

# review articles

DOI:10.1145/2810036

## Tracing the first four decades in the life of suffix trees, their many incarnations, and their applications.

BY ALBERTO APOSTOLICO, MAXIME CROCHEMORE,  
MARTIN FARACH-COLTON, ZVI GALIL, AND S. MUTHUKRISHNAN

# 40 Years of Suffix Trees

WHEN WILLIAM LEGRAND finally decrypted the string, it did not seem to make much more sense than it did before.

53#305))6\*,48264‡.)4z);806”,48†8P60))85;1‡  
(;‡\*8†83(88)5\*†,46(;88\*96\*?;8)\* †(;485);5\*†2:‡ †  
(;4956\*2(5\*Ñ4)8P8\*;4069285);)6‡8)4‡‡;1(‡9;48081;8:  
8‡1;4885;4)485†528806\*81(ddag9;48;(88;4(‡?34;  
48)4‡;161;:188;‡?;

The decoded message read: “A good glass in the bishop’s hostel in the devil’s seat forty-one degrees and thirteen minutes northeast and by north main branch seventh limb east side shoot from the left eye of the death’s-head a bee line from the tree through the shot fifty feet out.” But at least it did sound more like natural language, and eventually guided the main character of Edgar Allan Poe’s “The Gold-Bug”<sup>36</sup> to discover the treasure he had been after. Legrand solved a substitution cipher using symbol frequencies.

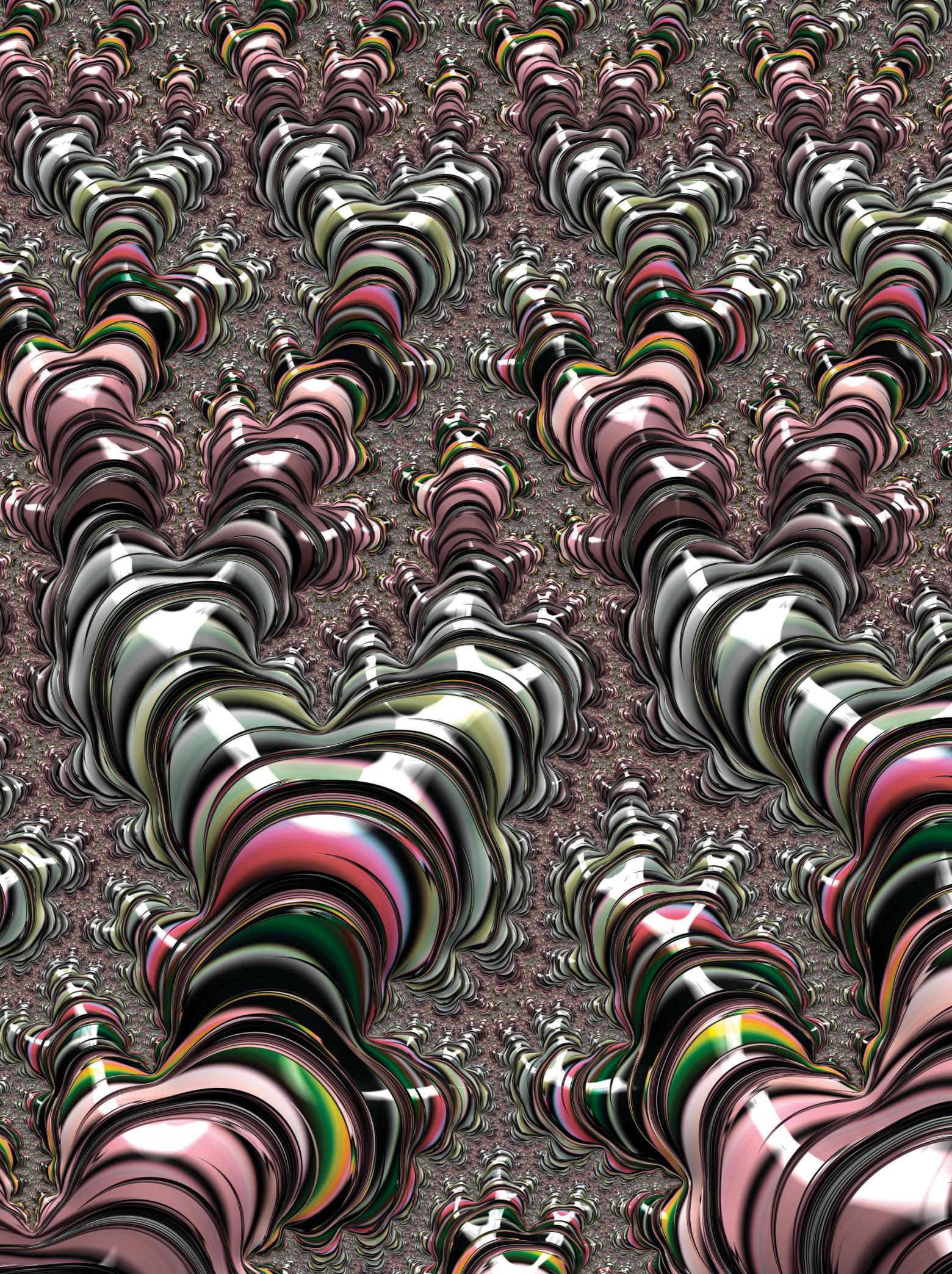
He first looked for the most frequent symbol and changed it into the most frequent letter of English, then similarly inferred the most frequent word, then punctuation marks, and so on.

