal AC-subterms of s are AC-matched by corresponding subterms of t. The second condition can be checked by AC-matching proper subterms of t and s and using bipartite graph matching. The above observations motivate the following definitions.

If T is a set of terms, we denote by $\hat{T}$ the set of all top-layers of terms in T. In a discrimination net for such a set $\hat{T}$, nodes v, for which there is an edge (u, v) labelled by an AC-symbol, are called *AC-nodes*. Furthermore, we denote by L_v the set of all terms $t|_{p_u}$, for which $t \in T$, $\hat{t} \in M_v$, and p_u is a non-variable position in t; and by R_v the set of all non-variable terms that occur as top-level arguments of terms in L_v.

An *AC-discrimination net* is a hierarchically structured collection of standard discrimination nets. Formally, an AC-discrimination net for a set of flat terms T consists of (i) a standard discrimination net D (the *top-level net*) for the set $\hat{T}$ of top-layers of T, and (ii) associated with each AC-node v in D, an AC-discrimination net (an *AC-subnet*) for the set R_v. By the *depth* of an AC-discrimination net for T we mean the maximal AC-nesting depth of any term in T.

An AC-discrimination net for a set of terms T defines an AC-matching automaton, which is deterministic and inspects each input symbol at most once, but differs from a standard matching automaton in that it dynamically computes current match sets.

The algorithm `AC-match`, which accepts as input a flat term s and an associative-commutative discrimination net D for a set of flat terms T, is defined as follows:

1. Let u be the root of D, D' its top-level standard discrimination net, and M'_u be T.
2. If u is leaf, then return $M_u \cap M'_u$.
3. Otherwise, let (u, v) be the edge in D' corresponding to the symbol $f = s(p_u)$. (If f is a variable, the corresponding edge is the one labelled by ω.)
4. If v is an AC-node, then $s|_{p_u}$ is a term $f(s_1, \ldots, s_n)$. Recursively apply the algorithm `AC-match` to each term s_i and the AC-subnet D_v associated with v, to determine which terms in R_v AC-match s_i.
   Define, for each term $t = f(t_1, \ldots, t_k, x_{k+1}, \ldots, x_m)$ in L_v with non-variable arguments $t_1, \ldots, t_k$, a bipartite graph G_t with vertices $s_1, \ldots, s_n$ and $t_1, \ldots, t_k$ and with all pairs (s_i, t_j), for which s_i AC-matches t_j, as edges. If (a) either $n = m$ or else $n > m > k$, and (b) there is a matching of size k in the graph G_t, then t AC-matches $f(s_1, \ldots, s_n)$.
   Let L'_v be the set of all terms in L_v that AC-match $f(s_1, \ldots, s_n)$. Let M'_v be the set of all terms in M'_u, such that either $t|_{p'}$ is a variable, for some prefix p' of p_u, or else $t|_{p_u}$ is in L'_v.
5. Let u be v and M'_u be M'_v and go to step 2.

This algorithm is correct:

Theorem 1. *Let D be an associative-commutative discrimination net for a set of flat terms T, and let s be a flat term. Then the algorithm* `AC-match` *returns, for input D and s, the set of all terms in T that AC-match s.*

Proof. The proof is by induction on the depth of D. Let T' be the set returned by `AC-match` for input D and s and let t be a term in T'. We first prove that (i)

$\hat{t}$ matches $\hat{s}$ and (ii) for all positions p of AC-nesting depth 0 in t, if $t(p)$ is an AC-symbol, then $t|_p$ AC-matches $s|_p$.

Let u be the root of the top-layer discrimination net D' of D and let v be the final state that is reached by the matching automaton A' defined by D' on input $\hat{s}$. The same node v is the last one reached by `AC-match`. Since t is in M_v, we may infer that $\hat{t}$ matches $\hat{s}$.

