String Mining in Bioinformatics

Mohamed Abouelhoda and Moustafa Ghanem

1 Introduction

Sequence analysis is a major area in bioinformatics encompassing the methods and techniques for studying the biological sequences, DNA, RNA, and proteins, on the linear structure level. The focus of this area is generally on the identification of intra- and inter-molecular similarities. Identifying intra-molecular similarities boils down to detecting repeated segments within a given sequence, while identifying inter-molecular similarities amounts to spotting common segments among two or multiple sequences.

From a data mining point of view, sequence analysis is nothing but *string-* or *pattern mining* specific to biological strings. For a long time, this point of view, however, has not been explicitly embraced neither in the data mining nor in the sequence analysis text books, which may be attributed to the co-evolution of the two apparently independent fields. In other words, although the word "data-mining" is almost missing in the sequence analysis literature, its basic concepts have been implicitly applied. Interestingly, recent research in biological sequence analysis introduced efficient solutions to many problems in data mining, such as querying and analyzing time series [49, 53], extracting information from web pages [20], fighting spam mails [50], detecting plagiarism [22], and spotting duplications in software systems [14].

In this chapter, we review the basic problems in the area of biological sequence analysis. We present a taxonomy of the main problems in this area and introduce basic solutions to them. Moreover, we show some interesting applications of the string data structures to some traditional problems in data mining, such as finding frequent itemsets, computing string kernels, and mining semi- and unstructured text documents.

M. Abouelhoda (✉)
Cairo University, Orman, Gamaa Street, 12613 Al Jizah, Giza, Egypt
Nile University, Cairo-Alex Desert Rd, Cairo 12677, Egypt

M.M. Gaber (ed.), *Scientific Data Mining and Knowledge Discovery: Principles and Foundations*, DOI 10.1007/978-3-642-02788-8_9,

2 Background

The three key types of biological sequences of interest to bioinformatics are DNA sequences, protein sequences and RNA sequences. A DNA sequence, e.g., GTAAACTGGTAC..., is a string formed from an alphabet of four letters (A, C, G, and T), each representing one of four different nucleotides. Using the genetic code, a DNA sequence can be translated into a corresponding protein sequence, whereby each triplet of nucleotides (letters) codes for one amino acid, e.g., the triplet GTA translates into the amino acid Valine represented by V, and the triplet AAC translates into the amino acid Asparagine represented by N, etc. A protein sequence, e.g., VNWYHLDKLMNEFF..., is thus also a string formed from an alphabet of twenty characters each representing an amino acid, these are (A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, and V). Similarly, RNA sequences are formed from a four-letter alphabet of nucleotides (A, C, G, and U).

Knowing the sequence of letters of a DNA, an RNA or a protein is not an ultimate goal in itself. Rather, the major task is to *understand* the sequence, in terms of its structure and biological function. This is typically achieved first by identifying individual regions or structural units within each sequence and then assigning a function to each structural unit. Overall, two broad strategies are used when identifying and understanding the features of a biological sequence; *intrinsic strategy* in which just a single sequence is studied and *comparative strategy* in which the sequence is compared to other sequences, in an attempt to infer how it relates to similar structural and functional units in them.

Intrinsic Strategy

Various methods within the intrinsic strategy can be used for the analysis of genomic (DNA) sequences to identify the roles that different segments (substrings) of the sequence play. Such analysis includes identifying sequence features, such as the G+C content of the different regions, and segmenting the genome into its key structural units, such as genes, repeats, and regulatory elements. In simple cases, identifying the different segments can be achieved by searching for known patterns of substrings that mark these segments. An example is searching for the patterns signifying the *promoter* regions that are essential in the biological transcription process.

In more complex cases, the use of probabilistic predictive methods is needed. These predictive methods take into account different types of evidence that are associated with the region to be identified (e.g., a gene), and are typically based on statistical properties. These predictive methods are typically known as ab-initio methods, (see for example [19] for an evaluation of gene prediction methods).

Similarly, various intrinsic analysis methods can be used for the analysis of protein sequences. In simple cases, the analyses include identifying the basic structural and functional subsequences of the protein, referred to as *domains* and *motifs*, respectively. The analyses can also include the use of statistical methods to predict the 3-D structure and/or the function of the protein (see for example [35] for the use of Hidden Markov Models for the prediction of protein structure).

Comparative Strategy

Comparative analysis methods are used over both DNA and protein sequences to study the relationship of genome structure and function across different biological species or strains. This is important not only to study phylogeny, or how different organisms may have evolved, but also as a means to understand the roles and functions of newly discovered genes and proteins. Given either a DNA or protein sequence, a typical recurring task in bioinformatics is to search known genomic and proteomics databanks for similar sequences. Many of the sequences stored in such databanks have been manually curated and annotated by experts over the years, and their properties and functions were experimentally verified. The identification and study of such similar sequences can provide many valuable clues relating to the sequence under examination.

A number of issues arise when conducting similarity searches over large biological sequence databases. The first relates to quantifying the measure of similarity, or conversely distance, between two sequences. The simplest approach is based on using a Hamming distance measure that counts the number of mismatches between two strings. As an example the Hamming distance between the two sequences SSHLDKLMNEFF and HSHLKLLMKEFF is four, as shown below where an * is used to signify the positions of the mismatches.

```
SSHLDKLMNEFF
*    **   *
HSHLKLLMKEFF
```

However, since typically two sequences being compared are unlikely to be of the same length, a simple sequence similarity algorithm needs to introduce gaps corresponding to insertions/deletions within the sequences. Comparing sequences with gaps can be shown in the example below:

```
MHHNALQRRTVWVNAY
MHH-ALQRRTVWVNAY
```

A second issue that arises when comparing the similarity of biological sequences is that the simple Hamming distance metric does not take into account the likelihood of one amino acid (or nucleotide in case of DNA) changing to another one due to mutations. Although some amino acid substitutions are disastrous and do not survive evolution, others have almost no effect because the two amino acids are chemically quite similar in terms of their properties. Such issues are typically addressed using a scoring scheme, such as Percent Accepted Mutation (PAM) and Block Substitution Matrices (BLOSUM) used in the widely used BLAST sequence comparison algorithm [10].

Taking into consideration the possibilities for *indels* (Insertions or Deletions) as well as the likelihood of amino acid (nucleotide) substitutions leads to a wide spectrum of string mining problems that the bioinformatics community has been addressing over the past two decades.

Data Structures and Algorithms for String Mining

In this chapter, we focus on string mining problems used in both intrinsic and comparative sequence analysis strategies.

In Sect. 3, we start by introducing some basic formal definitions and data structures that are typically used. These basic data structures include Look-up tables, Automata and Tries, and the Suffix Tree data structure. These data structures are key to the efficient execution of many algorithms over large collections of long sequences.

In Sect. 4, we present a taxonomy of the major repeat-related problems in bioinformatics, and describe the basic exact and approximate string mining algorithms algorithms used to address these problems. The problems considered include finding dispersed fixed length and maximal length repeats, finding tandem repeats, and finding unique subsequences and missing (un-spelled) subsequences.

In Sect. 5, we provide a taxonomy of the key algorithms for sequence comparison, including global, semi-global and local sequence alignment and biological database search methods. We also describe the basic algorithms used in each case covering exact and approximate algorithms for both fixed- and variable-length sequences, thus covering most popular sequence comparison algorithms in a unified framework.

In Sect. 6, we describe how the basic algorithms presented in this chapter can be easily applied to string mining applications beyond bioinformatics. These include applications in frequent itemset mining, string kernels as well as information extraction applications for free text and semi-structured data sources.

We note that the prediction of biological sequence features based on statistical inference or machine learning (as in the case of ab-initio gene identification), are not addressed directly in this chapter. However, the key data structures presented in this chapter provide the basis for calculating, in an efficient manner, the statistics required by such algorithms.

3 Basic Definitions and Data Structures

3.1 Basic Notions

Let Σ denote an ordered alphabet and let $|\Sigma|$ denote the size of this alphabet. Let S be a string over Σ of length $|S| = n$. We write $S[i]$ to denote the character at position i in S, for $0 \leq i < n$. For $i \leq j$, $S[i..j]$ denotes the *substring* S starting with the character at position i and ending with the character at position j. The substring $S[i..j]$ is also denoted by the *pair* (i, j) *of positions*. The substring $S[0..i]$ is a prefix of S. The substring $S[i..n-1]$ is the i-th suffix of S, and it is denoted by $S(i)$.

As introduced in Sect. 2, the three types of biological sequences: DNAs, RNAs, and proteins differ in their alphabets. For example, for DNA sequences, the DNA sequence is a string over an alphabet of four characters, namely A, C, G, and T. We also note that in Computer Science terminology, the notion "sequence" is not the same as "string." However, "sequence" in Biology corresponds actually to "string" in Computer Science. So, unless otherwise stated, we will use "sequence" and "string" interchangeably in this chapter.

The *complement* string of a DNA string is obtained, from biological knowledge, by replacing every C with G, G with C, T with A, and A with T; i.e., this replacement follows the base pairing rule. The *reverse* string of $S[1..n]$ is the string $S[n]S[n-1]..S[1]$. The *reverse complement* of S is the complement string of $S[n]S[n-1]..S[1]$. Neither the reverse nor the complement string is of interest in molecular biology. The reverse complement, however, is as important as the DNA string S because it represents the complementary strand of this molecule, where biological information is also encoded. Using the genetic code, any DNA or RNA sequence that encodes for a protein can be transformed to the corresponding sequence of amino acids.

3.2 Look-Up Table

For a given parameter d, the *look-up table* is a table of length $|\Sigma|^d$ such that each string of length d maps to just one entry in the table. Note that the table size corresponds to the $|\Sigma|^d$ possible strings. We call any of these strings *key*. Each entry of the table points to a linked list storing the positions where the associated string occurs in S. Note that some entries may be empty, as the respective substring does not occur in S. The mapping function, for a string $P[0..d-1]$, is

$$x = f(P[0..d-1]) = \sum_{i=0}^{d-1} |\Sigma|^{d-i-1} T(P[i])$$

where x is an integer value representing an index of the look-up table, Σ is the alphabet size, and $T(S[i])$ is a function that maps each character in the alphabet to a number in the range $[0 \ldots |\Sigma|]$. Figure 1 shows an example of a look-up table with the mapping $a \to 0$, $c \to 1$, $g \to 2$, and $t \to 3$.

The look-up table can be naively constructed by sliding a window of length d over S and computing the mapping function $f(w)$ for each substring $w \in S$. This takes $O(d|S|)$ time. This time complexity, however, can be reduced to $O(|S|)$ using the following recurrence, which can be implemented by scanning the string S from left to right.

$$f(S[i+1..i+d]) = (f(S[i..i+d-1]) - T(S[i])|\Sigma|^{d-1}) \times |\Sigma| + T(S[i+d-1])$$

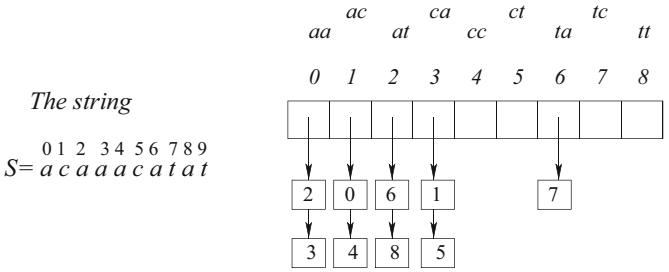


Fig. 1 The look-up table for $S = acaaacatat$

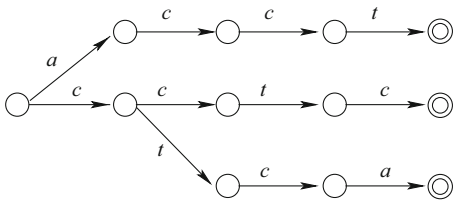


Fig. 2 A trie of the string *acctca*. The keys are the substrings of length 4

3.3 Automata and Tries

Aho and Corasick [9] showed how to construct a deterministic finite automata for a set of keys (words). The target was to improve online string matching for bibliographic search. For a genomic sequence, the keys can be taken as the set of its subsequences (more precisely, its substrings). To reduce the space, one can take subsequences up to certain length k. The locations where these substrings occur in the sequence can be attached to the leaves (the nodes of success) of the automaton. Figure 2 shows an example of an automaton for the string *acctca*, where $k = 4$. The string has the three keys *acct*, *cctc*, and *ctca*.

It is interesting to see that this automaton is nothing but a *pruned non-compact suffix tree*. (Suffix tree is presented in the next subsection.) A non-compact suffix tree, known as *trie*, has the edge labels explicitly stored with the tree, and for each label there is an edge. (As a consequence not all the nodes of the tree are branching.) Because the space consumption of a trie grows quadratically with the sequence length, it is necessary to prune it at an appropriate depth.

For maximum depth k, the automata can be constructed in $O(nk)$ time by adding the n keys successively to the automata as the original method of Aho and Corasick [9]. But this time complexity can be improved by constructing the suffix tree in $O(n)$ time, and pruning it such that the maximum depth is k characters from the root.

3.4 The Suffix Tree

The suffix tree is a focal data structure in string processing and the linear time algorithm for constructing it is a milestone in this area. The suffix tree is defined as follows: Let S be a string of n characters and let the character \$ be appended to S. A *suffix tree* for the string $S\$$ is a rooted directed tree with exactly $n + 1$ leaves numbered from 0 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of $S\$$. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the substring $S[i..n-1]\$$ that denotes the i-th nonempty suffix of the string $S\$$, $0 \leq i \leq n$. Moreover, the concatenation of the edge-labels on the path from the root to a non-leaf node spells out a substring of S that occurs z times in S, where z is the number of all the leaves under the subtree of this non-leaf node. Figure 3 shows the suffix tree for the string $S = acaaacatat\$$. The character \$ is known as the sentinel character and it is appended to obtain a tree in which every suffix corresponds to a leaf. More precisely, without the sentinel character some suffixes would be proper prefixes of other suffixes. For example, assume S to be aa, then without the sentinel character the second suffix is a prefix of the first and it will not be straightforward to have a leaf for the second suffix and to distinguish (lexicographically sort) the two suffixes of S. Throughout this chapter, we use the notion *suffix tree of* S but this implicitly means that the suffix tree is constructed for $S\$$; i.e., the sentinel character \$ is already appended to S.

