Full Compressed Affix Tree Representations

Rodrigo Cánovas and Eric Rivals

L.I.R.M.M. & Institut Biologie Computationnelle
Université de Montpellier, CNRS F-34392 Montpellier Cedex 5, France
canovas-ba@lirmm.fr, rivals@lirmm.fr

Abstract

The Suffix Tree, a crucial and versatile data structure for string analysis of large texts, is often used in pattern matching and in bioinformatics applications. The Affix Tree generalizes the Suffix Tree in that it supports full tree functionalities in both search directions. The bottleneck of Affix Trees is their space requirement for storing the data structure. Here, we discuss existing representations and classify them into two categories: *Synchronous* and *Asynchronous*. We design Compressed Affix Tree indexes in both categories and explored how to support all tree operations bidirectionally. This work compares alternative approaches for compressing the Affix Tree, measuring their space and time trade-offs for different operations. Moreover, to our knowledge, this is the first work that compares all Compressed Affix Tree implementations offering a practical benchmark for this structure.

Introduction

Stringology offers many indexes for unidirectionally searching a pattern in a text, but recently the need for bidirectional search tools arose for instance in genomics. An example is the search of hairpin like structures that consist of a *loop motif* (a short core sequence) and a *tail* (such that on each side of the loop motif is one half of a palindromic string) [1]. In this search, when a candidate loop is found, one can extend the search with each branch of the hairpin. The search of such motifs has triggered the development of bidirectional indexes: first the Affix Tree [2, 3], then the Affix Array [4], and more recently the Bidirectional Wavelet Tree [5]. In all these approaches the procedure can switch the search direction at any time during the search, which is needed when the whole pattern is not a single word given at start time, but can shrink or extend in piecewise during the search. Also, recently Affix Trees have been mentioned as a way to represent complex graphs encoding string overlaps, for example, the Hierarchical Overlap Graph [6].

Here, we introduce a classification of bidirectional indexes based on how they are traversed, *Asynchronous* and *Synchronous*. We propose new Compressed Affix Tree indexes in both categories: the *Asynchronous* Compressed Affix Tree (ACAT) and the *Synchronous* Compressed Affix Tree (SCAT), and explain their practical implementations. Moreover, we exhibit both a reduced version of the index of [7], and a sampled version of ACAT. To evaluate the potential of these approaches and to compare them with different indexes for bidirectional search, we have implemented the Affix Array [4] (given that the original was not available), and compared seven indexing structures in practice. This yields the first large and practical overview of bidirectional search indexes in term of memory usage, and of operation times. Last, all the implementations designed for this article can be found at `https://github.com/rcanovas`.

2375-0359/17 $31.00 © 2017 IEEE
DOI 10.1109/DCC.2017.39

Operation	Description
$Root()$	Root of the Suffix Tree.
$Depth(v)$	String-depth of v.
$Parent(v)$	Parent node of v.
$SLink(v)$	Suffix-link of v; the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma$.
$WLink(v, a)$	Weiner-link of v given a; the node w s.th. $\pi(w) = a\beta$ if $\pi(v) = \beta$.
$Child(v, a)$	The node w s.th. the first letter on edge (v, w) is $a \in \Sigma$.
$Children(v)$	The list of nodes w such that w is a child of v.
$Degree(v)$	Number of children of v.

Table 1: Example of navigational operations over the nodes and leaves of a suffix tree.

Basic Concepts

Let Σ denote a finite alphabet of cardinality σ and T be a sequence of characters from Σ. The length of T is denoted $|T|$ and let $n := |T|$. The characters of T are indexed from 1 to n, so that the i-th character of T is T_i or $T[i]$. A substring of T is denoted $T_{i,j} = T_i T_{i+1} \dots T_j$. A prefix of T is a substring of the form $T_{1,j}$, and a suffix is a substring of the form $T_{i,n}$.

