Text Indexing (1993; Manber and Myers)

Srinivas Aluru, Iowa State University, vulcan.ee.iastate.edu/~aluru
entry editor: Paolo Ferragina

INDEX TERMS: String algorithms and data structures, string indexing, text indexing.
SYNONYMS: String indexing.

1 PROBLEM DEFINITION

Text or string data naturally arises in many contexts including document processing, information retrieval, natural and computer language processing, and describing molecular sequences. In broad terms, the goal of text indexing is to design methodologies to store text data so as to significantly improve the speed and performance of answering queries. While text indexing has been studied for a long time, it shot into prominence during the last decade due to the ubiquity of web based textual data and search engines to explore it, design of digital libraries for archiving human knowledge, and application of string techniques to further our understanding of modern biology. Text indexing differs from the typical indexing of keys drawn from an underlying total order — text data can have varying lengths, and queries are often more complex and involve substrings, partial matches or approximate matches.

Queries on text data are as varied as the diverse array of applications they support. Consequently, numerous methods for text indexing have been developed and this continues to be an active area of research. Text indexing methods can be classified into two categories: i) methods that are generalizations or adaptations of indexing methods developed for an ordered set of one dimensional keys, and ii) methods that are specifically designed for indexing text data. The most classic query in text processing is to find all occurrences of a pattern P in a given text T (or equivalently, in a given collection of strings). Important and practically useful variants of this problem include finding all occurrences of P subject to at most k mismatches, or at most k insertions/deletions/mismatches. In this entry, we focus on these two basic problems and remark on generalizations of one-dimensional data structures to handle text data.

2 KEY RESULTS

Consider the problem of finding a given pattern P in text T, both strings over alphabet Σ. The case of a collection of strings can be trivially handled by concatenating the strings using a unique end of string symbol, not in Σ, to create text T. It is worth mentioning the special case where T is structured — i.e., T consists of a sequence of words and the pattern P is a word. Consider a total order of characters in Σ. A string (or word) of length k can be viewed as a k-dimensional key and the order on Σ can be naturally extended to lexicographic order between multidimensional keys of variable length. Any one dimensional search data structure that supports $O(\log n)$ search time can be used to index a collection of strings using lexicographic order such that a string of length k can be searched in $O(k \log n)$ time. This can be considerably improved as below [8]:

Theorem 1. *Consider a data structure on one dimensional keys that relies on constant-time comparisons among keys (e.g., binary search trees, red-black trees etc.) and the insertion of a key identifies either its predecessor or successor. Let $O(\mathcal{F}(n))$ be the search time of the data structure storing n keys (e.g., $O(\log n)$ for red-black trees). The data structure can be converted to index n strings using $O(n)$ additional space such that the query for a string s can be performed in $O(\mathcal{F}(n))$ time if s is one of the strings indexed, and in $O(\mathcal{F}(n) + |s|)$ otherwise.*

A more practical technique that provides $O(\mathcal{F}(n) + |s|)$ search time for a string s under more restrictions on the underlying one dimensional data structure is given in [9]. The technique is nevertheless applicable to several classic one dimensional data structures, in particular binary search trees and its balanced variants. For a collection of strings that share long common prefixes such as IP addresses and XML path strings, a faster search method is described in [5].

When answering a sequence of queries, significant savings can be obtained by promoting frequently searched strings so that they are among the first to be encountered in a search path through the indexing data structure. Ciriani *et al.* [4] use self-adjusting skip lists to derive an expected bound for a sequence of queries that matches the information-theoretic lower bound.

Theorem 2. *A collection of n strings of total length N can be indexed in optimal $O(N)$ space so that a sequence of m string queries, say $s_1, \ldots, s_m$, can be performed in $O(\sum_{j=1}^{m} |s_j| + \sum_{i=1}^{n} n_i \log \frac{m}{n_i})$ expected time, where n_i is the number of times the i-th string is queried.*

Notice that the first additive term is a lower bound for reading the input, and the second additive term is a standard information-theoretic lower bound denoting the entropy of the query sequence. Ciriani *et al.* also extended the approach to the external memory model, and to the case of dynamic sets of strings.

