On Learning Regular Expressions and Patterns Via Membership and Correction Queries

Efim Kinber

Department of Computer Science, Sacred Heart University, Fairfield, CT 06825-1000, U.S.A.
kinbere@sacredheart.edu

Abstract. Based on the ideas suggested in [5], the following model for learning from a variant of *correction* queries to an oracle is proposed: being asked a membership query, the oracle, in the case of negative answer, returns also a *correction* – a positive datum (that has not been seen in the learning process yet) with the smallest edit distance from the queried string. Polynomial-time algorithms for learning a class of regular expressions from one such query and membership queries and learning pattern languages from queries of this type are proposed and discussed.

1 Introduction

In this paper, we propose and discuss some algorithms for learning pattern languages and a class of regular expressions from positive data. There exist different models for learning languages from examples, in particular, the popular Gold's model [9] for learning languages in the limit from a stream of all positive examples, Angluin's model [3,4] for learning languages from queries to a teacher (oracle), and Valiant's PAC learning model [18]. In this paper, we follow the Angluin's query model representing learning process as an interactive session between a learner and a teacher. Specifically, the learner asks the teacher queries of a certain type, and the teacher returns correct answers to the queries. After a finite number of queries, the learner must output a correct description of the target language. This description of the target language must be within certain general class chosen by the teacher and known to the learner (for example, regular expressions, patterns, DFAs, an acceptable numbering of all recursively enumerable sets, etc.)

Over the years, capabilities of different types of queries for various classes of target languages have been explored within the framework of the Angluin's query model. In particular, in her seminal paper [3], D. Angluin presented a polynomial-time algorithm using membership and equivalence queries and learning (minimal) deterministic finite automata. Since then, query model has been used for studies of learning various classes of target concepts via membership, equivalence, subset, superset and disjointness queries. Among the classes of target concepts, context-free grammars ([14]), non-deterministic finite automata ([19]), regular tree languages ([17]), and some others have been used.

A. Clark, F. Coste, and L. Miclet (Eds.): ICGI 2008, LNAI 5278, pp. 125–138, 2008.

Using the general framework of the Angluin's query model, L. Becerra-Bonache, A. H. Dediu, and C. Tîrnăucă introduced in [5] a new type of queries - so called *correction queries*. A correction query is a modification of the most popular and most natural type of queries defined by D. Angluin - membership queries. While in response to a membership query, a teacher just answers 'yes' or 'no', responding to a correction query, the teacher, in case the answer is negative (and, thus, the queried string is not in the target language), provides also a shortest extension of the queried example (string) belonging to the target language. This approach to modeling learning processes stems from the following observation discussed, in particular, in [13]: while learning a language, in addition to *overt* explicit negative evidence (when a parent points out that a certain statement by a child is grammatically incorrect), a child often receives also *covert* explicit evidence in form of corrected or rephrased utterances. The choice of correction in form of a shortest extension of a wrong queried string in [5] and some other works following this line of research (e.g. [16]) has been dictated by the nature of classes of languages being learned. For example, C. Tîrnăucă and T. Knuutila in [16] consider learning so-called k-reversible regular languages (introduced and shown to be learnable in the limit within the Gold's learning model by D. Angluin in [2]). For these languages, where a prefix of a string in a language defines the set of all extensions belonging to the language, an extension of an incorrect string seems to be the most natural type of a "closest" correction. They also apply this model to learning pattern languages. In [6], the authors propose a different type of corrections: the correction string is "closest" to the queried one in terms of shortest *edit distance*; they do not impose any other restrictions on the correction string.

In this paper, we consider learning the class of patterns (introduced by D. Angluin in [1]) and a class of regular expressions using membership queries and a type of correction queries similar, but somewhat different from the one in [6], which is more suitable to the nature of main operations forming strings in the corresponding target languages. Namely, for our class of regular expressions, the main operation forming strings in corresponding languages will be looping, while in patterns such operation is substitution of variables by strings. Accordingly, for the concept classes under consideration in this paper, we consider the following type of correction queries: if the queried string is not in the target language, then the teacher returns the positive example with a smallest edit distance from the queried string and previously not used in the learning process (the requirement of correction being not previously used in the learning process is not a part of the learning model in [6]). For our class of regular expressions, we use one such query to get the shortest string in the target language, and then use membership queries. For learning patterns, we use correction queries only. Within this framework, we suggest polynomial-time algorithms for learning respective classes of concepts.

Learning pattern languages has been a subject of intensive study since the introduction of this concept by D. Angluin in 1979. A comprehensive survey of results in this area, including some interesting practical applications, can be found in [15].

While there is a significant body of work on polynomial-time learning of deterministic finite automata, relatively little has been done on learning regular expressions (practically all articles on this subject are listed in [10]), whereas regular expressions are often more suitable for specifying regular languages by human beings than, say, DFAs ([10]) and more suitable for representing various learning tasks (see, for example, [7] and [8] where learning algorithms for regular expressions have been designed for inference of Document Type Descriptions and XML Schema Definitions). Thus, we hope that this work will contribute to a better understanding of how regular expressions can be efficiently learned.

