Advanced Data Structures

Johannes Fischer

WS 2012/13

1 Compressed Suffix Arrays

We will show in this section that $O(n \log \sigma)$ bits suffice also for representing A. The price of this *compressed suffix array* is that the time for retrieving an entry from A is not constant any more, but rises from $O(1)$ to $O(\log^\epsilon n)$, for some arbitrarily small constant $0 < \epsilon \leq 1$.

1.1 Recommended Reading

- K. Sadakane *New Text Indexing Functionalities in the Compressed Suffix Arrays.* J. Algorithms 48(2): 294-313 (2003).
- G. Navarro and V. Mäkinen: *Compressed Full-Text Indexes.* ACM Computing Surveys 39(1), Article no. 2 (61 pages), 2007. Sect. 4.4, 4.5, 7.1.

1.2 The ψ-Function

The most important component of the compressed suffix array (abbreviated as CSA henceforth) is a function ψ that allows us to "jump one character forward" in the suffix array.

Definition 1. *Define ψ: $[1, n] \to [1, n]$ such that $\psi(i) = j \Leftrightarrow A[j] = A[i] + 1$, where position $n + 1$ is interpreted as the first position in T (read text circularly!).*

Example 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T=	C	A	C	A	A	T	A	C	A	T	T	A	T	A	C	$
A=	16	4	14	2	7	12	5	9	15	3	1	8	13	6	11	10
ψ=	11	7	9	10	12	13	14	16	1	2	4	8	3	5	6	15

Note the similarity of the ψ-function to *suffix links* in suffix trees: both "cut off" the first character of the corresponding substring.

Function ψ is *increasing* in areas where the corresponding suffixes start with the same character. For instance, in Ex. 1 we have that all suffixes from $A[2, 9]$ start with letter A; and indeed, $\psi[2, 9] = [7, 9, 10, 12, 13, 14, 16]$ is increasing. This is summarized in the following lemma.

Lemma 1. *If $i < j$ and $T_{A[i]} = T_{A[j]}$, then $\psi(i) < \psi(j)$.*

This lemma will be used in Sect. 1.6 to store ψ in a space-efficient form.

1.3 The Idea of the Compressed Suffix Array

We now present the general approach to store A in a space-efficient form. Instead of storing every entry in A, in a new bit-vector $B_0[1,n]$ we mark the positions in A where the corresponding entry in A is even:

$$B_0[i] = 1 \Leftrightarrow A[i] \equiv 0 \pmod{2} \ .$$

Bit-vector B_0 is prepared for $O(1)$ RANK-queries. We further store the ψ-values at positions i with $B_0[i] = 0$ in a new array $\psi_0[1, \lceil \frac{n}{2} \rceil]$. Finally, we store the even values of A in a new array $A_1[1, \lfloor \frac{n}{2} \rfloor]$, and divide all values in A_1 by 2.

Example 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T=$	C	A	C	A	A	T	A	C	A	T	T	A	T	A	C	\$
$A=$	16	4	14	2	7	12	5	9	15	3	1	8	13	6	11	10
$B_0=$	1	1	1	1	0	1	0	0	0	0	0	1	0	1	0	1
$\psi_0=$					12		14	16	1	2	4		3		6	
$A_1=$	8	2	7	1	6							4		3		5

Now, the three arrays, B_0, ψ_0 and A_1, completely substitute A: to retrieve value $A[i]$, we first check if $B_0[i] = 1$. If so, we know that $A[i]/2$ is stored in A_1, and that the exact position in A_1 is given by the number of 1-bits in B_0 up to position i. Hence, $A[i] = 2A_1[\text{RANK}_1(B_0, i)]$.

If, on the other hand, $B_0[i] = 0$, we follow $\psi(i)$ in order to get to the position of the $(A[i]+1)$st suffix, which must be even (and is hence stored in A_1). The value $\psi(i)$ is stored in ψ_0, and its position therein is equal to the number of 0-bits in B_0 up to position i. Hence, $A[i] = A[\psi_0(\text{RANK}_0(B_0, i))]-1$, which can be calculated be the mechanism of the previous paragraph.