To store the suffix tree in linear space, the edge-labels are not explicitly stored with the edges. Rather, for each edge a pair of positions (i, j) is attached to represent

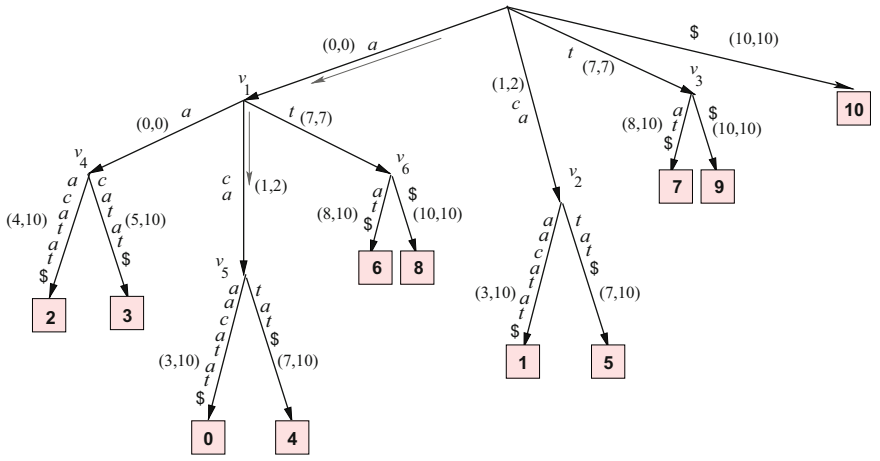


Fig. 3 The suffix tree for $S = acaaacatat\$$. The sentinel character is appended to S. A pair (i, j) represents a substring $S[i..j]$ corresponding to the respective edge-labels. The gray arrows indicates to the edges traversed while searching for the pattern ac

a substring $S[i..j]$ corresponding to the respective edge-labels. This suffix tree representation is commonly referred to as *compact suffix tree*. (If the labels are explicitly attached to the edges, the structure is called non-compact suffix tree or suffix trie. In this case, the space complexity becomes $O(n^2)$; consider, e.g., a string like *abcde*.)

The suffix tree can be constructed in $O(n)$ time and space [45, 60]; see also [28] for a simplified exposition. Once constructed, it can be used to efficiently solve the problems specified in Fig. 5, but it can also be used to solve other string processing applications; see [12, 28].

It is worth mentioning that the space consumption and the poor cache performance of the suffix tree is a bottleneck for large scale applications [25, 33, 42]. The enhanced suffix array [1] represents an alternative data structure to the suffix tree that requires less space and achieves better cache performance. In [1] it was shown that any algorithm using the suffix tree can be systematically replaced with an equivalent one using the enhanced suffix array. Nevertheless, for clarity of presentation, we will describe the algorithms in this chapter over the suffix tree. For mapping the algorithms to the enhanced suffix array, we refer the reader to [1, 7].

4 Repeat-Related Problems

A pair of substrings $R = ((i_1, j_1), (i_2, j_2))$ is a *repeated pair* if and only if $(i_1, j_1) \neq (i_2, j_2)$ and $S[i_1..j_1] = S[i_2..j_2]$. The length of R is $j_1 - i_1 + 1$. A repeated pair $((i_1, j_1), (i_2, j_2))$ is called *left maximal* if $S[i_1 - 1] \neq S[i_2 - 1]$ and *right maximal* if $S[j_1 + 1] \neq S[j_2 + 1]$. A repeated pair is called *maximal* if it is both left and right maximal. A substring ω of S is a (*maximal*) *repeat* if there is a (maximal) repeated pair $((i_1, j_1), (i_2, j_2))$ such that $\omega = S[i_1..j_1]$. A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat. Figure 4 shows maximal repeated pairs and supermaximal repeats of the string $S = gagctagagcg$.

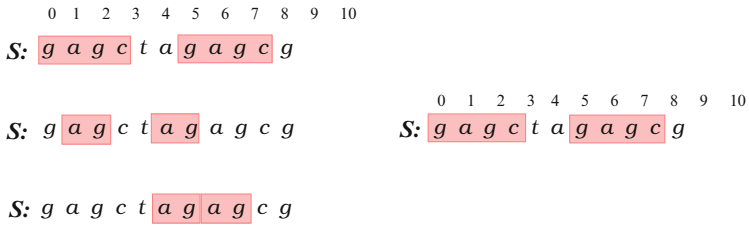


Fig. 4 Repeats of the string $S = gagctagagcg$. *Left*: three maximal repeated pairs of minimum length two. The pairs are $((0, 3), (6, 9))$, $((1, 2), (5, 6))$, and $((5, 6), (7, 8))$. *Right*: one supermaximal repeat of the substrings $(0, 3)$ and $(6, 9)$. The repeat ag is not supermaximal because it is not maximal and contained in $gagc$

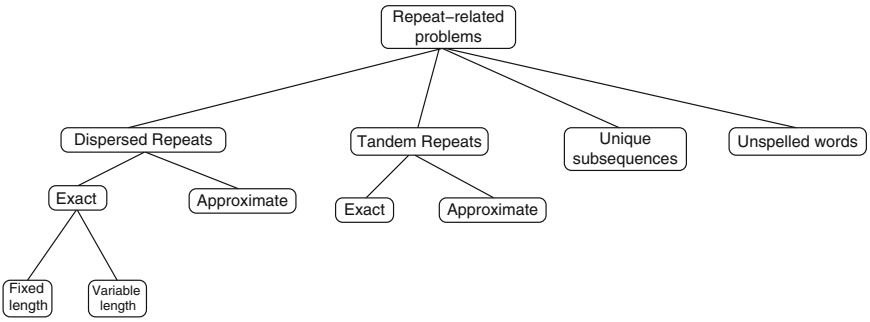


Fig. 5 Taxonomy of the major repeat-related problems in bioinformatics

If the repeated segments are occurring adjacent to each other, then we speak of tandem repeats. Formally, a substring of S is a *tandem repeat* if it can be written as $\omega\omega$ for some nonempty string ω. For example, the subtring $agag$ of the string $S = gagctagagcg$ is a tandem repeat, where $\omega = ag$.

Figure 5 is an overview of the basic string processing problems associated with repeat analysis. For biological sequences, approximate repeats (tandem or interspersed) are the ones occurring in reality. However, identification of exact repeats is also important because many algorithms that find approximate repeats are usually based on exact repeats. In this section, we will show how to solve the problems given in the figure.

4.1 Identifying Dispersed Repeats

Identifying Exact Fixed Length Repeats

The focus of this subsection is on the identification of repeats or repeated pairs of fixed length k, i.e., we find all repeats $\omega = S[i_1..j_1]$ such that $j_1 - i_1 + 1 = k$. Fixed length repeats of length k can be efficiently found using either the look-up table or the suffix tree. But to show the benefits of these data structures, we first briefly address a brute force method.

A brute force method to enumerate all repeats is to take each substring of S of length k and search whether it occurs somewhere else in S by scanning S from left to right. Because we have $O(n)$ substrings and each scan takes $O(nk)$ time (including character comparison), the brute-force method takes $O(kn^2)$ time. Note that by using a more advanced exact pattern matching algorithm (as will be discussed in Sect. 5.1), the running time of the scanning phase can be reduced to $O(k + n)$, which reduces the running time to $(n^2 + kn)$. But this is still quadratic and it is not so helpful.

The look-up table provides a better solution to this problem. If it is built for all substrings of length $d = k$ of S, then it becomes straightforward to extract all the repeats, as follows. The look-up table is sequentially scanned. If the linked

list attached to a cell of the table contains more than one element, then the positions where it starts in the string are reported. Repeated pairs can be computed by applying a Cartesian product operation over the linked list elements (excluding identical pairs). For example, if a k-length substring is repeated three time in S at positions 3, 7, and 20, then the repeated pairs are $((3, 3 + k - 1), (7, 7 + k - 1))$, $((3, 3+k-1), (20, 20+k-1))$, and $((7, 7+k-1), (20, 20+k-1))$. We do not need to report the pairs $((7, 7 + k - 1), (3, 3 + k - 1))$, $((20, 20 + k - 1), (3, 3 + k - 1))$, and $((20, 20 + k - 1), (7, 7 + k - 1))$, as well, because they are redundant.

Constructing the look-up table takes $O(|\Sigma|^d + n)$ time and $O(|\Sigma|^d)$ space. Finding the repeats takes $O(|\Sigma|^d + z)$ time, where z is the number of repeats. (This algorithm is said to be output sensitive, because the complexity is expressed in terms of the output size). Finding the repeated pairs takes $O(|\Sigma|^d + z')$ time, where z' is their number. It is not difficult to see that the space consumption of the look-up table becomes prohibitive for large d and Σ.

The automaton of Sect. 3.3 can also be used to report repeats of fixed length k. The idea is that the keys of the automaton are all the substrings of length k. If one leaf contains more than one position, these positions, which are the occurrences of a repeat, are reported. To find repeated pairs under this leaf, a Cartesian product operation is applied over the occurrences of the repeat (excluding identical pairs).

The suffix tree presents a more efficient solution to this problem. Recall from Sect. 3.4 that the concatenation of the edge-labels on the path from the root to a non-leaf node spells out a substring of S that occurs z times in S, where z is the number of all the leaves under the subtree of this non-leaf node. If each internal node is annotated with the length ℓ of the substring made up of the edge labels from the root to this node, then for each node with $\ell \geq k$ we report all the positions stored in the leaves of its subtree, as the starting positions of a repeated substring associated with this node. For example, if we use the suffix tree of Fig. 3 to find repeats of length two, then the nodes v_2, v_4, v_5, and v_6 are the ones with $\ell \geq 2$. For v_6, e.g., we report the positions 6 and 8 as the starting positions of the repeated string at, and compose the repeated pair $((6, 7), (8, 9))$.

Finding Variable Length Repeats

Maximal Repeated Pairs

The problem of reporting exact repeats of any length are referred to here as the problem of computing exact variable length repeats. The class of *maximal repeats* belongs to the set of variable length repeats. Recall from the introductory part of this section that a repeated pair $((i_1, j_1), (i_2, j_2))$ is called *maximal* if $S[i_1 - 1] \neq S[i_2 - 1]$ and $S[j_1 + 1] \neq S[j_2 + 1]$. That is, the repeated pair cannot extend to the left and to the right simultaneously in S. Recall also that a substring ω of S is a (*maximal*) *repeat* if there is a (maximal) repeated pair $((i_1, j_1), (i_2, j_2))$ such that $\omega = S[i_1..j_1]$. Fig. 4 (left) shows maximal repeated pairs of the string $S = gagctagagcg$.

To compute maximal repeated pairs, we use the algorithm of Gusfield based on the suffix tree [28, page 147]. This algorithm computes maximal repeated pairs of a sequence S of length n in $O(|\Sigma|n + z)$ time, where z is the number of maximal repeated pairs. To the best of our knowledge, Gusfield's algorithm was first used in the bioinformatics practice in the program *REPuter* [42], where maximal repeated pairs are used as seeds for finding approximate repeats, as we will discuss later when handling approximate dispersed repeats.

The algorithm for computing maximal repeated pairs is based on a bottom-up traversal of the suffix tree, in which all the child intervals are processed before the parent interval.

To explain the algorithm, we first introduce some notations: Let $\perp$ denote an undefined character, corresponding to a non-existing character before $S[0]$. We assume that this character is different from all characters in Σ. Let v be a node in the suffix tree and let ℓ_v be the length of the substring produced by concatenating the edge labels from the root to node v. Let u denote this substring. Define $\mathcal{P}_v$ to be the set of positions p such that u is a prefix of $S(p)$, i.e., $\mathcal{P}_v$ contains the positions attached to the leaves (suffixes) of the subtree at node v. We divide $\mathcal{P}_v$ into disjoint and possibly empty sets according to the characters to the left of each position: For any $a \in \Sigma \cup \{\perp\}$ define

$$\mathcal{P}_v(a) = \begin{cases} \{0 \mid 0 \in \mathcal{P}_v\} & \text{if } a = \perp \\ \{p \mid p \in \mathcal{P}_v, p > 0, \text{ and } S[p-1] = a\} & \text{otherwise} \end{cases}$$

Figure 6 shows an example of position sets associated to a suffix tree.