2 Notation and Preliminaries

Let Σ be a finite alphabet. Σ^* denotes the set of all finite strings (words) over Σ. A *language* is any subset of Σ^*. The length of a string w is denoted by $|w|$. uv denotes the concatenation of the strings u and v. Let ϵ be a character not belonging to Σ. Given any two strings u and v, the *Levenshtein edit distance* $d(u, v)$ is defined as the minimum number of operations needed to transform one string to another, where an operation is insertion, deletion, or substitution of a single character.

By a substring of a string w, we will understand any string u such that $w = vux$ for some strings v and x. For any string v and any number k, let v^k denote the concatenation of k copies of v.

Let $w = a_1a_2 \dots a_k$ be a string. Any string $a_ia_{i+1} \dots a_ka_1a_2 \dots a_{i-1}$ is called a *circular shift* of w. For any string v, the regular expression $(v)^+$ will be called a *loop*, and the string v will be called the *body* of this loop.

3 Learning Via Queries

The classes of languages considered in this paper are examples of so-called *indexable* classes of recursive languages over Σ^*. A class of recursive languages $\mathcal{L}$ is an *indexable* class if there exists an effective numbering $L_i, i = 0, 1, 2, 3, \dots$ of all the languages in $\mathcal{L}$ such that membership in these languages is uniformly decidable (in other words, there is a recursive function that, for any $w \in \Sigma^*$, and any index i, outputs 1 if $w \in L_i$ and 0, otherwise). We consider the following model of learning of an indexed class $\mathcal{L}$: a learner M is an algorithmic device that has access to an oracle that truthfully answers queries of a certain type. Having received an answer for its query, the learner M either generates a new query to the oracle, or returns a conjecture, an index of a language in $\mathcal{L}$, and halts. If the conjecture, say i, is an index of the target language L, then we say that M *learns* the target language L. We say that M learns the class $\mathcal{L}$ if M learns every language in this class. Obviously, for every language in a learnable class, the learner M asks a finite number of queries.

In this paper, we will consider the following types of queries:

Membership queries. The learner asks if a string w belongs to the target language and gets the answer "yes" or "no".

Correction queries. The learner asks if a string w belongs to the target language L. If the answer is negative, the teacher (oracle) returns also a *correction* - a string $v \in L \backslash A$ such that

$$|v| = |w| \text{ and } d(v,w) = min\{d(u,w) | u \in L, u \notin A\}$$

where A is the (finite) set of all strings for which the learner has received the answer "yes" on the previous steps and all the correction strings received by the learner on the previous steps (in other words, the oracle never returns positive examples of the target language seen on the previous steps of the learning process). If no such string v exists, the teacher returns a shortest string of the length different from $|w|$ such that $d(v,w) = min\{d(u,w) | u \in L \text{ and } u \notin A\}$. (In other words, preference is always given to correction strings of the same length, if any). If no such string v exists, the teacher returns $\epsilon \notin \Sigma$. We will also assume that, among the correction strings of the same length, the teacher always chooses the smallest one in the lexicographic order (for patterns, we will consider learning via correction queries for the languages over the binary alphabet $\Sigma = \{0,1\}$ only, and will assume that $0 < 1$, and empty character < 0).

The learning algorithms presented in the next two sections will have running times polynomial in the length of the shortest string in the target language (or, in the length of the target expression).

4 Learning a Class of Regular Expressions

We consider the following regular expressions over the alphabet Σ:

$$u_1(v_1)^+u_2(v_2)^+ \dots u_n(v_n)^+u_{n+1},$$

where $u_i, v_i, i = 1,2,\dots,n$ are strings over Σ (at least one of them must be nonempty). Thus, all loops must be used at least once, unions are not used, and the loop depth is 1 (similar regular expressions for a different learning task were considered in [11]).

An example of such an expression is:

$$a(aa)^+bccd(cd)^+ddabb^+b(abb)^+.$$

A regular expression α in our class is called *left-aligned* if, for any string $v \in \Sigma^+$, there are no segments $v^{m_1}(u_1)^+v^{m_2}(u_2)^+v^{m_k}(u_k)^+$ in α, where each $u_i, i = 1,2,\dots,k$ is v^r for some r, and at least one $m_j > 1$. For instance, the above example of the regular expression is not left-aligned, and it is equivalent to the (unique) left-aligned expression $(aa)^+abc(cd)^+cddab^+bb(abb)^+$.

In other words, in left-aligned expressions, all loops are shifted to the left as much as possible. We will limit expressions in our class to just left-aligned expressions. We will also assume that if, in an expression α in our class, there is a subexpression-loop $\beta = (v^k)^+$ for some (shortest possible) string v and some k, then there is no $(v^m)^+$ neighboring β on the left or on the right in α. For example, subexpressions $((ab)^3)^+((ab)^5)^+$ are not allowed.

