DOI-TR-128

DOI Technical Report

Efficient Learning of One-Variable Pattern Languages from Positive Data

by

THOMAS ERLEBACH, PETER ROSSMANITH,
HANS STADTHERR, ANGELIKA STEGER,
AND THOMAS ZEUGMANN

December 12, 1996

Department of Informatics
Kyushu University
Fukuoka 812-81, Japan

Email: thomas@i.kyushu-u.ac.jp Phone: +81-92-642-2688

Efficient Learning of One-Variable Pattern Languages from Positive Data

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
Department 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 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

Patterns are a very natural way to define formal languages. Suppose you are interested in the language of all strings over the alphabet $\mathcal{A} = \{0, 1\}$ starting with 11, ending with 010, and containing the substring 01011 somewhere, but may be otherwise arbitrary. Thus, all strings in your language follow the pattern $\pi_1 = 11x_001011x_1010$, provided you are willing to allow the variables x_0, x_1 to be substituted by any string over $\{0, 1\}$ including the empty one. As for another example, consider the set of all strings having even length $2n$ such that the prefix of length n is identical to the suffix starting at position $n + 1$. In that case, the wanted language follows the pattern $\pi_2 = x_0x_0$.

Though patterns have already been considered since the beginning of this century (cf., e.g., Thue [28], and Bean *et al.* [5]), the formal introduction of patterns and pattern languages goes back to Angluin [2]. Since then, pattern languages and variations thereof have been widely investigated (cf., e.g., Salomaa [21, 22], and Shinohara and Arikawa [26] for an overview).

This continuous interest in pattern languages has several reasons, among them the learnability in the limit of the class of all pattern languages from positive data (cf. Angluin [2, 3]). Gold [8] introduced the corresponding learning model. Let L be any formal language; then a *text* for L is any infinite sequence of strings that contains eventually all the strings of L, and nothing else. The information given to the learner are successively growing initial segments of a text. Processing these segments, the learner has to output *hypotheses* about the target language L. Thereby the hypotheses are chosen from a prespecified set of *admissible hypotheses* called *hypothesis space*. Additionally, the sequence of hypotheses has to *converge* to a hypothesis correctly describing the target language (cf. Definition 1).

For pattern languages, the relevant hypothesis space is the set of all patterns. In particular, Angluin [2] showed the pattern languages to be learnable in the limit by outputting so-called descriptive patterns (see below) as hypotheses. This approach has several advantages, since the resulting hypotheses are *consistent*, and the resulting learning algorithm is *set-driven*. Here consistency means that the hypothesis correctly and completely reflects the information provided so far. A learner is said to be set-driven provided its output depends exclusively on the range of its input. In general, consistency and set-drivenness considerably limit the learning capabilities (cf., e.g., Zeugmann and Lange [34]). On the other hand, this approach has also a major disadvantage, since no efficient algorithm is known for computing descriptive patterns, and finding a descriptive pattern of maximum possible length is known to be $\mathcal{NP}$-complete. Therefore, it is unlikely that there exists a polynomial-time algorithm for computing a descriptive pattern as hypothesis.

Consequently, it is only natural to ask whether efficient learning algorithms for pattern languages can benefit from the concept of descriptive patterns at all. For answering this question, several authors looked at special cases. For example, one can add the requirement that every variable appears at most once in the pattern. Such patterns are called *regular*, since the languages they generate are regular. For example, pattern π_1 above is regular while pattern π_2 is not. Computing descriptive patterns for regular pattern languages can be done in polynomial time (cf. Shinohara [24, 25]). Further examples comprise the *non-cross* pattern languages as well as the class of unions of at most k regular pattern languages, where k is *a priori* fixed (cf., e.g., [26]). Another natural restriction is obtained by *a priori* bounding the number k of different variables that are allowed to occur in a pattern. Such patterns are called k-variable patterns. For example, π_1 above is a 2-variable pattern while π_2 is a one-variable pattern. However, to our knowledge it is still unknown whether polynomial-time algorithms exist that compute descriptive k-variable patterns for any fixed $k > 1$. (Compare with [11, 16, 12].)

Therefore, Lange and Wiehagen [17] have considered learners that are allowed to output inconsistent guesses until enough shortest strings of the target pattern language have been provided. Their algorithm achieves polynomial update time. Moreover, it is still set-driven (cf. [33]), and outperforms Angluin's [2] algorithm with respect to its storage requirements, since it is *iterative.* That is, it is computing its current guess from its previously made one, and the next input string.

Nevertheless, what really counts in practical applications is not just the time for computing a single guess, but the overall time needed by the learner until convergence. We refer to this time as the *total learning time.* On the one hand, it is easy to show that the total learning time is unbounded in the worst case, while in the best case $\lfloor \log_{|\mathcal{A}|}(|\mathcal{A}|+k-1)\rfloor+1$ many strings are sufficient to achieve convergence (here k denotes the number of different variables in the target pattern, and $|\mathcal{A}|$ the alphabet size) (cf. Zeugmann [33]). Therefore, we are mainly interested in the *average* or *expected* total learning time. Unfortunately, the expected total learning time of Lange and Wiehagen's [17] algorithm was shown to be exponential in the number of different variables occurring in the target pattern (cf. [33]).

Throughout the present paper we consider the special case of one-variable target patterns. In the case of one-variable pattern languages, a polynomial-time algorithm for computing descriptive patterns has been known for a long time. It has running time $O(n^4 \log n)$ for inputs of size n and was presented by Angluin [2]. She was aware of possible improvements of the running time only for certain special cases, and she hoped that further study would provide insight for a uniform improvement. We obtain such a uniform improvement by presenting an algorithm that computes a descriptive one-variable pattern in only $O(n^2 \log n)$ steps (Section 2). In addition, we show that our algorithm can be parallelized efficiently, using $O(\log n)$ time and $O(n^3/\log n)$ processors on an EREW-PRAM (Section 3). Previously, it was not known whether there exist efficient parallel algorithms for learning one-variable pattern languages.

Moreover, we use a modified version of the sequential algorithm as a subroutine in order to devise a learning algorithm for one-variable patterns whose expected total learning time is $O(\ell^2 \log \ell)$ if 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 (cf. Section 4).

Finally, we turn our attention to *active* learning. In contrast to the model described above, now the learner gains information concerning the target object by *asking questions* to an oracle (cf. Definition 3). In particular, 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, i.e., by asking questions of the form "$L(\tau) \supseteq L(\pi)$?," where π is the pattern to be learned and τ is an arbitrary one-variable pattern. The number of queries asked by our algorithm is polynomial (cf. Section 5). A different algorithm that learns arbitrary pattern languages with a polynomial number of superset queries is known (cf. [4]), but that algorithm uses patterns with more than one variable in its queries even when the pattern to be learned is a one-variable pattern.

Finishing this section, we would like to mention that learning of pattern languages has also been investigated in Valiant's [29] *probably approximately correct* (PAC) learning model. Here, the learner has access to a random source of positive and negative examples and must produce in polynomial time with high probability a pattern which is consistent with nearly all future positive and negative examples from the source. Schapire [23] has shown that it is highly unlikely that general pattern languages can be learned in this model. On the other hand, Kearns and Pitt have shown that under certain assumptions k-variable pattern languages can be learned in this model for any fixed k (cf. [15]).

The paper is structured as follows. Subsection 1.1 formally defines the pattern languages and some additional preliminaries. The different learning models considered in this paper are introduced in Subsection 1.2. Next, we present our improved sequential algorithm for finding one-variable descriptive patterns (cf. Section 2). Its efficient parallelization is outlined in Section 3. In Section 4 we provide the announced average-case analysis concerning the expected total learning time of our resulting learning algorithm. Finally, we study the learnability of one-variable pattern languages from superset queries (cf. Section 5), and discuss the results obtained (cf. Section 6).

1.1. *The Pattern Languages*

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 $\lfloor x \rfloor$, the *floor function*, to be the greatest integer less than or equal to x.

Patterns and pattern languages are defined as follows (cf. [2]). Let $\mathcal{A} = \{0, 1, \ldots\}$ be any non-empty finite alphabet containing at least two elements. By $\mathcal{A}^*$ we denote the free monoid over $\mathcal{A}$ (cf. Hopcroft and Ullman [9]). The set of all finite non-null strings of symbols from $\mathcal{A}$ is denoted by $\mathcal{A}^+$, i.e., $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$, where ε denotes the empty string. By $|\mathcal{A}|$ we denote the cardinality of $\mathcal{A}$. Furthermore, let $X = \{x_i \mid i \in \mathbb{N}\}$ be an infinite set of variables such that $\mathcal{A} \cap X = \emptyset$. *Patterns* are non-empty strings over $\mathcal{A} \cup X$, e.g., 01, $0x_0111$, $1x_0x_00x_1x_2x_0$ 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 $\pi \in Pat$, $1 \le i \le |\pi|$; we use $\pi(i)$ to denote the i-th symbol in π, e.g., $0x_0111(2) = x_0$, and $0x_0111(5) = 1$. If $\pi(i) \in \mathcal{A}$, then we refer to $\pi(i)$ as a *constant*; otherwise $\pi(i) \in X$, and we refer to $\pi(i)$ as a *variable*. Analogously, we use $s(i)$ to denote the i-th symbol in a string $s \in \mathcal{A}^+$ for all $i = 1, \ldots, |s|$. By $\#\mathrm{var}(\pi)$ we denote the number of different variables occurring in π, and by $\#_{x_i}(\pi)$ we denote the number of occurrences of variable x_i in π. If $\#\mathrm{var}(\pi) = k$, then we refer to π as a *k-variable pattern*. Let $k \in \mathbb{N}$, by Pat_k we denote the set of all *k-variable patterns*. In the case $k = 1$ we conventionally denote the variable occurring by x, i.e., we omit its index. Furthermore, if $\#_{x_i}(\pi) = 1$ for all variables occurring in pattern π, then we call π a *regular* pattern.

Now let $\pi \in Pat_k$, and let $u_0, \ldots, u_{k-1} \in \mathcal{A}^+$. We denote by $\pi[u_0/x_0, \ldots, u_{k-1}/x_{k-1}]$ the string $s \in \mathcal{A}^+$ obtained by substituting u_j for each occurrence of x_j, $j = 0, \ldots, k-1$, in the pattern π. The tuple $(u_0, \ldots, u_{k-1})$ is called *substitution.* For every $\pi \in Pat_k$ 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.* Finally, $PAT = \bigcup_{k\in\mathbb{N}} PAT_k$ denotes the set of all pattern languages over $\mathcal{A}$.

From the viewpoint of formal language theory the class of pattern languages has several interesting properties. First, it is incomparable with the class of regular languages and the class of context-free languages (cf. [2]). Second, it is not closed under union, complement, intersection, Kleene closure, homomorphism, or inverse homomorphism. But it is closed under concatenation and reversal. Furthermore, it is one of the rare examples of a class of languages where the equivalence problem is easily decidable but the inclusion problem is undecidable (cf. [13]). One reason why one-variable pattern languages are more interesting for practical purposes than general pattern languages is that several problems are decidable or even efficiently solvable for one-variable pattern languages but undecidable or $\mathcal{NP}$–complete for general pattern languages. For example, in the case of general pattern languages the word problem is $\mathcal{NP}$–complete (cf. [2]) and the inclusion problem is undecidable (cf. [13]), but both problems are decidable in linear time for one-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 one-variable patterns and their languages.

A finite set $S = \{s_0, s_1, \ldots, s_r\} \subseteq \mathcal{A}^+$ of strings is called a *sample.* A pattern π is *consistent* with a sample S if $S \subseteq L(\pi)$. In order to formalize the notion of when a pattern is a concise description of a given sample S, a (one-variable) pattern π is called *descriptive* for S if it is consistent with S and there is no other consistent (one-variable) pattern τ such that $L(\tau) \subset L(\pi)$. Angluin [2] showed that any consistent one-variable pattern of maximum length is descriptive, except for the trivial case when the sample consists of a single string only.

