Frequency-Constrained Substring Complexity

Solon P. Pissis[1,2], Michael Shekelyan[3], Chang Liu[4], and Grigorios Loukides[5]

[1] CWI, Amsterdam, the Netherlands
[2] Vrije Universiteit, Amsterdam, the Netherlands
solon.pissis@cwi.nl
[3] Queen Mary University of London, London, UK
m.shekelyan@qmul.ac.uk
[4] Zhejiang University, Medical Center, Zhejiang, China
0623541@zju.edu.cn
[5] King's College London, London, UK
grigorios.loukides@kcl.ac.uk

Abstract. We introduce the notion of frequency-constrained substring complexity. For any finite string, it counts the distinct substrings of the string per length *and* frequency class. For a string x of length n and a partition of $[n]$ in τ intervals, $\mathcal{I} = I_1, \ldots, I_\tau$, the *frequency-constrained substring complexity* of x is the function $f_{x,\mathcal{I}}(i,j)$ that maps i,j to the number of distinct substrings of length i of x occurring at least α_j and at most β_j times in x, where $I_j = [\alpha_j, \beta_j]$. We extend this notion as follows. For a string x, a dictionary $\mathcal{D}$ of d strings (documents), and a partition of $[d]$ in τ intervals $I_1, \ldots, I_\tau$, we define a 2D array $S = S[1\,..\,|x|, 1\,..\,\tau]$ as follows: $S[i,j]$ is the number of distinct substrings of length i of x occurring in at least α_j and at most β_j documents, where $I_j = [\alpha_j, \beta_j]$. Array S can thus be seen as the distribution of the substring complexity of x into τ document frequency classes. We show that after a *linear-time* preprocessing of $\mathcal{D}$, for any x and any partition of $[d]$ in τ intervals given online, array S can be computed in near-optimal $\mathcal{O}(|x|\tau \log\log d)$ time.

Keywords: Substring complexity · Suffix tree · Predecessor search.

1 Introduction

The *substring complexity* or *subword complexity* of an infinite string x is the function that maps i to the number of distinct substrings (subwords) of length i in x. Substring complexity is one of the main topics in combinatorics on words [22]. The ultimate goal is to find explicit formulas for (or estimates of) the number of distinct fragments of length i occurring in a given infinite string [17, 9]. Substring complexity in finite strings plays also a crucial role in data compression [21]; it underlies a promising compressibility measure for repetitive sequences [13, 14].

We introduce the notion of frequency-constrained substring complexity of finite strings. For any finite string, it counts the distinct substrings of the string per length *and* frequency class. For a string x of length n and a partition of $[n]$ [6]

[6] By the notation $[u]$ we denote $\{1, 2, \ldots, u\}$.

in τ intervals $\mathcal{I} = I_1, \ldots, I_\tau$, the *frequency-constrained substring complexity* of x is the function $f_{x,\mathcal{I}}(i,j)$ that maps i,j to the number of distinct substrings of length i of x occurring at least α_j and at most β_j times in x, where $I_j = [\alpha_j, \beta_j]$. We extend this notion as follows. For a string x, a dictionary $\mathcal{D}$ of d strings (documents) and a partition of $[d]$ in τ intervals $\mathcal{I} = I_1, \ldots, I_\tau$, the function $f_{x,\mathcal{D},\mathcal{I}}(i,j)$ maps i,j to the number of distinct substrings of length i of x occurring in at least α_j and at most β_j documents in $\mathcal{D}$, where $I_j = [\alpha_j, \beta_j]$. In fact, computing $f_{x,\mathcal{D},\mathcal{I}}$ efficiently is the main problem we consider in this paper.

The frequency-constrained substring complexity of x is very descriptive as it provides subtle information about the substrings of x. It can thus help us tune string processing algorithms by setting bounds on the substrings length or on frequency; for example, when $\tau = 2$, the substrings of x are classified into frequent and infrequent [19]. We can also tune the output size of a document retrieval algorithm [20], the term's length used by a `tf-idf` algorithm [15], or the seed length used by seed-and-extend sequence alignment algorithms [5, 16].

