Contextual Pattern Matching*

Gonzalo Navarro[0000−0002−2286−741X]

CeBiB — Center for Biotechnology and Bioengineering,
Department of Computer Science, University of Chile.
Beauchef 851, Santiago, Chile. gnavarro@dcc.uchile.cl

Abstract. The research on indexing repetitive string collections has focused on the same search problems used for regular string collections, though they can make little sense in this scenario. For example, the basic pattern matching query "list all the positions where pattern P appears" can produce huge outputs when P appears in an area shared by many documents. All those occurrences are essentially the same.
In this paper we propose a new query that can be more appropriate in these collections, which we call *contextual pattern matching*. The basic query of this type gives, in addition to P, a context length ℓ, and asks to report the occurrences of all *distinct* strings XPY, with $|X| = |Y| = \ell$. While this query is easily solved in optimal time and linear space, we focus on using space related to the repetitiveness of the text collection and present the first solution of this kind. Letting $\overline{r}$ be the maximum of the number of runs in the BWT of the text $T[1..n]$ and of its reverse, our structure uses $O(\overline{r}\log(n/\overline{r}))$ space and finds the c contextual occurrences XPY of (P, ℓ) in time $O(|P|\log\log n + c\log n)$. We give other space/time tradeoffs as well, for compressed and uncompressed indexes.

1 Introduction

About a decade ago, it was realized that many of the fastest-growing text collections of the "data deluge" were highly repetitive [17]. Since then, a number of research results have focused on developing indexes whose size is related to some good measure of compressibility for highly repetitive string collections [21]. Today one can find indexes built on measures like the size of the Lempel-Ziv parse [16, 11, 9, 4], of a grammar generating only the text [8, 25], of a string attractor [23, 7], the number of runs in the Burrows-Wheeler Transform (BWT) [6] of the text [17, 12], or the size of an automaton [5] recognizing text substrings [2, 1].

All these indexes are devoted to the basic *pattern matching* query: given a short pattern string $P[1..m]$, output all the *occ* positions where it occurs in the text $T[1..n]$. Some indexes have managed to solve this problem in optimal time, $O(m + occ)$, using space bounded by some function of the above measures [1, 12], whereas others have low polylogarithmic factors multiplying m or *occ*.

While very reasonable in general, this query can be pretty useless in a highly repetitive text collection. A pattern P that appears inside a highly repeated

* Supported in part by Fondecyt grant 1-200038 and Basal Funds FB0001, Chile.

text area will be reported myriad times, wasting a lot of effort to produce and to handle the result. We are not aware of many efforts to propose queries that are better adapted to a scenario of high repetitiveness.

In this paper we make a first step in this direction. We propose a query called *contextual pattern matching* which, in addition to P, gives a context length ℓ. We then want one element of output per distinct context where P appears, that is, all the positions where P appears preceded by the same string X of length ℓ and followed by the same string Y of length ℓ shall be reported only once.

Definition 1. *The* contextual pattern matching *problem on a text $T[1..n]$ is, given a pair $(P[1..m], \ell)$, return a position in T for each of the c distinct strings XPY occurring in T, for all X, Y such that $|X| = |Y| = \ell$. For the occurrences near the extremes of T, assume T is preceded and followed by ℓ copies of the special symbol \$, which cannot appear in P.*

It is not hard to solve this query in optimal time $O(m + c)$ if we use linear space, $O(n)$, by using suffix trees [26] and other linear-space auxiliary structures. We are interested, however, in using space related to a relevant repetitiveness measure. We show that, if we call $\overline{r}$ the maximum of the number of equal-letter runs in the BWT of T or its reverse, then a data structure using $O(\overline{r} \log(n/\overline{r}))$ space can solve contextual pattern matching in time $O(m \log\log n + c \log n)$. We also show how any compressed text index can be extended with $O(n)$ *bits* and efficiently solve this query; this can be interesting for mildly repetitive texts.

2 Preliminaries

We index a text $T[0..n]$ over alphabet $[1..\sigma]$, where $T[0] = T[n] = \$$ is a special terminator smaller than all the other alphabet symbols. The *suffix array* [18] $SA[1..n]$ of T lists all the suffixes $T[i..n]$ for $i \geq 1$ in lexicographic order, and the *LCP array*, $LCP[1..n]$, gives the length of the longest common prefix between consecutive suffix array entries, $LCP[i] = lcp(T[SA[i]..n], T[SA[i-1]..n])$.

