数理解析研究所講究録
992巻 1997年 47-57

Learning One-Variable Pattern Languages Very Efficiently *

THOMAS ERLEBACH, PETER ROSSMANITH, HANS STADTHERR, ANGELIKA STEGER,
Institut für Informatik, Technische Universität München, D–80290 München, Germany
{erlebach|rossmani|stadther|steger}@informatik.tu-muenchen.de

AND THOMAS ZEUGMANN
Dept. of Informatics, Kyushu University, Fukuoka 812-81, Japan, thomas@i.kyushu-u.ac.jp

Abstract

A pattern is a finite string of constant and variable symbols. The language generated by a pattern is the set of all strings of constant symbols which can be obtained from the pattern by substituting non-empty strings for variables. Descriptive patterns are a key concept for inductive inference of pattern languages. A pattern π is descriptive for a given sample if the sample is contained in the language $L(\pi)$ generated by π and no other pattern having this property generates a proper subset of the language $L(\pi)$. The best previously known algorithm for computing descriptive one-variable patterns requires time $O(n^4 \log n)$, where n is the size of the sample. We present a simpler and more efficient algorithm solving the same problem in time $O(n^2 \log n)$. In addition, we give a parallel version of this algorithm that requires time $O(\log n)$ and $O(n^3/\log n)$ processors on an EREW-PRAM. Previously, no efficient parallel algorithm was known for this problem.

Using a modified version of the sequential algorithm as a subroutine, we devise a learning algorithm for one-variable patterns whose *expected total learning time* is $O(\ell^2 \log \ell)$ provided the sample strings are drawn from the target language according to a probability distribution with expected string length ℓ. The probability distribution must be such that strings of equal length have equal probability, but can be arbitrary otherwise. Furthermore, we show how the algorithm for descriptive one-variable patterns can be used for learning one-variable patterns with a polynomial number of superset queries.

1. Introduction

A *pattern* is a concatenation of constant symbols and variable symbols. The language generated by a pattern is the set of all strings obtained by substituting strings of constants for the variables of the pattern (cf. [1]). Pattern languages and variations thereof have been

*A full version of this paper appeared as *Efficient Learning of One-Variable Pattern Languages from Positive Data*, DOI-TR-128, Dept. of Informatics, Kyushu University 33, December 12, 1997; http://www.i.kyushu-u.ac.jp/~thomas/treport.html

widely investigated (cf., e.g., [14, 15, 16]).This continuous interest in pattern languages has many reasons, among them the learnability in the limit of all pattern languages from text (cf. [1, 2]). In particular, the pattern languages can be learned by outputting *descriptive patterns* as hypotheses (cf. [1]). The resulting learning algorithm is *consistent*, and *set-driven*. A learner is said to be set-driven provided its output depends only on the range of its input (cf., e.g., [17, 19]). In general, consistency and set-drivenness considerably limit the learning capabilities (cf., e.g., [12, 19]). But there is also a major disadvantage, since no efficient algorithm computing descriptive patterns is known. Finding a descriptive pattern of maximum possible length is $\mathcal{NP}$-complete. Thus, it is unlikely that there is a polynomial-time algorithm computing descriptive patterns.

Hence, it is natural to ask whether efficient pattern language learners can benefit from the concept of descriptive patterns. Several authors looked at special cases, e.g., *regular* patterns, *non-cross* patterns, and unions of at most k regular pattern languages (k *a priori* fixed). In all these cases, descriptive patterns are polynomial-time computable (cf., e.g., [16]). A further restriction is obtained by *a priori* bounding the number k of different variables occurring in a pattern (k-variable patterns). It is still open if there are polynomial-time algorithms computing descriptive k-variable patterns for any fixed $k > 1$ (cf. [8, 10]). Lange and Wiehagen [11] provided a learner *LWA* that is allowed to output inconsistent guesses. Their algorithm achieves polynomial update time. It is still set-driven (cf. [18]), and even *iterative*, thus beating Angluin's [1] algorithm with respect to its space complexity. But what can be said concerning the overall time needed until convergence, i.e., the so-called *total learning time*? Generally, the total learning time is unbounded in the worst case. Thus, we study the *expected* total learning time. For the *LWA*, the expected total learning time is exponential in the number of different variables occurring in the target pattern (cf. [18]).

In the present paper we consider the special case of 1-variable patterns. For that case, Angluin [1] provided an algorithm for computing descriptive patterns having running time $O(n^4 \log n)$ for inputs of size n. She hoped further study would provide insight for a uniform improvement. We present such an improvement, i.e., an algorithm computing descriptive 1-variable patterns in $O(n^2 \log n)$ steps. Moreover, our algorithm can be parallelized efficiently, using $O(\log n)$ time and $O(n^3 / \log n)$ processors on an EREW-PRAM. Previously, no efficient parallel algorithm for learning 1-variable pattern languages was known.

Additionally, we present a version of the sequential algorithm still learning all 1-variable pattern languages that has *expected total learning time* $O(\ell^2 \log \ell)$ if the sample strings are drawn from the target language according to a probability distribution D having expected string length ℓ. D can be arbitrary except that strings of equal length have equal probability.

