I

STRINGS PROCESSING AND APPLICATION TO BIOLOGICAL SEQUENCES

1

STRING DATA STRUCTURES FOR COMPUTATIONAL MOLECULAR BIOLOGY

Christos Makris and Evangelos Theodoridis

1.1 INTRODUCTION

The topic of the chapter is string data structures with applications in the field of computational molecular biology. Let Σ be a finite alphabet consisting of a set of characters (or symbols). The cardinality of the alphabet denoted by $|\Sigma|$ expresses the number of distinct characters in the alphabet. A *string* or *word* is an ordered list of zero or more characters drawn from the alphabet. A word or string w of length n is represented by $w[1 \cdots n] = w[1]w[2] \cdots, w[n]$, where $w[i] \in \Sigma$ for $1 \leq i \leq n$ and $|w|$ denotes the length of w. The empty word is the empty sequence (of zero length) and is denoted by ε. A list of characters of w, appearing in consecutive positions, is called a substring of w, denoted by $w[i \cdots j]$, where i and j are the starting and ending positions, respectively. If the substring starts at position 1, then it is called a prefix, whereas if it ends at position n, then it is called a suffix of w. However, an ordered list of characters of w that are not necessarily consecutive is called a *subsequence* of w.

Strings and subsequences appear in a plethora of computational molecular biology problems because the basic types of DNA, RNA, and protein molecules can be represented as strings—pieces of DNA as strings over the alphabet $\{A, C, G, T\}$ (representing the four bases adenine, cytosine, guanine, and thymine, respectively), pieces of RNA as strings over the alphabet $\{A, C, G, U\}$ (with uracil replacing thymine),

Algorithms in Computational Molecular Biology, Edited by Mourad Elloumi and Albert Y. Zomaya

and proteins as strings over an alphabet of 20, corresponding to the 20 amino acid residues.

The basic string algorithmic problems that develop in computational molecular biology are:

- Exact pattern matching: given a pattern P and a text T to locate the occurrences of P into T
- Approximate pattern matching: given a pattern P, a text T, a similarity metric distance function $d()$, and a threshold parameter k to locate all positions i and j such that $d(P, T_{i\ldots j}) \leq k$
- Sequence alignment: given two string sequences, T_1 and T_2, try to find the best alignment between the two sequences according to various criteria. The alignment can be either local or global. A special case of this problem, which has great biological significance, is the longest common subsequence problem in which we try to locate the longest subsequence that is common to both sequences
- Multiple approximate and exact pattern matching in which more than two strings are involved into the computation
- String clustering: given a set of stings, cluster them into a set of clusters according to the distance between the involved strings; this problem has great biological significance because DNA sequence clustering and assembling overlapping DNA sequences are critical operations when extracting useful biological knowledge
- Efficient implementation of indexing techniques for storing and retrieving information from biological databases

Besides these classical string algorithmic problems, there are also applications that demand the processing of strings whose form deviates from the classical definition. The most known category of such variations are the *weighted* strings that are used to model molecular weighted sequences [54]. A molecular weighted sequence is a molecular sequence (a sequence of either nucleotides or amino acids) in which in every position can be stored a set of characters each having a certain weight assigned. This weight can model either the probability of appearance or the stability of the character's contribution to the molecular complex. These sequences appear in applications concerning the DNA assembly process or in the modeling of the binding sites of regulatory proteins. In the first case, the DNA must be divided into many short strings that are sequenced separately and then are used to assemble the sequence of the full string; this reassembling introduces a degree of uncertainty that initially was expressed with the use of the "don't care" character denoted as "$*$," which has the property of matching against any symbol in the given alphabet. It is possible, though, that scientists are able to be more exact in their modeling and determine the probability of a certain character to appear at a position; j then a position that previously was characterized as a wild card is replaced by a probability of appearance for each of the characters of the alphabet and such a sequence is modeled as

a *weighted sequence*. In the second case, when a molecular weighted sequence models the binding site of a regulatory protein, each base in a candidate motif instance makes some positive, negative, or neutral contribution to the binding stability of the DNA–protein complex [37, 65], and the weights assigned to each character can be thought of as modeling those effects. If the sum of the individual contributions is greater than a threshold, then the DNA–protein complex can be considered stable enough to be functional.

A related notion is the notion of an *indeterminate* or (equivalently in the scientific literature) a *degenerate* string. This specific term refers to strings in which each position contains a set of characters instead of a simple character; in these strings, the match operation is replaced by the subset operation. The need for processing efficiently such strings is driven by applications in computational biology cryptanalysis and musicology [76, 45]. In computational biology, DNA sequences still may be considered to match each other if letter A (respectively, C) is juxtaposed with letter T (respectively, G); moreover, indeterminate strings can model effectively polymorphism in protein coding regions. In cryptanalysis, undecoded symbols can be modeled as the set or characters that are candidates for the specific position, whereas in music, single notes may match chords or a set of notes.

Perhaps the most representative application of indeterminate strings is haplotype inference [42, 63]. A haplotype is a DNA sequence that has been inherited by one parent. A description of the data from a single copy is called a haplotype, whereas a description of the mixed data on the two copies is called a genotype. The underlying data that form a haplotype is either the full DNA sequence in the region or, more commonly, is the values of only DNA positions that are single nucleotide polymorphisms (SNPs). Given an input set D of n genotype vectors, a solution to the haplotype inference problem is a set of n pairs of binary strings one pair for each genotype; for any genotype g, the associated binary vectors v_1, v_2 must be a "feasible resolution" of g into two haplotypes that could explain how g was created. Several algorithms have been proposed for the haplotype inference problem such as those based on the "pure parsimony criteria," greedy heuristics such as "Clarks rule," Expectation Maximization (EM)-based algorithms, and algorithms for inferring haplotypes from a set of Trios [42, 63]. Indexing all possible haplotypes that can be inferred from D as well as gathering statistical information about them can be used to accelerate these haplotype inference algorithms. Moreover, as new biological data are being acquired at phenomenal rates, biological datasets have become too large to be readily accessible for homology searches, mining adequate modeling, and integrative understanding. Scalable and integrative tools that access and analyze these valuable data need to be developed. The new generation of databases have to (i) encompass terabytes of data, often local and proprietary; (ii) answer queries involving large and complex inputs such as a complete genome; and (iii) handle highly complex queries that access more than one dataset. These queries demand the efficient design of string indexing data structures in external memory; the most prominent of these structures are: the string B-tree of Ferragina and Grossi [30], the cache oblivious string dictionaries of Brodal and Fagerberg [15], the cache-oblivious string B-trees [14], and various heuristic techniques for externalizing the suffix tree [28].

In the sequel, (i) we will present the main string indexing data structures (suffix trees and suffix arrays), (ii) we will present the main indexing structures for weighted and indeterminate strings, and (iii) we will present the main external memory string indexing structures.

1.2 MAIN STRING INDEXING DATA STRUCTURES

In this subsection, we will present the two main string indexing structures, suffix trees [92] and suffix arrays [67], and depict their special characteristics and capabilities.

1.2.1 Suffix Trees

The suffix tree is the most popular data structure for full string indexing, which was presented by Weiner in 1973 [92]. It is considered the oldest and most studied data structure in the area that, besides answering effectively to the pattern matching problem, can be used for the efficient handling of a plethora of string problems (see [41] for a set of applications in the area of computational molecular biology). More formally, the suffix tree ST_T of a text T is a compact digital search tree (trie) that contains all suffixes of T as keys. It is assumed that before building the suffix tree, the text T gets padded with an artificial character—the \$ character, which does not belong in the alphabet Σ from which T was formed. This assumption is used to guarantee that every suffix is stored to a distinct leaf of the tree (that is, no suffix is a prefix of another). The leaf of the tree that corresponds to the suffix $T_{i\cdots n}\$$ stores the integer i.