1.2. *Learning and Inductive Inference*

Pattern languages have been introduced originally in the context of inductive inference. Research on inductive inference is concerned with formalizing and analyzing the process of gradually learning concepts from successively larger sets of examples. Frequently, the concepts are taken to be formal languages (cf., e.g., Osherson *et al.* [20], Zeugmann and Lange [34]). In this case, a distinction is made between learning from *informant* and learning from *text.* If L is the language to be identified, every sequence $i = (s_0, b_0), (s_1, b_1), (s_2, b_2), \ldots$ with $b_j \in \{+, -\}$ for all $j \in \mathbb{N}$ satisfying $\{s_j \mid j \in \mathbb{N}\} = \mathcal{A}^*$, $(s_j, +) \in i \Rightarrow s_j \in L$, and $(s_j, -) \in i \Rightarrow s_j \notin L$ is said to be an informant for L. That is, every string over the relevant alphabet is classified as to whether it belongs to L or not. Furthermore, every infinite sequence $t = (s_j)_{j\in\mathbb{N}}$ of strings such that

[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 [6].

range$(t) = \{s_j \mid j \in \mathbb{N}\} = L$ is said to be a text for L, or, synonymously, a *positive presentation.* By $Text(L)$ we denote the set of all texts for L. Furthermore, let t be a text, and let $n \in \mathbb{N}$. We set $t_n = s_0, \dots, s_n$, and we refer to t_n as the initial segment of t of length $n+1$. Moreover, we define t_n^+ to denote the range of t_n, i.e., $t_n^+ = \{s_j \mid 0 \le i \le n\}$.

As in Gold [8], we define an *inductive inference machine* (abbr. IIM) to be an algorithmic device which works as follows: The IIM takes as its input larger and larger initial segments of a text t and on every input it first outputs a hypothesis, i.e., a pattern, and then it requests the next input.

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 $n \in \mathbb{N}$, $M(t_n) = \pi$.*

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

The pattern languages as well as the class of all one-variable pattern languages constitute an *indexable class* $\mathcal{L}$ of uniformly recursive languages. That is, there are an effective enumeration L_0, L_1, $L_2, \dots$ of all and only the languages in $\mathcal{L}$ and a recursive function f such that for all $j \in \mathbb{N}$ and all strings $s \in \mathcal{A}^*$ we have

$$f(j,s) = \begin{cases} 1, & \text{if} \quad s \in L_j, \\ 0, & \text{otherwise.} \end{cases}$$

In the following we refer to indexable classes with uniformly decidable membership as indexable classes for short. Note that Definition 1 is a bit sharper than the usual definition of learnability in the limit. The point here is that in our definition the IIM is requested to output on every input a pattern as hypothesis, while in the general case, the IIM is allowed to request the next input without making any guess. IIMs behaving thus are called *responsive.* Throughout this paper, we exclusively deal with responsive IIMs.

Angluin [3] gave sufficient and necessary conditions for indexable classes of languages being inferable from text (not necessarily responsively). In particular, she proved that an inference machine that always outputs as its hypothesis a pattern which is descriptive for the input read so far responsively infers all pattern languages (cf. [2]). This approach has another advantage, since it results in a *set-driven* learner. Set-drivenness has been introduced by Wexler and Culicover [30], and is defined as follows.

DEFINITION 2. *Let $\mathcal{L}$ be any indexable class. An IIM is said to be set-driven with respect to $\mathcal{L}$ iff its output depends only on the range of its input; that is, iff $M(t_x) = M(\hat{t}_y)$ for all $x, y \in \mathbb{N}$ and all texts t, $\hat{t} \in \bigcup_{L \in \mathcal{L}}$ Text(L) provided $t_x^+ = \hat{t}_y^+$.*

Note that in general one cannot expect to learn set-drivenly. For more information concerning this subject the reader is referred to Lange and Zeugmann [18].

When dealing with set-driven learners, it is technically advantageous to describe them in dependence of the relevant set t_n^+ obtained as input instead of the initial segments t_n

usually fed to an IIM. We refer to these sets as *samples* in accordance with the definition made above. Now, let $t = s_0, s_1, \dots$ be any text, and let $m \in \mathbb{N}$. We set $n = \sum_{j=0}^{m} |s_j|$, and refer to n as the *input length* of the initial segment t_m. Analogously, let $S = \{s_0, s_1, \dots, s_r\}$ be a sample. Then we set $n = \sum_{j=0}^{r} |s_j|$, and refer to n as the *input length* of sample S.

Furthermore, we mainly deal with the time complexity of the pattern learning algorithms considered throughout this paper. Except in Chapter 3, where we use the PRAM-model, we assume the same model of computation and the same representation of patterns as Angluin [2], i.e., in particular a random access machine that performs a reasonable menu of operations each in unit time on registers of length $O(\log n)$ bits, where n is the input length. The inputs are read via a serial input device, and reading a string of length n is assumed to require n steps.

Moreover, we aim at both determining the update time and the total learning time which we define next. Let M be any IIM. Then, for every $L \in PAT$ and $t \in Text(L)$, let

$$Conv(M,t) = \text{the least number } m \text{ such that for all } n \geq m,\ M(t_n) = M(t_m)$$

denote the *stage of convergence* of M on t. Moreover, 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 describe the model of *active* learning mentioned in the introduction we want to deal with, i.e., *learning via queries*. The objects to be learned are the elements of a prespecified indexable class $\mathcal{L}$ defined over some alphabet $\mathcal{A}$. Additionally, we assume an indexable class $\mathcal{H}$ again defined over $\mathcal{A}$ as well as a fixed effective enumeration $(h_j)_{j\in\mathbb{N}}$ of it as hypothesis space for $\mathcal{L}$. Clearly, $\mathcal{H}$ must *comprise* $\mathcal{L}$. A hypothesis h is said to describe a target language L iff $L = h$, i.e., for all $s \in \mathcal{A}^*$, $s \in h$ iff $s \in L$. The source of information about the unknown target L are queries to an *oracle*. Following Angluin [4] we distinguish *membership, equivalence, subset, superset*, and *disjointness* queries. Input to a membership query is a string s, and the output is *yes* if $s \in L$ and *no* otherwise. As 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 answer is no, additionally a *counterexample* is returned, 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. Throughout this paper we always assume that all queries are *answered truthfully*.

Definition 3 (Angluin [4]). *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 $L = h_j$.*

Besides the source of information, there is another major difference to Definition 1. That is, now the learner is allowed only *one guess*, and that hypothesis must be correct.

The *complexity* of a query learner is measured by the total number of queries to be asked in the worst-case.

2. An Improved Sequential Algorithm

In this section we present an algorithm that computes a descriptive one-variable pattern for a given sample. The input to the algorithm is a sample S, $S = \{s_0, s_1, \dots, s_{r-1}\}$, of r non-empty strings over $\mathcal{A}$. 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 the original algorithm by Angluin [2], which needs time $O(n^2 |s_0|^2 \log |s_0|)$. Recall that n denotes the input length of S.

Angluin's [2] algorithm computes explicitly a representation of the set of all consistent one-variable patterns for S. This requires a tricky data structure which can represent exponentially many patterns in polynomial space. In order to get a faster algorithm we avoid to represent all consistent patterns, but rather work with a polynomial-size subset thereof.

Before we describe our algorithm in detail, we need to review and to establish a few basic properties of one-variable patterns and the languages generated by them.

LEMMA 1 (Angluin [2], 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 $\alpha_0, \dots, \alpha_{r-1} \in \mathcal{A}^+$ such that s_i can be obtained from π by substituting α_i for x, for all $0 \le i \le r-1$. Given a consistent pattern π, the set $\{\alpha_0, \dots, \alpha_{r-1}\}$ 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 $\alpha(S, \pi)$ start with a different symbol.*

Proof. Let u denote the longest common prefix of all strings in S. The pattern ux is consistent with S because u is shorter than every string in S, since S is prefix-free. Consequently, there exists a descriptive pattern $\pi \in Pat_1$ for S with $L(\pi) \subseteq L(ux)$. Now, by Lemma 1 we know that there exists a pattern $\varrho \in Pat_1$ such that $\pi = ux[\varrho/x]$. Since u is a longest common prefix of all strings in S, we can conclude $\varrho(1) \notin \mathcal{A}$. Hence, $\varrho = x\tau$ for some $\tau \in Pat_1 \cup \{\varepsilon\}$, and at least two strings in $\alpha(S, ux\tau)$ must start with a different symbol. ∎

Let $Cons(S) = \{\pi \mid \pi \in Pat_1,\ S \subseteq L(\pi),\ \exists i,j[i \neq j,\ \alpha_i,\ \alpha_j \in \alpha(S,\pi),\ \alpha_i(1) \neq \alpha_j(1)]\}$, i.e., $Cons(S)$ is the set of all consistent patterns π for S such that at least two strings in $\alpha(S, \pi)$ start with a different symbol. Note that $Cons(S)$ is in general only a subset of all patterns that are consistent with S, and that $Cons(S) = \emptyset$ provided 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.*

Proof. Let S be prefix-free. According to Lemma 2 there exists a descriptive pattern for S belonging to $Cons(S)$; thus $Cons(S) \neq \emptyset$. Now, suppose there is a pattern $\pi \in Cons(S)$ of maximum length which is not descriptive for S. Thus, $S \subseteq L(\pi)$, and, moreover, there exists a pattern $\tau \in Pat_1$ such that $S \subseteq L(\tau)$ as well as $L(\tau) \subset L(\pi)$. Hence, by Lemma 1 we know that τ can be obtained from π by substituting a pattern ϱ for x. Since at least two strings in $\alpha(S,\pi)$ start with a different symbol, we immediately get $\varrho(1) \in X$. Moreover, at least two strings in $\alpha(S,\tau)$ must also start with a different symbol. Hence $\tau \in Cons(S)$ and $\varrho = x\nu$ for some pattern ν. Note that $|\nu| \geq 1$, since otherwise $\tau = \pi[\varrho/x] = \pi$ contradicting $L(\tau) \subset L(\pi)$. Finally, by $|\nu| \geq 1$, we may conclude $|\tau| > |\pi|$, a contradiction to π having maximum length. Thus, no such pattern τ can exist, and hence, π is descriptive. ∎

Note that the lemma above does heavily depend on the restriction to one-variable patterns. For example, let $S = \{111000, 000111\}$. Obviously, S is prefix-free, and there is only one consistent pattern $\pi \in Pat_1$, i.e., x. On the other hand, if we drop the restriction to exclusively consider elements from Pat_1, we may take $x_0x_0x_0x_1x_1x_1$ as a descriptive pattern.

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

In all other cases $|S|$ contains at least two strings and s_0 is not a prefix of all other strings in the sample. Hence, $|S|$ is prefix-free and Lemma 3 applies. Therefore, it is sufficient to find and output a longest pattern in $Cons(S)$. In the rest of this section we will show how this can be done efficiently.

The observation that leads to an improved algorithm for prefix-free samples is that the number of patterns in $Cons(S)$ is bounded by a small polynomial, as we show next. Let $k, l \in \mathbb{N}$, $k > 0$; we call patterns with k occurrences of x and l occurrences of constants (k,l)-patterns. Note that a (k,l)-pattern has length $k + l$. Furthermore, every pattern $\pi \in Cons(S)$ satisfies $|\pi| \leq |s_0|$, since we exclusively consider non-erasing substitutions. Obviously, there can only be a (k,l)-pattern in $Cons(S)$ if there is a positive integer m_0 satisfying $|s_0| = km_0 + l$. Clearly, 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. Thus, for $m_0 = 1$ there are $|s_0|$ many possible pairs (k,l) such that $k + l = |s_0|$. For $m_0 = 2$, the number l must satisfy $0 \leq l = |s_0| - 2k$, hence there are at most $\lfloor |s_0|/2 \rfloor$ many possible pairs (k,l) with $|s_0| = 2k + l$, and so on. 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 can exist in $Cons(S)$ 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 the strings $\alpha_i \in \alpha(S,\pi)$ must satisfy $m_i = (|s_i| - l)/k$. In addition, it is clear that α_i is simply the substring of s_i of length m_i starting at the first position d where the input strings differ. If $(|s_i| - l)/k$ is not integral for some i, then there is no consistent (k,l)-pattern and the algorithm does not need to perform any further computation for this particular (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 $S = \{s_0, \dots, s_{r-1}\}$ be any prefix-free sample. For every given (k,l)-pair, there is at most one (k,l)-pattern in $Cons(S)$.*

Proof. Let u be the longest common prefix of the strings in S, possibly $u = \varepsilon$. All patterns in $Cons(S)$ start with ux. Thus, if $|u| > l$ there cannot be any (k,l)-pattern in $Cons(S)$. Now, assume $|u| \le l$. If $(|s_i| - l)/k$ is not integral for some i, there is again no (k,l)-pattern in $Cons(S)$. Otherwise, we show by induction on j that the j-th symbol in a (k,l)-pattern $\pi \in Cons(S)$ is unique, or there is no such pattern π. Note that all α_i and m_i are already determined by the (k,l)-pair.

