数理解析研究所講究録
第1744巻 2011年 123-128

2010年度冬のLAシンポジウム [22]

圧縮文字列上での q-gram 頻度の高速な計算方法

後藤啓介* 坂内英夫† 稲永俊介‡ 竹田正幸§

2011年2月2日

Abstract

We present a simple and efficient algorithm for calculating q-gram frequencies on strings represented in compressed form, namely, as a straight line program (SLP). Given an SLP of size n that represents string T, the algorithm computes the frequencies of all q-grams in T in $O(qn)$ time and space. In the extreme case, n can be as small as $O(\log |T|)$, and thus the algorithm is exponentially faster than any algorithm working on the uncompressed representation, when q is considered constant. Computational experiments show that our algorithm and a variation of it is practical for small q, and actually runs faster on various real-word string data, compared to algorithms that work on the uncompressed representation. We also discuss applications in data mining and classification of string data, for which our algorithm can be useful.

1 Introduction

Many forms of data such as texts, biological sequences (DNA/proteins), MIDI sequences, etc. can be represented as a sequence of characters, or *strings*, and developing methods for efficiently processing string data is one of the most important areas of research in computer science. A major problem is the sheer size of the data, and efficient algorithms for processing huge amounts of data are in high demand. To cope with this problem, algorithms that work directly on compressed representation of strings have gained attention especially for the string pattern matching problem [1], and there has been growing interest in what problems can be efficiently solved in this kind of setting [12].

In this paper, we attempt to explore a more advanced field of application in this setting: data mining and classification of string data given in compressed form. Methods for discovering useful patterns hidden in strings as well as methods for automatic and accurate classification of the data into various groups, is an important topic in the field of data mining and machine learning with many applications. As a first step toward compressed string mining, we consider q-grams on strings. A q-gram is simply a string of length q. q-grams are simple but important features of string data, and have been used widely in many fields such as natural language processing, and bioinformatics.

We consider text strings represented as *straight line programs* (SLPs) [6]. An SLP is a context free grammar in the Chomsky normal form that derives a single string. SLPs are a widely accepted abstract model of various text compression schemes, since texts compressed by any grammar-based compression algorithms (e.g. [15, 10]) can be represented as SLPs, and those compressed by the LZ-family (e.g. [16, 17]) can be quickly transformed to SLPs. Also recently, *self-indexes* based on SLPs have appeared [3], and SLP is a promising representation of a given string, not only for reducing the storage size of the data, but for conducting various operations on it.

In this paper, we give an algorithm that computes all q-gram frequencies of a given text represented as an SLP of size n, in $O(qn)$ time and space. This generalizes and greatly improves on the $O(|\Sigma|^2 n^2)$-time $O(n^2)$-space algorithms presented in [4], and later improved to $O(|\Sigma|^2 n \log n)$-time $O(n \log |T|)$-space in [3], for finding the most frequent substring of length 2 of a given compressed text. Applying the previous algorithms to q-grams respectively require $O(|\Sigma|^q qn^2)$ and $O(|\Sigma|^q qn \log n)$

*九州大学大学院システム情報科学府
†九州大学大学院システム情報科学研究院
‡第2著者に同じ
§第2著者に同じ

time, since they essentially enumerate and count the occurrences of all substrings of length q. Our algorithm has profound applications in the field of string mining and classification. For example, it leads to an $O(q(n_1 + n_2))$ time algorithm for computing the q-gram spectrum kernel [11] between SLP compressed texts of size n_1 and n_2. It also leads to an $O(qn)$ time algorithm for finding the optimal q-gram (or emerging q-gram) that discriminates between two sets of SLP compressed strings, when n is the total size of the SLPs.

1.1 Related Work

Several algorithms for finding characteristic sequences from compressed texts have been proposed, e.g., finding the longest common substring of two strings [14], finding all palindromes [14], finding most frequent substrings [4], and finding the longest repeating substring [4]. However, none of them have reported results of computational experiments. This implies that this paper is the first to show the practical usefulness of a compressed text mining algorithm.

2 Preliminaries

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. For any integer $q > 0$, an element of Σ^q is called an *q-gram*. The length of a string T is denoted by $|T|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $T = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. The i-th character of a string T is denoted by $T[i]$ for $1 \leq i \leq |T|$, and the substring of a string T that begins at position i and ends at position j is denoted by $T[i:j]$ for $1 \leq i \leq j \leq |T|$. For convenience, let $T[i:j] = \varepsilon$ if $j < i$.

For a string T and $q \geq 0$, let $pre(T, q)$ and $suf(T, q)$ represent respectively, the length-q prefix and suffix of T. That is, $pre(T, q) = T[1 : \min(q, |T|)]$ and $suf(T, q) = T[\max(1, |T| - q + 1) : |T|]$.

