

## Lecture 18 — April 26, 2012

Prof. Erik Demaine

Scribes: Sanja Popovic(2012),

Jean-Baptiste Nivoit(2012), Jon Schneider(2012),

Sarah Eisenstat (2010), Martí Bolívar (2007)

## 1 Overview

In the last lecture we introduced the concept of implicit, succinct, and compact data structures, and gave examples for succinct binary tries, as well as showing the equivalence of binary tries, rooted ordered trees, and balanced parentheses expressions. Succinct data structures were introduced which solve the *rank* and *select* problems.

In this lecture, we introduce compact data structures for suffix arrays and suffix trees. Recall the problem that we are trying to solve. Given a text  $T$  over the alphabet  $\Sigma$ , we wish to preprocess  $T$  to create a data structure. We then want to be able to use this data structure to search for a pattern  $P$ , also over  $\Sigma$ .

A suffix array is an array containing all of the suffixes of  $T$  in lexicographic order. In the interest of space, each entry in the suffix array stores an index in  $T$ , the start of the suffix in question. To find a pattern  $P$  in the suffix array, we perform binary search on all suffixes, which gives us all of the positions of  $P$  in  $T$ .

## 2 Survey

In this section, we give a brief survey of results for compact suffix arrays. Recall that a compact data structure uses  $O(OPT)$  bits, where  $OPT$  is the information-theoretic optimum. For a suffix array, we need  $|T| \lg |\Sigma|$  bits just to store the text  $T$ .

### 2.1 Compact suffix arrays and trees

**Grossi and Vitter 2000 [3]** Suffix array in

$$\left(\frac{1}{\varepsilon} + O(1)\right) |T| \lg |\Sigma|$$

bits, with query time

$$O\left(\frac{|P|}{\log_{|\Sigma|}^{\varepsilon} |T|} + |\text{output}| \cdot \log_{|\Sigma|}^{\varepsilon} |T|\right)$$

We will follow this paper fairly closely in our discussion today. Note how one can trade better query time for higher space usage.

**Ferragina and Manzini 2000 [1]** The space required is

$$5H_k(T) \cdot |T| + O\left(|T| \cdot \frac{|\Sigma| + \lg |T|}{\lg \lg |T|} + |T|^\varepsilon \cdot |\Sigma|^{|\Sigma|+1}\right)$$

bits, for all fixed values of  $k$ .  $H_k(T)$  is the  $k^{\text{th}}$ -order empirical entropy, or the regular entropy conditioned on knowing the previous  $k$  characters. More formally:

$$H_k(T) = \sum_{|w|=k} \Pr\{w \text{ occurs}\} \cdot H_0(\text{characters following an occurrence of } w \text{ in } T).$$

Note that because we are calculating this in the empirical case,

$$\Pr\{w \text{ occurs}\} = \frac{\# \text{ of occurrences of } w}{|T|}.$$

For this data structure, query time is

$$O(|P| + |\text{output}| \cdot \lg^\varepsilon |T|).$$

**Sadakane 2003 [5]** Space in bits is

$$\frac{1 + \varepsilon'}{\varepsilon} H_0(T) |T| + O(|T| \lg \lg |\Sigma| + |\Sigma| \lg |\Sigma|),$$

and query time is

$$O(|P| \lg |T| + |\text{output}| \lg^\varepsilon |T|).$$

where  $0 < \varepsilon < 1$  and  $\varepsilon' > 0$  are arbitrary constants. Note that this bound is more like a suffix array, due to the multiplicative log factor. This bound is good for large alphabets.

## 2.2 Succinct suffix arrays and trees

**Grossi, Gupta, Vitter 2003 [2]** Space in bits is

$$H_k(T) \cdot |T| + O\left(|T| \lg |\Sigma| \cdot \frac{\lg \lg |T|}{\lg |T|}\right),$$

and query time is

$$O(|P| \lg |\Sigma| + \frac{\lg^2 |T|}{\lg \lg |T|}).$$

**Ferragina, Manzini, Mäkinen, Navarro [6]** Space in bits is

$$H_k(T) \cdot |T| + O\left(\frac{|T|}{\lg^\varepsilon n}\right),$$

and query time is

$$O(|P| + |\text{output}| \cdot \lg^{1+\varepsilon} |T|).$$

Also exhibits  $O(|P|)$  running time for counting queries.