Both before and after 1843, the natural impulse when faced with some mysterious message has been to count frequencies of individual tokens or subassemblies in search of a clue. Perhaps one of the most intense and fascinating subjects for this kind of scrutiny have been biosequences. As soon as some such sequences became available, statistical analysts tried to link characters or blocks of characters to relevant biological functions. With the early examples of whole genomes emerging in the mid-1990s, it seemed natural to count the occurrences of all blocks of size 1, 2, and so on, up to any desired length, looking for statistical characterizations of coding regions, promoter regions, among others.

This article is not about cryptography. It is about a data structure and its variants, and the many surprising and useful features it carries. Among these is the fact that, to set up a statistical table of occurrences for all substrings (also called *factors*), of any length, of a text string of  $n$  characters, it only takes time and space linear in the length of the text string. While nobody would be so foolish as to solve the problem by first generating all exponentially many possible strings and then counting their occurrences one by one, a text string may still contain  $\Theta(n^2)$  distinct substrings, so that tabulating all of them in linear space, never mind linear time, already seems puzzling.

---

We dedicate this article to our friend and colleague, Alberto Apostolico (1948–2015), who passed away on July 20. He was a major figure in the development of algorithms on strings.



Over the years, such structures have held center stage in text searching, indexing, statistics, and compression as well as in the assembly, alignment, and comparison of bi-

osequences. Their range of scope extends to areas as diverse as detecting plagiarism, finding surprising substrings in a text, testing the unique decipherability of a code, and more.

Their impact on computer science and IT at large cannot be overstated. Text searching and bioinformatics would not be the same without them. In 2013, the Combinatorial Pattern Matching symposium celebrated the 40<sup>th</sup> anniversary of the appearance of Weiner's invention of the suffix tree<sup>41</sup> with a special session entirely dedicated to that event.

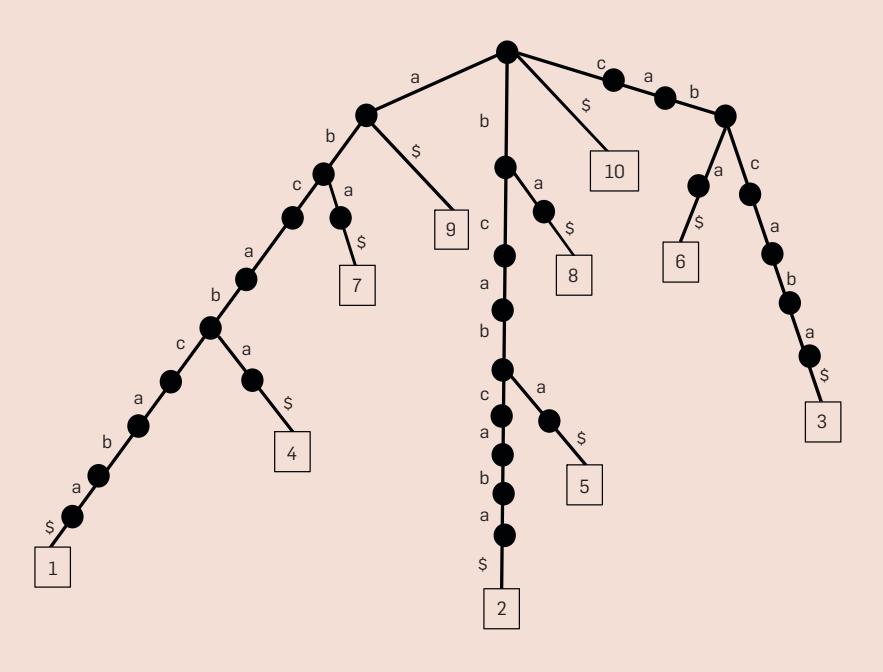
### History Bits and Pieces

At the dawn of "stringology," Donald Knuth conjectured the problem of finding the longest substring common to two long text sequences of total length  $n$  required  $(n \log n)$  time. An  $O(n \log n)$ -time had been provided by Karp, Miller, and Rosenberg.<sup>26</sup> That construction was destined to play a role in parallel pattern matching, but Knuth's conjecture was short lived: in 1973, Peter Weiner showed the problem admitted an elegant linear-time solution,<sup>41</sup> as long as the alphabet of the string was fixed. Such a solution was actually a byproduct of a construction he had originally set up for a different purpose, that is, identifying any substring of a text file without specifying all of them. In doing so, Weiner introduced the notion of a textual inverted index that would elicit refinements, analyses, and applications for 40 years and counting, a feature hardly shared by any other data structure.

