A Survey On Text Indexing Techniques And Their Implementation

Yidong Wei, Jianhuan Feng

1.Introduction

Most applications that aim at extracting information from text based on String matching. String matching is the process of finding a pattern's occurrences in a longer text. String matching can be performed in two forms. On one hand, sequential string matching does not require pre-processing of the text, but instead traverses the text sequentially to find the occurrence of the pattern, such techniques including KMP[14], Aho Corasick[1], BMH[21]. However, sequential scanning requires traversing the text at least once, which is not scalable when the amount of text and length of text is large. On the other hand, Indexed string matching preprocesses text then builds data structures (index) that allows finding all occurrences of any patterns without traversing the full text, such data structure including Suffix Tree[16,26], Suffix Array[18] etc. Indexing is a feasible solution when (a) the amount and length of the text is large and it is too costly to sequentially scan; (b) the text is stored static without frequent change thus the saving on search outweighs the cost of building and maintaining the index; (c) there is sufficient storage space to maintain index and provide efficient access to it. With the first two reasons referring to the advantage of indexing over sequential scanning, the last one is also the requirement for building a good indexing algorithm, and the goal of further indexing algorithms.

The process of indexing string, also called text indexing, is the act of processing a text in order to allow fast search on its content. After the indexing process, we are consider three operations for the indexing data structure: (a) count(search): given a pattern P, the data structure can tell the number of occurrences of P in text, (b) locate: after counting the Pattern P, the data structure can return its location in the text. (c) display/substring: given a index interval, the data structure could display the content in that interval.

In this project, we are running a survey on text indexing techniques that are based on suffix array and provide our ideas of implementing those techniques in practice. We will also test them in different scenarios, collect the test results, and conduct an objective evaluation of the algorithm through the analysis of the results. In addition, we will also provide the implementations of the tool data structures used in the project, such as bitmap, wavelet tree.

In section 2, we will introduce some important compact data representations that are essential to text indexing, including bitmap, wavelet tree. In section 3, we will introduce a linear time suffix construction algorithm: SA-IS[8]. In section 4, we will survey on text indexing techniques based on binary search, to be specific, section 4.1 will show operations in Suffix Array; section 4.2 will introduce,GV-CSA, a compressed version of Suffix Array from [10]; section 4.3 will introduce SAD-CSA, a fully-indexed Compress Suffix Array based on GV-CSA, proposed by [24]. Section 5 will introduce backward search technique, with section 5.1 will introduce backward search in the suffix array; section 5.2 will introduce WT-FMI[9] and section 5.3 will be RL-FMI[17]. Section 6 will list the results of our experiments. Section 8 will list some challenges we met in this project as well as our conclusion for the project. Section 9 lists our source of reference.

2.Basic Compact Data Representations

2.1 Compact Representation of Binary Sequence

We will learn later that nearly all approaches to indexing text data take advantage of the compressed representation of binary sequence. A space efficient and operation efficient representation of binary sequence regarded as the core of majority text indexing techniques.

That is, we require a representation of binary sequence B[1,n] that support the following operations:

1. Access(i): return B[i]
2. $Rank_b(i)$: return number of b appears in B[1..i]
3. $Select_b(i)$: return the index of i^{th} occurrence of b in B[1..n]

Other operation could be derived from these three operations:

4. $Prev_b(i)$/$Next_b(i)$: return the index of previous/next occurrence of b from i. This can be achieved by $Select_b(Rank_b(i))$ and $Select_b(Rank_b(i)+1)$

Jacobson[27] initiated the study of succinct representation of binary sequence by proposing a succinct representation of binary sequence that support constant time rank operation. The succinct binary sequence proposed by Jacobson uses three levels of lookups table and precomputes the rank of the end of each block, which totally take n + o(n) bits to represent n bits. Munro[13] and Clark[2] using the similar idea as Jacobson, obtain a constant time select operation using the same o(n) bits, but it requires twice such structure as select1 and select0 cannot transform to each other.

In our project's optimal implementation, we use the modified version of Jacobson's representative, for supporting Rank operation in our binary representation. However, for select operation, we did not adopt the representation from Munro and Clark due to (a)too much space overhead for supporting these two operations at the same time and (b)the text indexing technique we study mostly based on rank operation and only two of them will require the use of select operation. Thus, for the select operation, we use the binary search technique based on rank operation, more detail will be discussed in the following paragraphs.

Even though the representations of binary sequence from [2,13,27] are succinct, the amount of space overhead is critical in text indexing, as one of our ultimate goals is efficient space. As a result, a deep analysis of these representations is necessary. We refer to [22] for such analysis. In [22], the authors showed, in a fully implemented version of Jacobson's binary rank, the o(n) term will take 66.85% space extra when $n = 2^{30}$ bits. In specific, the lookup table in the first level will take about 6.67%, second level will take 60% and last level will take 0.18%. Similarly, for a fully implemented version of Munro and Clark's binary select, the space overhead will be 60% when $n = 2^{30}$bits, and sum up to 120% when we want to support both select1 and select0. The space overhead will increase as n decrease, and decrease as n increase, but this is still not negligible.

As a result, [22] proposed a few modified implementations of Jacobson's binary rank, which were based on the popcount table. Popcount refers to counting the number of bits that are set in a bit array. And the popcount table refers to the precomputed table that stores the number of bits that are set indexed by the bit array. For example, a popcount table for a bit array of size 8(figure 1) could be represented by a 256 byte(a byte array with size 256) or a 128 byte(a bit array with length 256*4 since at most 8 bits set, and $8 = 1000_2$, 4 bits in space). Since a bit array of size 8 could be represented by a byte, if we use that byte as index to access the popcount table, then the number of bits set is evaluated in O(1) time. The modified version proposed in [22] including different combination of block size(the 2nd level block in Jacobson's) and popcount table size to replace the direct lookuping in bottom level of table in Jacobson's by a sequential scanning, as well as only keep the superblock table and using popcount table and blocks to sequentially scanning the rest. According to the experiment in [22], the classical implementation of Jacobson's binary rank consumes the most space and performs the slowest among all testing versions when $0<\log(n)<30$, and such trend still pertains when $\log(n) > 30$(figure 2, figure 3). And they conclude that using 2 level classical table and block size of 32 bits and 256 bytes popcount table(8 bit length array), with at most 4 table access to replace the classical version's direct lookup in last level, produces the overall best result among all the testing versions, and with only 37.5% space overhead when $\log(n) < 30$. They also mention that their 1 level version could achieve the best result when $\log(n) > 30$, with only 10% space overhead. In our project, we adopt the implementation of 2 level, 32bit and 256 byte one.

To be detailed, in our binary sequence, we use block size b= 32bit, superblock size s = log(n)b bits and 256 byte popcount table. For a rank query, we calculate rank1, rank2 as classical version does, and we split the final block(where the query index i locate at) in to 4 byte and have at most 4 table lookup(for the byte where index i locates we could use bit shift operation to transform it into the correct index for accessing the popcount table). Thus the Rank operation is still run in constant time.

For select operation in binary sequence, [22] also run experiments in comparing different combinations of binary search/sequential search with Munro and Clark's classical method. The result shows that for log(n) up to 22 to 26, when the density of 1 in the sequence is high, the version of binary searching for locating superblock then sequential searching using popcount table in subblock is superior than the constant version. However, when n gets large, the log(n) nature of binary search shows up. Since the later text indexing techniques we study are mostly based on rank operation, and the space overhead of using the constant select, our implementation also adopts what the authors appreciated. To be detailed, for select i^{th} b, b = 0 or 1, in our binary sequence, we first binary the superblock B such that $T[B] >= i$ and $T[B-1] < i$. Then we use the popcount table to sequentially search blocks with size 32bits(4 access per block) to find the $(i-T[B-1])^{th}$ b inside the superblock, where T refers to the first level lookup table. Binary search for locating superblock takes O(log(n/logn)) times and sequential search inside superblock takes O(logn) time so totally O(logn) time.

index	binary represenation	popc
1	00000001	1
2	00000010	1
3	00000011	2
4	00000100	1
5	00000101	2
6	00000110	2
7	00000111	3
8	00001000	1
9	00001001	2
10	00001010	2
11	00001011	3
12	00001100	2
13	00001101	3
14	00001110	3
15	00001111	4
16	00010000	1
17	00010001	2
18	00010010	2
19	00010011	3
20	00010100	2

Figure 1, a popcount table for a bit array of size 8;only index 1 to 20 shown

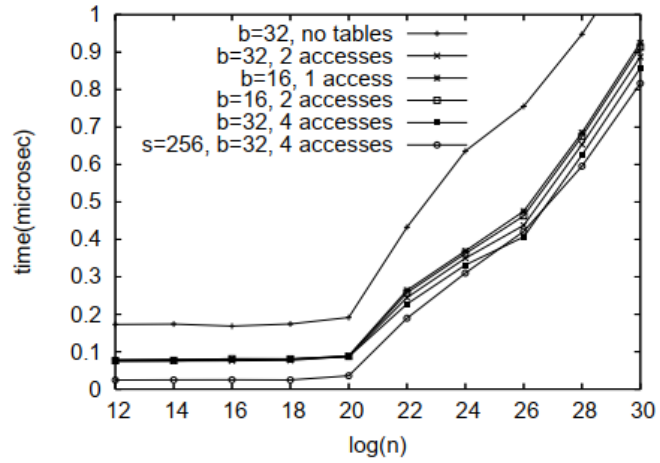

Figure 2. Experiment result from [22]

