# Two Simple Full-Text Indexes Based on the Suffix Array 

Szymon Grabowski and Marcin Raniszewski<br>Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, 90-924 Łódź, Poland<br>\{sgrabow|mranisz\}@kis.p.lodz.pl


#### Abstract

We propose two suffix array inspired full-text indexes. One, called SAhash, augments the suffix array with a hash table to speed up pattern searches due to significantly narrowed search interval before the binary search phase. The other, called FBCSA, is a compact data structure, similar to Mäkinen's compact suffix array, but working on fixed sized blocks, which allows to arrange the data in multiples of 32 bits, beneficial for CPU access. Experimental results on the Pizza \& Chili 200 MB datasets show that SA-hash is about 2.5-3 times faster in pattern searches (counts) than the standard suffix array, for the price of requiring $0.3 n-2.0 n$ bytes of extra space, where $n$ is the text length, and setting a minimum pattern length. The latter limitation can be removed for the price of even more extra space. FBCSA is relatively fast in single cell accesses (a few times faster than related indexes at about the same or better compression), but not competitive if many consecutive cells are to be extracted. Still, for the task of extracting e.g. 10 successive cells its time-space relation remains attractive.


Keywords: suffix array, compressed indexes, compact indexes, hashing

## 1 Introduction



## 2 Preliminaries

We use 0 -based sequence notation, that is, a sequence $S$ of length $n$ is written as $S[0 \ldots n-1]$, or equivalently as $s_{0} s_{1} \cdots s_{n-1}$.

One may define a full-text index over text $T$ of length $n$ as a data structure supporting at least two basic types of queries, both with respect to a pattern $P$ of length $m$, both $T$ and $P$ over a common finite integer alphabet of size $\sigma$. One query type is count: return the number occ $\geq 0$ of occurrences of $P$ in $T$. The other query type is locate: for each pattern occurrence report its position in $T$, that is, such $j$ that $P[0 \ldots m-1]=T[j \ldots j+m-1]$.

The suffix array $S A[0 \ldots n-1]$ for text $T$ is a permutation of the indexes $\{0,1, \ldots, n-$ $1\}$ such that $T[S A[i] \ldots n-1] \prec T[S A[i+1] \ldots n-1]$ for all $0 \leq i<n-1$, where the " $\prec$ " relation is the lexicographical order. The inverse suffix array $S A^{-1}$ is the inverse permutation of $S A$, that is, $S A^{-1}[j]=i \Leftrightarrow S A[i]=j$.

If not stated otherwise, all logarithms throughout the nonon on in hoan 2.

## 3 Related work

The full-text in collection is the that all non-br: can be built in 1 node, the searc this is cumber also makes the to handle, whi

The main
most economic
$16 n$ in the wor
Most impleme
An import
array of $n$ poil
the sequences (i.e., the suffixes) the pointers store references t
bits for its $n$ suffix pointers (indexes), plus $n \log \sigma$ bits for th
translate
and $O(m$
worst cas
time can
suffix pe
nice tricl
for all th
search in
bits of sp

search time and only $O(\lceil\mathrm{~m} / \mid$ cache_line $\mid\rceil)$ cache misses on average, where $\mid$ cache_line $\mid$ is the cache line length expressed in symbol: CPU. Unfortunately, real texts are far from are bytes, we can use $k$ up to 3 , which off This idea, later denoted as using a lookup $\begin{array}{lr}\text { the suffix } & \text { ose string } \\ \text { given tex } & \text { uirement } \\ \text { converte } & \text { structure } \\ \text { b constant } & \text { of a given } \\ n+o c c) \text { t } & \text { practice, }\end{array}$ as it requires using perfect hashing, which ly in expectation. A small alphabet is easier le use of the suffix tree in bioinformatics.
ree is its large space re ace use reaches almost 9 r $\sigma \leq 256$, and even $m$ or more. iffix tree is the suffix a anged in the order of le2.