## 2.3 Extras

**Hon, Sadakane, Sung 2003 [8] and Hon, Lam, Sadakane, Sung, Yiu 2007 [7]** details low-space construction with  $O(|T| \cdot \lg |\Sigma|)$  working space.

**Hon, Sadakane 2002 [9] and Sadakane 2007 [10]** detail suffix-tree operations like maximal common substrings.

**Sadakane 2007 [11]** solves the document retrieval problem on those structures.

**Chan, Hon, Lam, Sadakane 2007 [12]** describes the dynamic version of the problem.

## 3 Compressed suffix arrays

For the rest of these notes, we will assume that the alphabet is binary (in other words, that  $|\Sigma| = 2$ ). In this section, we will cover a simplified (and less space-efficient) data structure, which we will adapt in the next section for the compact data structure. Overall, we will cover a simplified version of Grossi's and Vitter's paper [3]. We will achieve the same space bound, but slightly worse query time.

The problem we are solving with suffix arrays is to find  $SA[k]$ , i.e. where does the  $k$ -th suffix start, assuming suffixes are sorted in lexicographic order. The data structure we will be using is similar to the one seen in Lecture 16. We are still using the same divide-and-conquer approach, except that it is 2-way instead of 3-way.

### 3.1 Top-Down

Let us introduce some notation we will use throughout the notes.

**start:** The initial text is  $T_0 = T$ , the initial size  $n_0 = n$ , and the initial suffix array  $SA_0 = SA$  where  $SA$  is the suffix array of  $T$ .

**step:** In every successive iteration, we are combining two adjacent letters:

$$T_{k+1} = \langle (T_k[2i], T_k[2i+1]) \text{ for } i = 0, 1, \dots, \frac{n}{2} \rangle.$$

This means the size is halved in each step:

$$n_{k+1} = \frac{n_k}{2} = \frac{n}{2^k}.$$

We define a recursive suffix tree as

$$SA_{k+1} = \frac{1}{2} \cdot (\text{extract even entries of } SA_k)$$

where “even entries” are defined to be suffixes whose index in  $T_k$  are even.

Clearly, it is fairly easy to calculate  $SA_{k+1}$  from  $SA_k$ , and since  $SA_0$  is known, this means that we can go top-down without much difficulty. However, in order to make this data structure work, we need to go bottom-up.

## 3.2 Bottom-Up

We need a way to represent  $SA_k$  using  $SA_{k+1}$ . To do so, we define the following functions:

*is-even-suffix<sub>k</sub>*( $i$ ) This tells us whether  $SA_k[i]$  is an even suffix. More formally:

$$is\text{-}even\text{-}suffix_k(i) = \begin{cases} 1 & \text{if } SA_k[i] \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

*even-succ<sub>k</sub>*( $i$ ) Returns  $j$ -th suffix such that  $j \geq i$  and  $SA_k[j]$  is even. More compact:

$$even\text{-}succ_k(i) = \begin{cases} i & \text{if } SA_k[i] \text{ is even} \\ j & \text{if } SA_k[i] = SA_k[j] - 1 \text{ is odd} \end{cases}$$

*even-rank<sub>k</sub>*( $i$ ) The “even rank” of  $i$  is the number of even suffixes preceding the  $i^{\text{th}}$  suffix.

Using this functions, we can write:

$$SA_k[i] = 2 \cdot SA_{k+1}[even\text{-}rank_k(even\text{-}succ_k(i))] - (1 - is\text{-}even\text{-}suffix_k(i))$$

Here, the factor of 2 is decompressing the size of the array. If the predicate *is-even-suffix<sub>k</sub>*( $i$ ) is true, *even-succ<sub>k</sub>*( $i$ ) =  $i$ , so this is equivalent to saying

$$SA_k[i] = 2 \cdot SA_{k+1}[even\text{-}rank_k(i)].$$

This basically means that we are looking up the correct value in the array  $SA_{k+1}$ . If *is-even-suffix<sub>k</sub>*( $i$ ) is false, then this is equivalent to performing the same action on  $i$ 's “even successor” — which is the index into  $SA_k$  of the suffix starting one position after  $SA_k[i]$  — and then subtracting 1 to get the correct starting position in the text  $T_k$ . We use  $(1 - is\text{-}even\text{-}suffix_k(i))$  instead of *is-odd-suffix<sub>k</sub>*( $i$ ) because we can use it to calculate *even-rank<sub>k</sub>*( $i$ ) as  $even\text{-}rank_k(i) = rank_1(is\text{-}even\text{-}suffix_k(i))$ .