Let p be a position of AC-nesting depth 0 in t, and suppose $t|_p = f(t_1, \ldots, t_m)$ and $s|_p = f(s_1, \ldots, s_n)$, for some AC-symbol f. Let (u', v') be the edge considered in step 3 of `AC-match` at the time when the symbol at position p in s is inspected. Since any AC-subnet of D is of smaller depth, we may use the induction hypothesis to infer that the set M'_v computed in the succeeding step 4 contains only terms t' such that $t'|_p$ AC-matches $s|_p$. Thus, conditions (i) and (ii) are indeed satisfied for t.

On the other hand, it can easily be seen that that if a term t is contained in T but not in T', then t does not AC-match s. This completes the proof. □

Theorem 2. *Let D be an associative-commutative discrimination net for a set of flat terms T, and let s be a flat term. The algorithm* `AC-match` *determines the set of all terms in T that AC-match s in time $O(n) + O(mn^{1.5})$, where n is the size of the subject term s and m is the sum of the sizes of all pattern terms in T.*

Proof. First note that each symbol in the input term s is inspected at most once. If the symbol inspected is not AC, then its processing takes only a constant amount of time (steps 3 and 5).

In the case of an AC-symbol, which is the root of a subterm $f(s_1, \ldots, s_{n_i})$ of s, we need to apply a bipartite graph matching algorithm to graphs G_t, for certain sets of patterns t, all of which are subterms of terms in the initial set T. Let $n_i + m_t$ be the number of vertices in G_t. Then the number of edges is at most $n_i m_t$, and a maximum bipartite matching on G_t can be computed in time $O(m_t n_i \sqrt{n_i})$. The total time for all bipartite matchings on graphs with $V_1 = \{s_1, \ldots, s_{n_i}\}$ is thus no more than $O(mn_i^{1.5})$. Since $\sum_i n_i \leq n$, the total time for all bipartite graph matchings done during the course of the algorithm is no more than $O(mn^{1.5})$.

Furthermore, the computation of current match sets M'_v and returned match sets $M_u \cap M'_u$ can be done in time proportional to the size of the sets involved, which is no more than m. Therefore the total time for all of these operations cannot exceed $O(nm)$. We have thus established the desired bound of $O(n) + O(mn^{1.5})$ on the running time of `AC-match`. □

Finally, a few remarks about the space complexity of the discrimination net. The time to build the discrimination net is directly related to its space complexity, and the size of the discrimination net can become quite large. In fact, the worst-case lower bound on space of a discrimination net is exponential in the number of terms [11]. However, at the expense of increasing the matching time it is possible to reduce the space complexity. One approach is to build a nondeterministic net based on strong compatibility. Another approach that can yield

much bigger space reductions is to build a different kind of discrimination net based on root-to-leaf path sequences which contain both positions and function symbols. Details about such nets appear in [10, 12, 8].

5 Secondary Automata

In the above AC-matching algorithm we have applied indexing techniques to the non-AC portions of given pattern terms, and have used bipartite graph matching to deal with AC-subterms. The running time of the algorithm is dominated by the cost of bipartite graph matching. The bipartite graphs that have to be considered are of the form $G = (V_1 \cup V_2, E)$, where $V_1 = \{s_1, \ldots, s_l\}$ is a set of input subterms, $V_2 = \{t_1, \ldots, t_k\}$ is a set of pattern subterms, and E contains an edge (s_i, t_j) if and only if t_j AC-matches s_i.

More precisely, the terms $t_1, \ldots, t_k$ are the non-variable top-level arguments of some occurrence of an AC-operator in a pattern term. It appears that in practice the number k is usually quite small. For instance, in all the associative-commutative rewrite systems listed by Hullot [5], there are no AC-subterms with more than *two* non-variable arguments. We have also analysed several benchmark problems for AC-completion, such as certain group examples similar to the ones used by Christian [1]. In one typical example, a total of 128 rewrite rules were generated during the completion of a canonical system of 35 rules. There were 183 occurrences of AC-symbols on the left-hand sides of these rules, of which 81 had only variables as arguments, while the remaining ones had only one non-variable argument. In the examples we looked at, there was only one instance of an AC-subterm with more than two non-variable arguments. (The number of non-variable arguments in that case was four.)

These observations have led us to design special techniques, called secondary automata, to efficiently handle bipartite graph matching for cases where V_2 is small, say $k \leq 4$.