Finally, we deal with *active* learning. Now the learner gains information by *querying* an oracle. The algorithm for descriptive 1-variable patterns can be used for learning 1-variable patterns by asking polynomially many superset queries. A different algorithm learning all pattern languages by asking polynomially many superset queries is known (cf. [3]), but it queries patterns with *more than one variable* even if the target pattern is a 1-variable pattern.

1.1. *The Pattern Languages and Inductive Inference*

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of all natural numbers, and let $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. For all real numbers x we define $\lfloor x \rfloor$ to be the greatest integer less than or equal to x.

Let $\mathcal{A} = \{0, 1, \ldots\}$ be any finite alphabet containing at least two elements. $\mathcal{A}^*$ denotes the free monoid over $\mathcal{A}$. The set of all finite non-null strings over $\mathcal{A}$ is denoted by $\mathcal{A}^+$, i.e., $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$, where ε denotes the empty string. Let $X = \{x_i \mid i \in \mathbb{N}\}$ be an infinite set of variables with $\mathcal{A} \cap X = \emptyset$. *Patterns* are strings from $(\mathcal{A} \cup X)^+$, e.g., 01, $1x_0x_00x_1x_2x_0$ are patterns. The length of a string $s \in \mathcal{A}^*$ and of a pattern π is denoted by $|s|$ and $|\pi|$, respectively. By *Pat* we denote the set of all patterns.

Let $\pi \in Pat$, $1 \leq i \leq |\pi|$; by $\pi(i)$ we denote the i-th symbol in π, e.g., $0x_0111(2) = x_0$. If $\pi(i) \in \mathcal{A}$, then $\pi(i)$ is called a *constant*; otherwise $\pi(i) \in X$, i.e., $\pi(i)$ is a *variable*. We use $s(i)$, $i = 1, \ldots, |s|$, to denote the i-th symbol in $s \in \mathcal{A}^+$. By $\#\mathrm{var}(\pi)$ we denote the number of different variables occurring in π, and $\#_{x_i}(\pi)$ denotes the number of occurrences of variable x_i in π. If $\#\mathrm{var}(\pi) = k$, then π is a *k-variable pattern*. By Pat_k we denote the set of all *k-variable patterns*. In the case $k = 1$ we denote the variable occurring by x.

Let $\pi \in Pat_k$, and $u_0, \ldots, u_{k-1} \in \mathcal{A}^+$. $\pi[u_0/x_0, \ldots, u_{k-1}/x_{k-1}]$ denotes the string $s \in \mathcal{A}^+$ obtained by substituting u_j for each occurrence of x_j in π. For all $\pi \in Pat_k$ we define the *language generated by pattern* π by $L(\pi) = \{\pi[u_0/x_0, \ldots, u_{k-1}/x_{k-1}] \mid u_0, \ldots, u_{k-1} \in \mathcal{A}^+\}$[1]. By PAT_k we denote the set of all *k-variable pattern languages*. $PAT = \bigcup_{k \in \mathbb{N}} PAT_k$ denotes the set of all pattern languages over $\mathcal{A}$.

1-variable pattern languages are more interesting for practical purposes than general pattern languages, since several problems are decidable or even efficiently solvable for PAT_1 but undecidable or $\mathcal{NP}$–complete for PAT; e.g. for general pattern languages the word problem is $\mathcal{NP}$–complete (cf. [1]) and the inclusion problem is undecidable (cf. [9]), but both problems are decidable in linear time for 1-variable pattern languages. On the other hand, PAT_1 is still incomparable to the regular and context free languages. In the remaining sections of this paper, we will be mainly concerned with Pat_1 and PAT_1.

A finite set $S = \{s_0, s_1, \ldots, s_r\} \subseteq \mathcal{A}^+$ is called a *sample*. A pattern π is *consistent* with a sample S if $S \subseteq L(\pi)$. A (1-variable) pattern π is *descriptive* for S if it is consistent with S and there is no other consistent (1-variable) pattern τ such that $L(\tau) \subset L(\pi)$.

Next, we define the relevant learning models. First, we deal with inductive inference of formal languages from text (cf., e.g., [13, 19]). Let L be a language; every infinite sequence $t = (s_j)_{j \in \mathbb{N}}$ of strings with $\mathrm{range}(t) = \{s_j \mid j \in \mathbb{N}\} = L$ is said to be a *text* for L. By *Text*(L) we denote the set of all texts for L. Let t be a text, and $n \in \mathbb{N}$. We set $t_n = s_0, \ldots, s_n$, and call t_n the initial segment of t of length $n + 1$. By t_n^+ we denote the range of t_n. We define an *inductive inference machine* (abbr. IIM) to be an algorithmic device that takes as its input larger and larger initial segments of a text t, outputs on every input a pattern as hypothesis, and then requests the next input (cf. [6]).

DEFINITION 1. *PAT is called learnable in the limit from text iff there is an IIM M such that for every* $L \in PAT$ *and every* $t \in Text(L)$,
(1) *for all* $n \in \mathbb{N}$, $M(t_n)$ *is defined,*
(2) *there is a pattern* $\pi \in Pat$ *such that* $L(\pi) = L$ *and for almost all* $n \in \mathbb{N}$, $M(t_n) = \pi$.