Let $R1^+$ denote the class of regular expressions satisfying the above conditions. For any expression α in this class, let $L(\alpha)$ denote language defined by α. We will also use $R1^+$ to denote the class of corresponding languages.

We will often use shifting of loops in regular expressions. Sometimes, in our algorithm, we will use shifting individual loops to the right as much as possible, right-aligning them (and, obviously, preserving equivalence). For example, in the left-aligned expression $(a)^+a(ab)^+ababa(b)^+b$, right-aligning the loop $(ab)^+$ will result in the expression $(a)^+aababab(ab)^+b^+$ (note that the loop $(b)^+$ has also been shifted to the right as the result of right-aligning the loop in question).

4.1 The Learning Algorithm for $R1^+$

We now present an algorithm that learns any language in $R1^+$ using one correction query and membership queries. The algorithm will always output a (unique) left-aligned expression representing the target language.

To make our presentation of the algorithm clear, we will begin with an example (the example uses a two-letter alphabet Σ, however, the algorithm works for expressions over arbitrary finite Σ). Consider the following (left-aligned) expression in $R1^+$:

$$a^+a(ab)^+ab^+(ab)^+ab^+a(bb)^+.$$

Let L denote the target language. On the very first (special) step of the algorithm, ask the correction query for the empty string. Obviously, the teacher will return the shortest string $w = aaababababbb$. Now we switch to the main part of the algorithm. On its first phase, the algorithm tries to find all one-letter loops. It finds all the longest one-letter substrings, listing them in the order they are found in the string w - in our case, they are $aaa, b, a, b, a, b, a, b, a, bb$. Then, for each block, it determines if there are loops associated with it. First, query '$aaaababababbb \in L$?' (using one extra a in the first block). As the answer is 'yes', transform w to $a^+aababababbb$. Using similar queries, the algorithm arrives to to the expression $r_1 = a^+aabab^+abab^+abb$.

Now the algorithm attempts to find all two and more-letter loops. First, the algorithm finds all two-letter loops. As we already have the loop a^+, the algorithm does not attempt to construct the loop $(aa)^+$. Thus, the algorithm will try all three strings ab and the tail bb of r_1 . First, it will query '$aaababababbababbb \in L$?'. Obviously, the answer is 'yes'. Note that the first substring abb in the queried string can be contributed only by (already found) subexpression ab^+, and, thus, the first substring $abab$ must have been contributed by the loop $(ab)^+$ *before* the subexpression ab^+. Note also that the (obvious) query '$aaababababbabbb \in L$?' (simply using the first substring ab twice) would not work, as the substring $abababab$ after the prefix aa could have been contributed by the subexpression $abab^+(ab)^+$ containing the loop $(ab)^+$ *after* the subexpression ab^+. Given the answer 'yes', the algorithm conjectures the expression $a^+a(ab)^+ab^+abab^+abb$, and queries '$aaababbababababbabb \in L$?' to find out if the substring ab after the first loop b^+ is the body of a loop. The answer is 'yes'. Note that, again, the first and the second substrings abb could have been contributed only by the (already

found) subexpressions ab^+, and, thus, the substring $abab$ has been contributed by the loop $(ab)^+$ between them. Thus, our next (intermediate) conjecture is $a^+a(ab)^+ab^+(ab)^+ab^+abb$. Now, using the query '$aababababbbabab \in L$?', the algorithm will try to determine if the last substring ab in w is the body of a loop. The answer is negative, and the last query '$aaababababbbbb \in L$'? will find the last two-letter loop $(bb)^+$. Then the algorithm, in a similar way, will determine that there are no three- or more letter loops, and output the correct expression.

Now we present a formal description of the learning algorithm.

Algorithm

Let L denote the target language.

On the Phase 1, query empty string. Let w be the correction string (obviously, the shortest one in the target language). Now, for any substring v in w, let $w_l(v)$ and $w_r(v)$ denote the strings such that $w = w_l(v)vw_r(v)$.

PHASE 2 (It uses membership queries only):

On the STEP 1, inserting one extra character a at the end of each (maximal) substring $aa \dots a$ in w (same for b), and making a query for the corresponding string (of the length $|w| + 1$), if the answer is 'yes', replace the first character of the substring by the loop $(a)^+$ (keeping the resulting exression, denoted r_1, left-aligned).

If there are no substrings of the type ab (after, possibly, shifting a loop $(b)^+$ to the right) in r_1, then terminate and output r_1 as the target expression. Otherwise, go to STEP 2.

STEP n: Let $r = r_{n-1}$ be the regular expression obtained on the step $n - 1$. By induction, we assume that r is left-aligned. Let

$$r = (\alpha_1)^+\beta_1(\alpha_2)^+\beta_2 \dots (\alpha_k)^+\beta_k$$

for some k and some $\alpha, \beta \in \Sigma^*$.

