Advantages of Backward Searching — Efficient Secondary Memory and Distributed Implementation of Compressed Suffix Arrays

Veli Mäkinen[1], Gonzalo Navarro[2], and Kunihiko Sadakane[3]

[1] Dept. of Computer Science, Univ. of Helsinki, Finland, vmakinen@cs.helsinki.fi
[2] Dept. of Computer Science, Univ. of Chile, Chile, gnavarro@dcc.uchile.cl
[3] Dept. of Computer Science and Communication Engineering, Kyushu Univ., Japan, sada@csce.kyushu-u.ac.jp

Abstract. One of the most relevant succinct suffix array proposals in the literature is the Compressed Suffix Array (CSA) of Sadakane [ISAAC 2000]. The CSA needs $n(H_0 + O(\log\log\sigma))$ bits of space, where n is the text size, σ is the alphabet size, and H_0 the zero-order entropy of the text. The number of occurrences of a pattern of length m can be computed in $O(m \log n)$ time. Most notably, the CSA does not need the text separately available to operate. The CSA simulates a binary search over the suffix array, where the query is compared against text substrings. These are extracted from the same CSA by following irregular access patterns over the structure. Sadakane [SODA 2002] has proposed using *backward searching* on the CSA in similar fashion as the FM-index of Ferragina and Manzini [FOCS 2000]. He has shown that the CSA can be searched in $O(m)$ time whenever $\sigma = O(\text{polylog}(n))$.
In this paper we consider some other consequences of backward searching applied to CSA. The most remarkable one is that we do not need, unlike all previous proposals, any complicated sub-linear structures based on the four-Russians technique (such as constant time *rank* and *select* queries on bit arrays). We show that sampling and compression are enough to achieve $O(m \log n)$ query time using less space than the original structure. It is also possible to trade structure space for search time. Furthermore, the regular access pattern of backward searching permits an efficient secondary memory implementation, so that the search can be done with $O(m \log_B n)$ disk accesses, being B the disk block size. Finally, it permits a distributed implementation with optimal speedup and negligible communication effort.

1 Introduction

The classical problem in string matching is to determine the occurrences of a short pattern $P = p_1 p_2 \ldots p_m$ in a large text $T = t_1 t_2 \ldots t_n$. Text and pattern are sequences of characters over an alphabet Σ of size σ. Usually the same text is queried several times with different patterns, and therefore it is worthwhile to preprocess the text in order to speed up the searches. Preprocessing builds an index structure for the text.

To allow fast searches for patterns of any size, the index must allow access to all suffixes of the text (the ith suffix of T is $t_i t_{i+1} \ldots t_n$). These kind of indexes are called *full-text indexes*. Optimal query time, which is $O(m)$ as every character of P must be examined, can be achieved by using the *suffix tree* [25, 12, 23] as the index.

The suffix tree takes much more memory than the text. In general, it takes $O(n \log n)$ bits, while the text takes $n \log \sigma$ bits[4]. A smaller constant factor is achieved by the *suffix array* [10]. Still, the space complexity does not change. Moreover, the searches take $O(m \log n)$ time with the suffix array (this can be improved to $O(m + \log n)$ using twice the original amount of space [10]).

The large space requirement of full-text indexes has raised the interest on indexes that occupy the same amount of space as the text itself, or even less. For example, the Compressed Suffix Array (CSA) of Sadakane [19] takes in practice the same amount of space as the text compressed with a zero-order model. Moreover, the CSA does not need the text at all, since the text is included in the index. Existence and counting queries on the CSA take $O(m \log n)$ time.

There are also other so-called *succinct* full-text indexes that achieve good tradeoffs between search time and space complexity [3, 9, 7, 22, 5, 14, 18, 16, 4]. Most of these are *opportunistic* as they take less space than the text itself, and also *self-indexes* as they contain enough information to reproduce the text: A self-index does not need the text to operate.

Recently, several space-optimal self-indexes have been proposed [5, 6, 4], whose space requirement dependson the k-th order empirical entropy with constant factor one (except for the sub-linear parts). These indexes achieve good query performances in theory, but they are complex to implement as such.