Suffix Tree Let $P = \{S^1, \cdots, S^N\}$ be a set of N strings. A *Trie* [8, 9] is a tree where each edge is associated with an element of Σ and each node represents a distinct prefix in the set P. This structure supports searching for a string in the set in time proportional to the length of the sought string (independently of the set size).

The *Suffix Trie* of a text T is a *Trie* built over the set of all the suffixes of T (assuming that T is terminated by a special element "\$"). The *Suffix Tree* (ST) [10, 11] of a text T is a *Suffix Trie* where each unary path is converted into a single edge. The leaf of the Suffix Tree indicates the text position where the corresponding suffix starts. Table 1 lists some of the operations used to navigate over the nodes and leaves of the Suffix Tree and introduces a notation for these.

Various *Compressed Suffix Tree* (CST) [12–15] implementations offering different space vs time trade-off are available.

Suffix Array A Suffix Array (SA) [16] is a permutation of the interval $[1, n]$ such that $T_{SA[i],n} <_L T_{SA[i+1],n}$ for all $1 \leq i < n$, where $<_L$ denotes lexicographically smaller than. Several implementations of *Compressed Suffix Arrays* have been designed [17, 18]. Each provides three main operations: 1/ given i extract the value of $SA[i]$, 2/ given a position j in the text T, fetch its position in the Suffix Array ($SA^{-1}[j]$), and 3/ find in $SA(T)$ the interval of the suffixes starting with a given pattern.

The Suffix Array and the Suffix Tree of a text are connected. When the children of each node of $ST(T)$ are ordered lexicographically by their string label, $SA(T)$ contains the positions of the Suffix Tree leaves visited in left-to-right order. Moreover, every internal node v of the $ST(T)$ can be represented as an interval $[i, j]$ of $SA(T)$ whose suffixes share a prefix corresponding to node v.

Generally the Suffix Array is enhanced with the Longest Common Prefix (LCP) array, which allows solving many string processing problems in optimal time and space. Conceptually, the LCP array defines the shape of the Suffix Tree and thus

allows any traversal of $ST(T)$ to be simulated using $SA(T)$.

Burrows-Wheeler Transform (BWT) For a text T of length n, the BWT (T^{bwt}) [19] is the last column of an $n \times n$ matrix, denoted M_T, whose rows are the rotations (or cyclic shifts) of T sorted in lexicographic order. Equivalently, the BWT collects the character preceding each suffix of T in the order of $SA(T)$. Hence the BWT can be identified with SA. Assuming that the last symbol of T is lexicographically smaller than all the others and unique, it is possible to reverse the transformation: $T[n-1]$ is located at $T^{bwt}[1]$, since the first element of M_T starts with $T[n]$.

The LF, or *Last-to-First* mapping, function allows us to move from the character $T^{bwt}[i]$, in the last column of M_T to its position in the first column of M_T. Thus, LF allows us to navigate T backwards, $T[n-2] = T^{bwt}[LF(1)]$, and for any position k, $T[n-k] = T^{bwt}[LF^{k-1}(1)]$. In general, T^{bwt} is stored using a *wavelet tree* [18, 20]. The T^{bwt} is mainly used for pattern search using a method called *backward search* (making use of LF and a sampling of the SA), which is implemented by the family of indexes known as FM-indexes [18].

Affix Tree The *Affix Tree* was introduced and tested by Stoye [3], then further studied by Maaß [2]. To support bidirectional applications, it combines the Suffix Tree of T (denoted ST) with the Suffix Tree of the reversed text of T (denoted RT). The Affix Tree distinguishes four kinds of nodes in a Suffix Tree: *explicit* nodes, which have at least two children; *implicit* nodes, which belong to an edge between two explicit nodes; *explicit leaves*, which cannot be extend any further in the tree; and *implicit leaves*, which belong to the edge linking an explicit leaf to its parent. In the Affix Tree, each internal node (explicit and implicit) of ST is associated to a node in RT, and vice versa, by an *affix link*. The construction of an Affix Tree theoretically takes linear time and space, but in practice it "required large amount of memory (approximately $45n$ bytes) and its complex data structure decreased the algorithmic performance" [4].