We now turn our attention to some of the widely used data structures specifically designed for string data, suffix trees and suffix arrays. These are particularly suitable for querying unstructured text data, such as the genomic sequence of an organism. We use the following notation: Let $s[i]$ denote the i^{th} character of string s, $s[i..j]$ denote the substring $s[i]s[i+1]\ldots s[j]$, and $S_i = s[i]s[i+1]\ldots s[|s|]$ denote the suffix of s starting at i^{th} position. The suffix S_i can be uniquely described by the integer i. In case of multiple strings, the suffix of a string can be described by a tuple consisting of the string number and the starting position of the suffix within the string. Consider a collection of strings over Σ, having total length n, each extended by adding a unique termination symbol $\$ \notin \Sigma$. The suffix tree of the strings is a compacted trie of all suffixes of these extended strings. The suffix array of the strings is the lexicographic sorted order of all suffixes of these extended strings. For convenience, we list '\$', the last suffix of each string, just once. The suffix tree and suffix array of strings 'apple' and 'maple' are shown in Figure 1. Both these data structures take $O(n)$ space and can be constructed in $O(n)$ time [11, 13], both directly and from each other.

Without loss of generality, consider the problem of searching for a pattern P as a substring of a single string T. Assume the suffix tree ST of T is available. If P occurs in T starting from position i, then P is a prefix of suffix $T_i = T[i]T[i+1]\ldots T[|T|]$ in T. It follows that P matches the path from root to leaf labeled i in ST. This property results in the following simple algorithm: Start from the root of ST and follow the path matching characters in P, until P is completely matched or a mismatch occurs. If P is not fully matched, it does

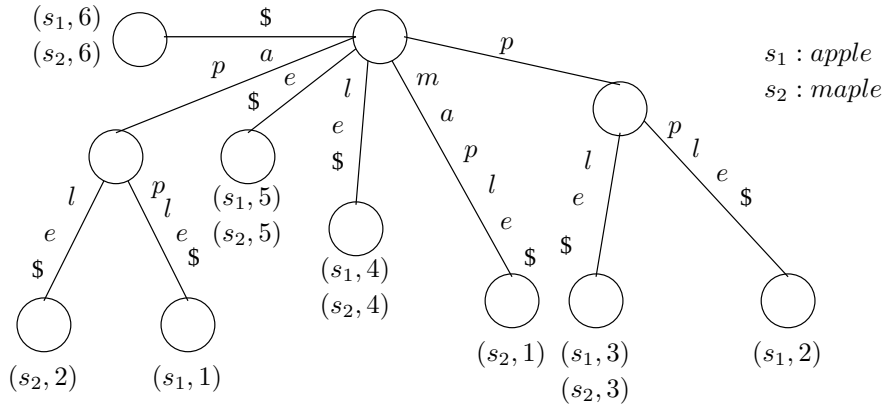

Figure 1: Suffix tree and suffix array of strings *apple* and *maple*.

not occur in T. Otherwise, each leaf in the subtree below the matching position gives an occurrence of P. The positions can be enumerated by traversing the subtree in $O(occ)$ time, where *occ* denotes the number of occurrences of P. If only one occurrence is desired, ST can be preprocessed in $O(|T|)$ time such that each internal node contains the suffix at one of the leaves in its subtree.

Theorem 3. *Given a suffix tree for text T and a pattern P, whether P occurs in T can be answered in $O(|P|)$ time. All occurrences of P in T can be found in $O(|P| + occ)$ time, where occ denotes the number of occurrences.*

Now consider solving the same problem using the suffix array SA of T. All suffixes prefixed by P appear in consecutive positions in SA. These can be found using binary search in SA. Naively performed, this would take $O(|P| * \log |T|)$ time. It can be improved to $O(|P| + \log |T|)$ time as follows [14]:

Let $SA[L..R]$ denote the range in the suffix array where the binary search is focused. To begin with, $L = 1$ and $R = |T|$. Let $\prec$ denote "lexicographically smaller", $\preceq$ denote "lexicographically smaller or equal", and $lcp(\alpha, \beta)$ denote the length of the longest common prefix between strings α and β. At the beginning of an iteration, $T_{SA[L]} \preceq P \preceq T_{SA[R]}$. Let $M = \lceil \frac{L+R}{2} \rceil$. Let $l = lcp(P, T_{SA[L]})$ and $r = lcp(P, T_{SA[R]})$. Because SA is lexicographically ordered, $lcp(P, T_{SA[M]}) \geq min(l, r)$. If $l = r$, then compare P and $T_{SA[M]}$ starting from the $(l+1)^{th}$ character. If $l \neq r$, consider the case when $l > r$.