On each iteration j of the FOR loop below, the algorithm will be constructing loops between (possibly shifted to the right on the previous iteration) the loop α_j^+ and the right-aligned loop α_{j+1}^+. Let $r^0 = r$.

FOR $j = 1, 2, \dots k$:

Step j:

Let t be the expression obtained from r^{j-1} (the expression from $j-1$ iteration of the FOR-loop) and by right-aligning α_{j+1}^+ as much as possible.

Let u be the substring in t between the loop preceding α_{j+1}^+ and α_{j+1}^+. If $|u| < n$ (thus, there is not enough space for another loop between two neighboring loops), then let $r^j = r^{j-1}$ and go to the next iteration of the FOR loop. Otherwise, let $u = a_1a_2 \dots a_p$ for $a_1, a_2, \dots, a_p \in \Sigma$.

In the WHILE loop below, the algorithm will attempt to construct loops based on the substrings in u (with the length of the body n), thus transforming t. Let $Tail = u$.

WHILE ($|Tail| \geq n$) DO

Let x be the prefix of $Tail$ with the length $|x| = n$. Let $x = v^i$ for some (shortest) v. If x is preceded in t by the loop $(v^k)^+$ for some k,

then remove from the $Tail$ the shortest prefix such that the prefix of the length n of the remaining string is neither x nor $(z)^i$ for some circular shift z of v. Let $Tail$ be the remaining string. (For example, if $n = 4$, $Tail$ is *abababababcccc* and preceded in t by $(ab)^+$, then $Tail$ is set to *babcccc*; if *abababababacccc* is preceded by the same loop, then $Tail$ is set to *abacccc*). Go to the top of the WHILE loop.

If x is not preceded by any such $(v^k)^+$, then the goal is to determine if x is the body of a loop in the target expression. Let z be the longest extension of x in $Tail$ equal to v^k for some k.

Consider all loops $(\gamma)^+$ to the right from z in the expression t. Let t' denote the expression obtained from t by substituting each loop $(\gamma)^+$ by the string γ. Let z' be the longest extension of z in t', of which z is a prefix, such that $z' = v^m$ for some m. Now substitute all loops $(\delta)^+$ to the left from $Tail$ in t by δ, and let t'' be the corresponding expression obtained from t. Let z'' be the longest substring in t'' extending $Tail$ to the left such that $z'' = v^m$ for some m.

Now the following cases are possible:

Case 1: $z' = z$ and $z'' = z$. Query '$w_l(x)xxw_r(x) \in L$?'.

Case 2: $|z'| > |z|$ and $z'' = z$. Then, there is a substring γ in $w_r(x)$, which is the body of the leftmost loop $(\gamma)^+$ mentioned above. Replace this γ in $w_r(x)$ by $\gamma\gamma$, and let w' be the string obtained from $w_r(x)$. Query '$w_l(x)xxw' \in L$?'.

Case 3: $z' = z$ and $|z''| > |z|$. Then, there is a substring δ in $w_l(x)$ which is the body of the rightmost loop $(\delta)^+$ mentioned above. Replace this δ in $w_l(x)$ by $\delta\delta$, and let w'' be the string obtained from $w_l(x)$. Query '$w''xxw_r(x) \in L$?'.

Case 4: $|z'| > |z|$ and $|z''| > |z|$. Then, as in cases 2 and 3, replace γ by $\gamma\gamma$ in $w_r(x)$, getting w', and replace δ by $\delta\delta$ in $w_l(x)$, getting w''. Now query '$w''xxw' \in L$?'.

In all four cases, if the answer is 'yes', substitute x in t by $(x)^+$, remove the prefix x from $Tail$, and go to the top of the loop. If the answer is 'no', then remove from $Tail$ the longest prefix $(x)^i y$, where $x = yz$ for some y and z, so that the prefix of the remaining string is not zy (for example, if $x = abb$ and $Tail$ is *abbabbabbaccc*, then the $Tail$ becomes *baccc*, rather than *bbaccc*); go to the top of WHILE loop.

EndWhile

Left-align all loops in the current expression t. Set $r^j = t$. End step j.

END STEP n.

Return r_k for $k = |w|$ as the target expression.

