Available online at www.sciencedirect.com
ScienceDirect
Progress in Natural Science 18 (2008) 325–330
Progress in Natural Science

Short communication

Canonical Huffman code based full-text index

Yi Zhang [a,b], Zhili Pei [a], Jinhui Yang [a], Yanchun Liang [a,*]

[a] College of Computer Science and Technology, Jilin University, Key Laboratory of Symbol Computation and Knowledge Engineering of the Ministry of Education, Changchun 130012, China
[b] Department of Computer Science, Jilin Technology and Business College, Changchun 130062, China

Received 18 June 2007; received in revised form 16 September 2007; accepted 12 November 2007

Abstract

Full-text indices are data structures that can be used to find any substring of a given string. Many full-text indices require space larger than the original string. In this paper, we introduce the canonical Huffman code to the wavelet tree of a string $\boldsymbol{T}[1..n]$. Compared with Huffman code based wavelet tree, the memory space used to represent the shape of wavelet tree is not needed. In case of large alphabet, this part of memory is not negligible. The operations of wavelet tree are also simpler and more efficient due to the canonical Huffman code. Based on the resulting structure, the multi-key *rank* and *select* functions can be performed using at most $nH_0 + |\Sigma|(\lg\lg n + \lg n - \lg|\Sigma|) + O(nH_0)$ bits and in $O(H_0)$ time for average cases, where H_0 is the zeroth order empirical entropy of $\boldsymbol{T}$. In the end, we present an efficient construction algorithm for this index, which is on-line and linear.

Keywords: Full-text index; Suffix automaton; Canonical Huffman code

1. Introduction

Full-text indices are data structures that can efficiently find any substring of a given string. The inverted index commonly used in text retrieval is space economical and fast. However, it is a kind of word index that is not suitable for sequences, such as Chinese texts or biological sequences. In such cases, three classic full-text indices suffix trees [1], suffix automata [2] and suffix arrays [3,4] are used. The major drawback that limits the applicability of these full-text indices is that their space required is quite larger than the original text.

Recent researches are focused on reducing the sizes of full-text indices [5–11]. The compressed suffix array structure [5] proposed by Grossi and Vitter is the first method for reducing the size of the suffix array from $O(n\log n)$ bits to $O(n)$ bits and supporting access to any entry of the original suffix array in $O(\lg^{\varepsilon}_{|\Sigma|} n)$ time, for any fixed constant $0 < \varepsilon < 1$ (without computing the entire original suffix array).

FM-index [7] proposed by Ferragina and Manzini is a self-index data structure with good compression ratio and fast decompressing speed. The FM-index occupies at most $5nH_k + O(n)$ bits of storage and allows the search for the *occ* occurrences of a pattern $\boldsymbol{P}$ $[1...p]$ within $\boldsymbol{T}$ in $O(p + occ\lg^{1+\varepsilon} n)$ time, where $\varepsilon > 0$ is an arbitrary constant fixed in advance. In Ref. [12], the authors design a variant of the FM-index. The size of the new index is bounded by $nH_k + O((n\lg\lg n)/\log_{|\Sigma|} n)$ bits, where H_k is the kth order empirical entropy of $\boldsymbol{T}$. Using this index, counting of the occurrences of an arbitrary pattern $\boldsymbol{P}[1..p]$ as a substring of $\boldsymbol{T}$ takes $O(p\lg|\Sigma|)$ time.

In Ref. [8], He et al. present a succinct representation of suffix arrays of binary strings that uses $n + o(n)$ bits. For the case of large alphabet, they suggested an approach which conceptually sets a bit vector for each alphabet

* Corresponding author. Tel.: +86 431 85153829; fax: +86 431 85168752.
E-mail address: ycliang@jlu.edu.cn (Y. Liang).

1002-0071/$ - see front matter © 2007 National Natural Science Foundation of China and Chinese Academy of Sciences. Published by Elsevier Limited and Science in China Press. All rights reserved.
doi:10.1016/j.pnsc.2007.11.001

character to support multi-key *rank* and *select* functions, and uses a wavelet tree in the actual implementation to reduce the space cost.