> Case I: $l < lcp(T_{SA[L]}, T_{SA[M]})$. In this case, $T_{SA[M]} \prec P$ and $lcp(P, T_{SA[M]}) = lcp(P, T_{SA[L]})$. Continue search in $SA[M..R]$. No character comparisons required.
>
> Case II: $l > lcp(T_{SA[L]}, T_{SA[M]})$. In this case, $P \prec T_{SA[M]}$ and $lcp(P, T_{SA[M]}) = lcp(T_{SA[L]}, T_{SA[M]})$. Continue search in $SA[L..M]$. No character comparisons required.
>
> Case III: $l = lcp(T_{SA[L]}, T_{SA[M]})$. In this case, $lcp(P, T_{SA[M]}) \geq l$. Compare P and $T_{SA[M]}$ beyond l^{th} character to determine their relative order and *lcp*.

Similarly, the case when $r > l$ can be handled such that comparisons between P and $T_{SA[M]}$, if at all needed, start from $(r+1)^{th}$ character. To start the execution of the algorithm, $lcp(P, T_{SA[1]})$ and $lcp(P, T_{SA[|T|]})$ are computed directly using at most $2|P|$ character

comparisons. It remains to be described how the $lcp(T_{SA[L]}, T_{SA[M]})$ and $lcp(T_{SA[R]}, T_{SA[M]})$ values required in each iteration are computed. Let $Lcp[1 \ldots |T| - 1]$ be an array such that $Lcp[i] = lcp(SA[i], SA[i+1])$. The *Lcp* array can be computed from SA in $O(|T|)$ time [12]. For any $1 \leq i < j \leq n$, $lcp(T_{SA[i]}, T_{SA[j]}) = min_{k=i}^{j-1} Lcp[k]$. In order to find the *lcp* values required by the algorithm in constant time, note that the binary search can be viewed as traversing a path in the binary tree corresponding to all possible search intervals used by any execution of the binary search algorithm [14]. The root of the tree denotes the interval $[1..n]$. If $[i..j]$ $(j - i \geq 2)$ is the interval at an internal node of the tree, its left child is given by $[i..\lceil \frac{i+j}{2} \rceil]$ and its right child is given by $[\lceil \frac{i+j}{2} \rceil..j]$. The *lcp* value for each interval in the tree is precomputed and recorded in $O(n)$ time and space.

Theorem 4. *Given the suffix array SA of text T and a pattern P, the existence of P in T can be checked in $O(|P| + \log |T|)$ time. All occurrences of P in T can be found in $O(occ)$ additional time, where occ denotes their number.*

PROOF: The algorithm makes at most $2|P|$ comparisons in determining $lcp(P, T_{SA[1]})$ and $lcp(P, T_{SA[n]})$. A comparison made in an iteration to determine $lcp(P, T_{SA[M]})$ is categorized *successful* if it contributes the *lcp*, and categorized *failed* otherwise. There is at most one failed comparison per iteration. As for successful comparisons, note that the comparisons start with $(max(l, r) + 1)^{th}$ character of P, and each successful comparison increases the value of $max(l, r)$ for next iteration. Thus, each character of P is involved only once in a successful comparison. The total number of character comparisons is at most $3|P| + \log |T| = O(|P| + \log |T|)$. ■

Abouelhoda *et al.* [1] reduce this time further to $O(|P|)$ by mimicking the suffix tree algorithm on a suffix array with some auxiliary information. The strategy is useful in other applications based on top-down traversal of suffix trees. At this stage, the distinction between suffix trees and suffix arrays is blurred as the auxiliary information stored makes the combined data structure equivalent to a suffix tree. Using clever implementation techniques, the space is reduced to approximately $6n$ bytes. A major advantage of the suffix tree and suffix array based methods is that the text T is often large and relatively static, while it is queried with several short patterns. With suffix trees and enhanced suffix arrays [1], once the text is preprocessed in $O(|T|)$ time, each pattern can be queried in $O(|P|)$ time for constant size alphabet. For large alphabets, the query can be answered in $O(|P| * \log |\Sigma|)$ time using $O(n|\Sigma|)$ space (by storing an ordered array of $|\Sigma|$ pointers to potential children of a node), or in $O(|P| * |\Sigma|)$ time using $O(n)$ space (by storing pointers to first child and next sibling).[1] For indexing in various text-dynamic situations, see [3, 7] and references therein. The problem of compressing suffix trees and arrays is covered in more detail in other entries.