Weiner's original construction processed the text file from right to left. As each new character was read in, the structure, which he called a "bi-tree," would be updated to accommodate longer and longer suffixes of the text file. Thus, this was an inherently offline construction, since the text had to be known in its entirety before the construction could begin. Alternatively, one could say the algorithm would build the structure for the reverse of the text online. About three years later, Ed McCreight provided a left-to-right algorithm and changed the name of the structure to "suffix tree," a name that would stick.<sup>32</sup>

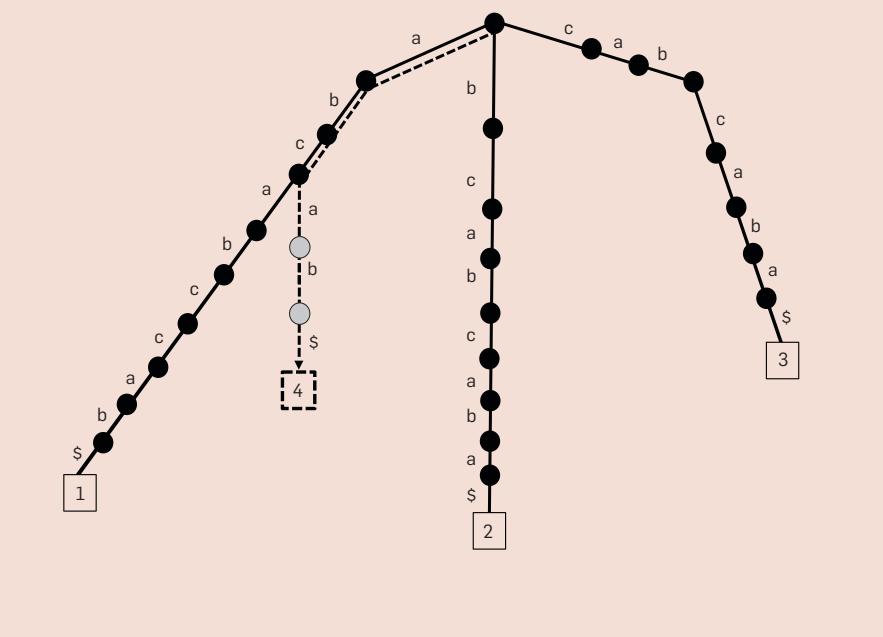
Let  $x$  be a string of  $n - 1$  symbols over some alphabet  $\Sigma$  and  $\$$  an extra character not in  $\Sigma$ . The expanded suffix tree  $T_x$  associated with  $x$  is a digital search tree collecting all suffixes of  $x\$$ . Specifically,  $T_x$  is defined as follows.

**Figure 1.** The expanded suffix tree of the string  $x = abcabcaba$ .



**Figure 2.** Building an expanded suffix tree by insertion of consecutive suffixes (showing here the insertion of  $abcaba\$$ ).

The insertion of suffix  $suf_i$  ( $i = 1, 2, \dots, n$ ) consists of two phases. In the first phase, we search for  $suf_i$  in  $T_{i-1}$ . Note the presence of  $\$$  guarantees that every suffix will end in a distinct leaf. Therefore, this search will end with failure sooner or later. At that point, we will have identified the longest prefix of  $suf_i$  that has a locus (that is, a terminal node) in  $T_{i-1}$ . Let head*i*abcab in the example be this prefix and  $\alpha$  the locus of head*i*. We can write  $suf_i = \text{head}_i \cdot \text{tail}_i$ , with  $\text{tail}_i$  ( $a\$$  in the example) nonempty. In the second phase, we need to add to  $T_{i-1}$  a path leaving node  $\alpha$  and labeled  $\text{tail}_i$ . This achieves the transformation of  $T_{i-1}$  into  $T_i$ .



1.  $T_x$  has  $n$  leaves, labeled from 1 to  $n$ .  
 2. Each arc is labeled with a symbol of  $\Sigma \cup \{\$\}$ . For any  $i$ ,  $1 \leq i \leq n$ , the concatenation of the labels on the path from the root of  $T_x$  to leaf  $i$  is precisely the suffix

$$\text{suf}_i = x_i x_{i+1} \dots x_{n-1} \$.$$

3. For any two suffixes  $\text{suf}_i$  and  $\text{suf}_j$  of  $x\$$ , if  $w_{ij}$  is the longest common prefix that  $\text{suf}_i$  and  $\text{suf}_j$  have in common, then the path in  $T_x$  relative to  $w_{ij}$  is the same for  $\text{suf}_i$  and  $\text{suf}_j$ .

An example of expanded suffix tree is given in Figure 1.

The tree can be interpreted as the state transition diagram of a deterministic finite automaton where all nodes and leaves are final states, the root is the initial state, and the labeled arcs, which are assumed to point downward, represent part of the state-transition function. The state transitions not specified in the diagram lead to a unique non-final sink state. Our automaton recognizes the (finite) language consisting of all substrings of string  $x$ . This observation also clarifies how the tree can be used in an online search: letting  $y$  be the pattern, we follow the downward path in the tree in response to consecutive symbols of  $y$ , one symbol at a time. Clearly,  $y$  occurs in  $x$  if and only if this process leads to a final state. In terms of  $T_x$ , we say the locus of a string  $y$  is the node  $\alpha$ , if it exists, such that the path from the root of  $T_x$  to  $\alpha$  is labeled  $y$ .

An algorithm for the direct construction of the expanded  $T_x$  (often called suffix trie) is readily derived (see Figure 2). We start with an empty tree and add to it the suffixes of  $x\$$  one at a time. This procedure takes time  $\Theta(n^2)$  and  $O(n^2)$  space, however, it is easy to reduce space to  $O(n)$  thereby producing a suffix tree in compact form (Figure 3). Once this is done, it becomes possible to aim for an expectedly non-trivial  $O(n)$  time construction.