[1] We study so-called *non-erasing* substitutions. It is also possible to consider *erasing* substitutions where variables may be replaced by empty strings, leading to a different class of languages (cf. [4]).

The learnability of the 1-variable pattern languages is defined analogously by replacing PAT and Pat by PAT_1 and Pat_1, respectively.

When dealing with set-driven learners, it is often technically advantageous to describe them in dependence of the relevant set t_n^+, i.e., a *sample*, obtained as input. A learner is said to be set-driven iff its output depends only on the range of its input (cf., e.g., [17, 19]). Let $S = \{s_0, s_1, \ldots, s_r\}$; we set $n = \sum_{j=0}^{r} |s_j|$, and refer to n as the *size* of sample S.

PAT and PAT_1 constitute an *indexable class* $\mathcal{L}$ of uniformly recursive languages, i.e., there are an effective enumeration $(L_j)_{j\in\mathbb{N}}$ of $\mathcal{L}$ and a recursive function f such that for all $j \in \mathbb{N}$ and all $s \in \mathcal{A}^*$ we have $f(j,s) = 1$ if $s \in L_j$, and $f(j,s) = 0$ otherwise. We refer to indexable classes of uniformly recursive languages as indexable classes for short.

Except in Section 3, where we use the PRAM-model, we assume the same model of computation and representation of patterns in [1]. Next we define the update time and the total learning time. Let M be any IIM. Then, for every $L \in PAT$ and $t \in Text(L)$, let $Conv(M,t)$ = the least number m such that for all $n \geq m$, $M(t_n) = M(t_m)$ denote the *stage of convergence* of M on t. By $T_M(t_n)$ we denote the time to compute $M(t_n)$, and we refer to $T_M(t_n)$ as the *update time* of M. Furthermore, the *total learning time* taken by the IIM M on successive input t is defined as $TT(M,t) = \sum_{n=0}^{Conv(M,t)} T_M(t_n)$.

Finally, we define *learning via queries*. The objects to be learned are the elements of an indexable class $\mathcal{L}$. We assume an indexable class $\mathcal{H}$ as well as a fixed effective enumeration $(h_j)_{j\in\mathbb{N}}$ of it as hypothesis space for $\mathcal{L}$. A hypothesis h describes a target language L iff $L = h$. The source of information about the target L are queries to an *oracle*. We distinguish *membership, equivalence, subset, superset*, and *disjointness* queries (cf. [3]). Input to a membership query is a string s, and the output is *yes* if $s \in L$ and *no* otherwise. For the other queries, the input is an index j and the output is *yes* if $L = h_j$ (equivalence query), $h_j \subseteq L$ (subset query), $L \subseteq h_j$ (superset query), and $L \cap h_j = \emptyset$ (disjointness query), and *no* otherwise. If the reply is no, a *counterexample* is returned, too, i.e., a string $s \in L \triangle h_j$ (the symmetric difference of L and h_j), $s \in h_j \setminus L$, $s \in L \setminus h_j$, and $s \in L \cap h_j$, respectively. We always assume that all queries are *answered truthfully*.

DEFINITION 2 (cf. [3]). *Let $\mathcal{L}$ be any indexable class and let $\mathcal{H}$ be a hypothesis space for it. A learning algorithm exactly identifies a target $L \in \mathcal{L}$ with respect to $\mathcal{H}$ with access to a certain type of queries if it always halts and outputs an index j such that $L = h_j$.*

Note that the learner is allowed only *one guess* which must be correct. The *complexity* of a query learner is measured by the number of queries to be asked in the worst-case.

2. An Improved Sequential Algorithm

We present an algorithm computing a descriptive pattern $\pi \in Pat_1$ for a sample $S = \{s_0, s_1, \ldots, s_{r-1}\} \subseteq \mathcal{A}^+$ as input. Without loss of generality, we assume that s_0 is the shortest string in S. Our algorithm runs in time $O(n\,|s_0| \log |s_0|)$ and is simpler and much faster than Angluin's [1] algorithm, which needs time $O(n^2|s_0|^2 \log |s_0|)$. Angluin's [1] algorithm computes explicitly a representation of the set of all consistent 1-variable patterns for S. We avoid to represent *all* consistent patterns, but work with a *polynomial-size* subset thereof. Next, we review and establish a few basic properties.

LEMMA 1 (Angluin [1], Lemma 3.9). *Let $\tau \in Pat$, and let $\pi \in Pat_1$. Then $L(\tau) \subseteq L(\pi)$ if and only if τ can be obtained from π by substituting a pattern $\varrho \in Pat$ for x.*

For a pattern π to be consistent with S, there must be strings $\alpha_0, \ldots, \alpha_{r-1} \in \mathcal{A}^+$ such that s_i can be obtained from π by substituting α_i for x, for all $0 \leq i \leq r-1$. Given a consistent pattern π, the set $\{\alpha_0, \ldots, \alpha_{r-1}\}$ is denoted by $\alpha(S, \pi)$. Furthermore, a sample S is called *prefix-free* if $|S| > 1$ and no string in S is a prefix of all other strings in S.