The suffix tree has the following structural properties that are induced by its definition:

- There are n leaves—a leaf for each suffix of T. The concatenation of the substrings at the edges, which we traverse when moving from the root to the leaf that stores i, forms the suffix $T_{i\cdots n}$.
- Consider two different suffixes of T, $T_{i\cdots n} = xa$ and $T_{j\cdots n} = xb$, that share a common prefix x. In the suffix tree, the two leaves that correspond to the two suffixes have a common ancestor u for whom the concatenation of the substrings at the edges that we traverse, moving from the root to the u, forms the common prefix x. This also can be phrased in a different way. For every internal node u of a suffix tree, *all* suffixes that correspond to the leaves of its subtree share a common prefix x that is represented from the edges of the path from the root to u. The substring that is created from the concatenation of the substrings of the edges traversed when moving from the root to u is called the *path label* of the node u.

A lot of sequential algorithms have been proposed for building a suffix tree in linear time. The algorithms provided in [92, 70, 88] are based on the assumption that

the strings have been formed from alphabets of constant size and basically are based on similar concepts. On the other hand, and for large alphabets (where we cannot ignore a time cost similar to its size), an elegant linear time algorithm has been proposed in [27]. Finally, in [10, 43, 81] parallel algorithms have been presented for the CRCW PRAM parallel model of computation.

Concerning implementation, a suffix tree ST_T for a string T of n characters will have at most $2n - 1$ nodes and $2n - 2$ edges. The edges of a node can be stored either in a linear list (unsorted or sorted according to the first character of the edge label) or in an array of size $|\Sigma|$. In the first case (space-efficient implementation), a node can be traversed in $O(|\Sigma|)$ time, whereas in the second case, a node can be transversed in $O(1)$ time (though the space complexity for large alphabets is clearly worse). Between these two extreme choices of implementation, we can choose other alternatives as search trees or hash tables. The most efficient implementation, especially in the average case, is based in the use of a hash table. In [70], it the usage of the hashing scheme of Lampson [61], is proposed which belongs to the class of hash functions with chaining.

The suffix tree data structure can be extended to store the suffixes of more than one strings. In this case, we talk about the *generalized suffix tree*. More formally, a generalized suffix tree (GST) ($\{T_1, T_2, \cdots T_k\}$) of a set of strings $\{T_1, T_2, \cdots T_k\}$ is the compact trie that contains all suffixes of these strings as keys. For the construction of a generalized suffix tree, we can use the known algorithms for constructing suffix trees by superimposing the suffixes of different strings in the same structure; when having completed the insertion of the suffixes for a string, the procedure is continued for the next string by beginning from the root. A generalized suffix tree occupies $O(|T_1| + |T_2| + \cdots + |T_k|)$ space, and it can be built in $O(|T_1| + |T_2| + \cdots + |T_k|)$ time [41].

Concerning applications, let us consider the pattern matching problem and see how the suffix tree deals with the specific problem. Consider a string T for which we have built the suffix tree ST_T and suppose that we want to locate the positions within it where a pattern P appears. By starting from the root of ST_T, we follow the path that is defined by P. After the i-th step of this procedure, if we are at an internal node and we have matched the i leftmost characters of P, then we follow the outgoing edge that starts with the character P_{i+1}, whereas if we are at the interior of the edge, then we test whether the next character at the edge is equal to P_{i+1}. If this traversal from the root to the leaves finishes by matching successfully all $|P|$ characters of the pattern, then according to the aforementioned properties, the suffixes that correspond to the subtree below the point where the pattern matching procedure ended, share the pattern P as a common prefix. Hence, the requested pattern appears at the positions that correspond to the leaves of that subtree. If the match procedure from the root to the leaf finishes before accessing all characters of the pattern, then no suffixes of T can have the pattern P as prefix; hence, the pattern does not appear anywhere inside the text. As a conclusion and with the assumption that in every internal node the edge that will be followed is being chosen in constant time, at most $|P|$ comparisons with the characters of the pattern are performed and the time complexity totals $(|P| + \alpha)$, where α is the size of the answer.

The suffix tree can answer to numerous other questions optimally besides performing pattern matching efficiently. The interested reader can consult [41] and the respective chapters in [72] and [3]. Some characteristic applications of the suffix tree are the *longest repeated substring* and the *longest common substring* (LCS) problems. In the longest repeated substring problem, we seek the longest substring of a text T that appears in T more than once. Initially, we built the suffix tree ST_T in $O(|T|)$ time, and then by performing a traversal of the nodes of the suffix tree, we compute for every node the number of characters of the string from the root to the node. Then, we locate the internal node with the label of maximum length; the positions that are stored in the leaves of the subtrees below that node are the positions where the longest repeated substring appears. In the LCS problem, we search for the longest common substring of two strings T_1 and T_2. Initially and in time $O(|T_1| + |T_2|)$, we construct the generalized suffix tree $gST(\{T_1, T_2\})$ of the two sequences. In this generalized tree, some leaves will store suffixes of the one string, some of the other, and some will store suffixes of both strings. We traverse all nodes of the tree, and we compute for every node the number of the characters from the root to it; by a similar traversal, we mark the nodes in whose subtrees are stored leaves of both strings. Then to get our reply, we simply have to select the internal marked node with the path label of maximum length. Then the positions that correspond to the leaves of the corresponding subtrees are the positions where the longest common substring appears. With a similar linear time algorithmic procedure, we can locate the longest common substring between a set of more than two strings.

Concluding the suffix tree, is the main and better representative for data structures for full text indexing. The cost for this enhanced functionality is the extra space complexity. There are cases in which the required space can be 25 times more than the indexed data. This fact and the poor behavior when being transferred in secondary memory restricts the use of suffix trees in applications that are limited in the main memory of a computer system.

Optimizations of the suffix tree structure to face these disadvantages were undertaken by McCreight [70] and more recently by Kurtz [62]. Kurtz reduced the RAM required to around 20 bytes per input character indexed on the worst case and to 10,1 bytes per input character on average. Compact encodings of the suffix tree based on a binary representation of the text have been investigated by Munro and Clark [20] Munro *et al.* [73] and Anderson *et al.* [7]. There are also other works concerning efficient compression of the suffix tree; the interested reader should consult [32, 38, 39, 73, 80] for further details on this interesting algorithmic area.

1.2.2 Suffix Arrays

The suffix arrays have been introduced in [67] and constitute the main alternative full text indexing data structure as compared with the suffix tree. The main advantages of the suffix array are its small memory requirements, its implementation simplicity, and the fact that the time complexities for constructing and query answering are independent from the size of the alphabet. Its main disadvantages are that the query time is usually larger than the respective query time of the suffix tree and that the

range of applications where it can be used is smaller than the range of applications of the suffix tree.

More formally, a suffix array SA_T for a string T of n characters is an array that stores the suffixes of T in lexicographic order. That is, in every position i of the suffix array, the starting position j of the suffix $T_{j\cdots n}$ ($SA_T[i] = j$) is stored in such a way that the suffixes that are lexicographically smaller than $T_{j\cdots n}$ are located in positions smaller than i, whereas the suffixes that are lexicographically larger than $T_{j\cdots n}$ are located in positions larger than i. Hence, we get $T_{SA_T[1]\cdots n} <_L T_{SA_T[2]\cdots n} <_L \cdots <_L T_{SA_T[n]\cdots n}$ where $<_L$ designates the lexicographic order. Because suffix arrays store the suffixes of T lexicographically ordered they have the following property: suppose that the suffixes, located at positions i, j, with $i < j$ have a common prefix x, that is LCP $(_{SA_T[i]\cdots n}, {}_{SA_T[j]\cdots n}) = x$. Then all suffixes $T_{SA_T[w]\cdots n}$ that are located in positions $i \leq w \leq j$ have x as a prefix.