In this paper, we introduce a canonical Huffman code based wavelet tree [5]. The wavelet tree is a data structure that efficiently supports basic operations for space-economical full-text indices. Huffman code based wavelet tree can be found in Ref. [5], which greatly reduces the memory consumption of wavelet tree. In our approach, the memory space used to represent the shape of wavelet tree is not needed. In case of large alphabet, this part of memory is not negligible. The data structure can be stored in continuous memory. This makes the operations of wavelet tree simpler and more efficient. It supports multi-key *rank* and *select* functions using at most $nH_0 + |\Sigma|(\lg\lg n + \lg n - \lg|\Sigma|) + O(nH_0)$ bits and taking $O(H_0)$ time on average, in $O(|\Sigma|)$ in the worst case, where H_0 is the 0th order empirical entropy of $\boldsymbol{T}$. The number of characters which are not greater than a character in a string can be computed simultaneously with multi-key rank function without using any additional space. The same function in FM-index [7] and Ref. [8] is implemented via a table of $|\Sigma|\lg n$ bits. Based on this data structure, we implement the suffix automaton [2] in a space economical way. In the end, we present an efficient on-line construction algorithm for this structure. It runs in linear time using very small auxiliary memory space.

2. Preliminaries

Let Σ be a nonempty alphabet and $|\Sigma|$ the number of characters in Σ. Let $\boldsymbol{T} = [t_1, t_2, \ldots, t_n]$ be a word over Σ, $|\boldsymbol{T}|$ denoting its length, $T[i]$ or t_i its ith character, and $T[i\ldots]$ or T_i its suffix that begins at position i, $1 \leqslant i \leqslant |\boldsymbol{T}|$. Denote $\boldsymbol{T^R}$ the reverse string of $\boldsymbol{T}$. *Suff*($\boldsymbol{T}$) denotes the set of all suffixes of $\boldsymbol{T}$ and *Fact*($\boldsymbol{T}$) the set of its factors.

2.1. Suffix automaton

The suffix automaton [2] (also called DAWG) of a string $\boldsymbol{T}$ is the minimal DFA that accepts all the suffixes of $\boldsymbol{T}$. For any string $u \in \Sigma^*$, let $u^{-1}S = \{x | ux \in S\}$. The syntactic congruence associated with *Suff*(w) is denoted by $\equiv_{Suff(w)}$ [2] and is defined, for x, y, $w \in \Sigma^*$, by

$$x \equiv_{Suff(w)} y \iff x^{-1}Suff(w) = y^{-1}Suff(w)$$

We call classes of factors the congruence classes of the relation $\equiv_{Suff(w)}$. Let $[u]_w$ denote the congruence class of $u \in \Sigma^*$ under $\equiv_{Suff(w)}$. The longest element in $[u]_w$ is called its *representative*, denoted by $rp([u]_w)$.

Definition 1. The DAWG of w is a directed acyclic graph with set of states $\{[u]_w | u \in Fact(w)\}$ and set of edges $\{([u]_w, a, [ua]_w) | u, ua \in Fact(w), \quad a \in \Sigma\}$, denoted by $DAWG(w)$. The state $[\varepsilon]_w$ is called the *root* of $DAWG(w)$.

The suffix link of a state p is the state whose representative v is the longest suffix of u such that v not $\equiv_{Suff(w)} u$.

2.2. The rank and select functions on bit vectors

The *rank* and *select* functions on bit vectors are extensively used in succinct index data structures. Function $rank_1(\boldsymbol{B},i)$ and function $rank_0(\boldsymbol{B},i)$ return the number of 1s and 0s in the bit vector $\boldsymbol{B}[1\ldots n]$ up to position i, respectively. The *rank* function can be computed in constant time by using a data structure of size $n + o(n)$ bits [13]. Function $select_1(\boldsymbol{B}, i)$ and function $select_0(\boldsymbol{B}, i)$ return the positions of ith 1 and 0, respectively. The select function can be computed in constant time by using a data structure of size $n + o(n)$ bits [14].

For convenience, we use function $rank_1(\boldsymbol{B})$ and function $rank_0(\boldsymbol{B})$ to denote function $rank_1(\boldsymbol{B},n)$ and function $rank_0(\boldsymbol{B},n)$, respectively. We will also use function $rank_1(\boldsymbol{B}[s\ldots i])$ and $rank_0(\boldsymbol{B}[s\ldots i])$, $1 \leqslant s \leqslant i \leqslant n$, to denote functions $rank_1(\boldsymbol{B},i) - rank_1(\boldsymbol{B},s-1)$ and $rank_0(\boldsymbol{B},i) - rank_0(\boldsymbol{B},s-1)$. Both functions $rank_0(\boldsymbol{B}[s\ldots i])$ and $rank_1(\boldsymbol{B}[s\ldots i])$ run in constant time, just as function *rank* does.