At the CPM Conference of 2013, McCreight revealed his  $O(n)$  time construction was not born as an alternative to Weiner's—he had developed it in an effort to understand Weiner's paper, but when he showed it to Weiner asking him to confirm he had understood that paper the answer was "No, but you have come

up with an entirely different and elegant construction!" In unpublished lecture notes of 1975, Vaughan Pratt displayed the duality of this structure and Weiner's "repetition finder."<sup>37</sup> McCreight's algorithm was still inherently offline, and it immediately triggered a search for an online version. Some partial attempts at an online algorithm were made, but such a variant had to wait almost two decades for Esko Ukkonen's paper in 1995.<sup>39</sup> In all these linear-time constructions, linearity was based on the assumption of a finite alphabet and took  $\Theta(n \log n)$  time without that assumption. In 1997, Martin Farach introduced an algorithm that abandoned the one suffix-at-time approach prevalent until then; this algorithm gives a linear-time reduction from suffix-tree construction to character sorting, and thus is optimal for all alphabets.<sup>17</sup> In particular, it runs in linear time for a larger class of alphabets, for example, when the alphabet size is polynomial in input length.

Around 1984, Blumer et al.<sup>9</sup> and Crochemore<sup>14</sup> exposed the surprising result that the smallest finite automaton recognizing all and only the suffixes of

## » key insights

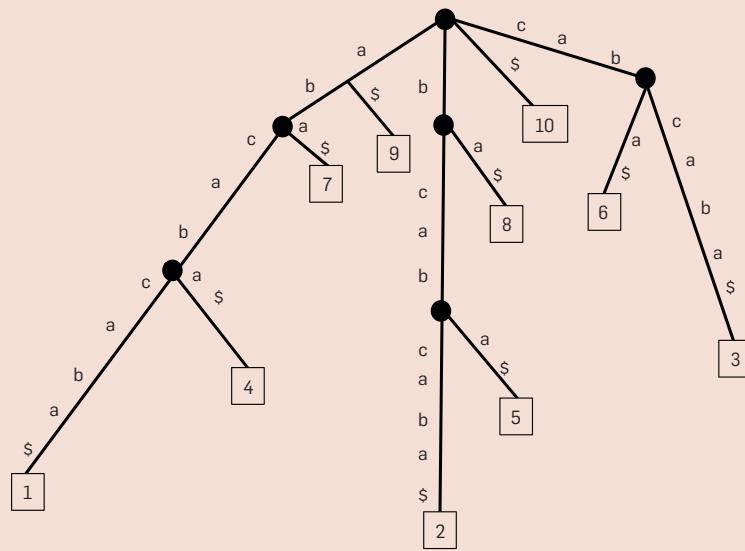
- The suffix tree is the core data structure in string analysis.
- It has a rich history, with connections to compression, matching, automata, data structures and more.
- There are powerful techniques to build suffix trees and use them efficiently in many applications.

a string of  $n$  characters has only  $O(n)$  states and edges. Initially coined a directed acyclic word graph (DAWG), it can even be further reduced if all states are terminal states.<sup>14</sup> It then accepts all substrings of the string and is called the factor—substring automaton. There is a nice relation between the index data structures when the string has no end-marker and its suffixes are marked with terminal states in the tree.

Then, the suffix tree is the edge-compacted version of the tree and its number of nodes can be minimized like with any automaton thereby providing the compact DAWG of the string. Permuting the two operations, compaction and minimization, leads to the same structure. Apparently Anatoli Slissenko (see the appendix avail-

**Figure 3. A suffix tree in compact form.**

This is obtained by first collapsing every chain formed by nodes with only one child into a single arc. The resulting compact version of  $T_x$  has at most  $n$  internal nodes, since there are  $n + 1$  leaves in total and every internal node is branching. The labels of the generic arc are now a substring, rather than a symbol of  $x\$$ . However, arc labels can be expressed by suitable pairs of pointers to a common copy of  $x\$$  thus achieving  $O(n)$  space bound overall.



able with this article in the ACM Digital Library under Source Material) ended up with a similar structure for his work on the detection of repetitions in strings. These automata provide another more efficient counterexample to Knuth's conjecture when they are used, against the grain, as pattern-matching machines (see Figure 4).

The appearance of suffix trees dovetailed with some interesting and independent developments in information theory. In his famous approach to the notion of information, Kolmogorov equated the information or structure in a string to the length of the shortest program that would be needed to produce that string by a Universal Turing Machine. The unfortunate thing is this measure is not computable and even if it were, most long strings are incompressible (that is, lack a short program producing them), since there are increasingly many long strings and comparatively much fewer short programs (themselves strings).