A Classification of Bidirectional Indexes

A bidirectional index in general includes a data structure to search from left to right in T, and another to search from right to left. We call them respectively the Forward Structure (FoS) and the Backward Structure (BaS) and say that each is the *companion structure* of the other. In this work we consider two classifications for storing and maintaining the current state: *Asynchronous* and *Synchronous*.

In the *Asynchronous* mode, the index maintains a single *current state* (or *current node*), say the current state for FoS, and when a switch of direction is needed, it computes the corresponding state in BaS, before carrying on the search in the opposite direction. To facilitate updating the current state, the index stores in an extra table the corresponding node in BaS for each explicit node in FoS. A symmetrical table is needed to switch from BaS to FoS. Notice that under this definition, it is only possible to switch between the structures from a explicit node to an internal

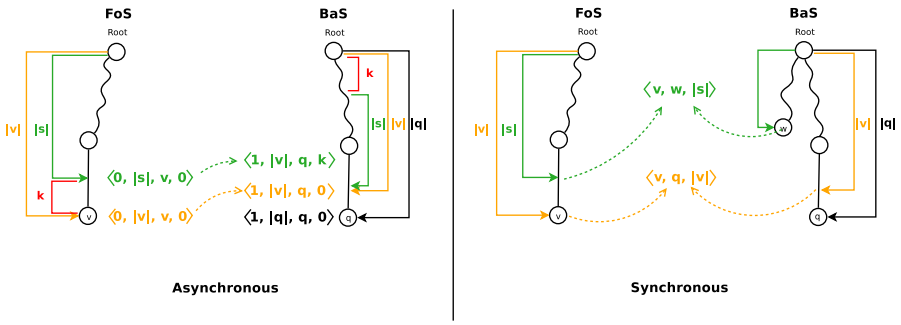


Figure 1: An *Asynchronous* state is always based on one side of the data structure being necessary to support transitions between FoS and BaS. A *Synchronous* state contains information about FoS and BaS keeping them always updated after each operation.

node. In this mode an state is defined as a quadruplet $\langle d, l, node, k\rangle$, where d is the direction of the data structure to which the (explicit) *node* with path length l belong, and k is the context length.

In the *Synchronous* mode, one maintains a pair of current states: one for FoS, one for BaS. Hence, at each modification of the matched word, both states are updated. These indexes store less information then *Asynchronous* indexes, but pay an extra time at each extension of the search. The pair of states must always be synchronized. In this case an state is defined by the triplet $\langle node_{\text{FoS}}, node_{\text{BaS}}, l\rangle$. Figure 1 shows an example of states for both categories.

Asynchronous Approaches

The first attempt to reduce the space required by Affix Trees was the *Affix Array* (AFA) proposed by Strothmann [4]. He replaced each Suffix Tree by a Suffix Array plus an LCP array. In addition, he included two *child tables* [21] to ease locating the children of any node/interval. Finally, the AFA also stores for each explicit node (of FoS and BaS) the pointer to its corresponding node in the companion structure. The arrays storing these pointers are called the *Affix Link* arrays (*ALink*). The AFA allows both fast access to the child of a node, and fast switching between FoS and BaS. Also it supports the operation *suffix-children* (*prefix-children*), which returns all possible nodes that are an extension of a node by one character to the right (left).

To compute the *ALink* arrays, Strothmann's algorithm precomputes for FoS the *SLink* (see Table 1) of all explicit nodes. Then, it traverses the explicit nodes in depth-first preorder, checks whether the *ALink* value of the current node is already stored, otherwise computes it. For a node v whose *ALink* value is unknown, it uses the formula: $ALink(v) := Child_{\text{BaS}}(ALink(SLink(v)), c)$, where c is the first character on the string path of v. Computing the *ALink* array of BaS is symmetrical.