One relevant measure of repetitiveness is called r, the number of equal-letter runs in the Burrows-Wheeler Transform (BWT) of $T[1..n]$. The BWT [6] is a reordering of the symbols of T obtained by collecting the symbol preceding the lexicographically sorted suffixes of T. That is, if $SA[1..n]$ is the suffix array of T, then $BWT[i] = T[SA[i] - 1]$. For example, it is known that $r = O(\gamma \log^2 n)$ [14], where γ is the smallest attractor of T [15].

Gagie et al. [12] introduce data structures of size $O(r)$ that can find the suffix array range of any pattern $P[1..m]$ in time $O(m \log\log(\sigma + n/r)) \subseteq O(m \log\log n)$, and of size $O(r \log(n/r))$ that can compute any entry $SA[i]$, $SA^{-1}[i]$, and $LCP[i]$, in time $O(\log(n/r))$. The $O(\log(n/r))$-space data structures are binary context-free grammars of height $O(\log(n/r))$ built on the differential versions of the arrays, for example, $DSA[i] = SA[i] - SA[i-1]$ in the case of the suffix array. The grammars exploit the fact that these differential sequences inherit the repetitiveness of the text.

3 Our Solution

We present a suffix-array-oriented solution that solves a stronger variant of the problem: we give the c suffix array ranges of all the distinct contexts XPY where P occurs in T. We can then report one text position for each, but also determine how many times each context occurs, and report its occurrences one by one.

We store the r-bounded data structures of Gagie et al. [12] for both $T[0..n]$ and its reverse $T^{rev}[0..n]$. We call $\overline{r}$ the maximum of the number of equal-letter runs in the BWT of T and of T^{rev}, therefore the structures we use take space $O(\overline{r}\log(n/\overline{r}))$. The general strategy to solve a query $(P[1..m], \ell)$ is as follows:

1. We first find, in $O(m \log\log n)$ time, the suffix array range $[rs..re]$ of P^{rev} (i.e., P read backwards) in the suffix array SA' of T^{rev}.
2. We then partition $[rs..re]$ into $k \leq c$ maximal consecutive intervals $[rs_i, re_i]$ where the suffixes in each interval share their first $m + \ell$ symbols, that is, $T^{rev}[SA'[p]..SA'[p] + m + \ell - 1] = P^{rev}X_i^{rev}$ for all $rs_i \leq p \leq re_i$.
3. We map each interval $SA'[rs_i, re_i]$ to the interval $SA[ds_i..de_i]$ corresponding to the suffixes that start with X_iP.
4. We partition each interval $SA[ds_i..de_i]$ into k_i maximal consecutive subintervals $SA[ds_i^j..de_i^j]$ where the suffixes in each subinterval share their first $m+2\ell$ symbols, $T[SA[p]..SA[p]+m+2\ell-1] = X_iPY_j$ for all $ds_i^j \leq p \leq de_i^j$.
5. We report the $c = \sum_{i=1}^{k} k_i$ resulting subintervals $SA[ds_i^j..de_i^j]$ and, if desired, a text position $SA[p]$ with $ds_i^j \leq p \leq de_i^j$ for each.

We now solve the two nonobvious subproblems of our general strategy. The first, in points 2 and 4, is to partition a suffix array interval into subintervals of suffixes sharing their first t symbols. The second, in point 3, is how to map an interval of the suffix array of T^{rev} into the corresponding interval in the suffix array of T. The solutions we find have a complexity of $O(\log n)$ per item output, which leads to our promised result.

Theorem 1. *Let T be a text of length n, and let $\overline{r}$ be the maximum of the number of equal letter runs of its BWT and the BWT of its reverse. Then there is a data structure of size $O(\overline{r}\log(n/\overline{r}))$ that finds the c contextual occurrences of $(P[1..m], \ell)$ in time $O(m \log\log n + c \log n)$.*

The data structures [12] can be built in $O(n)$ time and space, or in $O(n \log n)$ time and $O(\overline{r}\log(n/\overline{r}))$ space, the same as the final space of the structures. The extra data we add next do not change the space nor construction complexities.