Obviously, the hypothesis is true for $1 \le j \le |u| + 1$. Let us assume that the first $j-1$ symbols of π are already fixed and contain $b \le k$ variables and $c \le l$ constants so far. For each i, the prefix of s_i corresponding to the first $j-1$ symbols of π has length $bm_i + j - 1 - b$. If all strings s_i have the same symbol $a \in \mathcal{A}$ in position $bm_i + j - b$, the j-th symbol of π must be a. In that case, the candidate pattern is extended by a, and $c := c+1$. If all strings s_i have the symbol $\alpha_i(1)$ in position $bm_i + j - b$, the j-th symbol of π must be x, and another occurrence of x has been detected. Thus, the candidate pattern is extended by x, and $b := b+1$. If none of the two conditions holds, there is no (k,l)-pattern in $Cons(S)$. Otherwise, the candidate pattern π is uniquely determined.

Now, three cases are possible. First, $c > l$, i.e., more than l constants occur in the candidate pattern π, and therefore, no (k,l)-pattern in $Cons(S)$ can exist. Second, $b > k$, i.e., more than k variables appear in π. Consequently, there is again no (k,l)-pattern in $Cons(S)$. Finally, k variables and l constants appear in the candidate pattern π. Now, one can easily check whether π is consistent. If it is, $\pi \in Cons(S)$, and we are done. Otherwise, there is no (k,l)-pattern in $Cons(S)$. ∎

An algorithm computing the unique candidate π for the (k,l)-pattern in $Cons(S)$ follows straightforwardly from the proof above. For the sake of presentation, we omit the consistency test here. Additionally, Algorithm 1 does not test whether the numbers m_i are integral, since it is run later only for pairs (k,l) leading to integral values of the m_i. Furthermore, we assume a subprocedure taking as input a sample S, and returning the longest common prefix u as well as the first position d where the input strings differ. The following algorithm either returns a pattern π or *NIL*. If *NIL* is returned then there is no (k,l)-pattern in $Cons(S)$.

***Algorithm* 1**

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

for $j = 0, \dots, r-1$ **do** $m_j \leftarrow (|s_j| - l)/k$ **od**;

$\pi \leftarrow u$; $b \leftarrow 0$; $j \leftarrow d$; $c \leftarrow |u|$;
while $j \leq k + l$ **do**
  if $s_0(bm_0 + j - b) = s_i(bm_i + j - b)$ for all $1 \leq i \leq r - 1$ **then**
    if $c < l$ **then** $\pi \leftarrow \pi s_0(bm_0 + j - b)$; $c \leftarrow c + 1$
      else return *NIL*
    fi
  else if $b < k$ **then** $\pi \leftarrow \pi x$; $b \leftarrow b + 1$
      else return *NIL*
    fi
  fi;
  $j \leftarrow j + 1$
od;
return π

Note that minor modifications of Algorithm 1 perform the consistency test $S \subseteq L(\pi)$ even while π is constructed.

Example 1 Let $S = \{0001000010,$
$1011010110,$
$10010101001010,$
$1100101011001010,$
$110111110110111110\}$.

Then $|s_0| = 10$ and there are 27 possible (k, l)-pairs, i.e., $(1, 9)$, $(2, 8)$, $\ldots$, $(10, 0)$, $(1, 8)$, $\ldots$, $(5, 0)$, $(1, 7)$, $(2, 4)$, $(3, 1)$, $(1, 6)$, $(2, 2)$, $(1, 5)$, $(2, 0)$, $(1, 4)$, $\ldots$, $(1, 0)$. Next, checking for $|s_1|$, $|s_2|$, $|s_3|$, and $|s_4|$ whether or not $m_i = (|s_i| - l)/k$ is integral immediately rules out 12 of the listed (k, l)-pairs. Clearly, for S the longest common prefix is $u = \varepsilon$, and thus $d = 1$. Running Algorithm 1 on the remaining 15 pairs returns *NIL* for $(1, 9)$, $(1, 8)$, $(1, 7)$, $(1, 6)$, $(1, 5)$, $(1, 4)$, $(1, 3)$ as well as for $(2, 8)$ and $(2, 6)$. Thus, $Cons(S)$ contains only patterns for the pairs $(2, 4)$, $(2, 2)$, $(1, 2)$, $(2, 0)$, $(1, 1)$, and $(1, 0)$. These are $x10x10$, $x0x0$, $x10$, xx, $x0$, and x, respectively. The longest among these patterns is $x10x10$, and it is descriptive for S.

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

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 $k = 1, \ldots, |s_0|$ **do**

  for $m_0 = 1, \ldots, \lfloor \frac{|s_0|}{k} \rfloor$ **do**

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

  od

od;

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

It should be noted that the number of (k, l)-pairs for which the test has to be performed is actually smaller than $O(|s_0| \log |s_0|)$ for many inputs. This is because the requirement that $(|s_i| - l)/k$ is integral for all i restricts the set of possible values of k if not all strings have the same length. More precisely, only those values of k that are divisors of $|s_i| - |s_j|$ for all $0 \leq i, j \leq r-1$ must be considered.

Another remark concerns the order in which the (k, l)-pairs are processed. Since it suffices to output a consistent pattern having maximum length, it is practically helpful to process the (k, l)-pairs in order of non-increasing $k+l$. This ensures that the algorithm can terminate as soon as it encounters the first consistent pattern. The worst-case complexity is not improved, however, because all (k, l)-pairs have to be processed if the descriptive pattern is x.

3. An Efficient Parallel Algorithm

Whereas the RAM (*random access machine*) 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. Nevertheless, the PRAM (*parallel random access machine*) model, introduced by Fortune and Wyllie [7], is usually considered an acceptable compromise (cf., e.g., Zeugmann [32] for an overview).

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 (cf., e.g., JáJá [10]). The advantages of the PRAM model over competing parallel machine models include the simplicity with which PRAM algorithms can be described and analyzed, and the high level of abstraction this model offers. Its drawback is the somewhat unrealistic assumption of a globally shared memory which causes problems for the implementation of PRAM algorithms on existing parallel computers. It has been shown, however, that PRAM algorithms can be executed efficiently on a number of parallel architectures [1, 14].

Different variants of the PRAM model have been considered with respect to 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 (CREW stands for *concurrent read – exclusive write*). The various PRAM models, ranging from EREW to CRCW, have been shown to be equally powerful except for logarithmic factors in running time or processor requirements. For ease of presentation, we will 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. For any further information, we refer the reader to JáJá [10].

To our knowledge, no parallel algorithm for computing descriptive one-variable patterns has been known previously. We show how Algorithm 2 can be efficiently parallelized by using several well-known techniques including prefix-sums, tree-contraction, and list-ranking as subroutines (cf. [10]).

First, we observe that a parallel algorithm can deal with non-prefix-free samples S in the same way as the sequential algorithm. Checking whether S is singleton or whether s_0 is a prefix of all other strings can be easily done in time $O(\log n)$ using $O(n/\log n)$ processors. Therefore, without loss of generality we assume that these prefix-tests have been performed, and that the input sample $S = \{s_0, \ldots, s_{r-1}\}$ is prefix-free. In addition, we assume that the prefix-tests have returned the first position d where the input strings differ and an index t, $1 \leq t \leq r-1$, such that $s_0(d) \neq s_t(d)$.

Obviously, a parallel algorithm can deal with all $O(|s_0| \log |s_0|)$ possible (k,l)-pairs in parallel. For each (k,l)-pair, our parallel algorithm computes the unique (k,l)-pattern in $Cons(S)$, if it exists. Finally, it suffices to output any obtained pattern having maximum length. Next, we show how to efficiently parallelize the two steps to be performed for each (k,l)-pair, i.e., computing a candidate (k,l)-pattern π and checking whether $S \subseteq L(\pi)$. An illustrative example can be found at the end of this section.

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 $Cons(S)$. Considering only two strings reduces the processor requirements, and an easy modification of the proof of Lemma 4 shows that the candidate pattern remains unique even if only two strings that differ in position d are considered.

Position j_t in s_t is said to be b-corresponding to position j_0 in s_0 if $j_t = j_0 + b(m_t - m_0)$, where $0 \leq b \leq k$. Intuitively, the meaning of b-corresponding positions is as follows. Suppose there is a (k,l)-pattern π which is consistent with s_0 and s_t. Furthermore, assume that position j_0 in s_0 corresponds to a constant symbol in π, and that b occurrences of x are to the left of that constant symbol. Then that symbol corresponds to position $j_t = j_0 + b(m_t - m_0)$ in s_t. For example, let $\pi = 0x0x00x1$, $s_0 = \pi[0/x] = 00000001$, and $s_t = \pi[11111/x] = 01111101111100111111$. Thus, $m_0 = 1$ and $m_t = 5$. Hence position 20 in s_t is 3-corresponding to position 8 in s_0, since $8 + 3(5-1) = 20$.

In order to be able to compute the candidate pattern from s_0 and s_t, the algorithm calculates the entries of an auxiliary array EQUAL$[j,b]$ of Boolean values first, where j ranges from 1 to $|s_0|$ and b ranges from 0 to k. Intuitively, 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. Formally, the array is defined as follows:

$$\text{EQUAL}[j,b] = \text{true iff } s_0(j) = s_t(j + b(m_t - m_0))$$

The array EQUAL has $O(k|s_0|)$ entries, and each value can be calculated in constant time.

Using $O(k|s_0|/\log n)$ processors, the resulting time requirement is $O(\log n)$ for computing EQUAL. The following lemma shows how a candidate pattern can be obtained from EQUAL within the same time and processor bounds.

LEMMA 6. *Let $S = \{s_0, \ldots, s_{r-1}\}$ be a sample, and let n be 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.*

Proof. Sequentially, the candidate pattern can be obtained from EQUAL in a way similar to Algorithm 1 in Section 2, namely by starting at EQUAL$[1,0]$ with the empty pattern π and visiting entries of EQUAL according to the following rule: If EQUAL$[j,b]$ is true, append $s_0(j)$ to π and go to EQUAL$[j+1,b]$. Otherwise, append x to π and go to EQUAL$[j+m_0,b+1]$. This is iterated until at least one index lies outside the valid range. If the procedure terminates because it tries to visit EQUAL$[|s_0|+1,k]$, a candidate pattern has been obtained. Otherwise, there is no (k,l)-pattern in *Cons*(S), and *NIL* can be returned.

To perform this calculation efficiently in parallel, a directed graph $G=(V,E)$ is built from the array EQUAL. The set of nodes V consists of one node for each entry of EQUAL and one additional node, i.e., nodes $v_{j,b}$ for $1 \le j \le |s_0|$ and $0 \le b \le k$ and an additional node $v_{|s_0|+1,k}$. Note that $|V| = O(k|s_0|)$. The following arcs are added to E:

$$\begin{aligned} (v_{j,b}, v_{j+1,b}) \in E \quad &\text{iff} \quad \text{EQUAL}[j,b] = \text{true}, \\ (v_{j,b}, v_{j+m_0,b+1}) \in E \quad &\text{iff} \quad \text{EQUAL}[j,b] = \text{false} \end{aligned}$$

These are the only arcs in the graph G. Note that each node of G has outdegree at most 1 and indegree at most 2, and that G is acyclic. In other words, G is a forest of binary in-trees.

Now we observe that a candidate (k,l)-pattern exists if and only if there is a path in G from $v_{1,0}$ to $v_{|s_0|+1,k}$. In addition, this path can be used to determine the candidate pattern. In order to extract the path, the tree-contraction technique can be used. We view each in-tree of G as an arithmetic expression tree where every internal node represents a $+$ operation and every leaf node gets a 0, except for the leaf node $v_{1,0}$ which gets a 1. The expressions represented by all trees and subtrees of G can be evaluated simultaneously in time $O(\log n)$ using $O(k|s_0|/\log n)$ processors on an EREW-PRAM. At the end, precisely the nodes reachable from $v_{1,0}$ have the value 1, whereas all other nodes are 0. Hence, the path from $v_{1,0}$ to the root of its in-tree can easily be extracted using prefix-sums and list-ranking.

