Text matching of strings in terms of straight line program by compressed aleshin type automata

[1]A.Jeyanthi, [2]B.Stalin
[1] Faculty, [2] Assistant Professor
[1] Department of Mathematics, [2] Department of Mechanical Engineering
[1, 2] Anna University, Regional Office Madurai,Tamilnadu, India

***Abstract* - In this paper we are checking the equivalence of any given text of strings is represented by a straight line program (SLP) with model text. For a given SLP-compressed Aleshin type automata D of size n and height h representing m patterns of total length N, we present an O (n^2 log N)-size representation of Aho-Corasick automaton which recognizes all occurrences of the patterns in D in amortized O (h + m) running time per character. We also propose an algorithm to construct this compressed Aho-Corasick automaton in O (n^3 logn log N) time and O (n^2 log N) space. In a special case where D represents only a single pattern, we present an O (n log N)-size representation of the Morris-Pratt automaton which permits us to find all occurrences of the pattern in amortized O (h) running time per character, and to construct this representation in O (n^3 logn log N) time with O (n^2 log N) working space.**

***Index Terms* - Aho-Corasick automata, straight line program, Morris-Pratt automaton, Aleshin Type Automata.**

I. INTRODUCTION

In this paper, we introduce a new notion of the pattern is given in compressed form while the text is given in uncompressed form. In particular, we are interested in a setting where a set of patterns is given in compressed form in advance, and the text is given in a streaming fashion. A typical application would be an SDI (Selective Dissemination of Information) service.

A straight line program (SLP) is a context-free grammar in the Chomsky normal form which generates a single string. It is well known that outputs of various grammar-based compression algorithms (e.g., [3,4]), as well as those of dictionary-based compression algorithms (e.g., [5-8]), can be regarded as, or be quickly transformed to, SLPs [9]. We use an SLP to represent a dictionary consisting of m patterns, by designating m variables in the SLP as the start symbols. The classical pattern matching problem is, given two strings called the pattern and the text, to find all occurrences of the pattern within the text. The full compressed pattern matching problem [1] is the pattern matching problem where both the pattern and the text are given in compressed form. A variant of this problem where the text is given in compressed form while the pattern is given in uncompressed form, has been extensively studied for various compression format [2] .Given a compressed automata D represented as an SLP of size n, we consider how to efficiently construct an AhoCorasick (AC) automaton [10] for D. Since the total length N of patterns in D can be as large as Θ (2n), a naïve method which decompresses D takes exponential time and space in the worst case. By exploiting some combinatorial properties on SLP-compressed automata, to present a compressed representation of AC automaton which requires O (n^2 log N) space. Hence, our representation is useful when the patterns in the automata are compressible. This representation allows us to recognize all occurrences of the patterns in D in amortized O (h + m) running time per character, where h is the height of the derivation tree of the SLP representing D, and m is the number of patterns in D. We also show how to construct our compressed Aleshin type AC automata in O (n^3 logn log N) time using O (n^2 log N) space. The size of our compressed Aleshin type AC automaton is independent of the number m of patterns represented by the compressed text D, and hence it requires O (n^2 log N) space even if D represents only a single pattern. We also present a more space-efficient solution to the case of a single pattern, namely, a compressed representation of Morris-Pratt (MP) automaton [11] which requires O (n log N) space. The compressed MP automaton permits us to perform pattern matching on a compressed pattern and an uncompressed text in amortized O (h) time per character, and can be constructed in O (n^3 logn log N) time using O (n^2 log N).

II. PRELIMINARIES

Strings

Let Σ be a finite alphabet. An element of Σ ∗ is called a string. The length of a string w is denoted by |w |. The empty string ε is a string of length 0, namely, |ε| = 0. Strings x, y and z are, respectively, called a prefix, substring, and suffix of the string w = xyz. Let Prefix(w) denote the set of prefixes of w . A prefix (suffix) of a string w is said to be proper if it is shorter than w . The i -th character of a string w is denoted by w [i], where $1 \le i \le |w|$. For a string w and two integers i , j with $1 \le i \le j \le |w|$, let w [i .. j] denote the substring of w that begins at position i and ends at position j , that is, w[i.. j] = w [i] · · · w [j].