2 Summary of Results

In this paper, we concentrate on simplifying and generalizing earlier work on succinct self-indexes. We build on the Sadakane's CSA [19], which we briefly review in the following.

The CSA searches the pattern by simulating a binary search over the suffix array. The search string must be compared against some text substring at each step of the binary search. Since the text is not stored, the CSA must be traversed in irregular access patterns to extract each necessary text substring. This makes it unappealing e.g. for a secondary memory implementation.

Sadakane [20] has proposed using *backward searching* on the CSA in similar fashion as the FM-index of Ferragina and Manzini [3]. Sadakane has shown that the CSA can be searched in $O(m)$ time whenever $\sigma = O(\text{polylog}(n))$. The CSA scales much better than the FM-index as the alphabet size grows.

Backward searching has also some other consequences than improved search time, as we will show in this paper. We exploit the fact that the access pattern becomes much more regular. The most important consequence of this is that a

[4] By log we mean $\log_2$ in this paper.

simpler implementation of the CSA is possible: All previous proposals heavily rely on sublinear structures based on the so-called four-Russians technique [1] to support constant time *rank* and *select* queries on bit arrays [8, 13, 2] ($rank(i)$ to find out how many bits are set before position i, and $select(j)$ to find out the position of the jth bit from the beginning). We show that these structures are not needed for an efficient solution, but rather one can do with sampling and traditional compression. This is a distinguishing feature of our proposal. The absence of four-Russians tricks makes our index usable on weaker machine models and also makes it easier to implement.

Under this simpler implementation scenario, we are able to retain the original $O(m \log n)$ search time, and improve the original $n(H_0 + O(\log\log\sigma))$ space to $n(H_0 + \varepsilon)(1 + o(1))$, for any $\varepsilon > 0$. The search time can be reduced gradually, to $O(\lceil m/\ell \rceil \log n)$ time, up to $O(m \log\sigma + \log n)$. The price is that the space requirement increases to $n(\sum_{i=0}^{\ell-1} H_i + \varepsilon)(1 + o(1))$, being H_i the order-i empirical entropy of T. We also give an alternative implementation where the space requirement depends on H_k, for any k. Furthermore, the CSA becomes amenable of a secondary memory implementation. If we call B the disk block size, then we can search the CSA in secondary memory using $O(m \log_B n)$ disk accesses. Finally, we show that the structure permits a distributed implementation with optimal speedup and negligible communication effort. Table 1 compares the original and the new CSA.

	Original CSA	Our CSA, version 1	Our CSA, version 2
Space (bits)	$n(H_0 + O(\log\log\sigma))$	$n(\sum_{i=0}^{\ell-1} H_i + \varepsilon)$	$2n(H_k(\log\sigma + \log\log n) + \varepsilon)$
Search time	$O(m \log n)$	$O(\lceil m/\ell \rceil \log n)$	$O(m \log n)$
Disk search time	$O(m \log n)$	$O(\lceil m/\ell \rceil \log_B n)$	$O(m \log_B n)$
Remote messages	$m \log n$	$\lceil m/\ell \rceil$	m

Table 1. Space and time complexities of the original and new CSA implementations.

Our solution takes about the same amount of space as Sadakane's improvement in [20]. Our search time is worse by a $\log n$ factor. However, our structure works on more general alphabets; we only assume that $\sigma = o(n/\log n)$, as Sadakane's improvement in [20] assumes that $\sigma = O(\text{polylog}(n))$. The space-optimal solutions [5, 6, 4] have also similar restrictions on the alphabet size.

3 The Compressed Suffix Array (CSA) Structure

Let us first describe the basic suffix array data structure. Given a text $T = t_1 t_2 \ldots t_n$, we consider the n text *suffixes*, so that the j-th suffix of T is $t_j t_{j+1} \ldots t_n$. We assume that a special endmarker "\$" has been appended to T, such that the endmarker is smaller than any other text character. The *suffix array* $\mathcal{A}$ of T is the set of suffixes $1 \dots n$, arranged in lexicographic order. That is, the $\mathcal{A}[i]$-th suffix is lexicographically smaller than the $\mathcal{A}[i+1]$-th suffix of T for all $1 \le i < n$.