LEMMA 2. *If S is a prefix-free sample then there exists a descriptive pattern $\pi \in Pat_1$ for S such that at least two strings in $\alpha(S, p)$ start with a different symbol.*

Let $Cons(S) = \{\pi \mid \pi \in Pat_1,\ S \subseteq L(\pi),\ \exists i, j[i \neq j,\ \alpha_i, \alpha_j \in \alpha(S, \pi),\ \alpha_i(1) \neq \alpha_j(1)]\}$. $Cons(S)$ is a subset of all consistent patterns for S, and $Cons(S) = \emptyset$ if S is not prefix-free.

LEMMA 3. *Let S be any prefix-free sample. Then $Cons(S) \neq \emptyset$, and every pattern $\pi \in Cons(S)$ of maximum length is descriptive for S.*

Next, we explain how to handle non-prefix-free samples. The algorithm checks whether the input sample consists of a single string s. If this happens, it outputs s and terminates. Otherwise, it tests whether s_0 is a prefix of all other strings $s_1, \ldots, s_{r-1}$. If it is, it outputs $ux \in Pat_1$, where u is the prefix of s_0 of length $|s_0|-1$, and terminates. Clearly, $S \subseteq L(ux)$. Suppose there is a pattern τ such that $S \subseteq L(\tau)$, and $L(\tau) \subset L(ux)$. Then Lemma 1 applies, i.e., there is a ϱ such that $\tau = ux[\varrho/x]$. But this implies $\varrho = x$, since otherwise $|\tau| > |s_0|$, and thus, $S \not\subseteq L(\tau)$. Consequently, ux is descriptive.

Otherwise, $|S| \geq 2$ and s_0 is not a prefix of all other strings in S. Hence, S is prefix-free and Lemma 3 applies. Thus, it suffices to find and output a longest pattern in $Cons(S)$.

Let $k, l \in \mathbb{N}$, $k > 0$; we call patterns π with $\#_x(\pi) = k$ and l occurrences of constants (k, l)-patterns. Every pattern $\pi \in Cons(S)$ satisfies $|\pi| \leq |s_0|$. Thus, there can only be a (k, l)-pattern in $Cons(S)$ if there is an $m_0 \in \mathbb{N}^+$ satisfying $|s_0| = km_0 + l$. Here m_0 refers to the length of the string substituted for the occurrences of x in the relevant (k, l)-pattern to obtain s_0. Therefore, there are at most $\lfloor |s_0|/k \rfloor$ possible values of l for a fixed value of k. Hence, the number of possible (k, l)-pairs for which (k, l)-patterns in $Cons(S)$ can exist is bounded by $\sum_{k=1}^{|s_0|} \left\lfloor \frac{|s_0|}{k} \right\rfloor = O(|s_0| \cdot \log |s_0|)$.

The algorithm considers all possible (k, l)-pairs in turn. We describe the algorithm for one specific (k, l)-pair. If there is a (k, l)-pattern $\pi \in Cons(S)$, the lengths m_i of $\alpha_i \in \alpha(S, \pi)$ must satisfy $m_i = (|s_i| - l)/k$. α_i is the substring of s_i of length m_i starting at the first position where the input strings differ. If $(|s_i| - l)/k \notin \mathbb{N}$ for some i, then there is no consistent (k, l)-pattern and no further computation is performed for this (k, l)-pair. The following lemma shows that the (k, l)-pattern in $Cons(S)$ is unique, if it exists at all.

LEMMA 4. *Let $S = \{s_0, \ldots, s_{r-1}\}$ be any prefix-free sample. For every given (k, l)-pair, there is at most one (k, l)-pattern in $Cons(S)$.*

The proof of Lemma 4 directly yields Algorithm 1 below. It either returns the unique (k, l)-pattern $\pi \in Cons(S)$ or *NIL*. We assume a subprocedure taking as input a sample S, and returning the longest common prefix u.

***Algorithm* 1.** On input (k, l), $S = \{s_0, \ldots, s_{r-1}\}$, and u do the following:

$\pi \leftarrow u$; $b \leftarrow 0$; $c \leftarrow |u|$;

while $b + c < k + l$ and $b \leq k$ and $c \leq l$ **do**

  if $s_0(bm_0 + c + 1) = s_i(bm_i + c + 1)$ for all $1 \leq i \leq r-1$

    then $\pi \leftarrow \pi s_0(bm_0 + c + 1)$; $c \leftarrow c + 1$ **else** $\pi \leftarrow \pi x$; $b \leftarrow b + 1$ **fi**

od;

if $b = k$ and $c = l$ and $S \subseteq L(\pi)$ **then return** π **else return** *NIL* **fi**

Note that minor modifications of Algorithm 1 perform the consistency test $S \subseteq L(\pi)$ even while π is constructed. Putting Lemma 4 and the fact that there are $O(|s_0| \cdot \log |s_0|)$ many possible (k,l)-pairs together, we directly obtain:

LEMMA 5. $|Cons(S)| = O(|s_0| \log |s_0|)$ *for any prefix-free sample* $S = \{s_0, \ldots, s_{r-1}\}$.