Example 1. Let $\mathcal{D} = \{$`a,ananan,baba,ban,banna,nana`$\}$. For $x =$ `banana` and $I_1 = [1,2], I_2 = [3,4], I_3 = [5,6]$, we have $f_{x,\mathcal{D},\mathcal{I}}(2,2) = 3$: `ba` occurs in $3 \in I_2$ documents; `an` occurs in $4 \in I_2$ documents; and `na` occurs in $3 \in I_2$ documents.

Our contribution. Let S be a 2D array such that $S[i,j] = f_{x,\mathcal{D},\mathcal{I}}(i,j)$. We show that after a *linear-time* preprocessing of $\mathcal{D}$, for any x and any partition $\mathcal{I}$ of $[d]$ in τ intervals given online, array S can be computed in near-optimal $\mathcal{O}(|x|\tau \log\log d)$ time. Since array S is of size $|x| \times \tau$, our data structure is nearly-optimal with respect to the preprocessing and query times. The main ingredients of our data structure are suffix trees [23, 7, 4] and predecessor search [6, 18].

2 The Data Structure

Let us denote by $\mathcal{D} = \{y_1, \ldots, y_d\}$ the input dictionary consisting of $d = |\mathcal{D}|$ strings (documents). We assume that all strings in $\mathcal{D}$ are over an integer alphabet Σ of size $\sigma \leq ||\mathcal{D}||^{\mathcal{O}(1)}$, where $||\mathcal{D}||$ is the total length of all the strings in $\mathcal{D}$.

Let us denote by $y = y_1\$_1 \ldots y_d\$_d$ the concatenation of the d documents in $\mathcal{D}$ in some arbitrary but fixed order; the $\$_i$ letters, $i \in [1,d]$, are unique letters not from Σ. We construct the suffix tree $\mathsf{ST}(y)$ of y (with suffix links) in linear time [7]. We implement $\mathcal{O}(1)$-time transitions in the suffix tree in linear time using perfect hashing [10]. For any string w, we define its *document frequency* in $\mathcal{D}$ as the number of distinct documents in $\mathcal{D}$ in which w has at least one occurrence. We decorate each node u of $\mathsf{ST}(y)$ with the document frequency of the string spelled from the root of $\mathsf{ST}(y)$ to u. This is done in linear time [12].

Upon a query string x, we construct the suffix tree $\mathsf{ST}(x)$ of x in $\mathcal{O}(|x|)$ time [7]: if any letter of x is not in $\mathcal{D}$, which is checked using $\mathsf{ST}(y)$, we replace it with a unique letter not in Σ, and hash the letters of x into the range $[0, |x|]$ [10]. We first show how to compute, for each node u of $\mathsf{ST}(x)$, the document frequency of the string spelled from the root of $\mathsf{ST}(x)$ to u in $\mathcal{O}(|x|)$ total time.

We perform a DFS on $\mathsf{ST}(x)$. Every leaf in a standard suffix tree is labeled with the starting position of the suffix it represents. While traversing $\mathsf{ST}(x)$, we propagate upwards the labels of the leaf nodes maintaining only the *smallest* label (starting position) in every node. For any $\mathsf{ST}(\cdot)$, we denote the smallest label i for node u by $\mathsf{start}(u) = i$. Consider now a node u of $\mathsf{ST}(x)$ which stores label $\mathsf{start}(u)$. Then the path from the root to u spells the string $x[\mathsf{start}(u)\,..\,\mathsf{start}(u) + d(u) - 1]$, where $d(u)$ is the string depth of node u. At the end of the DFS, we group the nodes per label i, for all $i \in [1, |x|]$, using radix sort. Specifically, two nodes u, v of $\mathsf{ST}(x)$ are in group G_i if and only if $\mathsf{start}(u) = \mathsf{start}(v) = i$. By construction (i.e., by choosing the smallest label) one node represents a prefix of the other node. The whole process takes $\mathcal{O}(|x|)$ time.