Given the suffix array, the search for the occurrences of the pattern $P = p_1 p_2 \cdots p_m$ is trivial. The occurrences form an interval $[sp, ep]$ in $\mathcal{A}$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \le i \le ep$, contain the pattern as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$.

The *compressed suffix array (CSA)* structure of Sadakane [19] is based on that of Grossi and Vitter [7]. In the CSA, the suffix array $\mathcal{A}[1 \dots n]$ is represented by a sequence of numbers $\Psi(i)$, such that $\mathcal{A}[\Psi(i)] = \mathcal{A}[i] + 1$. Furthermore, the sequence is differentially encoded, $\Psi(i) - \Psi(i-1)$. If there is a *self-repetition*, that is $\mathcal{A}[j \dots j + \ell] = \mathcal{A}[i \dots i + \ell] + 1$, then $\Psi(i \dots i + \ell) = j \dots j + \ell$, and $\Psi(i') - \Psi(i' - 1) = 1$ for $i < i' \le i + \ell$. Hence the differential array is encoded with a method that favors small numbers and permits constant time access to Ψ. Note in particular that Ψ values are increasing in the areas of $\mathcal{A}$ where the suffixes start with the same character a, because $ax < ay$ iff $x < y$.

Additionally, the CSA stores an array $C[1 \dots \sigma]$, such that $C[c]$ is the number of occurrences of characters $\{\$, 1, \dots, c-1\}$ in the text T. Notice that all the suffixes $\mathcal{A}[C[c]+1] \dots \mathcal{A}[C[c+1]]$ start with character c. The text is discarded.

A binary search over $\mathcal{A}$ is simulated by extracting from the CSA strings of the form $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} t_{\mathcal{A}[i]+2} \dots$ for any index i required by the binary search. The first character $t_{\mathcal{A}[i]}$ is easy to obtain because all the first characters of suffixes appear in order when pointed from $\mathcal{A}$, so $t_{\mathcal{A}[i]}$ is the character c such that $C[c] < i \le C[c+1]$. This is found in constant time by using small additional structures based on the four-Russians technique [8, 13, 2]. Once the first character is obtained, we move to $i' \leftarrow \Psi(i)$ and go on with $t_{\mathcal{A}[i']} = t_{\mathcal{A}[i]+1}$. We continue until the result of the lexicographical comparison against the pattern P is clear. The overall search complexity is the same as with the original suffix array, $O(m \log n)$.

Note that each string comparison may require accessing up to m arbitrary cells in the Ψ array (see Fig. 1). Hence using the CSA in secondary memory is not attractive because of the scattered access pattern. Also, a complex part in the implementation of the CSA is the compression of the Ψ array, since it must support constant time direct access at any position. This is achieved in [19] by using four-Russians techniques, in $n(H_0 + O(\log \log \sigma))$ bits of space.

Notice that the above search only solves *existence* and *counting* queries: We find the interval of the suffix array that would contain suffixes of the text matching the pattern. The pointers to suffixes are not stored explicitly, and hence we cannot report the occurrences or show the text context around them. The solution is to sample suffixes $1, \log n, \dots$, and use the Ψ function to retrieve the unsampled ones [19]. We will only consider counting queries in the sequel, since we can use the sampling technique as is to report occurrences.

4 Backward Search on CSA

Sadakane [20] has proposed using *backward search* on the CSA. Let us review how this search proceeds. We use the notation $R(X)$, for a string X, to denote

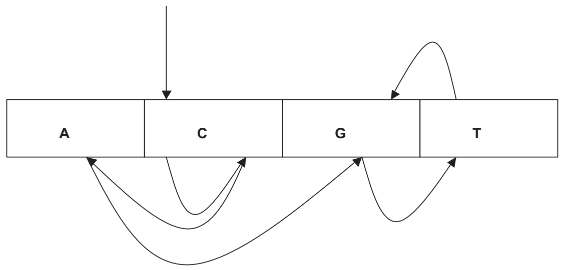

Fig. 1. One step of binary search for pattern $P = CCAGTA$. The blocks correspond to the areas of suffix array whose suffixes start with the corresponding letter. The straight arrow on top indicates the suffix the pattern is compared against. Other arrows indicate the extraction of the prefix of the compared suffix. The extraction ends at letter G, and hence the suffix does not correspond to an occurrence, and the search is continued to the left of the current point.

