N. Cannata, N. Cocco, M. Simeoni

k-tries and abstraction for biosequences analysis
k-tries and abstraction for biosequences analysis

Nicola Cannata1, Nicoletta Cocco2, and Marta Simeoni2

1 CRIBI Università di Padova, viale G. Colombo 3, 35121 Padova, Italy
   nicola@cribi.unipd.it
2 Università di Venezia, via Torino 155, 30172 Venezia, Italy
   {cocco,simeoni}@dsi.unive.it


Abstract. Pattern discovery is used for determining, in a blind way, subsequences characterizing a given sequence or set of sequences. It is applied in genomics and proteomics for discovering interesting biosequences which are usually very short when compared to the length of the analyzed sequence. Abstraction of subsequences, that is grouping similar subsequences and representing them in a compact way as patterns, seems particularly useful in the field of pattern discovery in order to stress similarities among interesting subsequences. In this paper we propose a set of techniques for pattern discovery which makes use of abstraction. We define a data structure, the $k$-trie, which is essentially an enriched and truncated suffix trie, intended to collect and count subsequences up to a given length $k$. We give an on-line algorithm for building a $k$-trie in linear time. We also propose some pruning techniques for reducing the space complexity of the tree in the context of pattern discovery. Besides we consider three simple scores to be associated to the subsequences represented in the tree in order to estimate their "interest" and we study their properties w.r.t. symbol concatenation and string abstraction. In particular, for the chi-square score we supply a method for collecting the most interesting subsequences. We also propose some techniques for using abstraction in efficient searching of patterns and in representing concisely the most interesting subsequences.

1 Introduction

1.1 Motivation

The statistical analysis of substring occurrences in biological sequences is commonly used in bioinformatics as an instrument to discover bio-molecular signals in the sequences and to hypothesize their functional or structural properties [4]. Both the two groups of over- and under-represented oligo-sequences occurring with a significant deviation from the expected frequency (in a model of random symbols distribution or in a more sophisticated data driven model) could reveal some interesting biological meaning. As an example, in [26] the detection of over-represented oligonucleotides is adopted as a simple and fast method to isolate DNA binding sites for transcription factors from families of co-regulated genes.
Other applications concern the analysis of other nucleic acid binding sites (i.e. for the ribosome to start the mRNA translation [12]), the identification of sorting signals in protein sequences [9], the discovery and representation of protein domains [15], the search for background regularities in the DNA or in proteins (i.e. secondary structures).

However the molecular machinery allows some degrees of freedom in the binding sites and in the functional domains and structures, since it often permits the presence of one nucleotide (or aminoacid, for protein sequences), chosen from a set of possible ones, in a fixed position of the sequence. This fact is reflected in the common pattern representations, for example by adopting the IUPAC alphabet [7] when dealing with DNA or by including into brackets all the aminoacid or nucleotide symbols that could occur in a given position of a sequence. The usage of degenerate symbols and simplified alphabets may allow us to discover hidden properties or regularities otherwise not easily seen from the original sequences [6].

In the present paper we propose to combine these two techniques: a blind search for over- and under-represented oligo-sequences and the possibility of abstracting sequences, that is the possibility to have a set of alternative symbols in some positions of each sequence.

Blind search methods and analyses are generally applied to unaligned sequence sets in order to detect unknown signals that can then be further refined and investigated with more specific techniques (e.g. multiple sequence alignment [24], Position Weight Matrices [23], HMM [8], Sequence Logos [22]) by restricting the analysis to the discovered patterns or regions of interest. Besides we expect that the comparison of the characteristic patterns we can obtain from two sets of sequences related to different biological features (e.g. exon/intron, coding/non-coding, different secondary structures of protein sequences, protein sequences sorted in different organelles) could produce very interesting results in order to infer putative discriminating tools to be used in sequences classification and annotation.