We run the *matching statistics* algorithm [3, 11] using x and $\mathsf{ST}(y)$: for each starting position i in x, we compute the longest match of length $\ell_i \geq 0$ in any document in $\mathcal{D}$. In particular, this algorithm gives us a locus on $\mathsf{ST}(y)$, which represents the longest match $x[i\,..\,i + \ell_i - 1]$, for all $i \in [1, |x|]$. More formally, a *locus* in a suffix tree is a pair (v, ℓ_i) where $d(\mathsf{parent}(v)) < \ell_i \leq d(v)$, for some node v of the suffix tree and some string depth ℓ_i. Provided that $\mathsf{ST}(y)$ is already constructed, computing the matching statistics takes $\mathcal{O}(|x|)$ time [11].

Let this locus on $\mathsf{ST}(y)$ be (v, ℓ_i) and let it represent $y[\mathsf{start}(v)\,..\,\mathsf{start}(v) + \ell_i - 1]$. In particular, substring $x[i\,..\,i + \ell_i - 1] = y[\mathsf{start}(v)\,..\,\mathsf{start}(v) + \ell_i - 1]$ is precisely this longest match. We consider G_i: the group of nodes from $\mathsf{ST}(x)$ having label i. Say we are processing such a node $u \in G_i$. We have two cases:

- If $\ell_i < d(u)$ the frequency assigned to node u is 0. This is correct because $x[\mathsf{start}(u)\,..\,\mathsf{start}(u)+d(u)-1]$ does not occur in any document in $\mathcal{D}$ otherwise a longer than the longest match would be output by the matching statistics.
- If $\ell_i \geq d(u)$ then we ask a weighted ancestor query [8] to locate the substring $y[\mathsf{start}(v)\,..\,\mathsf{start}(v)+d(u)-1]$ of y, and it gives us a locus $(w, d(u))$ in constant time after a linear-time preprocessing of y [2]. More formally, the *weighted ancestor* problem on suffix trees is defined as follows: given $\mathsf{ST}(y)$, we are asked to preprocess it so that can find the locus of any substring $y[p\,..\,q]$ of y on $\mathsf{ST}(y)$. We read the frequency stored at node w, and this is precisely the frequency we assign to node u. This is correct, because $x[\mathsf{start}(u)\,..\,\mathsf{start}(u)+d(u)-1]$ is a prefix of $x[\mathsf{start}(u)\,..\,\mathsf{start}(u)+\ell_i-1]$, by $\ell_i \geq d(u)$, and because $x[\mathsf{start}(u)\,..\,\mathsf{start}(u) + d(u) - 1] = y[\mathsf{start}(v)\,..\,\mathsf{start}(v) + d(u) - 1]$.

Since the matching statistics algorithm finds a locus (v, ℓ_i) for every starting position i of x, we can assign the correct document frequency to every node of $\mathsf{ST}(x)$ in $\mathcal{O}(|x|)$ total time. We obtain the following result, which we refine next.

Lemma 1. *The document frequency for all nodes of $\mathsf{ST}(x)$ can be computed in the optimal $\mathcal{O}(|x|)$ time after a linear-time preprocessing of dictionary $\mathcal{D}$.*

Let us now describe in detail how we can efficiently compute array S. The first step is to construct $\mathsf{ST}(x)$ and compute for all of its nodes the document frequency using Lemma 1. This takes $\mathcal{O}(|x|)$ time after a linear-time preprocessing of dictionary $\mathcal{D}$. Up to this point, we have correctly identified the document