As we shall see later, ψ_0 can be stored very efficiently (basically using $O(n \log \sigma)$ bits). Hence, we have almost halved the space with this approach (from $n \log n$ bits for A to $\frac{n}{2} \log \frac{n}{2}$ for A_1).

1.4 Hierarchical Decomposition

We can use the idea from the previous section recursively in order to gain more space: instead of representing A_1 plainly, we replace it with bit-vector B_1, array ψ_1 and A_2. Array A_2 can in turn be replaced by B_2, ψ_2, and A_3, and so on. In general, array $A_k[1, n_k]$, with $n_k = \frac{n}{2^k}$, implicitly represents T's suffixes that are a multiple of 2^k, in the order as they appear in the original array $A_0 := A$.

Example 3.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T=$	C	A	C	A	A	T	A	C	A	T	T	A	T	A	C	\$
$A=$	16	4	14	2	7	12	5	9	15	3	1	8	13	6	11	10

ψ_0= 12 14 16 1 2 4 3 6

B_0=1 1 1 1 0 1 0 0 0 0 0 1 0 1 0 1

A_1=8 2 7 1 6 4 3 5

level 0

ψ_1=1 2 6 5

B_1=1 1 0 0 1 1 0 0

A_2=4 1 3 2

level 1

etc.

A_k can be seen as a suffix array of a new string T^k, where the i'th character of T^k is the concatenation of 2^k characters $T_{i2^k\dots(i+1)2^k-1}$ (we assume that T is padded with sufficiently enough \$-characters). This means that the alphabet for T^k is Σ^{2^k}, i. e., all 2^k-tuples from Σ.

Example 4. $A_2 = [4, 1, 3, 2]$ *can be regarded as the suffix array of*

$$T^2 = \underbrace{(\mathrm{AATA})}_{T^2_1}\underbrace{(\mathrm{CATT})}_{T^2_2}\underbrace{(\mathrm{ATAC})}_{T^2_3}\underbrace{(\$\$\$\$)}_{T^2_4}\ .$$

This way, on level k we only store B_k and ψ_k. Only on the last level h we store A_h. We choose $h = \lceil \log\log_\sigma \frac{n}{\log n}\rceil$ such that the space for storing A_h is

$$O\left(n_h \log n_h\right) = O\left(n_h \log n\right) = O\left(\frac{n}{2^h}\log n\right) = O\left(\frac{n\log\sigma}{\log\frac{n}{\log n}}\log n\right) = O(n\log\sigma) \text{ bits.}$$

However, storing B_k and ψ_k on all h levels would take too much space. Instead, we use only a *constant* number of $1 + \frac{1}{\epsilon}$ levels, namely 0, $h\epsilon$, $2h\epsilon$, …,h (constant $0 < \epsilon \le 1$).

Example 5.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T=$	C	A	C	A	A	T	A	C	A	T	T	A	T	A	C	\$
$A_0=$	16	4	14	2	7	12	5	9	15	3	1	8	13	6	11	10
$\psi_0=$	9	10	12	14	16	1	2	4	3	5	6	15				
$B_0=$	1	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0
$A_2=$	4	1	3	2												
$\psi_2=$	4	1	3													
$B_2=$	1	0	0	0												
$A_4=$	1															

n=16

h=4

$\epsilon=\frac{1}{2}$

Hence, bit-vector B_k has a '1' at position i iff $A_k[i]$ is a multiple of $2^{h\epsilon+k}$.
Given all this, we have the following algorithm to compute $A[i]$, to be invoked with `lookup`$(i, 0)$.