In addition, if the path does indeed end at $v_{|s_0|+1,k}$, the candidate (k,l)-pattern can be obtained directly from the path and the string s_0. This is because the length of the candidate pattern π is equal to the number of nodes on the path minus one, and because π can be computed from the path by replacing each of the first $|\pi|-1$ nodes by the appropriate symbol. More specifically, each node $v_{j,b}$ corresponding to a true entry EQUAL$[j,b]$ is replaced by the constant $s_0(j)$, and each node $v_{j,b}$ corresponding to a false entry is replaced by the variable x. ∎

At this stage, 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 needs to check whether π is indeed consistent with S.

LEMMA 7. *Given a candidate pattern π with k occurrences of x and l occurrences of constants, 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.*

Proof. Given the candidate pattern π, each symbol of a string $s_i \in S$ either corresponds to a constant in the pattern or to an occurrence of x (more specifically, it is the z-th symbol of an instantiation of x in s_i, for some z). We associate one bit with each symbol in each s_i, such that $S \subseteq L(\pi)$ if and only if all these bits are true. For a symbol corresponding to a constant in the pattern, the bit is set to true if the symbol is equal to that constant. For a symbol which is the z-th symbol of an instantiation of x in s_i, the bit is set to true if the symbol is equal to the $(d+z-1)$-th symbol of s_i, where d is the position of the first x in π. In other words, α_i is set to the substring of s_i that starts in position d and has length m_i, and it is checked whether this choice is consistent with all other occurrences of x in π. The Boolean AND of these n bits can then be computed in time $O(\log n)$ with $O(n/\log n)$ processors on an EREW-PRAM. The resulting value is true if and only if π is consistent with the whole sample S.

Now we show how the bits associated to all symbols in the sample can be calculated in the same time and processor bounds. In the following, the calculation is described for the symbols in one specific string s_i. First, the algorithm assigns a value to each symbol $\pi(j)$ of the pattern. If $j = 1$ or $\pi(j-1) \in \mathcal{A}$, the value 1 is assigned to $\pi(j)$. If $j > 1$ and $\pi(j-1) = x$, the value m_i is assigned to $\pi(j)$. A prefix-sums calculation over the values assigned to the symbols in the pattern yields, for each symbol in the pattern, the position of its corresponding symbol in s_i (for constant symbols) or the position of the first symbol of its corresponding instantiation in s_i (for variable symbols).

Next, a pair (j, z) is associated with each symbol in s_i and initialized to $(0, 0)$. For the j-th symbol $\pi(j)$ in the pattern $(1 \le j \le k + l)$, the $(0, 0)$-pair associated to its corresponding symbol in s_i is replaced by $(j, 0)$ if $\pi(j) \in \mathcal{A}$, and by $(j, 1)$ if $\pi(j) = x$. As a consequence, now the pair associated to a symbol in s_i is $(j, 0)$ if the symbol corresponds to $\pi(j) \in \mathcal{A}$ and $(j, 1)$ if the symbol starts the instantiation of $\pi(j) = x$. Symbols in the z-th position, $z > 1$, of an instantiation of x still have $(0, 0)$. The following operation on pairs of integers can easily be shown to be associative:

$$(j_1, z_1) \oplus (j_2, z_2) = \begin{cases} (j_2, z_2) & \text{if } j_2 \neq 0 \\ (j_1, z_1 + 1) & \text{if } j_2 = 0 \end{cases}$$

After a parallel prefix computation using the operation $\oplus$ on the pairs associated to the symbols in s_i, the pair associated to a symbol in the z-th position of an instantiation of $\pi(j) = x$ is (j, z). The desired bit values can then be computed in constant time per bit as follows: Assume that the result of the prefix computation is (j, z) for symbol $s_i(h)$. If $\pi(j) \in \mathcal{A}$, the bit associated to $s_i(h)$ is true if and only if $s_i(h) = \pi(j)$. If $\pi(j)$ is x, the bit associated to $s_i(h)$ is true if and only if $s_i(h) = s_i(d + z - 1)$.

For the prefix-sum computations over the values associated to symbols of the pattern time $O(\log n)$ and $O((k + l)/\log n) = O(|s_0|/\log n)$ processors are sufficient. The prefix-sum computations over the pairs associated to symbols of s_i can be performed in time

$O(\log n)$ using $O(|s_i|/\log n)$ many processors. Performing these computations in parallel for all i, $0 \le i \le r-1$, requires $O(\log n)$ time and $O(n/\log n)$ processors on an EREW-PRAM. A straightforward implementation of the computation of the bit values requires concurrent read access, however, because several processors may access the same symbol in π or in some s_i when comparisons are performed. ∎

Theorem 1. *There exists a parallel algorithm that computes descriptive one-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.*

Proof. If $|S| = 1$ or S is not prefix-free, $O(n/\log n)$ processors and $O(\log n)$ time clearly suffice. Otherwise, the number of processors required for calculating the array EQUAL and the candidate pattern in parallel for all possible (k, l)-pairs according to Lemma 6 can be bounded as follows:

$$\sum_{(k,l)\text{-pairs}} O\left(\frac{|s_0| k}{\log n}\right) = O\left(\frac{|s_0|}{\log n} \sum_{k=1}^{|s_0|} k \cdot \left\lfloor \frac{|s_0|}{k} \right\rfloor\right) = O\left(\frac{|s_0|^3}{\log n}\right)$$

Furthermore, checking which of the candidate patterns are consistent with S using the algorithm from Lemma 7 for all (k, l)-pairs in parallel requires $O(n\,|s_0| \log |s_0|/\log n)$ processors. A simple maximum calculation then suffices to output a consistent pattern with maximum $k + l$, which is descriptive by Lemma 3. ∎

As already mentioned earlier, it is not difficult to make the algorithm run on an EREW-PRAM in the same time and processor bounds. For this purpose, it is only necessary to make a sufficient number of copies of all memory locations that are accessed by several processors at the same time in order to ensure that each processor can work with its private copy. The details are somewhat tedious but standard, and therefore omitted here.

Furthermore, note that the work performed by the parallel algorithm, i.e., the product of its time and processor requirements, is the same as that performed by the improved sequential algorithm from Section 2 whenever $|s_0|^2 = O(n \log |s_0|)$. If $|s_0|$ is larger, the work performed by the parallel algorithm exceeds that of the sequential algorithm by a factor less than $O(|s_0|^2/n) = O(|s_0|)$.

Example 2 We give an example for the computation of a (k, l)-pattern in $Cons(S)$ by the parallel algorithm. Let the sample S contain the following strings:

$s_0 = 0000101001$	$\|s_0\| = 10$	$m_0 = 2$
$s_1 = 010101011011$	$\|s_1\| = 12$	$m_1 = 3$
$s_2 = 00101010101011$	$\|s_2\| = 14$	$m_2 = 4$

The values for m_i are given for the (k, l)-pair $(2, 6)$, which is the pair which we focus on for the rest of this example. First, the algorithm finds that S is prefix-free. It determines that $d = 2$ (the first position where the strings differ) and $t = 1$ (s_t is a string that differs from s_0 in position d). The entries of the array EQUAL are then computed from s_0 and s_1. The resulting values are given next, using 0 to denote “false” and 1 to denote “true.”

EQUAL

b \ j	1	2	3	4	5	6	7	8	9	10
0	1	0	1	0	0	0	0	0	0	0
1	0	1	0	1	1	1	1	0	1	1
2	1	0	1	0	0	0	1	1	0	1

In order to check whether there exists a candidate $(2,6)$-pattern, the algorithm creates a graph G from the array EQUAL. The graph is depicted in Figure 1. Obviously, there

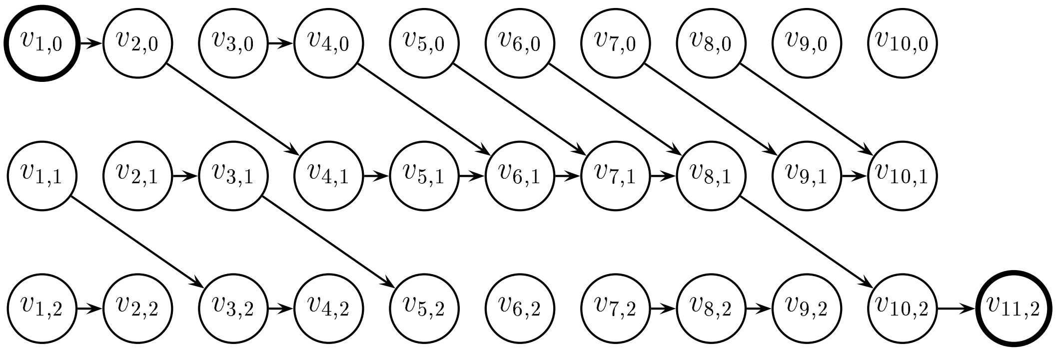

Figure 1: The graph obtained from the EQUAL array

is a path from $v_{1,0}$ to $v_{|s_0|+1,2}$ in G. Hence, there is a candidate $(2,6)$-pattern, and it can be derived from the path and from s_0 as follows:

Path:	$v_{1,0} \rightarrow$	$v_{2,0} \rightarrow$	$v_{4,1} \rightarrow$	$v_{5,1} \rightarrow$	$v_{6,1} \rightarrow$	$v_{7,1} \rightarrow$	$v_{8,1} \rightarrow$	$v_{10,2} \rightarrow$	$v_{11,2}$
Pattern:	0	x	0	1	0	1	x	1	

The nodes $v_{2,0}$ and $v_{8,1}$ on the path correspond to false-entries of the array EQUAL. As these nodes appear in positions 2 and 7 on the path, respectively, the candidate pattern π has occurrences of x in positions 2 and 7. The constant symbols in π are obtained by replacing $v_{j,b}$ by $s_0(j)$ for the remaining nodes on the path except the last one.

The algorithm verifies that the resulting $(2,6)$-pattern $0x0101x1$ is consistent with the sample and concludes that it is contained in $Cons(S)$. In addition, this pattern turns out to be a longest pattern in $Cons(S)$ and therefore descriptive for the given sample.

It should be mentioned that there is a special case for which the processor requirements of the algorithm can be improved. If the input sample contains two strings that are not identical but have equal length, the algorithm can be modified to use these two strings instead of s_0 and s_t to calculate the candidate (k,l)-patterns. This has the advantage that the value of an entry EQUAL$[j,b]$ of the array EQUAL does not depend on the second index b, because the value m_i is identical for the two strings. Hence, the array EQUAL is only one-dimensional in this case. Therefore, the size of EQUAL and the corresponding graph is only $O(n)$, and the candidate pattern π can be obtained in time $O(\log n)$ with $O(n/\log n)$ processors instead of $O(k|s_0|/\log n)$ processors as before. Consequently, the overall processor requirement is reduced to $O(n\,|s_0|\log|s_0|/\log n)$ in this case. Furthermore, note that it is not necessary to compute candidate patterns for all $O(|s_0|\log|s_0|)$ (k,l)-pairs. Let s_i and s_j be the two strings that are not identical but have equal length. Then it is sufficient to check for the $O(|s_i|)$ possible substitution lengths m_i whether there is a candidate pattern which can generate s_i and s_j by substituting strings of length m_i for x. The processor requirement for this alternative implementation is $O(n\,|s_i|/\log n)$, which is an improvement if $|s_i| = o(|s_0|\log|s_0|)$. A corresponding improvement is possible for the sequential algorithm from Section 2 in this special case as well.

4. Analyzing the Expected Total Learning Time

Until now, we have dealt with the update time of our set-driven learners by analyzing the proposed algorithms for computing descriptive one-variable patterns. Now, we are interested in the resulting total learning time.

Let π be the pattern to be inferred. The total learning time of any algorithm that tries to infer π is *unbounded* in the worst case, because there are infinitely many strings in $L(\pi)$ that can mislead the algorithm. To see this, assume that $\pi = x$ and that the learning algorithm is initially given strings of the form $0u$ for some $u \in \mathcal{A}^+$. The algorithm cannot rule out the possibility that the pattern to be inferred is $0x$ until it sees a string 0 or au for some $a \in \mathcal{A}\setminus\{0\}$ and $u \in \mathcal{A}^*$. On the other hand, in the *best case* two examples, namely $\pi[0/x]$ and $\pi[1/x]$, do always suffice for a learning algorithm that outputs descriptive patterns as hypotheses.