For any strings T and P, let $Occ(T, P)$ be the set of occurrences of P in T, i.e., $Occ(T, P) = \{k > 0 \mid T[k : k + |P| - 1] = P\}$. The number of elements $|Occ(T, P)|$ is called the *occurrence frequency* of P in T.

2.1 Straight Line Programs

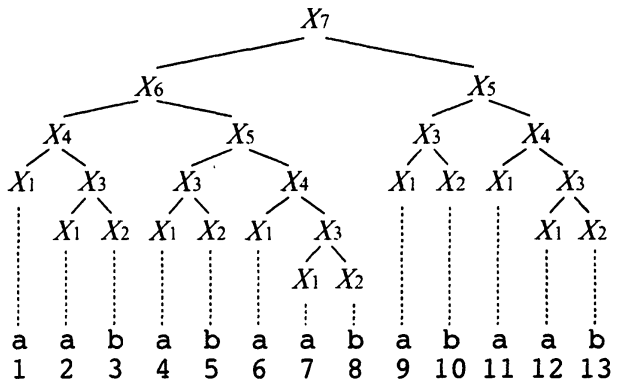

Figure 1: The derivation tree of SLP $\mathcal{T} = \{X_i\}_{i=1}^{7}$ with $X_1 = \mathtt{a}$, $X_2 = \mathtt{b}$, $X_3 = X_1X_2$, $X_4 = X_1X_3$, $X_5 = X_3X_4$, $X_6 = X_4X_5$, and $X_7 = X_6X_5$, representing string $T = val(X_7) = \mathtt{aababaababaab}$.

A *straight line program* (*SLP*) $\mathcal{T}$ is a sequence of assignments $X_1 = expr_1, X_2 = expr_2, \ldots, X_n = expr_n$, where each X_i is a variable and each $expr_i$ is an expression, where $expr_i = a$ ($a \in \Sigma$), or $expr_i = X_\ell X_r$ ($\ell, r < i$). Let $val(X_i)$ represent the string derived from X_i. When it is not confusing, we identify a variable X_i with $val(X_i)$. Then, $|X_i|$ denotes the length of the string X_i derives. An SLP $\mathcal{T}$ *represents* the string $T = val(X_n)$. The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$. (See Fig. 1)

The substring intervals of T that each variable derives can be defined recursively as follows: $itv(X_n) = \{[1 : |T|]\}$, and $itv(X_i) = \{[u + |X_\ell| : v] \mid X_k = X_\ell X_i, [u : v] \in itv(X_k)\} \cup \{[u : u + |X_i| - 1] \mid X_k = X_i X_r, [u : v] \in itv(X_k)\}$ for $i < n$. For example, $itv(X_5) = \{[4 : 8], [9 : 13]\}$ in Fig. 1. Considering the transitive reduction of set inclusion, the intervals $\cup_{i=1}^{n} itv(X_i)$ naturally form a binary tree (the derivation tree). Let $vOcc(X_i) = |itv(X_i)|$ denote the number of times a variable X_i occurs in the derivation of T. $vOcc(X_i)$ for all $1 \leq i \leq n$ can be computed in $O(n)$ time by a simple iteration on the variables, since $vOcc(X_n) = 1$ and for $i < n$, $vOcc(X_i) = \sum\{vOcc(X_k) \mid X_k = X_\ell X_i\} + \sum\{vOcc(X_k) \mid X_k = X_i X_r\}$. (See Algorithm 1)

Algorithm 1: Calculating $vOcc(X_i)$ for all $1 \leq i \leq n$.

Input: SLP $\mathcal{T} = \{X_i\}_{i=1}^{n}$ representing string T.
Output: $vOcc(X_i)$ for all $1 \leq i \leq n$

```
1 vOcc[X_n] ← 1;
2 for i ← 1 to n − 1 do vOcc[X_i] ← 0;
3 for i ← n to 2 do
4     if X_i = X_ℓ X_r then
5         vOcc[X_ℓ] ← vOcc[X_ℓ] + vOcc[X_i];
6         vOcc[X_r] ← vOcc[X_r] + vOcc[X_i];
```

Algorithm 2: A naïve algorithm for computing q-gram frequencies.