A number of suffix well, including the suf space use usually betw are not faster than the

On a theoretical fr $\log \sigma$ ) search time (wi
space), which was recently improved by Fisc $\log \log \sigma$ ) deterministic time, with preserved c

The common wisdom about the practical 1 are comparable, but Grimsmo in his interesting experime a careful ST implementation may be up to about $50 \%$ fas of matches is very small (in particular, one hit), but if the

[^0] SA becomes more competitive, sometimes being even ab faster. Another conclusion from Grimsmo's experiments is moderately faster th


SA easily wins
structures, ed are the referred to
ssed in-
1 count

## ches is

## ate the

issed in
ay (the
al data

## 4

The
and Myers with precomputed interval (bucket) boundaries for $k$ starting symbols tends to b computing costs grow exponentially. Ol the lookup table. Our proposal is to apl extra trick to reduce the number of unr

We start with building the hash tabl for the distinct $k$-symbol ( $k \geq 2$ ) prefixe bwing $k$, but also pre-- needed to be kept in long strings, with an the text.
function is calculated previously built) suffix
array. That is, we process the suffixes 78). The value written suffix with the given siven prefix). Linear probing ash function, we used sdbm

It is assumed that the patvs in the suffix array corren a "standard" lookup table ch with no matches returned ash function over the pattern prefix is calculated and a scan over

HT_build $(T[0 \ldots n-1], S A[0 \ldots n-1], k, z, h()$.
Precondition: $k \geq 2$

```
(01) allocate \(H T[0 \ldots z-1]\)
(02) for \(j \leftarrow 0\) to \(z-1\) do \(H T[j] \leftarrow N I L\)
(03) prevStr \(\leftarrow \varepsilon\)
(04) \(j \leftarrow N I L\)
(05) left \(\leftarrow\) NIL; right \(\leftarrow\) NIL
(06) for \(i \leftarrow 0\) to \(n-1\) do
(07) if \(S A[i] \geq n-k\) then continue
(08) if \(T[S A[i] \ldots S A[i]+k-1] \neq\) prevStr then
(09) if \(j \neq N I L\) then
(10) \(\quad\) right \(\leftarrow i-1\)
                \(H T[j] \leftarrow(\) left, right \()\)
            \(j \leftarrow h(T[S A[i] \ldots S A[i]+k-1])\)
            prevStr \(\leftarrow T[S A[i] \ldots S A[i]+k-1]\)
            repeat
                if \(H T[j]=N I L\) then
                                    left \(\leftarrow i\)
                                    break
                else \(j \leftarrow(j+1) \% z\)
            until false
    \(H T[j] \leftarrow(\) right \(+1, n-1)\)
    return \(H T\)
```

Figure 1. Building the hash table of a given size $z$
the hash table performed until no extra collisions (line 5; return no matches) or found a match over the pattern prefix, which give us information about the range of suffixes starting with the current prefix (line 6). In this case, the bin plied to narrow down the SA interval to contain exactly the whole pattern. (As an implementation note: the binary sea ignore the first $k$ symbols in the comparisons, but it did not due to specifics of the used A_strcmp function from the asr

Pattern_search $(T[0 \ldots n-1], S A[0 \ldots n-1], H T[0 \ldots z-1], k, h($.$) ,$ Precondition: $m \geq k \geq 2$
(1) beg, end $\leftarrow L U T_{2}\left[p_{0}, p_{1}\right]$
(2) if end $<$ beg then report no matches; return
(3) $j \leftarrow h(P[0 \ldots k-1])$
(4) repeat
(5) if $H T[j]=N I L$ then report no matches; return
(6) if $(b e g \leq H T[j] . l e f t \leq e n d)$ and $(T[S A[H T[j] . l e f t] \ldots S A[H T[j] . l e f t]+k-1]=P[0 \ldots k-1])$
(7) then binSearch $(P[0 \ldots m-1], H T[j]$.left, $H T[j]$.right $)$; return
(8) $\quad j \leftarrow(j+1) \% z$
(9) until false