4.2 Correctness of the Algorithm

Correctness of the algorithm is proved by induction. By induction, let us assume that all loops that have been created before some iteration of the WHILE loop on Step j of the FOR loop on STEP n are correct. Now, we will show that, on the

given interation of the WHILE loop, either a new loop cannot be created, or, if it is created, it is correct. Consider the prefix x of $Tail$ of the length n (as defined in the WHILE loop). As defined in the WHILE loop, let $x = v^i$ for the shortest possible v and some i. Consider the case when the loop to the left from x in the current expression t is $(v^k)^+$ for some k. Then x cannot be the body of a loop in the target expression, and $Tail$ is reduced appropriately (so that copies of x or its circular shifts following the given x would not be tested as bodies of possible loops later). In all four Cases (as defined in the WHILE loop), if the answer is 'no', then x obviously cannot be the body of a loop. Now, we will assume that the answer is 'yes'. In the *Case 1*, the second x in the query can be contributed to the string only by the loop corresponding to the first x (as the target expression is left-aligned). Now consider the remaining three cases and the strings γ and δ as defined in the WHILE loop. In the *Case 2*, γ (the body of the leftmost loop to the right from x) cannot be equal to any v^e or y^e for some circular shift y of v, as, otherwise, the corresponding loop would have been shifted to the left from x (note that every expression t in the loop WHILE is left-aligned). Thus, neither the string with the prefix x obtained when u is extended by substituting γ by $\gamma\gamma$, nor any its extension to the right in t can be equal to v^p for any p. Therefore, if the answer to the query in *Case 2* using xx is 'yes', the second x can be contributed by neither the loop $(\gamma)^+$, nor a loop to the right from it in the target expression. Now, in the *Case 3*, δ (the body of the rightmost loop to the left from x) cannot be equal to v^e (or its cicular shift), since, otherwise, x could not be a prefix of $Tail$. Thus, if the answer to the query in this case is 'yes', the second x cannot be contributed by the loop $(\delta)^+$ or a loop to the left from it, and, similarly, by neither $(\gamma)^+$, nor $(\delta)^+$, nor loops to the right from the former, or to the left from the latter, respectively, in the *Case 4*.

4.3 Complexity of the Learning Algorithm

The total number of queries asked is obviously $O(n^3)$, as on each step of the WHILE loop, the algorithm makes just one query. On each step of the WHILE loop, the algorithm performs some work that requires time $O(n^2)$. Thus, overall complexity of the algorithm is $O(n^5)$.

4.4 Modifications of the Class $R1^+$

It would be interesting to explore if some modifications of the class $R1^+$ were learnable in polynomial time. One such modification is the class $R1$ that contains loops $(v)^*$ rather than $(v)^+$. However, this class may be too hard to learn in polynomial time. A modification of this class, $R2$, would contain only those loops $(v)^*$ where the body v did not contain any repetitions (for example, $v = abb$ would not be allowed). We can exhibit a polynomial-time learning algorithm using one correction query and membership queries for the following limited version $R2_1$ of $R2$: one-letter loops are allowed only, and there is a nonempty string between any two loops. An example of such a regular expression is $a^*aab^*ac^*ca^*aa$. $R2_1$ is a subclass of the class of regular expressions considered in [10] (where unions

are allowed), however, the algorithm in [10] learns regular expressions in the limit, while in our model, the first conjecture must be correct. We omit details due to limitations on the size of the paper.

Another approach to learning modifications of $R1^+$ containing loops $(v)^*$ would involve variants of correction queries, or both membership and correction queries.

In, probably, the most recent paper on learing regular expressions [8], the authors define an interesting (and practically useful) class of regular exressions and design polyomial-time algorithms for inference of the expressions in this class from positive data. All expressions in [8] must be *deterministic* (or *one-unambiguous*) and cannot have more than a uniformly bounded number of occurrences of each alphabet symbol. Our class $R1^+$ does not have these restrictions.

5 Learning Pattern Languages Via Correction Queries

Learning patterns has been a subject of intensive study since their introduction by D. Angluin in [1]. A *pattern* π over a finite (in our case - binary) alphabet Σ and a countable infinite set $X = \{x_1, x_2, \ldots\}$ of variables is a (nonempty) string in $(\Sigma \cup X)^*$. An example of a pattern is $x_1x_1x_201x_1x_30$. The (non-erasable) *pattern language* $L(\pi)$ consists of all strings obtained by substituting the variables in the pattern π by arbitrary strings in Σ^+. For example, substituting x_1 by 01, x_2 by 00, and x_3 by 1 in the above example, we get the string 010100010110.

Different authors used different paradigms of learning to study learnability of the class of pattern languages. Among the latest works on this topic, is the paper [16], where patterns are being efficiently learnt from correction queries, and, in case of the answer 'no', the teacher returns the shortest *extension* u of the queried string v belonging to the target language. Unlike their approach, our algorithm for learning pattern languages uses the type of queries introduced in Section 3.

5.1 The Learning Algorithm

Let L be the target language. First, we present our algorithm on two examples. Our first example is the pattern $\pi = x_101x_2x_3x_3x_3x_3x_3x_3x_3x_3x_4x_4x_4x_5x_5x_5$. Let $\pi(r), 1 \leq r \leq |\pi|$, denote the r-th character in π (either a variable, or a constant).