Suppose graphs are represented by adjacency matrices, with rows corresponding to nodes s_i and columns corresponding to nodes t_j. The edge set E is computed row by row, as the algorithm `AC-match` is applied to each term s_i. Let us denote by G_i the subgraph of G consisting of the first i rows.

For example, suppose we have one pattern $f(g(a,b), g(a,x), y))$, where only f is an AC-symbol. We need to look at bipartite graphs $G = (V_1 \cup V_2, E)$, with $V_2 = \{g(a,b), g(a,x)\}$. The sets V_1 and E depend on the input term. For an input term $f(g(a,d), g(a,c), g(a,b))$, we have $V_1 = \{g(a,d), g(a,c), g(a,b)\}$. The two terms $g(a,d)$ and $g(a,c)$ are AC-matched only by the second term in V_2, while $g(a,b)$ is AC-matched by both pattern subterms, so that we obtain adjacency matrices

$$G_1 = (0\ 1), G_2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}, G = G_3 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

The maximum graph matching is of size 2.

Let us classify bipartite graphs $G = (V_1 \cup V_2, E)$ with $|V_2| = k$, according to the size of a maximum matching and the subset of vertices in V_2 matched in a maximum matching. More precisely, a *matching characteristic* is a collection δ of d-element subsets of V_2, where $d \leq k$. We say that G has matching characteristic δ if, for every set $M \in \delta$, there exists a maximum matching of G on which each vertex in M, but no vertex in $V_2 \setminus M$, is incident. (Thus, the size of the maximum matching is d.) Note that there are only finitely many different sets δ, for any fixed number k.

For example, graphs G_1 and G_2 above have matching characteristic $\{2\}$, while G_3 has matching characteristic $\{1, 2\}$.

We define an automaton S_k, for each k, as follows. Each matching characteristic δ is represented by a unique state of S_k. The input symbols for S_k are bitstrings b of length k (each such bitstring representing a possible row in adjacency matrix). There is a transition from δ to δ' on input b if there is some graph G of matching characteristic δ and with adjacency matrix A, such that the graph G' defined by the adjacency matrix obtained from A by adding row b, has matching characteristic δ'. There is exactly one state δ_0 with $|\delta_0| = 0$, which is taken as the initial state of S_k; and exactly one state δ_k with $|\delta_k| = k$, which is taken as the only final state of S_k. We call S_k a *secondary automaton.*

The secondary automata S_2 is shown in Figure 3. The states are numbered, with matching characteristic as indicated. For instance, state 3 has matching characteristic $\{\{1\}, \{2\}\}$, which is the property of any graph $G = (V_1 \cup V_2, E)$, where V_2 has two elements, E is non-empty, and all edges are incident on the same element (either the first or the second) of V_2.

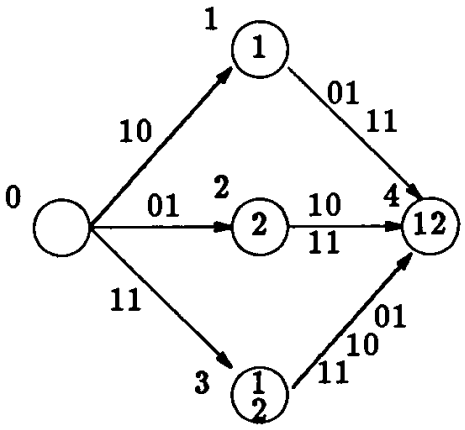

Fig. 3. Secondary automaton of depth 2.

Observe that all transitions lead from a state representing matchings of size d to a state representing matchings of size $d + 1$. In fact, we have:

Lemma 3. *Let $G = (V_1 \cup V_2, E)$ and $G' = (V_1' \cup V_2, E')$ be bipartite graphs, such that $V_1' = V_1 \cup \{v\}$ and $E' = E \cup E_v$, where E_v is a set of edges (v, v'), with $v' \in V_2$. Also, let d be the size of a maximum matching in G. Then G' has either*

a maximum matching of size $d+1$, or else has the same matching characteristics as G.