The set $\mathcal{P}_v$ is constructed during the bottom-up traversal as follows. If v is a leaf node of the suffix tree of S referring to the suffix $p = S(i)$. Then $\mathcal{P}_v = \{p\}$ and $\mathcal{P}_v(a) = p$ if $a = S[p-1]$. If v is not a leaf, then for each $a \in \Sigma \cup \{\perp\}$, $\mathcal{P}_v(a)$ is computed step by step while processing the children of v. Let $v_1, .., v_t$ denote the children of v. We start with v_2 and perform two operations:

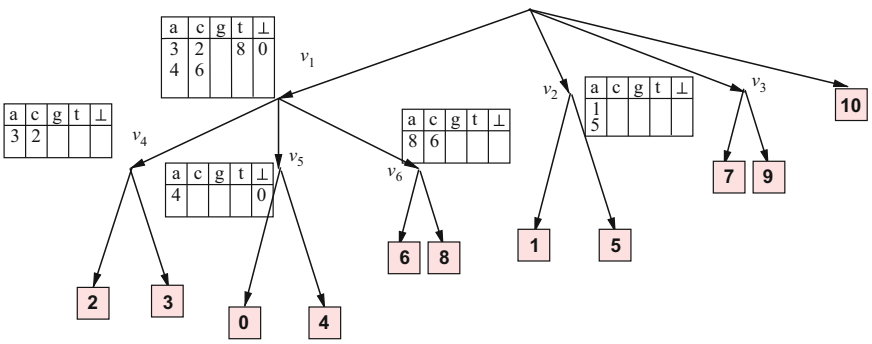


Fig. 6 The position sets of a sub-tree of the suffix tree of Fig. 3. We show these sets together just for illustration, but during the bottom up traversal the sets at v_4, v_5 and v_6, for example, coalesce together into the set at v_1

1. A Cartesian product between the sets $\mathcal{P}_{v_1}$ and $\mathcal{P}_{v_2}$. More precisely, maximal repeated pairs are output by combining the position sets $\mathcal{P}_{v_1}(a)$ with $\mathcal{P}_{v_2}(b)$, for each $a \neq b$. More precisely, a maximal repeated pair $((p, p + \ell_v - 1), (p', p' + \ell_v - 1))$, $p < p'$, is output for all $p \in \mathcal{P}_{v_1}(a)$ and $p' \in \mathcal{P}_{v_2}(b)$.
2. A union operation between $\mathcal{P}_{v_1}$ and $\mathcal{P}_{v_2}$ is performed to construct the position set $\mathcal{P}^{v_1..v_2}$, where $\mathcal{P}^{v_1..v_2}(a) = \mathcal{P}_{v_1}(a) \cup \mathcal{P}_{v_1}(a)$, for all $a \in \perp \cup \Sigma$.

Then we take v_3 and process it with $\mathcal{P}^{v_1..v_2}(a)$ to report *MEMs* and produce $\mathcal{P}^{v_1..v_3}(a)$, and so on until we process all the child intervals. The final set $\mathcal{P}^{v_1..v_t}$ is $\mathcal{P}_v$; see Fig. 6.

There are two operations performed when processing a node v: First, output of maximal repeated pairs by combining position sets. Second, union of position sets. The first step takes $O(z)$ time, where z is the number of repeats. The union operation for the position sets can be implemented in constant time, if we use linked lists. Altogether, the algorithm runs in $O(n + z)$ time. This algorithm requires linear space. This is because a position set $\mathcal{P}_v(a)$ is the union of position sets of the child intervals of v, and it is not required to copy position sets; rather they are linked together.

In practice, it is usually required to report maximal repeated pairs of length larger than a certain user-defined threshold. This reduces the output size while maintaining meaningful results. This length constraint can be incorporated in the above algorithm by examining for each internal node v the value ℓ_v. If ℓ_v falls below the given threshold, then the attached position set is discarded and one starts from the next non-visited leaf.

Identifying Supermaximal Repeats

A sub-class of the maximal repeats is the *supermaximal repeats*. A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat. Fig. 4 (right) shows supermaximal repeats of the string $S = gagctagagcg$.

We use the algorithm of [1] to compute supermaximal repeats. This algorithm was originally introduced over the enhanced suffix array but we will explain it here over the suffix tree. Let u_v denote the substring obtained by concatenating the characters on the path from the root to a node v. We call a node v *terminal node*, if all its children are leaves. To each leaf $S(p)$, we attach the character a such that $S[p - 1] = a$. We call this character the *burrows-wheeler* character, analogous to the burrows-wheeler transformation used in text compression.

Because any supermaximal repeat ω is right maximal, there should be a node v in the suffix tree such that $u_v = \omega$. Moreover, because ω does not occur as a substring of any other supermaximal repeat, the node v must be a terminal node and the burrows-wheeler characters of the leaves under v are pairwise distinct. The idea is that if these characters are not pairwise distinct, then there must be another repeat containing u_v as a substring. This suggests the following simple algorithm: Traverse each terminal node v of the suffix tree, and report the substring u_v if the respective *burrows-wheeler* characters are pairwise distinct. In Fig. 3, the terminal nodes are

v_2, v_3, v_4, v_5, and v_6. The condition on the burrows-wheeler character is satisfied for the nodes v_4, v_5, and v_6. Hence, the corresponding substrings aa, aca, and at are supermaximal repeats, respectively. The condition on the burrows-wheeler character is not satisfied for v_2 and v_3, because the suffixes 7 and 9 under v_2 are preceeded by the character a, and the suffixes 1 and 5 are preceeded by the character a. Because the number of terminal nodes and leaves is $O(n)$, this algorithm takes $O(n)$ time.

4.2 Identifying Tandem Repeats

A substring of S is a *tandem repeat*, if it can be written as $\omega\omega$ for some nonempty string ω. An occurrence of a tandem repeat $\omega\omega = S[p..p + 2|\omega| - 1]$ is *branching*, if $S[p + |\omega|] \neq S[p + 2|\omega|]$.

Stoye and Gusfield [56] described how all tandem repeats can be derived from branching tandem repeats by successively shifting a window to the left; see Fig. 7. They also showed that for each branching tandem repeat $\omega\omega$, there is one node in the suffix tree such that the substring obtained by concatenating the edge labels from the root to v is ω. For this reason, we focus on the problem of computing all branching tandem repeats using the suffix tree.

Let v denote a node in the suffix tree and let ω denote the substring obtained by concatenating the edge labels from the root to v. From each node v, one traverses the tree downwards and performs pattern matching, as will be explained in Sect. 5.1, to check if $\omega\omega$ occurs in S. Specifically, we search in the subtree at v to see if there is a path starting from v such that the concatenation of its edge labels spells out $\omega\alpha$, where α is any substring of S (α could be the empty string). For example, $\omega = at$ at the node v_6 in the suffix tree of Fig. 3. If we make a search in the subtree of this node, we find at on the path from this node to the leaf 6. Then we have an occurrence of a branching tandem repeat starting at position 6 in S and composed of the string $atat$.

The worst case running time of this algorithm is $O(n^2)$ (take, e.g., $S = a^n$). However, the expected length of the longest repeat of a string S is $O(\log n)$, where $n = |S|$. That is, $|\omega| = O(\log n)$ in average. Hence, the expected running time of the algorithm is $O(n \log n)$.

It is worth mentioning that there are algorithms that can compute tandem repeats in $O(n)$ time in the worst case; see [29,40]. However, these algorithms are complicated and less efficient in practice than the algorithm described above, see [1].

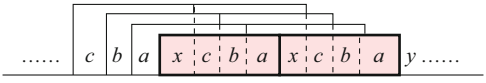


Fig. 7 Chain of non-branching tandem repeats $axcb$, $baxc$, and $cbax$, derived by successively shifting a window one character to the left, starting from the branching tandem repeat $xcba$

4.3 Identifying Unique Subsequences

The problem of finding all unique substrings is relevant for designing *oligonucleotides*. Oligonucleotides are short single-stranded DNA sequences. They are used either as *primers* in a PCR (Polymerase Chain Reaction) process to start the synthesis of a stretch of DNA having the complementary sequence of these oligonucleotides, or as *probes* to detect (and quantify) the existence of its complementary sequence in a pool of DNA sequences. These oligonucleotides should be unique to the DNA stretch they detect or synthesize. For example, if a certain oligonucleotide is used for the detection or isolation of a certain gene in a genome, then the complementary sequence should occur only in this gene, and nowhere else in the studied genome. Furthermore, the length of oligonucleotides should be within certain limits due to chemical and physical considerations (the length of a primer is between 18–30 and that of probes is between 4–10 base pairs). This leads to the definition of *unique substrings*.

Definition 1. A substring of S is *unique* if it occurs only once in S.

Definition 2. Given a string S of length n and two integers $0 < \ell_1 < \ell_2 \leq n$, the *unique substrings problem* is to find all unique substrings of S such that the length of each is between ℓ_1 and ℓ_2. The string S is a unique substring of length n.

For example, the strings ca, cac, aca, and $acac$ are unique substrings in $acac$. It is not difficult to verify that a unique substring in S is associated with a leaf in the suffix tree. In particular, if ω is a unique substring of S of length $\ell + 1$, then there is a leaf corresponding to the suffix $S(p)$ such that ω is a prefix of $S(p)$, and $u[\ell] \neq \$$.

To report each unique substring whose length ranges between ℓ_1 and ℓ_2, we perform a breadth-first traversal of the suffix tree. During the traversal, if the currently visited node v has $l(v) > \ell_2$, where $l(v)$ is the length of the substring obtained by concatenating the edge labels on the path from the root to v, then we visit no descendant nodes of v. If $\ell_1 \leq l(v) \leq \ell_2$ and v is a parent of a leaf corresponding to the suffix $S(p)$, then we report all unique matches starting at $S(p)$ in S and of length $l(v) < \ell \leq \ell_2$ only if $p + \ell < n$; i.e., we report the substrings $S[p..p + l(v)]$, $S[p..p + l(v) + 1], \ldots, S[p..p + \ell_2 - 1]$. For example, if it is required to report all unique matches of length $2 \leq l \leq 3$ of the string $acaaacatat$, whose suffix tree is given in Fig. 3, then the nodes v_2, v_3, v_4, and v_6 are considered. At v_2, for example, we report the substrings caa and cat.

It is not difficult to see that the time complexity of this algorithm is $O(n + z)$, where z is the number of unique matches.

4.4 Finding Absent Words

Absent words are strings not in S. It is clear that there is a prohibitively huge number of absent words. Therefore, we restrict ourselves to the shortest absent words. For short fixed-length absent words, say length $d < n$, we can use the look-up table

method in a straightforward way. We report the substrings corresponding to empty entries in the look-up table. For larger d, we can extend the algorithm for computing unique substrings using the suffix tree as follows. Let v be a node in the suffix tree, let u_v be the substring obtained by concatenating the edge labels on the path from the root to v. If $|u_v| < d$, then we produce all the substrings $u_v w$ such that $|u_v w| = d$ and $u_v w \notin S$, by computing all possible strings $w \in \Sigma^{|w|}$ and appending them to u_v.

4.5 Approximate Repeats

Computing approximate repeats can be achieved by using a local alignment algorithm in which the sequence is aligned to itself. The algorithm of Smith and Waterman that will be introduced in the next section for computing local alignments can be used in a straightforward manner to find approximate repeats. This algorithm takes $O(n^2)$ time, which is not feasible for long sequences. To overcome this, heuristic algorithms, addressed in the next section, were introduced to compute approximate repeats in a faster way.

4.6 Constraints on the Repeats

We address two types of constraints: (1) Constraints on the number of repeat occurrences and (2) proximity constraints.

Constraints on the number of occurrences

Posing constraints on the number of repeats or repeated pairs to be reported is beneficial for many applications in bioinformatics. As mentioned in the introduction, mammalian and plant genomes are full of repetitive elements of different kinds. Dispersed repeats are more abundant; they compose, for example, about 1.4 Gbp from the 3 Gbp human genome. From the different types of the dispersed repeats, three families are prevailing: LINE, SINE, and LTR. If a biologist is interested in locating these three families, then she/he searches for repeats appearing very often in the genome. Otherwise, she/he looks for non-abundant repeats.

In the following, we focus on posing an upper bound constraint on the number of repeats to be reported; posing a lower bound is complementary to what will be addressed here. We start with introducing the definitions of *rare* and *infrequent* maximal repeated pairs.

Definition 3. Let (l, p_1, p_2) denote a repeated pair $((p_1, p_1+l-1)(p_2, p_2+l-1))$, and let a threshold $t \in \mathbb{N}$ be given. A maximal repeated pair (l, p_1, p_2) is called *rare in* S if the string $w = S[p_1..p_1+l-1] = S[p_2..p_2+l-1]$ occurs at most $R_w \leq T$ times in S.

Definition 4. Let $(l, p_1^1, p_2^1), \ldots, (l, p_1^r, p_2^r)$ be maximal repeated pairs with $w = S[p_1^1..p_1^1+l-1] = \cdots = S[p_1^r..p_1^r+l-1]$ and let a threshold $t \in \mathbb{N}$ be given. Any of these repeated pairs is called *infrequent* if $|\{(l^1, p_1^1, p_2^1), \ldots, (l^r, p_1^r, p_2^r)\}| = r_w \leq t$.

Definition 3 adds constraints on the number of repeated substrings that are identical to the substrings from which the maximal repeated pair stems, and Definition 4 adds constraints on the number of the maximal repeated pairs themselves. That is, Definition 4 complements Definition 3 by further controlling the number of repeats.

It is our next goal to calculate the values r_w and R_w. This can be achieved by modifying the algorithm introduced in Sect. 4.1 for computing maximal repeated pairs based on the suffix tree. We use the following notations. For a position set $\mathcal{P}_v$, let $C_{\mathcal{P}_v}(S, a) = |\mathcal{P}_v(S, a)|$ be the number of suffixes under the subtree at the node v and preceeded by character $a \in \Sigma$. Let $C_{\mathcal{P}_v}(S) = \sum_{a \in \Sigma \cup \{\perp\}} C_{\mathcal{P}_v}(S, a)$ be the total number of suffixes under the subtree of the node v. Clearly, $C_{\mathcal{P}_v}(S)$ is the number of occurrences of the substring $w = S[i \ldots i + \ell]$ in S, where $S(i)$ is one of these suffixes and ℓ is the length of the substring produced by concatenating the edge labels from the root to v. Hence, it is easy to see that the value $C_{\mathcal{P}_v}(S) = R_w$.