If we can perform the above operations in constant time and a small amount of space, then we can reduce a query on  $SA_k$  to a query on  $SA_{k+1}$  in constant time. Hence, a query on  $SA_0$  will take  $O(\ell)$  time if our maximum recursion depth is  $\ell$ . If we set  $\ell = \lg \lg n$ , we will reduce the size of the text to  $n_\ell = n / \lg n$ . We can then use a normal suffix array, which will use  $O(n_\ell \lg n_\ell) = O(n)$  bits of space, and thus be compressed. If we want to, we can further improve by setting  $l = 2 \lg \lg n$  and get  $n_l = n / \lg^2 n$ .

## 3.3 Construction

We can store *is-even-suffix<sub>k</sub>*( $i$ ) as a bit vector of size  $n_k$ . Because  $n_k$  decreases by a factor of two with each level (geometric series), this takes a total of  $O(n)$  space. Then we can implement *even-rank<sub>k</sub>*( $i$ ) with the rank from last lecture on our bit vector, requiring  $o(n_k)$  space per level, for a total of  $O(n)$  space.

Notice that *even-rank<sub>k</sub>*( $i$ ) is  $rank_1(i)$  on the array where  $a[i] = is\text{-}even\text{-}suffix_k(i)$ . We saw in lecture 17 how to do that in  $O(n_k \frac{\lg \lg n_k}{\lg n_k})$ . Again, this decreases geometrically, so overall it takes  $o(n)$  space.

Doing *even-succ<sub>k</sub>*( $i$ ) is trivial in the case that  $SA_k[i]$  is even because it is an identity function. This leaves  $n_k / \text{odd values}$ , but we cannot store them explicitly because each takes  $\lg n_k$  bits.

Whatever data structure we use, let's order the values of  $j$  by  $i$ ; that is, if the answers are stored in array called *odd<sub>a</sub>answers*, then we would have  $even\text{-}succ_k(i) = odd\text{-}answers[i - even\text{-}rank_k(i)]$ , because  $i - even\text{-}rank_k(i)$  is the index of  $i$  among odd suffixes. This ordering is equivalent to ordering by the suffix in the suffix array, or  $T_k[SA_k[i] :]$ . Furthermore, this is equivalent to ordering by  $(T_k[SA_k[i]], T_k[SA_k[i] + 1 :]) = (T_k[SA_k[i]], T_k[SA_k[even\text{-}succ_k(i) :]])$ . Finally, this is equivalent to ordering by  $(T_k[SA_k[i]], even\text{-}succ_k(i))$ .

So to store  $even\text{-}succ_k(i)$ , we store items of the form  $(T_k[SA_k[i]], even\text{-}succ_k(i))$ . Each such item requires  $(2^k + \lg n_k)$  bits, because the characters in  $T_k$  are of length  $2^k$  and  $even\text{-}succ_k(i)$  takes  $\lg n_k$  bits. This means that the first  $\lg n_k$  bits won't change very often, so we can store the leading  $\lg n_k$  bits of each value  $v_i$  using unary differential encoding:

$$0^{\text{lead}(v_1)} 1 0^{\text{lead}(v_2) - \text{lead}(v_1)} 1 0^{\text{lead}(v_3) - \text{lead}(v_2)} 1 \dots$$

Where  $\text{lead}(v_i)$  is the value of the leading  $\lg n_k$  bits of  $v_i$ . There will then be  $n_k/2$  ones (one 1 per value) and at most  $2^{\lg n_k} = n_k$  zeros (the maximal value we can store with  $\lg n_k$  bits is  $n_k$  which is the maximal number of incrementations), and hence at most  $(3/2)n_k$  bits total used for this encoding. Again by the geometric nature of successive values of  $n_k$ , this will require  $O(n)$  bits total, so the overall data structure is still compressed. We can store the remaining  $2^k$  bits explicitly. This will take

$$2^k \cdot \frac{n_k}{2} = 2^k \frac{n}{2^{k+1}} = \frac{n}{2} \text{ bits.}$$

Note that if we maintain rank and select data structures, we can efficiently compute

