Chapter 4

Compressed Suffix Arrays

Fabian Pache

In this work I present Compressed Suffix Arrays from A to Z, starting with ordinary Suffix Arrays, covering Compressed Suffix Arrays as described in [GV00] by Grossi and Vitter in-depth and finishing with an outline of further improvements on Compressed Suffix Arrays developed by K. Sadakane described in [Sad00]

4.1 Introduction

Merely having a certain text usually is not very satisfying. Before long one wants to find the occurences of a smaller text within the larger text. This is called an *enumerative* query. If we are interested only in the number of occurences it is an *counting* query, while *existential* queries only return if there is at least one occurence of the subtext. The terms *larger text* and *smaller text* can be interpreted very liberally. While one of the obvious applications would be using something like this paper as the larger text and for instance 'Suffix Array' as the smaller text there are other applications. The human genome can be seen as a text, admittedly one with a rather small alphabet, with any subsequence being a word or rather a *pattern*.

4.2 Suffix Arrays

The entire idea of a suffix array is to find a certain pattern P within a text T as fast as possible, using as little additional space as possible. A suffix array SA for a text T has as many entries as T has characters. Each entry i of the suffix array points to the position of the i-smallest suffix of T. 'Smallest suffix' in this case refers to the lexicographic ordering of all suffices of T. The order in turn is defined by the alphabet Σ which contains all characters of T. Note that the suffix array is created independant of the pattern P or the length of the pattern. Therefore a suffix array can be used for multiple sequential queries using different patterns efficiently. Searching only once for a single instance of a certain pattern can be done more efficiently using other algorithms. Suffix arrays excel for multiple queries on a static text.

4.2.1 Algorithms

A suffix array in its basic form provides no more than the information where the smallest suffix, second smallest suffix, third smallest suffix and so on starts. It does

not, in itself, provide a function that given a certain pattern, returns the position of the pattern in the text.
However such an alogrithm is quickly outlined once one remembers that the sought pattern is a subset the text. Each subset can in turn be seen as a prefix of a suffix. This is where the suffix array comes in. Each and every suffix is referenced exactly once by the corresponding suffix array. The suffix array contains all suffixes in their lexicographic order therefore all entries of the suffix array pointing to occurences of the pattern are in an unbroken sequence. Note that the entries in the suffix array are sorted, the suffixes in turn usually are not.
Since the occurences of P are in sequence, finding all occurences of P boils down to finding the first and the last occurence. Everything in between matches the pattern as well.

4.2.2 Complexity

Since the only intrinsic function of a suffix array is *lookup*(i), returning the i-smallest suffix in T by table lookup, time complexity for one operation is $O(1)$. Space consumption at this point is considered $O(n)$ for both the text and the suffix array

4.3 Compressed Suffix Arrays

Considering that every text will be stored binary in a digital environment it seems prudent to reduce the alphabet to binary as well. This also gives the highest possible degree of freedom for pattern seeking operations. But it introduces a problem in the size analysis of the text and its accompanying suffix array. A text of n characters can be seen as a bit sequence of $O(m)$ bits where $m = n \log_2 n$. But for each bit, a entry in the suffix array is required and each entry must be able to adress one the $O(m)$ suffixes. Therefore the suffix array 'grows' to $O(m \cdot \log m)$ bits as $\log m$ bits are required to uniquely address one of m entries.
Grossi and Vitter pointed out a clever way of reducing the space requirement back to $O(m)$ without incuring a great speed penalty.

4.3.1 Compression of the Suffix Array

The basic idea of Grossi and Vitter is a recursive divide and conquer algorithm. For each step of the recursion, half the entries of the suffix array are retained for the next step while the other half are stored implicitly.
For now I will only describe the compression of the suffix array itself. The observant reader might note that additional data is required to recover the implictly stored data. Compression of this information is not trivial and will be covered later in section 4.3.2.