Example. Figure 1 shows an example on the text $T[0..17] =$ $\mathsf{alabaralalabarda\$}$, where we search for $P = \mathsf{a}$ with context length $\ell = 1$. Step 1 finds the interval $SA'[rs..re] = SA'[2..9]$ of all the occurrences of $P^{rev} = \mathsf{a}$ on T^{rev}. Step 2 finds the places where $LCP'[p] < m + \ell = 2$ (see Section 3.1), for $p \in [2..9]$, namely $2, 3, 5, 6, 9$. These are the starting positions of the intervals $[rs_i, re_i] = [2, 2], [3, 4], [5, 5], [6, 8], [9, 9]$, and correspond to the contexts $P^{rev}X_i^{rev} = \mathsf{a\$}, \mathsf{ab}, \mathsf{ad}$,

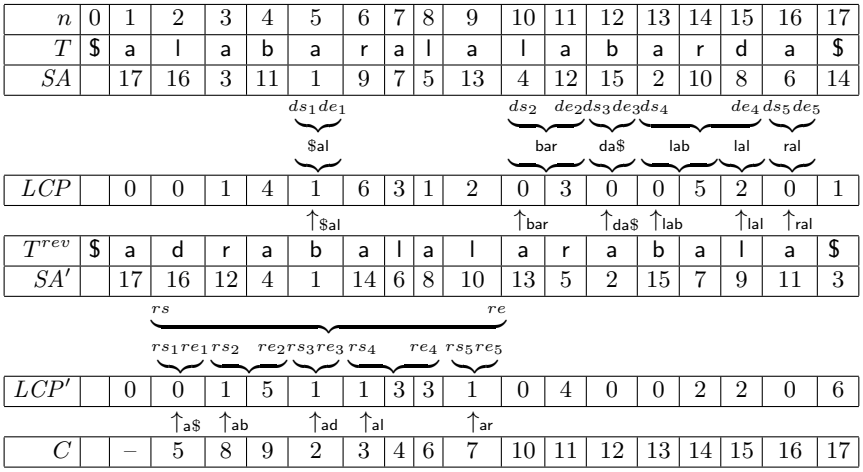

Fig. 1. Example trace.

al, ar. Step 3 maps those intervals to SA (see Section 3.2), $[ds_i, de_i] = [5,5], [10,11], [12,12], [13,15], [16,16]$; they retain the same order of SA' only because $\ell = 1$. Step 4 splits each interval at subintervals starting wherever $LCP[p] < m+2\ell = 3$, namely positions $5, 10, 12, 13, 15, 16$. Therefore, the resulting subintervals (i.e., the output) are [5, 5], [10, 11], [12, 12], [13, 14], [15, 15], [16, 16], corresponding to the contexts al, bar, da, lab, lal, ral.

We also show the array C used in Section 3.3; note that each ds_i corresponds to mapping the minimum position of C in $[rs_i, re_i]$.

3.1 Partitioning a suffix array interval

Given a range $[s..e]$ of the suffix array of a string S, and a length t, we must partition it into maximal subranges $[s_1..e_1], \ldots, [s_k..e_k]$ where the suffixes starting in each subrange share their first t symbols. Note that the positions $s_2, \ldots, s_k$ are the values in $[s..e]$ where $LCP[i] < t$, where LCP is the LCP array of S.

We store a binary grammar of height $h = O(\log n)$ on $DLCP[i] = LCP[i] - LCP[i-1]$, with $DLCP[1] = 0$ [12]. For each grammar nonterminal A, expanding to a sequence $exp(A)$ of positive and negative integers of $DLCP$ (and for terminals A, assuming $exp(A) = A$), we store

- $w(A) = |exp(A)|$, the number of consecutive $DLCP$ cells A expands to;
- $s(A)$, the sum of the differential values in $exp(A)$, $s(A) = \sum_{j=1}^{w(A)} exp(A)[j]$.
- $m(A)$, the minimum cumulative value reached inside $exp(A)$, that is, $m(A) = \min_{1 \le i \le w(A)} \sum_{j=1}^{i} exp(A)[j]$.

With $w(A)$ and $s(A)$, a standard procedure descends in time $O(h)$ to the sth and eth leaves in the parse tree of $DLCP$, finding both (1) the value of $LCP[s]$

and (2) the $O(h)$ maximal nodes (regarding ancesorship) that cover $DLCP[s..e]$ in the parse tree. To reach the xth leaf, we descend from the root node and move to the left child A if $w(A) \geq x$, otherwise we move to the right child and decrease x by $w(A)$. To find (1) we add up the values $s(A)$ of the left children A every time we descend to the right child in the path to the sth leaf. To find (2) we do the paths to the sth and eth leaves and, once they diverge at a node v, we collect the right children when we go left in our path from v to the sth leaf, and the left children when we go right in our path from v to the eth leaf.