Because the suffix array is basically an array of n elements without the need for extra pointers, its space requirements are significantly smaller (in terms of constant factors) from the respective space requirements that characterize the suffix trees. However, the use of suffix array without extra information does not permit efficient searching. To understand this concept, let us explain how the suffix tree can solve the problem of exact pattern matching of a pattern P into a text T. To accomplish the search, we need to locate two positions i, j with $i \leq j$ for which the following holds: the first $|P|$ characters of the suffixes at position j are lexicographically smaller or equal from the pattern (that is $T_{SA_T[j]\cdots SA_T[j]+|P|} \leq_L P$), and j is the maximum position with this property, whereas the first $|P|$ characters of the suffix at position i are lexicographically larger or are more equal than the pattern (that is $P \leq_L T_{SA_T[i]\cdots SA_T[i]+|P|}$) and i is smaller with that property. According to that, the suffixes that correspond to positions i, j, and all intermediate positions have P as a prefix. Consequently, the places where P appears in T can be located by finding the two extreme positions i, j in the suffix array and then scanning the intermediate positions. To locate the extreme positions, a binary search needs to be executed on the suffix array in which at each step of the search procedure, $|P|$ comparisons are needed and then the procedure moves right or left. Hence, the problem of pattern matching by using the suffix arrays is solved in $(|P| \log n + \alpha)$ time, where α is the size of the answer. This time complexity can be reduced significantly to $(|P| + \log n + \alpha)$ if we use two more arrays of $n - 2$ elements containing precomputed information; with the help of these elements, it is possible in every repetition of the binary search procedure, not to execute all $|P|$ comparisons that correspond to the middle of the active segment. In particular, suppose that the binary search procedure is in an interval $[L, R]$ of the suffix array, and we seek to compute the value $m = \text{LCP}(P, T_{SA_T[M]\cdots n})$ for the middle of the search interval. We suppose that the values $l = \text{LCP}(P, T_{SA_T[L]\cdots n})$ and $r = \text{LCP}(P, T_{SA_T[R]\cdots n})$ have been computed in a previous repetition of the binary search. The first remark that can be made is that we do not have to perform all $|P|$ comparisons from the beginning because of the basic property of the suffix array $m \geq \min\{l, r\}$; hence, the comparisons can continue from position $m + 1$, and hence, it is possible to save $\min\{l, r\}$ comparisons. However, despite this improvement, there are scenarios in which the order

of the total complexity does not change. Suppose now that we have as extra information the longest common prefix of the suffixes of the left edge L of the search interval, with the suffix at the middle and the longest common prefix of the right edge R of the search interval with the suffix of the middle. Let us symbolize them as $x = \text{LCP}(T_{SA_T[L]\cdots n}, T_{SA_T[M]\cdots n})$ and $x' = \text{LCP}(T_{SA_T[M]\cdots n}, SA_T[R]\cdots n)$, respectively. Let us suppose that $l \geq r$, and hence, we have the following, depending on whether x is the longest common prefix of the left edge with the medium, or is largest, smaller, or equal to l:

- If $x > l$, then because L has the first l characters equal to P and the first x equal to the suffix at position M and it holds $x > l$, the $l + 1$-st character of the suffix at position M does not match with the $l + 1$-st character of P. Hence, according to the basic property of the suffix array, no common prefix exists with P to the left side of M. Hence, we choose $[M, R]$ as the new interval.
- If $x < l$, then because P matches with the l characters of L and with the x characters of the middle suffix, we will have a nonmatching of the middle suffix at position $x + 1$. Hence, a larger prefix of P must exist in the left interval, and hence, we will choose $[L, M]$ as the new search interval.
- If x equals l, then we cannot deduce that the longest common prefix with P is in the left or the right interval. Hence, by a character to character comparison, we extend the common prefix (if it can be extended) beyond the position l. If we perform Δh successful comparisons, then the common prefix of the suffix of M with P will have length $l + \Delta h$. The failure in matching at position $l + \Delta h + 1$ guides us left or right depending on whether the character of the corresponding position at P is lexicographically smaller or larger than the respective position at the middle suffix.

Hence, every one of the $O(\log n)$ steps of the binary search either performs a constant number of operations (cases $x > l$ or $x < l$) or performs Δh comparisons (case $x = l$). The sum of comparisons in the last case does not exceed $|P|$ because the middle chosen element will be one of the extreme elements in the next repetition (its value is continuously increasing). Hence, the problem of pattern matching is being solved in $(|P| + \log n)$ time.

Concerning the needed space consumption, the improved time complexity is achieved by using the $\text{LCP}(T_{SA_T[L]\cdots n}, T_{SA_T[M]\cdots n})$ and $\text{LCP}(T_{SA_T[M]\cdots n}, T_{SA_T[R]\cdots n})$ values as precomputed information for every possible interval that can exist during binary searching. The number of different intervals is $n - 2$ because the role of middle elements can be played by all elements, except the first and the last. Hence, one array stores the values of the left extreme for every possible middle element, whereas the other array stores the values of the right extreme. The existing suffix array algorithms precompute in the arrays $\text{LCP}[i] = \text{LCP}(T_{SA_T[i]\cdots n}, T_{SA_T[i+1]\cdots n})$ for $i = 1 \cdots n$ in linear time. By using the relationship $\text{LCP}(L, R) = \min\{\text{LCP}(L, M), \text{LCP}(M, R)\}$ from this array, we can create the LCP

values for all possible intervals of binary searching. Concluding the enhanced suffix array construction occupies approximately $5n$ space (see also chapter 29 of [3]), which is less than the space complexity of the suffix tree. The time to solve a pattern matching problem is $O(|P| + \log n)$, which can be compared with the time required for the query replying of a suffix tree $O(|\Sigma|P)$ (in the implementation that is space effective) or $O(|P|)$ (in the implementation that is time effective). The construction algorithm that has been presented initially in [67] was not linear but needed $O(n \log n)$ time. A linear time procedure can be envisaged by simply constructing (in linear time) a suffix tree and then transforming it to the respective suffix array. This transformation is possible by traversing lexicographically the suffix tree in linear time and then computing the arrays LCP in linear time by nearest common ancestor queries. This procedure takes linear time but cancels the main advantage of the suffix tree, which is the small space consumption. In 2003, three papers were published [58, 75, 26] that describe a worst-case linear time construction of a suffix array without the need of an initial construction of the respective suffix tree. Other algorithms for constructing suffix arrays can be found in [17, 36, 47, 55, 68]. Moreover, a recent line of research concerns compressed suffix arrays [46, 38, 39, 35].

Concerning applications, the main weakness of the suffix array in comparison with the suffix tree data structure is that the range of application in which it can be used is limited. To resolve this handicap, in [59], a method was presented that combined the information of a suffix array with the LCP information, which simulates the postorder traversal in the equivalent suffix tree of the string, thus providing the so-called *virtual suffix tree*. This simulation (which was extended in [35] with a space efficient variant) gives the ability for some of the suffix tree applications whose algorithmic procedure is based in the bottom-up traversal of the suffix tree to be executed with some extra changes in the suffix array.

The suffix array table is being traversed from left to right, and an auxiliary stack is being used. Initially, the stack contains the root and $\text{LCP}[1] = \text{LCP}(T_{SA_T[1]\cdots n}, T_{SA_T[2]\cdots n}) \geq 0$. If this value is equal to zero, then the two leftmost leaf-suffixes have a minimum common ancestor in the root, and hence, during the implicit postorder traversal, we process the first and then the second element. If the value is greater than zero, then an internal node exists that is being inserted in the stack. More generally, during step i, if $\text{LCP}[i] = (T_{SA_T[i]\cdots n}, T_{SA_T[i+1]\cdots n})$ is larger than the depth of the node u at the top of the stack (that is, the length of the path label $L(u)$), then between the i-th leaf/suffix and the next, a deeper node exists that will be inserted in the stack; otherwise, the value $\text{LCP}[i]$ is smaller than the depth of node u, and the minimum common ancestor is higher in the path from u to the root. In this case, the stack is emptied until a node is located with smaller depth, and the first case is applied. Based on this described procedure, a node is inserted in the stack when it is seen during the top-down traversal, whereas it is removed from the stack when it is faced moving bottom up for the last time. Because in every step of the method, we either add a node in the stack or we have several deletions from the stack, every node is inserted and deleted from the stack once, and the whole procedure needs $O(n)$ time. In [1], other combinations of the suffix array with additional information

were provided (the so-called *enhanced suffix array*), and additional applications were described.