Algorithm 1: function `lookup`(i, k)

if $k = h$ **then**
  return $A_h[i]$;
end
if $k = \omega_k$ **then**
  return n_k;
end
if $B_k[i] = 1$ **then**
  return $2^{h\epsilon}$`lookup`$(\text{RANK}_1(B_k, i), k + h\epsilon)$;
else
  return `lookup`$(\psi_k(\text{RANK}_0(B_k, i), k)) - 1$;
end

Here, ω_k stores the position of the last suffix, i. e., $A_k[\omega_k] = n_k$. Checking if $i = \omega_k$ is necessary in order to avoid following ψ_k from the last suffixes to the first, because this would give incorrect results.

Example 6. $A[15] = \texttt{lookup}(15, 0) = \texttt{lookup}(\psi_0(11), 0) - 1 = \texttt{lookup}(6, 0) - 1 = 2^2\texttt{lookup}(3, 2) - 1 = 2^2(\texttt{lookup}(\psi_2(2), 2) - 1) - 1 = 2^2(\texttt{lookup}(1, 2) - 1) - 1 = 2^2(4 - 1) - 1 = 11$

To analyze the running time of the `lookup`-procedure, we first note that on every level k, we need to follow ψ_k at most $2^{h\epsilon}$ times until we hit a position i with $B_k[i] = 1$ (second case of the last if-statement). Because the number of "implemented" levels, $1 + \frac{1}{\epsilon}$, is constant (remember ϵ is constant!), the total time of the `lookup`-procedure is

$$O\left(2^{h\epsilon}\right) = O\left(\left(2^{\log\log_\sigma n}\right)^\epsilon\right) = O\left(\log_\sigma^\epsilon n\right) ,$$

which is sub-logarithmic for $\epsilon < 1$.

1.5 Elias-Codes

For coding the ψ-values in a space efficient form, we will use *Elias-*γ and *Elias-*δ codes, which we present in this section. Let us write $(x)_2$ for the *binary* representation of integer $x \geq 1$. Also $(x)_1$ denotes the *unary* representation of x, which consists of $x-1$ 0's, followed by a single 1. For example, $(5)_2 = 101$ and $(5)_1 = 00001$.
The Elias-γ code of a number x, denoted by $(x)_\gamma$, is defined as follows: first, write the length of the binary representation of x in unary, i. e., write bits $(|(x)_2|)_1$. Then append the bits from $(x)_2$, with the first (leftmost) '1' being omitted. For example, the first five γ-codes (representing the numbers $1, 2, \ldots, 5$) are 1, 010, 011, 00100 and 00101. The length in bits is

$$|(x)_\gamma| = \underbrace{\lfloor \log x \rfloor + 1}_{\text{unary part}} + \underbrace{\lfloor \log x \rfloor}_{\text{binary part}} .$$

The δ-code is obtained in a similar manner, but instead of encoding $|(x)_2|$ in unary, we encode it with the γ-code. That is, we first write $(|(x)_2|)_\gamma$, and then append $(x)_2$, again with the trailing '1' being omitted. Examples of δ-codes are shown in the following table.

Example 7.

x	$(x)_\delta$	x	$(x)_\delta$
1	1	9	00100001
2	0100	10	00100010
3	0101	11	00100011
4	01100	12	00100100
5	01101	13	00100101
6	01110	14	00100110
7	01111	15	00100111
8	00100000	16	001010000

The size of the δ-code is

$$\begin{aligned} |(x)_\delta| &= |(\lfloor \log x \rfloor + 1)_\gamma| + \lfloor \log x \rfloor \\ &= (\lfloor \log(\lfloor \log x \rfloor + 1) \rfloor + 1) + \lfloor \log(\lfloor \log x \rfloor + 1) \rfloor + \lfloor \log x \rfloor \\ &= \lfloor \log x \rfloor + 2\lfloor \log(\lfloor \log x \rfloor + 1) \rfloor + 1 \text{ bits.} \end{aligned}$$