Using Algorithm 1 as a subroutine, Algorithm 2 below for finding a descriptive pattern for a prefix-free sample S follows the strategy exemplified above. Thus, it simply computes all patterns in $Cons(S)$ and outputs one with maximum length. For inputs of size n the overall complexity of the algorithm is $O(n\,|s_0| \log |s_0|) = O(n^2 \log n)$, since at most $O(|s_0| \log |s_0|)$ many tests must be performed, which have time complexity $O(n)$ each.

***Algorithm* 2.** On input $S = \{s_0, \ldots, s_{r-1}\}$ do the following:

$P \leftarrow \emptyset$; **for** $k = 1, \ldots, |s_0|$ and $m_0 = 1, \ldots, \lfloor \frac{|s_0|}{k} \rfloor$ **do**

if there is a $(k, |s_0| - km_0)$-pattern $\pi \in Cons(S)$ **then** $P \leftarrow P \cup \{\pi\}$ **fi od**;

Output a maximum-length pattern $\pi \in P$.

Note that the number of (k,l)-pairs to be processed is often smaller than $O(|s_0| \log |s_0|)$, since the condition $(|s_i| - l)/k \in \mathbb{N}$ for all i restricts the possible values of k if not all strings are of equal length. It is also advantageous to process the (k,l)-pairs in order of non-increasing $k + l$. Then the algorithm can terminate as soon as it finds the first consistent pattern. However, the worst-case complexity is not improved, if the descriptive pattern is x.

Finally, we summarize the main result obtained by the following theorem.

THEOREM 1. *Using Algorithm 2 as a subroutine,* PAT_1 *can be learned in the limit by a set-driven and consistent algorithm having update time* $O(n^2 \log n)$ *on input samples of size* n.

3. An Efficient Parallel Algorithm

Whereas the RAM model has been generally accepted as the most suitable model for developing and analyzing sequential algorithms, such a consensus has not yet been reached in the area of parallel computing. The PRAM model introduced in [5], is usually considered an acceptable compromise. A PRAM consists of a number of processors, each of which has its own local memory and can execute its local program, and all of which can communicate by exchanging data through a shared memory. Variants of the PRAM model differ in the constraints on simultaneous accesses to the same memory location by different processors. The CREW-PRAM allows concurrent read accesses but no concurrent write accesses. For ease of presentation, we describe our algorithm for the CREW-PRAM model. The algorithm can be modified to run on an EREW-PRAM, however, by the use of standard techniques.

No parallel algorithm for computing descriptive 1-variable patterns has been known previously. Algorithm 2 can be efficiently parallelized by using well-known techniques including prefix-sums, tree-contraction, and list-ranking as subroutines (cf. [7]). A parallel algorithm can handle non-prefix-free samples S in the same way as Algorithm 2. Checking S to be singleton or s_0 to be a prefix of all other strings requires time $O(\log n)$ using

$O(n/\log n)$ processors. Thus, we may assume that the input sample $S = \{s_0, \ldots, s_{r-1}\}$ is prefix-free. Additionally, we assume the prefix-test has returned the first position d where the input strings differ and an index t, $1 \leq t \leq r-1$, such that $s_0(d) \neq s_t(d)$.

A parallel algorithm can handle all $O(|s_0| \log |s_0|)$ possible (k,l)-pairs in parallel. For each (k,l)-pair, our algorithm computes a unique candidate π for the (k,l)-pattern in $Cons(S)$, if it exists, and checks whether $S \subseteq L(\pi)$. Again, it suffices to output any obtained pattern having maximum length. Next, we show how to efficiently parallelize these two steps.

For a given (k,l)-pair, the algorithm uses only the strings s_0 and s_t for calculating the unique candidate π for the (k,l)-pattern in $Cons(S)$. This reduces the processor requirements, and a modification of Lemma 4 shows the candidate pattern to remain unique.

Position j_t in s_t is said to be b-corresponding to position j_0 in s_0 if $j_t = j_0 + b(m_t - m_0)$, $0 \leq b \leq k$. The meaning of b-corresponding positions is as follows. Suppose there is a consistent (k,l)-pattern π for $\{s_0, s_t\}$ such that position j_0 in s_0 corresponds to $\pi(i) \in \mathcal{A}$ for some i, $1 \leq i \leq |\pi|$, and that b occurrences of x are to the left of $\pi(i)$. Then $\pi(i)$ corresponds to position $j_t = j_0 + b(m_t - m_0)$ in s_t.

For computing the candidate pattern from s_0 and s_t, the algorithm calculates the entries of an array EQUAL$[j,b]$ of Boolean values first, where j ranges from 1 to $|s_0|$ and b from 0 to k. EQUAL$[j,b]$ is true iff the symbol in position j in s_0 is the same as the symbol in its b-corresponding position in s_t. Thus, the array is defined as follows: EQUAL$[j,b]$ = *true* iff $s_0(j) = s_t(j + b(m_t - m_0))$. The array EQUAL has $O(k|s_0|)$ entries each of which can be calculated in constant time. Thus, using $O(k|s_0|/\log n)$ processors, EQUAL can be computed in time $O(\log n)$. Moreover, a directed graph G, which is easily shown to be a forest of binary in-trees, can be built from EQUAL, and the candidate pattern can be calculated from G using tree-contraction, prefix-sums and list-ranking. Thus, we can prove:

LEMMA 6. *Let $S = \{s_0, \ldots, s_{r-1}\}$ be a sample, and n its size. Given the array* EQUAL, *the unique candidate π for the (k,l)-pattern in $Cons(S)$, or NIL, if no such pattern exists, can be computed on an EREW-PRAM in time $O(\log n)$ using $O(k|s_0|/\log n)$ processors.*

Now, the algorithm has either discovered that no (k,l)-pattern exists, or it has obtained a candidate (k,l)-pattern π. In the latter case, it has to test whether π is consistent with S.