For any node v containing k child intervals, $v_1, .., v_k$, the value r_w can be calculated according to the following formula, where $q, q' \in [1..k]$:

$$r_w = \frac{1}{2} \sum_{a \in \Sigma \cup \{\perp\}} \sum_{q \neq q'} C_{\mathcal{P}_{v_q}}(S, a) \cdot (C_{\mathcal{P}_{v_{q'}}}(S) - C_{\mathcal{P}_{v_{q'}}}(S, a))$$

In the example of Fig. 6, the calculation of the value r_w for the subtree at node v_1 yields $r_w = (1 * 1 + 1 * 2) + (1 * 1 + 1 * 1) + (1 * 1 + 1 * 2) = 8$.

The values $C_{\mathcal{P}_v}(S)$ and $C_{\mathcal{P}_v}(S, a)$ can be attached to the position set of each node and can be inherited from the child nodes to their parent node. Therefore, adding these constraints to the algorithm for computing maximal repeated pairs mentioned before entails no increase in the time complexity. Note that the values R_w and r_w are considered before performing the Cartesian product operation. That is, if they are under the certain threshold then a Cartesian product is performed, otherwise we discard the whole position sets, and start from the next non-visited leaf.

Proximity constraint

Consider a repeated pair $R(l, i, j)$, where the two substrings composing it are $S[i..i + l - 1]$ and $S[j..j + l - 1]$, $i < j$. We call the former substring the *first instance* and the latter the *second instance*. We define the *span*[1] of R as the number of characters $(j - i - l)$ between these two substrings.

[1] In [17] the word "gap" was used instead of span. But because the word gap has a specific meaning in sequence alignment, which will be addressed later in this chapter, we will use the word span in this section.

Adding a proximity constraint on the reported repeated pairs can be formulated as "find each repeated pair in S such that its span is between two constants c_1 and c_2" [17]. It is not difficult to see that the tandem repeats are a special case of such constrained repeated pairs where the span is zero. To avoid redundant output, we focus on finding maximal repeated pairs with constraints on the span. In the following, we show how to incorporate this constraint into the algorithm presented before for computing maximal repeated pairs using the suffix tree.

Recall from the algorithm presented in Sect. 4.1 for computing maximal repeated pairs that the Cartesian product operation reports the z maximal repeated pairs in time $O(z)$. The straightforward idea is to filter out those pairs not satisfying the proximity constraint. This immediately yields $O(n + z)$ algorithm. But the number of the constrained maximal repeated pairs is much less than z. Therefore, Brodal et al. [17] introduced an algorithm that takes $O(n \log n + z')$ time, where z' is the number of the reported maximal pairs satisfying the constraint. The idea of their algorithm is to represent the position sets as AVL-trees. The advantage is that the pairs satisfying the constraints can be directly reported in time proportional to their number. The union operation between two position sets will no longer be performed in constant time, but it will take $O(\log n)$ time. Hence, we have the $\log n$ factor in the time complexity given above. Because the details are to a great extent involved, we refer the reader to the original paper of Brodal et. al. [17].

5 Sequence Comparison

Due to its numerous applications, the area of string comparison is of great importance to bioinformatics in particular and to computational science in general. Figure 8 is an overview and taxonomy of the basic string comparison problems. In exact string comparison, the goal is to examine if the strings or substrings in comparison are either identical or not, while in approximate string comparison we can tolerate some differences. For applications in molecular biology, approximate comparison is the one of interest due to sequencing errors and mutations. Nevertheless, exact comparison is also important because many algorithms that find approximate similarities are usually based on exact common substrings. Note that some of the problems in the figure are addressed also in other computer science areas such as information retrieval and text mining. It is worth mentioning that the field of string comparison is also referred to as *pattern matching*, in which one string to be compared is considered as a *text* and the other as a *pattern*. Hence, we might use the terms "pattern matching" and "string comparison" synonymously.

5.1 Global and Semi-Global Exact Pattern Matching

The global exact pattern matching is the task we usually encounter in many applications, where one is given two strings S_1 and S_2 and it is required to report whether

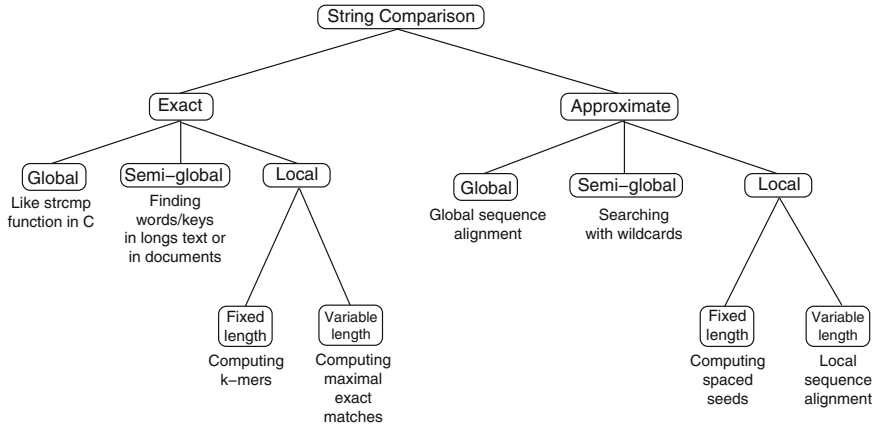


Fig. 8 A taxonomy of string comparison problems. We also show a real world application or problem for each type of string comparison

the two strings are identical or not. The algorithm is to compare the characters from left to right. If there is a mismatch, we report that $S_1 \neq S_2$ and terminate the comparison. If we reach the end of the strings, then we report that $S_1 = S_2$. The function `strcmp` in the programming language *C*, for example, is an implementation of this easy algorithm.

The semi-global exact pattern matching (known for short as exact pattern matching) problem is defined as follows. Given a string S of length n (usually called the *text*) and a pattern P of length $m \leq n$, the exact pattern matching problem is to find all the occurrences of P, if any, in S.

The brute force method takes $O(mn)$ time by sliding the pattern over S and comparing characters. This time complexity can be reduced to $O(n + m)$ if the pattern is pre-processed; the pre-processing time is $O(m)$ and the query time is $O(n)$. The algorithm of Knuth-Morris-Pratt, e.g., is one of these algorithms; see [28] for a simplified exposition and a survey of methods. Algorithms that pre-process the text itself, by constructing an indexing data structure for the text, achieve also $O(n + m)$ time; the pre-processing time is $O(n)$ and the query time is $O(m)$. But the advantage here is that the index is constructed once and millions of queries can be answered in time proportional to their length, independent of the length of S.

If it is required to search for patterns up to d characters, then one can use the look-up table presented in Sect. 3.2. The idea is to construct the look-up table over the text alphabet plus an additional dummy character, say "#." The reason for having this dummy character is to represent all the substrings of S of length up to d characters. If the dummy character were involved in the look-up table in Fig. 1, then we would have the extra entries corresponding to the substring $a\#$, $c\#$, $t\#$, $\#\#$. The last entry could be clearly not included in the table. The linked list for $a\#$ ($c\#$ or $t\#$) stores the positions where a (c or t) occurs in S.

If the pattern length is $m = d$, we directly compute the hashing function of the pattern, which maps us to a position in the look-up table. We then report the occurrences from the attached linked list. This takes $O(n + z)$ time, where z is the number of pattern occurrences in S. If the pattern length is $m < d$, then we append dummy characters so that the final pattern length becomes d. Thus, we can compute the hashing function and report the pattern occurrences. This takes also $O(d + z)$ time, which is not as optimal as the time complexity one can achieve using the suffix tree.

The exact pattern matching algorithm using the suffix tree works as follows. The suffix tree is traversed in a top-down fashion. We start from the root node and compare the edge labels of the outgoing edges to characters of the pattern. If there is a substring u labeling the edge from the root to a node v such that $P[0..|u-1|] = u$, then we move to node v and compare the sub-pattern $P[u..m-1]$ to the outgoing edge labels. This continues until there is a mismatch or all the characters of the pattern are compared successfully to the edge labels. In the former case the pattern does not occur in the text, while in the latter case the pattern exists and the positions where it occurs are the starting positions of the suffixes (leaves) in the subtree at the node on which the last edge traversed is incident. This procedure is illustrated in Fig. 3, where the gray arrows refer to the edges visited while searching for the pattern ac in S. This pattern starts at positions 0 and 4 in S.

5.2 Local Exact Matches

A similar region occurring in two strings S_1 and S_2 can be specified by the substrings $S_1[l_1 \ldots h_1]$, $S_2[l_2 \ldots h_2]$, where $1 \leq l_i < h_i \leq |S_i|$ and $i \in \{1, 2\}$. If $S_1[l_1 \ldots h_1] = S_2[l_2 \ldots h_2]$, i.e., the substrings are identical, then we have an *exact match*. This exact match can also be specified by the triple (ℓ, l_1, l_2), where $\ell = h_1 - l_1 = h_2 - l_2$. If the equality $S_1[l_1 - 1] = S_2[l_2 - 1]$ does not hold, then the exact match is called *left maximal*, i.e., it cannot extend to the left in the given strings. Similarly, if the equality $S_1[h_1 + 1] = S_2[h_2 + 1]$ does not hold, then the multiple exact match is called *right maximal*. The exact match is called *maximal*, abbreviated by *MEM*, if it is both left and right maximal. Fig. 9 (left) shows examples of maximal exact matches.

All the match types mentioned above can be computed using the exhaustive enumeration technique, which is one of the oldest techniques used to compute matches. It was first used in the software tool DIALIGN [46, 47] to compare protein and relatively short DNA sequences. The basic idea is that matches are the result of Cartesian products between all substrings of the sequences involved. In a sequence of average size n, we have $O(n^2)$ substrings. Thus, for two sequences, there are $O(n^4)$ possible matches. If the Cartesian products are limited to substrings of the same length, then the number of possible matches reduces to $O(n^3)$. To further reduce this number, only substrings of maximum length ℓ can be taken into account. This makes the number $O(n^2 l)$. Even with the $O(n^2 \ell)$ possible matches, this

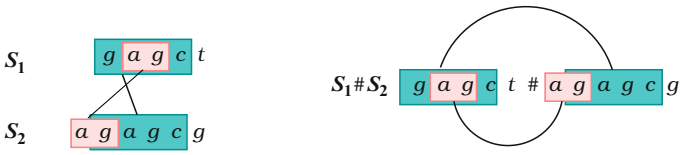


Fig. 9 Matches of the strings $S_1 = gagct$ and $S_2 = agagcg$. *Left*: two maximal exact matches of minimal length two. The first is $(4, 0, 1)$ composed of the substring "$gagc$," and the second is $(2, 1, 0)$ composed of the substring "ag." *Right*: these *MEMs* are maximal repeated pairs in $S_1\#S_2$

technique is still infeasible for comparing large sequences. In the following, better solutions will be presented. First, we will address efficient techniques for finding fixed length exact matches, where we report matches of fixed length $\ell = k$. Then we move to the computation of variable length matches, such as *MEMs*.

Finding Exact k-mers

Substring of S of fixed length k are called k-mers (also known as q-grams). If the matches reported are of certain fixed length, then they are called matches over k-mers. In this chapter, we call these matches, for short, k-mers.

The look-up table can be directly used to compute exact k-mers. We construct the table such that its entries contain the positions of all substrings of length k of one sequence, say S_1. Then the second sequence (say S_2) is streamed against this table to locate the k-mers. More precisely, each substring $S_2[i..i + k - 1]$ of length k of the second sequence is matched against this table, by computing the mapping function in Sect. 3.2. To save computation, we can use the recurrence in the same section to compute the mapping function of $S_2[i + 1..i + k]$ from $S_2[i..i + k - 1]$.

Constructing the look-up table for constant k takes linear time, specifically $O(n_1)$, where n_1 is the length of the first sequence. Deciding whether a query substring exists in the table takes $O(k)$ time and enumerating the matches, if any, takes time proportional to their number. Because there are $O(n_2)$ substrings in the second sequence, which is of length n_2, reporting all the z k-mers takes $O(n_2 + z)$ time. Therefore, the total time complexity is $O(n_1 + n_2 + z)$. The value k is usually chosen to be between 7 and 20, for genome size sequences, because the space consumption of the look-up table increases exponentially with it. Due to its simplicity, the look-up technique is widely used in software tools such as WABA [38], BLASTZ [51] (PipMaker [52]), BLAT [37], ASSIRC [58], GLASS [15], and LSH-ALL-PAIRS [18].

Automata can also be used to compute exact k-mers instead of constructing a look-up table. An automaton is constructed for one of the sequences, where its keys are are the set of the sequence substrings. The locations where these substrings occur in the sequence are attached to the leaves (nodes of success) of the automaton. For computing k-mers, each substring of length k of the other sequence is matched against this automaton.

We can also use the suffix tree for computing k-mers. But the suffix tree is too powerful to be used in this simple task, unless k is too large. We will explain in the next section how to use the suffix tree to compute variable length local matches, which is a more sophisticated task.

Maximal Exact Matches

Recall from the definitions given above that an exact match defined by the triple (l, p_1, p_2) is *maximal*, if $S[p_1 - 1] \neq S[p_2 - 1]$ and if $S[p_1 + l] \neq S[p_2 + l]$. It is not difficult to see that computing maximal exact matches between two strings S_1 and S_2 boils down to computing maximal repeated pairs of the string $S = S_1 \# S_2$, where # is a unique separator symbol occurring neither in the strings S_1 nor S_2. The triple (l, p_1, p_2) specifies a *MEM* if and only if $((p_1, p_1 + l - 1), (p_2, p_2 + l - 1))$ is a maximal repeated pair of the string S such that $p_1 + l - 1 < p < p_2$, where $p = |S_1|$ is the position of # in S.