the range of suffix array positions corresponding to suffixes that start with X. The search goal is therefore to determine $R(P)$. We start by computing $R(p_m)$ simply as $R(p_m) = [C[p_m] + 1, C[p_m + 1]]$. Now, in the general case, given $R(P[i + 1 \ldots m])$, it turns out that $R(P[i \ldots m])$ consists exactly of the suffix array positions in $R(p_i)$ containing values j such that $j + 1$ appears in suffix array positions in $R(P[i + 1 \ldots m])$. That is, the occurrences of $P[i \ldots m]$ are the occurrences of p_i followed by occurrences of $P[i + 1 \ldots m]$. Since $\mathcal{A}[\Psi(i)] = \mathcal{A}[i] + 1$, it turns out that

$$x \in R(P[i \ldots m]) \quad \Leftrightarrow \quad x \in R(p_i) \ \wedge \ \Psi(x) \in R(P[i + 1 \ldots m])$$

Now, Ψ can be accessed in constant time, and its values are increasing inside $R(p_i)$. Hence, the set of suffix array positions x such that $\Psi(x)$ is inside some range forms a continuous range of positions and can be binary searched inside $R(p_i)$, at $O(\log n)$ cost. Therefore, by repeating this process m times we find $R(P)$ in $O(m \log n)$ time.

Fig. 2 gives the pseudocode of the algorithm, and Fig. 3 illustrates.

Note that the backward search (as explained here) does not improve the original CSA search cost. However, it is interesting that the backward search does not use the text at all, while the original CSA search algorithm is based on the concept of extracting text strings to compare them against P. These string extractions make the access pattern to array Ψ irregular and non-local.

In the backward search algorithm, the accesses to Ψ always follow the same pattern: binary search inside $R(c)$, for some character c. In the next sections we study different ways to take advantage of this feature. This is where our exposition differs from [20].

Algorithm $BackwardCSA(P, C, \Psi)$:
$left_{m+1} := 1$; $right_{m+1} := n$;
for $i := m$ **downto** 1 **do begin**
$left_i = \min\{j \in [C[p_i] + 1, C[p_i + 1]],\ \Psi(j) \in [left_{i+1}, right_{i+1}]\}$;
$right_i = \max\{j \in [C[p_i] + 1, C[p_i + 1]],\ \Psi(j) \in [left_{i+1}, right_{i+1}]\}$;
if $left_i > right_i$ **return** "no occurrences found";
return "$right_1 - left_1 + 1$ occurrences found"

Fig. 2. Backward search algorithm over the CSA. Functions "min" and "max" stand for binary searches.

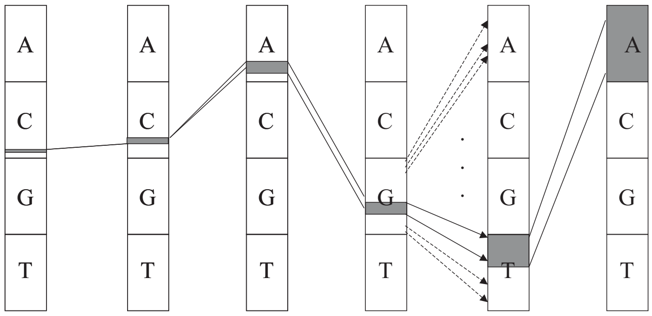

Fig. 3. Searching for pattern $P = CCAGTA$ backwards (right-to-left). The situation after reading each character is plotted. The gray-shaded regions indicate the interval of the suffix array that contain the current pattern suffix. The computation of the new interval is illustrated in the second step (starting from right). The Ψ values from the block of letter G point to consecutive positions in the suffix array. Hence it is easy to binary search the top-most and bottom-most pointers that are included in the previous interval.

5 Improved Search Complexity

A first improvement due to the regular access pattern is the possibility of reducing the search complexity from $O(m \log n)$ to $O(m \log \sigma + \log n)$. Albeit in [20] they obtain $O(m)$ search time, the more modest improvement we obtain here does not need any four-Russians technique.