For any strings p, t ∈ Σ + , let Occ(p, t) denote the set of all positions of t at which an occurrence of P begins, that is, Occ(p, t) = {k | k ∈ [1..|t | − |p| + 1], p = t [k..k + |p| − 1]}.

Periods and runs of strings

A period of a string w is a positive integer p such that w [i] = [i + p] for every i ∈ [1..|w | − p]. A run in a string w is an interval [i .. j] with 1 ≤ i ≤ j ≤ |w | such that:

(i) the smallest period p of w [i.. j] satisfies 2 p ≤ j − i + 1.

(ii)the interval can be extended neither to the left nor the right, without violating the above condition, that is, w [i − 1] ≠ w[i + p − 1] and w [j − p + 1] ≠ w [j + 1], provided that respective symbols exist.

Construction of 4-state Aleshin type automaton, A(S)

The constructed Aleshin type automaton S, [A(S)] over the alphabet X= {0, 1} with the set of internal states Q ={a, b, c, d}. The state transition function φ and the output function ψ of A(S) are defined as follows: δ (a,0)=d, δ (a,1)=b,

δ (b,0)=b, δ (b,1)=c, δ(c,0)=d, δ(c,1)=d, δ(d,0)=a, δ(d,1)=a ; ψ(a,0)=1, ψ(a,1)=0, ψ(b,0)=1, ψ(b,1)=0, ψ(c,0)=0, ψ(c,1)=1, ψ(d,0)=0, ψ(d,1)=1.

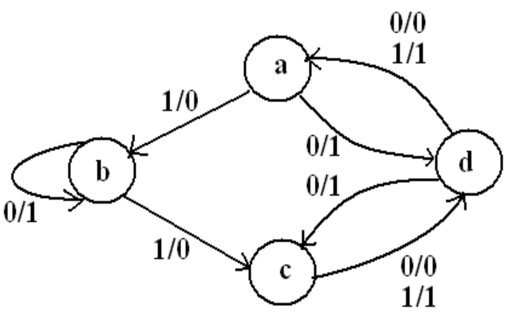

Fig. 1 Aleshin type automaton

Theorem 1

Let p and q be two periods of a string x. If p + q − gcd(p, q) ≤ |x|, then gcd(p, q) is also a period of x.

Theorem 2

The periods of any x ∈ Σ^+ are partitioned into O (log |x|)-arithmetic progressions.

Aho-Corasick automata

The Aho-Corasick automaton (AC automaton for short) [10] is a finite state machine which simultaneously recognizes all occurrences of multiple patterns in a single pass through a text. The AC automaton for a dictionary Π consists of three functions: goto, failure, and output.

The g-trie for a dictionary Π is a trie representing Π . There is a natural one-to-one correspondence between the states (nodes) of the g-trie and the pattern prefixes. State q is said to represent string u if the path from the initial state 0 to q spells out u . For example, the initial state 0 represents the empty string ε and the state 4 represents the string abab . Let Q denote the set of states of the g-trie, and let ⊥ be an auxiliary state not in Q . The g-trie defines the goto function g so that every edge q to r labeled c implies g (q, c) = r . In addition, we set g (⊥, a) = 0 for all a ∈ Σ . The output function λ and the failure function f are defined as follows.

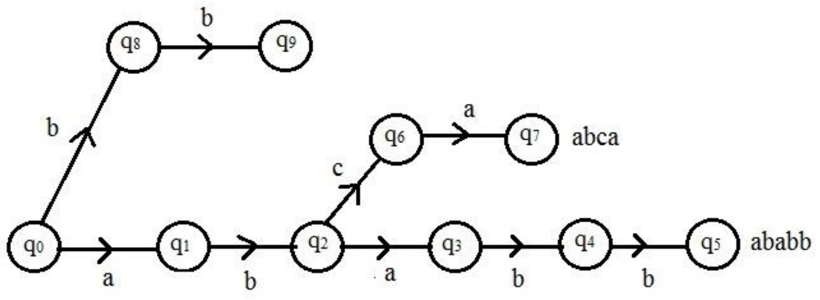

Fig. 2 On the left the Aho-Corasick automaton for Π = {aba, ababb, abca, bb} is displayed, where the circles denote states, the solid and the broken arrows represent the goto and the failure functions, respectively, and the underlined strings adjacent to states mean the outputs from them. On the right the g-trie for Π is shown

Definition 1

Let q be any state. Suppose q represents string u . Then λ(q) is the set of patterns in Π that are suffixes of u.