1.2 Structure of the Paper

The paper is organized as follows. In Section 2 we define a tree data structure, the k-trie, for representing all the subsequences, up to a fixed length, contained in a sequence or a set of sequences. Each node of the tree represents a subsequence with associated the number of its occurrences and possibly other useful information related to that subsequence. The tree is essentially an enriched and cut suffix trie [27, 14, 25, 10], representing all substrings of the given input sequence up to a fixed length $k$. In Section 3 we show how the tree can be pruned with respect to some filtering conditions in order to reduce its complexity. In Section 4 we discuss how we can exploit abstraction while searching in the tree. We consider two possibilities: either we abstract the subsequences we search for, or we abstract the alphabet itself, thus making explicit the intended similarities among symbols. In Section 5 three simple scores are introduced for evaluating the significance of subsequences. We discuss how such scores distributes over the
k-trie and how they are affected by string concatenation and string union. In Section 6 we describe some techniques for computing the most interesting subsequences in the k-trie w.r.t. the chi-square score. In Section 7 we propose some techniques for computing an abstraction of the most interesting subsequences. In Section 8 we show how the k-tries can be used for comparing families of sequences in order to single out their similarities and differences. A brief comparison with related papers and some concluding remarks and perspectives for future work follows in Sections 9 and 10.

2 The k-trie

The substrings we search for in biological sequences are generally very short in comparison to the length of sequences: motifs and signals are at most ten or twenty symbols long while sequences under examination can be for example large parts of a whole genome and therefore also millions of symbols long. For this reason we want to represent all the substrings in a biological sequence (or in a set of sequences) up to a fixed length. In this section we introduce a tree structure which supplies such a representation and we show how to build it efficiently.

The reader is assumed to be familiar with the terminology and the basic concepts of language theory, such as symbol, string, alphabet, regular expression, see for example [11].

2.1 The tree structure

Let $T[1..m]$ be a text of length $m$ on an alphabet $\Sigma$; we want to single out all the substrings of length at most $k$ in $T$, where $1 \leq k \leq m$. We choose to represent all such substrings in a tree (a suffix trie [10]) of depth $k$, called the k-trie of $T$ and denoted by $T_k$.

We call $k$-substring of $T$ any substring $s = a_1a_2\ldots a_k$ of $T$ of length $k$. Each complete path $p = n_1n_2\ldots n_l$ ($l \leq k$) in the tree $T_k$ represents an $l$-substring, namely the nodes in the path are labeled with the symbols of the corresponding substring. In particular, each node $n_j$ of level $j$ in $p$, $1 \leq j \leq l$, (except the root which is actually not even represented in the implemented tree), is labeled by a symbol $a_j$ of $\Sigma$ and by a counter $c_j$ representing the number of occurrences of a substring $a_1a_2\ldots a_j$ in the text $T$. Further information can be associated to each node in the tree as we will see in the next sections.

The condition $l \leq k$ on the length of the path $p$ reflects the fact that $T_k$, in general, is not a complete tree. Note that there are at most $|\Sigma|^k$ distinct complete paths (and substrings) in $T_k$.

Consider the following simple example of k-trie.

Example 1. Let $T = AGAGGAC$ and $k = 2$. Then the 2-trie associated to $T$ is the following:
From the 2-trie associated to $T$ we see that there are four different sub-
strings of length 2, \{AC, AG, GA, GG\}, and three different substrings of length
1, \{A, G, C\}. The tree contains also their number of occurrences in $T$: for exam-
ple AG occurs two times, GG occurs one time, and GA occurs two times.

We need to be able to traverse a k-trie in different ways:

– along the paths from the root, i.e. following the substrings of $T$;
– along all the sons $n_j$ of a node of level $j$, $1 \leq j \leq k - 1$, i.e. along the alter-
native symbols in position $j + 1$ which may follow the substring corresponding
to the path from the root to $n_j$;
– along all the nodes of level $j$, i.e. the symbols in position $j$, $1 \leq j \leq k$, in all
the substrings of $T$.