The idea is that we can extend the C array so as to work over strings of length ℓ (ℓ-grams) rather than over single characters. Given ℓ-gram x, $C[x]$ is the number of text ℓ-grams that are lexicographically smaller than x. The final $\ell - 1$ suffixes of length less than ℓ are accounted as ℓ-grams by appending them as many "\$" characters as necessary.

With this C array, we can search for pattern P of length m in $O(\lceil m/\ell \rceil \log n)$ time as follows. We first assume that m is a multiple of ℓ. Let us write $P =$

$G_1 G_2 \ldots G_{m/\ell}$, where all G_i are all of length ℓ. We start by directly obtaining $R(G_{m/\ell}) = [C[G_{m/\ell}] + 1, C[next(G_{m/\ell})]]$, where $next(x)$ is the string of length $|x|$ that lexicographically follows x (if no such string exists, then assume $C[next(G_{m/\ell})] = n$). Once this is known, we binary search in $R(G_{m/\ell-1})$ the subinterval that points inside $R(G_{m/\ell})$. This becomes $R(G_{m/\ell-1}G_{m/\ell})$. The search continues until we obtain $R(P)$. The number of steps performed is m/ℓ, each being a binary search of cost $O(\log n)$.

Let us consider now the case where ℓ does not divide m. We extend P so that its length is a multiple of ℓ. Let $e = m - (m \bmod \ell)$. Then we build two patterns out of P. The first is P_l, used to obtain the left limit of $R(P)$. P_l is the lexicographically smallest ℓ-gram that is not smaller than P, $P_l = P\$^e$, that is, P followed by e occurrences of character "\$". The second is P_r, used to obtain the right limit of $R(P)$. P_r is the smallest ℓ-gram that is lexicographically larger than any string prefixed by P, $P_r = next(P)\$^e$. Hence, we search for P_l and P_r to obtain $R(P_l) = [sp_l, ep_l]$ and $R(P_r) = [sp_r, ep_r]$. Then, $R(P) = [sp_l, sp_r - 1]$.

Note that $next(x)$ is not defined if x is the largest string of its length. In this case we do not need to search for P_r, as we use $sp_r = n + 1$.

We have obtained $O(\lceil m/\ell \rceil \log n)$ search time, at the cost of a C table with σ^ℓ entries. If we require that C can be stored in n bits, then $\sigma^\ell \log n = n$, that is, $\ell = \log_\sigma n - \log_\sigma \log n$. The search complexity becomes $O(\lceil m/\log_\sigma n \rceil \log n) = O(m \log \sigma + \log n)$ as promised.

Moreover, we can reduce the C space complexity to $O(n/\log^t n)$ for any constant t. The condition $\sigma^\ell \log n = n/\log^t n$ translates to $\ell = \log_\sigma n - (t + 1)\log_\sigma \log n$, and the search cost remains $O(m \log \sigma + \log n)$.

Notice that we cannot use the original Ψ function anymore to find the subintervals, since we read ℓ characters at a time. Instead, we need to store values $\Psi^\ell[i] = \Psi[\Psi[\cdots \Psi[i]] \cdots]$, where Ψ function is recursively applied ℓ times. Next section considers how to represent the Ψ^ℓ values succinctly.

6 A Simpler and Smaller CSA Structure

One of the difficulties in implementing the CSA is to provide constant time access to array Ψ (or Ψ^ℓ using the search procedure from previous section). This is obtained by storing absolute samples every $O(\log n)$ entries and differential encoding for the others, and hence several complex four-Russians-based structures are needed to access between samples in constant time.

Binary Search on Absolute Samples. Our binary searches inside $R(p_i)$, instead, could proceed over the absolute samples first. When the correct interval between samples has been found, we decode the $O(\log n)$ entries sequentially until finding the appropriate entry. The complexity is still $O(\log n)$ per binary search (that is, $O(\log n)$ accesses for the initial binary search plus $O(\log n)$ for the final sequential search), and no extra structure is needed to access Ψ (or Ψ^ℓ). The search is illustrated in Fig. 4.