Definition 2

Let q be any state with q ≠ 0. Suppose q represents string u . Then state f (q) represents the longest proper suffix of u that is also a prefix of some pattern.

Let $\delta : Q \times \Sigma \rightarrow Q$ be the state-transition function defined by:

$$\delta(q, a) = \begin{cases} g(q,a), \text{ if } g(q, a) \text{ is defined} \\ \delta(f(q), a), \text{ otherwise.} \end{cases}$$

We extend δ to the domain $Q \times \Sigma^*$ in the standard way.

Theorem 3 *[10]*

For any string $w \in \Sigma^*$, $\delta(0, w)$ is the state that represents the longest suffix of w that is also a prefix of some pattern. The number of goto and failure transitions required in computing $\delta(0, w)$ is at most $2|w|$.

We say that a state is branching if it is of out-degree ≥ 2, and terminating if it represents some pattern. We say that a state is explicit if it is branching or terminating, and implicit otherwise.

Theorem 4

The number of explicit states is at most $2|\Pi|$.

Straight line programs

A straight-line program (SLP) is a set of assignments $D = \{X_1 \rightarrow expr_1, X_2 \rightarrow expr_2, \ldots, X_n \rightarrow expr_n\}$, where each X_i is a variable and each expr i is an expression, where expri = a ($a \in \Sigma$), or expri = $Xl_{(i)}\, Xr_{(i)}$ ($i > l(i)$, $r(i)$). It is essentially a context-free grammar in the Chomsky normal form, that derives a single string. Let val(Xi) denote the string derived from variable Xi . To ease notation, we sometimes identify val(Xi) with Xi , and denote |val(Xi)| as |Xi |, and val(Xi)[b..e] as Xi [b..e] for any interval [b..e]. An SLP D represents the string s = val(Xn). The size of D, denoted by |D|, is the number n of assignments in D. Note that N = |s| can be as large as $\Theta(2^n)$.

Our model of computation is the word RAM: We shall assume that the computer word size is at least $\log_2 N$, and hence, standard operations on values representing lengths and positions of string s can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

We will use the following result.

Theorem 5 *[15]*

We can pre-process an SLP D = {Xi → expri }n=1 in O (n3) time and O (n^2) space to answer the following query in O (n^2) time: given two variables Xi and X j ($1 \leq i, j \leq n$), compute the length of the longest common prefix of val(Xi) and val(X j).

The derivation tree of an SLP D = {Xi → expri }n=1 is a labeled ordered binary tree where each internal node is labeled with a non-terminal variable in {X_1, . . . , Xn }, and each leaf is labeled with a terminal character in Σ . The root node has label Xn . Let height(Xi) denote the height of derivation tree of Xi , and let height(D) = height(Xn).

For each variable Xi we store the length |Xi | of the string derived by Xi , which can be computed in a total of O (n) time using O (n) space by a simple dynamic programming algorithm.

The sorted index of an SLP D = {Xi → expri }n=1 is the permutation σ of [1..n] such that the strings val($X_{\sigma(1)}$), . . . , val($X_{\sigma(n)}$)) are arranged in the lexicographical order.

Theorem 6

The sorted index σ of an SLP of size n can be computed in O (n^3 logn) time using O (n^2) space.

Proof

We compute the length l of the longest common prefix of two variables Xi and X j using Theorem5. Then, comparing val(Xi) and val(X j) reduces to comparing the (l + 1)-th leaves of the derivation trees of Xi and X j , which can be done in

The derivation tree of SLP $D = \{X_1 \rightarrow a, X_2 \rightarrow b, X_3 \rightarrow X_1 X_2, X_4 \rightarrow X_1 X_3, X_5 \rightarrow X_3 X_4, X_6 \rightarrow X_4 X_5, X_7 \rightarrow X_6 X_5\}$, representing string val(X_7) = aababaababaab.

O (n) time using the length of the string that each variable derives (note that the case where = min{|Xi |, |X j |} is easier). Hence the sorted index σ can be computed in O ($n^3 + n^3$ logn) = O (n^3 logn) time using any O (n logn)-time comparison sort.