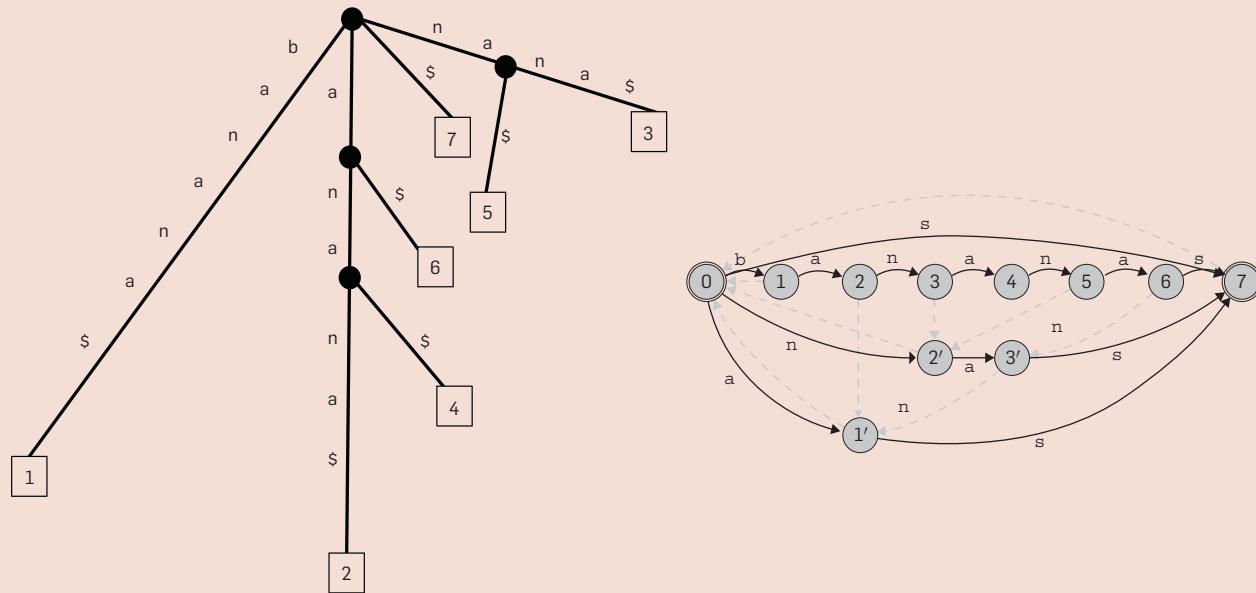
The regularities exploited by Kolmogorov's universal and omniscient machine could be of any conceivable kind, but what if one limited them to the syntactic redundancies affecting a text in the form of repeated substrings? If a string is repeated many times one could profitably encode all occurrences by a pointer to a common copy. This copy could be internal or external to the text. In the former case one could have pointers going in both directions or only in one direction, allow or forbid nesting of pointers, and so on. In his doctoral thesis, Jim Storer showed that virtually all such "macro schemes" are intractable, except one. Not long before that, in a landmark paper entitled "On the Complexity of Finite Sequences,"<sup>30</sup> Abraham Lempel and Jacob Ziv had proposed a variable-to-block encoding, based on a simple parsing of the text with the feature that the compression achieved would match, in the limit, that produced by a compressor tailored to the source probabilities.

Thus, by a remarkable alignment of stars, the compression method brought about by Lempel and Ziv was not only optimal in the information theoretic sense, but it found an optimal, linear-time implementation by the suffix tree, as was detailed immediately by Michael Rodeh, Vaughan Pratt, and Shimon Even.<sup>38</sup>

In his original paper, Weiner listed a few applications of his "bi-tree" including most notably offline string searching: preprocessing a text file to support queries that return the occurrences of a given pattern in time linear in the length of the pattern. And of course, the "bi-tree" addressed Knuth's conjecture, by showing how to find the longest substring common to two files in linear time for a finite alphabet. There followed unpublished notes by Pratt entitled "Improvements and Applications for the Weiner Repetition Finder."<sup>37</sup> A decade later, Alberto Apostolico would list more applications in a paper entitled "The Myriad Virtues of Suffix Trees,"<sup>2</sup>

**Figure 4. The compact suffix tree (left) and the suffix automaton (right) of the string "bananas."**

Failure links are represented by the dashed arrows. Despite the fact it is an index on the string, the same automaton can be used as a pattern-matching machine to locate substrings of "bananas" in another text or to compute their longest common substring. The process runs online on the second string. Assume for example "bana" has just been scanned from the second string and the current state of the automaton is state 4. If the next letter is "n," the common substring is "banan" of length 5 and the new state is 5. If the next letter is "s," the failure link is used and from state 3' corresponding to a common substring "ana" of length 3 we get the common substring "ana" with the new state 7. If the next letter is "b," iterating the failure link leads to state 0 and we get the common substring "b" with the new state 1. Finally, any other next letter will produce the empty common substring and state 0.



and two decades later suffix trees and companion structures with their applications gave rise to several chapters in reference books by Crochemore and Rytter, Dan Gusfield, and Crochemore, Hancart, and Lecroq (see the appendix available with this article in the ACM Digital Library).

The space required by suffix trees has been a nuisance in applications where they were needed the most. With genomes on the order of gigabytes, for instance, the space difference between 20 times larger than the source versus, say, only 11 times larger, can be substantial. For a few lustra, Stefan Kurtz and his co-workers devoted their effort to cleverly allocating the tree and some of its companion structures.<sup>28</sup> In 2001, David R. Clark and J. Ian Munro proposed one of the best space-saving methods on secondary storage.<sup>13</sup> Clark and Munro's "succinct suffix tree" sought to preserve as much of the structure of the suffix tree as possible. Udi Manber and Eugene W. Myers took a different approach, however. In 1990, they introduced the "suffix array,"<sup>31</sup> which eliminated most of the structure of the suffix tree, but was still able to implement many of the same operations, requiring space equal to 2 integers per text character and searching in time  $O(|P| + \log n)$  (reducible to 1 by accepting search time  $O(|P| + \log n)$ ). The suffix array stores the suffixes of the input in lexicographic order and can be seen as the sequence of leaves' labels as found in the suffix tree by a preorder traversal that would expand each node according to the lexicographic order.