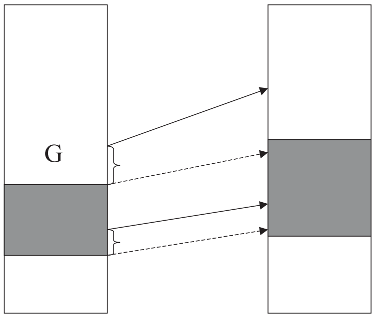

Fig. 4. Binary search followed by sequential search. The top-most sampled value closest to the previous interval is found using binary search (indicated by the top-most solid arrow). Then the next Ψ values are decoded until the first value inside the interval (if exists) is encountered (indicated by the top-most dashed arrow). The same is repeated to find the bottom-most sampled value and then the bottom-most encoded value.

We first consider how Ψ can be encoded. We store $\frac{\varepsilon n}{2 \log n} = O(n/\log n)$ absolute samples of Ψ. For each such sample, we need to store value Ψ in $\log n$ bits, as well as a pointer to its corresponding position in the differentially encoded Ψ sequence, in other $\log n$ bits. Overall, the absolute samples require εn bits, for any $\varepsilon > 0$, and permit doing each binary search in $\log n + \frac{2}{\varepsilon}\log n = O(\log n)$ steps. On the other hand, array C needs $o(n)$ bits by choosing any $t > 1$ for its $O(n/\log^t n)$ entries.

The most important issue is how to efficiently encode the differences between consecutive Ψ cells. The $n(H_0 + O(\log\log\sigma))$ space complexity of the original CSA is due to the need of constant time access inside absolute samples, which forces the use of a zero-order compressor. In our case, we could use any compression mechanism between absolute samples, because they will be decoded sequentially.

Compressing Ψ using Elias encoding. We now give a simple encoding that slightly improves the space complexity of the original CSA.

The differences $\Psi(i) - \Psi(i-1)$ can be encoded efficiently using Elias delta coding [26]. Let $b(p)$ be the binary string representation of a number p. We use $1^{|b(r)|}0b(r)b(p)$ to encode p, where $r = |b(p)|$. The length of the encoding is $\log(2\log p + 1) + 1 + \log p = \log p(1 + o(1))$. The overall length of the codes for all differences can be bounded using the following observation: The Ψ values are increasing inside a suffix array region corresponding to a character c. In other words,

$$\sum_{i,i-1 \in R(c)} |\Psi(i) - \Psi(i-1)| = \sum_{i,i-1 \in R(c)} \Psi(i) - \Psi(i-1) \le n. \qquad (1)$$

To encode the differences for character c, we thus need $\sum_{i,i-1 \in R(c)} (1 + o(1)) \log(\Psi(i) - \Psi(i-1))$ bits. This becomes $|R(c)|(1 + o(1)) \log(n/|R(c)|)$ in the worst case, where $|R(c)| = r - \ell + 1$ is the length of the range $R(c) = [\ell, r]$.

Summing over all characters gives

$$\sum_{c \in \Sigma} |R(c)|(1 + o(1)) \log(n/|R(c)|) = nH_0(1 + o(1)). \qquad (2)$$

Hence the overall structure occupies $n(H_0 + \varepsilon)(1 + o(1))$ bits.

Also, the "small additional structures" mentioned in Section 3, used to find in constant time the character c such that $C[c] < i \leq C[c+1]$, are not anymore necessary. These also made use of four-Russians techniques.

Let us consider how to decode a number p coded with Elias. We can read $1^{|b(r)|}0$ bitwise in $O(|b(r)|) = O(\log r) = O(\log |b(p)|) = O(\log \log p) = O(\log \log n)$ time. Then we get $b(r)$ and $b(p)$ in constant time. The complexity can be lowered to $O(\log \log \log n)$ if we code r as $1^{b(r')}0b(r')b(r)$, where $r' = |b(r)|$. Indeed, we can apply this until the number to code is zero, for a complexity of $O(\log^* n)$. Alternatively, we can decode Elias in constant time by only small abuse of four-Russians technique: Precompute the first zero of any sequence of length $\log \log n$ and search the table with the first bits of $1^{|b(r)|}0b(r)b(p)$. This table needs only $O(\log n \log \log n)$ space.