The implementation of the k-trie has to take care of these requirements.
Hence, the structure of each node in the tree can be described by the following
type.

\[
\text{type node} = \text{record} \\
\hspace{1cm} \text{symbol : char;} \\
\hspace{1cm} \text{counter : int;} \\
\hspace{1cm} \text{sons : pointer(node);} \quad \{\text{pointer to the sons}\} \\
\hspace{1cm} \text{level : pointer(node);} \quad \{\text{pointer to the nodes of the same level}\} \\
\hspace{1cm} \text{end;}
\]

Additional fields will be necessary, when further information will be associated
to each substring.

To traverse the tree by levels we also need an array of $k$ pointers to the levels
of the tree:

\[
\text{lev} = \text{array } [1, k] \text{ of pointer(node);} 
\]
such pointers give access to the list of nodes at each level. All such pointers are
initialized to nil.
2.2 Building the k-trie

In order to efficiently build the tree, we also need an array of \( k \) pointers to nodes in the tree:

\[
ptab = \text{array}[1, k] \text{ of pointer (node)};
\]

such an array is useful while constructing the tree, but it can be deallocated afterwards.

A simple on-line algorithm for building the k-trie associated to the input text \( T \) is given below. It linearly scans the text from left to right and for each symbol \( T(i) \), it inserts \( T(i) \) in the tree by counting an occurrence in each position (level) in \([1, k]\) for each substring of \( T \) ending in position \( i \): one occurrence of \( T(i) \) in position (level) 1 for \( T[i .. i] \), one in position 2 for \( T[i-1 .. i] \), .. , one in position \( j \) for \( T[i-j+1 .. i] \), .. , one in position \( k \) for \( T[i-k+1 .. i] \).

In order to count the occurrences of \( T(i) \) in each position of the k-substrings with a linear scan of \( T \), we save in \( ptab \) the positions at each level in the tree where the occurrences of \( T(i-1) \) have been counted in the previous step.

The procedure \( \text{Build}\_\text{tree} \) makes use of \( \text{lev}[1] \), to access the first level of the tree (i.e. \( \text{lev}[1] \) can be thought of as the root) and of the procedure \( \text{Insert}(t, j, p1, p2) \). Such procedure looks for a node with a symbol \( t \) in the list pointed by \( p1 \) at level \( j \) in the tree. If \( t \) is already in the list, it increments the counter associated to \( t \), otherwise it adds a new node with associated symbol \( t \) to the beginning of the list pointed by \( p1 \) and by \( \text{lev}[j] \). If \( p1 = \text{nil} \ (\text{lev}[j] = \text{nil}) \) it initializes the list by setting \( p1 \ (\text{lev}[j]) \) to point to the newly created node. If a new node is created, then it initializes its fields by setting \( \text{symbol} \) to \( t \), \( \	ext{counter} \) to 1, \( \text{sons} \) to \( \text{nil} \) and \( \text{level} \) to \( \text{lev}[j] \), that is the level of the tree to which the node is added. In either case it returns \( p2 \) which is a pointer to the node found or created for \( t \).

\[
\text{Build}\_\text{tree}(T, k);
\]

\[
\text{Insert}(T(1), 1, \text{lev}[1], ptab(1)); \ { \text{initalize both the tree and ptab} } \]

\[
\text{for } i := 2 \text{ to } k-1 \text{ do begin} \]

\[
\text{for } j := i \text{ downto } 2 \text{ do } \ { j \text{ is the level in which } T(i) \text{ is inserted} } \]

\[
\text{Insert}(T(i), j, ptab(j-1).\text{sons}, ptab(j));
\]

\[
\text{Insert}(T(i), 1, \text{lev}[1], ptab(1));
\]

\[
\text{end; for } i := k \text{ to } m \text{ do begin} \]

\[
\text{for } j := k \text{ downto } 2 \text{ do } \ { j \text{ is the level in which } T(i) \text{ is inserted} } \]

\[
\text{Insert}(T(i), j, ptab(j-1).\text{sons}, ptab(j));
\]

\[
\text{Insert}(T(i), 1, \text{lev}[1], ptab(1));
\]

\[
\text{end;}
\]

Note that the first symbol \( T(1) \) of a text \( T \) is counted only at the first level of the tree, the second one only at the first and second level, and so on. Hence, only the symbols in \( T[k .. m] \) are counted in all the \( k \) levels of the tree.
To state the time complexity of the procedure BUILD_TREE, we may observe that, in the worst case, each INSERT(t, j, p1, p2) has to scan a list pointed by p1 of length |Σ|. Hence in the worst case the tree can be built in time \( O(k \cdot m \cdot |Σ|) \).