Remember that each suffix of T, and therefore each index of $1, \ldots, m$ is refered to only once in SA, the suffix array can be interpreted as a permutation. It follows that the initial steps are applicable to all permutations. However section 4.3.2 will show that it is not a generic method that can be applied to all permutations.
As mentioned before, we will compress the suffix array recursively. In each step of the recursion we remove half the entries. The original suffix array is stored at level $k = 0$ and the recursion is applied often enough so that the suffix array shrinks back to m bits. Since the size of the initial suffix array SA_0 was m entries of $\log m$ bits and the number of bits for each entry is assumed constant Grossi and Vitter reduce the

number of explicitly stored entries. The number of levels K required therefore is

$$\begin{aligned} \frac{m}{2^K} \cdot \log m &= m \\ \log m &= 2^K \\ &\Rightarrow \\ K &= \log\log m \end{aligned}$$

Each step of the recursion removes the odd values and keeps the even values of SA_k. For reasons discussed in section 4.3.2 the kept entries are divided by two in each step. As a side effect of this division the new array SA_{k+1} again contains odd and even values, so the recursion can be applied again.

How to recover the elements removed from SA_k? For now, do not consider the required space, only keep in mind that we want to retain constain time access for each level. Therefore Grossi and Vitter introduce 3 new arrays on each level:

1. A bit vector B_k where

$$B_k[i] = \begin{cases} 1 & \text{if } SA_k[i] \text{ is even} \\ 0 & \text{if } SA_k[i] \text{ is odd} \end{cases}$$

2. A integer mapping ψ_k that is only required to be defined for all i where $B_k[i] = 0$ i.e the entry *will not* be kept in the recursion. The mapping $j = \psi_k[i]$ is used to find the entry $SA_k[j]$ that is one less than $SA_k[i]$. In other words

$$SA_k[i] = SA_k[\psi_k(i)] + 1 \text{ iff } B_k[i] = 0$$

3. A integer vector $rank_k$ where $rank_k[i]$ contains the number of 1s in $B[0..i]$. Like ψ this array is not required to be defined for all entries, but only for those where $B_k[i] = 1$ i.e the entry *will* be kept in the recursion. This array denotes the position of the halfed entry in the next level.

Using these arrays, it is possible to recover SA_k, using SA_{k+1}, B_k, $rank_k$ and ψ_k as follows:

$$SA_k(i) = \begin{cases} 2 \cdot SA_{k+1}[rank_k[i]] & \text{iff } B_k[i] = 1 \\ 2 \cdot SA_{k+1}[rank_k[\psi_k[i]]] + 1 & \text{iff } B_k[i] = 0 \end{cases}$$

The evaluation of $rank_k$ in the second statement is possible since $B_k[\psi[i]] = 1$ iff $B_k[i] = 0$

Grossi and Vitter combine the above formulas by filling the unneccessary entries ($rank_k[i]$ for $B_k[i] = 0$ and ψ_k for $B_k[i] = 1$) with neutral operations and are therefore able to put both cases in a single statement. While this is mathematically very sophisticated it is not a representation that makes the compression and reconstruction scheme easier to understand. Considering modern CPU architectures that have a operations pipeline that can reach its full potential primarily on code that is executed without conditions a unified statement might have advantages in the implementation. Please refer to the original work of Grossi and Vitter [GV00] for details.

This scheme is obviously applied only until the last level of the compression is reached. At this point the values of SA are stored explicitly. Figure 4.1 shows a suffix array for a binary text of length 32. Therefore there are levels $k = 0 \ldots \lceil \log\log 32 \rceil = 0 \ldots 3$. Entries of *rank* and *psi* that are not required in the decompression are left blank.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
T	a	b	b	a	b	b	a	b	b	a	b	b	a	b	a	a	a	b	a	b	a	b	b	a	b	b	b	a	b	b	a	$\#$
SA_0	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25
B_0	0	1	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	1	0	1	0	0	0	1	1	0	1	1	0
$rank_0$		1					2	3		4			5	6	7	8	9	10			11		12				13	14		15	16	
ψ_0	2		14	15	18	23			28		30	31							7	8		10		13	16	17			21			27