Let A_i be the $O(h)$ maximal parse tree nodes that cover $DLCP[s..e]$. They start in $[s..e]$ at positions $p_1 = s$ and $p_{i+1} = p_i + w(A_i)$. The LCP values at the positions p_i are $l_1 = LCP[s]$ and $l_{i+1} = l_i + s(A_i)$. Note then that each A_i where $l_i + m(A_i) < t$ contains at least one position s_j where $LCP[s_j] < t$; the others can be discarded.

For each A_i where $l_i + m(A_i) < t$, we consider its rule $A_i \to BC$. Note that B and C start at $p = p_i$ and $p' = p_i + w(B)$ and their first LCP values are $l = l_i$ and $l' = l_i + s(B)$, respectively. We recursively continue with B if $l + m(B) < t$ and with C if $l' + m(C) < t$ (we can continue by both). When we arrive at a terminal grammar symbol, we can report its value p as a new position s_j.

We then report the positions $s_2, \ldots, s_k$ in left-to-right order by considering $A_1, A_2, \ldots$ in turn and considering B before C when $A_i \to BC$. Since every time we consider a node we know that it contains an answer, the total time is $O(h)$ plus $O(h)$ for each of the $k-1$ starting positions $s_2, \ldots, s_k$. The total time is then $O(k\,h) \subseteq O(k \log n)$, that is, $O(\log n)$ per range we output.

3.2 Mapping suffix array intervals

Given the suffix array interval $SA'[s'..e']$ of T^{rev}, consisting of all the suffixes that start with a string of length t, we want to find the corresponding suffix array interval $SA[s..e]$ of T. With the suffix array SA' of T^{rev} and the inverse suffix array SA^{-1} of T, we can translate any such suffix, say $p = SA^{-1}[n - SA'[s'] - (m + \ell - 1)]$ (or $p = SA^{-1}[1]$ if $n - SA'[s'] - (m + \ell - 1) \leq 0$). Our index stores the structures to compute those in time $O(\log n)$ [12].

We know that $s \leq p \leq e$, so the task is to extend p in both directions: $s \leq p$ is the largest position where $LCP[s] < t$ and $e \geq p$ is the smallest position where $LCP[e+1] < t$. We show how to find e; the case of s is analogous.

Just as in Section 3.1, we compute $LCP[p]$ and find the $O(h)$ maximal nodes $A_1, \ldots$ that cover the area $DLCP[p..n]$. We then compute the values p_i and l_i, and scan $A_1, \ldots$ for the first A_i such that $l_i + m(A_i) < t$. Then, if $A_i \to BC$, we continue by B if $l_i + m(B) < t$; otherwise we continue by C with values $p = p_i + w(B)$ and $l = l_i + s(B)$. In $O(h)$ time we reach a terminal symbol, whose position p is, precisely, $e + 1$. The total time is then $O(h) = O(\log n)$.

3.3 Running on Other Indexes

If we are willing to store uncompressed data structures of $O(n)$ space, we can find the interval of point (1) in RAM-optimal time $O(m/\log_\sigma n)$ using an enhanced

suffix tree [22] on T^{rev}. The k intervals $[rs_i, re_i]$ of point (2) can be found in $O(k)$ time using range minimum queries on the LCP array of T^{rev}, LCP': $rmq(i,j) = \min_{i \le p \le j} LCP'[p]$. We use the standard procedure for 3-sided queries: compute $p = rmq(rs, re)$ and, if $LCP'[p] < t$, recurse on $[rs, p-1]$, report p, and recurse on $[p+1, re]$. Queries rmq take constant time even using $2n + o(n)$ bits of space [10]. Each such interval $SA'[rs_i, re_i]$ can then be mapped (point 3) to $SA[ds_i, de_i]$ by storing an array $C[1..n]$ with $C[i] = SA^{-1}[n - SA'[i]]$ and building an rmq data structure on C, so that $ds_i = SA^{-1}[n - SA'[rmq_C(rs_i, re_i)] - (m + \ell - 1)]$ and $de_i = ds_i + (re_i - rs_i)$. (Note that we build C on the values $SA^{-1}[n - SA'[i]]$, not $SA^{-1}[n - SA'[i] - (m - \ell + 1)]$, because the latter depend on ℓ and all the suffixes in this range share their first $m + \ell$ symbols anyway, so the lexicographic comparison is the same.) Finally, point (4) on each $SA[ds_i, ds_i]$ is solved as for point (2), now on the LCP array of T. The total time is then the optimal $O(m/\log_\sigma n + c)$.