## 5 Fixed Block based Compact S

We propose a variant of Mäkinen's compact sı y feature is finding repeating suffix areas of fixed size, e.g maintain a byte aligned data layout, beneficial for speed by setting a natural restriction on one of the key param building bricks to be multiples of 32 bits, which preven

Mäkinen's index was the first opportunistic scheme for compressing a suffix array, that is such that uses less space on compressible texts. The key idea was to exploit runs in the SA, that is, maximal segments $S A[i \ldots i+\ell-1]$ for which there exists another segment $S A[j \ldots j+\ell-1]$, such that $S A[j+s]=S A[i+s]+1$ for all $0 \leq s<\ell$. This structure still allows for binary search, only the accesses to SA cells require local decompression.

FBCSA_build( $S A[0 \ldots n-1], T^{B W T}$, bs, ss)
/* assume $n$ is a multiple of $b s$ */
(01) $\quad \operatorname{arr}_{1} \leftarrow[] ;$ arr $_{2} \leftarrow[]$
(02) $j \leftarrow 0$
(03) repeat
/* current block of the suffix array is $S A[j \ldots j+b s-1]$ */
(04) find 3 most frequent symbols in $T^{B W T}[j \ldots j+b s-1]$ and store them in $M F S[0 \ldots 2]$
/* if there are less than 3 distinct symbols in $T^{B W T}[j \ldots j+b s-1]$, the trailing cells of MFS[0 $\ldots 2]$ are set to NIL) */
(05) $\quad$ for $i \leftarrow 0$ to $b s-1$ do
(06) if $T^{B W T}[j+i]=M F S[0]$ then $\operatorname{arr}_{1}$.append $(00)$
(07) $\quad$ else if $T^{B W T}[j+i]=M F S[1]$ then $\operatorname{arr}_{1}$.append $(01)$
air 2 .append $\left(S A-\left[S A\left(j+\right.\right.\right.$ pos $\left.\left.\left._{0}\right]-1\right]\right)$
(21) $\operatorname{arr}_{1} \cdot \operatorname{append}(a 2 s)$
(22) $j \leftarrow j+b s$
(23) if $j=n$ then break
(24) until false
(25) return $\left(a r r_{1}, a r r_{2}\right)$


The elements appended to $\operatorname{arr}_{1}$ are single bits or pairs of bits while $a r r_{2}$ stores suffix array indexes (32-bit integers).

The construction makes use of the suffix array $S A$ of text $T$, the inverse suffix array $S A^{-1}$ and $T^{B W T}$ (which can be obtained from $T$ and $S A$, that is, $T^{B W T}[i]=$ $T[(S A[i]-1) \bmod n])$.

Additionally, there are two construction-time parameters: block size $b s$ and sampling step ss. The block size tells how many successive $S A$ indexes are encoded together and is assumed to be a multiple of 32 , for int32-alignment of the structure layout. The parameter $s s$ means that every $s s$-th $S A$ index will be represented verbatim. This sampling parameter is a time-space tradeoff; using larger $s s$ reduces the overall space but decoding a particular SA index typically involves more recursive invocations.

Let us describe the encoding procedure for one block, $S A[j \ldots j+b s-1]$, where $j$ is a multiple of $b s$.

First we find the three most frequent symbols in $T^{B W T}[j \ldots j+b s-1]$ and store them (in arbitrary order) in a small helper array $\operatorname{MFS}[0 \ldots 2]$ (line 04). If the current block of $T^{B W T}$ does not contain three different symbols, the NIL value will be written in the last one or two cell(s) of MFS. Then we write information about the symbols from $M F S$ in the current block of $T^{B W T}$ into arr $_{1}$ : we append 2-bit combination ( 00,01 or 10 ) if a given symbol is from $M F S$ and the remaining combination (11) otherwise (lines 05-09). We also store the positions of the first occurrences of the symbols from $M F S$ in the current block of $T^{B W T}$, using the variables $\operatorname{pos}_{0}, \operatorname{pos}_{1}$, $\operatorname{pos}_{2}$ (lines $10-12$ ); again NIL values are used if needed. These positions allow to use links to runs of suffixes preceding subsets of the current ones marked by the respective symbols from MFS.