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA_1	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11
B_1	1	1	0	1	1	1	0	0	1	0	0	1	0	1	0	0
$rank_1$	1	2		3	4	5			6			7		8		
ψ_1			9				1	6		12	14		2		4	5

	1	2	3	4	5	6	7	8
SA_2	4	7	1	6	8	3	5	2
B_2	1	0	0	1	1	0	0	1
$rank_2$	1			2	3			4
ψ_2		5	8			1	4	

	1	2	3	4
SA_3	2	3	4	1

Figure 4.1: Example of a compressed suffix array

4.3.2 Compression of the Auxiliary Arrays

Meanwhile we compressed the suffix array down to an acceptable size of m bits. On the other hand we optained 3 new array of which only B_k can be stored in m bits.

$rank_k$ is larger by far, requiring $m \log m$ bits on each level. But there is no need to store $rank_k$ explicitly as Guy Jacobson developed a method described in the thesis paper for [Jac89] that can be based on B_k and requires only $o(m)$ bits. The basic idea is to store a two level dictionary that allows for constant access time in the cost model used here.

Compression of ψ_k is more involved than compressing $rank_k$. This is also the point where the compression becomes inapplicable to ordinary permutations. For each level $k \in \{0 \ldots K-1\}$we create 2^{k+1} list. Each of the lists is labeled with a unique binary string of length $k+1$. For each entry $\psi_k(i)$ with $B_k(i) = 0$ we determine the array to store the entry in by looking up a substring t of T. t is defined as a prefix of a suffix in T. The suffix is the one pointed to by the corresponding entry in SA_k. The length of the prefix is determined by the current level of recursion. $t(i) = T((2^k \cdot SA[i]) \ldots (2^k \cdot SA[i] + 2^k - 1))$. We append $j = psi_k(i)$ to the list labeled $t(i)$.

Continuing the example of figure 4.1 the lists are shown in figure 4.2. Note that each of the lists is sorted and the maximum entry in Level k is $\frac{m}{2^k}$ due to the division by two of each entry in SA on each level of the recursion. Thus each of the 2^{k+1} lists can be stored using a bit-vector of length $\frac{m}{2^k}$. The space requirement (without assisting structures to optimize access) is therefore

$$2^{k+1} \cdot \frac{m}{2^k} = O(m)$$

In order to access the compressed entry $j = \psi_k(i)$ we have to determine the number of 0s preceeding entry i in B_k as this is the index to the concatenated lists for level k. This can be done by calculating $h = i - rank_k(i)$ using the technique based on [Jac89] for $rank$ outlined above. Finding the hth entry in the lexicographically ordered lists is not described in detail in [GV00], but Grossi and Vitter claim constant access time while using no more than $O(m)$ bits for access optimzing structures

Level 0:

a list:	2	14	15	18	23	28	30	31
b list:	7	8	10	13	16	17	21	27

Level 1:

aa list:				
ab list:	9			
ba list:	1	6	12	14
bb list:	2	4	5	

Level 2:

aaaa list:		*aaab* list:		
aaba list:		*aabb* list:		
abaa list:		*abab* list:		
abba list:		*abbb* list:	5	8
baaa list:		*baab* list:		
baba list:	1	*babb* list:	4	
bbaa list:		*bbab* list:		
bbba list:		*bbbb* list:		

Figure 4.2: Lists for ψ of figure 4.1

4.3.3 Complexity in general

On each level $0 \leq k \leq \log\log m$ we store B_k, $rank_k$ and the tables for psi_k in $O(m)$ bits. Therefore the entire structure can be stored in $m \log\log m$ bits. Since time is constant for each level $SA(i)$ can be accessed in $O(\log\log m)$.