A variable X with Xi → Xl Xr ∈ D is said to stab an interval [b..e] ⊆ [*1*..|Xi |] if b ∈ [1..|Xl |] and e ∈ [|Xl | + 1..|Xi |]. For any p ∈ Σ + , let Occ(p, Xi) denote Occ(p, val(Xi)), and let Occξ (p, Xi) be the set of positions α ∈ Occ(p, Xi) such that the interval [α ..α + |p| − 1] is stabbed by Xi.

Theorem 7 *[15]*

Occξ (p, Xi) forms an arithmetic progression. We will also use the following result:

Theorem 8 *[16]*

Given an SLP $D = \{Xi \rightarrow expri\}^n_{i=1}$ that represents a string T of length N , it is possible to pre-process D in O (n) time using O (n) space, so that any substring T [i ..i + m − 1] of length m of T can be computed in O (log N + m) time.

III. Compressed Aleshin type AC automaton

First, we show that the image set of each lattice-valued regular langauge is always a finite set of *l*.

AC Automaton

In this section, we consider the AC automaton for $\Pi D = \Pi_{(D,n)} = \{val(Xi) \mid i \in [1..n]\}$, not for $\Pi_{(D,m)}$. Independently of $m \in [1..n]$, we use the goto and the failure functions of this automaton, and adjust the output function appropriately for $\Pi_{(D,m)}$.

Compact representation of g-trie

For a compact representation of the g-trie, we can adopt the so-called path compaction technique like the suffix trees [17]. The compact g-trie for $D = \{Xi \rightarrow expri\}n=1$ is the path-compacted trie obtained from the g-trie for $\{val(Xi) \mid i \in [1..n]\}$ by removing the implicit states, where every edge e from q to r (let q and r be explicit states representing strings u and uv, respectively) is labeled by a, Xi _ such that a = v [1], Xi [1..|uv |] = uv and Xi stabs [1..|uv |]. The next lemma directly follows from Theorem4.

Theorem 9

There are at most 2n states in the compact g-trie for D of size n. displays the AC automaton and the compact g-trie for ΠD where D is identical to the example SLP An implicit state q*on edge e = (q, r) can be specified by an integer $h \geq 1$ such that q* represents the string Xi [1..|u| +h] and Xi stabs [1..|u| + h], where q represents string u and e is labeled by (a, Xi).

Theorem 10

An O(n)-space compact g-trie can be constructed in $O(n^3 \log n)$ time and $O(n^2)$ space so that for any state q and any character c , g (q, c) can be determined in O (log N) time.

Proof

We can compute in $O(n^3 \log n)$ time the sorted index σ of D and an array storing the longest common prefix length of val (Xσ (i)) and val(Xσ (i+1)) for all $i \in [1..n-1]$. Thus the compact g-trie can be constructed in $O(n^3 \log n)$ time. When q is an explicit state, we can find the edge e = (q, r) labeled by(c, Xi) for some variable Xi in O (log |Σ |) time, if such e exists, and we thus determine g (q, c) in O (log |Σ |) time, if such e exists, and we thus determine g (q, c) in O (log |Σ |) time.. When q is an implicit state on edge e specified by integer h, we can compute the (h + 1)th character in the string spelled out by e in O (log N) time by using the technique of Theorem8, and then compare it with c to determine g (q, c).

Thus, we can represent the goto function compactly. A naive implementation of the failure function, however, requires exponential space. In the following two subsections, we describe how to represent the failure and the output functions in polynomial space with respect to n. By combining those results, we will finally show our main theorem in Section Main result on compressed Aleshin type AC automata.

Compact representation of failure function

As stated in the previous subsection we can represent any implicit state of the compact g-trie as a pair of an edge e = (q, r) and an integer h. Here, we show another representation of states in the compact g-trie: A reference-pair of explicit/implicit state q is defined to be (Xi , h)such that q represents string Xi [1..h] and Xi stabs [1..h].

Theorem 11

A mutual conversion between the two state representations can be performed in O (logn) time using some data structure of size $O(n^2)$.

Proof

Let q be any state that represents string u . Suppose q is an explicit state. If q is terminating, let Xi be the variable corresponding to q, and otherwise, let Xi be the variable such that some out-going edge e from q is labeled by (a, Xi). Then, (Xi , |u |)gives a reference-pairs of q. Suppose q* is an implicit state on edge e = (q, r) specified by integer h, and e is labeled by (a, Xi). Then,(Xi , |u| +h) gives a reference pairs of q. Conversely, suppose we are given a reference-pair (Xi , h) of some state q* . Then, it is possible to determine in O (logn) time the explicit state q that is the nearest ancestor of q*, by using a simple binary search over the lengths of strings represented by the explicit states on the path from the initial state to the terminating state for Xi .