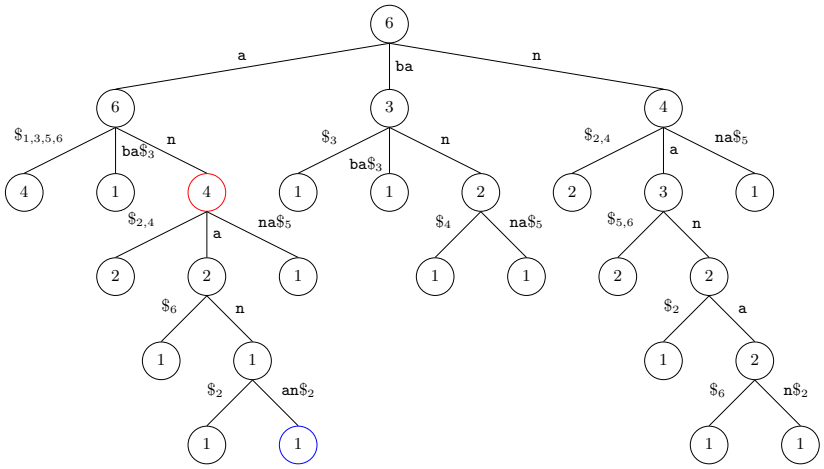

Fig. 1. $\mathsf{ST}(y)$ with nodes weighted by *document frequency* for $\mathcal{D} = \{\texttt{a,ananan,baba,ban,banna,nana}\}$ from Example 1. A successor weighted ancestor query of the blue node with argument $\alpha_j = 3$, takes us to the ancestor of the blue node with the smallest frequency at least 3. This is the red node. Any such query can be answered in $\mathcal{O}(\log\log d)$ time after a linear-time preprocessing of $\mathsf{ST}(y)$ [8, 1].

frequency for every substring of x that is spelled from the root of $\mathsf{ST}(x)$ ending exactly at some node of $\mathsf{ST}(x)$. However, we have no access to the document frequency of the substrings of x that *end in the middle of an edge* of $\mathsf{ST}(x)$.

We thus need to have an efficient way to subdivide the edges of $\mathsf{ST}(x)$ accordingly. The crucial observation is that we have only τ frequency intervals $\mathcal{I} = I_1, \ldots, I_\tau$, and thus it suffices to split every edge of $\mathsf{ST}(x)$ in at most τ subedges. To achieve this, we also *preprocess* $\mathsf{ST}(y)$ for successor weighted ancestor queries with respect to document frequency as node weights. This is possible because of the *max-heap property*: any node on $\mathsf{ST}(y)$ has equal or smaller weight than any of its ancestors. Recall that for any node u in $\mathsf{ST}(x)$ we can find the corresponding locus $(w, d(u))$ in $\mathsf{ST}(y)$ (see Lemma 1) in $\mathcal{O}(|x|)$ total time. In the second step, we enhance $\mathsf{ST}(x)$ with at most τ new nodes per edge using τ weighted ancestor queries on $\mathsf{ST}(y)$. In particular, we ask one weighted ancestor query α_j per interval $I_j = [\alpha_j, \beta_j]$ (see Figure 1). Each new node stores a document frequency and it takes $\mathcal{O}(\log\log d)$ time to find its locus on $\mathsf{ST}(y)$ using a weighted ancestor query, after a linear-time preprocessing of $\mathsf{ST}(y)$ [8, 1].

The third step is to traverse the enhanced $\mathsf{ST}(x)$ and construct a collection of labeled length intervals $[i, j]_f$, one for each node u of $\mathsf{ST}(x)$, defined as follows: $i = d(\mathsf{parent}(u)) + 1$, $j = d(u)$, and f is the document frequency stored in u. We do this in $\mathcal{O}(\tau|x|)$ total time because we have $\mathcal{O}(\tau|x|)$ nodes in $\mathsf{ST}(x)$. We ignore labeled length intervals with $f = 0$ (no occurrence) or $j = 0$ (empty string).

The fourth step is to sort these intervals and view each interval, say of node u, as a line starting at point (i, u) and ending at point (j, u) on the $[0, |x|] \times [0, 2\tau|x|]$ plane. The y-axis represents the distinct lines (we have no more than $2\tau|x|$

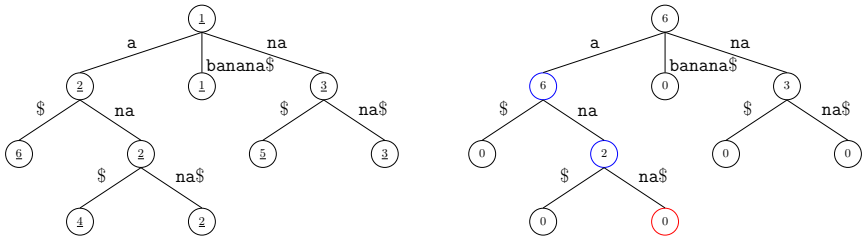

Fig. 2. Step 1 from Example 2. We assume that x ends with a unique letter $\$ \notin \Sigma$.

intervals because we have no more than $2\tau|x|$ nodes), and the x-axis represents the lengths of substrings x (the maximum length is $|x|$). We can do this in $\mathcal{O}(\tau|x|)$ time using radix sort because for any interval $[i,j]_f$, $i,j \in [|x|]$.