We notice that Strothmann's algorithm for computing the *ALink* for a node v can take many recursive calls (in the worse case, a number equal to the length of the string path of v). Instead, we propose to visit the nodes in level order, such that the computation of the *ALink* for a node v takes a number of recursive steps

that is bounded by the length of the edge linking v to its parent. We experimentally observed that Strothmann's algorithm exhausted the available memory for a 50 MB text. So, in this work we compute the *ALink* array using our proposed methodology.

Strothmann also proved that for every state $\langle 0, |s|, v, 0\rangle$, its reverse associated state is given by $\langle 1, |v|, ALink(v), k = |v| - |s|\rangle$. Note that when $|s| < |v|$ (*i.e.*, s is an implicit node), the path of s is a prefix of the path of v. Thus, the path of the reverse of s is a suffix of the path of $ALink(v)$ starting at position k (see Figure 1).

While the AFA improves on the Affix Tree regarding space, it still requires a huge amount of memory and supports only operations that can be solved using the *Child* operation. Since Strothmann's approach, several compressed representations of the Suffix Tree [12–15] have been reported. We propose to upgrade the AFA by keeping the *ALink* arrays but representing FOS and BAS using CSTs. This approach, denote as ACAT (*Asynchronous* Compressed Affix Tree) uses less memory than AFA and supports all tree operations.

Moreover, given the *ALink* formula, it is clear that we don't need to store all the *ALink* values (provided that FOS and BAS support *SLink*). According a user defined parameter z, indicating the maximum distance to a sample node in number of suffix links, we store whether or not the *ALink* value of a node is sampled. To check if we need to sample the *ALink* value of a node, we just need to examine that none of the nodes traversed by doing the operation *SLink* z times is already sampled. Otherwise the node is not sampled. We name this approach the ACATS (*Asynchronous* Compressed Affix Tree Sampled). It reduces the space used by the Affix Link arrays, at the cost of longer time for switching between the structures. In the worse case, for each switch at a unsampled node, ACATS would take additional z *SLink* and *Child* operations to get the *ALink* value of that node.

The lightest scenario for ACATS is when we do not store the *ALink* arrays at all. Then, each time we want to compute the corresponding state in the companion structure, we compute it starting from the root. Albrecht and Heun [22] studied how to optimally obtain the *ALink* in this case using binary search. A faster approach was presented by Gog *et al.*[7]. As the procedure is symmetrical for a node BAS or of FOS, we explain it for the latter. They noted that if an internal explicit node v is represented as a SA interval $[l, r]$, then all the text positions $SA(k)$ (with $l \leq k \leq r$) where the string path of the current node occurs are known. This also implies that it is possible to compute all the positions where the reverse string path occurs in the reverse of T as $w_k = n - (SA(k) + Depth(v))$. With this on mind, Gog *et al.*proposed first to compute the array $A[i] := SA^{-1}_{\text{BAS}}(n - SA(i))$ for $1 \leq i \leq n$, which converts an entry of the SA of FOS into the matching entry in the SA of BAS. Then to find the reverse representation of v, it is enough to have an structure to compute the minimum and another to compute maximum values of $A[k]$ for $l \leq k \leq r$, which points to the leafs in BAS associated with these values, and to return their lower common ancestor. We refer to this approach as ACATN (ACAT Non-sampled). For more details of how the array A is stored and queried, we refer the reader to [7].

Finally we notice that the space used by ACATN could be further reduced. When we switch from one structure to another, it is enough to know only one extremity of the SA interval in the reverse structure, given that the lengths of both intervals are

equal [4, 5]. This simple fact implies that one of two structures is superfluous (say that for compute the maximum), and that it reduces the number of operations performed during a switch. We denote this reduced approach as RACATN (Reduced ACATN).