3. Compact implementation of suffix automata

Let u be a substring of $\boldsymbol{T}$, $\boldsymbol{SA}$ the suffix array of $\boldsymbol{T}$. All the occurrences of u in $\boldsymbol{T}$ are grouped consecutively in $\boldsymbol{SA}$. Therefore, an internal node $\bar{u}$ of the suffix tree, where u is the longest string in the node, $\bar{u}$ can be represented by an interval $[s, e]$ over $\boldsymbol{SA}$ where all suffixes with prefix u are included [8]. $SA[s]$ and $SA[e]$ are the lexically smallest and greatest suffixes in this interval. This approach leads to a space economical implementation of the suffix automaton. The nodes and suffix links of $DAWG(\boldsymbol{T})$ form the suffix tree of $\boldsymbol{T^R}$ [2]. A state of $DAWG(\boldsymbol{T})$ can therefore be represented by an interval of suffix array of $\boldsymbol{T^R}$ [11]. Denote by $\boldsymbol{SA'}$ the suffix array of $\boldsymbol{T^R}$ for a state $r = [s, e]$, for any suffix of $\boldsymbol{T^R}$, say T_i^R, if $rp(r)^R$ is a prefix of T_i^R, $T^R_{SA'[s]} \leqslant T_i^R \leqslant T^R_{SA'[e]}$.

Through this representation, an edge (p, a, q) of DAWG, say $q = goto(p, a)$, need not be stored explicitly and the *goto* function of DWAG can be computed efficiently [11]. This representation includes an array $\boldsymbol{E}$ of size $n + 1$ defined as follows:

$$E[i] = \begin{cases} T[1], & \text{if } i = 0 \\ T^R[SA'[i] - 1] = T[n + 2 - SA'[i]], & \text{if } 0 < i < n + 1 \end{cases}$$

Each entry of this array stores the character after each prefix in $\boldsymbol{SA'}$. It is similar to the reverse version of FM-index [7]. Another part of the representation is the data structure to support the multi-key *rank* function on $\boldsymbol{E}$ used to implement the *goto* function of DAWG. It extends *rank* function operation from bit vectors to strings. Let function $rank_\alpha(\boldsymbol{E}, i)$ return the number of as in $\boldsymbol{E}$ up to position i, function $rank_a^{\leqslant}(\boldsymbol{E}, i)$ return the number of characters which are not greater than a in string $\boldsymbol{T}$, plus function $rank_\alpha(\boldsymbol{E}, i)$. We have $goto([s, e], a) = [rank_\alpha^{\leqslant}(\boldsymbol{E}, s), rank_\alpha^{\leqslant}(\boldsymbol{E}, e)]$. The array $\boldsymbol{E}$ along with the data structure to support *rank* operation produces an implementation of suffix automaton which is space economical and not slowed down.

4. Multi-key *rank* and *select* functions canonical Huffman code based wavelet tree

We use the canonical Huffman code to encode the array $\boldsymbol{E}$ (defined in Section 3). Based on the encoded array, we introduce an efficient wavelet tree [5] to support the multi-key *rank* and *select* functions on $\boldsymbol{E}$. Its space occupation is smaller than wavelet tree and Huffman code based wavelet tree. The speed of the new structure is not slowed down. By canonical Huffman code, the wavelet tree can be stored in continuous memory. Compared with Huffman code based wavelet tree, the memory space used to represent the shape of wavelet tree is not needed. In case of large alphabet, this part of memory is not negligible. The operation of the wavelet tree is also simpler and more efficient due to the continuous storing of the data structure.

The wavelet tree is a binary tree of height $\lceil \log|\Sigma| \rceil$ built on the alphabet characters, such that each leaf represents a distinct alphabet character and each inner node represents a distinct binary prefix of alphabet characters. The root is in layer 0. The bit vector of a node of layer i is the $(i+1)$th bits of all characters whose first i bit is the binary prefix of the node, the order of bits agrees with that of characters in string. For the root, its bit vector is the first bit of all characters of string, the order of these bits agrees with the order of characters in string.

For a character α in a string $\boldsymbol{T}$, let *Huff*(α) be a Huffman code of α, $LH(\alpha)$ be the length of *Huff*(α), and $f_T(\alpha)$ be the frequency of α in $\boldsymbol{T}$. By ordering the characters decreased by the length of their Huffman codes, an order of characters is available. We call this order the "decreasing Huffman order" of Σ according to $\boldsymbol{T}$. To make the decreasing Huffman order consistent with the lexical order, we use a special optimal prefix code, namely "Canonical Huffman code". That is, if $LH(\alpha) < LH(\beta)$, the codeword of α is lexically less than the codeword of β. Denote the canonical Huffman encoding of α by $CH(\alpha)$. Here, we use CH code to refer to the canonical Huffman code and CH tree the Huffman tree corresponding to canonical Huffman code. Because the length of CH codeword of each character is equal to that of Huffman codeword, the length of CH encoded texts is equal to that of Huffman encoded texts. An example of CH tree is shown in Fig. 1.