Input: string T, integer $q \geq 1$
Output: $(P, |Occ(T, P)|)$ for all $P \in \Sigma^q$ where $Occ(T, P) \neq \emptyset$.

```
1 S ← ∅; // associative array
2 for i ← 1 to |T| − q + 1 do
3     qgram ← T[i : i + q − 1];
4     if qgram ∈ keys(S) then
          S[qgram] ← S[qgram] + 1;
5     else S[qgram] ← 1;
6 return S
```

2.2 Suffix Arrays and LCP Arrays

We will make use of the *suffix array* and *lcp array*. It is well known that the suffix array for any string of length $|T|$ can be constructed in $O(|T|)$ time [5, 8, 9] assuming an integer alphabet. Given the text and suffix array, the lcp array can also be calculated in $O(|T|)$ time [7].

Definition 1 (Suffix Arrays) *The* suffix array *[13] SA of any string T is an array of length $|T|$ such that $SA[i] = j$, where $T[j : |T|]$ is the i-th lexicographically smallest suffix of T.*

Definition 2 (LCP Arrays) *The* lcp *array of any string T is an array of length $|T|$ such that $LCP[i]$ is the length of the longest common prefix of $T[SA[i-1] : |T|]$ and $T[SA[i] : |T|]$ for $2 \leq i \leq |T|$, and $LCP[1] = 0$.*

3 Algorithm

3.1 Computing q-gram Frequencies on Uncompressed Strings

We first describe two very simple algorithms for computing the q-gram frequencies of a given uncompressed string T.

3.1.1 A Naïve Algorithm

A very simple and naïve algorithm for computing the q-gram frequencies is given in Algorithm 2. The algorithm constructs an associative array, where keys consist of q-grams, and the values correspond to the occurrence frequencies of the q-grams. The time complexity depends on the implementation of the associative array, but requires at least $O(q|T|)$ time since each q-gram is considered explicitly, and the associative array is accessed $O(|T|)$ times.

Algorithm 3: A linear time algorithm for computing q-gram frequencies.

Input: string T, integer $q \geq 1$
Output: $(i, |Occ(T, P)|)$ for all $P \in \Sigma^q$ and some position $i \in Occ(T, P)$.

```
1 SA ← SUFFIXARRAY(T);
2 LCP ← LCPARRAY(T, SA);
3 count ← 1;
4 for i ← 2 to |T| + 1 do
5     if i = |T| + 1 or LCP[i] < q then
6         if count > 0 then Report
          (SA[i − 1], count);
7         count ← 0;
8     if i ≤ |T| and SA[i] ≤ |T| − q + 1 then
9         count ← count + 1;
```

3.1.2 $O(|T|)$ Time Algorithm

It is straightforward to compute the q-gram frequencies of string T using suffix array SA and lcp array LCP. For each $1 \leq i \leq |T|$, the suffix $SA[i]$ represents an occurrence of q-gram $T[SA[i] : SA[i] + q - 1]$, if the suffix is long enough, i.e. $SA[i] \leq |T| - q + 1$. The key is that since the suffixes are lexicographically sorted, intervals on the suffix array where the values in the lcp array are at least q represent occurrences of the same q-gram. The procedure is shown in Algorithm 3. The algorithm

runs in $O(|T|)$ time, since the construction of SA and LCP can be done in $O(|T|)$. The rest is a simple $O(|T|)$ loop. A technicality is that we encode the output for a q-gram as one of the positions in the text where the q-gram occurs, rather than the q-gram itself. This is because there are a total of $O(|T|)$ q-grams, and outputting them as length-q strings would require at least $O(q|T|)$ time.

Lemma 1 *Given a string T, the q-gram frequencies of T can be computed in $O(|T|)$ time for any $q > 0$, assuming an integer alphabet $\Sigma \subseteq \{1, \ldots, |T|\}$.*

3.2 Computing q-gram Frequencies on SLP

We now describe the core idea of our algorithms, and then go on to explain two variations which utilize variants of the two algorithms presented in the previous subsection. For $q = 1$, the 1-gram frequencies are simply the frequencies of the alphabet and the output is $(a, \sum\{vOcc(X_i) \mid X_i = a\})$ for each $a \in \Sigma$, which takes only $O(n)$ time. For $q \geq 2$, we make use of Lemma 2 below. The idea is similar to the *mk Lemma* [2] but more specifically tailored for our needs.

Lemma 2 *Let $\mathcal{T} = \{X_i\}_{i=1}^n$ be an SLP that represents string T. For an interval $[u : v]$ $(1 \leq u < v \leq |T|)$, there exists exactly one variable $X_i = X_\ell X_r$ such that for some $[u' : v'] \in itv(X_i)$, the following holds: $[u : v] \subseteq [u' : v']$, $u \in [u' : u' + |X_\ell| - 1] \in itv(X_\ell)$ and $v \in [u' + |X_\ell| : v'] \in itv(X_r)$.*

From Lemma 2, we have that each occurrence of a q-gram ($q \geq 2$) represented by some length-q interval of T, corresponds to a single variable $X_i = X_\ell X_r$, and is split in two by intervals corresponding to X_ℓ and X_r. On the other hand, consider all length q intervals that correspond to a given variable. Then, counting the frequencies of the q-grams they represent, and summing them up for all variables would give the frequencies of all q-grams of T.