Proof. Let the matching characteristics of G and G' be δ and δ' respectively. Suppose G' has a maximum matching of size d, $\delta \neq \delta'$ and $M' \in \delta' \setminus \delta$. For any $M \in \delta$, there must be a vertex $w \in M'$ such that $w \notin M$. Clearly, v is matched with a vertex $u \in M$ because otherwise M' would be either of size $d+1$ or included in δ. Let v' be the vertex matched with u in a matching characterized by M. If v' is not matched with any vertex in M', then $M' \in \delta$, which is a contradiction. Suppose u' is v''s match in M'. If $u' \in M$, one can repeat the above argument until a vertex $w \in V_2$ but $w \notin M$ is found; otherwise, we take $w = u'$. In both cases, we have shown for G' the existence of maximum matchings of size $d+1$ incident upon the subset $M \cup \{w\}$ of V_2. A contradiction. □

As an immediate corollary we obtain:

Corollary 4. *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph with $|V_1| = l$ and $|V_2| = k$. Let A be the corresponding adjacency matrix, and A_i be the i-th row of A. Then the secondary automaton S_k, for input $A_1 \ldots A_l$, will reach the state δ that represents the matching characteristics of G.*

In the context of the algorithm AC-match the secondary automata can be used in step 4, to determine whether there exists a bipartite matching of the required size. The number of states in a secondary automaton increases exponentially with k, so that these automata are only practical for small values of k. (The automaton S_1 has only two states, while S_2 has five, and S_3 sixteen.) Fortunately, the value of k tends to be small enough in practical examples, as we have mentioned above.

Secondary automata can also be adapted to yield the actual matching substitutions. If, in traversing a secondary automaton, one can not move to a new state on a row of the adjacency matrix, the input subterm corresponding to the row is a candidate term for the substitution of some variable $x_{k+1}, \cdots, x_m$. By keeping track of such candidates, once the AC-matching is done, one can find consistent substitutions for all the variables in a pattern. For instance, in the pattern $f(g(a,x), g(b,y))$ and the subject $f(g(a,b), g(b,c))$, where both f and g are in AC, y has two candidate substitutions a and c while x has only b. From these candidates, one can determine the final substitution $x \mapsto b$ and $y \mapsto c$ once the algorithm has finished.

Theorem 5. *All bipartite matching problems at all AC-nodes visited during traversal of an AC-discrimination net for a set of flat terms, with total size m, on behalf of an input term of size n can be done in $O(mn)$ time.*

Proof. Since the transition function at each state of a secondary automaton of depth k can be implemented as a k-dimensional table, each transition takes $O(k)$. Thus, using the notation as in proof of Theorem 2, a maximum bipartite matching on G_t can be computed in time $O(m_t n_i)$. Therefore, the time to compute maximum bipartite matchings on graphs with $V_1 = \{s_1, \cdots, s_{n_i}\}$ is no more than $O(mn_i)$ and the total time for all bipartite matchings done during the course of AC-match is no more than $O(mn)$. □

6 Nonlinear Matching

The above (polynomial-time) algorithm applies to linear pattern terms. In its simplest form, the algorithm just determines which patterns AC-match a given input term. In situations where most match attempts are likely to fail (as is typical of theorem proving applications), it may thus make sense to use this version of the algorithm as a "filter," to quickly sort out failed matching attempts. In case of success, a *single* substitution for a pattern can easily be extracted during the algorithm, which is sufficient for the purpose of rewriting with linear terms.

In the case of nonlinear terms, we apply the matching algorithm to linearized versions of patterns in a first phase, and, in a second phase, check whether the proposed substitutions for different occurrences of the same variable are consistent. This requires that *all* matching substitutions be computed, and the AC-matching algorithm has to be extended correspondingly. In particular, we have to compute all maximal bipartite graph matchings—instead of just determining whether a maximum-sized matching exists. (The nonlinear AC-matching problem is NP-complete, so that a considerable increase in computing time, as compared to linear AC-matching, is not surprising.)