In the last step, for each length in $[|x|]$, we count how many lines it stabs after classifying the lines in frequency intervals. For the latter, we employ predecessor/successor search after $\mathcal{O}(d)$-time and space preprocessing [6]: we insert the endpoints of every interval in $\mathcal{O}(\tau \log\log d)$ total time as we have 2τ endpoints in total. A line with frequency f belongs to the frequency interval $I_j = [\alpha_j, \beta_j]$ if and only if the predecessor of f is α_j and its successor is β_j. The search takes $\mathcal{O}(\log\log d)$ time per line [6]. The endpoints are then deleted from the structure in $\mathcal{O}(\tau \log\log d)$ total time [6]. We sweep through the lines from left to right and maintain counters on the sum of currently "active" lines per frequency interval. We do this in $\mathcal{O}(\tau)$ time per length. We have arrived at the following result.

Theorem 1 (Main Result). *After a linear-time preprocessing of a dictionary $\mathcal{D}$ of d strings, for any query string x and any partition $\mathcal{I}$ of $[d]$ in τ intervals, array S, such that $S[i,j] = f_{x,\mathcal{D},\mathcal{I}}(i,j)$, can be computed in $\mathcal{O}(|x|\tau \log\log d)$ time.*

Example 2. Consider the dictionary $\mathcal{D}$, the query string $x = x[1\,..\,|x|] = \texttt{banana}$, and the partition $\mathcal{I}$ from Example 1. We show in Figure 2 (on the left) the suffix tree $\mathsf{ST}(x)$ and the smallest label $\mathsf{start}(u)$ (underlined) for every node u after the DFS. We have three groups of nodes: $G_1 = \{\texttt{-banana}\}$, $G_2 = \{\texttt{-a,-ana,-anana}\}$ and $G_3 = \{\texttt{-na,-nana}\}$. (Here we use the notation `-` before a string to denote a node.) From the matching statistics algorithm, we know the longest match for each position i of x: $\mathcal{L} = [3,5,4,3,2,1]$; e.g., $\ell_1 = \mathcal{L}[1] = 3$ tells us that the longest match of $x[1\,..\,6] = \texttt{banana}$ in $\mathcal{D}$ is $x[1\,..\,3] = \texttt{ban}$. For the only node in G_1 we have frequency 0 since $3 < |\texttt{banana}|$. For G_2, we first find the document frequency for the deepest node `-anana` and we reach the node `-ananan` in $\mathsf{ST}(y)$ (see Figure 1). Then from this node (`-ananan`) we ask for depths 3 and 1 (using weighted ancestor queries), and get to nodes `-ana` and `-a` in $\mathsf{ST}(y)$, which give the corresponding document frequencies in $\mathsf{ST}(x)$. Similarly we process G_3 and get the $\mathsf{ST}(x)$ in Figure 2 (on the right) with document frequencies (Lemma 1).

We next show in Figure 3 how we enhance $\mathsf{ST}(x)$ (on the left) with at most τ nodes per edge (on the right) using weighted ancestor queries on $\mathsf{ST}(y)$. Let us consider the edge `-banana`, for which we need to add new nodes. First, we

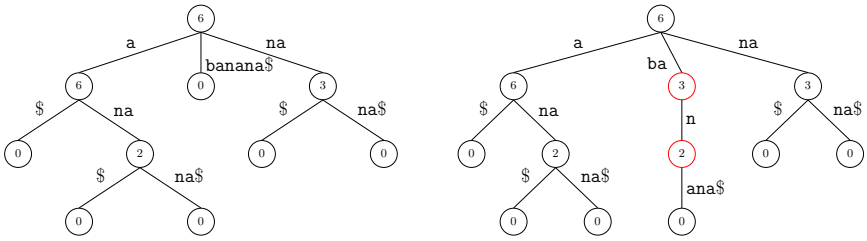

Fig. 3. In Step 2 from Example 2 the edge `-banana` is subdivided to `-ba-n-ana`.