$$\text{lead}(v_i) = \text{rank}_0(\text{select}_1(i)).$$

Therefore, the requirement for *even - succ<sub>k</sub>* is

$$\frac{1}{2}n + \frac{3}{2}n_k + O\left(\frac{n_k}{\log \log n_k}\right).$$

The total requirement for the entire structure is summing this expression plus  $n_k$  (for *is - even - suffix<sub>k</sub>*) over all  $k$ s:

$$\sum_{k=0}^{\log \log n} \left( \frac{1}{2}n + \frac{3}{2}n_k + O\left(\frac{n_k}{\log \log n_k}\right) + n_k \right) = \frac{1}{2}n \log \log n + 5n + O\left(\frac{n_k}{\log \log n_k}\right).$$

Unfortunately, the factor  $n \log \log n$  makes this not compact, so we need to improve further.

## 4 Compact suffix arrays

To reduce the space requirements of the data structure, we want to store fewer levels of recursion. We choose to store  $(1 + 1/\varepsilon)$  levels of recursion, one for the following values of  $k$ :

$$0, \varepsilon\ell, 2\varepsilon\ell, \dots, \ell = \lg \lg n.$$

In other words, instead of pairing two letters together with each recursive step, we are now clustering  $2^{\varepsilon\ell}$  letters in a single step. We now need to be able to jump  $\varepsilon\ell$  levels at once.

## 4.1 Level jumping

In order to find a formula for  $SA_{k\varepsilon\ell}$  in terms of  $SA_{(k+1)\varepsilon\ell}$ , we proceed similarly as when our construction for compressed suffix arrays, but we redefine the word “even” as follows. A suffix  $SA_{k\varepsilon\ell}[i]$  is now “even” if its index in  $T_{k\varepsilon\ell}$  is divisible by  $2^{\varepsilon\ell}$ . This changes the definition of  $even\text{-}rank_{k\varepsilon\ell}(i)$  in the obvious way. However, we will not change the definition of  $even\text{-}succ_{k\varepsilon\ell}(i)$ : it should still return the value  $j$  such that  $SA_{k\varepsilon\ell}[i] = SA_{k\varepsilon\ell}[j] - 1$ . It should do this for all “even” values of  $SA_{k\varepsilon\ell}[i]$ .

With these modified definitions, we can compute  $SA_{k\varepsilon\ell}[i]$  as follows:

- Calculate the even successor repeatedly until index  $j$  is at the next level down — in other words, so that  $SA_{k\varepsilon\ell}[j]$  is divisible by  $2^{\varepsilon\ell}$ .
- Recursively compute  $SA_{(k+1)\varepsilon\ell}[even\text{-}rank_{k\varepsilon\ell}(j)]$ .
- Multiply by  $2^{\varepsilon\ell}$  (the number of letters we clustered together) and then subtract the number of calls to successor in the first step.

This process works for much the same reason that it does in the compressed suffix array. We first calculate the  $j$  such that  $SA_{k\varepsilon\ell}[j]$  is divisible by  $2^{\varepsilon\ell}$  and  $SA_{k\varepsilon\ell}[j] - m = SA_{k\varepsilon\ell}[i]$ , where  $0 \leq m < 2^{\varepsilon\ell}$ . The recursive computation gives us the index in  $T_{(k+1)\varepsilon\ell}$  of the suffix corresponding to  $SA_{k\varepsilon\ell}[j]$ . We can compute the true value of  $SA_{k\varepsilon\ell}[j]$  if we multiply the result of the recursive computation by  $2^{\varepsilon\ell}$ . We then subtract the value  $m$  to get  $SA_{k\varepsilon\ell}[i]$ .

## 4.2 Analysis

We may have to look up the even successor of an index  $2^{\varepsilon\ell}$  times before getting the value we can recur on. Therefore, the total search time is  $O(2^{\varepsilon\ell} \lg \lg n) = O(\lg^\varepsilon n \lg \lg n) = O(\lg^{\varepsilon'} n)$  for any  $\varepsilon' > \varepsilon$ .