In this section the array $\boldsymbol{E}$ is obtained from $\boldsymbol{T}$ according to the decreasing Huffman order other than the lexical order as in Section 4. We use $\widehat{E}$ to denote the canonical Huffman prefix encoding of $\boldsymbol{E}$, that is, $\widehat{E}[i] = CH(E[i])$, for $1 \leqslant i \leqslant n$.

Define a series of bit vectors of variable length: $\boldsymbol{F}^1, \ldots, \boldsymbol{F}^L$, where L is the length of the longest canonical Huffman codeword in $\widehat{E}$. First, $\boldsymbol{F}^1$ is defined as follows:

$$\boldsymbol{F}^1_i = \widehat{E}[i]_1, \quad 1 \leqslant i \leqslant n$$

The length of bit vector $\boldsymbol{F}^2$ is equal to the number of characters in $\boldsymbol{E}$ whose CH code's length is greater than 1. If the length of CH code of any character is greater than 1, $\boldsymbol{F}^2$ is defined by

$$F^2_i = \begin{cases} \widehat{E}[\text{select}_0(F^1, i)]_2 & \text{for } 0 < i < rank_0(F^1) \\ & \text{and } rank_0(F^1) \neq 0 \\ \widehat{E}[\text{select}_0(F^1, i)]_2 & \text{for } i = rank_0(F^1) \\ & \text{and } rank_0(F^1) \neq 0 \\ \widehat{E}[\text{select}_1(F^1, i)]_2 & \text{for } rank_0(F^1) < i < n+1 \\ & \text{and } rank_1(F^1) \neq 0 \end{cases}$$

If there exists a character which is encoded as 1, $\boldsymbol{F}^2$ is defined by

$$\boldsymbol{F}^2_i = \widehat{E}[select_0(\boldsymbol{F}^1, i)]_2, \quad 1 \leqslant i \leqslant rank_0(\boldsymbol{F}^1)$$

The bit vector $\boldsymbol{F}^k (1 < k \leqslant L)$ can be generated by the following procedure: First, order the positions in $\widehat{E}$, on which the CH code's length is not less than k, by the first $k-1$ bits of each CH code on these positions. For positions on which the CH codes have the same first $k-1$ bits, keep their order in $\widehat{E}$. This step generates a series of positions:

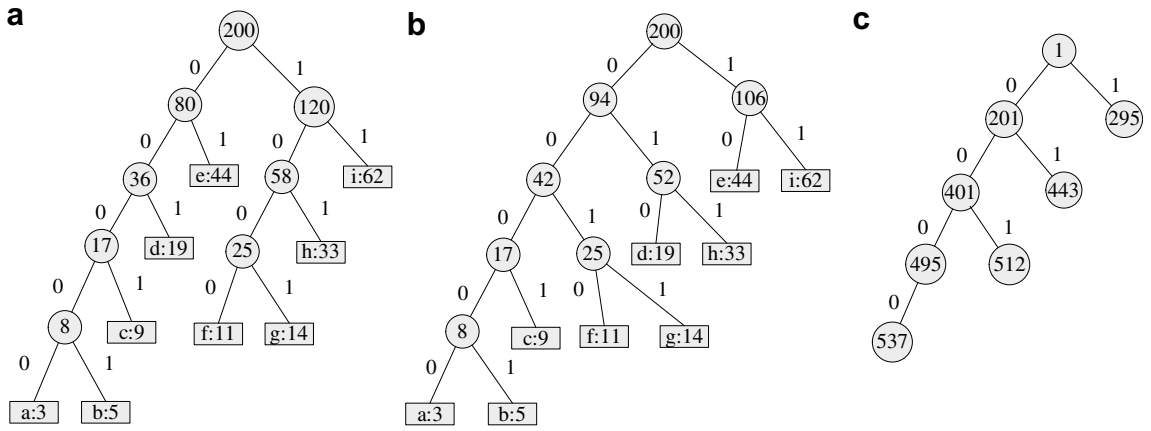

Fig. 1. Binary trees of Huffman code and canonical Huffman code (each leaf in tree (a) and (b) is labeled with a character and its frequency of occurrence). (a) The binary tree corresponding to a Huffman code; (b) the binary tree corresponding to a canonical Huffman code; (c) segment tree of (b) used in construction.

$pos_1, pos_2, \ldots, pos_{c_k}$, where c_k is the number of characters in $\boldsymbol{E}$ whose CH code's length is not less than k. Second, set $\boldsymbol{F}_i^k$, $1 \leqslant i \leqslant c_k$, to the kth bit of the CH code on position pos_i. The length of $\boldsymbol{F}^k$ is equal to c_k.