Synchronous Approaches

The *Synchronous* scheme aims at getting rid of Affix Link arrays by keeping FoS and BaS continuously synchronized. This necessarily increases the search time. For example, if in FoS one extends the current string path by one symbol to the right, then one also needs to extend the string path in BaS by the same symbol to the left.

To improve over the AfA in terms of space and still support bidirectional search, two groups have proposed in parallel to use an Affix Array like structure based on the *BWT* [5, 23]. This approach is called the *Bidirectional BWT* (BidWT for short). The underlying idea is to store FoS and BaS using data structures that support backward search based on a wavelet tree (FM-Indexes). Then backward and forward extensions of a search are both possible. Indeed, to extend a current node to the right with character c, one performs a backward search of c from the current state in BaS. For more details, we refer the reader to [5].

While the BidWT requires much less space than the Affix Array and offers fast search extension by one character, it does not support any tree operation which allow to shrink the current pattern. Moreover, the BidWT only offers search extension for a single character. This means that performing an operation like *suffix-children*, one needs to explore the possible extensions for all letters of the alphabet. A similar issue occurs for the *Child* operation, whenever the child path is longer than one symbol.

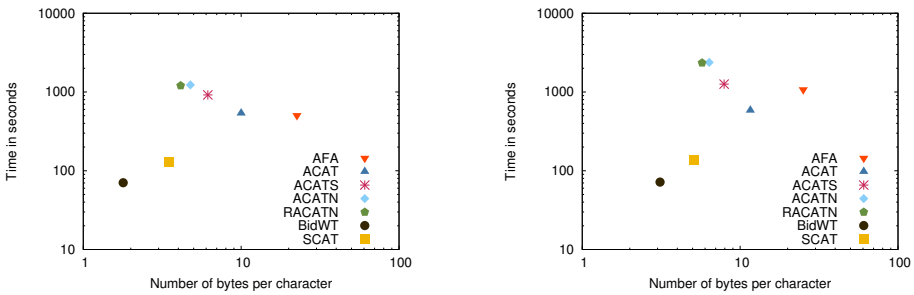
We propose to upgrade the BidWT by storing FoS and BaS as *CST* instead of FM-Indexes. We call this approach the *Synchronous* Compressed Affix Tree (SCAT). The SCAT can support all tree operations within each structure, but the issue of maintaining both structures synchronized remains. Fortunately, we noticed that each tree operation has an equivalent operation in the reverse tree. That is, *Parent-Slink* and *Child-WLink*, and all other operations can be solved using these four operations. Let us give the algorithms for computing *Parent* as an illustration.

Let $\langle node_{\text{FoS}}, node_{\text{BaS}}, l\rangle$ be the current state. To compute the *Parent* in FoS, we return $\langle Parent(node_{\text{FoS}}), SLink^p(node_{\text{BaS}}), l-p\rangle$, where p is the length of the edge between the $node_{\text{FoS}}$ and its parent.

Results

In order to compare the approaches presented in this work, we used four different input files obtained from the pizzachilli text collection[1]: DNA, PROTEINS, XML, and ENGLISH. The alphabet of size of the text tested are 16, 27, 97, and 239 respectively, and we use the 50 Megabytes version of each of them (given the high memory requirement used by AfA). The pizzachilli web-page provides a detailed descriptions of the input files. All experiments were performed on a computer with Intel(R) Xeon(R) CPU E5-2623 v3 up to 3.00 GHz, The operating system was Ubuntu 14.04.01, version

[1] `http://pizzachili.dcc.uchile.cl/texts.html`



(a) DNA (b) ENGLISH

Figure 2: Construction time versus storage space.

3.19.0-59-generic-generic Linux kernel. Finally all the approaches described here were implemented in C++11, using version 4.8.4 of the g++ compiler.