Hence, we assume that the strings presented to the algorithm are drawn from $L(\pi)$ according to a certain probability distribution. This allows us to obtain results about the *expected total learning time* of an 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. The assumption that strings of equal length are equally probable seems a very natural assumption.

We present a learning algorithm that infers a one-variable pattern π with expected total learning time $O(\ell^2\log\ell)$, where ℓ is the expected length of a string drawn from $L(\pi)$ according to the probability distribution.

It turns out advantageous not to calculate a descriptive pattern each time a new string is read. Instead, our proposed one-variable pattern learning algorithm, Algorithm 1LA, reads a certain number of strings before it starts to perform any calculations 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 a hypothesis as long as all sample strings read so far are the same, and it outputs x as a hypothesis once a sample string different from s_1 has been encountered. If the pattern π is a constant pattern, i.e., $|L(\pi)| = 1$, the correct hypothesis is output after the first sample string is read, and the algorithm never reaches the third phase. Otherwise, the algorithm uses a modified version of Algorithm 2 from Section 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. It rather uses the function *first_candidate* once to obtain a longest pattern in P', and the function *next_candidate* repeatedly to obtain the remaining patterns of P' in order of non-increasing length. This has the benefit of reducing the memory requirements for Algorithm 1LA substantially.

The pattern τ obtained from the call to *first_candidate* is used as the *current* candidate pattern. Each new sample string s is then compared to the current candidate pattern τ. If $s \in L(\tau)$, τ is output as a hypothesis. Otherwise, *next_candidate* is called to obtain a new candidate pattern τ'. Now, τ' becomes the current candidate pattern and is output as a hypothesis, no matter whether $s \in L(\tau')$. 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_candidate* is called again and a different list of candidate patterns is considered. Note that Algorithm 1LA may output *inconsistent* hypotheses, because the current candidate pattern τ may fail to generate a previously read sample string that has been discarded or even the current sample string if τ has just been obtained by a call to *next_candidate* in order to replace a previous candidate pattern that failed to generate the current sample string.

Algorithm 1LA is shown in Figure 2. The functions *first_candidate* and *next_candidate* represent a modified version of the algorithm from Section 2. Let $S = \{s, s'\}$. Let $P'(S) = \{vx\}$, if $s = vc$ for some $c \in \mathcal{A}$ and $s' = sw$ for some string w. Otherwise, denote by u the longest common prefix of s and s', and let $P'(S)$ 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 $P'(S)$ if we omit the consistency check. Hence, $P'(S) \supseteq Cons(S)$, where $Cons(S)$ is as defined in Section 2. Note that $P'(S)$ 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, $P'(S)$ contains at most $O(|s| \log |s|)$ patterns. Since we omit the consistency checks, a call to *first_candidate* and all subsequent calls to *next_candidate* until either the correct pattern is found or the prefix changes can be performed in time $O(|s|^2 \log |s|)$ altogether.

We will show that Algorithm 1LA correctly infers one-variable pattern languages from text in the limit, and that it correctly infers one-variable pattern languages from text with probability 1 if the sample strings are drawn from $L(\pi)$ according to a proper probability distribution.[2] In the latter case, the expected total learning time is shown to be $O(\ell^2 \log \ell)$, where ℓ is the expected length of a sample string.

THEOREM 2. *Let π be an arbitrary one-variable pattern. Algorithm 1LA correctly infers π from text in the limit.*

Proof. If π is a constant pattern, Algorithm 1LA outputs π after reading a single sample string and doesn't change its hypothesis later on. Otherwise, let $\pi = ux\mu$, where $u \in \mathcal{A}^*$ is a string of $d - 1$ constant symbols and $\mu \in Pat_1 \cup \{\varepsilon\}$. As every string in $L(\pi)$ will be eventually presented to Algorithm 1LA, sooner or later it will have encountered two strings that differ in position d. At this point, pattern π will be among the candidate patterns in the set P' implicitly maintained by Algorithm 1LA. For every pattern $\pi' \in P' \setminus \{\pi\}$ with

[2]Note that learning a language L in the limit and learning L from strings that are drawn from L according to a proper probability distribution are not the same.

```
{ 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_candidate(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_candidate(s, s')
    else if s'' ∉ L(τ) then
        τ ← next_candidate(s, s', τ)
    fi;
    output hypothesis τ;
od
```

Figure 2: Algorithm 1LA

$|\pi'| \geq |\pi|$, there are infinitely many strings in $L(\pi) \setminus L(\pi')$. Hence, all these patterns π' will be eventually discarded by the algorithm, and it will output the correct hypothesis π. After that, the algorithm will never change its hypothesis. ∎

THEOREM 3. *Let π be an arbitrary one-variable pattern. If sample strings are drawn from $L(\pi)$ according to a proper probability distribution, Algorithm 1LA correctly infers π with probability* 1.

Proof. If π is a constant pattern, Algorithm 1LA outputs π after reading a single sample string and converges. Otherwise, let $\pi = ux\mu$, where $u \in \mathcal{A}^*$ is a string of $d-1$ constant symbols and $\mu \in Pat_1 \cup \{\varepsilon\}$. After Algorithm 1LA has read two strings that differ in position d, pattern π will be one of the candidate patterns in the set P' implicitly maintained by the algorithm. As each new sample string differs from the first sample string in position d with probability $(|\mathcal{A}|-1)/|\mathcal{A}| \geq 1/2$ (this is a consequence of the fact that strings in $L(\pi)$ of equal length are equally probable), this event will happen eventually with probability 1. After that, as long as the current candidate pattern τ is different from π, the probability that the next sample string read is not in the language of τ is at least $1/2$ (cf. Lemma 8 below). Hence, all candidate patterns will be discarded with probability 1 until π becomes the current candidate pattern and is output as a hypothesis. After that, the algorithm converges. ∎

LEMMA 8. *Let $\pi = ux\mu$ be a one-variable pattern with constant prefix u, and $\mu \in Pat_1 \cup \{\varepsilon\}$. Let s_0, $s_1 \in L(\pi)$ be arbitrary such that $s_0(|u|+1) \neq s_1(|u|+1)$. Let $\tau \neq \pi$ be a pattern from $P'(\{s_0, s_1\})$ with $|\tau| \geq |\pi|$. Then τ fails to generate a string s drawn from $L(\pi)$ according to a proper probability distribution with probability at least* $(|\mathcal{A}|-1)/|\mathcal{A}|$.

Proof. Denote the number of occurrences of constant symbols in π and τ by $\#_{\mathcal{A}}(\pi)$, and $\#_{\mathcal{A}}(\tau)$, respectively. Recall that $\#_x(\pi)$ and $\#_x(\tau)$ denotes the number of occurrences of x in π and τ, respectively. Note that either $\#_x(\tau) > \#_x(\pi)$ or $\#_{\mathcal{A}}(\tau) > \#_{\mathcal{A}}(\pi)$ (or both). This is obvious if $|\tau| > |\pi|$, and for $|\tau| = |\pi|$ this follows from the fact that there is at most one (k,l)-pattern in $P'(\{s_0, s_1\})$ for each (k,l)-pair.

Let n be the length of the string s drawn from $L(\pi)$. If $n - \#_{\mathcal{A}}(\tau)$ is not divisible by $\#_x(\tau)$, then $\Pr(s \in L(\tau) \mid |s| = n) = 0$. Otherwise, we distinguish the following cases:

Case 1. $\#_x(\tau) > \#_x(\pi)$. Let I_π be the set of positions in s which correspond to the first symbol of a substitution string α_π that is substituted for x in π to obtain s. Similarly, let I_τ be the set of positions in s which correspond to the first symbol of a substitution string that is substituted for x in τ to obtain s, assuming such a substitution exists. As $|I_\tau| > |I_\pi|$, I_τ contains a position i_τ which corresponds to the first symbol of a substitution string with respect to τ, but to a constant or a second, third, fourth, etc. symbol of α_π with respect to π. Hence, τ can only generate s if that constant or second, third, fourth, etc. symbol of α_π is equal to the first symbol of a substitution string with respect to τ, which must in turn be equal to the first symbol of α_π because π and τ have the same constant prefix. The probability for this to happen is $1/|\mathcal{A}|$.

Case 2. $\#_{\mathcal{A}}(\tau) > \#_{\mathcal{A}}(\pi)$. Let I_π be the set of positions in s which correspond to constants with respect to π, and let I_τ be the set of positions in s which correspond to a constant with respect to τ. As $|I_\tau| > |I_\pi|$, I_τ contains a position i_τ which corresponds to a constant c with respect to τ but to a symbol of α_π with respect to π. The probability that this symbol of α_π is equal to c is $1/|\mathcal{A}|$. ∎

Now, we analyze the total expected learning time of Algorithm 1LA. Obviously, the total expected learning time is $O(\ell)$ if the pattern π to be learned is a constant pattern. Hence, we assume in the following that π contains at least one occurrence of x.

The total learning time of Algorithm 1LA is divided into three phases. Phase 1 refers to the time when the algorithm reads strings but does not yet perform any calculations because $|s_r| \geq r$. Phase 2 refers to the time when the algorithm has already encountered a sample string s_r with $|s_r| < r$, but is waiting for a sample string that differs from s_1. Phase 3 starts when the algorithm makes the first call to *first_candidate*, and it ends when the algorithm outputs the correct hypothesis for the first time.

Next, we recall the definition of the median, and establish a basic property of it that is used later. As usual, we use $E(R)$ to denote the *expectation* of a random variable R.

DEFINITION 4. *Let R be any random variable with* $\operatorname{range}(R) \subseteq \mathbb{N}$. *The median of R is the number $\mu \in \operatorname{range}(R)$ such that $Pr(R < \mu) \leq \frac{1}{2}$ and $Pr(R > \mu) < \frac{1}{2}$.*

PROPOSITION 1. *Let R be a random variable with* $\operatorname{range}(R) \subseteq \mathbb{N}^+$. *Then its median μ satisfies $\mu \leq 2E(R)$.*

Proof.

$$\begin{aligned} E(R) = \sum_{n \geq 1} n\Pr(R = n) &\geq \sum_{n \geq \mu} n\Pr(R = n) \geq \mu\Pr(R \geq \mu) \\ &= \mu(1 - \Pr(R < \mu)) \geq \frac{\mu}{2} \end{aligned}$$

∎

This proposition also follows from the Markov inequality $\Pr(R \geq t) \leq E(R)/t$, which holds for all random variables R assuming only non-negative values and for all positive real numbers t. In addition, note that the median can be significantly smaller than the expectation. For example, the median is always finite, even if the expectation is not. The proposition above is only a worst-case estimate.

LEMMA 9. *Let D be any proper probability distribution, and let L be the random variable taking as values the length of a string drawn from $L(\pi)$ with respect to D. Furthermore, let μ be the median of L and let ℓ be its expectation. Then, the expected number of steps performed by Algorithm 1LA during Phase 1 is $O(\mu\ell)$.*

Proof. Let L_i be the random variable whose value is the length of the i-th string read by the algorithm. Obviously, the distribution of L_i is the same as that of L. Let R be the random variable whose value is the number of strings Algorithm 1LA reads in Phase 1. Let $L_\Sigma := L_1 + \cdots + L_R$ be the number of symbols read by Algorithm 1LA during Phase 1. Let W_1 be the random variable whose value is the time spent by Algorithm 1LA in Phase 1. Obviously, $W_1 = O(L_\Sigma)$.

Claim 1. *For $i < r$, we have:*

$$E(L_i \mid R = r) \le \frac{E(L)}{Pr(L_i \ge i)} \tag{1}$$

As L_i must be at least i provided $R > i$, Equation (1) can be proved as follows:

$$\begin{aligned}
E(L_i \mid R = r) &= \sum_{k=i}^{\infty} k\Pr(L_i = k \mid R = r) \\
&= \sum_{k=i}^{\infty} k \frac{\Pr(L_i = k \wedge R = r)}{\Pr(R = r)} \\
&= \sum_{k=i}^{\infty} k \frac{\Pr(L_1 \ge 1)\cdots\Pr(L_i = k)\cdots\Pr(L_{r-1} \ge r-1)\Pr(L_r < r)}{\Pr(L_1 \ge 1)\cdots\Pr(L_i \ge i)\cdots\Pr(L_{r-1} \ge r-1)\Pr(L_r < r)} \\
&= \sum_{k=i}^{\infty} k\Pr(L_i = k) \Big/ \Pr(L_i \ge i) \\
&\le \frac{E(L_i)}{\Pr(L_i \ge i)} = \frac{E(L)}{\Pr(L_i \ge i)}
\end{aligned}$$

Thus, Claim 1 is proved.