1.3 INDEX STRUCTURES FOR WEIGHTED STRINGS

The notion of the weighted sequence extends the notion of a regular string by permitting in each position the appearance of more than one character, each with a certain probability. In the biological scientific literature weighted sequences also are called position weight matrices (PWM). More formally, a weighted word $w = w_1 w_2 \cdots w_n$ is a sequence of positions with each position w_i consisting of a set of couples, of the form $(s, \pi_i(s))$, with $\pi_i(s)$ being the probability of having the character s at position i. For every position w_i, $1 \leq i \leq n$, $\sum \pi_i(s) = 1$.

For example, if we consider the DNA alphabet $\Sigma = \{A, C, G, T\}$, then the word $w =$ [(A,0.25), (C,0.5), (G,0.25), (T,0)][(A,1), (C,0), (G,0), (T,0)] [(A,1), (C,0), (G,0), (T,0)] represents a word having three letters; the first one is either A,C,G with probabilities of appearance of 0.25, 0.5, and 0.25, respectively; the second one is always A, whereas the third letter is necessarily an A because its probability of presence is equal to 1. The probability of presence of a subword either can be defined to be the cumulative probability, which is calculated by multiplying the relative probabilities of appearance of each character in every position, or it can be defined to be the average probability.

There have been published works in the scientific literature [19, 5, 6, 54] concerning the processing of string sequences; we will refer to these works giving more emphasis to the structure presented in [54]. In [19], a set of efficient algorithms were presented for string problems developing in the computational biology area. In particular, assume that we deal with a weighted sequence X of length n and with a pattern p of length m, then (i) the occurrences of p in X can be located in $O((n + m) \log m)$ time and linear space; the solution works for both the multiplicative and the average model of probability estimation, although it can be extended also to handle the appearance of gaps; (ii) the set of repetitions and the set of covers (of length m) in the weighted sequence can be computed in $O(n \log m)$ time. In [6] and for the multiplicative model of probability estimation the problem of approximately matching a pattern in a weighted sequence was addressed. In particular, two alternative definitions were given for the *Hamming distance* and two alternative definitions for the *edit distance* in weighted sequences with the aim of capturing the aspects of various applications. The authors presented algorithms that compute the two versions of the Hamming distance in time $O(n\sqrt{m \log m})$, where the length of the weighted text is n, and m is the pattern length; the algorithms are based in the application of nontrivial bounded divide-and-conquer algorithms coupled with some insights on weighted sequences. The two versions of the edit distance problem were solved by applying dynamic programming algorithm with the first version being solved in $O(nm)$ time and the other version in $O(nm^2)$ time. Finally, the authors extended the notion of weighted matching in infinite alphabets and showed that exact weighted matching can be computed in $O(s \log^2 s)$

time, where s is the number of text symbols with nonzero probability, and they also proved that the weighted Hamming distance over infinite alphabets can be computed in $\min(O(kn\sqrt{s} + s^{3/2}\log^2 s), O(s^{4/3}m^{1/3}\log s))$, where m is the length of the pattern. In [5], a different approach was followed, and a transformation was proved between weighted matching and *property matching* in strings; the pattern matching with properties (property matching for short) was introduced in the specific paper in which pattern matching with properties involves a string matching between the pattern and the text and the requirement that the text part satisfies some property. The aforementioned reduction allows off-the-self solutions to numerous weighted matching problems (some were not handled in the previously published literature) such as scaled matching, swapped matching, pattern matching with swaps, parameterized matching, dictionary matching, and the indexing problem. All presented results are enabled by a reduction of weighted matching to property matching that creates an ordinary text of length $O(n(\frac{1}{\epsilon})^2\log\frac{1}{\epsilon})$ for the weighted matching problem of length n and the desired probability of appearance ϵ. Based on this reduction, all pattern matching problems that can be solved in ordinary text can have their weighted versions solved with the time degradation of the reduction.

Finally in [54], a data structure was presented for storing weighted sequences that can be considered the appropriate generalization of the suffix tree structure to handle weighted sequences. A resemblance exists between this structure and the work related to *regulatory motifs* [71, 66, 83, 60] and *probabilistic suffix trees* [78, 82, 69]. Regulatory motifs characterize short sequences of DNA and determine the timing location and level of gene expression, and the approaches extracting regulatory motifs can be divided into two categories: those that exploit word-counting heuristics [57, 69] and those based on the use of probabilistic models [40, 48, 64, 79, 85, 87]; in the second category of approaches, the motifs are represented by position probabilistic matrices, whereas the remainder of the sequences are represented by background models. The probabilistic or *prediction* suffix tree is basically a stochastic model that employs a suffix tree as its index structure to represent compactly the conditional probabilities distribution for a cluster of sequences. Each node of a probabilistic suffix tree is associated with a probability vector that stores the probability distribution for the next symbol given the label of the node as the preceding segment, and algorithms that use probabilistic suffix trees to process regulatory motifs can be found in [82, 69]. However, the probabilistic suffix tree is inefficient for efficiently handling weighted sequences, which is why the weighted suffix tree was introduced; however, it could be possible for a suitable combination of the two structures to be effective to handle both problem categories.

The main idea behind the weighted suffix tree data structure is to construct the suffix tree for the sequence incorporating the notion of probability of appearance for each suffix; that is, for every suffix $x[i \cdots n]$, we store in a set of leaves labeled S_i the first l characters so that $\pi(x_i \cdots x_{i+l-1}) \geq 1/k$. In more detail, for every suffix starting at position i, we define a list of possible weighted factors (not suffixes because we may not eventually store the entire suffix) so that the probability of appearance for each one of them is greater than $1/k$; here, k is a user-defined parameter that is used to denote substrings that are considered valid.

The formal definition as provided in [54] is that the weighted suffix tree (WST) of a weighted sequence S (denoted as WST(S)) is the compacted trie of all weighted factors starting within each suffix S_i of S having a probability of appearance greater than 1/k. The leaf u of WST(S) is labeled with index i if $L(u) = S_{i,j}[i \cdots n]$ and $\pi(S_{i,j}[i \cdots n]) \geq 1/k$, where $j > 0$ denotes the j-th weighted factor starting at position i. $L(u)$ denotes the path label of node u in WST(S) and results by concatenating the edge labels along the path from the root to u. $D(u) = |L(u)|$ is the string-depth of u, whereas LL(u) is defined as the leaf list of the subtree below u. LL(u) $= \emptyset$ if u is a leaf.

It can be proven that the time and space complexity of constructing a WST is linear to the length of the weighted sequence.

The WST is endowed with most of the sequence manipulation capabilities of the generalized suffix tree, that is:

- Exact pattern matching: Let P and T be the pattern and the weighted sequence, respectively. Initially, the weighted suffix tree is built for T, and if the pattern P is weighted, too, then it is broken into solid subwords; for each of these subwords, the respective path is spelled by moving from the root of the tree until an internal node is reached then all leaves descending from this node are reported. The time complexity of the procedure is $O(m + n + a)$, where m and n are the sizes of the pattern and the text, respectively, and a is the answer size.
- Finding repetitions in weighted sequences: It is possible to compute all repetitions in a given weighted sequence, with each repetition having a probability of appearance greater than $1/k$; initially, the respective weighted suffix tree is constructed, and then the weighed suffix tree is traversed with a depth-first traversal, during which a leaf list is kept for each internal node. The elements of a leaf list are reported if the size of the list exceeds two; in total, the problem is solved in $O(n + a)$ time, where n is the sequence length and a is the answer size.
- Longest common substring of weighted sequences: The generalized weighted suffix tree is built for two weighted sequences, w_1 and w_2, and then the internal node with the greatest depth is located; the path label of this node corresponds to the longest weighted subsequence of the two weighted strings. The time complexity of the procedure is equal to $O(n_1 + n_2)$, with n_1 and n_2 being the sizes of w_1 and w_2, respectively.