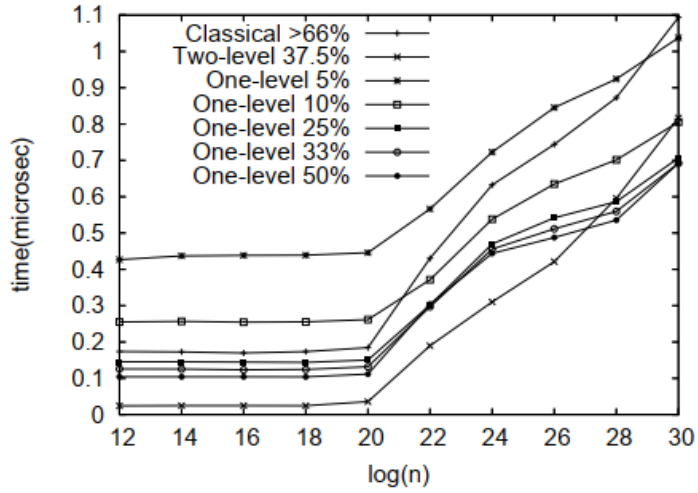

Figure 3, Experiment result 2 from [22]

2.2 Wavelet tree

Given a string S[1,n] for alphabet of size $\sigma \geq 2$, we want to find a representation of it that support the same operations as the binary string in the previous subsection:

1. Access(i): return S[i]
2. $Rank_c(i)$: return number of b appears in S[1..i]
3. $Select_c(i)$: return the index of i^{th} occurrence of c in S[1..n]

Other operation could be derived from these three operations:

4. $Prev_c(i)/Next_c(i)$: return the index of previous/next occurrence of b from i. This can be achieved by $Select_c(Rank_c(i))$ and $Select_c(Rank_c(i)+1)$

One could simply obtain such representation using the result from section 3.1 by building σ binary sequences, per sequence corresponding to per c, with $B_c[i] = 1$ if S[i] = c, which support all operations in the same running time as the representation of binary sequence chosen. However, such method would require σn + o(n) bits, which is impractical in text indexing when σis large.

On the other hand, a technique called wavelet tree was proposed by [9]. A wavelet tree is a balance binary tree with height logσ, build on alphabet symbols. At each node, it separates the characters it represents into two groups(0s and 1s) and represents them using a binary sequence, and its left child represents group 0 and right child represents group 1 such procedure recurse until a node represents a unique char. All queries can be answer in O(logσ) time given that we have a representation of a binary sequence that supports operation in constant time. In this project, we only require a wavelet tree that support $Rank_c(i)$ and Access(i) in constant time, so our choice of binary sequence representation could be used to build a wavelet tree that runs in O(logσ) time. For $Rank_c(i)$, we firstly check the group b of c belong to in the current node, and use $Rank_b(i)$ as input parameter to query on its child node(left if b=0, right if b = 1) until we meet a node that without the child we want. For Access(i), we firstly use Access(i) on the binary sequence to determine the group b, then using $Rank_b(i)$ as parameter to Access query on the correspond child until no more child to goto, and the result will be the unique character that group represent in the last node we reach. In implementation, we map each unique character to an integer number start from 0 to $\sigma - 1$, and use the mod 2 operation to determine the group. Since each level of the wavelet tree represent at most n character, thus the total space occupied the wavelet tree is O(logσn), and in our implementation logσn + o(logσn) bits.

2.3 Elias δ encoding

Before we talk about Elias δ coding, we need to introduce Elias γ encoding. Elias γ coding is a universal code invented by Peter Elias[3] for encoding positive integers. It is often used for positive integers where the upper bound is not known in advance. The code has two equivalent encoding methods. For a given positive integer x, for the first method, $N = \lfloor \log_2 x \rfloor$ is computed such that $2^N \leq x < 2^{N+1}$ and x is converted to a binary number. Then combining N zeros and the binary number is the γ encoding of x. While in the second method, x is decomposed into the form $x = 2^N + M$, where N is the highest power of 2. Then N is encoded into a unary number (N zeros followed by a 1) and M is encoded into a binary number of N bits. Finally combining the two yields the same γ encoding.

Elias δ coding is an improvement on the algorithm of γ coding, aiming at further compression of N, whereupon larger values can be handled. Likewise, δ coding possesses two equivalent coding methods. For simplicity, we present only one of the two that can take full advantage of γ coding. First, decompose x into the form $x = 2^N + M$. Then use γ coding to encode N + 1 and convert M into a binary number of N bits. Combining the two results yields the δ encoding of x. It can be seen that δ coding is to perform γ coding again for the first factor N of γ coding. Hence, the total bit for a δ encoding is $\lfloor \log_2 x \rfloor + 2\lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor + 1$.
As for decoding δ encoding, first count the number of zeros before reaching the first 1 of the encoding. The number is called L. Read the L bits after the first 1 and convert these bits into a decimal number, which is N+1. Continue reading the last N bits and convert them to a new decimal number M. Reconstruct x by computing $2^N + M$.

3. Suffix Array Construction

In this section, we will introduce the suffix array and suffix tree construction. Both processes are under the assumption that we will insert a sentinel '$' at the end of the text, and the '$' is defined to be lexicographically smaller than any alphabet in the text. In implementation, one could use the char with byte value 3 to represent the sentinel, as in ASCII encoding, it represents 'End of Text'.

Definition 1: The suffix array of a text T[1,n] is an array SA[1,n] containing a permutation of [1.n] such that T[SA[i]] is the i^{th} lexicographically smallest suffix in T. In order word, T[SA[i]...n] <T[SA[i+1]...n]for all 1 < i < n

According to **definition 1**, a suffix array is simply the index of suffix In text T that is sorted in lexicographical order. A suffix array can be simply obtained by in-order traversing the suffix tree(given that children of the suffix tree node is ordered lexicographically) in O(n) times. However, it is more practical to build the suffix array directly since the working space of building suffix tree and transforming to suffix array is too large. Since the proposal of suffix array[18], several suffix array construction methods were proposed, including the native O(nlogn) method[18], to the later O(n) method, called SA-IS(Suffix Array Induced Sort)[8] and its variant or improved version. Most later algorithms[15] proposed were based on the SA-IS method and aimed to reduce its working memory in the construction process. For simplicity, our project only adopts the classical SA-IS algorithm. In this section, we will summarize the SA-IS algorithm and provide a run through example.

The construction of the suffix array by the SA-IS algorithm[8] is based on the type of suffix and type of substring. Following will be the definition of such terms as well as some facts related to them[19].

Definition 2: A suffix T[i...n] is called S-type if T[i...n] < T[i+1...n] or i = n, otherwise it is L-type.

Fact from def 2: A suffix T[i...n] is called S-type if (1)T[i] < T[i+1];(2) i = n;(3)T[i] = T[i+1] and T[i+1] is S-type; otherwise, it is L- type.

Proof for case (3): assume T[i] = T[i+1], assume T[i+1] is S-type, then there must be an index j such that T[j] < T[j+1], i < j, otherwise T[i+1] is L-type. Then we align T[i...n] with T[i+1...n] with T[i] align with T[i+1], T[i+1] with T[i+2], ...T[j] with T[j+1] and T[j]<T[j+1], thus it returns that T[i...n] < T[i+1...n].

By using the fact from def 8, one scans the text from right to left to determine the type of each suffix in O(n) time without comparing the actual suffix string.

```
(1)     T[n] is S-type // the suffix starting at the sentinel is considered an S-type suffix
(2)     For i = n - 2, i > 0,  i--
(3)         if T[i] > T[i+1]
(4)             T[i] is L-type
(5)         else if T[i] == T[i+1] and T[i+1] is L-type
(6)             T[i] is L-type
(7)         else
(8)             T[i] is S-type
```

Algotirhm 1. Determine the type of a suffix

Definition 3: A character T[i] is S-type if suffix T[i...n] is S-type, and it is LMS-character if T[i] is S-type and T[i-1] is L-type.
Definition 4: A suffix T[i..n] is a LMS-suffix if T[i] is a LMS-charactr
Definition 5: A LMS-substring is a slice of T[i...j] such that T[i] and T[j] are both LMS; or T[n], the sentinel itself.

Thus, we could identify whether a character is LMS once we know the type of the suffix, and determine the LMS-substring and LMS-suffix at the same time.

The sorting process is conducted by the bucket sort technique, suppose there are σunique characters in the text(including the sentinel '$'), thus each bucket corresponds to one unique character in the text, allocating in lexicographical order. The length of a bucket is equal to the number of characters in the text it represents. For example, if we have a text T = 'abcabxabcd$', then bucket for '$' is length one, bucket for 'a' and 'b' is length 3, since the number of suffixs start with 'a' or 'b' is 3, etc. And the bucket could be sorted in lexicographical order, thus bucket order is '$abcdx$.

To proceed the bucket sort, a few facts related to definition 8, 9, 10, 11 could help to give us a clear idea of how the algorithm works.

Fact (a)[19]: In any bucket, the L-suffix always appear before the S-suffix
Fact (b) [8]: If we know the lexicographical order of all LMS-substrings, then we can use their ranks to construct the reduced problem T1. Sorting the suffixes of T1 is equivalent to sorting the LMSsuffixes of T
Fact (b)[8]: Suppose all LMS-suffixes of T are already sorted and stored in the tail of their buckets. Then using induced sorting, all L-suffixes can be sorted correctly.
Fact(c)[19]: Suppose all S-suffixes (or L-suffixes resp.) of T are already sorted. Then using induced sorting, all L-suffixes (or S-suffixes resp.) can be sorted correctly.