We use the same unary differential encoding for successor as in the compressed construction, for a total of  $2n_{k\varepsilon\ell} + n + o(n)$  bits per level in total. We also must store the  $is\text{-}even\text{-}suffix_{k\varepsilon\ell}(\cdot)$  vectors and the rank data structure, for a total of  $n_{k\varepsilon\ell} + o(n)$  per level. There are  $1 + 1/\varepsilon$  levels in total. Hence, the total space is something like  $(6 + 1/\varepsilon)n + o(n)$  bits, which is compact.

There are some optimizations we can perform to improve the space. We don’t have to store the data for  $even\text{-}succ_0(\cdot)$ , because it’s the top level, which means that the total space required storing even successor information is:

$$O\left(\frac{n}{2^{\varepsilon\ell}}\right) = O\left(\frac{n}{\lg^\varepsilon n}\right) = o(n).$$

If we store the  $is\text{-}even\text{-}suffix_{k\varepsilon\ell}(\cdot)$  vectors as succinct dictionaries (because they are, after all, fairly sparse), then the space required is:

$$\lg\left(\frac{n_{k\varepsilon\ell}}{n_{(k+1)\varepsilon\ell}}\right) \approx n_{(k+1)\varepsilon\ell} \lg \frac{n_{k\varepsilon\ell}}{n_{(k+1)\varepsilon\ell}} = n_{(k+1)\varepsilon\ell} \lg 2^{\varepsilon\ell} = \frac{n_{k\varepsilon\ell} \cdot \varepsilon\ell}{2^{\varepsilon\ell}} = \frac{n_{k\varepsilon\ell} \cdot \varepsilon \lg \lg n}{\lg^\varepsilon n} = o(n_{k\varepsilon\ell})$$

Hence, the total space is  $o(n)$ . This gives us a total space of  $(1 + 1/\varepsilon)n + o(n)$  bits.

**OPEN:** Is it possible to achieve  $o(\lg^\varepsilon n)$  in  $O(n)$  space?

## 5 Suffix trees [4]

### 5.1 Construction

In the previous lecture, we saw how to store a binary trie with  $2n + 1$  nodes on  $4n + o(n)$  bits using balanced parens. We can use this to store the structure of the compressed suffix tree. Unfortunately, we don't have enough space to store the edge lengths or the letter depth, which would allow us to traverse the tree with no further effort.

To search for a pattern  $P$  in the tree, we must calculate the letter depth as we go along. Say that we know the letter depth of the current node  $x$ . To descend to its child  $y$ , we need to compute the difference in letter depths, or the length in letters of the edge between them.

The letter depth of  $y$  is equivalent to the length of the substring shared by the leftmost descendant of  $y$  and the rightmost descendant of  $y$ . Let  $\ell$  be the leftmost descendant, and let  $r$  be the rightmost descendant. If we know the index in the suffix array of both  $\ell$  and  $r$ , then we can use the suffix array to find their indices in the text. Because  $\ell$  and  $r$  are both descendants of  $x$ , we know that they both match for  $letter-depth(x)$  characters. So we can skip the first  $letter-depth(x)$  characters of both, and start comparing the characters of  $\ell$ ,  $r$ , and  $P$ . If  $P$  differs from  $\ell$  and  $r$  before they differ from each other, we know that there are no suffixes matching  $P$  in the tree, and we can stop the whole process. Otherwise,  $\ell$  and  $r$  will differ from each other at some point, which will give us the letter depth of  $y$ . Note that the total number of letter comparisons we perform is  $O(|P|)$ , for the entire process of searching the tree.

### 5.2 Necessary binary trie operations

To find  $\ell$ ,  $r$ , and their indices into the suffix array, note that in the balanced parentheses representation of the trie, each leaf is the string “ $()$ ”.

**leaf-rank**(here) The number of leaves to the left of the node which is at the given position in the string of balanced parentheses. Can be computed by getting  $rank_{()}(n)$

**leaf-select**( $i$ ) The position in the balanced parentheses string of the  $i^{\text{th}}$  leaf. Can be computed by calculating  $select_{()}(i)$ .