1.4 INDEX STRUCTURES FOR INDETERMINATE STRINGS

Indeterminate or (equivalently in the scientific literature) *degenerate* strings are strings that in each position contain a set of characters instead of a simple character. The simplest form of indeterminate string is one in which indeterminate positions can contain only a do-not-care letter that is a letter "$*$," which matches any letter in the alphabet on which x is defined. In 1974, an algorithm was described [33] for computing all occurrences of a pattern in a text where both the pattern and the text

can contain do-not-care characters, but although efficient in theory, the algorithm was not useful in practice; the same remark also holds for the algorithms presented in [2]. In [11, 93], the shift-or technique (a bit-mapping technique for pattern matching) was applied to find matches in indeterminate strings. In [51], an easily implemented average case $O(n)$ time algorithm was proposed for computing all periods of every prefix of a string with do-not-care characters. In [44], this work was extended by distinguishing two forms of indeterminate match ("quantum" and "deterministic") and by refining the definition of indeterminate letters so that they can be restricted to matching only with specified subsets of Σ rather than with every letter of Σ. More formally, a "quantum" match allows an indeterminate letter to match with two or more distinct letters during a single matching process, whereas a "deterministic" match restricts each indeterminate letter to a single match.

These works were continued by researchers in [9, 8, 50, 52, 53, 76, 45], in which a set of algorithms were presented for computing repetitive structures, computing covers, computing longest common subsequences, and performing approximating and exact pattern matching; some of them improved the aforementioned previous constructions. From these structures, special emphasis should be given to the works in [76] and [45] because they fell in the focus of interest of this chapter. In particular, in [45], efficient practical algorithms were provided for pattern matching on indeterminate strings where indeterminacy may be determined either as "quantum" or "deterministic"; the algorithms are based on the Sunday variant of the Boyer–Moore pattern matching algorithm and are applied more generally to all variants of Boyer–Moore (such as Horspool's) that depend only on the calculation of the "rightmost shift" array. It is assumed that Σ is indexed being essentially an integer alphabet. Moreover, three pattern-matching models are considered in increasing order of sophistication: (i) the only indeterminate letter permitted is the do-not-care character, whose occurrences may be either in the pattern or in the text, (ii) arbitrary indeterminate letters can occur but only in the pattern, (iii) indeterminate letters can occur in both the pattern and the text. In [76], and asymptotically faster algorithms were presented for finding patterns in which either the pattern or the text can be *degenerate* but not both. The algorithms for DNA and RNA sequences work in $O(n \log m)$ time, where n and m are the lengths of the text and the pattern, respectively. Efficient implementations also are provided that work in $O(n + m + n \lceil m/w \rceil + \lceil n/w \rceil)$ time, where w is the word size; as can be seen, for small sizes of the text and the pattern, the algorithms work in linear time. Finally it also is shown how their approach can be used to solve the distributed pattern matching problem.

Concerning indexing structures, there are some results that can be divided into two categories, one based on the use of compressed tries and the other based on the used of finite automata.

Concerning results in the first category in [63], the dictionary matching problem was considered in which the dictionary D consists of n indeterminate strings and the query p is a string over the given alphabet Σ. A string p matches a stored indeterminate string s_i if $|p| = |s_i|$ and $p[j] \in s_i[j]$ for every $1 \leq j \leq |p|$. The goal is to preprocess D for queries that search for the occurrence of pattern p in D and count the number of appearances of p in D.

Let m denote the length of the longest string in D and let D' be the set of all patterns that appear in D. For example, if D contains a single indeterminate string $cd\{a, b\}c\{a, b\}$, then $D' = \{cdaca, cdacb, cdbca, cdbcb\}$. The data structure is basically a compressed trie of D' that can be constructed naively in $O(|\Sigma|^k nm)$ time and $O(|\Sigma||D'|)$ space, assuming every $s \in D$ has at most k locations in which $|s[i]| > 1$. With this structure, a query time of $O(|p|)$ can be supported for a pattern p plus a time complexity equal to the size of the output. Using techniques presented in [22], the structure can be modified to solve the problem in $O(nm \log(nm) + n(c_1 \log n)^{k+1}/k!)$ preprocessing time, and $O(m + (c_2 \log n)^k \log\log n)$ query time (c_1 and c_2 are constants); this approach is worse than the trie approach for small values of Σ.

In [63], two faster constructions of the trie have been presented. The first construction is based on the divide-and-conquer paradigm and requires $O(nm + |\Sigma|^k n \log n)$ preprocessing time, whereas the second construction uses ideas introduced in [4] for text fingerprinting and requires $O(nm + |\Sigma|^k n \log m)$ preprocessing time. The space complexity is $O(|\Sigma||D'|)$, and it can be reduced to $O(|D'|)$ by using the suffix tray [23] ideas. The query time becomes $O(|p| + \log\log|\Sigma|)$, and it is also possible by cutting the dictionary strings and constructing two tries to obtain $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$ preprocessing time at the cost of $O(|p| \log\log|\Sigma| + \min\{|p|, \log|D'|\} \log\log|D'|) = O(|p| \log\log|\Sigma| + \min\{|p|, \log(|\Sigma|^k n)\} \log\log(|\Sigma|^k n))$ query time. The first two constructions can calculate the number of appearances in D of each pattern in D', a knowledge that can be useful in a possible application of the structures to the Haplotype inference problem [63].

On the other hand, there are works based in the use of finite automata, which are based in indexing small factors (that is, substrings of small size). Indexing of short factors is a widely used and useful technique in stringology and bioinformatics, which has been used in the past to solve diverse text algorithmic problems. More analytically, in [90], the *generalized factor automaton* (GFA) was presented, which has the disadvantage that it cannot be used to index large texts because, experimentally, it tends to grow super-quadratically with respect to the length of the string. Later in [91], the truncated generalized factor automaton (TGFA) was presented that is basically a modification of GFA that indexes only factors with length not exceeding a given constant k having at most a linear number of states. The problem with the specific algorithm is that it is based on the subset construction technique and inherits its space and time complexity that is a bottleneck of the algorithm when indexing very long text because the corresponding large Nondeterministic Finite Automaton needs to be determinized. Finally, in [34], an efficient on-line algorithm for the construction of the TGSA was presented, which enables the construction of TGSAs for degenerate strings of large sizes (order of Megabytes (MBs)). The proposed construction works in $O(n^2)$ time, where n is the length of the input sequence. TGSA has, at most, a linear number of states with respect to the length of the text and enables the location of the list $occ(u)$ of all occurrences of the given pattern u in the degenerate text in time $|u| + |\text{occ}(u)|$.

1.5 STRING DATA STRUCTURES IN MEMORY HIERARCHIES

We consider string data structures in external memory [89]. The design of such data structures is a necessity in the computational molecular biology area, as the datasets in various biological applications and the accumulated data in DNA sequence databases are grown exponentially, and it is not possible to have them in main memory; a characteristic number is provided in [74] where it is mentioned that in the GenBank, the DNA sequence database has crossed the 100 Gbp (bp stands for base pairs), with sequences from more than 165,000 organisms. The basic external memory model used to analyze the performance of the designed algorithms is the two-level memory model; in this model, the system memory is partitioned into a small but fast in access partition (the main memory with size M) and into a (theoretically unbounded) secondary part (the disk). Computations are performed by central processing unit (CPU) on data that reside in main memory while data transfer between memory and disk take place in contiguous pieces of size B (the block size).

In string algorithmics, there are two lines of related research, one that focuses on transferring the main-memory-tailored design of the suffix tree and/or suffix array data structures to secondary memory and another that tries to envisage novel, external-memory-tailored data structures with the same functionality as the suffix tree.