From these facts, it it clear that the order of SA-IS algorithm:(1)label the type of each suffix, (2)label the LMS-character and LMS-substring, (3)sorting the LMS-substring, (4)sorting the LMS-suffixes, (5)sorting the L-suffixes, (6) sort the S-suffixes

The authors also defined LMS-prefix T[i….j] of a suffix T[i...n] when j > i and T[j] is the first LMS-character after T[i]. A LMS-prefix T[i…j] is L-type if T[i] is L-type, and vice versa for S-type.

To sort the LMS-substring, we initialize the suffix array by setting all its items to -1, and maintain an end and head pointer for each bucket. In step 1, we scan T[1,n] from left to right, and put the index of the LMS-character into the position of the end of the bucket according to the character value. In step 2, we scan from left to right, when we are meeting i s.t. SA[i] != -1, we check the type of T[SA[i]-1], if it is L-type, we put SA[i]-1 to the head of the bucket according to T[SA[i]-1], and move the head pointer right by 1 position. Step 2 will sort all L-type prefixes. In step 3, we scan SA from right to left, if SA[i] != -1 and T[SA[i]-1] is S-type, we put SA[i]-1 to the end of the bucket according to T[SA[i]-1], and move the head pointer left by 1 position. After step 3, all the S-type and L-type LMS prefixes are sorted, thus the LMS-prefixes are sorted accordingly. Sorted LMS-prefixed implies sorted LMS-substring.

For sorting LMS-suffix by the sorted LMS-substring, we firstly scan the SA from left to right to name each LMS-substring, where the name of LMS-substring is the order of the substring, two equal LMS-substrings will have the same name. One can observe that if LMS-substring S1 appears before LMS-substring S2, then S1 <= S2(since they are sorted). After naming, we check whether the names of all LMS-substrings are unique, if they are unique, it means that the LMS-Suffixes are already sorted(recall if T[i1...j1] < T[i2...j2], T[i1...n]<T[i2...n]). Otherwise, it means there are at least two LMS-suffix's relative order if not determined yet. Sorting the LMS-suffixes from the sorted LMS-substring can be transformed to find the suffix array of a smaller string formed by the concatenation of the LMS-substrings name in the order they appear in the text. Basically, this is a recursive call of the SA-IS algorithm and the recursion will end once the name of LMS-substring is unique in the inner calling stack.

After the LMS-suffix is sorted, we can simply do the LMS-substring sorting procedure again but the LMS-characters will be inserted to the end of the bucket based on the descending order of the LMS-suffix they lead.

For illustration, here we will provide a run through example for building the suffix array for the text "bcbcabxcaabcd$". The procedure of (1)label the type of each suffix, (2)label the LMS-character and LMS-substring, (3)sorting the LMS-substring, will be shown in (figure 4) with each row corresponding to one step. For convenient, "@" denote the current character, "^"denote the head of each bucket, "&" denote the end of each bucket, "L" denote L-type, "S" denote S-type, "*" denote a LMS- characters

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
T	b	c	b	c	a	b	x	c	a	a	b	c	d	$
Type	S	L	S	L	S	S	L	L	S	S	S	S	L	S
LMS			*		*				*					*
Step 1:														
bucket	$		a			b				c			d	x
SA	13	-1	8	4	-1	-1	-1	2	-1	-1	-1	-1	-1	-1
Step 2:														
SA:	13	-1	8	4	-1	-1	-1	2	-1	-1	-1	-1	12	-1
T[12] is L type	@^	^			^				^				^	^
	13	-1	8	4	-1	-1	-1	2	7	-1	-1	-1	12	-1
T[7] is L type	^	^	@		^					^			^	^
	13	-1	8	4	-1	-1	-1	2	7	3	-1	-1	12	-1
T[3] is L type	^	^		@	^						^		^	^
	13	-1	8	4	-1	-1	-1	2	7	3	1	-1	12	-1
T[1] is L-type	^	^			^			@				^	^	^
	13	-1	8	4	-1	-1	-1	2	7	3	1	-1	12	6
T[6] is L-type	^	^			^				@			^	^	^
	13	-1	8	4	-1	-1	-1	2	7	3	1	-1	12	6
T[2] is S-type, skip	^	^			^					@		^	^	^
	13	-1	8	4	-1	-1	-1	2	7	3	1	-1	12	6
T[0] is S-type, skip	^	^			^						@	^	^	^
	13	-1	8	4	-1	-1	-1	2	7	3	1	-1	12	6
T[11] is S-type, skip	^	^			^							^	@^	^
	13	-1	8	4	-1	-1	-1	2	7	3	1	-1	12	6
T[5] is S-type, skip	^	^			^							^	^	@^
Step 3:														
SA	13	-1	8	4	-1	-1	-1	5	7	3	1	-1	12	6
T[5] is S-type	&			&			&					&	&	@&
	13	-1	8	4	-1	-1	-1	5	7	3	1	11	12	6
T[11] is S-type	&			&			&				&		@&	&
	13	-1	8	4	-1	-1	10	5	7	3	1	11	12	6
T[10] is S-type	&			&		&					&	@	&	&
	13	-1	8	4	-1	0	10	5	7	3	1	11	12	6
T[0] is S-type	&			&	&						@&		&	&
	13	-1	8	4	2	0	10	5	7	3	1	11	12	6
T[2] is S-type	&			&						@	&		&	&
	13	-1	8	4	2	0	10	5	7	3	1	11	12	6
T[6] is L-type, skip	&			&					@		&		&	&
	13	-1	8	4	2	0	10	5	7	3	1	11	12	6
T[4] is S-type	&		&					@			&		&	&
	13	-1	9	4	2	0	10	5	7	3	1	11	12	6
T[9] is S -type	&	&					@				&		&	&
	13	-1	9	4	2	0	10	5	7	3	1	11	12	6
T[1] is L-type, skip	&	&			@						&		&	&
	13	-1	9	4	2	0	10	5	7	3	1	11	12	6
T[3] is L-type, skip	&	&		@							&		&	&
	13	8	9	4	2	0	10	5	7	3	1	11	12	6
T[8] is S-type	&		@								&		&	&
	13	8	9	4	2	0	10	5	7	3	1	11	12	6
T[7] is L-type	&	@									&		&	&
	13	8	9	4	2	0	10	5	7	3	1	11	12	6
T[12] is L-type	@										&		&	&

Figure 4. A run through example of sorting the LMS-substrings.

Thus, the LMS-substring {“bca”, “abxca”, “aabcd$”, “$”} are with names {3,2,1,0}. In this case, the names of LMS-substrings are unique by coincidence, meaning LMS-suffixes are already sorted. Hence, we insert LMS-suffix into the bucket by their descending order and do the same process again to obtain a suffix array for the text. Calful reader would notice that the final result is the same as what we obtain here, due to the coincidence that we already inserted the LMS-suffix in descending order. If there is a case that the names of the LMS-substrings are not unique, take {2,1,1,0} for example, then we would need to find the suffix array of the text T_1=“2110” by the same procedure.(Here the sentinel “$” is not required to added to the end of T_1 since T_1 will always ended by 0)

Here we show that the SA-IS algorithm is a O(n) time algorithm. It is obvious that the number of LMS characters is no more than n/2, thus the number of LMS-substrings is no more than n/2. Sorting the LMS-substring will require multiple sequential scans of the SA, which still takes O(n) time. The naming process, with the help of the previous sorting process, we always be sure that the LMS-substring appears first in the SA is less or equal to the LMS-substring appears next, we could finish the naming process by one sequential scanning of the template suffix array, with each character in the LMS-substring could be compared at most twice and the total length of LMS-substrings is no more than 1.5n. As a result, the naming process is also running in O(n) times. Since the number of substrings is no more than n/2, the largest recursive problem will be with size no more than n/2. We could easily have a recurrence relation: T(n) <= T(n/2)+O(n), which totally come with O(n) suffix array creation time.

4.Techniques based on Suffix Array

In this section, we will survey on text indexing techniques that were based on the suffix array, and the Count,Locate,Display operations will be discussed.

4.1 Classical Suffix Array

We start by introducing the data structure containing the classical suffix array and original text. For Display the text T[i..j], one can simply array access the character in the string by index in O(j-i) time as nearly all programming languages support such operation. For Locate the pattern P after the search/count operation, given occurrence index i in the suffix array, one could easily know the location in the text by access$SA[i]$, which return in $O(Occ)$ time for occ matches.

For search or count all instances of pattern$P$ with length p in the suffix array, a binary search technique was proposed by [18]. Which firstly binary search for $Lw = min\{k: P \leq p\, T[SA[k]... n]\}$ and $Rw = max\{k: P\, p \geq T[SA[k]... n]$(Algorithm 2), where the $p \geq$and $\leq p$ refer to comparing the p-length prefix, that is, the suffix's first p character with pattern P, Then the number of occurrence of P in text T[1..n] is $Lw - Rw + 1$, and the match instance in SA is [Rw….Lw]. Binary search for Lw and Rw take O(logn) comparisons and comparing with pattern P with length p will take O(p) per comparison, thus search on the suffix array will take O(plogn) time. The authors[18] also mention that, in the process of binary search, if we record the number of matching characters,LCP(longest common prefix), in each comparison, then we could skip some unnecessary character comparison in the next iteration. However, this technique does not reduce the running time asymptotically, which still requires O(plogn). To facilitate the binary search process, we could also store a Rlcp and Llcp array to reduce the total number of single character comparisons to less than the length p of pattern P, which results in a O(p+logn) time search[18].