1.6 Storing ψ

Let us first concentrate on level 0, i. e., on storing ψ_0. From Lemma 1, we know that ψ is piecewise increasing in areas $A[l, r]$ where the suffixes start with the same character (i. e., where $T_{A[i]} = T_{A[j]}$ for all $i, j \in [l, r]$). Let $[l, r]$ be one such area. Instead of storing $\psi_0[l, r]$ plainly, we first compute the differences $\Delta_0[i] = \psi_0[i] - \psi_0[i-1]$ for $l < i \leq r$. This produces a list of positive integers from the range $[1, n]$, which will be encoded space-efficiently in a subsequent step. In general, we define

$$\Delta_0[i] = \begin{cases} \psi_0[i] - \psi_0[i-1] & \text{if } T_{A_0[i]} = T_{A_0[i-1]}, \\ \psi_0[i] & \text{otherwise.} \end{cases}$$

Example 8.

$$\Delta_0 = \overset{1}{9}\,\overset{2}{1}\,\overset{3}{2}\,\overset{4}{2}\,\overset{5}{2}\,|\,\overset{6}{1}\,\overset{7}{1}\,\overset{8}{2}\,|\,\overset{9}{3}\,\overset{10}{2}\,\overset{11}{1}\,\overset{12}{9}$$

These Δ-values are now encoded with Elias δ-code; the resulting bit stream is called S_0.

Example 9.

$$\begin{array}{rcccccccccccccc} \Delta_0 = & 9 & 1 & 2 & 2 & 2 & 1 & 1 & 2 & 3 & 2 & 1 & 9 \\ S_0 = & 00100001 & 1 & 0100 & 0100 & 0100 & 1 & 1 & 0100 & 0101 & 0100 & 1 & 0010001 \end{array}$$

In general, because A_k can be regarded as the suffix array of a text T^k, we can compress ψ_k on levels $k > 0$ by the same mechanism, i. e., by using Elias δ-codes on the list of differences of consecutive ψ_k-values. We therefore define

$$\Delta_k[i] = \begin{cases} \psi_k[i] - \psi_k[i-1] & \text{if } T^k_{A_k[i]} = T^k_{A_k[i-1]}, \\ \psi_k[i] & \text{otherwise.} \end{cases}$$

How can we decompress the ψ_k-values from the stream S_k of δ-encoded Δ_k-values? For this purpose we store $\psi_k[i]$ *explicitly* if either position i marks the beginning of a new character in T^k (second case in the definition of Δ_k), or if the length of the encoded bit-stream since the last sampled ψ_k-value exceeds $s = \frac{\log n}{2}$ bits. To implement this, we introduce three new arrays:

1. D_k is a bit vector such that $D_k[i] = 1$ iff $\psi_k[i]$ is sampled. D_k is enhanced with data structures for constant-time RANK and SELECT queries.

2. R_k is an array that stores the sampled values of ψ_k. All ψ_k-values stored in R_k are removed from the bit-stream S_k (they need not to be stored twice!).

3. P_k is a bit stream of the same size as S_k and marks those positions in S_k with a '1' where a δ-encoded Δ_k-value starts. P_k is prepared for $O(1)$ SELECT_1-queries. Then $\text{SELECT}_1(P_k, i)$ points to the i'th Δ_k-value $S_k[i]$.

Example 10. Assuming $s = 5$, we have the following structures:

$$\begin{array}{rl} \psi_0 = & 9\ 10\ 12\ 14\ 16\ 1\ 2\ 4\ 3\ 5\ 6\ 15 \\ \Delta_0 = & 9\ 1\ \ 2\ \ 2\ \ 2\ \ 1\ 1\ 2\ 3\ 2\ 1\ 9 \\ D_0 = & 1\ 0\ \ 0\ \ 1\ \ 0\ \ 1\ 0\ 0\ 1\ 0\ 0\ 1 \\ & \\ R_0 = & 9\ 14\ 1\ 3\ 15 \\ P_0 = & \qquad\qquad\quad 11000 \qquad\quad 1000 \qquad 11000 \qquad\quad 10001 \\ S_0 = & (00100001)\ \underset{1\,2\,3\,4\,5}{10100}\ (0100)\ \underset{6\,7\,8\,9\ldots}{0100}\ (1)\ 10100\ (0101)\ 01001\ (00100001) \end{array}$$