In the first line of work, a plethora of published material exists dealing with the externalization of the suffix tree: [49, 74, 86, 16, 49, 12, 13, 18, 21, 56, 84, 86] and the suffix array [24, 25]. Most of these works suffer from various problems such as nonscalability, nonavailability of suffix links (that are necessary for the implementation of various operations) and nontolerance to data skew, and a few are the works that manage to face effectively these problems; from these works, we will present briefly the approach in [74]. More specifically, the authors in [74] present TRELLIS, an algorithm for constructing genome-scale suffix trees on disk with the following characteristics: (i) it is an $O(n^2)$ time and $O(n)$ space algorithm that consists of four main phases—the prefix creation phase, the partitioning phase, the merging phase, and the suffix link recovery phase; the novel idea of the algorithm lies in the use of variable length prefixes for effective disk partitioning and in a fast postconstruction phase for recovering the suffix links; (ii) it can scale effectively for very large DNA sequences with suffix links; (iii) it is shown experimentally that it outperforms most other constructions because it is depicted as faster than the other algorithms that construct the human genome suffix tree by a factor of 2–4 times; moreover, its query performance is between 2–15 times faster than existing methods with each query taken on the average between 0.01–0.06 seconds.

In the second line of research, string B-trees [30], cache oblivious string dictionaries [14], and the cache oblivious string B-tree [15] come into play.

The string B-tree [30] is an extension of the B-tree suitable for indexing strings with a functionality equivalent to the functionality of the suffix tree. More analytically, assume that we have to process a set S of n strings with a total number of N characters and suppose that each of the strings is stored in a contiguous

sequence of disk pages and is represented by its logical pointer to the external memory address of its first character. In the leaves of the string B-tree, we store the logical pointers of the strings lexicographically ordered, and the leaves also are linked together to form a bidirectional list. In every internal node v, we store as search keys the leftmost and the rightmost string stored in each of the node's children. Hence, if v has k children c_i, $(1 \leq i \leq k)$ then the keys stored in v will be $K_v = \{L(c_1), R(c_1), L(c_2), R(c_2), \cdots, L(c_k), R(c_k)\}$, where $L(c_i)$ and $R(c_i)$ are the leftmost and the rightmost keys stored in the child c_i. The string B-tree in this form can answer prefix queries for a query string P. The total number of disk accesses will be $O(\frac{|P|}{B} \log_B |S| \log_2 B)$ I/Os because (i) $O(\log_2 B)$ accesses are needed in every internal node for locating the proper subtree via binary search, (ii) in every binary search step all characters of P will need to be loaded from the disk, and thus, a total of $O(\frac{|P|}{B})$ disk I/O accesses are needed. The whole procedure is executed in every internal node moving from the root to the leaf, and hence, it is repeated $O(\log_B |S|)$ times.

The time complexity of the aforementioned procedure can be reduced by organizing the elements stored in each node of the string B-tree as a Patricia trie. A Patricia trie is a compact digital search tree (trie) that can store a set of k strings in $O(k)$ space as follows: (i) a compacted trie of $O(k)$ nodes and edges is built on the k strings; (ii) in each compacted trie node we store the length of the substring into it, and the substring that normally would label each edge is replaced by its first character. This construction gives the possibility to fit $O(B)$ strings into one node independently of the length of the strings and allows lexicographic searches by branching out from a node without further disk accesses.

By using Patricia tries for storing the strings in internal nodes, we see that we do not need binary search in each node, but it is possible to select the proper subtree in $O(\frac{|P|}{B})$ I/Os, and hence, the total time complexity of disk accesses when moving from the root to the leaf becomes $O(\frac{|P|}{B} \log_B |S|)$ I/Os. The query time complexity can be reduced further by a more careful search procedure that will take into account the observation that the longest common prefix that a query can have with the keys of a node is at least equal to the longest common prefix between the query and the keys stored in the parent of the node; in this case, the query time becomes $O(\frac{|P|}{B} + \log_B |S|)$ I/Os for completing the traversal from the root to the leaves. Concerning dynamic operations to insert/delete a string T' a query initially is executed for locating the appropriate leaf position among the leaves of the string B-tree. If space exists for inserting the appropriate leaf, it is inserted; otherwise, the leaf gets split, and the father node is updated with appropriate pointers. The rest of the insertion and deletion procedure is similar to the balancing operations performed in the traditional B-tree with the difference that in the worst case the balancing operations can be propagated until the root of the tree, and hence, the total number of disk accesses will be bounded from above by $O(\frac{|T'|}{B} + \log_B |S|)$ I/Os.

The above lead to the following theorems:

Theorem 1.1 *The string B-tree can store a set S of n strings with a total number of N characters in $O(n/B)$ space (the index) plus $O(N/B)$ space (the characters*

of the string) so that the strings in S that have a query prefix P can be computed in $O(\frac{|P|}{B} + \log_B n + \frac{a}{B})$ I/Os, where a is the size of the answer. To insert or delete a string T' in S, $O(\frac{|T'|}{B} + \log_B n)$ I/Os are needed.

To use the string B-tree for efficient pattern matching, we should insert all suffixes of the involved strings, because in this case, prefix matching with a given pattern becomes equivalent to pattern matching with the given pattern. It can be proved that the produced structure will have the following properties:

Theorem 1.2 *The string B-tree can store a set S of n strings with a total number of N characters in $O(N/B)$ space (the index) plus $O(N/B)$ space (the characters of the string) so that the strings in S that contain a given query pattern P can be computed in $O(\frac{|P|+a}{B} + \log_B n)$ I/Os, where a designates the size of the answer. To insert or delete a string of length m in S $O(m \log_B(N + m))$ I/Os are needed.*

The specific structure has been improved with two other structures [14, 15] that are valid for the cache oblivious model of computation. The cache-oblivious model is a generalization of the two-level I/O model to a multilevel memory model, by employing a simple trick: the algorithm is not allowed to know the value of B and M, and thus, its functionality and working remains valid for any value of B and M. In particular, in [15], a cache-oblivious string dictionary structure was presented supporting string prefix queries in $O(\log_B n + |P|/B)$ I/Os, where P is the query string and n is the number of stored strings. The dictionary can be constructed in $O(\text{Sort}(N))$ time where N is the total number of characters in the input, and $\text{Sort}(N)$ is the number of I/Os needed for comparison-based sorting. The input as in the string B-tree can be either a set of strings to store or a single string for which all suffixes are to be stored; moreover, if it is given as a list of edges of the appropriate tree, then it also can accept a trie, a compressed trie, or a suffix tree. It is assumed that $M \geq B^{2+\delta}$. The aforementioned structure has the following two novel characteristics: (i) it uses the notion of the *giraffe tree* that provides an elegant linear space solution to the path traversal problem for trees in external memory; the giraffe trees permit the exploitation of redundancy because parts the path in the trie may be stored multiple times but with only a constant factor blowup in total space as the trie gets covered by them; (ii) it exploits a novel way for decomposing a trie into components and subcomponents based on judiciously balancing the progress in scanning the query pattern with the progress in reducing the number of strings left as matching candidates.

The aforementioned contribution was improved in [14] where a cache-oblivious string B-tree (COSB-tree) was presented that can search asymptotically optimal and insert/delete nearly optimal and can perform range queries with no extra disk seeks. An interesting characteristic of the structure is that it employs front compression to reduce the size of the stored set. In particular for a set D, assume that we denote by $||D||$ the sum of key lengths in D and by $\text{front}(D)$ the size of the front-compressed D. The proposed structure has space complexity $O(\text{front}(D))$ and possesses the following performance characteristics: (i) insertion of a key k

requires $O(1 + ||k||(\log^2 \text{front}(D))/B + \log_B N)$ memory transfers with high probability (w.h.p.), (ii) searches and successor/predecessor queries for a key k' require an optimal $O(1 + ||k'||/B + \log_B N)$ block transfers w.h.p. The result set Q is returned in compressed representation and can be decompressed in additional $O(||Q||/B)$ memory transfers, which is optimal for front compression. Because COSB-trees, store all keys in order on disk range, queries involve no extra disk seeks.