Search-L_w (A suffix array, P pattern)
(1) *if* $P \leq_P Ap_{os}[0]$ *then*
(2) $L_W \leftarrow 0$
(3) *else if* $P >_P Ap_{os}[N-1]$ *then*
(4) $L_W \leftarrow N$
(5) *else*
(6) { $(L, R) \leftarrow (0, N-1)$

```
(7)     while R - L > 1 do
(8)     {   M ← (L + R)/2
(9)         if P ≤P Apos[M] then
(10)            R ← M
(11)        else
(12)            L ← M
(13)    }
(14)    LW ← R
(15) }
```

Algorithm 2. Calculate L_w by binary search

The space occupied by the classical suffix array will take nlogn bits plus the original text, or O(n) words plus the original text. When it incorporates the addition lcp array represented by a O(n) space Cartesian trees-like structure, the total space is O(n) words with some constant factor. Even though the suffix array structure is simple, the nlogn bits property makes it impractical when the text is large, which requires too much space overhead, let alone after incorporating with the lcp array. Thus, in our project, we only implement the classical suffix array and speedup string comparison without using the lcp array. The space drawback of the classical suffix array was the motivation of further research, which will be surveyed in the following sections.

4.2 GV-CSA(Grossi and Vitter's Compressed Suffix Array)

The compressed suffix array from Grossp and Vitters's[10] is based on the idea of succinctly representing suffix array SA and providing efficient access to SA[i], while the text T itself is stored explicitly.

Definition 6: Given a suffix Array of a text T, SA[1,n], function Ψ: $[1, n] \rightarrow [1, n]$ is defined so that, for all $1 \leq i \leq n$, $SA[\Psi(i)] = SA[i] + 1$,except that $SA[\Psi(1)] = 1$. That is, $\Psi(i)$is the location of the suffix T[SA[i]+1….n] in SA[1,n]

To build an array Phi such that Phi[i] = $\Psi(i)$, we could iterate through SA to build SA^{-1} such that SA^{-1}[j] = i if SA[i] = j, then Phi[i] = SA^{-1}[SA[i]+1]

The GV-CSA decomposed SA recursively based on the ψfunction. In this survey we will focus on describing the process to decompose the SA in the first level following the level can be built recursively in a similar manner.

For the original suffix array SA, we let SA_0=SA, we build a bit vector B_0 such that B_0[i] = 1 if SA_0[i] is even, Then we could build an array $Phi_0[1,\frac{n}{2}]$ contain the value of Ψ(i) when B_0[i] = 0. Now, note that if we want to access Ψ(i) when B_0[i] = 0, we need to access $Phi_0[Rank_0(B_0,i)]$ since Phi_0 only store the Ψ(i) for B_0[i] = 0. Then create a new array SA_1 such that SA_1 is formed by SA_0[i]/2 sequentially, for all i such that B_0[i] = 1.(note, the value in SA_1 range from 1 to $\frac{n}{2}$

In the 2 level decompose structure, if we want to access SA[i], we firstly check B_0[i], there is two case: (1)B_0[i] = 1, then SA[i] = 2*SA_1[$Rank_1(B_0,i)$]; (2)B_0[i] = 0, then we know SA[i] is odd, and SA[i] + 1 must be even,and stored in SA_1, from Phi_0[$Rank_0(B_0,i)$], we know j such that SA[j] = SA[i] + 1, and B_0[j] = 1. Then we use case (1) to get SA[j] and SA[i] = SA[j] - 1.

This idea could apply recursively, using B_1,Phi_1,and SA_2 to represent SA_1 until SA_h is small enough to represent explicitly. It is convenient to use h = loglogn, thus there are $\frac{n}{logn}$elements in SA_h, each require O(logn) bits, which overall take O(n) bits[10]. The bit array B_k,$1 \leq k \leq h$ is halved in each level, thus it will cost at most 2n + o(n) bits if we use the binary sequence representation in section 2.1.

What the GV-CSA problem left is how to represent Phi_k efficiently. The original solution from [10] was based on the property of Ψ**.**

Property of Ψ: Given a suffix array SA of T, for i< j such that T[SA[i]] = T[SA[j]], $\Psi(i) < \Psi(j)$.
Proof: Given T[SA[i]] = T[SA[j]], and i < j, we know that T[SA[i]+1...n] < T[SA[j]+1...n], then for v, w such that SA[v] = SA[i]+1, and SA[w] = SA[j] + 1, we have v < w.

By this property, we could say that for an interval [j...k] in the SA such that T[SA[j]] = T[SA[j+1]] = … = T[SA[k]], Ψ(i) is monotone increasing when $j \leq i \leq k$.

In [10], Ψ_k(i) function is grouped in to $\sigma^{2\wedge k}$ lists, each list represent a sequence of 2^k characters which corresponds to T[$2^k SA_k$[i]....2^k(SA_k[i]+1)-1], that is the leading hyper-characters in each increasing interval at level k. Then the name is assigned to each list from 0 to $\sigma^{2\wedge k}$-1. Value x in Ψ_k(i), will be transform into x', which uses log($\sigma^{2\wedge k}$) + log(n^k) bits, with the binary representation of the list's name, that x belongs to, prepending to x-1. By using this technique, the transform x' in Ψ_k(i) is in one single increasing sequence, and the value of x can be extracted in constant time once we have x'. Afterall, the transformed Ψ_k(i) function with value x'1, x'2, x'3, …, x'n_k can be transformed into Ψ'_k(i) with value x'1, x'2-x'1, x'3-x'2, …, x'n_k-x'(n_k-1) and represented by unary representation U= 0^i1, where 0^i refers to Ψ'_k(i) 0s. Then, the h^{th} Ψ'_k(i) is can be answered by select(U,h) - h, and Ψ_k(i) can be extracted from Ψ'_k in constant time.

A second solution proposed by Sadakane[24] using Elias δ- encoding to represent Phi_k in little space, based on the underlying property of Ψfunction. Since Elias δ- encoding is sensitive to the magnitude of the integer it represent, a solution to reduce the magnitude of data is necessary for a efficient space Elias δ- encoding. Knowing that $\Psi(i)$for a Phi array containing multiple increasing intervals, we could simply transform the Phi array with only the first element in each interval pertaining to the absolute value and the rest element in the interval replaced by its difference with the previous element. For example, given a Phi = {4,6,7,9,10,1,3,4,5,2,4,5,6}, we could transform it into Phi*={4,2,1,2,1,1,2,1,1,2,2,1,1}, and create another bit array D and set D[i] = 1 if i is the first element in an increasing interval. Then we apply the Elias δ-encoding on Phi* to achieve the space efficient goal. There is more details about Elias δ-encoding in practice, including decoding procedure and setting more absolute value in the

interval for speeding up. These detail will be discussed in next section together with SAD-CSA[24], which proposed the idea of using Elias δ- encoding, and building based on GV-CSA.

Here we will discuss the count, locate and display operation of GV-CSA. For locating SA[i], we need to recursively access $SA_1$to SA_h , where h = loglogn, thus locating an occurrence will take O(loglogn) time. For counting, we could simply apply the same binary search technique, except that when comparing T[SA[i]...SA[i]+p] to pattern P, we need to locate SA[i], which is O(loglogn), thus totally come with O(plognloglogn) time for searching. Since the text is stored explicitly, we could display the content in constant time.

The space of GV-CSA is nH0loglog n + O(nloglogσ) + n log σ bits, with the Elias δ- encoding for Ψfunction dominates the space and h = loglogn levels of Elias δ- encoding.

4.3 SAD-CSA(Sadakane's Compressed Suffix Array)

The GV-CSA reduces the space by representing the Suffix-Array succinctly, while the text is still stored explicitly. There are also further researches on representing the text implicitly, that is, finding a data structure that supports count/search, locate and display without the existence of the original text, which is also called self indexed. The SAD-CSA proposed by Sadakane[24] was a further development of GV-CSA, by converting GV-CSA into a self index data structure.

The SAD-CSA uses the decompose suffix array, a full ψfunction(as opposite to only n/2 ψ function in CSA) and a C function to represent the indexed text. The C function is mapping function such that $C(i)$return the characters in the text T[SA[i]]. Thus, one could use a bit sequence D, for which D[i] = 1 if $T[SA[i]] \neq T[SA[i] - 1]$and an array C of size σ, for which C[i] = i^{th} lexicographically smallest character in text T, to represent the C function. Then, $C(i)$ =C[$Rank_1$(D,i)].

As mentioned in section 4.2, SAD-CSA use also use the Elias δencoding technique to store ψ function. In addition to storing the absolute leading value of each increasing interval, the absolute value in each log(n)bits is also stored as representative value, with an bit array set E, E[i] = 1 when ψ(i) is explicitly stored, which helps to reduce the length of the sequence required to be decoded. Then, instead of storing the Elias δencoded bit sequence with log length, the bit sequence was decomposed into length of log(n)bits, and the chunks are stored in an array S.When decode ψ(i), we need to calculate $Rank_1$(E,i), then decode i -$Select_1$(E, $Rank_1$(E,i)) value in the chunk S[$Rank_1$(E,i)], where $Select_1$(E, $Rank_1$(E,i)) refers to the index of the representative value in the chunk. In the process of decoding, when we meet a leading value(which means there is another increasing sequence), we will discard the value we calculate and cumulate our calculation again. The leading value can be easily detected by checking bit array D. To support constant decode, a precomputed table for the total difference of each logn/2 bits in the chunk is stored then each chunk could be processed in constant time, which will require o(n) bits, this is similar to other four-Russians technique.