LEMMA 7. *Given a candidate pattern π, a parallel algorithm can check whether π is consistent with a sample S of size n in time $O(\log n)$ with $O(n/\log n)$ processors on a CREW-PRAM.*

Putting it all together, we obtain the following theorem.

THEOREM 2. *There exists a parallel algorithm that computes descriptive 1-variable patterns in time $O(\log n)$ using $O(|s_0| \max\{|s_0|^2, n \log |s_0|\}/\log n) = O(n^3/\log n)$ processors on a CREW-PRAM for samples $S = \{s_0, \ldots, s_{r-1}\}$ of size n.*

Note that the product of the time and the number of processors of our algorithm is the same as the time spent by the improved sequential algorithm above whenever $|s_0|^2 = O(n \log |s_0|)$.

4. Analyzing the Expected Total Learning Time

In this section, we deal with the total learning time of our sequential learner. Let π be the target pattern. The total learning time of any algorithm trying to infer π is *unbounded* in the worst case, since there are infinitely many strings in $L(\pi)$ that can mislead it. However, in the *best case* two examples, i.e., $\pi[0/x]$ and $\pi[1/x]$, always suffice for a learner outputting descriptive patterns as guesses. Hence, we assume that the strings presented to the algorithm are drawn from $L(\pi)$ according to a certain probability distribution and compute the *expected total learning time* of our algorithm. The probability distribution must satisfy two criteria: any two strings in $L(\pi)$ of equal length must have equal probability, and the expected string length must be finite. We refer to such distributions as *proper* probability distributions.

We design a learning algorithm inferring a pattern $\pi \in Pat_1$ with expected total learning time $O(\ell^2 \log \ell)$, where ℓ is the expected length of a string drawn from $L(\pi)$. It is advantageous *not* to calculate a descriptive pattern each time a new string is read. Instead, our learning Algorithm 1LA reads a certain number of strings before it starts to perform any computations at all. It waits until the length of a sample string is smaller than the number of sample strings read so far and until at least two *different* sample strings have been read. During these first two phases, it outputs s_1, the first sample string, as its hypothesis if all sample strings read so far are the same, and x otherwise. If π is a constant pattern, i.e., $|L(\pi)| = 1$, the correct hypothesis is always output and the algorithm never reaches the third phase. Otherwise, the algorithm uses a modified version of Algorithm 2 to calculate a set P' of candidate patterns when it enters Phase 3. More precisely, it does not calculate the whole set P' at once. Instead, it uses the function *first_cand* once to obtain a longest pattern in P', and the function *next_cand* repeatedly to obtain the remaining patterns of P' in order of non-increasing length.

The pattern τ obtained from calling *first_cand* is used as the *current* candidate pattern τ. Each new sample string s is then compared to τ. If $s \in L(\tau)$, τ is output. Otherwise, *next_cand* is called to obtain a new candidate pattern τ'. Now, τ' is the current candidate pattern and output, no matter whether or not $s \in L(\tau')$. If the longest common prefix of all sample strings including the new string s is shorter than that of all sample strings excluding s, however, *first_cand* is called again and a new list of candidate patterns is considered. Thus, Algorithm 1LA may output *inconsistent* hypotheses.

Algorithm 1LA is shown in Figure 1. Let $S = \{s, s'\}$, and let $P' = P'(S)$ be defined as follows. If $s = vc$ for some $c \in \mathcal{A}$ and $s' = sw$ for some string w, then $P' = \{vx\}$. Otherwise, denote by u the longest common prefix of s and s', and let P' be the set of all patterns $\tau = u\varrho$ that can generate s and s' if we allow substitutions that replace different occurrences of x by different strings of the same length. The algorithm from Section 2 yields exactly P' if we omit the consistency check. Hence, $P' \supseteq Cons(S)$, where $Cons(S)$ is as defined in Section 2. Note that P' necessarily contains the pattern π if s and s' are in $L(\pi)$ and if the longest common prefix of s and s' is the same as the longest constant prefix of π. Furthermore, P' contains at most $O(|s| \log |s|)$ patterns. Assuming $P' = \{\pi_1, \ldots, \pi_t\}$, $|\pi_i| \geq |\pi_{i+1}|$ for $1 \leq i < t$, *first_cand*(s, s') returns π_1, and *next_cand*(s, s', π_i) returns π_{i+1}. Since we omit the consistency checks, a call to *first_cand* and all subsequent calls to *next_cand* until either the correct pattern is found or the prefix changes can be performed in time $O(|s|^2 \log |s|)$.

```
{ Phase 1 }
r ← 0;
repeat r ← r + 1;  read string s_r;
   if s_1 = s_2 = ··· = s_r then output hypothesis s_1 else output hypothesis x fi
until |s_r| < r;

{ Phase 2 }
while s_1 = s_2 = ··· = s_r do r ← r + 1;  read string s_r;
   if s_r = s_1 then output hypothesis s_1 else output hypothesis x fi
od;

{ Phase 3 }
s ← a shortest string in {s_1, s_2, ..., s_r};
u ← maximum length common prefix of {s_1, s_2, ..., s_r};
if u = s then s' ← a string in {s_1, s_2, ..., s_r} that is longer than s
else s' ← a string in {s_1, s_2, ..., s_r} that differs from s in position |u| + 1 fi
τ ← first_cand(s, s');
forever do
   read string s'';
   if u is not a prefix of s'' then
      u ← maximum length common prefix of s and s'';
      s' ← s'';
      τ ← first_cand(s, s')
   else if s'' ∉ L(τ) then τ ← next_cand(s, s', τ) fi;
   output hypothesis τ;
od
```