Similarly, it can be shown that

$$E(L_r \mid R = r) \le \frac{E(L)}{\Pr(L_r < r)} \tag{2}$$

Furthermore, it is clear that

$$E(L_r \mid R = r) \le r - 1 \tag{3}$$

Now, we rewrite $E(L_\Sigma)$:

$$E(L_\Sigma) = \underbrace{\sum_{r=1}^{\mu} E(L_\Sigma \mid R = r)\Pr(R = r)}_{(\alpha)} + \underbrace{\sum_{r>\mu} E(L_\Sigma \mid R = r)\Pr(R = r)}_{(\beta)}$$

Using $\Pr(L_i \ge i) \ge 1/2$ for $i \le \mu$ as well as Equations (1) and (3), we obtain:

$$\begin{aligned}
(\alpha) &= \sum_{r=1}^{\mu} \Big(E(L_1 \mid R = r) + \cdots + E(L_r \mid R = r) \Big) \Pr(R = r) \\
&= \sum_{r=1}^{\mu} \left(\sum_{i=1}^{r-1} E(L_i \mid R = r) + E(L_r \mid R = r) \right) \Pr(R = r) \\
&\le \sum_{r=1}^{\mu} \left(\sum_{i=1}^{r-1} \frac{E(L)}{\Pr(L_i \ge i)} + (r-1) \right) \Pr(R = r) \\
&\le \sum_{r=1}^{\mu} \left((r-1)2E(L) + (r-1) \right) \Pr(R = r) \\
&\le (\mu - 1)(2E(L) + 1) = O(\mu\ell)
\end{aligned}$$

For (β), we use Equations (1) and (2) to obtain:

$$
\begin{aligned}
(\beta) = & \sum_{r=\mu+1}^{\infty} \Big(E(L_1 \mid R=r) + \cdots + E(L_r \mid R=r) \Big) \Pr(R=r) \\
\leq & \sum_{r=\mu+1}^{\infty} \Big(\frac{E(L)}{\Pr(L_1 \geq 1)} + \cdots + \frac{E(L)}{\Pr(L_{r-1} \geq r-1)} + \frac{E(L)}{\Pr(L_r < r)} \Big) \Pr(R=r) \\
\leq & \sum_{r=\mu+1}^{\infty} \Big(\frac{(r-1)E(L)}{\Pr(L_{r-1} \geq r-1)} + \frac{E(L)}{\Pr(L_r < r)} \Big) \Pr(L_1 \geq 1) \cdots \Pr(L_{r-1} \geq r-1) \Pr(L_r < r) \\
\leq & \sum_{r=\mu+1}^{\infty} rE(L) \Pr(L_1 \geq 1) \cdots \Pr(L_{r-2} \geq r-2) \\
\leq & \sum_{r=\mu+1}^{\infty} rE(L) \left(\frac{1}{2}\right)^{r-2-\mu} \qquad (\text{using } \Pr(L_i \geq i) \leq \tfrac{1}{2} \text{ for } i > \mu) \\
= & \; E(L) \sum_{r=0}^{\infty} (r+\mu+1) \left(\frac{1}{2}\right)^{r-1} \\
= & \; E(L) \Big(\underbrace{\sum_{r=0}^{\infty} r \left(\frac{1}{2}\right)^{r-1}}_{=4} + \underbrace{\sum_{r=0}^{\infty} (\mu+1) \left(\frac{1}{2}\right)^{r-1}}_{=4(\mu+1)} \Big) \\
= & \; E(L)(4\mu+8) = O(\mu\ell)
\end{aligned}
$$

Hence, the expected number of steps performed in Phase 1 is $E(W_1) = O(E(L_\Sigma)) = O(\mu\ell)$. ∎

Now, under the same assumptions as in Lemma 9, we can estimate the expected number of steps performed in Phase 2 as follows.

LEMMA 10. *During Phase 2, the expected number of steps performed by Algorithm 1LA is $O(\ell)$.*

Proof. If different sample strings have already been read during Phase 1, no work is performed in Phase 2. Otherwise, denote by T the number of strings read in Phase 2. As π contains at least one variable, the probability that a newly read sample string differs from the previously read strings is at least $\frac{1}{2}$. Hence, $\Pr(T=t) \leq 2^{-(t-1)}$ for $t > 1$.

Denote by W_2 the number of steps performed in Phase 2. Furthermore, since it is clear that $E(W_2 \mid T=t) = tE(L)$, we obtain:

$$
\begin{aligned}
E(W_2) &= E(W_2 \mid T=0)\Pr(T=0) + E(W_2 \mid T=1)\Pr(T=1) \\
&\quad + \sum_{t=2}^{\infty} E(W_2 \mid T=t)\Pr(T=t) \\
&\leq 0 + E(L) + \sum_{t=2}^{\infty} tE(L) \left(\frac{1}{2}\right)^{t-1} = O(\ell)
\end{aligned}
$$

∎

Finally, we deal with Phase 3. Again, let L be as in Lemma 9. Then, the average amount of time spent in Phase 3 can be estimated as follows.

LEMMA 11. *During Phase 3, the expected number of steps performed in calls to the functions first_candidate and next_candidate is* $O(\mu^2 \log \mu)$.

Proof. Denote by W_3^c the number of steps performed in all calls to the functions *first_candidate* and *next_candidate* in Phase 3. We study the conditional expectation $E(W_3^c \mid R = r)$ first. We account for the function calls in groups consisting of one call to *first_candidate* and all subsequent calls to *next_candidate* prior to another call to *first_candidate*. Note that such a *group of candidate calls* requires time $O(|s|^2 \log |s|)$ altogether, where s is the shorter of the two parameters for the call to *first_candidate*.

If $R = r$, the shortest string read in Phase 1 has length at most $r - 1$, and each group of candidate calls takes at most $O((r-1)^2 \log(r-1))$ steps. How many groups of candidate calls are performed? With probability 1, at least one group is started. A new group is started only when the longest common prefix of all sample strings becomes shorter. Since at least two strings are read before the first group of candidate calls is started, the probability for a second group is at most $1/2$. Whenever the longest common prefix of all sample strings changes, the probability that the new prefix is the final one and that, therefore, the next group of candidate calls is the last one is at least $1/2$. Hence, the probability that k groups of candidate calls are performed is at most $2^{-(k-1)}$, and we have:

$$
\begin{aligned}
E(W_3^c \mid R = r) &= O((r-1)^2 \log(r-1)) \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) \\
&= O((r-1)^2 \log(r-1)) \qquad (4)
\end{aligned}
$$

Furthermore, we can rewrite $E(W_3^c)$ as follows:

$$
\begin{aligned}
E(W_3^c) &= \sum_{r=1}^{\infty} E(W_3^c \mid R = r)\Pr(R = r) \\
&= \underbrace{\sum_{r=1}^{\mu} E(W_3^c \mid R = r)\Pr(R = r)}_{(\alpha)} + \underbrace{\sum_{r=\mu+1}^{\infty} E(W_3^c \mid R = r)\Pr(R = r)}_{(\beta)}
\end{aligned}
$$

Obviously, Equation (4) implies $(\alpha) = O(\mu^2 \log \mu)$. In addition, (β) can be bounded as follows:

$$
\begin{aligned}
(\beta) &= \sum_{r=\mu+1}^{\infty} O((r-1)^2 \log(r-1)) \cdot \Pr(R = r) \\
&= \sum_{r=0}^{\infty} O((r+\mu)^2 \log(r+\mu)) \cdot \Pr(R = r + \mu + 1) \\
&\leq \sum_{r=0}^{\infty} O((r^2 + \mu^2 + 2r\mu)(\log r + \log \mu)) \cdot \left(\frac{1}{2}\right)^{r+1} \\
&\leq O(\mu^2 \log \mu)
\end{aligned}
$$

Hence, we obtain $E(W_3^c) = O(\mu^2 \log \mu)$. ∎

LEMMA 12. *During Phase 3, the expected number of steps performed in reading strings is $O(\mu\ell\log\mu)$.*

Proof. Denote by W_3^r the number of steps performed while reading strings in Phase 3. We make a distinction between strings read before the correct set of candidate patterns is considered, and strings read afterwards until the end of Phase 3. The former are accounted for by random variable V_1, the latter by V_2.

If the correct set of candidate patterns, i.e., the set containing π, is not yet considered, the probability that a new string does not force the correct set of candidate patterns to be considered is at most $1/2$. Denote by K the random variable whose value is the number of strings that are read in Phase 3 before the correct set of candidate patterns is considered. We have:

$$E(V_1) = \sum_{k=0}^{\infty} E(V_1 \mid K = k)\Pr(K = k) \le \sum_{k=0}^{\infty} kE(L)\left(\frac{1}{2}\right)^k = O(E(L))$$

Assume that the correct set of candidate patterns P' contains M patterns that are considered before pattern π. For any such pattern τ, the probability that a string drawn from $L(\pi)$ according to a proper probability distribution is in the language of τ is at most $1/2$, because either τ has an additional variable or τ has an additional constant symbol (Lemma 8). Denote by V_2^i the steps performed for reading strings while the i-th pattern in P' is considered.

$$E(V_2 \mid M = m) = \sum_{i=1}^{m} E(V_2^i | M = m) \le \sum_{i=1}^{m}\sum_{k=0}^{\infty} kE(L)\left(\frac{1}{2}\right)^k = O(mE(L))$$

Since $M = O(R\log R)$, we obtain:

$$\begin{aligned}
E(V_2) &= \sum_{r=1}^{\infty} E(V_2 \mid R = r)\Pr(R = r) \\
&= \underbrace{\sum_{r=1}^{\mu} E(V_2 \mid R = r)\Pr(R = r)}_{=O(E(L)\mu\log\mu)} + \underbrace{\sum_{r=\mu+1}^{\infty} E(V_2 \mid R = r)\Pr(R = r)}_{(\alpha)}
\end{aligned}$$

and

$$\begin{aligned}
(\alpha) &\le \sum_{r=\mu+1}^{\infty} O(E(L) r\log r)\left(\frac{1}{2}\right)^{r-\mu} \\
&= E(L)\sum_{r=1}^{\infty} O((r+\mu)\log(r+\mu))\left(\frac{1}{2}\right)^{r} \\
&= O(\mu E(L)\log\mu)
\end{aligned}$$

Hence, we have $E(W_3^r) = E(V_1) + E(V_2) = O(\mu\ell\log\mu)$. ∎

LEMMA 13. *During Phase 3, the expected number of steps performed in checking whether the current candidate pattern generates a newly read sample string is $O(\mu\ell\log\mu)$.*

Proof. Denote by W_3^{cd} the number of steps performed during Phase 3 for checking whether a newly read sample string can be generated by the current candidate pattern. Each such check requires a number of steps that is linear in the length of the respective sample string. For each string read in Phase 3, only one such check is performed. Hence, each check can be charged to the corresponding read, and we obtain $E(W_3^{cd}) = O(E(W_3^r)) = O(\mu E(L) \log \mu)$. ∎

Putting it all together, we arrive at the following expected total learning time required by Algorithm 1LA.

THEOREM 4. *If the sample strings are drawn from $L(\pi)$ according to a proper probability distribution with expected string length ℓ the expected total learning time of Algorithm 1LA is $O(\ell^2 \log \ell)$.*

Proof. Taking into account that $\mu = O(E(L))$, the assertion follows directly from the Lemmas 9 through 13. ∎

It should be mentioned that Algorithm 1LA can be implemented with very small space requirements. It suffices to store only two sample strings and one candidate pattern in memory. The two sample strings are the shortest string s encountered so far and a sample string s' that differs from s, either by being longer than s (if all sample strings have the form sv) or by having a different character in position $|u| + 1$ (if the longest common prefix u of the sample strings satisfies $|u| < |s|$). In addition, a third string, namely the newly read sample string, must be stored temporarily.