4.3.4 Complexity optimization

With a small penalty in time requirement it is possible to further reduce the size of the structure to $O(m)$ bits. This is done by using only a subset of the levels. As long as the size of the subset is constant the asymptotic size remains at $O(m)$. Level $k = \log\log m$ is always stored explicitly. Of the other array it suffices to store $1 \leq L < \log\log m$ levels with minor modifications if these L levels are $l(j) = \left\lceil \frac{j}{L} \log\log m \right\rceil$ for $0 \leq j \leq L-1$. To be able to reconstruct $SA_{l(j+1)}$ from $SA_{l(j)}$ it is neccessary modify $B_{l(j)}(i)$ to be 1 only if $SA_{l(j)}(i)$ can be found in $SA_{l(j+1)}(rank_{l(j)}(i))$ instead of $SA_{l(j)+1}(rank_{l(j)}(i))$. Note that $rank$ needs not be modified if based solely on B.

$\psi_{l(j)}$ still has to be defined for all entries where $B_{l(j)} = 0$, only now there are more then half of the entries defined.

This modifactions enable us to use ψ to seek an entry of $B = 1$. The length of the seeking process in turn determines the time requirements. The process is bounded by longest possible sequence that has to be investigated. For a compressed suffix array with L levels this sequence can be no longer than the sequence required to move from level $l(0)$ to $l(1)$. Since all upper bound of sequences on the same level are of equal length, the longest seeking process is

$$\begin{aligned}
\sum_{j=0}^{L-1} \frac{\|l(j)\|}{\|l(j+1)\|} &= \sum_{j=0}^{L-1} \frac{\frac{m}{2^{l(j)}}}{\frac{m}{2^{l(j+1)}}} \\
&= \sum_{j=0}^{L-1} \frac{2^{l(j+1)}}{2^{l(j)}} \\
&= \sum_{j=0}^{L-1} \frac{2^{\frac{j+1}{L}\log\log m}}{2^{\frac{j}{L}\log\log m}} \\
&= \sum_{j=0}^{L-1} \frac{(2^{\log\log m})^{\frac{j+1}{L}}}{(2^{\log\log m})^{\frac{j}{L}}} \\
&= \left(2^{\log\log m}\right)^{\frac{1}{L}} \sum_{j=0}^{L-1} 1 \\
&= (\log m)^{\frac{1}{L}} \cdot L \\
&= O(\log^{\epsilon} m) \text{ with } \epsilon = \frac{1}{L}
\end{aligned}$$

4.4 Extensions of Compressed Suffix Arrays

Leaving the most general case of suffix arrays on binary texts, Sadakane proposes some extensions of suffix arrays on human readable texts. For a more in-depth description of Sadakanes datastructures and algorithms please refer to [Sad00]. Here I can only give a short summary of his work and what it might imply.

4.4.1 Operations

While the original suffix array offers no more functionality than a mere lexicagraphic ordering new operations are included.

Inverse Suffix Array A suffix array answers the question 'At which position i does the j-smallest suffix begin?'. Or more concise $i = SA[j]$ To answer the inverse question 'What is the order j of the suffix starting at position i?'($j = SA^{-1}[i]$) we have no methods available up to now. Sadakane proposes a structure that contains SA^{-1} that has the same time and space requirements as SA.

Searching There are algorithms that allow for searching operations in suffix arrays. Sadakane augments the structure so that searching is an intrinsic feature of the suffix array.

Decompression Using only the suffix array and the introduced extensions but without the original text, recover a substring defined by first and last index in the original text.

The last two functions are based on something Sadakane calls 'the inverse of the array of cumulative frequencies'. The abstract does not elaborate enough on the subject to make a further description possible in this paper.

4.4.2 Complexity

Obviously Sadakane requires more memory to store the suffix array itself, but on the other hand removes the need to store the text. His claim is that the entire search index can be reduced to below the size of the original text for certain texts.

4.5 Conclusion

We started out with a large text and an even larger suffix array. In a first step Grossi and Vitter drastically reduced the size of the suffix array without unreasonable penalties in usability. Sadakane took this process even further and removed the neccessity for storing the text, at the same time adding to the functionality of the datastructure. Meanwhile we are storing neither the text nor the suffix arrays at all. All we retained is a set of hints and literally pointers to both the text and the suffix array. This is a fascinating evolution of a once seeming unchangable structure.