While exact pattern matching has many useful applications, the need for approximate pattern matching arises in several contexts ranging from information retrieval to finding evolutionary related biomolecular sequences. The classic approximate pattern matching problem is to find substrings in the text T that have an edit distance of k or less to the pattern P, i.e., the substring can be converted to P with at most k insert/delete/substitute operations. This problem is covered in more detail in other entries. Also see [15], the references therein, and Chapter 36 of [2].

[1] Recently, Cole *et al.* (2006) showed how to further reduce the search time to $O(|P| + \log |\Sigma|)$ while still keeping the optimal $O(|T|)$ space.

3 APPLICATIONS

Text indexing has many practical applications — finding words or phrases in documents under preparation, searching text for information retrieval from digital libraries, searching distributed text resources such as the web, processing XML path strings, searching for longest matching prefixes among IP addresses for internet routing, to name just a few. The reader interested in further exploring text indexing is referred to the book by Crochemore and Rytter [6], and to other entries in this Encyclopedia. The last decade of explosive growth in computational biology is aided by the application of string processing techniques to DNA and protein sequence data. String indexing and aggregate queries to uncover mutual relationships between strings are at the heart of important scientific challenges such as sequencing genomes and inferring evolutionary relationships. For an in depth study of such techniques, the reader is referred to Parts I and II of [10] and Parts II and VIII of [2].

4 OPEN PROBLEMS

Text indexing is a fertile research area, making it impossible to cover many of the research results or actively pursued open problems in a short amount of space. Providing better algorithms and data structures to answer a flow of string-search queries when caches or other query models are taken into account, is an interesting research issue [4].

5 CROSS REFERENCES

2D-Pattern Indexing, Compressed Text Indexing, Compressed Suffix Array, Indexed Approximate String Matching, String Indexing in Hierarchical Memory, Suffix Array Construction, Suffix Tree Construction in Hierarchical Memory, and Suffix Tree Construction in RAM.

6 RECOMMENDED READING

[1] M. Abouelhoda, S. Kurtz, and E. Ohlebusch, *Replacing suffix trees with enhanced suffix arrays*, Journal of Discrete Algorithms, 2 (2004), pp. 53–86.

[2] S. Aluru, ed., *Handbook of Computational Molecular Biology*, Chapman and Hall/CRC Press Computer and Information Science Series, Boca Raton, FL, 2005.

[3] A. Amir, T. Kopelowitz, M. Lewenstein, and N. Lewenstein, *Towards real-time suffix tree construction*, in Proc. String Processing and Information Retrieval Symposium, 2005, pp. 67–78.

[4] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan, *A data structure for a sequence of string acesses in external memory*, ACM Transactions on Algorithms, 3 (2007).

[5] P. Crescenzi, R. Grossi, and G. Italiano, *Search data structures for skewed strings*, in International Workshop on Experimental and Efficient Algorithms (WEA), Springer Lecture Notes in Computer Science, vol. 2, 2003, pp. 81–96.

[6] M. Crochemore and W. Rytter, *Jewels of Stringology*, World Scientific Publishing Company, Singapore, 2002.

[7] P. Ferragina and R. Grossi, *Optimal On-Line Search and Sublinear Time Update in String Matching*, SIAM Journal on Computing, 3 (1998), 713-736.

[8] G. Franceschini and R. Grossi, *A general technique for managing strings in comparison-driven data structures*, in Annual International Colloquium on Automata, Languages and Programming (ICALP), 2004.

[9] R. Grossi and G. Italiano, *Efficient techniques for maintaining multidimensional keys in linked data structures*, in Annual International Colloquium on Automata, Languages and Programming (ICALP), 1999, pp. 372–381.

[10] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, New York, 1997.

[11] J. Karkkainen, P. Sanders, and S. Burkhardt, *Linear work suffix arrays construction*, Journal of the ACM, 53 (2006), pp. 918–936.

[12] T. Kasai, G. Lee, H. Arimura, and S. A. *et al.* , *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. 12th Annual Symposium, Combinatorial Pattern Matching, 2001, pp. 181–192.

[13] P. Ko and S. Aluru, *Space efficient linear time construction of suffix arrays*, Journal of Discrete Algorithms, 3 (2005), pp. 143–156.

[14] U. Manber and G. Myers, *Suffix arrays: a new method for on-line search*, SIAM Journal on Computing, 22 (1993), pp. 935–948.

[15] G. Navarro, *A guided tour to approximate string matching*, ACM Computing Surveys, 33 (2001), pp. 31–88.