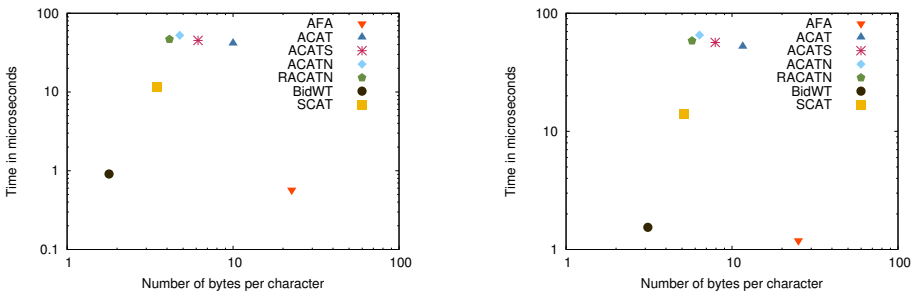
Each implementation uses one specific type of data structure to represent FoS and BaS (but the same for both). We compared the following methodologies: AFA (which we implemented following Strothmann's description [4], since we did not find an available implementation); BIDWT (a wrap implementation using the *sdsl library* [20], which includes the bidirectional backward and forward search presented in [5]); ACAT; ACATS (using as sample parameter $z = 1$); ACATN (based on the work of Gog et al. [7]); RACATN (our reduced version of ACATN); and SCAT.

In this work all the approaches that used a CST were implemented using the *sct3* version from the sdsl library [13], which offers a good space/time trade-off in practice. Even if other CST implementations could offer different trade-offs, or reduce the number of operations to run, we assume that the *sct3* implementation gives a fair methodology to compare all studied approaches.

For space limitations, we show the results of each experiment only over the DNA and ENGLISH texts, which are the ones with smallest and largest alphabet. Also in all figures of this section, the horizontal and vertical axes are in logarithmic scale.

Figure 2 shows the space used by each structure (measured in bytes per character of the original text) vs its construction time (in seconds). We observe that our AFA implementation takes over 20 bpc, which is consistent with the figures in [4]. In comparison the ACAT takes 10 bpc (less than half the AFA), which makes it scalable enough to index a Human genome (whose length is $\sim$ 3 Gb). Clearly, the *Asynchronous* indexes takes longer time and more space than *Synchronous* indexes, but most of their construction time is spent for computing the Affix Link arrays (which use $\sim$ 6.5 bpc when is completely stored) or some equivalent (the array A for ACATN and RACATN). The SCAT structure uses almost twice the space of the BIDWT, auguring well of their scalability. This comparatively higher memory usage is the price to pay for supporting all tree operations, and improving the versatility.

We tested three different operations (chosen to be supported by all indexes): *Forward-Backward* search, *suffix-children* and *prefix-children*. Among the operations that are supported by full Affix Trees (*i.e.*, not by AFA nor BIDWT), we only show experiments results for *SLink* due to space limitations (Note that the performace of tree operations can change depending on the CST structure used). For the



(a) DNA (b) ENGLISH

Figure 3: *Forward-Backward* search time vs space usage.

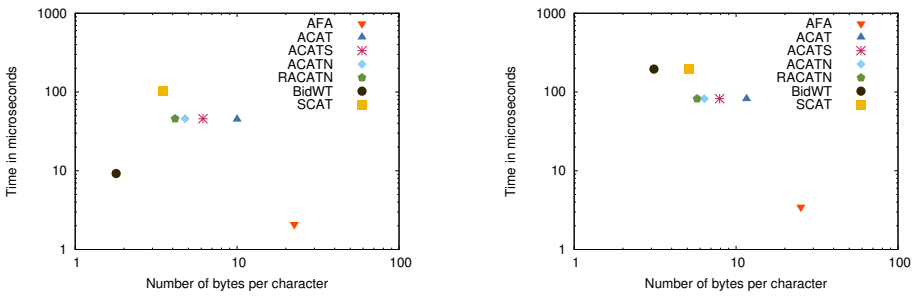
Forward-Backward search, we took samples of 1000 random sequences of length 100 from the text and searched them performing forward and backward searches alternatively (thereby switching the direction after each symbol), and repeating the experiment with 1000 samples. For the operations *suffix-children*, *prefix-children*, and *SLink* we performed each on random samples of 10000 internal nodes. In all experiments, we report the average time taken to perform one operation.