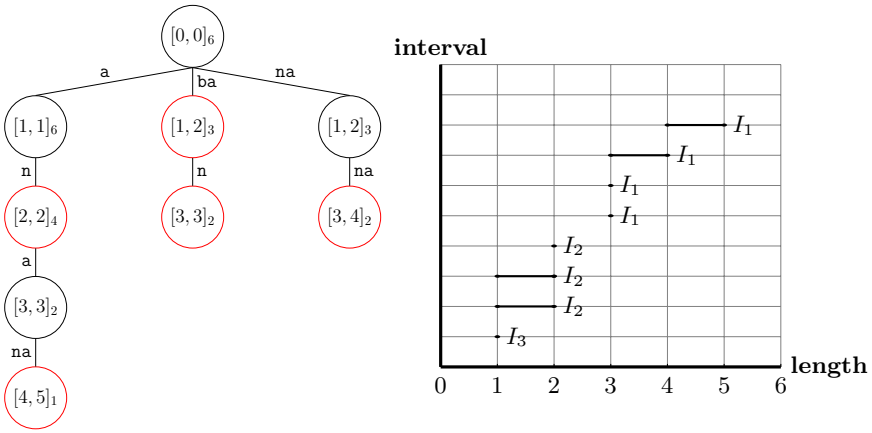

Fig. 4. Steps 3-4 from Example 2. On the left: the added nodes are in red; the nodes with $f = 0$ are pruned. On the right: the length intervals labeled by frequency interval.

have $\mathcal{L}[1] = 3$. We add a node at depth 3 and separate `-banana` to `-ban-ana`. From $\mathsf{ST}(y)$, we know the frequency of node `-ban` is 2. Then we ask whether node `-ban` in $\mathsf{ST}(y)$ has an ancestor node with a frequency at least 3 (I_2) or at least 5 (I_3). Indeed we find that node `-ba` in $\mathsf{ST}(y)$ has frequency 3, and so we add a node in $\mathsf{ST}(x)$ subdividing `-ban` to `-ba-n`. No ancestor of `-ban` in $\mathsf{ST}(y)$ has a frequency of at least 5, so we do not need to add any more nodes in $\mathsf{ST}(x)$.

After the end of the second step, we construct a labeled length interval for each node of the enhanced $\mathsf{ST}(x)$, and so we get the tree in Figure 4 (on the left). In the fourth step, we sort these length intervals and view them as lines (on the right). In the last step, after classifying the lines in frequency intervals, we sweep through them from left to right, and compute array S.

S	[1,2]	[3,4]	[5,6]
1	0	2	1
2	0	3	0
3	3	0	0
4	2	0	0
5	1	0	0
6	0	0	0

References

1. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. ACM Trans. Algorithms **3**(2), 19 (2007). https://doi.org/10.1145/1240233.1240242, https://doi.org/10.1145/1240233.1240242
2. Belazzougui, D., Kosolobov, D., Puglisi, S.J., Raman, R.: Weighted ancestors in suffix trees revisited. In: Gawrychowski, P., Starikovskaya, T. (eds.) 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland. LIPIcs, vol. 191, pp. 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CPM.2021.8, https://doi.org/10.4230/LIPIcs.CPM.2021.8
3. Chang, W.I., Lawler, E.L.: Sublinear approximate string matching and biological applications. Algorithmica **12**(4/5), 327–344 (1994). https://doi.org/10.1007/BF01185431, https://doi.org/10.1007/BF01185431
4. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press (2007)
5. Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., Salzberg, S.L.: Alignment of whole genomes. Nucleic Acids Research **27**(11), 2369–2376 (01 1999). https://doi.org/10.1093/nar/27.11.2369, https://doi.org/10.1093/nar/27.11.2369
6. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett. **6**(3), 80–82 (1977). https://doi.org/10.1016/0020-0190(77)90031-X, https://doi.org/10.1016/0020-0190(77)90031-X
7. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997. pp. 137–143. IEEE Computer Society (1997). https://doi.org/10.1109/SFCS.1997.646102, https://doi.org/10.1109/SFCS.1997.646102
8. Farach, M., Muthukrishnan, S.: Perfect hashing for strings: Formalization and algorithms. In: Hirschberg, D.S., Myers, E.W. (eds.) Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1075, pp. 130–140. Springer (1996). https://doi.org/10.1007/3-540-61258-0_11, https://doi.org/10.1007/3-540-61258-0_11
9. Ferenczi, S.: Complexity of sequences and dynamical systems. Discret. Math. **206**(1-3), 145–154 (1999). https://doi.org/10.1016/S0012-365X(98)00400-2, https://doi.org/10.1016/S0012-365X(98)00400-2
10. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. J. ACM **31**(3), 538–544 (1984). https://doi.org/10.1145/828.1884, https://doi.org/10.1145/828.1884
11. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997). https://doi.org/10.1017/cbo9780511574931, https://doi.org/10.1017/cbo9780511574931
12. Hui, L.C.K.: Color set size problem with application to string matching. In: Apostolico, A., Crochemore, M., Galil, Z., Manber, U. (eds.) Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings. Lecture Notes in Computer Science, vol. 644, pp. 230–243. Springer (1992). https://doi.org/10.1007/3-540-56024-6_19, https://doi.org/10.1007/3-540-56024-6_19