Figure 1: Algorithm 1LA

The following theorem summarizes the main result of this section. Note that its proof requires a series of lemmata which had to be omitted here because of the page limit.

THEOREM 3. *Let π be a 1-variable pattern. Algorithm 1LA correctly infers π from text in the limit. Furthermore, if the sample strings are drawn from $L(\pi)$ according to a proper probability distribution with expected string length ℓ then Algorithm 1LA correctly infers π from random text with probability 1, and the expected total learning time is $O(\ell^2 \log \ell)$.*

5. Learning with Superset Queries

Angluin [3] showed that PAT is not learnable with polynomially many queries if only equivalence, membership, and subset queries are allowed provided range($\mathcal{H}$) = PAT. This result may be easily extended to PAT_1. However, positive results are also known. First, PAT is exactly learnable using polynomially many disjointness queries with respect to the hypothesis space $PAT \cup FIN$, where FIN is the set of all finite languages (cf. [11]). The proof technique easily extends to PAT_1, too. Second, Angluin [3] established an algorithm exactly learning PAT with respect to PAT by asking polynomially many *superset queries*. However, it requires choosing general patterns τ for asking the queries, and does definitely not work if the hypothesis space is PAT_1. Hence, the following question arises.

Does there exist a superset query algorithm learning PAT_1 with respect to PAT_1 that uses only polynomially many superset queries?

Using the results of previous sections, we are able to answer this question affirmatively.

Nevertheless, whereas PAT can be learned with respect to PAT by *restricted* superset queries, i.e., superset queries not returning counterexamples, our query algorithm needs counterexamples. But it does not need a counterexample for every query answered negatively, instead *two* counterexamples always suffice. First, we show that 1-variable patterns are not learnable by asking polynomially many restricted superset queries.

THEOREM 4. *Any algorithm exactly identifying all $L \in PAT_1$ generated by a pattern π of length n with respect to PAT_1 by using only restricted superset queries and restricted equivalence queries must make at least $|\mathcal{A}|^{n-2} \geq 2^{n-2}$ queries in the worst case.*

Furthermore, we can show that learning PAT_1 with a polynomial number of superset queries is impossible if the algorithm may ask for a single counterexample only.

THEOREM 5. *Any algorithm that exactly identifies all 1-variable pattern languages by restricted superset queries and one unrestricted superset query needs at least $2^{(k-1)/4} - 1$ queries in the worst case, where k is the length of the counterexample returned.*

```
if L(π) ⊆ L(0) then τ ← 0
else i ← 1;
    while L(π) ⊆ L(C(0)^{≤i}x) do i ← i + 1 od;
    if L(π) ⊆ L(C(0)) then τ ← C(0)
    else S ← {C(0), C(C(0)^{≤i}x)}; R ← Cons(S);
        do τ ← max(R); R ← R \ {τ} until L(π) ⊆ L(τ)
    fi
fi; return τ
```

Figure 2: Algorithm QL. This algorithm learns a pattern π by superset queries. The queries have the form "$L(\pi) \subseteq L(\tau)$," where $\tau \in Pat_1$ is chosen by the algorithm. If the answer to "$L(\pi) \subseteq L(\tau)$" is *no*, QL can ask for a counterexample $C(\tau)$. By $w^{\leq i}$ we denote the prefix of w of length i and by $\max(R)$ some maximum–length element of R.