Therefore, one can use the algorithm of Subsection 4.1 for computing maximal repeated pairs to compute maximal exact matches, but with the following modification. Each position set is further divided into two disjoint and possibly empty sets: One that contains all positions of the suffixes belonging to S_1 (these are smaller than $|S_1|$) and another one that contains all positions of the suffixes belonging to S_2 (these are greater than $|S_1|$). More precisely, we construct the set $\mathcal{P}_v(S_1, a)$ to contain all positions $p \in \mathcal{P}_v$ such that p corresponds to a position in S_1 and $S[p - 1] = a \in \Sigma \cup \{\perp\}$. $\mathcal{P}_v(S_2, a)$ is constructed analogously. (Figure 10 shows an example of position sets for computing *MEMs*.) To compute maximal exact matches, the Cartesian product is built from each position set $\mathcal{P}_v(S_1, a)$,

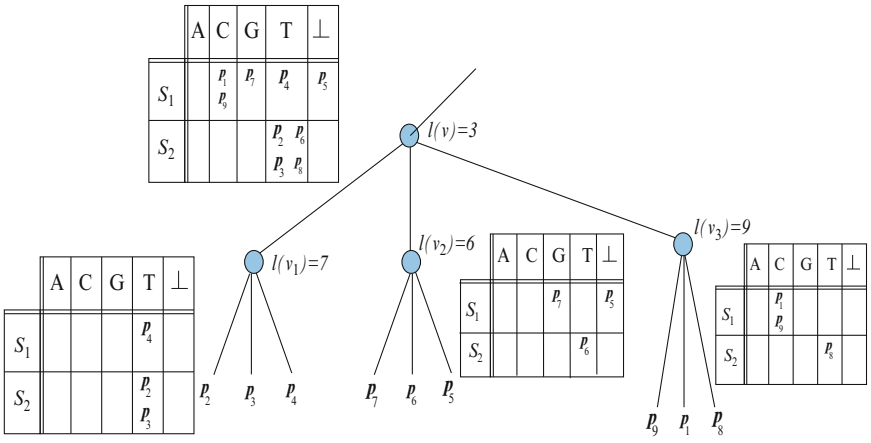


Fig. 10 The position sets of a part of the suffix tree. p_i denotes the suffix $S(p_i)$ starting at position p_i in S. We show these sets together just for illustration, but during the bottom up traversal the sets at v_1, v_2 and v_3 coalesce together into the set at v

$a \in \Sigma \cup \{\perp\}$ and the position sets $\mathcal{P}_v(S_2, b)$, where $a \neq b \in \Sigma \cup \{\perp\}$. It is not difficult to see that this modification does not affect the time and space complexity of the algorithm.

Constraints on the match occurrences

As we did before for maximal repeated pairs, we can add constraints on the number of matches and the substrings composing them. An important constraint is called *rare-MEMs*. A *MEM* is called *rare* if the constituent substrings $S_i[l_i \ldots h_i]$ appear at most T times in S_i, for any $i \in \{1, 2\}$ and T is a natural number specified by the user. A *MEM* is called *unique* if $T = 1$. In this case, we speak of a *maximal unique match*. In the example of Fig. 9 the *MEM* $(2, 1, 0)$ composed of the substring ag is not unique, because ag occurs more than one time in S_1 or S_2, while the *MEM* $(4, 0, 1)$ is unique.

The previous definitions given for maximal repeated pairs can be readily modified as follows. For a position set $\mathcal{P}$, let $C_{\mathcal{P}}(S_1, a) = |\mathcal{P}(S_1, a)|$ and $C_{\mathcal{P}}(S_1) = \sum_{a \in \Sigma \cup \{\perp\}} C_{\mathcal{P}}(S_1, a)$ (the values $C_{\mathcal{P}}(S_2, a)$ and $C_{\mathcal{P}}(S_2)$ are defined similarly). These values refer to the number of the constituent substrings of a maximal exact match at a node in the suffix tree. These values are considered before performing the Cartesian product operation. That is, if they fall under the given threshold, i.e., $C_{\mathcal{P}}(S_1, a) \leq T$ and $C_{\mathcal{P}}(S_2, a) \leq T$, then a Cartesian product is performed. Otherwise we discard the whole position sets, and start from the next non-visited leaf.

We can also have a constraint on the number of *MEMs* in addition to the number of the constituent substrings. For any node v, which contains k children, the number of *MEMs* associated with it (denoted by the value r_w) can be calculated according to the following formula, where q, q' are two child nodes:

$$r_w = \frac{1}{2} \sum_{a \in \Sigma \cup \{\perp\}} \sum_{q \neq q'} C_{\mathcal{P}_v^q}(S_1, a) \cdot (C_{\mathcal{P}_v^{q'}}(S_2) - C_{\mathcal{P}_v^{q'}}(S_2, a))$$

The Cartesian product operation is performed only if $r_w \leq t$, where t is a user defined threshold. Otherwise we discard the whole position sets, and start from the next non-visited leaf.

5.3 Approximate String Comparison and Sequence Alignment

Biological sequence comparison became a known practice since the emergence of biological sequence databases. There were technical as well as biological need that necessitated sequence comparison. The technical need was to avoid redundancy in the database, and the biological need was to infer the structure and function of new sequences by comparing them to those that already existed in the database. For example, two genes might have the same ancestor and function if they have

sufficiently similar sequences. Furthermore, the degree of similarity can indicate how long ago the two genes, and the organisms including them, diverged.

At that time, two or multiple such short sequences were compared on the character level (nucleotides in case of DNA/RNA and amino acids in case of proteins), and the result was delivered in terms of

1. *Replacements*, where characters in one sequence are replaced by other characters in the other sequence. If two characters replacing each other are identical, then they are called a *match*. Otherwise, they are called a *mismatch*.
2. *Indels*, where characters are inserted/deleted in either sequence. (A deletion in one sequence is an insertion in the other.)

In computer science terminology, character matches, mismatches, deletions, insertions are referred to as *edit operations*, and in biology they represent the mutation events.

The Global Alignment Algorithm

Computer scientists introduced sequence alignment algorithms to efficiently compare biological sequences in terms of these edit operations. If two sequences are pre-known to be similar, then *global sequence alignment* is the procedure to be used. Global alignment of two sequences is a series of successive edit operations progressing from the start of the sequences until their ends. This alignment can be represented by writing the sequences on two lines on the page, with replaced characters (matches/mismatches) placed in the same column and inserted/deleted characters placed next to a *gap*, represented by the symbol "_." In Fig. 11 (left), we show an example of aligning the two sequences ACTTAGTG and ACACCTG. From left to right, the series of edit operations is two matches of AC, one insertion (deletion) of T, three mismatches, and two matches of TG.

S_1 : ACTTAGTG
xxx
S_2 : AC_ACCTG

S		A	C	T	T	A	G	T	G
	0	-1	-2	-3	-4	-5	-6	-7	-8
A	-1	1	0	-1	-2	-3	-4	-5	-6
C	-2	0	2	1	0	-1	-2	-3	-4
A	-3	-1	1	2	1	1	-1	-2	-3
C	-4	-2	0	1	2	1	1	0	-2
C	-5	-3	-1	-1	1	2	1	1	0
T	-6	-4	-2	-2	0	1	2	2	1
G	-7	-5	-3	-3	-1	0	2	2	3

Fig. 11 *Left*: Global alignment, where gaps correspond to indels. The positions where mismatches exist are marked by the symbol x. *Right*: The alignment table (known also as the dynamic programming matrix). The colored cells correspond to the traceback path that recovers the optimal alignment. For this alignment, the score of a match $\sigma = 1$, the cost of a mismatch $\alpha = 0$, and the gap cost $\gamma = -1$

If each of the edit operations has a certain score, then the alignment can be scored as the summation of the edit operation scores. In the example of Fig. 11 (left), if a match scores 1, a mismatch scores 0, and an indel has penalty of -1, then the score of the alignment is 4. In the literature, the indels penalties are also referred to as *gap costs*.

Because the number of possible alignments is very large, one is interested only in an alignment of highest score, which is referred to as *optimal global alignment*. In an optimal alignment, the amount of characters replacing each other (identical or similar characters) are maximized and the amount of gaps (insertions and deletions) is minimized. In 1970, Needleman and Wunsch [48] introduced a dynamic programming algorithm to find an optimal global sequence alignment.

Let two strings S_1 and S_2 be given, and let $A(i, j)$ denote the optimal score of aligning $S_1[0..i]$ to $S_2[0..j]$. We define $A(-1,-1) = 0$, $A(0,-1) = \gamma(S_1[i])$, and $A(-1, 0) = \gamma(S_2[j])$, where $\gamma(a)$ is the cost of deleting character a (gap cost). The following recurrence computes the optimal alignment.

$$A(i, j) = \max \begin{cases} A(i-1, j-1) + \delta(S_1[i], S_2[j]) \\ A(i, j-1) + \gamma(S_1[i]) \\ A(i-1, j) + \gamma(S_2[j]) \end{cases}$$

where δ is the score of replacing one character with another. This score can be based on a substitution matrix such as PAM or BLOSUM in case of comparing protein sequences. In other cases, and also here, we assume a fixed model, which is defined as follows

$$\delta(S_1[i], S_2[j]) = \begin{cases} \sigma \text{ if } S_1[i] = S_2[j] \\ \alpha \text{ otherwise} \end{cases}$$

This recurrence simply determines that the optimal alignment up to (i, j) in S_1 and S_2 can be computed based on the optimal alignments $A(i-1, j-1)$, $A(i, j-1)$, and $A(i-1, j)$. The first clause of the recurrence refers to the case where we replace $S_1[i]$ with $S_2[j]$. Hence, the optimal alignment assuming this replacement would be $A(i-1, j-1) + \delta(S_1[i], S_2[j])$. The second clause of the recurrence refers to the case where character $S_1[i]$ is deleted/inserted. Hence, the optimal alignment assuming this indel would be $A(i, j-1) + \gamma(S_1[i])$. Finally, the third clause of the recurrence refers to the case where character $S_2[j]$ is deleted/inserted. Hence, the optimal alignment assuming this indel would be $A(i-1, j) + \gamma(S_2[j])$. That is, we have three scores: 1) optimal alignment assuming replacement, 2) optimal alignment assuming indel in S_1, and 3) optimal alignment assuming indel in S_2. The ultimate optimal alignment at (i, j) is the one of the highest score among these three possible alignments. Fig. 11 (right) shows an example of the alignment table, in which $\sigma = 1$, $\alpha = 0$, and $\gamma = -1$.

The easiest way to compute this recurrence is to use a 2D table (known also as the dynamic programming matrix) and filling the cell (i, j) based on the cells $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$. The optimal score is at the cell on the lower right corner. The alignment itself in terms of matches, mismatches, indels can be spelled out by tracing back the cells from $A(n, m)$ to $A(-1,-1)$. A backward diagonal move from $A(i, j)$ to $A(i-1, j-1)$ corresponds to match/mismatch between

$S_1[i]$ and $S_2[j]$, i.e., $S_1[i]$ matches/mismatches $S_2[j]$. A backward horizontal move from $A(i, j)$ to $A(i-1, j)$ corresponds to a gap in S_2, and an upward vertical move from $A(i, j)$ to $A(i, j-1)$ corresponds to a gap in S_1. The move direction itself is determined such that the score $A(i, j)$ equals either $A(i-1, j-1)+\delta(S_1[i], S_2[j])$, $A(i, j-1)+\gamma(S_1[i])$, or $A(i-1, j)+\gamma(S_2[j])$. The colored cells in the alignment table of Fig. 11 corresponds to the traced back cells, which recover the alignment on the left part of the figure. Multiple move possibilities imply that more than one optimal alignment exist.

The Needleman and Wunsch [48] algorithm takes $O(n^2)$ time and space. This complexity scales up to $O(n^k)$ in case of comparing k sequences of average length n.

Interestingly, it is possible to use only $O(n)$ space to compute the optimal score and to spell out the edit operations of an optimal alignment. This is based on the observation that computing the score at cell (i, j) depends only on the score at the three cells $(i-1, j-1)$, $(i, j-1)$, $(i-1, j)$. Hence, if we fill the array row-wise (column-wise), we need just to store the previous row (column) to compute the optimal score. Recovering the alignment itself by back tracing requires one additional trick, because the rows except for the last two ones of the matrix are no longer stored. This trick for a linear space dynamic programming algorithm is given in [32] and explained in [28].

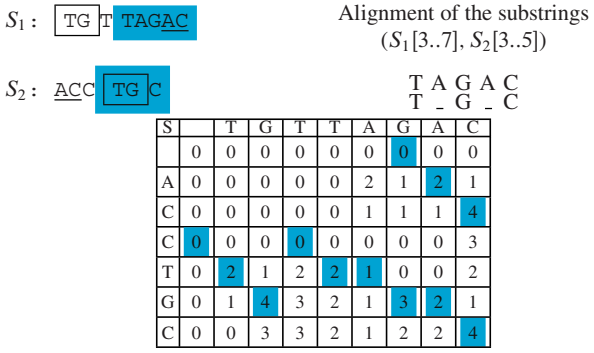
The Local Alignment Algorithm

In 1981, Smith and Waterman [54] introduced an essential modification to the Needleman and Wunch algorithm. Instead of computing an optimal global alignment, they computed local alignments of subsequences of the given sequences; their algorithm is, therefore, known as *local sequence alignment*. In Fig. 11 on the left, we show two sequences that are locally aligned. Note that the locally aligned regions are not in the same order in the two sequences.

Local alignments are important for three reasons: First, many biologically significant regions are subregions that are similar enough and the remaining regions are made up of unrelated sequences; the similar regions are known in proteins, for example, as *domains*. Second, insertions and deletions of any sizes are likely to be found as evolutionary changes; this is known as *domain insertions/deletions*. Third, the alignable subregions do not necessarily occur in the same order in the compared sequences; this corresponds to *domain shuffling*.