Theorem 2. *Let T be a text of length n over an alphabet of size σ. Then there is a data of size $O(n)$ that finds the c contextual occurrences of $(P[1..m], \ell)$ in time $O(m/\log_\sigma n + c)$.*

More generally, if we have an index that finds the suffix array range $[rs..re]$ for P in T^{rev}, and can extract any cell of SA, SA^{-1}, and SA', we can use it for contextual reporting using our general solution. We need $O(n)$ extra bits for the various rmq data structures. Note we do not need to store C explicitly because we can simulate it using SA' and SA^{-1}. Further, the arrays LCP' and LCP are simulated with other $2n + o(n)$ bits if we have access to SA' and SA [24]. We then have the following result.

Theorem 3. *Let T be a text of length n and an index on T^{rev} using $\mathcal{S}$ bits of space that finds the suffix array range of $P[1..m]$ in time $t_s(m)$, and computes any cell of SA, SA', or SA^{-1} in time t_{SA}, where SA and SA' are the suffix arrays of T and T^{rev}, respectively. Then there is a data structure using $\mathcal{S} + O(n)$ bits of space that finds the c contextual occurrences of $(P[1..m], \ell)$ in time $O(t_s(m) + c\, t_{SA})$.*

Building on an index [3] that uses $nH_k(T^{rev}) + o(n \log \sigma) + O(n)$ bits of space for any $k < \alpha \log_\sigma n$ and constant $0 < \alpha < 1$, where $H_k(S) < \log \sigma$ is the kth order empirical entropy of string S [19], we have $t_s(m) = O(m)$ and $t_{SA} = O(\log n)$. The index provides access to SA' and $(SA')^{-1}$ by storing their values at regular intervals of T^{rev}, of length $s = \Theta(\log n)$ in our case, and marking the sampled positions of SA' in a bitvector. It provides a way to move in constant time from i such that $SA'[i] = j$ to $i' = LF(i)$ such that $SA'[i'] = j - 1$. Thus, if $SA'[i]$ is not sampled, it can move $s' < s$ times until finding a sampled cell $SA'[LF^{s'}(i)] = j'$, and then $SA'[i] = j' + s'$. The same LF function is used $j' - j < s$ times, for $j' = \lceil j/s \rceil \cdot s$, to find $(SA')^{-1}[j]$, by starting from the sampled value $(SA')^{-1}[j']$ and tracing it back to $(SA')^{-1}[j] = LF^{j'-j}((SA')^{-1}[j'])$. Enhancing it to computing values of SA and SA^{-1} (which correspond to T) requires to store their sampled values as well, because $T^{rev}[j] = T[n - j]$. Finally, because $H_k(T) = H_k(T^{rev})$ [20, Sec. 11.3.2], we have the following result.

Theorem 4. *Let T be a text of length n over an alphabet of size σ, with kth order empirical entropy $H_k(T)$, for any $k < \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. Then there is a data structure of $nH_k(T) + o(n \log \sigma) + O(n)$ bits that finds the c contextual occurrences of $(P[1..m], \ell)$ in time $O(m + c \log n)$.*

We can speed up this index by using *compact* space, $O(n \log \sigma)$ bits (i.e., proportional to a plain representation of T). In this case, any cell of SA or SA^{-1} (and of SA' by building the structures on T^{rev} as well) can be computed in time $O(\log_\sigma^\epsilon n)$ for any constant $\epsilon > 0$ [13]. Further, this index finds the suffix array interval of P in almost RAM-optimal time, $O(m/\log_\sigma n + \log_\sigma^\epsilon n)$.