First, query the empty string. The oracle will obviously return the correction string $w = 0010^{14}$ (of total length 17). Now query '$1^{17} \in L$?' . The answer is 'no', and the oracle returns the correction string 101^{15}. Now we know that $\pi(2) = 0, \pi(3) = 1$, and all other $\pi(r)$ are variables. Our goal now is to find these variables. First, query '$1010^{14} \in L$?', trying 1 for the first position. The answer is 'yes'. The algorithm sets $\pi(1) = x_1$. Now, query '$00110^{13} \in L$?', trying 1 for the fourth position. The answer is 'yes', and the algorithm sets $\pi(4) = x_2$. Now, query '$001010^{12} \in L$?', trying 1 for the 5-th position. The answer is 'no', and the correction string is 10110^{13} (note that strings 1010^{14} and 00110^{13} are closer to the queried string, however, both of them have already been used). Now query '$101110^{13} \in L$?', trying 1 for the 5-th position again.

The answer is 'no' again, and this time the (smallest in lexicographic order) correction string is $10110^{10}111$. As the tail 111 has never appeared, the algorithm sets $\pi(15) = \pi(16) = \pi(17) = x_3$ (the numbers of variables in the output of the algorithm can obviously be different from the ones in the original pattern). Now, query '$101110^9111 \in L$?', trying 1 for the 5-th position once again. This time the correction string is 10111111111000111, and the algorithm sets $\pi(r) = x_4$ for $5 \leq r \leq 11$. Now, it queries '$0010^8100000 \in L$?', trying 1 for the 12-th position. The answer is 'no' and the correction string is $0010^8111000$. The algorithm sets $\pi(12) = \pi(13) = \pi(14) = x_5$. As all variables have been found, the algorithm returns π as the target pattern.

Our next example is the pattern $\pi = 10x_1x_1$. Using first two queries (and getting correction strings 1000 and, respectively, 1011), the algorithm will find constant characters 1 and 0. Then it will query '$1010 \in L$?'. The answer is 'no', but the oracle must return a string u with the length $|u| > 4$, as all the strings in L of the length 4 (in particular, 1011) have already been seen. In this case, the algorithm sets $\pi(r) = x_1$ for $r \in \{3, 4\}$ and terminates.

Now we give a description of the learning algorithm.

Algorithm

Query if the empty string is in L. The teacher will return a string $w = a_1a_2 \dots a_n$. Note that all characters 1 in this string must be constants, while all values 0 are either constants, or correspond to occurrences of variables, as the string w must be the smallest in the lexicographic order among all the shortest strings (of the length n) in the target language. If there are no 0 in w, the algorithm returns the result 1^n and terminates.

Otherwise, first consider the special case of $w = 0$. Query '$1 \in L$?'. If 'yes' then set $\pi = x_1$, if 'no', set $\pi = 0$ and terminate the algorithm.

Now consider the case $|w| > 1$. Our goal is to determine which characters a_i must be replaced by variables, and which are constants 0 (constants 1 have already been determined). Query '$1^n \in L$?'. If the answer is 'no' (thus, the target pattern contains some constants 0) , let $u = b_1b_2 \dots b_n$ be the correction string returned (note that it will have the same length n as the queried string). If the answer is 'yes', let $b_r = 1, 1 \leq r \leq n$.

Now the algorithm enters the FOR loop executed for $j = 1, 2, \dots n$. In this loop, let π_i denote the pattern output at the end of the step i. In π_i, let $\pi_i(j)$ denote the symbol on the j-th position in π_i (either a variable, or a constant). For all $j \in \{1, 2, \dots, n\}$, we set $\pi_0(j) = 1$ if $a_j = 1$, and $\pi_0(j) = 0$, otherwise.

Now we describe the step j of the loop. Let $\pi = \pi_{j-1}$. If either $\pi(j) = 1$, or $\pi(j) = b_j = 0$, or $\pi(j)$ is a variable, then the corresponding symbol in the target pattern is a constant or an already found variable. Thus, set $\pi_j = \pi$ and go to the step $j + 1$ of the loop (or terminate if $j = n$).

Otherwise, we have $\pi(j) = 0$ and $b_j = 1$. This means that there must be a variable on the position j in the target pattern. If all symbols $\pi(r)$ for $r > j$ are already variables or constants 1, then just set $\pi_j(j)$ to a new variable, set $\pi_j(r) = \pi(r)$ for all $r \neq j$ and terminate the loop. Otherwise, let α be the string obtained from π by substituting $\pi(j)$ by 1 and all occurrences of variables by 0.

Query '$\alpha \in L$?' (note that α contains at least one symbol 0 on a position $s > j$ where $b_s = 1$, and, thus, it is not u or 1^n seen before). If the answer is 'yes', then set $\pi_j(j)$ to a new variable, set $\pi_j(r) = \pi(r)$ for all other r and go to step $j+1$.