We believe that a small example will be useful here. Let $b s=8$ and the current block be $S A[400 \ldots 407]$ (note this is a toy example and in the real implementation bs must be a multiple of 32 ). The $S A$ block contains the indexes: 1000, 522, 801, 303, 906, 477, 52, 610. Let their preceding symbols (from $T^{B W T}$ ) be: $a, b, a, c, d, d$, $b, b$. The three most frequent symbols, written to MFS, are thus: $b, a, d$. The first occurrences of these symbols are at positions: 401, 400 and 404, respectively (that is, $400+\operatorname{pos}_{0}=401$, etc.). The $S A$ offsets: $521(=522-1), 999(=1000-1)$ and $905(=906-1)$ will be linked to the current block. We conclude that the preceding groups of suffix offsets are: $[521,522,523]$ (as there are three symbols $b$ in the current block of $\left.T^{B W T}\right),[999,1000]$ and $[905,906]$.

We come back to the pseudocode. The described (up to three) links are obtained thanks to $S A^{-1}$ (lines 14-16) and are written to $\operatorname{arr}_{2}$. Finally, the offsets of the suffixes preceded with a symbol not from $\operatorname{MFS}$ (if any) have to be written to $\operatorname{arr}_{2}$ explicitly. Additionally, the sampled suffixes (i.e., those whose offset modulo $s s$ is 0 ) are handled in the same way (line 18). To distinguish between referrentially encoded and explicitly written suffix offsets, we spent a bit per suffix and append them to $\operatorname{arr}_{1}$ (lines 19-20). To allow for easy synchronization between the portions of data in arr $_{1}$ and $a r r_{2}$, the size of $\operatorname{arr_{2}}$ (in bytes) as it was before processing the current block is written to $\operatorname{arr}_{1}$ (line 21).

## 6 Experimental results

All experiments were run on a laptop computer with an Intel i3 2.1 GHz CPU, equipped with 8 GB of DDR3 RAM and running Windows 7 Home Premium


In the first experiment we compared pattern search (count) speed using the following indexes:

- plain suffix array (SA),
- suffix array with a lookup table over the first 2 symbols (SA-LUT2),
- suffix array with a lookup table over the first 3 symbols (SA-LUT3),
- the proposed suffix array with deep buckets, with hashing the prefixes of length $k=8$ (only for dna $k=12$ and for proteins $k=5$ is used); the load factor $\alpha$ in the hash table was set to $50 \%$ (SA-hash),
- the proposed fixed block based compact suffix array with parameters $b s=32$ and $s s=5(\mathrm{FBCSA})$,
- FBCSA (parameters as before) with a lookup table over the first 2 symbols (FBCSA-LUT2),
- FBCSA (parameters as before) with a lookup table over the first 3 symbols (FBCSA-LUT3),
- FBCSA (parameters as Enefixes of lenoth $k=8$ (only for dna $k=12$ and for prot $\quad$ sh table was set to $50 \%$ (FBCSA-has
The results are present among the tested ones. The achieved speedups, where $t$ denoted with 1.00.

|  | dna | english | prote |  | xml |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: |
| $m=16$ |  |  |  |  |  |  |
| SA | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |  |
| SA-LUT2 | 1.13 | 1.34 | 1.36 | 1.43 | 1.35 |  |
| SA-LUT3 | 1.17 | 1.49 | 1.61 | 1.65 | 1.47 |  |
| SA-hash | 3.75 | 2.88 | 2.70 | 2.90 | 2.03 |  |
| $m=64$ |  |  |  |  |  |  |
| SA | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |  |
| SA-LUT2 | 1.12 | 1.33 | 1.34 | 1.42 | 1.34 |  |
| SA-LUT3 | 1.17 | 1.49 | 1.58 | 1.64 | 1.44 |  |
| SA-hash | 3.81 | 2.87 | 2.62 | 2.75 | 1.79 |  |