7 Concluding Remarks

In this paper, we have proposed a new data structure, called AC-discrimination nets, and used it to speed up many-to-one AC-matching. In any set of patterns the nesting depth of AC-symbols defines a number of levels of non-AC-parts. An AC-discrimination net is a collection of standard discrimination nets, structured in a way that reflects this hierarchy. The AC-matching algorithm, recursively traversing the levels of the net, uses bipartite graph matching on each level to interpret and combine the AC-matching results from lower levels. If no AC-symbols are present in the patterns, the AC-discrimination net specializes to a standard discrimination net.

We have also proposed secondary automata as a novel data structure for further speeding up matching of AC-patterns typically encountered in practice. Secondary automata are only dependent on the number of non-variable arguments within AC-subterms of the pattern, so that all AC-subterms having the same number of non-variable arguments can be handled by a single secondary automaton. This feature makes it attractive for completion as there is no need to create them dynamically when patterns change.

Let us briefly discuss some of the other proposals for many-to-one AC-matching. Lugiez and Kounalis [6] proposed an algorithm to compile pattern matching of AC function definitions in functional languages. In their approach a complete pattern tree (that guides matching) is constructed by enumerating certain AC terms. Such an enumeration appears to be prohibitively expensive in terms of both time and space. Nicolaita [9] described a method which is based on Christian's discrimination net [1]. The main drawback with his method is that it requires that the arguments of AC-subterms in the patterns as well as in

the subject be ordered. It is not clear whether the ordering constraints can be efficiently maintained in completion, where both the patterns and the subjects keep changing dynamically. (The kind of ordering to be used is left unspecified and the complexity of the algorithm is left open.) Lugiez and Moysset describe a bottom-up procedure for AC-matching in which patterns are preprocessed into a large nondeterministic tree automaton [7]. The running time of the algorithms is not analysed, but the construction of a deterministic automaton (from the nondeterministic one) is bound to be expensive.

Finally we remark that our linear matching algorithm can be directly used to deal with function symbols that are only commutative, in which case no flattening rules are applied to such function symbols. For function symbols that are only associative we simply need to replace bipartite graph matching with ordered bipartite matching.

Acknowledgments. We wish to thank the referees for their comments. This research has been supported in part by NSF grant CCR-9102159.

References

1. J. Christian. Fast Knuth-Bendix completion : Summary. In *RTA'89*, pages 551–555. Springer-Verlag LNCS 355, 1989.
2. D. Kapur D. Benanav and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3:203–216, 1987.
3. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–309. Elsevier, 1990.
4. A. Graf. Left-to-right tree pattern matching. In *RTA'91*, pages 323–334. Springer-Verlag LNCS 488, 1991.
5. J.-M. Hullot. A catalogue of canonical term rewriting systems. Technical Report CSL-113, SRI International, April 1980.
6. E. Kounalis and D. Lugiez. Compilation of pattern matching with associative-commutative functions. In *TAPSOFT'91*, pages 57–73. Springer-Verlag LNCS 493, 1991.
7. D. Lugiez and J.L. Moysset. Complement problems and tree automata in AC-like theories. To appear in *STACS'93*.
8. W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. Technical Report MCS-P191-1190, Argonne National Laboratory, Argonne, Illinois, USA, January 1991.
9. D. Nicolaita. An indexing scheme for AC-equational theories. Technical report, Research Institute for Infomatics, Bucharest, Romania, February 1992.
10. R. Ramesh, I.V. Ramakrishnan, and D.S. Warren. Automata-driven indexing of prolog clauses. In *Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, San Francisco, 1990.
11. R.C. Sekar, R. Ramesh, and I.V. Ramakrishnan. Adaptive pattern matching. In *ICALP'92*, pages 247–260. Springer-Verlag LNCS 623, 1992.
12. M. E. Stickel. The path-indexing method for indexing terms. Technical Report 473, SRI International, Menlo Park, California, USA, October 1989.
13. R. M. Verma and I.V. Ramakrishnan. Tight complexity bounds for term matching problems. *Information and Computation*, 101(1):33–69, November 1992.