Precisely, $pos_1, pos_2, \ldots, pos_{c_k}$ are divided into l_k groups denoted by $G_0^k, \ldots, G_{l_k-1}^k$; CH codes on positions in G_i^k have the same first $k-1$ bits, which are equal to the binary representation of integer i. The vector $\boldsymbol{F}^k$ is composed of l_k non-overlapping segments $\boldsymbol{S}_0^k, \ldots, \boldsymbol{S}_{l_k-1}^k$. The bits in $\boldsymbol{S}_i^k$ are the kth bits of CH codes on positions in G_j^i of $\widehat{E}$; the order of these bits is in accordance with the order of their positions in $\widehat{E}$. Denote the start position of $\boldsymbol{S}_i^k$ in $\boldsymbol{F}^k$ by $S(S_i^k)$ and the end position of $\boldsymbol{S}_i^k$ in $\boldsymbol{F}^k$ by $L(\boldsymbol{S}_i^k)$. $\boldsymbol{F}^1$ has only one segment $\boldsymbol{S}_\varepsilon^1 = F^1$ and $S(S_\varepsilon^1) = 1$, $L(S_\varepsilon^k) = n$, where ε is the "empty" string. Segments of $\boldsymbol{F}^k$, $1 < k \leqslant L$, are defined recursively as follows:

For t from 0 to $l_k/2 - 1$

$$\begin{cases} S(S_{2t}^k) = S(S_t^{k-1}) \\ L(S_{2t}^k) = S(S_t^{k-1}) + rank_0(S_t^{k-1}) - 1 \end{cases}$$

$$\begin{cases} S(S_{2t+1}^k) = S(S_t^{k-1}) + rank_0(S_t^{k-1}) \\ L(S_{2t+1}^k) = L(S_t^{k-1}) \end{cases}$$

An example of CH code based wavelet tree is shown in Fig. 2.

We store the bit vectors $\boldsymbol{F}^1, \ldots, \boldsymbol{F}^L$ in continuous memory, and take them as one bit vector, named $\boldsymbol{F}$. An integer array $LD[1 \ldots L-1] = c_1 - c_2,\ c_2 - c_3,\ c_i - c_{i+1},\ \ldots,\ c_{L-1} - c_L$ is employed to compute the beginning position in $\boldsymbol{F}$ of each bit vector. The bit vector $\boldsymbol{F}$ with its *rank* structures [13] and array LD are our main indexing data structures. Since the nodes that contain empty characters in wavelet tree do not need storing, $\boldsymbol{F}$ takes nH_0 bits, and the *rank* structures of $\boldsymbol{F}$ take $O(nH_0/\lg(nH_0))$ bits. LD can be stored in a succinct way that encodes $LD[i]$ with $\lg\lg n + \lg(LD[i])$ bits. The first $\lg\lg n$ bits store the value of $\lg(LD[i])$ and the next $\lg(LD[i])$ bits the value of $LD[i]$. Because $\sum_{i=1}^{L-1} LD[i] = n - c_L$, the LD is limited by $L(\lg\lg n + \lg n - \lg|\Sigma|)$ bits. In the worse case, it is $nH_0 + |\Sigma|\ (\lg\lg n + \lg n - \lg|\Sigma|)$. Thus the index together uses at most $nH_0 + |\Sigma|\ (\lg\lg n + \lg n - \lg|\Sigma|) + o(nH_0)$ bits.

In Huffman code based wavelet tree, the address of bit vector of each tree node should be stored explicitly, but these addresses can be computed on the fly in a constant time in canonical Huffman code based wavelet tree. The details will be given in the following section.

4.1. Multi-key rank and select algorithms

Based on bit vector $\boldsymbol{F}$, the algorithm to compute the number of occurrences of character b in a string $\boldsymbol{E}$ up to position *end* is given below.

```
rank_b(E, end)
 1: s ← 1; e ← n;c ← end;I ← F;len ← n
 2: for i ← 1 to LH(b) do
 3:    c ← rank_{b_i}(I[s ... e])
 4:    if b_i = 1 then
 5:       s ← s + rank_0(I[s...e])
 6:    else
 7:       e ← e − rank_1 (I[s...e])
 8:    end if
 9:    I ← I + len
10:    len ← len − LD[i]
11: end for
12: return c
```

In running of the function *rank*, after each loop i, the number of characters in $\boldsymbol{E}$ up to position *end*, whose first $k-1$ bits are $b_1 \ldots b_{k-1}$, say c, is available. The start position s and end position e of segment $\boldsymbol{S}_{b_1 \ldots b_i}^{i+1}$ in $\boldsymbol{F}^{i+1}$ are available. The start position of $\boldsymbol{F}^{i+1}$ in $\boldsymbol{F}$ is also computed. Therefore, when function *rank* returns, the number of bs in $\boldsymbol{E}$ up to position *end* is computed.