An important component of the COSB-tree of independent interest is the front-compressed packed memory array (FC-PMA) data structure. The FC-PMA maintains a collection of strings D stored in order with a modified front compression. As is shown in [14], the FC-PMA has the following properties: (i) for any ϵ, the space usage of the FC-PMA can be set to $(1 + \epsilon)$ front(D) while enabling a string k to be reconstructed with $O(1 + ||k||/(\epsilon B))$ memory transfers, (ii) inserting and deleting a string k into a FCPMA requires $O(||k||(\log^2 \text{front}(B))/(\epsilon B))$.

The interested reader can find a nice exposition of some of these plus other structures in [77, 28].

1.6 CONCLUSIONS

String indexing algorithms and data structures play a crucial role in the field of computational molecular biology, as most information is stored by means of symbol sequences. Storing, retrieving, and searching in this vast volume of information is a major task to have several specific queries and problems being solved efficiently. In this chapter, we have presented the main indexing structures in the area.

We conclude by noting that despite the great progress in the string indexing research field in the last decade, the frontiers need to move a little bit further by means of: (i) minimizing the volume of data with compression and searching in compressed files, (ii) minimizing the extent of the indexing structures by compressing them, too [29], (iii) building and placing the indexing structures cache obliviously to minimize the cache misses [31], and (iv) building the indexing structures efficiently in parallel, using the model multiprocessor machines and operating systems.

REFERENCES

1. M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Enhanced suffix arrays and applications. In S. Aluro, editor, *Handbook of Computational Molecular Biology*. Chapman and Hall, New York, 2006.
2. K. Abrahamson. Generalized string matching. *SIAM J Comput*, 16(6):1039–1051, 1987.
3. S. Aluru. *Handbook of Computational Molecular Biology*. Chapman and Hall, New York, 2006.
4. A. Amir, A. Apostolico, G.M. Landau, and G. Satta. Efficient text fingerprinting via parikh mapping. *J. Discrete Algorithm*, 1(5–6):409–421, 2003.

5. A. Amir, E. Chencinski, C. Iliopoulos, T. Kopelowitz, and H. Zhang. Property matching and weighted matching. *Theor Comput Sci*, pp. 298–310, 2006.
6. A. Amir, C. Iliopoulos, O. Kapah, and E. Porat. Approximate matching in weighted sequences. *Combin Pattern Matching*, pp. 365–376, 2006.
7. A. Andersson, N.J. Larsson, and K. Swanson. Suffix trees on words. *CPM '96: Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, London, UK, 1996, New York, pp. 102–115. Springer-Verlag.
8. P. Antoniou, M. Crochemore, C. Iliopoulos, I. Jayasekera, and G. Landau. Conservative string covering of indeterminate strings. *Proceedings of the 13th Prague Stringology Conference (PSC 2008)*, Prague, Czech Republik, 2008.
9. P. Antoniou, C. Iliopoulos, I. Jayasekera, and W. Rytter. Computing repetitive structures in indeterminate strings. *Proceedings of the 3rd IAPR, International Conference on Pattern Recognition in Bioinformatics (PRIB)*, Melborne, Australia, 2008.
10. A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree. *Algorithmica*, 3:347–365, 1988.
11. R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Commun ACM*, 35(10):74–82, 1992.
12. S. Bedathur and J. Haritsa. Engineering a fast online persistent suffix tree construction. *Proceedings of the 20th International Conference on Data Engineering*, Boston, MA, 2004, pp. 720–720.
13. S. Bedathur and J. Haritsa. Search-optimized suffix-tree storage for biological applications. *IEEE International Conference on High Performance Computing*, 2005, pp. 29–39.
14. M.A. Bender, M. Colton, and B. Kuzsmaul. Cache-oblivious string b-trees. *25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS06)*, Chicago, IL, 2006, pp. 233–242.
15. G. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Miami, Fl, 2006, pp. 581–590.
16. A.L. Brown. Constructing genome scale suffix trees. In Yi-Ping Phoebe Chen, editor, *Second Asia-Pacific Bioinformatics Conference (APBC2004)*, volume 29 of *CRPIT*. ACS, Dunedin, New Zealand, 2004.
17. S. Burkhardt and J. Karkkainen. Fast lightweight suffix array construction and checking. In *Symposium on Combinatorial Pattern Matching*, Springer Verlag, LNCS, New York, 2003.
18. C. Cheung, J. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Trans Knowl Data Eng*, 17(1):90–105, 2005.
19. M. Christodoulakis, C. Iliopoulos, L. Mouchard, K. Perdikuri, A. Tsakalidis, and K. Tsichlas. Computation of repetitions and regularities on biological weighted sequences. *J Comput Biol*, 13(6):1214–1231, 2006.
20. D. Clark and I. Munro. Efficient suffix trees on secondary storage. *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 1996, pp. 383–391.
21. R. Clifford and M. Sergot. Distributed and paged suffix trees for large genetic databases. *Combinatorial Pattern Matching, 14th Annual Symposium*, Michocan, Mexico, 2003, pp. 70–82.

22. R. Cole, M. Gottlieb, and L. Lewenstein. Dictionary matching and indexing with errors and dont cares. *Proceedings of the 36th Annual Symposium on Theory of Computing (STOC)*, Chicago, Il, 2004, pp. 91–100.
23. R. Cole, M. Kopelowitz, and L. Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. *Proceedings of the 33rd International Colloquium on Automata Languages and Programming(ICALP)*, Venice, Italy, 2006, pp. 358–369.
24. A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32:1–35, 2002.
25. R. Dementiev, J. Karkkainnen, J. Menhert, and P. Sanders. Better external memory suffix array construction. *Workshop on Algorithm Engineering and Experiments*, 2005, pp. 86–97.
26. D.K. Kim, J. Sim, H. Park, and K. Park. Linear time construction of suffix arrays. *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Michoacan, Mexico, 2003, pp. 186–199.
27. M. Farach. Optimal suffix tree construction with large alphabets. *38th Annual Symposium on the Foundations of Computer Science (FOCS)*, New York, 1997, pp. 137–143.
28. P. Ferragina. String search in external memory: Data structures and algorithms. In S. Aluro, editor, *Handbook of Computational Molecular Biology*, Chapman & Hall, New York, 2006.
29. P. Ferragina, R. Gonzalez, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM J Exper Algorithmics*, 13, 2008.
30. P. Ferragina and R. Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *J ACM*, 46(2):236–280, 1999.
31. P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. Vitter. On searching compressed string collections cache-obliviously. *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Vancouver, BC, Canada, 2008, pp. 181–190.
32. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness and beyond. *IEEE Symposium on Foundations of Computer Science*, pp. 184–196, 2005.
33. M.J. Fischer and M.S. Paterson. String-matching and other products. Technical Report, Cambridge, MA, 1974.
34. T. Flouri, C. Iliopoulos, M. Sohel Rahman, L. Vagner, and M. Voracek. Indexing factors in dna/rna sequences. *BIRD*, pp. 436–445, 2008.
35. G. Manzini. Two space saving tricks for linear time lcp array computation. In J. Gudmundssson, editor, *Scandinavian Workshop on Algorithm Theory*. Springer, New York, 2004.
36. G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. In B. Frakes and R.A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, Prentice Hall, Upper Saddle R, 1992.
37. G. Grillo, F. Licciuli, S. Liuni, E. Sbisa, and G. Pesole. Patsearch: A program for the detection of patterns and structural motifs in nucleotide sequences. *Nucleic Acids Res*, 31:3608–3612, 2003.
38. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments on suffix arrays and trees. In *ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, LA, 2004, pp. 636–645.