Although the suffix array seemed at first to be a different data structure than the suffix tree, the distinction has receded. For example, Manber and Myers's original construction of the suffix array took  $O(n \log n)$  time for any alphabet, but the suffix array could be constructed in linear time from the suffix tree for any alphabet. In 2001, Toru Kasai et al.<sup>27</sup> showed the suffix tree could be constructed in linear time from the suffix array. Therefore, the suffix array was shown to be a succinct representation of the suffix tree. In 2003, three groups presented three different modifications of Farach's algorithm for suffix tree con-

## Although the suffix array seemed at first to be a different data structure than the suffix tree, the distinction has receded.

struction to give the first linear-time algorithms for directly constructing the suffix array; that is, the first linear-time algorithms for computing suffix arrays that did not first compute the full suffix tree. Since then, there have been many algorithms for fast construction of suffix arrays, notably by Nong, Zhang, and Chan,<sup>35</sup> which is linear time and fast in practice. With fast construction algorithms and small space required, the suffix array is the suffix-tree variant that has gained the most widespread adoption in software systems. A more recent succinct suffix tree and array, which take  $O(n)$  bits to represent for a binary alphabet ( $O(n \log \sigma)$  bits otherwise), was presented by Grossi and Vitter.<sup>21</sup>

Actually, the histories of suffix trees and compression are tightly intertwined. This should not come as a surprise, since the redundancies that pattern discovery tries to unearth are ideal candidates to be removed for purposes of compression. In 1994, M. Burrows and D.J. Wheeler proposed a breakthrough compression method based on suffix sorting.<sup>11</sup> Circa 1995, Amihood Amir, Gary Benson, and Martin Farach posed the problem of searching in compressed texts.<sup>1</sup> In 2000, Paolo Ferragina and Giovanni Manzini introduced the FM-index, a compressed suffix array based on the Burrows-Wheeler transform.<sup>19</sup> This structure, which may be smaller than the source file, supports searching without decompression. This was extended to compressed tree indexing problems in Ferragina et al.<sup>18</sup> using a modification of the Burrows-Wheeler transform.

### Fallout, Extensions, and Challenges

As highlighted out the outset, there has been hardly any application of text processing that did not need these indexes at one point or another. A prominent case has been searching with errors, a problem first efficiently tackled in 1985 by Gad Landau in his Ph.D. thesis.<sup>29</sup> In this kind of search, one looks for substrings of the text that differ from the pattern in a limited number of errors such as a single character deletion, insertion or substitution. To efficiently solve this problem, Landau combined suf-

fix trees with a clever solution to the so-called lowest common ancestor (LCA) problem. The LCA problem assumes a rooted tree is given and then it seeks, for any pair of nodes, the lowest node in the tree that is an ancestor of both.<sup>23</sup> It is seen that following a linear-time preprocessing of the tree any LCA query can be answered in constant time. Landau used LCA queries on suffix trees to perform constant-time jumps over segments of the text that would be guaranteed to match the pattern. When  $k$  errors are allowed, the search for an occurrence at any given position can be abandoned after  $k$  such jumps. This leads to an algorithm that searches for a pattern with  $k$  errors in a text of  $n$  characters in  $O(nk)$  steps.

Among the basic primitives supported by suffix trees and arrays, one finds, of course, the already mentioned search for a pattern in a text in time proportional to the length of the pattern rather than the text. In fact, it is even possible to enumerate occurrences in time proportional to their number and, with trivial preprocessing of the tree, tell the total number of occurrences for any query pattern in time proportional to the pattern size. The problem of finding the longest substring appearing twice in a text or shared between two files has been noted previously: this is probably where it all started. A germane problem is that of detecting squares, repetitions, and maximal periodicities in a text, a problem rooted in work by Axel Thue dated more than a century ago with multiple contemporary applications in compression and DNA analysis. A square is a pattern consisting of two consecutive occurrences of the same string. Suffix trees have been used to detect in optimal  $O(n \log n)$  time all squares (or repetitions) in a text, each with its set of starting positions,<sup>5</sup> and later to find and store all distinct square substrings in a text in linear time. Squares play a role in an augmentation of the suffix tree suitable to report, for any query pattern, the number of its non-overlapping occurrences.<sup>6,10</sup>

There are multiple uses of suffix trees in setting up some kind of signature for text strings, as well as measures of similarity or difference.

## There are multiple uses of suffix trees in setting up some kind of signature for text strings, as well as measures of similarity or difference.

Among the latter, there is the problem of computing the forbidden or absent words of a text, which are minimal strings that do not appear in the text (while all their proper substrings do).<sup>8,15</sup> Such words lead to, among other things, an original approach to text compression.<sup>16</sup> Once regarded as the succinct representation of the “bag-of-words” of a text, suffix trees can be used to assess the similarity of two text files, thereby supporting clustering, document classification, and even phylogeny.<sup>4,12,40</sup> Intuitively, this is done by assessing how much the trees for the two input sequences have in common. Suitably enriched with the probability of the substring ending at each node, a tree can be used to detect surprisingly over-represented substrings of any length,<sup>3</sup> for example, in the quest of promoter regions in biosequences.