For function *rank* on bit vectors runs in constant time, each iteration of multi-key *rank* function runs in a constant time. The time of running of the loop inside *rank* varies with the length of the codeword of input character. Therefore, by this algorithm, $rank_b(\boldsymbol{E}, end)$ can be calculated within $O(|\Sigma|)$ time in the worst case and within $O(H_0)$ time on average. In other words, we can find the interval corresponding to $goto(s,a)$ in $O(H_0)$ time on average.

The function $rank^{\leqslant}$ can be computed using the same data structure by replacing the returning value c with $s + c - 1$. In FM-index [7] and Ref. [8], the same function is performed via a table of $|\Sigma|\lg n$ bits which stores the number of occurrences of each character in the text.

Function *select* is the reverse computing of function *rank*. $Select_b(\boldsymbol{E}, i)$ returns the index of the ith b character of $\boldsymbol{E}$. Its algorithm is given below.

i	1	2	3	4	5	6	7	8	9	10	11
E	**a**	**c**	**b**	**g**	**c**	**e**	**f**	**e**	**h**	**c**	**d**
$\mathbf{b_1}$	0	1	0	0	1	1	1	1	0	1	0
$\mathbf{b_2}$	0	1	0	1	1	0	0	0	1	1	0
$\mathbf{b_3}$	0		0	0		1	0	1	1		1
$\mathbf{b_4}$	0		1								

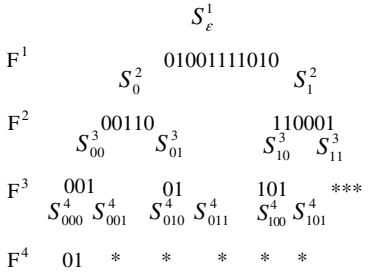

Fig. 2. Example of the bit vectors for $\boldsymbol{E}$ = acbgcefehcd (*denotes the empty character).

Select$_b(\boldsymbol{E}, count)$
1: $s \leftarrow 1$; $e \leftarrow n$; $len \leftarrow n$
2: **for** $i \leftarrow 1$ **to** $LH(b)$ **do**
3: **if** $b_i = 1$ **then**
4: $s_{i+1} \leftarrow s_i + rank_0(F^i[s_i..e_i])$
5: **else**
6: $c_{i+1} \leftarrow c_i - rank_1(F^i[s_i..e_i])$
7: **end if**
8: $\boldsymbol{F}^{i+1} \leftarrow \boldsymbol{F}^i + len$
9: $len \leftarrow len - LD[i]$
10: **end for**
11: $c \leftarrow count$
12: **for** $i = 1$ **to** $LH(b)$ **do**
13: $c \leftarrow select_{b_i}(F^i[s_i \ldots e_i], c)$
14: **end for**
15: **return** c

Denote s, e and c of each iteration in the running of the for loop of $rank_b(\boldsymbol{E}, end)$ by s_i,e_i *and* c_i, where i is the value of loop variable and $c_0 = end$, $\lambda = LH(b)$. The sequence of the computing of s_i,e_i *and* c_i in *rank* is $c_1 = rank_{b_1}(F^1[s_1 \ldots e_1], c_0)$, …, $c_{\lambda-1} = rank_{b_{\lambda-1}}(F^{\lambda-1}[s_{\lambda-1} \ldots c_{\lambda-1}], c_{\lambda-2})$, $count = rank_{b_\lambda}(F^\lambda[s_\lambda \ldots c_\lambda], c_{\lambda-1})$.The first for loop in lines 2–10 in function *select* is to compute $rank_b(\boldsymbol{E})$, therefore $s'_i = s_i$, $e'_i = e_i$. Denote the value of c in each for loop in lines 12–14 of *select* by c'_k (k is the value of loop variable) and the initial value of it by $c'_\lambda = count$, then $Select_b(E, count) = c'_0$. According to the function *select*, the sequence of computing of c_i is $c'_{\lambda-1} = select_{b_\lambda}(F^\lambda[s_\lambda..c_\lambda], count)$, $c'_{\lambda-2} = select_{b_{\lambda-1}}(F_{\lambda-1}[s_{\lambda-1}..c_{\lambda-1}], c'_{\lambda-1})$, …, $c'_0 = select_{b1}(F^1[s_1 \ldots c_1], c'_1)$, where s'_i and e'_i are replaced with s_i and e_i. It is the reverse computing of $rank_b(\boldsymbol{E}, end)$.We have $c'_{\lambda-1} = c_{\lambda-1}$, $c'_i = c_i$ for $0 \leqslant i \leqslant \lambda$. If $E[end]$=b, $c'_0 = c_0$. Then $select_b(\boldsymbol{E}, count)$=$end$.