The new algorithm QL works as follows (cf. Figure 2). Assume it should learn some pattern π. QL asks whether $L(\pi) \subseteq L(0) = \{0\}$. This is the case iff $\pi = 0$, and if the answer is *yes* QL knows the right result. Otherwise, QL obtains a counterexample $C(0) \in L(\pi)$. By asking $L(\pi) \subseteq L(C(0)^{\leq j}x)$ for $j = 1, 2, 3, \ldots$ until the answer is *no*, QL computes $i = \min\{\, j \mid L(\pi) \not\subseteq L(C(0)^{\leq j}x) \,\}$. Now we know that π starts with $C(0)^{\leq i-1}$, but what about the i-th position of π? If $|\pi| = i-1$, then $\pi \in \mathcal{A}^+$ and therefore $\pi = C(0)$. QL asks $L(\pi) \subseteq L(C(0))$ to determine if this is the case. If $L(\pi) \not\subseteq L(C(0))$, then $\#_x(\pi) \geq 1$ and $\pi(i) \notin \mathcal{A}$, since this would imply that $\pi(i) = C(0)(i)$ and, thus, $L(\pi) \subseteq L(C(0)^{\leq i}x)$, a contradiction. Hence, $\pi(i) = x$. Now QL uses the counterexample for the query $L(\pi) \subseteq L(C(0)^{\leq i}x)$ to construct a set $S = \{C(0), C(C(0)^{\leq i}x)\}$. By construction, the two counterexamples differ in their i-th position, but coincide in their first $i-1$ positions.

Algorithm 2 on page 6 computes $R = Cons(S)$. Since $S \subseteq L(\pi)$ and since π coincides with S in the first $i-1$ positions, $\pi \in R$. Again we narrowed the search for π to a set R of candidates. Let m be the length of the shortest counterexample in S. Then $|R| = O(m \log m)$ by Lemma 5. Now, the only task left is to find π among all patterns in R. We find π by removing other patterns from R by using the following lemma.

LEMMA 8. *Let $S \subseteq \mathcal{A}^+$ and $\pi, \tau \in Cons(S)$, $|\pi| \leq |\tau|$. If $L(\pi) \subseteq L(\tau)$ then $\pi = \tau$.*

QL tests $L(\pi) \subseteq L(\tau)$ for a maximum length pattern $\tau \in R$ and removes τ from R if $L(\pi) \not\subseteq L(\tau)$. Iterating this process finally yields the longest pattern τ for which $L(\pi) \subseteq L(\tau)$. Lemma 8 guarantees $\tau = \pi$. Thus, we have:

THEOREM 6. *Algorithm QL learns PAT_1 with respect to PAT_1 by asking only superset queries. The query complexity of QL is $O(|\pi| + m \log m)$ many restricted superset queries plus two superset queries (these are the first two queries answered no) for every language $L(\pi) \in PAT_1$, where m is the length of the shortest counterexample returned.*

References

[1] D. Angluin. Finding patterns common to a set of strings. *J. Comp. Syst. Sci.*, 21:46–62, 1980.

[2] D. Angluin. Inductive inference of formal languages from positive data. *Inf. Control*, 45:117–135, 1980.

[3] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[4] G. Filé. The relation of two patterns with comparable languages. In *Proc. 5th Ann. Symp. Theoretical Aspects of Computer Science*, LNCS 294, pp. 184–192, Berlin, 1988.

[5] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Ann. ACM Symp. on Theory of Computing*, pp. 114–118, New York, 1978. ACM, ACM Press.

[6] E. M. Gold. Language identification in the limit. *Inf. Control*, 10:447–474, 1967.

[7] J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley, Reading, Mass., 1992.

[8] K. P. Jantke. Polynomial time inference of general pattern languages. In *Proc. Symp. Theoretical Aspects of Computer Science*, LNCS 166, pp. 314–325, Berlin, 1984.

[9] T. Jiang, A. Salomaa, K. Salomaa, and S. Yu. Inclusion is undecidable for pattern languages. In *Proc. 20th Int. Colloquium on Automata, Languages and Programming*, LNCS 700, pp. 301–312, Berlin, 1993.

[10] K.-I. Ko and C.-M. Hua. A note on the two-variable pattern-finding problem. *J. Comp. Syst. Sci.*, 34:75–86, 1987.

[11] S. Lange and R. Wiehagen. Polynomial-time inference of arbitrary pattern languages. *New Generation Computing*, 8:361–370, 1991.

[12] S. Lange and T. Zeugmann. Set-driven and rearrangement-independent learning of recursive languages. *Mathematical Systems Theory*, 29:599–634, 1996.

[13] D. Osherson, M. Stob and S. Weinstein. *Systems that learn, An introduction to learning theory for cognitive and computer scientists.* MIT Press, Cambridge, Mass., 1986

[14] A. Salomaa. Patterns. EATCS Bulletin 54:46 – 62, 1994.

[15] A. Salomaa. Return to patterns. EATCS Bulletin 55:144 – 157, 1994.

[16] T. Shinohara and S. Arikawa. Pattern inference. In *Algorithmic Learning for Knowledge-Based Systems*, LNAI 961, pp. 259 – 291, Berlin, 1995.

[17] K. Wexler and P. Culicover. *Formal Principles of Language Acquisition.* MIT Press, Cambridge, Mass., 1980.

[18] T. Zeugmann. Lange and Wiehagen's pattern language learning algorithm: An average-case analysis with respect to its total learning time. *Annals of Mathematics and Artificial Intelligence*, 1996. to appear.

[19] T. Zeugmann and S. Lange. A guided tour across the boundaries of learning recursive languages. In *Algorithmic Learning for Knowledge-Based Systems*, LNAI 961, pp. 190 – 258, Berlin, 1995.