The algorithm of Smith and Waterman [54] is a very simple modification to the recurrences of the global alignment, but proofing the correctness of the recurrences is worth reading [54]. The modification is to use negative mismatch and gap costs and not allow any cell to have a negative score, which leads to the recurrence

$$A(i, j) = \max \begin{cases} A(i-1, j-1) + \delta(S_1[i], S_2[j]) \\ A(i, j-1) + \gamma(S_1[i]) \\ A(i-1, j) + \gamma(S_2[j]) \\ 0 \end{cases}$$



S		T	G	T	T	A	G	A	C
	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	2	1	2	1
C	0	0	0	0	0	1	1	1	4
C	0	0	0	0	0	0	0	0	3
T	0	2	1	2	2	1	0	0	2
G	0	1	4	3	2	1	3	2	1
C	0	0	3	3	2	1	2	2	4

Fig. 12 *Upper left*: Local alignment, where the underlined region in S_1 aligns to the underlined region in S_2, and the region in the rectangle in S_1 aligns to the one in rectangle in S_2. *Bottom*: The alignment table storing scores for local alignment. There are three possible highest scoring local alignments. The first is between the substrings (S_1[0..1], S_2[3..4]), marked by *boxes*, see the bottom part of the figure. The second is between the substrings (S_1[3..7], S_2[3..5]), marked by *colored boxes*, whose detailed alignment is in the *top right* part of the figure. The third is between the substrings (S_1[6..7], S_2[0..1]), which are underlined. For this alignment, the match score $\sigma = 2$, the mismatch cost $\alpha = -1$, and the gap cost $\gamma = -1$

Regarding the complexity, the Smith–Waterman algorithm still takes $O(n^2)$ time; this complexity scales up to $O(n^k)$ in case of comparing k sequences of average length n. In Fig. 12 on the right, we show the alignment table for computing the local alignment of the two strings TGTTAGAC and ACCTGC. In this example, $\sigma = 2$, $\alpha = -1$, and $\gamma = -1$. A best local alignment ends at a cell of maximum score. The edit operations themselves can be recovered by back tracing, as we did before for global alignment. But in this case, we start from this highest scoring cell until reaching a cell of value zero. Note that there might be more than one cell of maximum score, implying that more than one local alignment over different subsequences exist. In the example of Fig. 12, we have three highest scoring alignments, two of them are overlapping. Recovering the best set of non-overlapping local alignments is an interesting problem addressed in the paper of Bafna et al. [13].

Semi-Global Sequence Alignment

Semi-global sequence alignment (usually known as approximate pattern matching) produces alignment that is global with respect to one sequence but local with respect to the other. For example, if we search for a short biological sequence S_1 in a much larger biological sequence S_2 then S_1 should be globally aligned to a subsequence of S_2. The following is a definition given by Gusfield [28] that specifies the semi-global sequence alignment problem.

Definition 5 ([28]). Given a parameter δ, a substring $S_2[l..h]$ of S_2 is said to be an approximate match of $S_1[0..n-1]$ if and only if the optimal alignment of S_1 to $S_2[l..h]$ has value at least δ.

To solve the semi-global alignment problem, we use the same recurrence for global sequence alignment, but we initialize the cells of the initialization row (row -1) with zeros (assuming that the horizontal axis of the table is arrayed with S_2). There is an occurrence of S_1 in S_2 ending at position j of S if and only if $A(n-1, j) \geq \delta$. In other words, each cell in the last row and in column j of the alignment matrix with score at least δ corresponds to an occurrence of S_1 ending at position j of S_2. To find the start position of an occurrence ending at position j of S_2 and to spell out the alignment, we trace back from the cell $(n-1, j)$ until we reach a cell in row -1. Assume the column number of the last traversed cell is l, then the occurrence starts at position $l+1$ in S_2. Fig. 13 shows an example, in which the score of a match $\sigma = 2$, the cost of a mismatch $\alpha = -1$, and the gap cost $\gamma = -1$.

Variations of Approximate Matching

Fixed-length local approximate matches

The notion of exact k-mers of Sect. 5.2 can be extended to allow some errors within the fixed-length match. There are two kinds of non-exact k-mers: The first restricts the allowed mismatches to fixed positions of the k-mer, but the second does not. The former kind of k-mers can be specified by a certain *pattern* or *mask* of ones and zeros. For example, it can be required to report all 8-mers such that the third and sixth characters are not necessary identical (do not care) in both sequences.

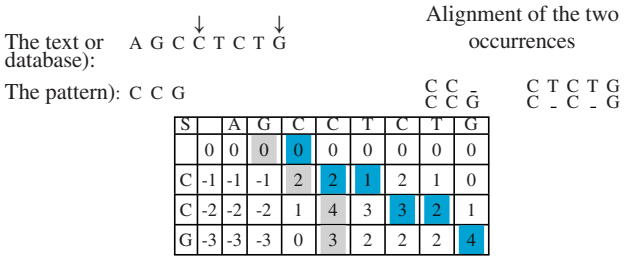


Fig. 13 *Upper left*: The given database (text) and the pattern. The *arrows* above the text indicates to where the matching pattern ends. *Top right*: Alignment of the two occurrences, recovered from the dynamic programming matrix given on the right. We report all occurrences with score larger than or equal to 3. The traceback starts from the cells with score larger than 3 and ends at the row -1. *Bottom*: The alignment matrix in which the cells of row -1 are initialized with zeros. The dark grey back tracing path corresponds to one occurrence of the pattern starting at position 3 in the text and the light grey path corresponds to another occurrence starting at position 2. For this alignment, the match scores $\sigma = 2$, a mismatch costs $\alpha = -1$, and a gap costs $\gamma = -1$

The respective mask is 11011011, where the ones correspond to the locations of the identical characters, and the zeros correspond to the locations where mismatches are allowed.

The look-up table can be used to compute k-mers according to a pre-determined mask, as follows. The look-up table is constructed such that its entries contain positions of substrings of the first sequence S_1, but the indices encode subsequences of these substrings. For a substring of length k, the respective subsequence is made up of the $d' < k$ characters that match the 1s of the mask. Let us take one example, suppose that S_1 contains the substring $w = aabc$ at position l_1. Assuming that $k = 4$ and the mask is 1101, then the index corresponding to w is computed according to the subsequence aac, and the entry next to it stores l_1. After constructing the look-up table, the second sequence is streamed against it as follows. For each substring of length k of the second sequence, one constructs the substring β, $|\beta| = d'$, containing the subsequence of characters that fit the 1s of the mask. For example, for a substring $aagc$ at position l_2, $\beta = aac$. If β is used as a query against the above look-up table, which has an entry for w, then the match (l_1, l_2, k) is obtained.

The look-up table was used first in PatternHunter [44] to locate non-exact k-mers according to a certain mask, and it was employed later in a recent version of BLASTZ [51].

The look-up table can also be used to find non-exact k-mers so that mismatches are allowed at any positions. Assume that it is required to generate non-exact k-mers such that at most one mismatch is allowed at any position of the k-mer. The look-up table is constructed as usual for the first sequence. For each substring of the second genome, the streaming phase has to be modified: Every character is replaced by every other character in the alphabet one at a time. Then the resulting substring is queried against the look-up table. In total, there are $k(|\Sigma|-1)+1$ queries launched for each substring: one for the original substring and $k(|\Sigma - 1|)$ for the replaced characters. If more than one mismatch is allowed in the same k-mer, then the queries are composed by considering all combinations of all replaced characters at every two positions of each substring.

Non-exact k-mers have been shown useful for comparing distantly related sequences. For example, a recent version of BLASTZ used them to investigate more diverged regions between human and mouse genomes.

Note that the look-up table based technique can be used for small and moderate k. For larger k, the suffix tree is to be used. We leave the details as an exercise for the reader.

Finding approximate tandem repeats

Recall from Sect. 4.2 that an exact *tandem repeat* is a substring of S that can be written as $\omega\omega$ for some nonempty string ω. An approximate tandem repeat can also be written as $\omega_1\omega_2$ such that ω_1 and ω_2 are not exact but are similar.

For each position i in the given sequence S, $0 \leq i < n$, we run a variation of the global alignment algorithm on the substrings $S[0..i]$ and $S[i+1..n]$ to find the

best alignment between a suffix of $S[0..i]$ and a prefix of $S[i+1..n]$, a variation known as suffix-prefix alignment. As we shall see in the next section, computing an optimal suffix-prefix alignment takes $O(n^2)$, the same as the global alignment problem. Therefore, the complexity of finding all approximate tandem repeats is $O(n^3)$ time and $O(n)$ space. It is worth mentioning that more sophisticated algorithms with better complexity already exist, see for example [36,43].

Now we discuss how to compute an optimal suffix–prefix alignment. Let $S_{suf}[0..i] = S[0..i]$ and let $S_{prf}[0..n-i-2] = S[i+1..n-1]$. We use the same recurrence for global sequence alignment, but we initialize the cells of the initialization column (column -1) with zeros (assuming that the vertical axis of the table is arrayed with S_{suf}). An optimal suffix-prefix alignment ends at the highest scoring cell in the last row. To spell out the alignment, we trace back from this cell until we reach column -1. Assume that the highest scoring cell is at column l and we end the traceback procedure at row k, then the optimal prefix alignment is between $S_{suf}[k+1..i]$ and $S_{prf}[0..l]$. Fig. 14 shows an example of a suffix–prefix alignment.

It is worth mentioning that the suffix-prefix alignment plays an important role in genome assembly, where a whole genome is given as a set of substrings (called reads) and it is required to reconstruct it. The first step towards genome assembly is to solve the suffix-prefix alignment problem for each pair of reads. The pair of reads with high suffix–prefix alignment score are likely following each other in the genome. Assuming enough overlapping reads, the genome can be assembled by compiling the reads based on their suffix–prefix overlapping.

Heuristics for sequence alignment

Undoubtedly, alignment algorithms based on dynamic programming are invaluable tools for the analysis of short sequences. However, the running time of these

S_{prf} : TTTGACC

S_{suf} : AGCCTCTG

Alignment of S_{prf} and S_{suf}

```
    A G C C T C T G
  T T T G A - C C
```

S		A	G	C	C	T	C	T	G
	0	-1	-2	-3	-4	-5	-6	-7	-8
T	0	-1	-2	-3	-4	-2	-3	-4	-5
T	0	-1	-2	-3	-4	-2	-3	-1	-2
T	0	-1	-2	0	-1	-2	-3	-1	-2
G	0	-1	1	0	-1	-2	-3	-2	1
A	0	2	1	0	-1	-2	-3	-3	0
C	0	-1	1	3	2	1	0	-1	-1
C	0	-1	0	3	5	4	3	2	1

Fig. 14 *Upper left*: The two strings S_{suf} and S_{prf}. *Top right*: The best suffix-prefix alignment of S_{suf} and S_{prf}, recovered from the alignment matrix on the right. *Bottom*: The alignment table storing scores for the suffix-prefix alignment, in which the cells of column -1 are initialized with zeros. The highest score at the last row is 5. From the cell containing the highest score we trace back until reaching column -1. For this alignment, a match scores $\sigma = 2$, a mismatch costs $\alpha = -1$, and a gap costs $\gamma = -1$

algorithms renders them unsuitable for large scale and high-throughput comparisons. In the following, we will classify and address the basic heuristics for fast sequence alignment.

All the heuristics that speed-up the alignment process are based on what is called the *filtering technique*, see [5] for a survey. The idea of this technique is to first generate matches, usually exact ones. Then the regions containing no matches are not processed (filtered out), and the regions containing matches are further processed to produce detailed alignment. There are two basic techniques to process the matches: seed-and-extend and chaining.

Seed-and-extend techniques

The seed and extend strategy determines regions of local similarity by extending each match (seed) to the left and to the right by applying a standard sequence alignment algorithm (based on dynamic programming) starting from the terminals of the seed. This extension continues until the score falls under a certain user defined threshold. An extended seed whose score is less than a certain threshold is discarded. Seed extension is carried out to filter out seeds that appeared by chance and are not part of a shared and inherited (homologous) region. Fig. 15 (left) illustrates the strategy of single seed extension. To avoid redundancy, a seed is extended only if it belongs to no region that is a result of an extension of another seed.

Some tools start the extension only if there are two matches occurring near each other; this is referred to as *double seed extension*, see Fig. 15 (right). The reason for this is to reduce the number of false positive extensions and speed up the alignment process. The idea of double seed extension was suggested first in Gapped-BLAST [11] for searching sequence databases, and was later used in BLAT [37] for searching EST databases.

The seed and extend strategy is suitable for finding short local similarities with higher sensitivity. However, to start from a single seed and to extend it to a large region of a genome is very time consuming. In other words, this would be nothing but a standard dynamic programming algorithm on the character level except that it starts from the seed positions in the sequences. Therefore, this strategy is recommended for finding relatively short similarities of the genomes in comparison.

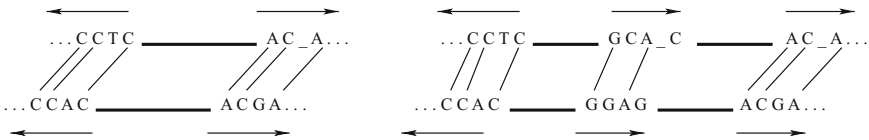


Fig. 15 Seed-and-extend technique. *Left*: single seed extension. *Right*: double seed extension. The bold *solid lines* are the seeds. The *arrows* show the direction of extension

Chaining techniques

Chaining techniques became the strategy of choice for comparing large genomes. This is because of their flexibility for various tasks, and their ability to accommodate large gaps. A chaining algorithm does not handle one match (or two) at a time. Rather, it tries to find a set of colinear non-overlapping matches according to certain criteria. This can be viewed as a generalization of the seed and extend strategy. Two matches are said to be colinear and non-overlapping, if the end positions of one match is strictly before the start positions of the second in all genomes. For example, in Fig. 16 (left) the two matches 2 and 3 are colinear and non-overlapping, while the matches 1 and 3 are not colinear, and the matches 1 and 2 are overlapping. (In the literature of chaining algorithms, the matches are referred to as *fragments* or *blocks*.)