39. R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, ACM Press, New York, 2000, pp. 397–406.
40. M. Gupta and J. Liu. Discovery of conserved sequence patterns using a stochastic dictionary model. *J Am Stat Assoc*, 98:55–66, 2003.
41. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK, 1997.
42. D. Gusfield and S. Orzack. Haplotype inference. In S. Aluru, editor, *Handbook of Computational Molecular Biology,* Chapman and Hall, New York, 2006.
43. R. Hariharan. Optimal parallel suffix tree construction. *J Comput Syst Sci*, 55(1):44–69, 1997.
44. J. Holub and W.F. Smyth. Algorithms on indeterminate strings. *Proeedings of the 14th Australasian Workshop on Combinatorial Algorithms*, Seoul, Korea, 2003.
45. J. Holub, W.F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *J Discrete Algorithm*, 6:37–50, 2006.
46. W. Hon, T.W. Lam, K. Sadakane, and W. Sung. Constructing compressed suffix arrays with large alphabets. *International Symposium on Algorithms and Computation*, Kolkata, India, 2006, pp. 505–516.
47. W. Hon, K. Sadakane, and W. Sung. Breaking a time-space barrier in constructing full-text indices. *IEEE Symposium on Foundations of Computer Science*, 2003, pp. 251–260.
48. J. Hughes, P. Estep, S. Tavazoie, and G. Church. Computational identification of cis-regulatory elements associated with groups of functionally related genes in sacharomyces cerevisae. *J Mol Biol*, 296:1205–1214, 2000.
49. E. Hunt, M. Atkinson, and R. Irving. A database index to large biological sequences. *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Francisco, CA, 2001.
50. C. Iliopoulos, I. Jayasekera, B. Melichar, and J. Supol. Weighted degenerated approximate pattern matching. *Proceedings of the 1st International Conference on Language and Automata Theory and Applications*, Tarragona, Spain, 2007.
51. C. Iliopoulos, M. Mohamed, L. Mouchard, W. Smyth, K. Perdikuri, and A. Tsakalidis. String regularities with don't cares. *Nordic J Comput*, 10(1):40–51, 2003.
52. C. Iliopoulos, M. Rahman, and W. Rytter. Algorithms for two versions of lcs problem for indeterminate strings. *International Workshop on Combinatorial Algorithms* (IWOCA), Newcastle, Australia, 2007.
53. C. Iliopoulos, M. Rahman, M. Voracek, and L. Vagner. Computing constrained longest common subsequence for degenerate strings using finite automata. *Proceedings of the 3rd Symposium on Algorithms and Complexity*, Rome, Italy, 2007.
54. C.S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundamenta Informaticae*, 71(2–3):259–277, 2006.
55. H. Itoh and H. Tanaka. An eficient method for in memory construction of suffix arrays. *Symposium on String Processing and Information Retrieval*, 2006, pp. 81–88.
56. R. Japp. The top-compressed suffix tree: A disk-resident index for large sequences. *Bioinformatics Workshop, 21st Annual British National Conference on Databases*, 2004.

57. L. Jensen and S. Knudsen. Automatic discovery of regulatory patterns in promoter regions based on whole cell expression data and functional annotation. *Bioinformatics*, 16:326–333, 2000.
58. J. Karkkainen and P. Sanders. Simple linear work suffix array construction. *Proceedings of 30th International Colloquium on Automata, Languages and Programming(ICALP)*, Eindhoven, The Netherlands, 2003, pp. 943–955.
59. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-preffix computation in suffix arrays and its applications. *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, Jerusalem, Israel, 2001, pp. 181–192.
60. S. Keles, M. van der Laan, S. Dudoit, B. Xing, and M. Eisen. Supervised detection of regulatory motifs in dna sequences. *Statistical Applications in Genetics and Molecular Biology*, 2, 2003.
61. D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, 1998.
62. S. Kurtz. Reducing the space requirement of suffix trees. *Softw—Prac Exp*, 29(13):1149–1171, 1999.
63. G. Landau, D. Tsur, and O. Weimann. Indexing a dictionary for subset matching queries. *SPIRE*, pp. 195–204, 2007.
64. C. Lawrence, S. Altschul, M. Boguski, J. Liu, A. Neuwald, and J. Wootton. Detecting subtle sequence signals: A Gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, 1993.
65. H. Li, V. Rhodius, C. Gross, and E. Siggia. Identification of the binding sites of regulatory proteins in bacterial genomes. *Genetics*, 99:11772–11777, 2002.
66. Y. Liu, L. Wei, S. Batzoglou, D. Brutlag, J. Liu, and X. Liu. A suite of web-based programs to search for transcriptional regulatory motifs. *Nucleic Acids Res*, 32:204–207, 2004.
67. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J Comput*, 25(5):935–948, 1993.
68. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.
69. L. Marsan and M.F. Sagot. Extracting structured motifs using a suffix tree- algorithms and application to promoter concensus identification. *RECOMB*, pp. 210–219, 2000.
70. E.M. McCreight. A space-economical suffix tree construction algorithm. *J Assoc Compu Mach*, 23(2):262–272, 1976.
71. A. McGuire, J. Hughes, and G. Church. Conservation of DNA regulatory motifs and discovery of new motifs in microbial genomes. *Nucleic Acids Res*, 10:744–757, 2000.
72. D. Mehta and S. Sahni. *Handbook of Data Structures and Applications*. Chapman and Hall, New York, 2005.
73. J. Munro, V. Raman, and S.S. Rao. Space efficient suffix trees. *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag, London, UK, 1998, pp. 186–196.
74. B. Phoophakdee and M. Zaki. Genome-scale disk-based suffix tree indexing. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, 2007. ACM, pp. 833–844.
75. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Combin Pattern Matching*, pp. 200–210, 2003.

76. M. Rahman, C. Iliopoulos, and L. Mouchard. Pattern matching in degenerate dna/rna algorithms. *Proceedings of the Workshop on Algorithms and Computation*, pp. 109-120, 2007.

77. K. Roh, M. Crochemore, C. Iliopoulos, and K. Park. External memory algorithms for string problems. *Fundamenta Informaticae*, 84(1):17–32, 2008.

78. D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Mach Learn*, 25:117–149, 1996.

79. D. Roth, J. Hughes, P. Esterp, and G. Church. Finding dna regulatory motifs within unaligned noncoding sequence clustered by whole genome mrna quantitation. *Nat Biotechnol*, 16:939–945, 1998.

80. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J Algorithms*, 48(2):294–313, 2003.

81. S. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. *26th Annual ACM Symposium on the Theory of Computing*, Quebec, Canada, 1994, pp. 300–309.

82. Z. Sun, J. Yang, and J. Deogun. Misae: A new approach for regulatory motif extraction. *Proceedings of the IEEE Computational Systems Bioinformatics Conference*, 2004.

83. M. Tadesse, M. Vannucci, and P. Lio. Identification of dna regulatory motifs using bayesian variable selection suite. *Bioinformatics*, 20:2553–2561, 2004.

84. S. Tata, R. Hankins, and J. Patel. Practical suffix tree construction. *30th International Conference on Very Large Data Bases*, Ontario, Canada, 2004, pp. 36–47.

85. W. Thompson and C. Lawrence. Gibbs recursive sampler: Finding transcription factor binding sites. *Nucleic Acids Res*, 31:3580–3585, 2003.

86. Y. Tian, S. Tata, R. Hankins, and J. Patel. Practical methods for constructing suffix trees. *VLDB J*, 14(3):281–299, 2005.

87. M. Tompa and S. Sinha. A statistical method for finding transcription factor binding sites. *Proceedings Internatioanl Conference on Intelligent Systems in Molecular Biology*, pp. 37–45, 2000.

88. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

89. J.S. Vitter. *Algorithms and Data Structures for External Memory*. Publishers Inc., Hanover, USA, 2008.

90. M. Voracek, B. Melichar, and M. Christodoulakis. Generalized and weighted strings: Repetitions and pattern matching. In *String Algorithmics*, KCL Publications, King's College London, 2004, pp. 225–248.

91. M. Voracek, V. Vagner, and T. Flouri. Indexing degenerate strings. *Proceedings of International Conference on Computational Methods in Science and Engineering*. American Mathematical Institute of Physics, 2007.

92. P. Weiner. Linear pattern matching algorithms. In *14th IEEE Annual Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

93. S. Wu and U. Manber. Fast text searching: Allowing errors. *Commun ACM*, 35(10):83–91, 1992.