5. Learning with Superset Queries

Angluin [4] showed that the class of all pattern languages is not learnable with polynomially many queries if only equivalence, membership, and subset queries are allowed as long as any hypothesis space $\mathcal{H}$ is considered such that range$(\mathcal{H}) = PAT$. Applying the same proof technique, this result may be easily extended to the one-variable pattern languages. That is, there is no query learning algorithm exactly inferring PAT_1 with respect to any hypothesis space $\mathcal{H}$ with range$(\mathcal{H}) \subseteq PAT$ that uses only polynomially many queries.

On the other hand, positive results are also known. First, Lange and Wiehagen [17] showed PAT to be 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. Additionally, they proved that $PAT \cup FINQ$ is exactly learnable with respect to $PAT \cup FINQ$ using polynomially many disjointness queries, too, where $FINQ$ is the set of all finite query languages really needed. Moreover, their proof technique easily extends to the one-variable pattern languages, i.e., PAT_1 is exactly inferable with respect to $PAT_1 \cup FIN$ by asking polynomially many disjointness queries. It remains open, however, whether polynomially many disjointness queries do suffice, if the admissible hypothesis space is PAT or PAT_1 itself, respectively.

Second, Angluin [4] established an algorithm exactly learning PAT with respect to PAT by asking polynomially many *superset queries.* Clearly, this algorithm also learns one-variable patterns as a special case. But what can be said about the hypothesis space needed? Angluin's [4] algorithm starts by determining the length of π by asking $L(\pi) \subseteq L(\tau)$ for $\tau = x_1,\ x_1x_2,\ x_1x_2x_3$ and so on until the answer is *no*. If i is the minimum for $L(\pi) \not\subseteq L(x_1 \dots x_i)$, then $|\pi| = i - 1$. Thus, this approach works for one-variable patterns, too, but it requires choosing general patterns τ for asking the queries. It does not work if the hypothesis space contains only one-variable patterns, however. Moreover, there seems to be no way to modify this superset query algorithm for general pattern languages to obtain an algorithm that learns PAT_1 with respect to PAT_1. Hence, the following question arises naturally.

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 algorithm for exactly learning PAT_1 with respect to PAT_1 needs counterexamples. Interestingly enough, it does not need a counterexample for every query answered negatively, instead two counterexamples always suffice.

The reason why counterexamples are so useful to learn one-variable patterns seems to be that they limit the search space tremendously. The next theorem shows that one-variable patterns are not learnable by a polynomial number of restricted superset queries.

Lemma 14. *Let $w \in \mathcal{A}^*$. If $L(xwx) \subseteq L(\tau)$, then $\tau = x$ or $\tau = xwx$.*

Proof. By Lemma 1 we know that $L(\pi) \subseteq L(\tau)$ implies $\pi = \tau[\varrho/x]$ for some $\varrho \in Pat$. Here $xwx = \tau[\varrho/x]$ for some $\varrho \in Pat_1$. Obviously, τ must begin and end with an x. If $\tau = x$, we are done. Now, let $\tau \neq x$. Thus, $\tau = x\tau'x$, and since $xwx = (x\tau'x)[\varrho/x]$, the pattern ϱ must also begin and end with an x.

If $\varrho = x$ then $\tau = xwx$, and again we are done. Finally, if $\varrho = x\varrho'x$, then $xwx = \tau[\varrho/x] = x\varrho'x\tau'[\varrho/x]x\varrho'x \neq xwx$, a contradiction. ∎

Now, we are ready for proving the theorem announced above.

Theorem 5. *Any algorithm exactly identifying all $L \in PAT_1$ generated by a pattern π of length n with respect to PAT_1 by using restricted superset and equivalence queries only must make at least $|\mathcal{A}|^{n-2} \geq 2^{n-2}$ queries in the worst case.*

Proof. Assume any learning algorithm LA. For constructing an adversary, we consider all patterns $\pi = xwx$, $w \in \mathcal{A}^+$, of length n and show that LA has to ask at least $|\mathcal{A}|^{n-2}$ many queries when learning one of the languages $L(\pi)$. Note that there are $|\mathcal{A}|^{n-2}$ many pairwise different such patterns. We denote the target language by L_*.

We can assume that the Algorithm LA does not ask the question $L_* \subseteq L(x)$, since this question is answered *yes* for each possible π. Furthermore, without loss of generality we can assume that no question is asked twice.

Now, Lemma 14 is the only ingredient needed. Suppose, the Algorithm LA is asking any pattern τ, i.e., whether or not $L_* \subseteq L(\tau)$ or $L_* = L(\tau)$. The adversary maintains a list S of all patterns π described above, and simply answers each superset and equivalence query with *no*, until $|S| = 1$, in which case he gives up. If τ has been one of the patterns xwx, this pattern is deleted from S, otherwise S remains unchanged. By Lemma 14, all answers given are compatible with the remaining element π of S, and thus at least $|\mathcal{A}|^{n-2}$ many queries are necessary for exactly identifying $L_* = L(\pi)$. ∎

Furthermore, we can show that learning one-variable pattern languages with a polynomial number of superset queries is not possible if the algorithm may ask for a single counterexample only.

THEOREM 6. *Any algorithm that exactly identifies all one-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.*

Proof. Take an arbitrary algorithm that learns one-variable patterns with superset queries and asks only for one counterexample. Assume again that there is no query $L(\pi) \subseteq L(x)$. The alphabet $\mathcal{A}$ should contain the symbol 0.

Let $\tau_1, \ldots, \tau_j$ be the patterns queried by the algorithm if all questions are answered *no* and let $L(\pi) \subseteq L(\tau_j)$ be the first (and only) unrestricted query. Let $\tau_{j+1}, \tau_{j+2}, \ldots$ be the following queries, if the algorithm gets 0^k as the counterexample and all following questions are also answered *no*. The constant k is some number such that $0^k \notin L(\tau_j)$ and either k or $k-1$ is a multiple of 4. Such a k must exist unless $\tau_j \in 0^*x0^*$ (when we choose 1^k instead and change in the rest of the proof all 0's to 1).

Let $\bar{x} = 0$, $\bar{0} = x$, and let

$$A = \left\{ xa_1xa_2xa_3 \ldots xa_{m-1}xa_m\bar{a}_m0\bar{a}_{m-1}0 \ldots \bar{a}_30\bar{a}_20\bar{a}_1x \;\middle|\; a_1, \ldots, a_m \in \{0, x\} \right\}$$

or

$$A = \left\{ xa_1xa_2xa_3 \ldots xa_{m-1}xa_m0\bar{a}_m0\bar{a}_{m-1}0 \ldots \bar{a}_30\bar{a}_20\bar{a}_1x \;\middle|\; a_1, \ldots, a_m \in \{0, x\} \right\}$$

be a set of patterns of length k with $m = k/4$ or $m = (k-1)/4$. Obviously, $0^k \in L(\pi)$ for all $\pi \in A$.

We force the algorithm to ask at least $2^m - 1$ queries by letting it identify a pattern $\pi \in A$ with $\pi \neq \tau_i$ for $1 \leq i < 2^m$. Such a π exists because $|A| = 2^m$.

How does the algorithm react, if it learns π and eventually gets 0^k as the counterexample? We claim that the first $2^m - 1$ questions are answered *no* because $L(\tau_i) \not\supseteq L(\pi)$. The counterexample 0^k is then really a counterexample because $0^k \in L(\pi)$ by definition of A.

It remains to be shown that $\tau_i \neq \pi$ implies $L(\tau_i) \not\supseteq L(\pi)$. Assume $L(\tau_i) \supseteq L(\pi)$. Then $\tau_i[\varrho/x] = \pi$ for some $\varrho \in Pat_1$. Since $\tau_i \neq x$ and π begins and ends with an x, τ_i must be of the form $\tau_i = x\omega x$ and ϱ of the form $\varrho = x\varrho' x$. Then $\pi = x\varrho' x\omega' x\varrho' x$ for some ω'. We can, however, show that π has no proper prefix that is also a suffix and whose length is between 2 and $\lfloor |\pi|/2 \rfloor$. Here $x\varrho' x$ would be such a prefix/suffix, so we have a contradiction and $L(\tau_i) \not\supseteq L(\pi)$ follows.

Why does π not have such a prefix/suffix? Let us assume ν is such a prefix/suffix. If $|\nu|$ is odd, then $\nu = xa_1xa_2xa_3 \ldots xa_ix$ because it is a prefix. On the other hand

$\nu = 0\bar{a}_i 0\bar{a}_{i-1}0\ldots\bar{a}_2 0\bar{a}_1 x$ because it is a suffix, and thus $\nu = 0\ldots = x\ldots$, a contradiction. If $|\nu|$ is even, then $\nu = xa_1xa_2xa_3\ldots xa_i$ and $\nu = \bar{a}_i 0\bar{a}_{i-1}0\ldots\bar{a}_2 0\bar{a}_1 x$. Now $x = \bar{a}_i$ (first symbols) and $a_i = x$ (last symbols), again a contradiction. ∎

```
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);
        repeat τ ← max(R); R ← R \ {τ}
        until L(π) ⊆ L(τ)
    fi
fi;
return τ
```

Figure 3: Algorithm Q. This algorithm learns a pattern π by superset queries. The queries have the form "$L(\pi) \subseteq L(\tau)$," where $\tau \in \mathit{Pat}_1$ is chosen by the algorithm. If the answer to a query $L(\pi) \subseteq L(\tau)$ is *no*, the algorithm 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 works as follows (see Figure 3). Assume the algorithm should learn some pattern π. First the algorithm asks whether $L(\pi) \subseteq L(0) = \{0\}$ holds. This is the case iff $\pi = 0$, and if the answer is *yes* the algorithms knows the right result.

Otherwise, the algorithm obtains a counterexample $C(0) \in L(\pi)$. Let $C(0)^{\leq j}$ be the prefix of $C(0)$ with length j. By asking questions $L(\pi) \subseteq L(C(0)^{\leq j}x)$ for $j = 1, 2, 3, \ldots$ until the answer is *no*, the algorithm computes

$$i = \min\{\, j \mid L(\pi) \not\subseteq L(C(0)^{\leq j}x)\,\}.$$

Now we know that π starts with $C(0)^{\leq i-1}$ (since $L(\pi) \subseteq L(C(0)^{\leq i-1}x) = C(0)^{\leq i-1}\mathcal{A}^+$); but what about the i-th position of π? If π has no i-th position, i.e., $|\pi| = i - 1$, then π contains no variable and therefore $\pi = C(0)$. The algorithms asks $L(\pi) \subseteq L(C(0))$ to determine whether this is indeed the case. An example is $\pi = 1011$ and $C(0) = 1011$; here $i = 4$ since $\{1011\} = L(\pi) \subseteq L(C(0)^{\leq 3}x) = 101\mathcal{A}^+$, but $\{1011\} = L(\pi) \not\subseteq L(C(0)^{\leq 4}x) = 1011\mathcal{A}^+$. The question $L(\pi) \subseteq L(C(0))$ is answered affirmatively and the algorithm presents $C(0) = 1011$ as its correct hypothesis.

Otherwise, i.e., if $L(\pi) \not\subseteq L(C(0))$, then π contains at least one variable and $\pi(i)$ cannot be a symbol from $\mathcal{A}$ since this would imply that $\pi(i)$ is equal to the i-th symbol of $C(0)$ and, therefore, $L(\pi) \subseteq L(C(0)^{\leq i}x)$, a contradiction. Thus $\pi(i) = x$. At this point the algorithm 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 12 computes $R = Cons(S)$, the set of all patterns consistent with S and coinciding with the words in S in their first $i-1$ positions. Since π is consistent with S and 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 to be performed is to find π among all patterns in R. We find π by removing other patterns from R. The following lemma gives a sufficient and necessary condition for this end.

LEMMA 15. *Let $S \subseteq \mathcal{A}^+$ and let $\pi, \tau \in Cons(S)$ with $|\pi| \leq |\tau|$. Then $L(\pi) \subseteq L(\tau)$ implies $\pi = \tau$.*

Proof. If $|\pi| < |\tau|$ then $L(\pi) \not\subseteq L(\tau)$. Let us therefore assume $|\pi| = |\tau|$.