Let Prefix(D) denote the set of prefixes of val(Xi) for all variables Xi inD. For any variable $Xi \rightarrow Xl\,Xr \in D$, an f-interval of Xi is a maximal element in the set $\{[b..e] \mid 1 < b \leq |Xl| < e \leq |Xi|, Xi[b..e] \in Prefix(D)\}$ with respect to the set inclusion relation ⊆. The f-interval sequence of Xi , denoted F (Xi), is defined to be the sequence {[bk ..ek]}s =1 of all f-intervals of Xi arranged in the increasing order of b k . By definition $e_1, \ldots, e_s$ are also arranged in the increasing order of e_k The set of f-interval sequences represents the failure function f as follows:

Theorem 12

Let q be any state. Suppose q represents string Xi [1..h]. If h = 1, then f (q) is the initial state. Suppose $h \geq 2$. Choose Xi so that Xi stabs [1..h]. Let $\{[bk..ek]\}^{s}_{k=1}$ be the f-interval sequence of Xi , and let $k\_ \in [1..s]$ be the smallest integer such that $h \in [b\,k\_ ..ek\_]$. Then, the state f (s) represents the string Xi [bk_ ..h]. If no such k_ exist, then f (q) represents the string Xr [1..h − |Xl |]
where $Xi \rightarrow Xl\,Xr \in D$.

A naive way of encoding the f-interval sequence {[bk ..ek]}s =1 of a variable Xi is to have a linear-list of triples of bk , ek , X_j _ such that Xi [bk ..ek] = X_j [1..ek − bk + 1] and X j stabs [1..ek − bk + 1]. The list length s can, however, be exponential with respect to n.

Example 2

Consider the SLP D = $\{X_1 \to a\} \cup \{X_i \to X_{i-1} X_{i-1}\}_{n-3} \cup \{X_{n-2} \to b, X_{n-1} \to X_{n-2} X_{n-3}, X_n \to X_{n-1} X_{n-3}\}$. Then there are $2^{n-4} - 1$ f-intervals of Xn .

Fortunately, we can prove by making use of cyclic structures on f-intervals.

For any variable $Xi \to X_l Xr \in D$ and any f-interval [b..e] ∈ F (Xi), if there is a run [α ..β] with period p such that $\alpha \le b < e \le \beta$ and $e - b + 1 \ge 2p$, we say that the run [α..β] subsumes the f-interval [b..e]. Note that if such run exists, p is the smallest period of Xi [b..e] and the run is unique with respect to [b..e]. If a run [α ..β] subsumes two distinct f-intervals [b..e] and [b_ ..e_] such that Xi [b..e] = Xi [b_ ..e_] and $b < b\_ \le |Xl| - p$, [α ..β] is said to be f-rich.

Theorem 13

For any variable $Xi \to X_l X_r$ in D, there is at most one f-rich run.

Proof

The existence of an f-rich run [α ..β] with period p implies that u = Xi [|Xl | − p + 1..|Xl |] = Xi [|Xl | + 1..|Xl | + p]. Also, from the definition of f-rich run, there must exist

an f-interval [b..e] such that [b..e] ⊒ [|X l | − p + 1..|Xl | + p].

Assume on the contrary that there is another f-rich run [α'..β'] with period p' (w.l.o.g. assume p' < p). Since Xi [$|X_l|$ − p' + 1..$|X_l|$] = Xi [$|X_l|$ + 1..$|X_l|$ + p'], p – p' is a period of u . Since any interval contained in [b..e] cannot be an f-interval, at least one of $\alpha' < b \le |X_l| - p + 1$ or $|X_l| + p \le e < \beta'$ must hold. In either case, u has a period p' depicts the situation when $|X_l| + p \le e < \beta'$ is assumed). It follows from the periodicity lemma that gcd(p – p' , p') is a period of u , which means that p is not the smallest period of Xi [α ..β], a contradiction.