We can decode $\psi_k[i]$ as follows. First compute the number of sampled Δ_k-values up to position i by $y = \text{RANK}_1(D_k, i)$. Then check if $\Delta_k[i]$ is represented explicitly ($D_k[i] = 1$), and return $R_k[y]$ in this case. Otherwise ($D_k[i] = 0$), compute the greatest index j such that ψ_k is sampled by $j = \text{SELECT}_1(D_k, y)$. The result is then $R_k[y]$ ($= \Delta_k[j]$), plus the sum of the $(i - j)$ values $\Delta_k[j+1], \ldots, \Delta_k[i]$ that follow $\Delta_k[j]$ in S_k. Note that $D_k[j+1] = 0$, and that the 0's in D_k corresponds to the 1's in P_k. As $\Delta_k[j+1]$ is the z'th encoded Δ_k-value in S_k, with $z = \text{RANK}_0(D_k, j+$

$1) = j+1-\text{RANK}_1(D_k, j+1) = j+1-y$, we thus go to position $\text{SELECT}_1(P_k, z)$ in S_k, from where we decode the values $\Delta_k[j+1], \ldots, \Delta_k[i]$, and return $R_k[y] + \sum_{l=j+1}^{i} \Delta_k[l]$ as the result $\psi_k[i]$. This decoding is possible because the δ-code is prefix-free (no codeword is a prefix of a different codeword).

To compute this sum in $O(1)$ time, we use again the *Four-Russians-Trick*: in a global lookup-table G, for all bit-vectors V of length s and all positions $i \in [1, s]$, $G[V][i]$ stores the answer to $\sum_{j=1}^{i} y_j$, if we interpret V as a sequence of δ-encoded values $y_1, y_2, \ldots$. Note that some values in G are undefined, because not at all positions $i \in [1, s]$ there ends a δ-encoded value in V, and not all bit-vectors V represent a correct sequence of δ-codes, but these values will never be accessed by the algorithm.

Example 11.

G: $s = 5$

V	1	2	3	4	5
00000	-	-	-	-	-
.					
.					
.					
10100	1	3	-	-	-
.					
.					
.					
11111	1	2	3	4	5

1.7 Space Analysis

We now analyze the space requirement of the compressed suffix array. Recall that on level $k < h$, we store bit-vectors B_k, D_k, S_k, and P_k (plus some data structures for RANK and SELECT), and array R_k. On level h, we only store A_h, which needs $O(n \log \sigma)$ bits. Thus it remains to be shown that an level $k < h$ the space is $O(n \log \sigma)$ bits. Then the total space on all $1 + \frac{1}{\epsilon}$ levels is $O\left(\frac{1}{\epsilon} n \log \sigma\right)$ bits.

The bit-vectors B_k and D_k are certainly of size $O(n)$ bits each, as they are never longer than n, the length of the text. Actually, the *total* size of all B_k's can be bounded by $2n$ bits, because the length of the B_k-vectors is at least halved from one level to the next:

$$\sum_{k=0}^{h-1} |B_k| = \sum_{k=0}^{h-1} n_k = \sum_{k=0}^{h-1} \frac{n}{2^k} = n \sum_{k=0}^{h-1} \frac{1}{2^k} \leq n \sum_{k=0}^{\infty} \frac{1}{2^k} = 2n \ .$$

The total size of the D_k's is even smaller. Together with the data structures for constant-time RANK- and SELECT-queries, the space for all B_k's and D_k's can be upper bounded by $4n + o(n)$ bits in total.