By Lemma 1, we have $L(\pi) \subseteq L(\tau)$ iff $\pi = \tau[\varrho/x]$ for some $\varrho \in Pat_1$. Since $|\pi| = |\tau|$, we must have $|\varrho| = 1$; thus either $\varrho = x$ or $\varrho \in \mathcal{A}$. If $\varrho \in \mathcal{A}$, then $\pi \in \mathcal{A}^*$, a contradiction since all patterns in $Cons(S)$ contain at least one x. Hence, $\varrho = x$ and $\pi = \tau[x/x] = \tau$. ∎

The algorithm 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 15 guarantees that $\tau = \pi$. It is important to start with long patterns, since Lemma 15 does not hold for $|\tau| < |\pi|$.

For finding the correct i, the algorithm asks up to $|\pi|$ queries followed by up to $O(m \log m)$ additional queries to identify π. The number of queries (and the running time) is therefore polynomial in the pattern length and the length of the counterexamples returned.

Example 3 Let $\mathcal{A} = \{0,1\}$ and let $\pi = 01x0xx1x$. The algorithm starts by asking $L(\pi) \subseteq L(0)$ and the answer is *no*. The algorithm now asks for a counterexample to $L(0)$ and gets 0110101011011101. To compute the fixed prefix up to the first x of π, the next queries are $L(\pi) \subseteq L(0x)$, $L(\pi) \subseteq L(01x)$, and $L(\pi) \subseteq L(011x)$. Since the first two answers are *yes*, but the third answer is *no*, the algorithm knows that either $\pi \in \mathcal{A}^+$ or the fixed prefix is 01 followed by an x. By asking $L(\pi) \subseteq L(0110101011011101)$ (*no*) the second possibility turns out to be right.

Now we have to give a counterexample, say, 0100000000001000, to $L(\pi) \subseteq L(011x)$. The algorithm constructs the sample $S = \{0110101011011101,\ 0100000000001000\}$ and computes $R = Cons(S) = \{01x0x0x0xx0x1x0x, 01x0xx1x, 01x\}$, which must contain π. After the question $L(\pi) \subseteq L(01x0x0x0xx0x1x0x)$ is answered *no*, $01x0x0x0xx0x1x0x$ is removed from R and $01x0xx1x$ is the longest remaining pattern. The next question $L(\pi) \subseteq L(01x0xx1x)$ is answered *yes* and $01x0xx1x$ is identified as π.

In this example seven queries suffice to identify π.

The following theorem states the main result of this section.

THEOREM 7. *There exists an algorithm Q learning PAT_1 with respect to PAT_1 by asking only superset queries. The query complexity of Q 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.*

6. Conclusions and Open Problems

During the last 15 years pattern languages have attracted considerable attention in machine learning, formal language theory, and several interesting applications have emerged (cf., e.g., Shinohara and Arikawa [26] and the references therein). Taking the growing interest in applications into account, the problem of efficient learning becomes a major issue. This demand leads to several new and interesting questions. Clearly, any efficient learner is required to have a polynomial update time. Furthermore, it is highly desirable to provide performance bounds concerning the total learning time, and/or to elaborate efficient parallel learners.

The present paper addressed these issues for the special case of one-variable pattern language inference. First, we provided a new algorithm learning PAT_1 consistently, set-drivenly, and responsively from positive data (cf. Algorithm 2). This algorithm achieves update time $O(n^2 \log n)$ for input samples of size n, which saves a factor of n^2 over the best previously known algorithm by Angluin [2]. Moreover, the computation of descriptive patterns was efficiently parallelized achieving parallel update time $O(\log n)$ using $O(n^3/\log n)$ processors on an EREW-PRAM. The resulting parallel learning algorithm is still consistent, set-driven and responsive. As far as we know, this is the first parallel algorithm computing descriptive patterns.

Next, we turned our attention to the total learning time of Algorithm 2, and arrived at a modified version of it which has optimal *expected total learning time*. Here, by optimal we mean that the total learning time of Algorithm 1LA equals the update time of Algorithm 2 up to a constant factor. The price paid is giving up the requirements to learn consistently and set-drivenly. It is also interesting to compare our algorithm to Lange and Wiehagen's [17] inference procedure learning all pattern languages. This algorithm has been analyzed with respect to its expected total learning time, too (cf. [33]). An easy inspection of the analysis given in [33] shows that the expected total learning time of Lange and Wiehagen's algorithm is $O(\ell\mu^{-1})$ for target languages $L(\pi)$, where ℓ is again the expected string length, and μ is the probability to see a *shortest* string from $L(\pi)$. Hence, whenever $\mu < 1/(\ell \log \ell)$ our Algorithm 1LA behaves better than Lange and Wiehagen's [17] with respect to its expected total learning time. The difference in behavior finds its explanation in the fact that Lange and Wiehagen's algorithm definitely needs two different shortest strings from $L(\pi)$ for achieving convergence, while our Algorithm 1LA does not. As far as we know, 1LA is the first pattern language learning algorithm which provably converges even in case it has not received any string of minimal length from $L(\pi)$.

Moreover, we could successfully apply our basic technique for computing descriptive patterns to obtain an efficient active learning algorithm asking superset queries. Additionally, we established a tight bound on the number of *unrestricted* superset queries needed by any algorithm learning PAT_1 with respect to PAT_1 by asking superset queries: just two are necessary and sufficient.

Next, we discuss possible generalizations of the results obtained. Let us start with erasing one-variable pattern languages. Taking into account that $\{\pi[\varepsilon/x]\}$ is singleton for all $\pi \in Pat_1$, our results easily generalize to erasing one-variable pattern languages. Clearly, if a pattern does not contain any variable, there is only one text for it. Otherwise, the resulting pattern language is infinite. Thus, as long as any of our learners has just seen a singleton sample, it outputs the only string in it as hypothesis. Otherwise, it always ignores the shortest string in the sample but behaves otherwise as described. The same idea applies *mutatis mutandis* to our query learner.

Further research should deal with unions of one-variable pattern languages. This class is much richer and more interesting than PAT_1 itself (cf., e.g. [21, 22, 26]). Furthermore, a distinction has to be made between unbounded unions and an *a priori* restricted number m of allowed unions. In the first case, the whole class is not learnable if erasing substitutions are allowed (cf. [26]). As far as *a priori* bounded unions are concerned, their learnability by a polynomial update-time algorithm has been known for a rather long time (cf., e.g. [26]), but nothing is known concerning the resulting total learning time and efficient parallelizations.

ACKNOWLEDGEMENTS

A substantial part of this work has been done while the second author was visiting the Research Institute of Fundamental Information Science (RIFIS) (now Department of Informatics) of Kyushu University at Fukuoka, Japan. This visit has been supported by the Japanese Society for the Promotion of Science under Grant No. 106011. He is greatfully indebted to Setsuo Arikawa for providing excellent working conditions during his stay at RIFIS. He also thanks Janos Csirik for several discussions and suggestions.

The fifth author kindly acknowledges the support by the Grant-in-Aid for Scientific Research (C) from the Japan Ministry of Education, Science, Sports, and Culture under Grant No. 07680403.

References

[1] H. Alt, T. Hagerup, K. Mehlhorn, and F. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM J. Comput.*, 16(5):808–835, 1987.

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

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

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

[5] D. R. Bean, A. Ehrenfeucht and G. F. McNulty. Avoidable patterns in strings of symbols. *Pacific J. of Mathematics* 85(2):261–294, 1979.

[6] G. Filé. The relation of two patterns with comparable languages. In R. Cori and M. Wirsing, editors, *Proceedings of the 5th Ann. Symposium on Theoretical Aspects of Computer Science (STACS 88) (Bordeaux, France)*, LNCS 294, pages 184–192, Berlin, 1988. Springer-Verlag.

[7] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Ann. ACM Symposium on Theory of Computing (San Diego, CA)*, pages 114–118, New York, 1978. ACM, ACM Press.

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

[9] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata* Addison-Wesley Publishing Company, Reading, Massachusetts, 1969.

[10] J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[11] K. P. Jantke. Polynomial time inference of general pattern languages. In M. Fontet and K. Mehlhorn, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (Paris, France)*, LNCS 166, pages 314–325, Berlin, 1984. Springer-Verlag.

[12] K. P. Jantke. Monotonic and nonmonotonic inductive inference of functions and patterns. In J. Dix, K. P. Jantke, and P. H. Schmitt, editors, *Proceedings of the 1st Internat. Workshop on Nonmonotonic and Inductive Logic, (Karlsruhe, Germany)*, LNAI 543, pages 161–177, Berlin, 1991. Springer-Verlag.

[13] T. Jiang, A. Salomaa, K. Salomaa, and S. Yu. Inclusion is undecidable for pattern languages. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of the 20th International Colloquium on Automata, Languages and Programming, ICALP 93 (Lund, Sweden)*, LNCS 700, pages 301–312, Berlin, 1993. Springer-Verlag.

[14] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th Ann. ACM Symposium on Theory of Computing STOC '92 (Victoria, British Columbia, Canada),* pages 318–326, New York, 1992. ACM SIGACT, ACM Press.

[15] M. Kearns and L. Pitt. A polynomial-time algorithm for learning k-variable pattern languages from examples. In *Proceedings of the 2nd Ann. Workshop on Computational Learning Theory, COLT'89 (Santa Cruz, CA)*, pages 57–71, San Mateo, CA, 1989. Morgan Kaufmann.

[16] K.-I. Ko and C.-M. Hua. A note on the two-variable pattern-finding problem. *J. Comput. Syst. Sci.*, 34(1):75–86, 1987.

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

[18] S. Lange and T. Zeugmann. Set-driven and rearrangement-independent learning of recursive languages. *Mathematical Systems Theory*, 29(6):599–634, 1996.

[19] A. Marron and K.-I. Ko. Identification of pattern languages from examples and queries. *Information and Computation*, 74:91–112, 1987.

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

[21] A. Salomaa. Patterns (The Formal Language Theory Column). EATCS Bulletin 54:46–62, 1994.

[22] A. Salomaa. Return to patterns (The Formal Language Theory Column). EATCS Bulletin 55:144–157, 1994.

[23] R. E. Schapire. Pattern languages are not learnable. In M. Fulk and J. Case, editors, *Proceedings of the 3rd Ann. Workshop on Computational Learning Theory, COLT'90 (Rochester, NY)*, pages 122–129, San Mateo, CA, 1990. Morgan Kaufmann.

[24] T. Shinohara. Polynomial time inference of pattern languages and its applications. *Proceedings of the 7th IBM Symposium on Mathematical Foundations of Computer Science*, pages 191–209, 1982.

[25] T. Shinohara. Polynomial time inference of extended regular pattern languages. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, editors, *Proceedings RIMS Symposia on Software Science and Engineering*, pages 115–127, LNCS 147, Berlin, 1983, Springer-Verlag.

[26] T. Shinohara and S. Arikawa. Pattern inference. In K. P. Jantke and S. Lange, editors, *Algorithmic Learning for Knowledge-Based Systems*, LNAI 961, pages 259–291, Berlin, 1995. Springer-Verlag.

[27] T. Shinohara and H. Arimura. Inductive inference of unbounded unions of pattern languages from positive data. In S. Arikawa and A. K. Sharma, editors, *Proceedings 7th International Workshop on Algorithmic Learning Theory, ALT'96, (Sydney, Australia)*, LNAI 1160, pages 256–271, Berlin, 1996, Springer-Verlag.

[28] A. Thue. Über unendliche Zeichenreihen. *Norske Vid. Selsk. Skr. I. Mat. Nat. Kl., Christiana* No. 7, 1-22, 1906

[29] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

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

[31] R. Wiehagen and T. Zeugmann. Ignoring data may be the only way to learn efficiently. *Journal of Experimental and Theoretical Artificial Intelligence* 6(1):131–144, 1994.

[32] T. Zeugmann. Parallel Algorithms. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology* Vol. 21, Supplement 6, pages 223–244, Marcel Dekker Inc., New York and Basel, 1990.

[33] 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*, 1997. to appear.

[34] T. Zeugmann and S. Lange. A guided tour across the boundaries of learning recursive languages. In K. P. Jantke and S. Lange, editors, *Algorithmic Learning for Knowledge-Based Systems*, LNAI 961, pages 190–258, Berlin, 1995. Springer-Verlag.