By a similar discussion of *rank*, $select_b(\boldsymbol{E}, end)$ can be calculated in $O(|\Sigma|)$ time in the worst case and in $O(H_0)$ time on average.

4.2. On-line linear construction of the canonical Huffman code based wavelet tree

In practice, $\boldsymbol{F} = \boldsymbol{F}^1, \ldots, \boldsymbol{F}^L$ can be constructed in linear time, provided that the frequencies of characters occur in $\boldsymbol{E}$ are known. In construction, each codeword of $\widehat{E}$ is read one by one from $\widehat{E}[1]$ to $\widehat{E}[n]$, where $\widehat{E}$ is the CH encoding of $\boldsymbol{E}$, and each bit of the codeword is processed from left to right. In the procedure, a position of $\boldsymbol{F}$ is set according to an input bit. For $\widehat{E}[i]$, the kth bit, $1 \leqslant k \leqslant LH(E[i])$, of $\widehat{E}[i]$ is linked uniquely with a position x of $\boldsymbol{F}^k$, that is, $F^k_x = \widehat{E}[i]_k$. Function BP is used to represent this relation. $BP(i,k)$ returns the position x.

In construction, we use a binary tree to compute $BP(i,k)$. Consider the CH tree for $\boldsymbol{E}$, the tree has exactly $|\Sigma|$ leaves, one for each character of the alphabet, and exactly $|\Sigma| - 1$ internal nodes. Each leaf of tree is labeled with the character's frequency of occurrence and each internal node is labeled with the sum of frequencies of the leaves in its subtree. Each segment of $\boldsymbol{F}$ is corresponding to an internal node in this tree. Precisely, segment S^k_i corresponds to a node, say n^k_i, of depth k whose concatenation of path label from root to it is the binary representation of i, and the length of S^k_i is the label of the node. Then the start position of segment S^k_i, say $S(S^k_i)$, in $\boldsymbol{F}^k$ is the sum of labels of the nodes whose depth is k and whose path is lexically less than i, plus 1 (see Fig. 1(c)). The start position of segment S^k_i in $\boldsymbol{F}$ is $S(S^k_i)$ plus the sum of labels of the nodes whose depth is less than k. By replacing the label of each node with the start position in $\boldsymbol{F}$ of each segment, and deleting the leaf nodes, we build the tree we need, namely "segment tree". The label of a node n is denoted by $CurTail(n)$. In practice, the tree can be stored in continuous memory and constructed on line in linear time on $|\Sigma|$. We augment the segment tree with auxiliary state $\perp$ connected to eliminator ξ, such that $\xi a = \varepsilon$ of all characters $a \in \Sigma$. Therefore, $goto_{FT}(\perp, \alpha)$ equals $root = n^1_\varepsilon$ of segment tree for $\alpha \in \{0, 1\}$.

In the construction of $\boldsymbol{F}$, $\widehat{E}[1] = b_1 \ldots b_{LH(E[1])}$ is input initially, and $BP(1,1) = CurTail(n^1_\varepsilon) = 1$. Therefore, F^1_1 is set to $\widehat{E}[1]_1$, and $CurTail(n^1_\varepsilon)$ is increased by 1. For each depth k, let the current node be $n^k_{b_1 \ldots b_k}$, and $BP(1,k)$ equals $CurTail(n^k_{b_1 \ldots b_k})$, then $F^k_{CurTail(n^k_{b_1 \ldots b_k})}$ should be set to $\widehat{E}[1]_k$, and $CurTail(n^k_{b_1 \ldots b_k})$ be increased by 1. The same procedure is used to process $\widehat{E}[i]$. When all the characters in $\widehat{E}$are processed, $\boldsymbol{F}^1, \boldsymbol{F}^2, \ldots, \boldsymbol{F}^L$ are constructed successfully. The algorithm is given below.

Build_Index($\boldsymbol{E}$, $\boldsymbol{FT}$)
1: build segment tree $\boldsymbol{FT}$ of $\boldsymbol{E}$; $s \leftarrow 1$; $e \leftarrow n$
2: **for** $i = 1$ **to** n **do**
3: $c \leftarrow \perp$
4: $\alpha \leftarrow \widehat{E}[i]$
5: **for** $k = 1$ **to** $LH(E[i])$ **do**
6: $c \leftarrow goto_{FT}(c, \alpha_k)$
7: $\boldsymbol{F}[CurTail(c)] \leftarrow \alpha_{k+1}$
8: $CurTail(c) \leftarrow CurTail(c) + 1$
9: **end for**
10: **end for**
11: **return** $\boldsymbol{F}$

5. Experiments

We implement the canonical Huffman code based wavelet tree, Huffman code based wavelet tree and wavelet tree. And we used them to implement the succinct suffix automaton as full-text index. To compare fairly, all three structures were represented in space-economical ways. The tests ran on a Pentium processor at 2.4 GHz, 2 GB of RAM. The compiler is Microsoft Visual C++ 6.0. Chinese texts were from two Chinese classic novels and essays of Steven Cheung. Programs were executable files of PowerPoint and Spsswin program. These sequences are all read word by word (16 bits). In the test program, the address took 4 bytes.