To support display operation in SAD-CSA, Sadakane[24] also proposed the function SA^{-1}(i), which is the inverse of SA[i], that is SA^{-1}(i) = j such that SA[j] = i. The inverse function is based on the relationship between SA_k and SA_{k+1}, assume we know that SA_{K+1}[i] = q or SA_{k+1}^{-1}[q]= i,

then there must be a j in B_k such that $B_k[j] = 1$, and $Rank_1(B_k,j) = i$. Thus, we could have $j = Select_1(B_k,i)$, for which $SA_k[j] = 2*SA_{k+1}[i]$ or $SA_K^{-1}[j] = 2q$. Extending this idea, one could easily know $SA_k^{-1}[2^e q]$ when $SA_{k+e}^{-1}[q]$ is known. By using this relationship, one could easily get $SA^{-1}[q]$, for $q < i$, given that $SA_h^{-1}[2^{-h}q]$ is known, then using the ψ function (i-q) times to get $SA^{-1}[i]$. But such naive use of the relationship will produce a slow retrieving time as there are h=loglogn levels and there are at most n accesses to the ψ function, totally come with O(n) time. Instead, the authors proposed use the ψ function in lower level(thus ψ_k instead of ψ), so that ψ_k will move the index forward by 2^k instead of 1, to retrieve $SA^{-1}[i]$ in O(loglogn) time.

The locate method in SAD-CSV is identical with GV-CSA, so we skip the discussion here. For the search operation, with the help of the C function, one could easily retrieve the character corresponding to SA[i] by C(i) in constant time without the recursive lookup in GV-CSA. Further with the help of the full ψfunction, the content of T[SA[i]..SA[i]+p] could be retrieved in O(p) times, thus the searching time is asymptotically equal to classical suffix array, O(plogn) time.

The space of SAD-CSV is nH0loglogn+ O(nloglogσ) + σlogσ bits.

5. Technique Based on Backward Search

The inescapable O(logn) search time of the suffix array was ended by the proposal of backward search from Ferragina and Manzini in 2000[5]. Backward searching is a completely different process, intensively adopting the property of the suffix array and suffix. In section 6.1, we will introduce backward searching on the suffix array; and section 6.2 and 6.3 will introduce two more advanced but easy techniques called WT-FMI and RL-FMI respectively.

5.1 Backward searching on Suffix Array.

We start here by an example of backward searching on the suffix array. Given a text T = “abcabcabxaabcd$” and the suffix array SA of T. One could easily obtain an array of string S such that, S[i] = T[SA[i]...n], and a string L = “dx$cacaaaabbbcb” such that L[i] = T[SA[i]-1](figure 5). Now suppose we have a pattern P = “bcb”, and start with the last character “b”, we know the suffix in S[7,12] start with the character “b” in step 1, then in the step 2, we locating suffix S[14,15]. Understanding how to transform from step 1 to step 2 is the core idea of the backward search. It is easy to observe that (1)for $0<i < j < n+1$, if $T[SA[i]] = T[SA[j]]$ and $T[SA[i]-1] = T[SA[j]-1]$, then $\Psi^{-1}(i) < \Psi^{-1}(j)$ and (2)for any i, j such that $T[SA[i]] < T[SA[j]]$ and $T[SA[i]-1] = T[SA[j]-1]$, then $\Psi^{-1}(i) < \Psi^{-1}(j)$. (1) tells us that if two suffixes have a common starting character and a common proceeding character, their relative order is preserved in the suffix starting with the preceding character. (2) tells that if there are n suffix lexicographically smaller than suffixes T[SA[i]...n] and have the same proceeding character T[SA[i]-1], then there are also n suffixes starting with character T[SA[i]-1] lexicographically smaller than T[SA[i]-1...n].

index	L	S
1	d	$
2	x	aabcbd$
3	b	abcabxaabcbd$
4	$	abcbabcabxaabcbd$
5	a	abcbd$
6	c	abxaabcbd$
7	c	babcabxaabcbd$
8	a	bcabxaabcbd$
9	a	bcbabcabxaabcbd$
10	a	bcbd$
11	c	bd$
12	a	bxaabcbd$
13	b	cabxaabcbd$
14	b	cbabcabxaabcbd$
15	b	cbd$
16	b	d$
17	b	xaabcbd$
	step1	

index	L	S
1	d	$
2	x	aabcbd$
3	b	abcabxaabcbd$
4	$	abcbabcabxaabcbd$
5	a	abcbd$
6	c	abxaabcbd$
7	c	babcabxaabcbd$
8	a	bcabxaabcbd$
9	a	bcbabcabxaabcbd$
10	a	bcbd$
11	c	bd$
12	a	bxaabcbd$
13	b	cabxaabcbd$
14	b	cbabcabxaabcbd$
15	b	cbd$
16	b	d$
17	b	xaabcbd$
	step2	

index	L	S
1	d	$
2	x	aabcbd$
3	b	abcabxaabcbd$
4	$	abcbabcabxaabcbd$
5	a	abcbd$
6	c	abxaabcbd$
7	c	babcabxaabcbd$
8	a	bcabxaabcbd$
9	a	bcbabcabxaabcbd$
10	a	bcbd$
11	c	bd$
12	a	bxaabcbd$
13	b	cabxaabcbd$
14	b	cbabcabxaabcbd$
15	b	cbd$
16	b	d$
17	b	xaabcbd$
	step3	

Figure 5, A example of backward search

In the example, by looking at L, there are 1 suffixes preceding with "c" that are smaller than S[7], thus the mapping of S[7] on the 2nd step is S[13+1] = S[14]. For the upper bound index, by (1), we could just do the same checking for S[12], since there are 2 suffixes preceding with "c" that are smaller than S[12], thus the mapping of S[12] is S[13+2] = S[15], the term 13 in S[13+1] or S[13+2] is actually the index of the first suffix starting with "c" in S. In the 3rd step, we have 2 suffix preceding with "b" smaller than S[14] and 3 for S[15], and first index of suffix starting with "b" is 7, thus the final mapping is S[7+2] = S[9] and S[7+3] = S[10].

Hence, by the example above, we could observe, backward search is such a simple technique, once we know the index of first suffix that starts with character c for$1 \leq c \leq \sigma$, and know the value of $Rank_c(L,i)$ for $1 \leq c \leq \sigma$ and $1 \leq i \leq n$. A simple way to achieve this could using an hash table C(with character c,$1 \leq c \leq \sigma$, as key and smallest index of the suffix start with character c in suffix array as value) for the first index recording, and using σ indicator bit vectors B_c($B_c[i] = 1$ if $L[i] = c$, for $1 \leq c \leq \sigma$). For a given S[i] could be simply mapped to $S[C[L[i]]] + Rank_1(B_{L[i]},i)-1]$.

For mapping a single character would take constant time, and for matching a pattern P with size p would require p times character mapping. Thus the count/search time will be O(p). Since the search process will return an matched index interval [sp,ep] of the suffix array and the suffix array is stored explicitly, to locate an occurrence will take O(1) time and O(occ) time in total for locate all occurrences. Since the text is also presented, display time is the same as the length of the content.

For a suffix array to support backward search, locate, and display at the same time, we would need σ indicators bit vector that support fast rank operation, which take $\sigma n + o(\sigma n)$bits, the hash table C, which is negligible if σ is small relative to n, the suffix array $nlogn$bits. Thus the total space is $nlogn + \sigma n + o(\sigma n)$bits.

5.2 WT-FMI(wavelet tree FMI)

From 6.1, It is not hard to notice that we only need L, C, and σ indicator bit vectors for backward searching, while the suffix array and text mostly serve for locate and displaying. And

the nlogn bits property of the suffix array is a drawback for the data structure. Thus, a full text indexed technique called WT-FMI(wavelet tree FMI) , proposed by [25, 9], discards the suffix array and original text to achieve space efficiency.

Before we introduce the FMI techniques, a few related definitions will be listed.

Definition 7[28]: Given a text $T[1,n]$, and $SA[1,n]$ of T, the Burrows-Wheeler transform(BWT) of T, $T^{bwt}[1,n]$, is defined as $T^{bwt}[i] = T[SA[i]-1]$, except when $A[i] = 1$, where $T^{bwt}[i] = T[n]$. That is, $T^{bwt}[i]$ equal to the character precede the i^{th} lexicographically large suffix in T

Definition 8[5]: Given a string $F[1,n]$ and $L[1,n]$,where $F[i] = T[SA[i]], for 1 \leq i \leq n$and $L[i] = T^{bwt}[i]$, The LF-mapping is a function that LF(i) is the location in F that L[i] occurs.

The following lemma gives the formula for the LF-mapping [28, 5].

Lemma 1: Given F and L of text T, and alphabet set {a} in the T, Let C:= {σ} => [1,n], and OCC:= {σ}×[1,n] => [1,n], such that C(c) is the index of first occurrence of c in F and OCC(c,i) is the number of occurrence of character c in L[1..i]. Then, LF(i) = C(L[i]) + OCC(L[i],i)

By **lemma 1**, it could be easily find that the σ indicator bit vectors in 6.1 accomplish the functionality of the OCC(c,i) function, and the hashmap C is equivalent to function C. In addition, it is obvious that if j = LF[i], SA[j] = SA[i]-1, or LF(i) = ψ^{-1}(i)

The WT-FMI used the same C hashmap to represent the C function, which could be easily build by iterate through F and marked C[F[i]] = i if F[i]≠F[i-1], and used a wavelet tree to represent L instead of σ indicator bit vectors. As shown in section 2.2, a wavelet tree support O(logσ) time Access[i], Rank[i] operation. Thus, the Occ(L[i],i) operation is equivalent to $Rank_{Access[i]}(i)$, which will take O(logσ) times, and a search/count operation will totally take O(plogn) times.