For variable $X_i = X_\ell X_r$, let $t_i = suf(X_\ell, q-1)pre(X_r, q-1)$. Then, all q-grams represented by length q intervals that correspond to X_i are those in t_i. (Fig. 2). If we obtain q-gram frequencies of t_i, and then multiply the frequencies of each q-gram by $vOcc(X_i)$, we obtain the frequencies of the

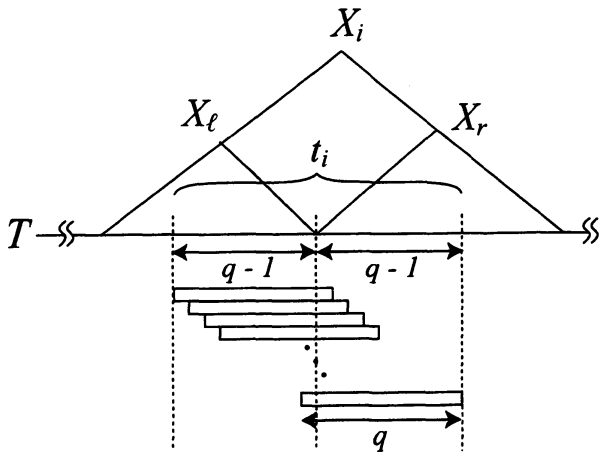

Figure 2: Length-q intervals corresponding to variable $X_i = X_\ell X_r$.

q-grams that occur in all intervals derived by X_i. It remains to sum up the q-gram frequencies of t_i for all $1 \leq i \leq n$. Thus, we can regard the problem as obtaining the *weighted* q-gram frequencies in the set of strings $\{t_1, \ldots, t_n\}$, where each q-gram in t_i is weighted by $vOcc(X_i)$.

The procedure is shown in Algorithm 4. A single string z is constructed by concatenating t_i such that $q \leq |t_i| \leq 2(q-1)$, and the weights of q-grams starting at each position in z is held in array w. On line 9, 0's instead of $vOcc(X_i)$ are appended to w for the last $q-1$ values corresponding to t_i. This is to avoid counting unwanted q-grams that are generated by the concatenation of t_i to z on line 7, which are not substrings of each t_i. Line 10 can be calculated by a slight modification of Algorithm 2 or 3. For Algorithm 2, line 4 should be modified to increment $\mathbf{S}[qgram]$ by $w[i]$ rather than 1, and line 5 should read: **else if** $w[i] > 0$ **then** $\mathbf{S}[qgram] \leftarrow w[i]$;. For Algorithm 3, the increment on line 9 should simply use $w[SA[i]]$.

Theorem 1 *Given an SLP $\mathcal{T} = \{X_i\}_{i=1}^n$ of size n representing a string T, the q-gram frequencies of T can be computed in $O(qn)$ time for any $q > 0$, assuming an integer alphabet $\Sigma \subseteq \{1, \ldots, n\}$.*

Note that the time complexity for using the weighted version of Algorithm 2 for line 10 of Algorithm 4 would be at least $O(q^2 n)$.

Algorithm 4: Calculating q-gram frequencies of an SLP for $q \geq 2$

Input: SLP $\mathcal{T} = \{X_i\}_{i=1}^{n}$ representing string T, integer $q \geq 2$.

```
1  Calculate vOcc(X_i) for all 1 ≤ i ≤ n;
2  Calculate pre(X_i, q − 1) and suf(X_i, q − 1) for
   all 1 ≤ i ≤ n − 1 ;
3  z ← ε; w ← [];
4  for i ← 1 to n do
5      if X_i = X_ℓ X_r and |X_i| ≥ q then
6          t_i = suf(X_ℓ, q − 1)pre(X_r, q − 1);
7          z.append(t_i);
8          for j ← 1 to |t_i| − q + 1 do
           w.append(vOcc(X_i));
9          for j ← 1 to q − 1 do w.append(0);
10 Calculate q-gram frequencies in z, where each
   q-gram starting at position i is weighted by
   w[i].
```

4 Experiments

To evaluate the effectiveness of our algorithms, we implemented 4 algorithms (NMP, NSA, SMP, SSA) to solve the simple problem of calculating the most frequent q-gram. NMP (Algorithm 2) and NSA (Algorithm 3) work on the uncompressed text. SMP (Algorithm 4 + weighted version of Algorithm 2) and SSA (Algorithm 4 + weighted version of Algorithm 3) work on SLPs. The algorithms were implemented using the C++ language. We used `std::map` from the Standard Template Library (STL) for the associative array implementation. [1]

The running time is measured in seconds, starting from after reading the uncompressed text into memory for NMP and NSA, and after reading the text represented as an SLP into memory for SMP and SSA. Each computation is repeated at least 3 times, and the average is taken.