If the answer is 'no' (this means that there must be other occurrences of the same variable as on the position j in the target pattern), then let β be the correction string. If $|\beta| > n$ (thus, all corrections of the length n have already been seen), then set $\pi_j(j)$ and all other $\pi_j(r)$ for r such that $\pi(r) = 0$ and $b_r = 1$ to a new variable (same for all such r) and terminate the algorithm.

Otherwise, let $\beta = c_1 c_2 \dots c_n$. The algorithm enters the following WHILE loop that runs until c_j in the last correction string β becomes 1. On each step of this loop,

(1) set all $\pi_j(r)$ for r such that $\pi(r) = 0$ and $c_r = 1$ to a new variable; if $c_j = 1$, terminate the WHILE loop.

(2) replace c_j by 1 in β, and let γ be the modified string; query '$\gamma \in L$?' (note that the answer is 'no' - otherwise, the answer 'yes' would have been given earlier). Let β be the correction string. If $|\beta| > n$, then, as in the corresponding case above, substitute all $\pi(r) = 0$ for which $b_r = 1$ by a new variable and terminate the algorithm. Otherwise, let $\beta = c_1 c_2 \dots c_n$. Return to the top of WHILE loop.

Once the loop WHILE has terminated, go to step $j+1$ of the FOR loop.

Return the pattern π_j created on the last executed STEP j of the FOR loop.

5.2 Correctness of the Algorithm

By induction on j, let us assume that the initial segment $\pi_{j-1}(r), r \leq j-1$, of π_{j-1} and all variables and constants 1 among $\pi_{j-1}(r)$ for $r > j$ of the target pattern being constructed, have been constructed correctly. Now, we will show that the same is true for π_j .

Obviously, the only interesting case is when α is queried as described in the algorithm. Note that this α has not been used yet in the learning process: the only strings used before in the learning process and having 1 on the position j were 1^n and, possibly, the correction string u (containing 1-s for all variables), however, as it is pointed out in the description of the algorithm, α is neither u, nor 1^n. If the answer is 'yes', then substituting just $\pi_{j-1}(j) = 0$ by the single occurrence of a new variable is obviously correct. Now suppose the answer is 'no'. Then there must be some new variable (not used before), say x, on the position j, however, there are some other occurrences of this variable in the target pattern, and we don't know them yet. Now, suppose the correction string β has the length greater than n. If some 0 on a position $r > j$ in π_{j-1} for which $b_r = 1$ is not a value of x, then the oracle should have returned some string $\beta \in L$ with 1 on the position j and 0 on the position r, as such a string has not been seen yet. However, it returned a longer string. Therefore, all values 0 on the positions $> j$ in π_{j-1} where $b_r = 1$ are values of the same variable x and, thus, π_j is the correct (final) target pattern.

If $|\beta| \leq n$, then the algorithm enters the WHILE loop. In the case (1), the algorithm observes the correction string, where some already existing variables have been replaced by 1, and, possibly 0-s on the positions $> j$ in π; as the correction

string is at the shortest distance from the queried one, all these replaced 0-s (if any) must be the values of the same variable. Thus, the algorithm is obviously correct, having replaced all c_r in question by occurrences of a new variable.

In the case (2), if $|\beta| > n$, then the analysis is as above. Otherwise, the algorithm returns to the case (1) with the new correction string.

Now note that, on some iteration of the WHILE loop, the oracle must return either a correction string β with $|\beta| = n$ and $c_j = 1$ (as c_j is a value of a variable, and, since oracle each time returns a string of the length n not seen yet, some correction string of the length n must have $c_j = 1$), or with $|\beta| > n$. Thus WHILE loop always terminates, whenever entered.

Therefore, the algorithm correctly learns the target pattern π.

5.3 Complexity

The WHILE loop runs at most $O(n)$ times, as every correction string contains more characters 1 than the one on the previous step. Creating (possibly) a new vaiable in the body of this loop requires time $O(n)$. Thus, the total running time of the algorithm comes to at most $O(n^3)$. The total number of queries in the WHILE loop is $O(n)$, and, thus, the total number of queries is $O(n^2)$.

5.4 Discussion

Two distinctive features of the oracle in our model are that a) it returns a correction not previously seen, and b) it gives preference to corrections of the size equal to the size of the queried string. Both of them are important for the success of our algorithm. If the oracle may return previously seen corrections, then, in many cases, it will keep providing the string that has been utilized, and the learner will not be able to get necessary information about unknown variables. Also, corrections of arbitrary length (even closest to the queried string) are of little help (if a returned string is longer than the pattern being constructed, our algorithm just uses this fact to finish its work, but not the correction string itself). The fact that positive examples of arbitary length are not helpful for learning patterns when the shortest string in the target language becomes available, was noticed (and successfully utilized) yet in [12], where the learning algorithm, while learning patterns in the limit from all positive data, simply ignores all examples longer than the shortest positive example seen so far. Our algorithm is similar to the algorithm in [12] in this respect - however, it gets (and, accordingly, utilizes) input data differently.