Table 1
Size statistics of different compact indices

Source	$\|\Sigma\|$	Length (byte)	Index length (byte)		
			CHI	HI	WT
Program 1	63,759	6,146,760	11,961,456	12,059,749	16,565,164
Program 2	45,154	3,969,076	5,948,909	6,047,201	10,806,470
Chinese 1	4804	2,769,718	4,278,707	4,377,000	7,603,526
Chinese 2	2390	686,436	1,097,127	1,195,421	1,962,659

Table 1 compares the size of canonical Huffman code based index (CHI), Huffman code based index (HI) and wavelet tree based index (WT) for different sequences.

The canonical Huffman code based index takes about 1.5 times the text size, which is better than the other two indices. The memory for representing the shape of wavelet tree is not needed in CHI. This part of memory is related to the size of alphabet. In case of very large alphabets, for example a 32 bits word, even the RAM cannot hold it. But canonical Huffman code based index can work well in such cases without extra data structures.

6. Conclusions

We have presented a canonical Huffman code based wavelet tree structure. And we implement the suffix automaton in a space-economical way. The advantages of this structure over Huffman code based wavelet tree are that the tree structure need not store and operations on it are simple and efficient. We have also proposed a fast linear on-line construction algorithm for it. Compared with other indexes, it is better when applied to the text on large alphabet.

Acknowledgements

This work was supported by National Natural Science Foundation of China (Grant Nos. 60433020, 60673023), the Science Technology Development Project of Jilin Province of China (Grant No. 20050705-2), the European Commission under Grant No. TH/Asia Link/010 (111084), and 985 Project of Jilin University.

References

[1] Gusfield D. Algorithms on strings trees and sequences. New York: Cambridge University Press; 1997, p. 87–207.
[2] Blumer A, Blumer J, Haussler D, et al. The smallest automation recognizing the subwords of a text. Theor Comput Sci 1985;40:31–55.
[3] Manber U, Myers G. Suffix arrays: a new method for on-line string searches. SIAM J Comput 1993;22(10):935–48.
[4] Gonnet G, Baeza-Yates R, Snider T. New indices for text: PAT trees and PAT arrays. In: Proceedings of information retrieval: algorithms and data structures. Englewood Cliffs: Prentice-Hall; 1992, p. 66–82.
[5] Grossi R,Vitter J. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proceedings of the 32nd annual ACM symposium on theory of computing, Portland, USA; 2000. p. 397–406.
[6] Grossi R, Gupta A, Vitter J. When indexing equals compression: experiments with compressing suffix arrays and applications. In: Proceedings of the 15th annual ACM-SIAM symposium on discrete algorithms, New Orleans, USA; 2004. p. 636–45.
[7] Ferragina P, Manzini G. Opportunistic data structures with applications. In: Proceedings of the 41st annual symposium on foundations of computer science, CA, USA; 2000. p. 390–98.
[8] He M, Munro JI, Rao SS. A categorization theorem on suffix arrays with applications to space efficient text indexes. In: Proceedings of the 16th annual ACM-SIAM symposium on discrete algorithms, Vancouver, Canada; 2005. p. 23–32.
[9] Sadakane K. Compressed text databases with efficient query algorithms based on the compressed suffix arrays. In: Proceedings of the 11th international symposium on algorithms and computation, Taipei, China; 2000. p. 410–21.
[10] Gupta A, Hon WK, Shah R, et al. Compressed data structures: dictionaries and data-aware measures. In: Proceedings of data compression conference, Snowbird, UT, USA; 2006. p. 213–22.
[11] Zhang M, Tang J, Guo D, et al. Succinct text indexes on large alphabet. In: Proceedings of theory and applications of models of computation, the third international conference, Beijing, China; 2006. p. 528–37.
[12] Ferragina P, Manzini G, Makinen V, et al. An alphabet-friendly FM-index. In: Proceedings of string processing and information retrieval, 11th international conference, Padova, Italy; 2004. p. 150–60.
[13] Jacobson G. Space-efficient static trees and graphs. In: Proceedings of the 30th IEEE symposium on foundations of computer science, NC, USA; 1989. p. 549–54.
[14] Munro JI. Tables. In: Proceedings of the 16th conference on foundations of software technology and computer science, Hyderabad, India; 1996. p. 37–42.