We applied the algorithms on texts XML, DNA, ENGLISH, and PROTEINS, with sizes 50MB, 100MB, and 200MB, obtained from the Pizza & Chili Corpus[2]. To obtain SLPs for this data, we used a linear time greedy algorithm based on RE-PAIR [10] which recursively replaces the most frequent 2-grams occurring in the string, until the string is converted to a single variable.

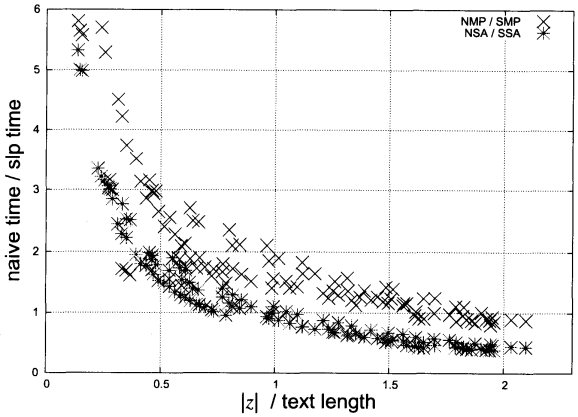

Figure 3: Time ratios: NMP/SMP and NSA/SSA plotted against ratio: (length of z in Algorithm 4)/(length of uncompressed text).

Fig. 3 shows the time ratio, where q is varied from 2 to 10: NMP/SMP and NSA/SSA, plotted against ratio: (length of z in Algorithm 4)/(length of uncompressed text). As expected, the SLP versions are basically faster than their naïve counterparts, when $|z|$/(text length) is less than 1, since the SLP versions run the weighted versions of the naïve algorithms on a text of length $|z|$.

5 Conclusion

We showed that for an SLP $\mathcal{T}$ of size n representing string T, q-gram frequency problems on T can be reduced to *weighted* q-gram frequency problems on a string of length $O(qn)$, which can be much shorter than T. This idea can further be applied to obtain faster algorithms for many interesting problems.

This paper is a new addition in the line of work aimed for efficient string processing on compressed texts. The main conceptual contribution of the paper in this sense is that it takes the first steps in showing the potential of these approaches for developing efficient and *practical* algorithms for very large scale data, to problems in the field of string mining and classification.

[1] We also used `std::hash_map` but omit the results due to lack of space. Choosing the hashing function to use is difficult, and we note that its performance was unstable and sometimes very bad when varying q.

[2] `http://pizzachili.dcc.uchile.cl/texts.html`

Acknowledgments

This work was supported by KAKENHI 22680014.

References

[1] Amihood Amir and Gary Benson. Efficient two-dimensional compressed matching. In *Data Compression Conference*, pages 279–288, 1992.

[2] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[3] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, To appear.

[4] Shunsuke Inenaga and Hideo Bannai. Finding characteristic substring from compressed texts. *International Journal of Foundations of Computer Science*, accepted for publication.

[5] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. ICALP'03*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer-Verlag, 2003.

[6] M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing*, 4:172–186, 1997.

[7] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. of CPM'01*, volume 2089 of *LNCS*, pages 181–192. Springer-Verlag, 2001.

[8] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199, 2003.

[9] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210, 2003.

[10] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference 1999*, pages 296–305. IEEE Computer Society, 1999.

[11] Christina Leslie, Eleazar Eskin, and William Stafford Noble. The spectrum kernel: A string kernel for SVM protein classification. In *Pacific Symposium on Biocomputing*, volume 7, pages 566–575, 2002.

[12] Yury Lifshits. Processing compressed texts: A tractability border. In *Proc. Combinatorial Pattern Matching*, pages 228–240, 2007.

[13] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.

[14] Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoretical Computer Science*, 410(8–10):900–913, 2009.

[15] C. G. Nevill-Manning, I. H. Witten, and D. L. Maulsby. Compression by induction of hierarchical grammars. In *Data Compression Conference 1994*, pages 244–253. IEEE Computer Society, 1994.

[16] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–349, 1977.

[17] J. Ziv and A. Lempel. Compression of individual sequences via variable-length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.