It is possible to slightly modify our query model, leaving the learning algorithm intact. Instead of using requirement of selecting lexicographically smallest correction string, we could use correction strings that contain largest number of 0-s among the ones at the shortest edit distance from the queried string (still giving preference to correction strings of the length equal to the length of the queried string).

As it was shown in [16], patterns cannot be learned in polynomial time using membership queries. Thus, corrections are essential for polynomial-time learnability.

There are multiple ways of relaxing constraints of our query model. First, within the framework of the given model, it would be interesting to find out if polynomial-time learnability can be preserved for patterns over arbitrary alphabets Σ. Another interesting question is if the requirement of preference given to correction strings smallest in the lexicographic order (or containing largest number of 0-s) can be lifted. Yet another interesting case would be when a correction string were just at the shortest edit distance from the queried one - without any other constraints. However, we conjecture that a polynomial time algorithm could not exist in this case. One more interesting open problem is to find out if polynomial-time learning of patterns is possible while dropping the requirement that correction string must be selected among those not seen so far in the learning process (in this case, selection of the correction string will not depend on the learning algorithm, and the teacher will not need to remember which strings have been used in the learning process). Yet another interesting open problem is whether, within the framework of our model, polynomial-time algorithms exist for patterns over alphabets of the size 3 or more.

Acknowledgments. The author is grateful to C. Tîrnăucă for a useful discussion and to anonymous referees for several helpful comments and suggestions.

References

1. Angluin, D.: Finding Patterns Common to a Set of Strings (extended abstract). In: 11th Annual ACM Symposium on Theory of Computing, pp. 130–141. ACM Press, New York (1979)
2. Angluin, D.: Inference of Reversible Languages. Journal of the ACM 29(3), 741–765 (1982)
3. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75(2), 87–106 (1987)
4. Angluin, D.: Queries and Concept Learning. Machine Learning 2, 319–342 (1988)
5. Becerra-Bonache, L., Dediu, A.H., Tîrnăucă, C.: Learning DFA from Correction and Equivalence Queries. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 281–292. Springer, Heidelberg (2006)
6. Becerra-Bonache, L., de la Higuera, C., Janodet, J.C., Tantini, F.: Learning Balls of Strings with Correction Queries. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) ECML 2007. LNCS (LNAI), vol. 4701, pp. 18–29. Springer, Heidelberg (2007)
7. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of Concise DTDs from XML Data. In: 32nd International Conference on Very Large Data Bases VLDB (2006)
8. Bex, G.J., Gelade, W., Neven, F., Vansummeren, S.: Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data. In: WWW Conference 2008, Beijing, China, pp. 825–836 (2008)
9. Gold, E.M.: Language Identification in the Limit. Information and Control 10, 447–474 (1967)

10. Fernau, H.: Algorithms for Learning Regular Expressions. In: Jain, S., Simon, H.U., Tomita, E. (eds.) ALT 2005. LNCS (LNAI), vol. 3734, pp. 297–311. Springer, Heidelberg (2005)
11. Kinber, E.: Learning a Class of Regular Expressions via Restricted Subset Queries. In: Jantke, K.P. (ed.) AII 1992. LNCS, vol. 642, pp. 232–243. Springer, Heidelberg (1992)
12. Lange, S., Wiehagen, R.: Polynomial-time Inference of Arbitrary Pattern Languages. New Generation Computing 8(4), 361–370 (1991)
13. Rohde, D.L.T., Plaut, D.C.: Language Acquisition in the Absence of Explicit Negative Evidence: How Important Is Starting Small? Cognition 72, 67–109 (1999)
14. Sakakibara, Y.: Learning Context-free Grammars from Structural Data in Polynomial Time. Theoretical Computer Science 76, 223–242 (1990)
15. Shinohara, T., Arikawa, S.: Pattern Inference. In: Lange, S., Jantke, K.P. (eds.) GOSLER 1994. LNCS, vol. 961, pp. 259–291. Springer, Heidelberg (1995)
16. Tîrnăucă, C., Knuutila, T.: Polynomial Time Algorithms for Learning k-reversible Languages and Pattern Languages with Correction Queries. In: Hutter, M., Servedio, R.A., Takimoto, E. (eds.) ALT 2007. LNCS (LNAI), vol. 4754, pp. 264–276. Springer, Heidelberg (2007)
17. Tîrnăucă, C.I., Tîrnăucă, C.: Learning Regular Tree Languages from Correction and Equivalence Queries. Journal of Automata, Languages and Combinatorics 12 (2007)
18. Valiant, L.G.: A Theory of the Learnable. Communications of the ACM 27(11), 1134–1142 (1984)
19. Yokomori, T.: Learning Non-deterministic Finite Automata from Queries and Counterexamples. Machine Intelligence 13, 169–189 (1994)