Despite the simplicity of the search/count in WT-FMI, the locate and display operation tends to be more challenging, as the original text and the suffix array are not presented. Ferragina and Manzini[5] propose two ways to locate a given occurrence by sampling on the exact position. The first one is novel but slower. For the suffix array SA, we could store the value of SA[i] for $1 \leq i \leq n$, if SA[i] mod $\log^2 n = 0$ explicitly in an array, and mark a bit sequence B, B[i] = 1 if SA[i] is stored. Then for locating SA[i], there is two case, **case 1**:B[i] = 1, or SA[i] is marked equivalently, we could have the result directly, **case 2:**B[i] = 0, then we will run LF(i) at most $\log^2 n$ time to reach j, where j = $LF^v(i) = \psi^{-1\ v}(i)$, $0 < v \leq log^2 n$ and SA[j] is stored explicitly, then SA[i] = SA[j] - v. In this way, locating an occurrence will take $O(\log^2 n)$ time. There are $\frac{n}{log^2 n}$value stored, and each value take logn bits, thus the space occupancy is $O(\frac{n}{logn})$ bits.

The second method was a similar sampling technique, but it used a smaller sampling interval

and multiple sampling levels, only the position at the bottom level was explicitly stored. For a given text T[1,n] = T_1[1,n], its suffix array SA[1,n] = SA_1[1,n] and the corresponding LF(i)=LF_1(i) function of SA. With a bit array B_1[1,n], we set B_1[i] = 1 if SA_1[i] mod l_1 = 0, l_1=$\log^e$n, 1>e>0. Then, we obtain a hypertext T_2[1,n/l_1], where T_2[i] = T_1[(i-1)*l_1+1...i*l_1]. That is, grouping every l_1 characters in T_1 and forming a new character belong to alphabet σ^{l_1}, we then find the SA_2[1,n_1], LF_2(i) of T_2, and build T_3[1,n/l_2], T_4[1,n/l_3]...T_h[1,n/l_{h-1}] and B_2,B_3...B_h in a similar manner, where h is ceil(1/e) in convention(note, l_k= $\log^{ke}$n, instead of $\log^{ke}n_k$). In level h, we store the text position of all the marked indexes(the text position in T instead of in T_h). For locate SA[i], we firstly observe that LF_2(i) <=> LF_1^{l1}(i), LF_3(i)<=> LF_2^{l2}(i), here it is not exactly equal, it just represented that effect of one invoke of LF_2(i) equal to l1 invoke of LF_1(i). In the locating process, there is two cases in level k: if $B_k[i_k]$ = 1, we go into level k+1,and i_{k+1}=$Rank_1(B_k,i_k)$ and set v_k=0; if $B_k[i_k]$ = 0, we do LF_k^{vk}(i) to reach j_k such that $B_k[j_k]$ is 1, then i_{k+1}=$Rank_1(B_k,j_k)$ then we go into level k+1 and record v_k. Such process is recursive from k = 1 to k = h, finally reaching the position i' explicitly stored in level h. Then SA[i] = v_1 + l_1v_2+l_2v_3+....$l_{h-1}v_h$. Since each level there are at most $\log^e$n invokes of LF,which take O($l_k$log σ) time, and the number of levels is a constant h, the final locating time is O($\log^{1+e}$nlogσ), dominated by the $\log^e$n LF mapping in the h level, where l_h = logn. Each level contains a LF_k function, which could be represented by a wavelet tree and take $\frac{n(l(k-1))log\sigma}{(l(k-1))}$=nlogσ+o(nlogσ)bits and a bit array B_k with $n_k$bits. There are n/$\log^{e*ceil(1/e)}$n = n/$\log^{1+\varepsilon}$n explicitly store indexes in last levels, each index will take logn bits, so storing the index will take sublinear bits. Thus, the space occupancy for supporting locate operation is dominated by the wavelet tree and bit array. We also notice that this technique is somehow based on a similar idea as the decompose suffix array in GV-CSA and SAD-CSA, but with different representations.

For displaying T[i...j], we could also accomplish it by using the same technique except that we store the index in a suffix array instead of the text position, but still sampling based on the interval in text position. Then we use the index j of the last character to search for SA[j] by firstly find the marked text position j' > j, and SA[j'] is known, then use the LF function j'-j time to find SA[j] and invoke LF function j-i time to access SA[i] to SA[j]. Then access T[SA[i]] can be achieve by wavelet tree' access operation (access(SA[j+1] to get SA[j],SA[j] to get SA[j-1]) in O(logσ) time for each access,which totally comes in O($\log^{1+e}$nlogσ+(j-i)logσ) time.

The space of WT-FMI is nH_0 + o(n log σ), dominated by the space occupied by the wavelet tree.

5.3 RL-FMI(Run Length FMI)

The RL-FMI proposed by Makinen and Navarro[17] improved over the WT-FMI by taking the identical character sequence in T^{bwt} into consideration. It uses the run-length compressed version of the T^{bwt} to compute the Occ(c,i) function, in other word, the LF function. A run-length compression means that a sequence of consecutive characters c is represented by a single c. For example, let T^{bwt} = "dxb$accaaacabbbbb", then a run-length compression of T^{bwt} result in S = "dxb$acacab".

The main challenge of RL-FMI is that $Rank_c$(i) operation in the wavelet tree of S does not return the correct Occ(c,i), as some consecutive identical characters are compressed into one. Two bit

vectors are required to help the query process, among which: bit vector B[1,n], B[i] =1 if T^{bwt}[i] starts a length, a bit vector B'[1,n], which reorder the runs in B lexicographically, and a function C_s, which function the same as C for T in FMI. To calculate the Occ(c,i), the occurrences of character c in T^{bwt} before index i+1. By B, we could use j = $Rank_1$(B,i) to know the number of runs before index i in T^{bwt}, then by k = $Rank_c$(S,j) we know the number of runs with character c in T^{bwt} before index i. By C_S[c], we could know the number of runs lexicographically smaller than runs with character c in S. Then we have two cases: (1)S_j = c; then we must fully count the number of c in the first j-1 run plus the number of c left over in the j^{th} run by $Select_1$(B',C_s[c]+k)-$Select_1$(B',C_S[c]+1) + (i-$Select_1$(B,j)+1). (2)S_j!=c, then we could count all numbers of c in the first j run by $select_1$(B',C_S[c]+k+1)-$Select_1$(B',C_S[c]+1).

Algorithm RLFM-Occ(c, i, S, B, B′, CS)
(1) $i' \leftarrow rank_I(B, i);$
(2) $j' \leftarrow rank_C(S, i');$
(3) $j \leftarrow select_I(B', C_S[c] + 1);$
(4) *if* $S_i' = c$ *then*
(5) $j' \leftarrow j' - 1;$
(6) $ofs \leftarrow i - select_I(B, i') + 1;$
(7) *else* $ofs \leftarrow 0;$
(8) *return* $select_I(B', C_S[c] + 1 + j') - j + ofs;$
Algorithm 2. Compute Occ(c, i) with the RL-FMI.[12]

For example, B for T^{bwt}="dxb$accaaacabbbbb" is "11111101001110000" and B' is "11100111000010111", S = "dxb$acacab", C_s={"$":0, "a":1, "b": 4, "c": 6, "d":8, "x":9}. To calculate $Rank_a$(T^{bwt},9), we have $Rank_1$(B,9) = j = 7 runs before T^{bwt}[9]. Then $Rank_a$(S,7) = 2 runs corresponding to character "a" before T^{bwt}[9], C_S[a] = 1, there is 1 run of other characters that are lexicographically smaller than "a" in T^{bwt}. Since S[7] = "a", then we need to fully calculate the number of a in the first run and the number of leftover "a" in the second run. The number of leftovers "a" in the second run (i-$Select_1$(B,j)+1) = (9-$Select_1$(B,7)+1) = 9-8+1=2. The number of a in the first run is $Select_1$(B',C_s[c]+k)-$Select_1$(B',C_S[c]+1) = $Select_1$(B',3)-$Select_1$(B',2) = 3-2 = 1. Thus $Rank_a$(T^{bwt},9) = 1+2=3.

With a bit vector that support Select and Rank operation in constant time, the LF(i) function of RL-FMI can also return in O(logσ) time, thus the search/count operation, same as WT-FMI could return in O(plogσ) time. For locating and displaying, one could use an identical as WT-FMI. On the other hand, Navarro and Prezza[11] also proposed a locating algorithm specify for the RL-BWT, based on the finding, one can store O(r) SA values to locate at least one matched occurrence in O(loglogn) time, from Prezza[20], extending to find each occurrence in O(loglogn) time. Interested readers could refer to the original publication for more details.

The space of RL-FMI is nH_k log σ + 2n + o(n log σ), due to the run length compression of the T^{bwt}, H_0 is changed to H_k.

6. Experiment

Experimental setup and environment.

We performed experiments with a 4.20 GHz AMD Ryzen 5 5600X 6-Core Processor equipped with 384 KB L1 cache, 3MB L2 cache, 32 MB L3 cache and 32GB of DDR4 main memory with 64 bits Windows 10. Our program was implemented in Golang version 1.15.8. Our code can be accessed at Github: https://github.com/needon1997/4420project.git.