Due the lack of space we omit the analysis for encoding Ψ^ℓ. In the full version we show that Ψ^ℓ can be encoded to $n \sum_{0 \leq i < \ell} H_i$ bits. We also give an alternative encoding for Ψ combining run-length encoding and Elias encoding to achieve a $2n(H_k(H + \log \log n) + \varepsilon)(1 + o(1))$ bits representation of the structure. (Meanwhile, these analyses appear in a technical report [15, Chapter 5].)

7 A Secondary Memory Implementation

We show now how the regular access pattern can be exploited to obtain an efficient implementation in secondary memory, where the disk blocks can accommodate $B \log n$ bits.

Let us consider again the absolute samples. We pack all the $O(n/\log n)$ absolute samples together, using $O(n/(B \log n))$ blocks. However, the samples are stored in a level-by-level order of the induced binary search tree: For each character c, we store the root of the binary search hierarchy corresponding to the area $C[c] + 1 \ldots C[c+1]$, that is, $\Psi(\lfloor((C[c] + 1) + C[c+1])/2\rfloor)$. Then we store the roots of the left and right children, and so on. When the disk block is filled, the subtrees are recursively packed into disk pages. Fig. 5 illustrates.

Using this arrangement, any binary search inside the area of a character c can make the first $\log B$ accesses by reading only the first disk block. Each new disk block read permits making other $\log B$ accesses. Overall, we find the interval between two consecutive samples in $O(\log(n)/\log(B)) = O(\log_B n)$ disk accesses.

The compressed entries of Ψ are stored in contiguous disk pages. Once we determine the interval between consecutive samples, we sequentially read all the

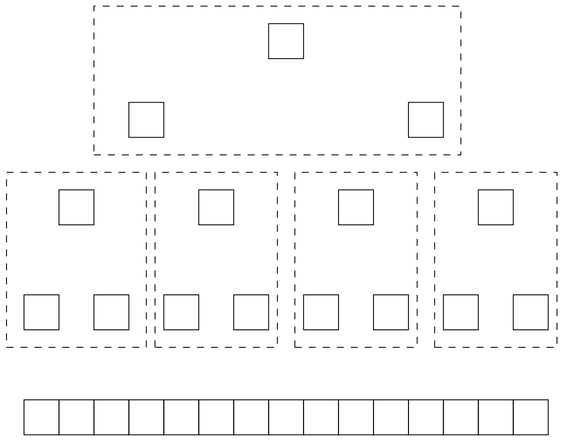

Fig. 5. Packing of array cells to optimize binary search in secondary memory. The dashed boxes indicate cells mapped to a disk block. Any binary search on the array at the bottom is carried out with 2 disk accesses.

necessary disk pages. This requires reading $O(\log(n)/B)$ additional disk pages, which contributes a lower order term to the cost.

Overall, we can maintain the CSA on disk and search it in $O(m \log_B n)$ disk accesses. The original structure requires $O(m \log n)$ disk accesses. If we can hold $O(n)$ bits in main memory, then we can cache all the absolute samples and pay only $O(m \log(n)/B)$ disk accesses.

This scheme can be extended to use a table C of ℓ-grams rather than of individual characters. Each individual binary search takes still $O(\log_B n)$ time, but we perform only $\lceil m/\ell \rceil$ of them.

One obstacle to a secondary memory CSA implementation might be in building such a large CSA. This issue has been addressed satisfactorily [21].

8 A Distributed Implementation

Distributed implementations of suffix arrays face the problem that not only the suffix array, but also the text, are distributed. Hence, even if we distribute suffix array $\mathcal{A}$ according to lexicographical intervals, the processor performing the local binary search will require access to remote text positions [17]. Although some heuristics have been proposed, $\log n$ remote requests for m characters each are necessary in the worst case.

The original CSA does not help solve this. If array Ψ is distributed, we will need to request cells of Ψ to other processors for each character of each binary search step, for a total of $m \log n$ remote requests. Actually this is worse than $\log n$ requests for m characters each.

The backward search mechanism permits us to do better. Say that one processor is responsible for the interval corresponding to each character. Then, we can process pattern characters $p_m, p_{m-1}, \ldots, p_1$ as follows: The processor responsible for p_m sends $R(p_m)$ to the processor responsible for p_{m-1}. That processor binary searches in its local memory for the cells that point inside $R(p_m)$, without any communication need. Upon completing the search, it sends $R(p_{m-1}p_m)$ to the processor responsible for p_{m-2} and so on. After m communication steps exchanging $O(1)$ data, we have the answer.