Theorem 5. *Let T be a text of length n over an alphabet of size σ. Then there is a data structure using $O(n \log \sigma)$ bits that finds the c contextual occurrences of $(P[1..m], \ell)$ in time $O(m/\log_\sigma n + (c+1)\log_\sigma^\epsilon n)$, for any constant $\epsilon > 0$.*

4 Conclusions

We have proposed a query that should be more meaningful than standard pattern locating in the case of highly repetitive text collections. Instead of simply locating all the positions of $T[1..n]$ where $P[1..m]$ appears, we give a context length ℓ and ask for the occurrences of all the c distinct strings XPY in the text, for any X, Y where $|X| = |Y| = \ell$. If P occurs inside a highly repeated substring, many essentially identical occurrences will be reported one by one with the standard locating, whereas we will report only a single suffix array range comprising all the occurrences of the same context XPY.

While the query can be solved in $O(n)$ space and RAM-optimal $O(m/\log_\sigma n + c)$ time, we focus on using space proportional to the repetitiveness of T. We use one such measure, the number $r(S)$ of equal-letter runs of the Burrows-Wheeler Transform of the string S. Within space $O(\overline{r} \log(n/\overline{r}))$, where $\overline{r} = \max(r(T), r(T^{rev}))$, we solve the problem in time $O(m \log \log n + occ \log n)$. We also show how to adapt our general strategy to any compressed text index.

This is a first step towards studying queries that make more sense on highly repetitive text collections, possibly deviating from the classical ones used for regular collections. Some relevant remaining questions are: Can the obtained space/time tradeoffs be improved? Are there other relevant and challenging queries that are better suited to highly repetitive text collections?

References

1. Belazzougui, D., Cunial, F.: Representing the suffix tree with the CDAWG. In: Proc. 28th CPM. pp. 7:1–7:13 (2017)
2. Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., Raffinot, M.: Composite repetition-aware data structures. In: Proc. 26th CPM. pp. 26–39 (2015)
3. Belazzougui, D., Navarro, G.: Alphabet-independent compressed text indexing. ACM Transactions on Algorithms **10**(4), article 23 (2014)

4. Bille, P., Ettienne, M.B., Gørtz, I.L., Vildhøj, H.W.: Time-space trade-offs for Lempel-Ziv compressed indexing. Theoretical Computer Science **713**, 66–77 (2018)
5. Blumer, A., Blumer, J., Haussler, D., McConnell, R.M., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. Journal of the ACM **34**(3), 578–595 (1987)
6. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
7. Christiansen, A.R., Ettienne, M.B., Kociumaka, T., Navarro, G., Prezza, N.: Optimal-time dictionary-compressed indexes. CoRR **1811.12779** (2019)
8. Claude, F., Navarro, G.: Improved grammar-based compressed indexes. In: Proc. 19th SPIRE. pp. 180–192 (2012)
9. Ferrada, H., Kempa, D., Puglisi, S.J.: Hybrid indexing revisited. In: Proc. 20th ALENEX. pp. 1–8 (2018)
10. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing **40**(2), 465–492 (2011)
11. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: LZ77-based self-indexing with faster pattern matching. In: Proc. 11th LATIN. pp. 731–742 (2014)
12. Gagie, T., Navarro, G., Prezza, N.: Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. Journal of the ACM **67**(1), article 2 (2020)
13. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing **35**(2), 378–407 (2006)
14. Kempa, D., Kociumaka, T.: Resolution of the Burrows-Wheeler Transform conjecture. CoRR **1910.10631** (2019)
15. Kempa, D., Prezza, N.: At the roots of dictionary compression: String attractors. In: Proc. 50th STOC. pp. 827–840 (2018)
16. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theoretical Computer Science **483**, 115–133 (2013)
17. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology **17**(3), 281–308 (2010)
18. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing **22**(5), 935–948 (1993)
19. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM **48**(3), 407–430 (2001)
20. Navarro, G.: Compact Data Structures – A practical approach. Cambridge University Press (2016)
21. Navarro, G.: Indexing highly repetitive string collections. CoRR **abs/2004.02781** (2020)
22. Navarro, G., Nekrich, Y.: Time-optimal top-k document retrieval. SIAM Journal on Computing **46**(1), 89–113 (2017)
23. Navarro, G., Prezza, N.: Universal compressed text indexing. Theoretical Computer Science **762**, 41–50 (2019)
24. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems **41**(4), 589–607 (2007)
25. Takabatake, Y., Tabei, Y., Sakamoto, H.: Improved ESP-index: A practical self-index for highly repetitive texts. In: Proc. 13th SEA. pp. 338–350 (2014)
26. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. 14th FOCS. pp. 1–11 (1973)