Lemma 14

Let $Xi \to X_l Xr$ be any variable in D. Let [b..e] and [b_ ..e_] be the first and the last f-intervals subsumed by a run [α ..β] with period p , respectively. For any d with $p \le d < d + p < b' - b$, [b + d..e''] ∈ F (Xi) ⟺ [b + d + p..e'' + p] ∈ F (Xi).

Lemma 15

Let $Xi \to X_l Xr$ be any variable in D. Let [b..e] and [b' ..e'] be the first and the last f-intervals subsumed by a run [α ..β] with period p, respectively. For any d with $0 \le d < d + p < b' - b$, |{[b'' ..e''] [b'' ..e''] ∈ F (Xi), b + d ≤ b'' <b +d + p}| ≤ n.

Proof

Assume on the contrary that |{[b'' ..e''] | [b'' ..e''] ∈ F (Xi), b + d ≤ b'' <b + d + p}| > n. By the pigeon hole principle, there exists X j such that Xi [b_1 ..e_1] and Xi [b_2 ..e_2] are prefixes of X_j , where [b_1 ..e_1], [b_2 ..e_2] ∈ F (Xi) with $b + d \le b_1 < b_2 < b + d + p$. Also, recall that from the definition of f-rich run, [|Xl | − p + 1..|Xl | + p] is covered by an f-interval, and the length of any f-interval subsumed by the f-rich run is longer than p . Consider u = Xi [b_1 ..$b_2 + p - 1$] and observe that p and b2 − b1 (< p) are both periods of u . Then, it follows from the periodicity lemma that gcd(b2 − b1 , p) is a period of u which contradicts that p is the smallest period of the f-rich run.

In light of Theorem 14 we consider storing cyclic f-intervals in a different way from the naive list of F (Xi). Since information of f-intervals for one period is enough to compute failure function for any state within the cyclic part, it can be stored in an O (n)-size list Lc (Xi) by Theorem 15. Let L(Xi) denote the list storing F (Xi) other than cyclic f-intervals. Note that L(Xi) includes O(n) f-intervals subsumed by the f-rich run but not in the cyclic part.

Theorem16

For any $Xi \to X_l X_r \in D$, the size of L(Xi) is bounded by O (n log N).

Proof

Let X_j be any variable and let c_0 , . . . , cs ($c_0 < \cdots < c_s$) be the positions of val(X_l) at which a suffix of val(X_l) overlaps with a prefix of val(X_j). We note that each ck is a candidate for the beginning position of an f-interval of Xi . It follows from Theorem2 that c_0 , . . . , cs can be partitioned into at most O (log |Xl |) disjoint segments such that each segment forms an arithmetic progression.

Let $0 \le k < k' \le s$ be integers such that C = c k , . . . , ck' is represented by one arithmetic progression. Let d be the step of C , i.e., c k' = $c_{k'-1}$ + d = · · · = ck + (k' − k)d. We show that if more than two elements of C are related to the beginning positions of f-intervals of Xi , the f-rich run subsumes all those f-intervals but the last one. Suppose that for some $k \le h_1 < h_2 < h_3 \le k'$, c_{h1} , c_{h2} , c_{h3} ∈ C are corresponding to f-intervals, namely, [c_{h1} ..e], [c_{h2} ..e'], [c_{h3} ..e''] ∈ F (Xi)

with e − c_{h1} + 1 = L_{CP}(Xi [c_{h1} ..|Xi |], X j),

e' -c_{h2} + 1 = L_{CP}(Xi [c_{h2} ..|Xi |], X j) and

e'' − c_{h3} + 1 = L_{CP}(Xi [c_{h3} ..|Xi |], X_j).

It is clear that d is the smallest period of Xi [c_{h1} ..|X_l |] and |X_l | − c_{h1} + 1 > 2d. Let β be the largest position of val(Xi) such that Xi [c_{h1} ..β] has period d, i.e., there is a run [α , β] with $\alpha \le c_{h1} < |X_l| < \beta$. Let β_ be the largest position of val(X j) such that X j [1..β_] has period d.

• If β < e'' . Note that this happens only when β − c_{h3} + 1 = β' . Consequently,

L_{CP}(Xi [c_{h1} ..|Xi |], X_j) = L_{CP}(Xi [ch2 ..|Xi |], X j) =β' .