**leaf-count**(here) The number of leaves in the subtree of the node at the given position in the string of balanced parens. Can be computed using the formula:

$$rank_{()}(\text{matching } ) \text{ of parent} - rank_{()}(\text{here}).$$

**leftmost-leaf**(here) The position in the string of the leftmost leaf of the node at the given position. Given by the formula:

$$leaf-select(leaf-rank(\text{here}) + 1).$$

**rightmost-leaf**(here) The position in the string of the rightmost leaf of the node at the given position. Given by the formula:

$$\text{leaf-select}(\text{leaf-rank}(\text{matching}) \text{ of parent} - 1).$$

Hence, we can use a rank and select data structure on the string of balanced parentheses to find the first and last leaves in the part of the string representing the subtree rooted at  $y$ . We can then calculate the rank of those leaves to determine the index into the suffix array. Hence, we can perform the search described in the previous paragraph at a cost of  $O(1)$  operations per node.

### 5.3 Analysis

The total time required to search for the pattern  $P$  is

$$O(|P| + |\text{output}|) \cdot O(\text{cost of suffix array lookup}).$$

Grossi and Vitter [3] improved upon this to achieve a search time of

$$O\left(\frac{P}{\log_{\Sigma} T} + |\text{output}| \cdot \log_{\Sigma}^{\epsilon} T\right)$$

### 5.4 Succinct Suffix Trees

It is also possible to improve this, creating a succinct suffix tree given a suffix array. In the above algorithm, storing the suffix tree takes too much space to achieve succinctness. Instead, we store the above compact suffix tree on every  $b^{\text{th}}$  entry in the suffix array, which gives us an extra storage cost of  $O(n/b)$ .

First, modify the tree to return something reasonable if  $P$  doesn't match any of the items in the tree, such as returning the predecessor of  $P$  in the set of leaves. If we consider the suffixes to be divided into blocks of size  $b$ , then when we query on  $P$ , the suffix tree will give us an interval of block dividers such that any suffix matching  $P$  must lie in a block touching one of those dividers. This gives us a range of blocks in which to look for the true answer.

The rest of the algorithm was not covered in lecture, but is explained in [4] and in the handwritten lecture notes. The main idea is to use a look-up table which stores the following information: given a sorted array  $A$  of  $b$  length (at most)  $b$  bit-strings and a pattern string  $P$  of length at most  $b^2$ , our table stores the first and last positions of  $P$  in  $A$ . Note that this takes space  $O(2^{b^2+b} \lg b) = O(\sqrt{n})$  bits of space if  $b \leq \frac{1}{2}\sqrt{\lg n}$ . We can then use this to efficiently find the first and last match in a block.

## References

- [1] P. Ferragina and G. Manzini, *Indexing Compressed Text*, Journal of the ACM, Vol. 52 (2005), 552-581.

- [2] R. Grossi, A. Gupta, J. S. Vitter, *High-order entropy-compressed text indexes*, SODA 2003: 841-850.
- [3] R. Grossi and J. S. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, Thirty-Second Annual ACM Symposium on Theory of Computing, pp. 397-406 (2000).
- [4] J. I. Munro, V. Raman, and S. S. Rao, *Space Efficient Suffix Trees*, Journal of Algorithms, 39(2):205-222.
- [5] K. Sadakane, *New text indexing functionalities of the compressed suffix arrays*, Journal of Algorithms, 48(2): 294-313 (2003).
- [6] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, *Compressed Representations of Sequences and Full-Text Indexes*, ACM Transactions on Algorithms, 3(2) article 20 (2007).
- [7] W-K. Hon, T-W. Lam, K. Sadakane, W-K. Sung, and S-M. Yiu *A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays*, Algorithmica, 48(1) : 23-36 (2007).
- [8] W-K. Hon, K. Sadakane, and W-K. Sung *Breaking a time-and-space barrier in constructing full-text indices*, 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 251-260 (2003).
- [9] W-K. Hon and K. Sadakane *Space-Economical Algorithms for Finding Maximal Unique Matches*, Combinatorial Pattern Matching, Lecture Notes in Computer Science, 2373 : 17-29 (2002).
- [10] N. J. Larsson, and K. Sadakane *Faster suffix sorting*, Theoretical Computer Science, 387(3) : 258-272 (2007).
- [11] K. Sadakane *Succinct data structures for flexible text retrieval systems*, Journal of Discrete Algorithms, 5(1) : 12-22 (2007).
- [12] H-L. Chan, W-K. Hon, T-W. Lam, and K. Sadakane *Compressed indexes for dynamic text collections*, ACM Transactions on Algorithms, 3(2) article 21 (2007).