Let us now analyze the space for the bit-stream S_k on a fixed level $k < h$. For simplicity, we assume that S_k stores *all* Δ_k-values, also those that are stored explicitly in R_k and thus deleted from S_k. Let n_k^c denote the number of positions in ψ_k corresponding to suffixes that start with the same character $c \in \Sigma^{2^k}$, and let $\Delta_k^c[1, n_k^c]$ denote the corresponding sub-array in Δ_k. Thus, by Lemma 1, S_k stores at most σ^{2^k} increasing sequences from the range $[1, n_k]$, each encoded by δ-codes of the differences Δ_k. Therefore, the space is

$$\begin{aligned}
|S_k| &= \sum_{c \in \Sigma^{2^k}} \sum_{i=1}^{n_k^c} \left(\lfloor \log \Delta_k^c[i] \rfloor + 2 \lfloor \log \left(\lfloor \log \Delta_k^c[i] \rfloor + 1 \right) \rfloor + 1 \right) \\
&= \sum_{c \in \Sigma^{2^k}} \sum_{i=1}^{n_k^c} \left(\lfloor \log \Delta_k^c[i] \rfloor + 2 \log \log \Delta_k^c[i] \right) + O(n_k) \\
&\leq \sum_{c \in \Sigma^{2^k}} \sum_{i=1}^{n_k^c} \left(\log \frac{n_k}{n_k^c} + 2 \log \log \frac{n_k}{n_k^c} \right) + O(n_k) \\
&= \sum_{c \in \Sigma^{2^k}} n_k^c \left(\log \frac{n_k}{n_k^c} + 2 \log \log \frac{n_k}{n_k^c} \right) + O(n_k) \\
&\leq \sum_{c \in \Sigma^{2^k}} n_k^c \left(\log \sigma^{2^k} + 2 \log \log \sigma^{2^k} \right) + O(n_k) \\
&= \left(\log \sigma^{2^k} + 2 \log \log \sigma^{2^k} \right) \sum_{c \in \Sigma^{2^k}} n_k^c + O(n_k) \\
&= \left(2^k \log \sigma + 2 \log 2^k \log \sigma \right) n_k + O(n_k) \\
&= \left(2^k \log \sigma + 2 \log 2^k \log \sigma \right) \frac{n}{2^k} + O(n_k) \\
&= n \log \sigma + O(n \log \log \sigma) \text{ bits.}
\end{aligned}$$

Here, both inequalities follow from the fact that the sum of logarithms is largest when the values are spread *evenly* over the interval: if $\sum_{i=1}^{m} x_i \leq x$ for a sequence of m real numbers with $x_i \geq 1$ for all i, then $\sum_{i=1}^{m} \log x_i \leq \sum_{i=1}^{m} \log \frac{x}{m}$.

Because P_k is of the same size as S_k, we can upper bound the space for P_k (including the data-structure for SELECT) by $O(n \log \sigma)$ bits.

Finally, the array R_k of sampled values consist of

$$\begin{aligned}
|R_k| &= \Bigg(\underbrace{|\Sigma^{2^k}|}_{\text{new character}} + \underbrace{\frac{|S_k|}{\log n}}_{\text{length exceeds } s \text{ bits}} \Bigg) \times \underbrace{\log n_k}_{\text{value from } [1, n_k]} \\
&= \left(\sigma^{2^k} + \frac{n \log \sigma}{\log n} \right) \log n_k \\
&\leq O\left(\left(\sigma^{2^h} + \frac{n \log \sigma}{\log n} \right) \log n \right) \\
&= O\left(\left(\frac{n}{\log n} + \frac{n \log \sigma}{\log n} \right) \log n \right) \\
&= O(n \log \sigma) \text{ bits.}
\end{aligned}$$

We summarize this section in a final theorem:

Theorem 2. *The suffix array A of a text of length n over an alphabet of size σ can be stored in $O\left(\frac{1}{\epsilon} n \log \sigma\right)$ bits such that retrieving an arbitrary entry $A[i]$ from the suffix array with $1 \leq i \leq n$ takes $O(\log^{\epsilon} n)$ time.* $\square$