• If $\beta \geq e''$. It is clear that $\beta' < e'' - c_{h2} + 1$, since otherwise $[c_{h3} ..e'']$ would be contained in $[c_{h2} ..e']$. Then, $L_{CP}(Xi [c_{h1} ..|Xi|], X_j) = L_{CP}(Xi [c_{h2} ..|Xi|], X_j) = \beta'$.

In either case $Xi [c_{h1} ..e] = Xi [c_{h2} ..e'] = X j [1..\beta']$ holds, which means that except for at most one f-interval [c..e] satisfying $\beta < e$ the others are all subsumed by the f-rich run $[\alpha..\beta]$.

Since in each segment there are at most two f-intervals which are not subsumed by the f-rich run, the number of such f-intervals can be bounded by O (log N). Considering every variable X j , we can bound the size of L(Xi) by O (n log N).

In light of Theorems 14 and 16 we get the next lemma.

Theorem 17

An O (n^2 log N)-size representation of the failure function f can be constructed in O (n^3 logn log N) time using O (n^2 log N) space so that given reference-pair of any state q, a reference-pair of the state f (q) can be computed in O (log n) time.

Proof

In [18] a compressed overlap table OV for an SLP of size n such that for any pair of variables X and Y , OV (X , Y) contains O (log N)-size representation of overlaps between suffixes of val(X) and prefixes of val(Y). They showed how to compute OV in O (n^3 logn log N) time. Actually, their algorithm can be extended to compute L(Xi) and Lc (Xi) for all variable $Xi \in D$ in O (n^3 logn log N) time From Lemma 14 and Lemma 16, the total size for L(Xi) and Lc (Xi) for all variable $Xi \in D$ is bounded by O (n^2 log N). Using L(Xi) and Lc (Xi), we can compute f (q) for any state q = Xi , h_ in O (logn) time. If q is not in cyclic part of f-intervals, we conduct binary search on L(Xi), otherwise on Lc (Xi) with proper offset. It takes O (log(n log N)) = O (logn) time.

Compact representation of output function

Theorem 18

An O (nm)-size representation of the output function λ can be computed in O (n^3 logn) time and O (n^2) space so that given any state q = (Xi , h)we can compute λ(q) in O (height(Xi) + m) time.

Proof

First we construct a tree with nodes $\Pi \cup \{\varepsilon\}$ such that for any $p \in \Pi_{(D,m)}$ the parent of p is the longest element of $\Pi(_{D,m}) \cup \{\varepsilon\}$ which is also a suffix of p . The tree can be constructed in O (n^3 logn) time in a similar way to the construction of the compact g-trie. Note that λ(q) can be computed by detecting the longest member p of $\Pi_{(D,m)}$ which is also a suffix of Xi [1..h], and outputting all patterns on the path from p to the root of the tree. In addition, we compute in O (n^3) time a table of size O (nm) such that for any pair of $p \in \Pi_{(D,m)}$ and variable X j the table has Occξ (p, X j) in a form of one arithmetic progression.

Now we show how to compute the longest member of $\Pi_{(D,m)}$ which is also a suffix of Xi [1..h]. We search for it in descending order of pattern length. We use three variables p' , i' and h' , which are initially set to the longest pattern in $\Pi_{(D,m)}$, i and h, respectively. We omit the case when $|p'| = 1$ or $|p'| > h$ since it is trivial. If the end position of Xi [1..h] is contained in X r (i') and $|p'| > h' - |X\mathit{l}(i')|$, using arithmetic progression of Occξ (p' , Xi'), we can check if p' is a suffix of Xi [1..h] or not in constant time by simple arithmetic operations. If the above condition does not hold, we traverse the derivation tree of X i' toward the end position of Xi [1..h] updating i' and h' properly until meeting the above situation, where h' is updated to be the length of the overlapped string between X i' and Xi [1..h].

It is not difficult to see that the total time is O (height(Xi) + m).

Main result on compressed Aleshin type AC automata

Lemma1

Given any DSLP D, m_ of size n that represents dictionary $\Pi_{(D,m)}$ of total length N , it is possible to build, in O (n^3 logn log N) time and O (n^2 log N) space, an O (n^2 log N)-size compressed automaton that recognizes all occurrences of patterns in $\Pi_{(D,m)}$ within an arbitrary string with O (height(D) + m) amortized running time per character.