13. Kociumaka, T., Navarro, G., Prezza, N.: Toward a definitive compressibility measure for repetitive sequences. IEEE Trans. Inf. Theory **69**(4), 2074–2092 (2023). https://doi.org/10.1109/TIT.2022.3224382, https://doi.org/10.1109/TIT.2022.3224382
14. Kutsukake, K., Matsumoto, T., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: On repetitiveness measures of Thue-Morse words. In: Boucher, C., Thankachan, S.V. (eds.) String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12303, pp. 213–220. Springer (2020). https://doi.org/10.1007/978-3-030-59212-7_15, https://doi.org/10.1007/978-3-030-59212-7_15
15. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, 2nd Ed. Cambridge University Press (2014), http://www.mmds.org/
16. Loukides, G., Pissis, S.P.: Bidirectional string anchors: A new string sampling mechanism. In: Mutzel, P., Pagh, R., Herman, G. (eds.) 29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference). LIPIcs, vol. 204, pp. 64:1–64:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ESA.2021.64, https://doi.org/10.4230/LIPIcs.ESA.2021.64
17. Mignosi, F.: Infinite words with linear subword complexity. Theor. Comput. Sci. **65**(2), 221–242 (1989). https://doi.org/10.1016/0304-3975(89)90046-7, https://doi.org/10.1016/0304-3975(89)90046-7
18. Navarro, G., Rojas-Ledesma, J.: Predecessor search. ACM Comput. Surv. **53**(5), 105:1–105:35 (2021). https://doi.org/10.1145/3409371, https://doi.org/10.1145/3409371
19. Pissis, S.P.: MoTeX-II: structured MoTif eXtraction from large-scale datasets. BMC Bioinform. **15**, 235 (2014). https://doi.org/10.1186/1471-2105-15-235, https://doi.org/10.1186/1471-2105-15-235
20. Puglisi, S.J., Zhukova, B.: Document retrieval hacks. In: Coudert, D., Natale, E. (eds.) 19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France. LIPIcs, vol. 190, pp. 12:1–12:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.SEA.2021.12, https://doi.org/10.4230/LIPIcs.SEA.2021.12
21. Raskhodnikova, S., Ron, D., Rubinfeld, R., Smith, A.D.: Sublinear algorithms for approximating string compressibility. Algorithmica **65**(3), 685–709 (2013). https://doi.org/10.1007/s00453-012-9618-6, https://doi.org/10.1007/s00453-012-9618-6
22. Shallit, J.O., Shur, A.M.: Subword complexity and power avoidance. Theor. Comput. Sci. **792**, 96–116 (2019). https://doi.org/10.1016/j.tcs.2018.09.010, https://doi.org/10.1016/j.tcs.2018.09.010
23. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973. pp. 1–11. IEEE Computer Society (1973). https://doi.org/10.1109/SWAT.1973.13, https://doi.org/10.1109/SWAT.1973.13