Experiment 1: Space Usage of Bit sequence

In this experiment, we compare the space usage of Jacobson's bit sequence and the modified version we adopt from [22], for $n=2^i$, $8 \leq i \leq 31$. The plot of the result is shown (in figure 6, 7). And the percentage of space overhead is shown in Table 1.

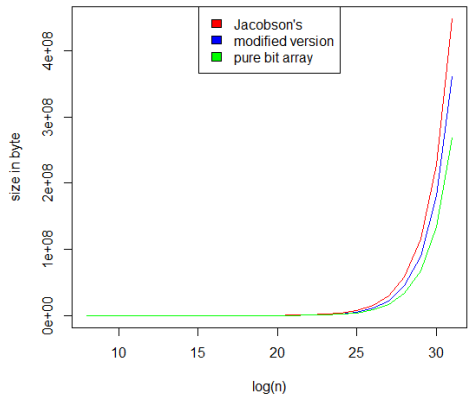

Figure 6. Space of bit sequence

logn	Jacobson's	modified	logn	Jacobson's	modified
8	1 090.625%	1 559.3750%	20	199.354%	134.8808%
9	734.375%	845.3125%	21	193.058%	134.6981%
10	491.406%	489.0625%	22	188.764%	134.6043%
11	379.297%	309.7656%	23	191.972%	134.5545%
12	318.750%	220.7031%	24	188.859%	134.5271%
13	283.496%	176.0742%	25	183.836%	134.5109%
14	250.293%	153.8086%	26	181.555%	134.5006%
15	241.162%	142.6270%	27	177.229%	134.4935%
16	234.473%	140.1367%	28	175.470%	134.4880%
17	224.591%	137.3474%	29	171.710%	134.4834%
18	213.724%	135.9467%	30	170.294%	134.4795%
19	205.998%	135.2386%	31	167.050%	134.4760%

Figure 7. Space overhead with different logn

Experiment 2: Space Usage of Different Techniques on Random Generated Text

In this experiment, we compare the space usage of the surveyed indexing techniques. Original text data were generated randomly in ASCII encoding(1 byte each character), with alphabet size 4 or 26. As the result shows(Figure 8 and Figure 9), the surveyed techniques do perform better than the original suffix array to some extent, with FMI techniques in particular, their space usages are close to original text data. And we could also observe that when the alphabet size is small, the occupied space also reduce, due to wavelet tree with less levels in FMI and longer increasing interval in the Elias δEncoding of CSA. For larger text data, compression ratio also tends to be lower, due to the space overhead decreasing in the bit sequence of our choice, corresponding to the result of experiment 1.

type	size in byte	ratio	type	size in byte	ratio
text alphabet = 26	512016	100%	text alphabet = 26	1048592	100%
suffix array(words)	4608049	900%	suffix array(words)	9437233	900%
suffix array(logn bits)	2935276	573%	suffix array(logn bits)	6273554	598%
gv-csa	2762627	540%	gv-csa	5889938	562%
sad-csa	2873025	561%	sad-csa	5676921	541%
wt-fml [log^2(n) locate]	463138	90%	wt-fml [log^2(n) locate]	938032	89%
rl-fml [log^2(n) locate]	619998	121%	rl-fml [log^2(n) locate]	1258676	120%
type	size in byte	ratio	type	size in byte	ratio
text alphabet = 26	5242896	100%	text alphabet = 26	134217744	100%
suffix array(words)	47185969	900%	suffix array(words)	1207959601	900%
suffix array(logn bits)	34500722	658%	suffix array(logn bits)	1040187442	775%
gv-csa	28386501	541%	gv-csa	719326586	536%
sad-csa	29342209	560%	sad-csa	737845717	550%
wt-fml [log^2(n) locate]	4581492	87%	wt-fml [log^2(n) locate]	114380216	85%
rl-fml [log^2(n) locate]	6182380	118%	rl-fml [log^2(n) locate]	155336768	116%

Figure 8, Space Usage when alphabet size = 26

type	size in byte	ratio	type	size in byte	ratio
text alphabet = 4	1048592	100%	text alphabet = 4	4194320	100%
suffix array(words)	9437233	900%	suffix array(words)	37748785	900%
suffix array(logn bits)	NA	NA	suffix array(logn bits)	NA	NA
gv-csa	4483699	428%	gv-csa	17758332	423%
sad-csa	4308903	411%	sad-csa	17010181	406%
wt-fml [log^2(n) locate]	481690	46%	wt-fml [log^2(n) locate]	1865114	44%
rl-fml [log^2(n) locate]	735658	70%	rl-fml [log^2(n) locate]	2879101	69%
type	size in byte	ratio	type	size in byte	ratio
text alphabet = 4	16777232	100%	text alphabet = 4	67108880	100%
suffix array(words)	150994993	900%	suffix array(words)	603979825	900%
suffix array(logn bits)	NA	NA	suffix array(logn bits)	NA	NA
gv-csa	70256597	419%	gv-csa	277712436	414%
sad-csa	67007437	399%	sad-csa	264270169	394%
wt-fml [log^2(n) locate]	7279264	43%	wt-fml [log^2(n) locate]	28560014	43%
rl-fml [log^2(n) locate]	11334538	68%	rl-fml [log^2(n) locate]	44778881	67%

Figure 9, Space Usage when alphabet size = 4

Experiment 3: Space Usage when Text is Repetitive for FMI techniques

The result of experiment 2 does not prove the benefit of RL-FMI over WT-FMI, since the testing data is not repetitive. As a result, we also run a separate experiment to compare their performance when the text data is repetitive. We randomly generate text data with various sizes and repeat the text by 32 times, then build the RL-FMI and WT-FMI over the repeated text. The text results are shown in (Figure 10). When alphabet size is 26, the RL-FMI outperforms the WT-FMI due to shorted T^{bwt} in the wavelet tree. However, when the alphabet size is 4, their performances get close, as extra space for B and B' required in RL-FMI could take about 50% space of the whole FMI data structure since the wavelet tree has only 2 levels.

type	size in byte	ratio	type	size in byte	ratio
text alphabet = 26, repeated times = 32	33554448	100%	text alphabet = 4, repeated times = 32	33554448	100%
wt-fml [log^2(n) locate]	28845640	86%	wt-fml [log^2(n) locate]	14405763	43%
rl-fml [log^2(n) locate]	13820045	41%	rl-fml [log^2(n) locate]	13297258	40%
type	size in byte	ratio	type	size in byte	ratio
text alphabet = 26, repeated times = 32	134217744	100%	text alphabet = 4, repeated times = 32	134217744	100%
wt-fml [log^2(n) locate]	114384743	85%	wt-fml [log^2(n) locate]	56657183	42%
rl-fml [log^2(n) locate]	54278873	40%	rl-fml [log^2(n) locate]	52202949	39%

Figure 10, Space usage of FMI technique when text is repetitive.

Experiment 4: Search, Locate And Display Query Time

In addition to the space usage, the query running time is also tested. We perform search, locate and display in various sizes of text data(Figure 11). For Search, we will query on random generated patterns with length p = 20, and report the average query time. For locate, we will locate random indexes in the suffix array and report the average query time. For display, random

content of original text with length 20 will be tested. With the help of golang Benchmark function, we do not need to specify the number of tests to calculate the average query time as Benchmark will run the test numerous times until the average query time is stable.

search		
logn = 20	alphabet = 26	alphabet=4
SA	959 ns/op	1058 ns/op
GV-CSA	396064 ns/op	465043 ns/op
SAD-CSA	15002 ns/op	15266 ns/op
WT-FMI	3378 ns/op	2329 ns/op
RL-FMI	20332 ns/op	18133 ns/op
logn = 22		
SA	1261 ns/op	1378 ns/op
GV-CSA	443500 ns/op	645005 ns/op
SAD-CSA	16775 ns/op	16368 ns/op
WT-FMI	3559 ns/op	2494 ns/op
RL-FMI	22665 ns/op	22090 ns/op
logn = 24		
SA	1774 ns/op	1888 ns/op
GV-CSA	539728 ns/op	783262 ns/op
SAD-CSA	17459 ns/op	17359 ns/op
WT-FMI	3893 ns/op	2702 ns/op
RL-FMI	25494 ns/op	26562 ns/op
logn = 28		
SA	2975 ns/op	3295 ns/op
GV-CSA	790738 ns/op	1045631 ns/op
SAD-CSA	20927 ns/op	20786 ns/op
WT-FMI	6149 ns/op	4450 ns/op
RL-FMI	32665 ns/op	37167 ns/op

locate		
logn = 20	alphabet = 26	alphabet=4
SA	NA	NA
GV-CSA	2286 ns/op	2086 ns/op
SAD-CSA	2280 ns/op	2083 ns/op
WT-FMI (log^2n)	77514 ns/op	32660 ns/op
RL-FMI (log^2n)	285473 ns/op	193588 ns/op
logn = 22		
SA	NA	NA
GV-CSA	2612 ns/op	2386 ns/op
SAD-CSA	2601 ns/op	2411 ns/op
WT-FMI (log^2n)	97832 ns/op	42178 ns/op
RL-FMI (log^2n)	381461 ns/op	277223 ns/op
logn = 24		
SA	NA	NA
GV-CSA	2920 ns/op	2617 ns/op
SAD-CSA	2906 ns/op	2646 ns/op
WT-FMI (log^2n)	129751 ns/op	51280 ns/op
RL-FMI (log^2n)	573398 ns/op	372264 ns/op
logn = 28		
SA	NA	NA
GV-CSA	3677 ns/op	3207 ns/op
SAD-CSA	3728 ns/op	3225 ns/op
WT-FMI (log^2n)	377489 ns/op	172495 ns/op
RL-FMI (log^2n)	1080137 ns/op	805659 ns/op

display		
logn = 20	alphabet = 26	alphabet=4
SA	na	na
GV-CSA	na	na
SAD-CSA	214044 ns/op	185068 ns/op
WT-FMI (log^2n)	53032 ns/op	26405 ns/op
RL-FMI (log^2n)	282026 ns/op	182345 ns/op
logn = 22		
SA	na	na
GV-CSA	na	na
SAD-CSA	287111 ns/op	255331 ns/op
WT-FMI (log^2n)	69368 ns/op	31746 ns/op
RL-FMI (log^2n)	388219 ns/op	265066 ns/op
logn = 24		
SA	na	na
GV-CSA	na	na
SAD-CSA	374540 ns/op	342911 ns/op
WT-FMI (log^2n)	94147 ns/op	38111 ns/op
RL-FMI (log^2n)	488226 ns/op	372197 ns/op
logn = 28		
SA	na	na
GV-CSA	na	na
SAD-CSA	607150 ns/op	557505 ns/op
WT-FMI (log^2n)	166863 ns/op	66051 ns/op
RL-FMI (log^2n)	842176 ns/op	650798 ns/op