In the BSP model [24], we need m supersteps of $O(\log n)$ CPU work and $O(1)$ communication each. In comparison, the CSA needs $O(m \log n)$ supersteps of $O(1)$ CPU and communication each, and the basic suffix array needs $O(\log n)$ supersteps of $O(m)$ CPU and communication each.

More or less processors can be accommodated by coarsening or refining the lexicographical intervals. Although the real number of processors, r, is irrelevant to search for one pattern (note that the elapsed time is still $O(m \log n)$), it becomes important when performing a sequence of searches.

If N search patterns, each of length m, are entered into the system, and we assume that the pattern characters distribute uniformly over Σ, then a pipelining effect occurs. That is, the processor responsible for p_m becomes idle after the first superstep and it can work on subsequent patterns. On average the N answers are obtained after Nm/r supersteps and $O(Nm \log(n)/r)$ total CPU work, with $O(Nm/r)$ communication work.

Hence, we have obtained $O(m \log(n)/r)$ amortized time per pattern with r processors, which is the optimal speedup over the sequential algorithm. The communication effort is $O(m/r)$, of lower order than the computation effort. We can apply again the technique of Section 5 to reduce the CPU time to $O((\lceil m/\ell \rceil \log n)/r)$.

9 Conclusions

We have proposed a new implementation of the backward search algorithm for the Compressed Suffix Array (CSA). The new method takes advantage of the regular access pattern to the structure, which allows several improvements over the CSA: (i) tradeoff between search time and space requirement, (ii) simpler and more compact structure implementation, (iii) efficient secondary memory implementation, and (iv) efficient distributed implementation. In particular, ours is the only succinct full-text index structure not relying on four-Russians techniques.

References

1. V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Acad. Nauk. SSSR 194*, 487–488, 1970 (in Russian). English translation in *Soviet Math. Dokl. 11*, 1209–1210, 1975.
2. D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

3. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pp. 390–398, 2000.
4. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An Alphabet-Friendly FM-index. To appear in *11th Symposium on String Processing and Information Retrieval (SPIRE 2004)*, Padova, Italy, October 5-8, 2004.
5. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pp. 841–850, 2003.
6. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, pp. 636-645, 2004.
7. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC'00*, pp. 397–406, 2000.
8. G. Jacobson. *Succinct Static Data Structures.* PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.
9. J. Kärkkäinen. *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.
10. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.
11. G. Manzini. An Analysis of the Burrows-Wheeler Transform. *J. of the ACM* 48(3):407–430, 2001.
12. E. M. McCreight. A space economical suffix tree construction algorithm. *J. of the ACM*, 23, pp. 262–272, 1976.
13. I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.
14. V. Mäkinen. Compact Suffix Array — A space-efficient full-text index. *Fundamenta Informaticae* 56(1-2), pp. 191–210, 2003.
15. V. Mäkinen and G. Navarro. New search algorithms and space/time tradeoffs for succinct suffix arrays. Technical report, C-2004-20, Dept. CS, Univ. Helsinki, April 2004. [http://www.cs.helsinki.fi/u/vmakinen/papers/ssa_tech_2004.ps.gz]
16. V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM'04*, LNCS 3109, pp. 420–433, 2004.
17. M. Marín and G. Navarro. Distributed query processing using suffix arrays. In *Proc. SPIRE'03*, pages 311–325, LNCS 2857, 2003.
18. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms* 2(1):87–114, 2004.
19. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC'00*, LNCS 1969, pp. 410–421, 2000.
20. K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. SODA 2002*, ACM-SIAM, pp. 225–232, 2002.
21. K. Sadakane. Constructing compressed suffix arrays with large alphabets. In *Proc. ISAAC'03*, LNCS 2906, pp. 240–249, 2003.
22. S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Inf. Proc. Lett.*, 82 (6), pp. 307-311, 2002.
23. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14, pp. 249–260, 1995.
24. L. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.
25. P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Ann. Symp. on Switching and Automata Theory*, pp. 1–11, 1973.
26. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, New York, second edition, 1999.