Table 1. Speedups with regard to the search speed of the plain suffix array, for the five datasets and pattern lengths $m=16$ and $m=64$

The SA-hash index has two drawbacks: it requires significantly more space than the standard SA and we assume (at construction time) a minimal pattern length $m_{\text {min }}$. The latter issue may be eliminated, but for the price of even more space use; namely,



Figure 4. Pattern search time (count query). All times are averages over 500K random patterns of the same length $m=\left\{m_{\text {min }}, 16,32,64\right\}$, where $m_{\text {min }}$ is 8 for most datasets except for dna (12) and proteins (5). The patterns were extracted from the respective texts.
we can build one hash table for each pattern length from 1 to $m_{\text {min }}$ (counting queries for those short patterns do not ever need to perform binary search over the suffix array). For the sho $\{1,2,3\})$ lookup tables may be alternatively used.

We have not memory use for ea $q$-grams for $q \leq m$

T" variant, but it is easy to estimate the one needs to know the number of distinct

| $q$ |  |  |  |  |  |
| :---: | ---: | ---: | ---: | ---: | ---: |
|  |  |  | rroteins | sources | xml |
| 1 |  |  | 25 | 230 | 96 |
| 2 |  |  | 607 | 9,525 | 7,054 |
| 3 | 0,222 | 589,000 | 11,607 | 253,831 | 141,783 |
| 4 | 5,892 | $2,150,525$ | $3,624,132$ | $1,719,381$ | $5,252,826$ |
| 5 | 12,804 | $5,566,993$ | $36,525,895$ | $10,669,627$ | $5,716,435,438$ |
| 6 | 28,473 | $11,599,445$ | $94,488,651$ | $17,826,241$ | $8,957,209$ |
| 7 | 80,397 | $20,782,043$ | $112,880,347$ | $26,325,724$ | $12,534,152$ |
| 8 | 279,680 | $33,143,032$ | $117,199,335$ | $35,666,486$ | $16,212,609$ |
| 9 | $1,065,613$ | $48,061,001$ | $119,518,691$ | $45,354,280$ | $20,018,262$ |

Table 2. The number of distinct $q$-grams ( $1 \ldots 10$ ) in the datasets. The number of distinct 12 -grams for dna is $13,752,341$.

The number of bytes for one hash table with $z$ entries and $0<\alpha \leq 1$ load factor is, in our implementation, $z \times 8 \times(1 / \alpha)$, since each entry contains two 4 -byte integers. For example, in our experiment $332,512,688$ bytes, i.e., $158.6 \%$

An obvious idea to reduce tl ing its load factor $\alpha$. The searc several values of $\alpha$ on two datas a reasonable alternative to $\alpha=$ $10 \%$.

|  |  |  |  |  |  |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  |  |  |  |  |
|  | 25 |  |  |  |  |  |

Table 3. A

(in $\mu \mathrm{s}$ ) in function of the HT load factor $\alpha$ for the SA-hash algorithm

Finall
the overall space use for the four non-compact SA variants: implemen each $k \in$ (including
-LUT3 and SA-hash, plus SA-allHT, which is a (not ng a suffix array, a LUT2 and one hash table for pace is expressed as a multiple of the text length $n$ xample 5.000 for the plain suffix array. We note that
the lookup table structures become a relatively smaller fraction when larger texts are indexed. For the variants with hash tables we take two load factors: $50 \%$ and $80 \%$.