Figure 11. Query time comparison with various text size and alphabet size

As the experiment result shows, for search operation, the classical suffix array performs the best, which is out of our expectation, as it is unlikely that O(plogn) in the suffix array is performing better than O(plogσ) in FMI. According to our analysis, it is probably due to

speeding up technique used by the suffix array that skips the matched character in the comparison.

Experiment 5: Precompress time

Lastly, we also conduct an experiment to compare the preprocessing time of each technique with various text size and alphabet size. For GV-CSA,SAD-CSA, WT-FMI and RL-FMI, the preprocessing time of the suffix array before transformation is included.

preprocess		
logn = 20	alphabet = 26	alphabet=4
SA	0.14s	0.13s
GV-CSA	0.63s	0.54s
SAD-CSA	0.82s	0.69s
WT-FMI (log^2n)	0.17s	0.16s
RL-FMI (log^2n)	0.48s	0.28s
logn = 22		
SA	1s	0.83s
GV-CSA	3s	2.6s
SAD-CSA	3.7s	3.0s
WT-FMI (log^2n)	1.1s	1.06s
RL-FMI (log^2n)	2.3s	1.4s
logn = 24		
SA	6.5s	5.6s
GV-CSA	15.2s	12.8s
SAD-CSA	17.9s	14.8s
WT-FMI (log^2n)	7.0s	6.1s
RL-FMI (log^2n)	11.3s	7.9s
logn = 28		
SA	139s	124s
GV-CSA	285s	237s
SAD-CSA	330s	268s
WT-FMI (log^2n)	155.8s	135.5s
RL-FMI (log^2n)	235s	158s

Figure 12. Preprocessing time in our implementation.

7. Challenge, Future Work and Conclusion

The greatest challenge we meet in our project including: (1) time constraint, as our initial proposal stated that we will also survey on suffix tree and LZ-compression, but we later notice that we cannot cover them in this project in the limited time, or if we do, the report will be way too longer than the current one, (2)some latest techniques on the topic of text indexing assumes that reader know these basic techniques really well and just briefly describe their idea without any example or procedure description. As a result, our survey is unfortunately only covering some basic and intuitive techniques appear at the beginning of 21st century

Our future work will be multiparts: (1) breaking through the latest techniques, (2) improve the performance of our implementation, (3) explore more compact data representation in addition to the technique itself, (4) try to make the data structure dynamic (5)try to modify the technique we learn to support utf-8 encoding, then (6) build a text indexing engine for practical usage.

In this project, we introduced some text indexing techniques as detailed as possible and shared our ideas of implementation of these techniques. For a review to the reader, a summary table of all of these techniques is provided.(Table 1) In the process of our project, we notice that the techniques we studies, are similar to each other, to some extent. We also learned that some theoretical algorithms might not be perfectly practical in reality, for example, the overhead space of Jacobson's binary sequence. There are many directions that can help to improve text indexing techniques. Finding a new and more efficient technique is undoubtedly a way of improvement. On the other hand, one could simply replace the current data representations, including binary sequence, wavelet tree and elias delta encoding, by some more efficient representations to achieve better performance without changing the technique itself. There are more sophisticated data representations beyond our study, and we unfortunately cannot cover them all due to time constraint and content limitation of the project.

techniques	space	search/count	locate	display m chars
SA	nlogn bits	O(plogn)	O(1)	O(m)
GV-CSA	nH0loglogn+ O(nloglogσ) + nlogσ bits	O(plognloglogn)	O(loglogn)	O(m)
SAD-CSA	nH0loglogn+ O(nloglogσ) + σlogσ bits	O(plogn)	O(logn)	O(m+loglogn)
WT-FMI	nH0 + o(n log σ) bits	O(plogσ)	$O(\log^2 n)$	$O(\log^{1+e} n\log\sigma+m\log\sigma)$ or $O(\log^2 n\log\sigma+m\log\sigma)$
RL-FMI	nHk log σ + 2n + o(n log σ) bits	O(plogσ)	$O(\log^2 n)$	$O(\log^{1+e} n\log\sigma+m\log\sigma)$ or $O(\log^2 n\log\sigma+m\log\sigma)$

Table 1, summary of the studied techniques

8. Reference

[1]Aho, Alfred V.; Corasick, Margaret J. (June 1975). "Efficient string matching: An aid to bibliographic search". Communications of the ACM. **18** (6): 333–340. doi:10.1145/360825.360855. MR 0371172.

[2]D. Clark. Compact Pat Trees. PhD thesis, Univ. Waterloo, 1996.

[3]Elias, Peter (March 1975). "Universal codeword sets and representations of the integers". IEEE Transactions on Information Theory. **21** (2): 194–203. doi:10.1109/tit.1975.1055349.

[4]Farach, Martin (1997), "Optimal Suffix Tree Construction with Large Alphabets" (PDF), 38th IEEE Symposium on Foundations of Computer Science (FOCS '97), pp. 137–143.

[5]Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS) (2000), pp. 390–398.

[6]Ferragina, P. and Manzini, G. 2005. Indexing compressed texts. Journal of the ACM 52, 4, 552–581.

[7]Gagie, T., Navarro, G. D., & Prezza, N. (2017, May). Fast Locating with the RLBWT.

[8]Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In Proceedings of IEEE Data Compression Conference (DCC), pages 193–202. IEEE, 2009

[9]Grossi, R., Gupta, A., and Vitter, J. 2003. High-order entropy-compressed text indexes. In Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (2003), pp. 841–850.

[10]Grossi, R. and Vitter, J. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing 35, 2, 378–407.

[11]Gagie, Travis & Navarro, Gonzalo & Prezza, Nicola. (2017). Fast Locating with the RLBWT.

[12]Gonzalo Navarro and Veli Mäkinen. 2007. Compressed full-text indexes. ACM Comput. Surv. 39, 1 (2007), 2–es. DOI:https://doi.org/10.1145/1216370.1216372

[13]I. Munro. Tables. In Proc. 16th FSTTCS, LNCS n. 1180, pp. 37–42, 1996.

[14]Knuth, Donald; Morris, James H.; Pratt, Vaughan (1977). "Fast pattern matching in strings". SIAM Journal on Computing. **6** (2): 323–350. CiteSeerX 10.1.1.93.8147. doi:10.1137/0206024

[15]Li, Zhize; Li, Jian; Huo, Hongwei (2016). Optimal In-Place Suffix Sorting. Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science. **11147**. Springer. pp. 268–284. arXiv:1610.08305.

[16]McCreight, Edward M. (1976), "A Space-Economical Suffix Tree Construction Algorithm", Journal of the ACM, **23** (2): 262–272, CiteSeerX 10.1.1.130.8022, doi:10.1145/321941.321946

[17]Makinen, V. and Navarro, G. ¨ 2004c. Run-length FM-index. In Proc. DIMACS Workshop: “The Burrows-Wheeler Transform: Ten Years Later” (Aug. 2004), pp. 17–19.

[18]Manber, Udi; Myers, Gene (1993). "Suffix arrays: a new method for on-line string searches". SIAM Journal on Computing. **22** (5): 935–948. doi:10.1137/0222058. S2CID 5074629.

[19]Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In Combinatorial Pattern Matching (CPM), pages 200–210. Springer, 2003.

[20]Nicola Prezza. Compressed Computation for Text Indexing. PhD thesis, Universit`a degli Studi di Udine, 2017.

[21]R. N. Horspool (1980). "Practical fast searching in strings". Software - Practice & Experience. **10** (6): 501–506. CiteSeerX 10.1.1.63.3421. doi:10.1002/spe.4380100608. S2CID 6618295.

[22]R. Gonzàlez, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05), pages 27–38. CTI Press and Ellinika Grammata, 2005.

[23]Sadakane, K. 2000. text databases with efficient Compressed query algorithms based on the compressed suffix array. In Proc. 11th International Symposium on Algorithms and Computation (ISAAC), LNCS v. 1969 (2000), pp. 410–421.

[24]Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48, 2, 294–313.

[25]Sadakane, K. 2002. Succinct representations of lcp information and improvements in the compressed suffix arrays. In Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (2002), pp. 225–232.

[26]Ukkonen, E. 1995. On-line construction of suffix trees. Algorithmica 14, 3, 249–260.

[27]G. Jacobson. Succinct Static Data Structures. PhD thesis, CMU-CS-89-112,
Carn. Mellon Univ., 1989

[28]Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.