Geometrically, a match f composed of the substrings $(S_1[l_1..h_1], S_2[l_2..h_2])$ can be represented by a rectangle in $\mathbb{N}^2_{\geq 0}$ with the two extreme corner points $beg(f) = (l_1, l_2)$ and $end(f) = (h_1, h_2)$. In Fig. 17, we show the matches of Fig. 16 (left) represented as two dimensional rectangles in a 2D plot.

For constructing a global alignment, a *global* chain of colinear non-overlapping matches is constructed. This global chain starts from the beginning of the sequences until their ends. The matches of the resulting global chain comprise what is called *the anchors*. These anchors are fixed in the alignment and the regions between the anchors are aligned using standard dynamic programming. The rationale is that if the two sequences share high similarity, then the anchors usually cover most of the sequences and the remaining parts to be aligned on the character level become very short. In Fig. 16 (left), we show an example of a global chain.

For constructing local alignments, sets of local chains of colinear non-overlapping matches are computed according to certain criteria. The score of each chain is the summation of the lengths of the involved matches minus the distances between the successive matches in the chain. (The distance between a match f

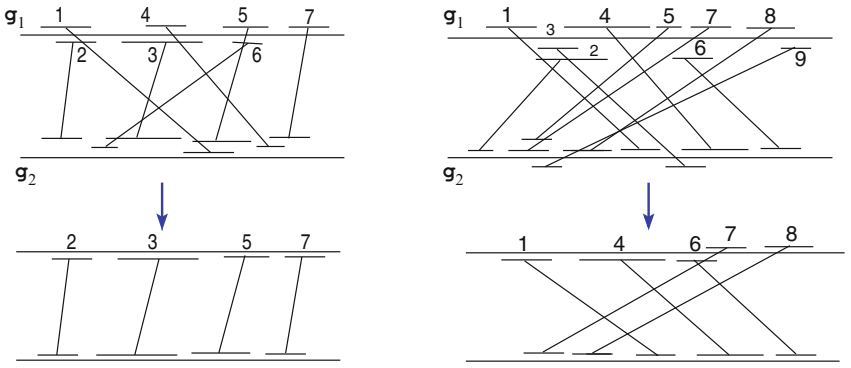


Fig. 16 *Left*: global chaining. The global chain is composed of the matches 2, 3, 5, and 7. *Right*: local chaining. There are two local chains. The first is composed of the three matches 1, 4, and 6, and the second is composed of the two matches 7, and 8

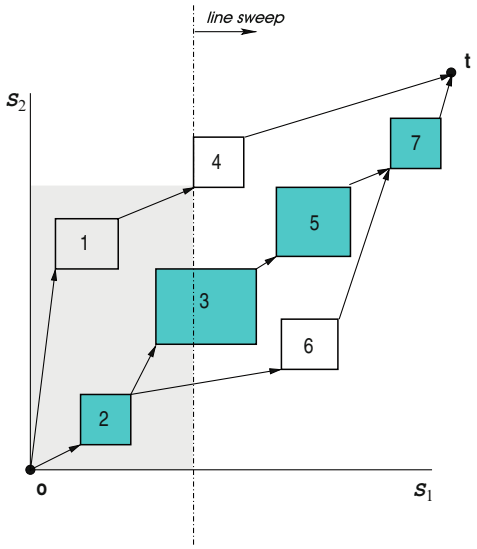


Fig. 17 Computation of a highest-scoring global chain of colinear non-overlapping matches. The search area (for finding the highest scoreing chain) at match 4 is bounded by the gray rectangle. The chain $\langle 2, 3, 5, 7\rangle$ is an optimal global chain, whose matches are plotted in a different color

followed by another match f' in a chain is the geometric distance between $end(f)$ and $beg(f')$ either in the L_1 or L_∞ metric.) Each local chain of a significant score (defined by the user) corresponds to a region of local similarity, and the matches of each local chain are the anchors for each local alignment. The regions between the anchors in each local chain are aligned on the character level to produce the final local alignment. In Fig. 16 (right), we show examples of local chains. Note that we might have more than one significant local chain, in an analogous way to the traditional local alignment on the character level.

The exact chaining algorithms (local and global) can be sub-divided into two groups according to the underlying computational paradigm/model: graph based and geometric based. In this chapter, we will briefly handle the global chaining problem. For the local chaining problem and other variations for comparative genomics, we refer the reader to [2, 3].

In the graph-based paradigm, the global chaining problem is solved as follows. A directed acyclic graph is constructed so that the matches are the nodes and the edges are added according to the colinearity and non-overlapping constraint. The edges are weighted, for example, by the length or statistical significance of the matches. The optimal solution is then reduced to finding an optimal path in this graph. In Fig. 17, we show an example of a graph constructed over the set of matches of Fig. 16 (left) (not all edges are drawn), and an optimal path in this graph.

The graph based solution takes $O(m^2)$ time to find an optimal solution, where m is the number of matches. This can be a severe drawback for large m. Fortunately, the complexity of the chaining algorithm can be improved by considering the geometric nature of the problem. A sweep line based algorithm can be used and it works as follows: The matches are first sorted in ascending order w.r.t. one se-

quence, and processed in this order to simulate a line scanning the points. During the line sweep procedure, if an end point of a match is scanned, then it is activated. If a start point of a match is scanned, then we connect it to an activated match of highest score occurring in the rectangular region bounded by the start point of the match and the origin, see Fig. 17. Details of this algorithm, which is subquadratic in time can be found in [4].

The repeat finding heuristics

It is worth mentioning that the seed-and-extend strategy and the chaining techniques can also be used to find local approximate repeats, in which the sequence is compared to itself. To take two examples, the program *REPuter* [42] first computes maximal repeated pairs then uses the seed-and-extend strategy to obtain approximate repeated pairs. The program CoCoNUT [2] uses a variation of the chaining technique over (rare) maximal repeated pairs to find the large segmental duplications in a genomic sequence. For a survey about methods and tools for repeat analysis, we refer the reader to [30].

6 Applications of String Mining in Other Areas

The string mining data structures and algorithms presented in this chapter are generic and provide efficient solutions to many problems in both exploratory and predictive data mining tasks both in bioinformatics and beyond. In this section, we review some of these applications. In particular, we show how the trie data structure is used in a popular algorithm for frequent itemset mining, a problem that is typically associated with business applications. We also describe how efficient string comparison methods can be used in computing string kernels required by data mining techniques in different applications. We also describe how sequence alignment algorithms have been used for pattern mining in unstructured (free text) and semi-structured (e.g., web-based) documents.

6.1 Applying the Trie to Finding Frequent Itemsets

The *association rule mining* problem is concerned with finding relations between database items. For example, given a set of purchased items in a supermarket, it would be useful to derive an assumption (rule) in the form "customers, who buy milk, also buy bread." Finding frequent itemsets is a basic step in association rule mining, in which the items bought frequently and together seem to be associated by

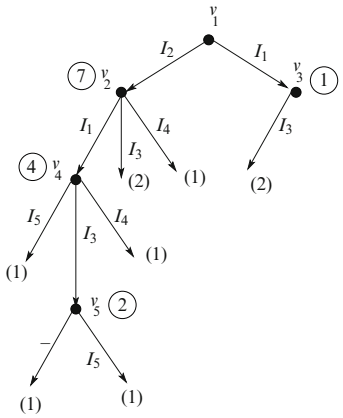
an implication rule. A related *sequence rule mining* problem is concerned with finding similar rules where the order of the transactions is important, e.g., "customers, who buy milk, also buy bread at a later date."

In this section we focus on the problem of finding frequent itemsets in a database of transactions, and we follow the definitions and notations introduced in [8]. Let $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$ be a set of items. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions, where each transaction T_i is made up of a set of items, i.e., $T_i \subset \mathcal{I}$. An *itemset* $IS \subseteq \mathcal{I}$ is a set of items. The *frequency* of an itemset IS, denoted by $F(IS)$, is the number of transactions containing IS as a subset. An itemset IS is called *frequent*, if $F(IS) > \tau$, where τ is a given threshold called *minimum support count*. A frequent itemset is called maximal if it is not a subset of another frequent itemset.

Figure 18 shows an example of transactions. For $\tau = 2$, the frequent itemsets with more than one element are $\{I_1, I_2\}$, $\{I_2, I_5\}$, $\{I_1, I_5\}$, $\{I_1, I_3\}$, $\{I_2, I_3\}$, $\{I_2, I_4\}$,

Transaction	Items	Sorted items
T_1	I_1, I_2, I_5	I_2, I_1, I_5
T_2	I_2, I_4	I_2, I_4
T_3	I_2, I_3	I_2, I_3
T_4	I_1, I_2, I_4	I_2, I_1, I_4
T_5	I_1, I_3	I_1, I_3
T_6	I_2, I_3	I_2, I_3
T_7	I_1, I_3	I_1, I_3
T_8	I_1, I_2, I_3, I_5	I_2, I_1, I_3, I_5
T_9	I_1, I_2, I_3	I_2, I_1, I_3

Item	Freq.
I_2	7
I_1	6
I_3	6
I_4	2
I_5	2



Item	FIS
I_5	$(\{I_2, I_5\}, 2)$, $(\{I_1, I_5\}, 2)$, $\underline{(\{I_2, I_1, I_5\}, 2)}$
I_4	$\underline{(\{I_2, I_4\}, 2)}$
I_3	$(\{I_2, I_3\}, 4)$, $(\{I_1, I_3\}, 2)$, $\underline{(\{I_2, I_1, I_3\}, 2)}$
I_1	$(\{I_2, I_1\}, 2)$

Fig. 18 A number of transactions over the set $\{I_1, I_2, I_3, I_4, I_5\}$ of items. The second column contains the items of each transaction sorted with respect to their count in the given dataset, which is shown in the table on its right. The FP-tree is shown on the right, where the numbers between the brackets are the frequency of each key (transaction). The circled numbers are the internal node counts. The table below shows the frequent itemsets (FIS) with minimum support 2. The maximal frequent itemsets are underlined

$\{I_1, I_2, I_3\}$, and $\{I_1, I_2, I_5\}$. The maximal frequent itemsets are $\{I_1, I_2, I_3\}$, $\{I_1, I_2, I_5\}$, and $\{I_2, I_4\}$. It is easy to see that reporting the maximal frequent itemsets only limits the redundancy without sacrificing meaningful results.

Naive solutions for finding frequent itemsets that are based on enumerating all possible combinations of candidate itemsets and then evaluating their support and confidence values are combinatorially expensive. Efficient solutions to the problem based on the *a priori principal* were developed in the mid nineties by Agarwal et al. [8] to prune candidate sets that have an infrequent sub-pattern.

An alternative solution to the problem that avoids candidate generation was developed by Han et al. [31]. This is known as the *FP-growth* algorithm and is one of the most efficient algorithms for finding the frequent itemsets [31]. The algorithm is based on constructing a data structure called *FP-tree* which is nothing but a slight variation of the trie data structure presented before in Sect. 3.2.

The FP-growth algorithm starts with counting how many times each item occurs in the transaction list, i.e., we compute $F(i)$, for each $i \in \mathcal{T}$. The second table in the example of Fig. 18 shows the frequency of each item. The items in each transaction are then sorted in decreasing order w.r.t. to their frequency, see the second column in the leftmost table of Fig. 18. Then a trie is constructed for all the transactions, where each transaction is considered as a key. If one key is a prefix of another key, then we add a dummy branching edge labeled with an empty character $''\_''$. For computing the frequent itemsets, we attach with each leaf the number of occurrences of the associated key, and for each node we attach the summation of the numbers attached to all the leaves in its subtree. In the sequel, we call this attached number the *node-count*, see Fig. 18. It is not difficult to see that the FP-tree can be constructed in $O(mn)$.

The FP-growth algorithm computes the frequent itemsets by processing the edges associated with the less frequent items before those associated with the more frequent items. Suppose that the edges associated with the items $I_{r_1}..I_{r_{x-1}}$ are already processed, where $F(I_{r_1}) < \cdots < F(I_{r_x})$. Assume that the edges associated with the item I_{r_x} are being currently processed, where $F(I_{r_{x-1}}) < F(I_{r_x})$. Then we run the following three steps.

1. We locate the internal nodes or the leaves such that the edges annotated with I_x are incident to them. Let the set $V = \{v_1, v_2, ..v_t\}$ denote such nodes.
2. For each node $v_i \in V$, $1 \leq i \leq t$, with node count greater than the minimum support count, we report the set $H \cup I_x$ as a frequent itemset, where the set H is made up of the edge labels of the nodes on the path from the root to the node v_i. Moreover, we report the sets $H' \cup I_x$, for each $H' \subset H$. Note that the set $H \cup I_x$ is the maximal frequent itemset and the sets $H' \cup I_x$ are non-maximal.
3. For each subset $V' \subset V$ such that $|V' > 1|$, we locate the node z in the tree such that z is the farthest node from the root and all the nodes of V' are descendant of z (z is called least common ancestor node of the nodes of V'). If the total node count of the nodes of V' in the subtree of z is greater than the minimum support count, then we report the set $H' \cup I_x$ as a frequent itemset, where the set H' is made up of the edge labels of the nodes on the path from the root to the node z. Moreover, we report the non-maximal frequent itemsets $H'' \cup I_x$, for each $H'' \subset H'$.