|  | dna | english | proteins | sources | xml |
| :--- | ---: | ---: | ---: | ---: | ---: |
| SA | 5.000 | 5.000 | 5.000 | 5.000 | 5.000 |
| SA-LUT2 | 5.001 | 5.001 | 5.001 | 5.001 | 5.001 |
| SA-LUT3 | 5.321 | 5.321 | 5.321 | 5.321 | 5.321 |
| SA-hash-50 | 6.050 | 6.587 | 5.278 | 7.010 | 5.958 |
| SA-hash-80 | 5.657 | 5.992 | 5.174 | 6.257 | 5.600 |
| SA-allHT-50 | 6.472 | 8.114 | 5.296 | 9.736 | 7.353 |
| SA-allHT-80 | 5.920 | 6.947 | 5.185 | 7.960 | 4 |

Table 4. Space use for the non-compact data structures as a multiple of th (including the text), with the assumption that text symbols are represented in offsets are represented in 4 bytes. The value of $m_{\text {min }}$ for SA-hash- 50 and SA-h construction of these structures and affecting their size, is like in the experimen index SA-allHT-* contains one hash table for each $k \in\left\{3,4, \ldots, m_{\text {min }}\right\}$, when $m$ current dataset, as explained. The -50 and -80 suffixes in the structure names d factors (in percent).

In the next set of experiments we evaluated the FBCSA index. Its properties of interest, for various block size ( $b s$ ) and sampling step he space use, pattern search times, times to access (extract) times to access (extract) multiple consecutive SA cells. For $t \quad$ and 64. The $s s$ was tested in a wider range ( $3,5,8,16,32$ ). in better compression but decoding a cell is also slightly slowe


Figure 5. FBCSA index sizes and cell access times with varying ss parameter (3, 5, 8, 16, 32). The parameter bs was set to 32 (left figures) or 64 (right figures). The times are averages over 10M random cell accesses.

Unfortunately, our tests we to adapt other competiti comparison with the resu few times faster in single (augmented with a comp: suffix array) and L c consecutive cells MakCSA and LCS growth is slower tl offsets in a typical our index is still cc





Figure 6. FBCSA, extraction time for $c=5$ (top figures) and $c=10$ (bottom figures) consecutive cells, with varying ss parameter $(3,5,8,16,32)$. The parameter $b s$ was set to 32 (left figures) or 64 (right figures). The times are averages over 1 M random cell run extractions.

## 7 Conclusions

We presented two simple full-text indexes. One, called SA-hash, speeds up standard suffix array searches with reducing significantly the initial search range, thanks to a hash table storing range boundaries of all intervals sharing a prefix of a specified length. Despite its simplicity, we are not aware of such use of hashing in exact pattern matching, and the approximately 3 -fold speedups compared to a standard SA may be worth the extra space in many applications.

The other pres to Mäkinen's con provides int32 ali but not competit

Several aspect scheme collisions
compact variant of the suffix array, related tion works on blocks of fixed size, which is index is rather fast in single cell access, nsecutive cells are to be extr: kes requires further study. In rated with using perfect hash hashing. This should also reduce the overall space use. In case of $p$. standard suffix array component may be replaced with a suffix array with possibly new interesting space-time tradeoffs. The idea of deep be incorporated into some compressed indexes, e.g., to save on the set mapping steps in the FM-index.

## Acknowledgement

The work was supported by the National Science Centre under the project DEC-2013/09/B/ST6/03117 (both authors).