The suffix tree of the concatenation of say,  $k \geq 2$  text files, supports efficient solutions to problems arising in domains ranging from plagiarism detection to motif discovery in biosequences. The need for  $k$  distinct end-markers poses some subtleties in maintaining linear time, for which the reader is referred to Gusfield.<sup>22</sup> In its original form, the problem of indexing multiple texts was called the “color problem” and seeks to report, for any given query string and in time linear in the query, how many documents out of the total of  $k$  contain at least one occurrence of the query. A simple and elegant solution was given in 1992 by Lucas C.K. Hui.<sup>25</sup> Recently, the combined suffix trees of many strings (also known as the generalized suffix tree) was used to solve a variety of document listing problems. Here, a set of text documents is preprocessed as a combined suffix tree. The problem is to return the list of all documents that contain a query pattern in time proportional to the number of such documents, not to the total number of occurrences (occ), which can be significantly larger. This problem was solved in Muthukrishnan<sup>33</sup> by reducing it to *range minimum queries*. This basic document-listing problem has since been extended to many other problems including listing the top- $k$  in various string and information distances. For example, in Hon

et al.,<sup>24</sup> the structure of generalized suffix tree is crucially used to design a linear machine-word data structure to return the top- $k$  most frequent documents containing a pattern  $p$  in time nearly linear in pattern size.

One surprising variant of the suffix tree was introduced by Brenda Baker for purposes of detection of plagiarism in student reports as well as optimization in software development.<sup>7</sup> This variant of pattern matching, called “parameterized matching,” enables one to find program segments that are identical up to a systematic change of parameters, or substrings that are identical up to a systematic relabeling or permutation of the characters in the alphabet. One obvious extension of the notion of a suffix tree is to more than one dimension, albeit the mechanics of the extension itself are far from obvious.<sup>34</sup> Among more distant relatives, one finds “wavelet trees.” Originally proposed as a representation of compressed suffix arrays,<sup>20</sup> wavelet trees enable one to perform on general alphabets the ranking and selection primitives previously limited to bit vectors, and more.

The list could go on and on, but the scope of this article was not meant to be exhaustive. Actually, after 40 years of unrelenting developments, it is fair to assume the list will continue to grow. Open problems also abound. For instance, many of the observed sequences are expressed in numbers rather than characters, and in both cases are affected by various types of errors. While the outcome of a two-character comparison is just one bit, two numbers can be more or less close, depending on their difference or some other metric. Likewise, two text strings can be more or less similar, depending on the number of elementary steps necessary to change one in the other. The most disruptive aspect of this framework is the loss of the transitivity property that leads to the most efficient exact string matching solutions. And yet indexes capable of supporting fast and elegant approximate pattern queries of the kind just highlighted would be immensely useful. Hopefully, they will come up soon and, in time, have their own 40<sup>th</sup>-anniversary celebration.

**Acknowledgments.** We are grateful to Ed McCreight, Ronnie Martin, Vaughan Pratt, Peter Weiner, and Jacob Ziv for discussions and help. We are indebted to the referees for their careful scrutiny of an earlier version of this article, which led to many improvements. 