We take the edges associated with the next less frequent item after I_x and repeat the three steps mentioned above. Iteratively, we repeat this procedure until no more edges are left unprocessed. In the example of Fig. 18, where the minimum support is two, we start with the edges associated with item I_5, which is the least frequent item. There are two edges labeled with I_5 and incident to two leaves. Each leaf has node count less than the minimum support. Hence, we ignore them. The least common ancestor node of these two leaves is the node v_4, and the total node count of the two leaves associated with I_5 is 2, which satisfies the minimum support. Hence, we report the set $\{I_2, I_1, I_5\}$ as a maximal frequent itemset and the sets $\{I_1, I_5\}$ and $\{I_2, I_5\}$ as frequent itemsets.

6.2 *Application of String Comparison to a Data Mining Technique: Computing String Kernels*

The classifiers used in many problems in machine learning may have to work on discrete examples (like strings, trees, and graphs) [59]. Because the discrete examples, if not strings, can be readily reduced to strings, string kernels are used to measure the similarity between two strings, and accordingly between the two structures. Roughly speaking, the measure of similarity in string kernels is the total number of all common substrings between two input strings. This is made precise in the following.

Let $num_u(v)$ denote the number of occurrences of substring v in string u and let $\Sigma^\star$ denote the set of all possible non-empty strings over the alphabet Σ. The kernel $k(S, R)$ on the two strings S and R is defined as follows:

$$k(u, v) = \sum_{v \in \Sigma^\star} (num_S(v) \times num_R(v))\omega(v)$$

where $\omega(v) \in \mathbb{R}$ weighs the number of occurrences of each string v in both S and R.

A crucial observation that makes the kernel computation straightforward is that maximal exact matches form a compact representation of all the substrings in both S and R. That is, any substring occurring in S and R should be a substring of a maximal exact match of S and R. This suggests the following two phases algorithm:

1. Count the number of all maximal exact matches of length ℓ. For this purpose, we can use the procedure used for handling constraints on the number of maximal exact matches *MEMs*. Specifically, we accumulate the r_w values associated with a *MEM* of length ℓ, Let T_ℓ denote occurrences of the ℓ length *MEMs*.
2. For each ℓ, the number of matches included in it with length $1 \leq h \leq l$ equals $\ell(\ell + 1)T_\ell/2$.
3. Summating over all the lengths of maximal exact matches, we obtain $k(u, v)$.

By using the suffix tree data structure, it is not difficult to see that computing the string kernel takes $O(|S| + |R|)$ time and space.

6.3 String Mining in Unstructured and Semi-Structured Text Documents

Many of the string mining algorithms presented in this chapter form the basis for techniques used in pattern mining over both free text documents and semi-structured documents. For example Rigoustos and Huynh [50] describe a system that learns patterns and keywords that typically appear in spam emails. However, rather than simply looking for the exact string patterns in new emails, their approach applies a sequence similarity search similar to the one used in comparing protein sequences. Such an approach can discover variations of the strings including deletions and insertions of characters as well as substitutions of individual characters. With such an approach, having identified the string "Viagra" as a spam string, strings such as "Viagra," "V|agra," "Vigra" and "v I a G r A" would all be automatically scored as similar with a high score. Similar concepts form the basis for identifying plagiarism in both free text documents as well semi-structured documents such as program code; see for example [14, 22]. Another typical application of string mining algorithms is in the automatic extraction of information from web pages. In such applications particular textual information of interest to the user, e.g., specifications of a particular product or its price, is embedded within the HTML tags of the document. These tags provide the overall structure of the document as well as its formatting. Manual extraction of the required information is typically easy for humans based on both the semantic clues (e.g., the meaning of keywords such as price) as well as formatting cues (titles, tables, itemized lists, images, etc.). Identifying and extracting the same information using automatic methods is, however, more challenging. One reason is that the web page layout is typically designed for human users rather than information extraction robots; the proximity of visual clues and cues on the 2-dimensional page layout does not directly translate to proximity at the string level or even at the HTML structure level. A second reason is that different products could be described differently on each web page; e.g., some information might be missing or described in slightly different ways on each web page.

A family of approaches typically known as *wrapper induction* methods [55] are used to address the problem. For systems based on these approaches, a user provides a set of different example web pages and annotates the target information he would like to extract from them. The system then uses machine learning methods to learn patterns that best describe the target information to be extracted and then generates rules that can be used by specialized extractors to extract such information. The patterns learned from each example patterns are typically strings of tokens that represent both HTML tags as well regular text. The string mining algorithms presented in this chapter form the basic components for addressing some of the key challenge for such applications. For example, these wrapper induction tools typically need to generalize multiple identification patterns into a single patterns. This can be achieved by using the basic alignment algorithms, or variations

that align multiple sequences (e.g., CLUSTALW [57]). For example, given the two patterns listed below:

```
String 1 : <div> <b> <br> <span> <br> a </br> <br>
target <br><br>
String 2 : <div> <b> <br> <span> <br> target> <br><br>
```

the system would first align both patterns indicating the missing tokens

```
String 1 : <div> <b> <br> <span> <br> a <br> target
<br><br>
 String 2 : <div> <b> <br> <span> <br> - -
target <br><br>
```

and then generate a generalized pattern. where * can match any number of tokens.

```
String 2 : <div> <b> <br> <span> <br> * target
<br><br>
```

Examples of using such alignment methods, and also methods for identifying and dealing with repeated patterns within the context of information extraction from web sources, are described in Chang et al. [20]. Examples of using other string mining techniques for context-specific searching in XML documents are described in [23].

7 Conclusions

In this chapter we addressed the basic data structures and algorithms used in analyzing biological sequences. We also explored the application of these techniques in other areas of data mining.

The look-up table is an easy to manipulate data structure. But its space consumption increases exponentially with the key size. A recent method that is suitable for our data, which is static in nature, is the perfect hashing technique [16, 24]. In this technique, the keys can be stored in linear space and the constant access time is still guaranteed. Applying this technique to biological sequences is addressed in [6].

For string comparison based on the suffix tree, the strategy was to concatenate the sequences to be compared and reduce the problem to a variation of the problem of finding repeats. This strategy requires to construct the suffix tree for the two sequences, which might be infeasible for large sequences. The method of Chang and Lawler [21], which was introduced for computing matching statistics, can be used to overcome this problem. The idea of their method is to construct the index of only one sequence (usually the shorter one) and matches the second against it. To achieve linear time processing, they augmented the suffix tree with extra internal links called *suffix links*.

The suffix tree is an essential data structure for sequence analysis, but its space consumption and cash performance is a bottleneck for handling large genomic sequences. The enhanced suffix array is a recent alternative that overcomes these problems, see the original paper of the enhanced suffix array [1] and the recent paper with theoretical fine tuning of it [6].

There are a number of packages implementing the algorithms in this chapter. The strmat [39] package implements many of the algorithms in this chapter based on the suffix tree. The Vmatch [41] and GenomeTools [27] implement many of the repeat analysis and sequence comparison algorithms based on the enhanced suffix array. Other packages in this regard include SeqAn [26] and mkESA [34].

References

1. M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.
2. M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. CoCoNUT: An efficient system for the comparison and analysis of genomes. *BMC Bioinformatics*, 9:476, 2008.
3. M.I. Abouelhoda and E. Ohlebusch. A local chaining algorithm and its applications in comparative genomics. In *Proceedings of the 3rd Workshop on Algorithms in Bioinformatics*, volume 2812 of *LNBI*, pages 1–16. Springer Verlag, 2003.
4. M.I. Abouelhoda and E. Ohlebusch. Chaining algorithms and applications in comparative genomics. *Journal of Discrete Algorithms*, 3(2-4):321–341, 2005.
5. M.I. Abouelhoda. *Algorithms and a software system for comparative genome analysis*. PhD thesis, Ulm University, 2005.
6. M.I. Abouelhoda and A. Dawood. Fine tuning the enhanced suffix array. In *Proceedings of 4th Cairo International Biomedical Engineering Conference, IEEExplore, DOI:10.1109/CIBEC.2008.4786047*, 2008.
7. M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix arrays. In Srinivas Aluru, editor, *Handbook of Computational Molecular Biology (Chapter 7)*. Chapman & Hall/CRC Computer and Information Science Series, 2006.
8. R. Agarwal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM-SIGMOD*, pages 207–216, 1993.
9. A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.
10. S.F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
11. S.F. Altschul, T. L. Madden, A. A. Schäffer, and et al.˙Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.
12. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI, pages 85–96. Springer-Verlag, 1985.
13. V. Bafna, B. Narayanan, and R. Ravi. Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles). *Discrete Applied Mathematics*, 71:41–53, 1996.
14. B.S. Baker. On finding duplications and near duplications in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, page 86, 1995.
15. S. Batzoglu, L. Pachter, J. P. Mesirov, B. Berger, and E. S. Lander. Human and mouse gene structure: Comparative analysis and application to exon prediction. *Genome Research*, 10:950–958, 2001.

16. F.C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th International Workshop on Algorithms and Data Structures*, volume 4619 of *LNCS*, pages 139–150. Springer, 2007.
17. G. Brodal, R. Lyngso, Ch. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. *Journal of Discrete Algorithms*, 1(1):134–149, 2000.
18. J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.
19. T. Schiex C. Mathe, M.F. Sagot and P Rouze. Current methods of gene prediction, their strengths and weaknesses. *NAC*, 30(19):4103–17, 2002.
20. C.H. Chang, C.N. Hsu, and S.C. Lui. Automatic information extraction from semi-structured web pages by pattern discovery. *Decision Support Systems*, 35:129–147, 2003.
21. W.I. Chang and E.L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.
22. X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transactions Information Theory,*, 50:1545–1550, 2004.
23. K.T. Claypool. SUSAX: Context-specific searching in xml documents using sequence alignment techniques. *Data & Knowledge Engineering*, 65(2):177 – 197, 2008.
24. Z. Czech, G. Havas, and B.S. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.
25. A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.
26. A. Döring, D. Weese, T. Rausch, and K. Reinert. SeqAn an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9:9, 2008.
27. G. Gremme, S. Kurtz, and et. al. GenomeTools. http://genometools.org.
28. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.
29. D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. Report CSE-98-4, Computer Science Division, University of California, Davis, 1998.
30. B.J. Haas and S.L. Salzberg. Finding repeats in genome sequences. In T. Lengauer, editor, *Bioinformatics - From Genomes to Therapies*. Wiley-VCH, 2007.
31. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of ACM-SIGMOD*, pages 1–12, 2000.
32. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* , 18:341–343, 1975.
33. M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18:S312–S320, 2002.
34. R. Homman. mkESA - enhanced suffix array construction tool. `http://bibiserv.techfak.uni-bielefeld.de/mkesa/`.
35. R.L. Dunbrack Jr. Hidden markov models in bioinformatics. *Current Opinion in Structural Biology*, 16(3):374–84, 2006.
36. S. Kannan and E.G. Myers. An algorithm for locating non-overlapping regions of maximum alignment score. *SIAM Journal on Computing*, 25(3):648–662, 1996.
37. W.J. Kent. BLAT—the BLAST-like alignment tool. *Genome Research*, 12:656–664, 2002.
38. W.J. Kent and A.M. Zahler. Conservation, regulation, synteny, and introns in large-scale *C. briggsae–C. elegans* genomic alignment. *Genome Research*, 10:1115–1125, 2000.
39. J. Knight, D. Gusfield, and J. Stoye. The Strmat Software-Package. `http://www.cs.ucdavis.edu/ gusfield/strmat.tar.gz`.
40. R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 596–604. IEEE, 1999.
41. S. Kurtz. The Vmatch large scale sequence analysis software. http://www.vmatch.de.
42. S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.

43. G. Landau, J. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.
44. B. Ma, J. Tromp, and M. Li. PatternHuner: Faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
45. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 23(2):262–272, 1976.
46. B. Morgenstern. DIALIGN 2: Improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15(3):211–218, 1999.
47. B. Morgenstern. A space efficient algorithm for aligning large genomic sequences. *Bioinformatics*, 16(10):948–949, 2000.
48. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.
49. L. Pichl, T. Yamano, and T. Kaizoji. On the symbolic analysis of market indicators with the dynamic programming approach. In *Proceedings of the 3rd International Symposium on Neural Networks 2006*, LNCS 3973, 2006.
50. I. Rigoutsos and T. Huynh. Chung-Kwei: a Pattern-discovery-based System for the Automatic Identification of Unsolicited E-mail Messages (SPAM). In *Proceedings of the 1st Conference on Email and Anti-Spam (CEAS)*, 2004.
51. S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Research*, 13:103–107, 2003.
52. S. Schwartz, Z. Zhang, K. A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller. PipMaker–a web server for aligning two genomic DNA sequences. *Genome Research*, 10(4):577–586, 2000.
53. R. Sherkat and D. Rafiei. Efficiently evaluating order preserving similarity queries over historical market-basket data. In *The 22nd International Conference on Data Engineering (ICDE)*, page 19, 2006.
54. W. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Computational Biology*, 147:195–197, 1981.
55. S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.
56. J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002.
57. J. Thompson, D. Higgins, and Gibson. T. CLUSTALW: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties, and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.
58. P. Vincens, L. Buffat, C. Andrè, J. Chevrolat, J. Boisvieux, and S. Hazout. A strategy for finding regions of similarity in complete genome sequences. *Bioinformatics*, 14(8):715–725, 1998.
59. S.V.N. Vishwanathan and A.J Smola. Fast kernels for string and tree matching. In K. Tsuda, editor, *Kernels and Bioinformatics*. Cambridge, MA: MIT Press, 2004.
60. P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.