## References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: The enhanced suffix array and its applications to genome analysis, in Algorithms in Bioinformatics, Springer, 2002, pp. 449-463.
2. R. Cole, T. Kopelowitz, and M. Lewenstein: Suffix trays and suffix trists: structures for faster text indexing, in Automata, Languages and Programming, Springer, 2006, pp. 358-369.
3. M. Farach: Optimal suffix tree construction with large alphabets, in Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science, 1997, pp. 137-143.
4. P. Ferragina and J. Fischer: Suffix arrays on words, in CPM, vol. 4580 of Lecture Notes in Computer Science, Springer, 2007, pp. 328-339.
5. P. Ferragina, R. González, G. Navarro, and R. Venturini: Compressed text indexes: From theory to practice. Journal of Experimental Algorithmics, 13(article 12) 2009, 30 pages.
6. P. Ferragina and G. Manzini: Opportunistic data structures with applications, in Proceedings of the 41st IEEE Annual Symposium on Foundations of Computer Science, 2000, pp. 390398.
7. J. Fischer and P. Gawrychowski: Alphabet-dependent string searching with wexponential search trees. arXiv preprint arXiv:1302.3347, 2013.
8. G. Franceschini and R. Grossi: No sorting? Better searching! ACM Transactions on Algorithms, 4(1) 2008.
9. S. Gog and A. Moffat: Adding compression and blended search to a compact two-level suffix array, in SPIRE, vol. 8214 of Lecture Notes in Computer Science, Springer, 2013, pp. 141-152.
10. S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth: Large-scale pattern search using reduced-space on-disk suffix arrays. IEEE Trans. Knowledge and Data Engineering (to appear), 2013, http://arxiv.org/abs/1303.6481.
11. S. Gog and M. Petri: Optimized succinct data structures for massive data. Software-Practice and Experience, 2013, DOI: 10.1002/spe.2198.
12. R. González and G. Navarro: Compressed text indexes with fast locate, in CPM, vol. 4580 of Lecture Notes in Computer Science, Springer, 2007, pp. 216-227.
13. R. González, G. Navarro, and H. Ferrada: Locally compressed suffix arrays. ACM Journal of Experimental Algorithmics, 2014, to appear.
14. N. Grimsmo: On performance and cache effects in substring indexes, Tech. Rep. IDI-TR-200704, NTNU, Department of Computer and Information Science, Sem Salands vei 7-9, NO-7491 Trondheim, Norway, 2007.
15. R. Grossi and J. S. Vitter: Compressed suffix arrays and suffix trees with applications to text indexing and string matching, in Proceedings of the 32nd ACM Symposium on the Theory of Computing, ACM Press, 2000, pp. 397-406.
16. J. KÄrkkÄinen: Suffix cactus: A cross between suffix tree and suffix array, in CPM, vol. 937 of Lecture Notes in Computer Science, Springer, 1995, pp. 191-204.
17. J. KÄrkkÄinen and S. J. Puglisi: Fixed block compression boosting in FM-indexes, in SPIRE, R. Grossi, F. Sebastiani, and F. Silvestri, eds., vol. 7024 of Lecture Notes in Computer Science, Springer, 2011, pp. 174-184.
18. S. Kurtz and B. Balkenhol: Space efficient linear time computation of the Burrows and Wheeler transformation, in Numbers, Information and Complexity, Kluwer Academic Publishers, 2000, pp. 375-383.
19. V. MÄkinen: Compact suffix array, in CPM, R. Giancarlo and D. Sankoff, eds., vol. 1848 of Lecture Notes in Computer Science, Springer, 2000, pp. 305-319.
20. V. MÄkinen: Compact suffix array - a space-efficient full-text index. Fundam. Inform., 56(1-2) 2003, pp. 191-210.
21. U. Manber and G. Myers: Suffix arrays: a new method for on-line string searches, in Proceedings of the 1st ACM-SIAM Annual Symposium on Discrete Algorithms, SIAM, 1990, pp. 319327.
22. G. Navarro and V. Mäkinen: Compressed full-text indexes. ACM Computing Surveys, 39(1) 2007, p. article 2.
23. R. Raman, V. Raman, and S. S. Rao: Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets, in SODA, 2002, pp. 233-242.
24. K. Sadakane: Succinct representations of lcp information and improvements in the compressed suffix arrays, in Proceedings of the 13th ACM-SIAM Annual Symposium on Discrete Algorithms, SIAM, 2002, pp. 225-232.
25. P. Weiner: Linear pattern matching algorithm, in Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, Washington, DC, 1973, pp. 1-11.

[^0]:    he