## References

- Amir, A., Benson, G. and Farach, M. Let sleeping files lie: Pattern matching in Z-compressed files. In *Proceedings of the 5<sup>th</sup> ACM-SIAM Annual Symposium on Discrete Algorithms* (Arlington, VA, 1994), 705–714.
- Apostolico, A. The myriad virtues of suffix trees. *Combinatorial Algorithms on Words*, vol. 12 of *NATO Advanced Science Institutes, Series F*. A. Apostolico and Z. Galil, Eds. Springer-Verlag, Berlin, 1985, 85–96.
- Apostolico, A., Bock, M.E. and Lonardi, S. Monotony of surprise and large-scale quest for unusual words. *J. Computational Biology* 10, 3 / 4 (2003), 283–311.
- Apostolico, A., Denas, O. and Dress, A. Efficient tools for comparative substring analysis. *J. Biotechnology* 149, 3 (2010), 120–126.
- Apostolico, A. and Preparata, F.P. Optimal off-line detection of repetitions in a string. *Theor. Comput. Sci.* 22, 3 (1983), 297–315.
- Apostolico, A. and Preparata, F.P. Data structures and algorithms for the strings statistics problem. *Algorithmica* 15, 5 (May 1996), 481–494.
- Baker, B.S. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.* 26, 5 (1997), 1343–1362.
- Béal, M.-P., Mignosi, F. and Restivo, A. Minimal forbidden words and symbolic dynamics. In *Proceedings of the 13<sup>th</sup> Annual Symposium on Theoretical Aspects of Computer Science*, vol. 1046 of *Lecture Notes in Computer Science* (Grenoble, France, Feb. 22–24, 1996). Springer, 555–566.
- Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T. and Seiferas, J. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.* 40, 1 (1985), 31–55.
- Brodal, G.S., Lyngsø, R.B., Östlin, A. and Pedersen, C.N.S. Solving the string statistics problem in time  $O(n \log n)$ . In *Proceedings of the 29<sup>th</sup> International Colloquium on Automata, Languages and Programming*, vol. 2380 of *Lecture Notes in Computer Science* (Malaga, Spain, July 8–13, 2002). Springer, 728–739.
- Burrows, M. and Wheeler, D.J. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corp., May 1994.
- Chairungsee, S. and Crochemore, M. Using minimal absent words to build phylogeny. *Theoretical Computer Science* 450, 1 (2012), 109–116.
- Clark, D.R. and Munro, J.I. Efficient suffix trees on secondary storage. In *Proceedings of the 7<sup>th</sup> ACM-SIAM Annual Symposium on Discrete Algorithms*, (Atlanta, GA, 1996), 383–391.
- Crochemore, M. Transducers and repetitions. *Theor. Comput. Sci.* 45, 1 (1986), 63–86.
- Crochemore, M., Mignosi, F. and Restivo, A. Automata and forbidden words. *Information Processing Letters* 67, 3 (1998), 111–117.
- Crochemore, M., Mignosi, F., Restivo, A. and Salemi, S. Data compression using antidictionaries. In *Proceedings of the IEEE Special Issue Lossless Data Compression* 88, 11 (2000). J. Storer, Ed., 1756–1768.
- Farach, M. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38<sup>th</sup> IEEE Annual Symposium on Foundations of Computer Science* (Miami Beach, FL, 1997), 137–143.
- Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. Compressing and indexing labeled trees with applications. *JACM* 57, 1 (2009).
- Ferragina, P. and Manzini, G. Opportunistic data structures with applications. In *FOCS* (2000), 390–398.
- Grossi, R., Gupta, A. and Vitter, J.S. High-order entropy-compressed text indexes. In *SODA* (2003), 841–850.
- Grossi, R. and Vitter, J.S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings ACM Symposium on the Theory of Computing* (Portland, OR, 2000). ACM Press, 397–406.
- Gusfield, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, U.K., 1997.
- Harel, D. and Tarjan, R.E. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.
- Hon, W.-K., Shah, R. and Vitter, J.S. Space-efficient framework for top- $k$  string retrieval problems. In *FOCS*. IEEE Computer Society, 2009, 713–722.
- Hui, L.C.K. Color set size problem with applications to string matching. In *Proceedings of the 3<sup>rd</sup> Annual Symposium on Combinatorial Pattern Matching*, no. 644 in *Lecture Notes in Computer Science*, (Tucson, AZ, 1992). A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, Eds. Springer-Verlag, Berlin, 230–243.
- Karp, R.M., Miller, R.E. and Rosenberg, A.L. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4<sup>th</sup> ACM Symposium on the Theory of Computing* (Denver, CO, 1972). ACM Press, 125–13.
- Kasai, T., Lee, G., Arimura, H., Arikawa, S. and Park, K. Linear-time longest-common-prefix computation in suffix arrays and its applications. *CPM*. Springer-Verlag, 2001, 181–192.
- Kurtz, S. Reducing the space requirements of suffix trees. *Softw. Pract. Exp.* 29, 13 (1999), 1149–1171.
- Landau, G.M. String matching in erroneous input. Ph.D. Thesis, Department of Computer Science, Tel-Aviv University, 1986.
- Lempel, A. and Ziv, J. On the complexity of finite sequences. *IEEE Trans. Inf. Theory* 22 (1976), 75–81.
- Manber, U. and Myers, G. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1<sup>st</sup> ACM-SIAM Annual Symposium on Discrete Algorithms* (San Francisco, CA, 1990), 319–327.
- McCreight, E.M. A space-economical suffix tree construction algorithm. *J. Algorithms* 23, 2 (1976), 262–272.
- Muthukrishnan, S. Efficient algorithms for document listing problems. In *Proceedings of the 13<sup>th</sup> ACM-SIAM Annual Symposium on Discrete Algorithms* (2002), 657–666.
- J. C. Na, P. Ferragina, R. Giancarlo, and K. Park. Two-dimensional pattern indexing. In *Encyclopedia of Algorithms*. 2008.
- Nong, G., Zhang, S. and Chan, W.H. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Comput.* 60, 10 (2011), 1471–1484.
- Poe, E.A. *The Gold-Bug and Other Tales*. Dover Thrift Editions Series. Dover, 1991.
- Pratt, V. Improvements and applications for the Weiner repetition finder. Manuscript, 1975.
- Rodeh, M., Pratt, V. and Even, S. Linear algorithm for data compression via string matching. *J. Assoc. Comput. Mach.* 28, 1 (1981), 16–24.
- Ukkonen, E. On-line construction of suffix trees. *Algorithmica* 14, 3 (1995), 249–260.
- Ulitsky, I., Burstein, D., Tuller, T. and Chor, B. The average common substring approach to phylogenomic reconstruction. *J. Computational Biology* 13, 2 (2006), 336–350.
- Weiner, P. Linear pattern matching algorithms. In *Proceedings of the 14<sup>th</sup> Annual IEEE Symposium on Switching and Automata Theory*, (Washington, D.C., 1973), 1–11.

**Alberto Apostolico** held joint appointments with Georgia Tech’s School of Computational Science and Engineering School of Interactive computing as a professor and a researcher. He passed away on July 20, 2015.

**Maxime Crochemore** (maxime.crochemore@kcl.ac.uk) is a professor at King’s College London and Université Paris-Est, France.

**Martin Farach-Colton** (farach@cs.rutgers.edu) is a professor in the Department of Computer Science at Rutgers University, Piscataway, NJ.

**Zvi Galil** (galil@cc.gatech.edu) is Dean of the College of Computing at Georgia Institute of Technology, Atlanta, GA.

**S. Muthukrishnan** (muthu@cs.rutgers.edu) is a professor in the Department of Computer Science at Rutgers University, Piscataway, NJ.