Proof

By Theorem 10, an O (n)-size representation of the g-trie can be obtained in O (n^3 logn) time and O (n^2) space. By Theorem 17, an O (n^2 log N)-size representation of the failure function can be obtained in O (n^3 logn log N) time and O (n^2 log N) space. By Theorem 18, an O (nm)-size representation of the output function can be obtained in O (n^3 logn) time and O (n^2) space. We also build an O (n2)-size data structure to conduct the bidirectional conversion between a state on the g-trie and its reference-pair (Theorem 11). Thus, the space occupancy of our compressed automaton is O (n^2 log N) which is dominated by the representation of the failure function. While pattern matching, the computations on the compact g-trie, the failure function and the output function require O (log N), O (logn) and O (height(Xi) + m) amortized time per character, respectively. Therefore we get the statement.

We note that when m = 1, the output function of Theorem 18 is not needed since it is enough to report the occurrence of Xn when we reach there. Hence, the following Corollary holds.

IV. CONCLUSION

For an SLP D of size n representing string T of length N, it is possible to build, in O (n^3 logn log N) time and O (n^2 log N) space, an O (n^2 log N)-size compressed Aleshin type automaton that recognizes all occurrences of T within an arbitrary string with O (log N) amortized running time per character.

REFERENCES

[1] L. G. Sieniec, M. Karpinski, W. Plandowski, W. Rytter, “Efficient algorithms for Lempel-Ziv encoding,” in: Proc. 4th Scandinavian Workshop on Algorithm Theory, in: Lecture Notes in Comput. Sci., Springer, vol.1097, pp.392-403, 1996.

[2] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa, “Collage system: a unifying framework for compressed pattern matching,” Theoret. Comput. Sci. ,vol.298, Issue.1, pp.253-272, 2003.

[3] C.G. Nevill-Manning, I.H. Witten, D.L. Maulsby, “Compression by induction of hierarchical grammars,” in: Proc. DCC’94, pp. 244-253, 1994.

[4] N.J. Larsson, A. Moffat, “Offline dictionary-based compression,” in: Proc. DCC’99, IEEE Computer Society, pp. 296-305, 1999.

[5] J. Ziv, A. Lempel, “Compression of individual sequences via variable-length coding,” IEEE Trans. Inform. Theory, vol.24, Issue 5, pp. 530-536, 1978.

[6] T.A. Welch, “A technique for high performance data compression,” IEEE Comput., vol.17, pp. 8-19, 1984.

[7] J. Ziv, A. Lempel, “A universal algorithm for sequential data compression,” IEEE Trans. Inform. Theory IT, vol.23, Issue 3, pp.337-349, 1977.

[8] J. Storer, T. Szymanski, “Data compression via textual substitution,” J. ACM, vol.29, Issue 4, pp.928-951, 1982.

[9] W. Rytter, “Application of Lempel-Ziv factorization to the approximation of grammar-based compression,” Theoret. Comput. Sci., vol.302, Issue 1-3, pp.211-222, 2003.

[10] A. Aho, M. Corasick, “Efficient string matching: an aid to bibliographic search,” Commun. ACM, vol.18, Issue 6, pp.333-340, 1975.

[11] J.H. Morris, V.R. Pratt, A linear pattern-matching algorithm, Tech. rep. 40, University of California, Berkeley, 1970.

[12] D. Belazzougui, “Succinct dictionary matching with no slowdown,” in: Proc. CPM 2010, pp. 88-100, 2010.

[13] J. Arz, J. Fischer, “LZ-compressed string dictionaries,” CoRR arXiv:1305.0674.

[14] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, New York, 1994.

[15] M. Miyazaki, A. Shinohara, M. Takeda, “An improved pattern matching algorithm for strings in terms of straight-line programs, in: Proc. CPM1997,” in: Lecture Notes in Comput. Sci., Springer, vol. 1264, pp. 1-11, 1997.

[16] P. Bille, G.M. Landau, R. Raman, K. Sadakane, S.R. Satti, O. Weimann, “Random access to grammar-compressed strings,” in: Proc. SODA’11, pp. 373-389, 2011.

[17] P. Weiner, “Linear pattern-matching algorithms,” in: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, Institute of Electrical Electronics Engineers, New York, pp. 1-11, 1973.

[18] M. Karpinski, W. Rytter, A. Shinohara, “An efficient pattern-matching algorithm for strings with short descriptions,” Nordic J. Comput., vol.4, pp.172-186, 1997.