Figure 3 shows the space used by the indexes (in bpc) versus the time (in microseconds) obtained for *Forward-Backward* search. The only competitive solutions against BidWT, are the AFA and SCAT approaches. While AFA offers a faster performance but required huge amounts of space, SCAT, which is almost 10 times slower, supports all tree operations. Note that the solutions containing a representation of the *CST* could instead incorporate a FM-index (like the one used by the BidWT). In this case, it would be possible to re-implement the operation *Forward-Backward* search to obtain the same performance than BidWT.

Given that the performance for *suffix-children* and *prefix-children* are similar, we only show the result for *suffix-children* in Figure 4. As for the *Forward-Backward* search, AFA approach displayed the best time performance, since almost all required information by these queries is explicitly stored. We also notice that with BidWT the query time for these two operations depends on the alphabet size of the input text, while SCAT depends on the *CST* structure chosen. The differences in time for the these operations between the *Asynchronous* approaches are due to the way that each index switches between FoS and BaS. So comparatively, a direct computation of *Children* is an advantage only with larger alphabets, when checking all possible symbols becomes expensive for the BidWT.

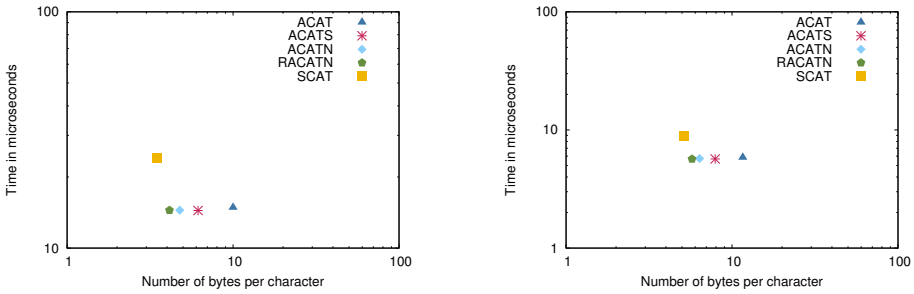
Figure 5 shows the average time to perform the *SLink* operation (which is not supported by BidWT and AFA). While the *Asynchronous* approaches needs only one operation to return the *SLink*, SCAT always had to perform two operations, which explains the longer average times obtained with SCAT than with the other methodologies. We found this behavior repeated when tested other tree operations.

Summarizing, while BidWT uses less space and offers faster backward/forward search of extension of one character, it doesn't supports all the tree operations and is sensitive to the alphabet size when computing operations related with child. The AFA approach offers generally fastest operation times, but requires too much space,



(a) DNA (b) ENGLISH

Figure 4: Time of operation *suffix-children* vs space usage.



(a) DNA (b) ENGLISH

Figure 5: Time of operation *Slink* vs space usage.

and also miss some tree operations. When full affix tree operations are required, SCAT uses less space, while being competitive with the *Asynchronous* approaches. We prove that RACATN improve both in space and time over ACATN. We show that ACATS with a sample parameter of 1 ($z := 1$) uses $\approx 70\%$ of the space of ACAT, while offering similar time performances. Moreover, although increasing z would reduce the space of ACATS, SCAT gives a lower bound for space and achieves almost equivalent running times (compared to ACATS with $z = 1$).

Conclusions

We investigated the design of compressed versions of Affix Trees proposing several new structures and introduced a way to classify these approaches (*Synchronous* or *Asynchronous*), offering a first benchmark of Affix Tree compressed data structures.

As future work, it will be interesting to see how the structures presented in this paper can be used for pattern search with errors, where it is necessary to be able to extend and shrink the pattern during the search. Also, new approaches for bidirectional search will keep emerging (for example [24]), being of great interest to explore how these new solutions would compare with the ones presented in this work.

References

[1] Y. Carafa, E. Brody, and C. Thermes, "Prediction of Rho-independent *Escherichia coli* Transcription Terminators," *Journal of Molecular Biology*, vol. 216, no. 4, pp. 835–858, 1990.

[2] M. G. Maaß, "Linear Bidirectional On-Line Construction of Affix Trees," *Algorithmica*, vol. 37, no. 1, pp. 43–74, 2003.

[3] J. Stoye, "Affix Trees," *Technical Report, University of Bielefeld*, 2000.

[4] D. Strothmann, "The affix array data structure and its applications to RNA secondary structure analysis," *Theoretical Computer Science*, vol. 389, no. 1–2, pp. 278–294, 2007.

[5] T. Schnattinger, E. Ohlebusch, and S. Gog, "Bidirectional search in a string with wavelet trees and bidirectional matching statistics," *Information and Computation*, vol. 213, pp. 13–22, 2012.

[6] B. Cazaux, R. Cánovas, and E. Rivals, "Shortest DNA cyclic cover in compressed space," in *Data Compression Conference DCC*, 2016, pp. 536–545.

[7] S. Gog, K. Karhu, J. Kärkkäinen, V. Mäkinen, and N. Välimäki, "Multi-pattern matching with bidirectional indexes," *Journal of Discrete Algorithms*, vol. 24, pp. 26–39, 2014.

[8] E. Fredkin, "Trie memory," in *Communications of the ACM 3*, 1960, pp. 490–500.

[9] D. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[10] E. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 32, no. 2, pp. 262–272, 1976.

[11] P. Weiner, "Linear pattern matching algorithms," in *Proc. 14th Annual Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

[12] A. Abeliuk, R. Cánovas, and G. Navarro, "Practical Compressed Suffix Trees," *Algorithms*, vol. 6, no. 2, pp. 319–351, 2013.

[13] E. Ohlebusch, J. Fischer, and S. Gog, "CST++," in *Proc. 17th International Symposium, SPIRE 2010*, 2010, pp. 322–333.

[14] L. Russo, G. Navarro, and A. Oliveira, "Fully-Compressed Suffix Trees," *ACM Transactions on Algorithms (TALG)*, vol. 7, no. 4, p. article 53, 2011, 35 pages.

[15] K. Sadakane, "Compressed suffix trees with full functionality," *Theory of Computing Systems*, vol. 41, no. 4, pp. 589–607, 2007.

[16] U. Manber and E. Myers, "Suffix arrays: a new method for on-line string searches," *SIAM Journal on Computing*, pp. 935–948, 1993.

[17] P. Ferragina, R. González, G. Navarro, and R. Venturini, "Compressed Text Indexes: From Theory to Practice," *ACM Journal of Experimental Algorithmics (JEA)*, vol. 13, p. article 12, 2009.

[18] G. Navarro and V. Mäkinen, "Compressed Full-Text Indexes," *ACM Computing Surveys*, vol. 39, no. 1, p. article 2, 2007.

[19] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep., 1994.

[20] R. Grossi, A. Gupta, and J. Vitter, "High-order entropy-compressed text indexes," in *Proc. 14th Symposium on Discrete Algorithms (SODA)*, 2003, pp. 841–850.

[21] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[22] B. Albrecht and V. Heun, "Space Efficient Modifications to Structator - A Fast Index-Based Search Tool for RNA Sequence-Structure Patterns," in *11th International Symposium on Experimental Algorithms, (SEA)*, 2012, pp. 27–38.

[23] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. M. Yiu, "High Throughput Short Read Alignment via Bi-directional BWT," in *Bioinformatics and Biomedicine*, 2009, pp. 31–36.

[24] C. Pockrandt, M. Ehrhardt, and K. Reinert, "Constant-time and space-efficient unidirectional and bidirectional FM-indices using EPR-dictionaries," *ArXiv e-prints*, 2016.