Regarding space complexity, the worst case is when the k-trie contains all possible strings on \( Σ \) up to length \( k \). This can happen for a large text and a small \( k \), which is our typical use case. Hence the tree requires at most \( S(|Σ|^k) \) memory cells. In fact since each node stores four item requiring constant space, we have \( 4 \sum_{i=1}^k |Σ|^i = 4(1 - |Σ|^k)/(1 - |Σ|) \leq 4|Σ|^k \). This indicates that space can be a problem for large alphabets, even for short substrings.

### 2.3 Comparison with similar structures

Our k-trie represents essentially the first \( k \) levels of a suffix trie for \( T \), with further information and links associated to each node.

In alternative to our construction we could build a compact suffix tree with a standard construction, such as Ukkonen’s algorithm [25, 10], and add the information on the number of occurrences of the corresponding substring at each node. This annotation can be obtained through a linear traversal of the suffix tree which counts and memorizes in each node the number of leaves of the subtree appended to it. Then we could cut the annotated suffix tree at string length equal to \( k \). This can be also done in a linear traversal of the suffix tree up to string length \( k \). In this way we would obtain an annotated and cut suffix tree, that is a data structure similar to our k-trie as far as the labelling is concerned. In fact, the resulting tree would have the same labelling in the paths of our k-trie, and it would be obtained with the same worst case time complexity, i.e. linearly dependent from \( k, |Σ| \) and \( m \).

Regarding space, the k-trie worsen the complexity of a \( k \) factor in the worst case w.r.t. the corresponding annotated and cut suffix tree. On the other hand, in our intended applications, with a large text \( T \), a small \( k \) and a small alphabet \( Σ \), we expect to obtain a k-trie which is close to a complete tree, with essentially the same structure (i.e. number of nodes) of the corresponding annotated and cut suffix tree. The main difference which remains between the two trees is in the linking of the nodes by level and by siblings in our k-trie. These links can be useful for building string abstractions and for comparing texts, as we will show in Sections 4, 7 and Section 8. We could add these further links to the annotated and cut suffix tree either by a further traversal or by complicating the traversal of the annotated suffix tree in order to both link the nodes and cut the tree to the desired depth. Our algorithm for building the k-trie does not require all the postprocessing which is required for the suffix tree.

A further difference between the suffix tree and the k-trie of a text \( T \) is that the former reports also the position of each substring, while the latter loose any such reference to the original text. However, in our context of signal discovery, this is not a priority information.

Note that the k-trie can be used also for representing substrings belonging to more than one text, analogously to what done by a generalized suffix tree.
In fact let us consider \( l \) texts on the alphabet \( |\Sigma| \), \( T_1, T_2, \ldots, T_l \), then we can apply the algorithm for building the k-trie to each text \( T_i \) separately and cumulate the substrings with their counters in one single tree. The worst time complexity is then \( \mathcal{O}(k \cdot m \cdot |\Sigma|) \), where \( m = \sum_{i=1}^{l} |T_i| \). In the following, for simplicity’s sake, we will consider the case of analyzing one single text, even if practical applications will generally consider a set of texts.

Moreover note that in the definition of the k-trie we have considered the minimal information associated to each node. In describing how to use it for analyzing the text \( T \), we will actually associate other information to its nodes, such as expected frequencies of substrings and scores.

### 3 Pruning the k-trie

As pointed out in the previous section, a k-trie \( T_k \) contains information on substrings of length at most \( k \) in \( T \), but it is unpractically costly in terms of space for a large alphabet and text. Therefore in order to use it, it could be useful to find ways for pruning the tree, while maintaining the information we are interested in, that is the number of occurrences of the “most interesting” substrings. To this end, we can point out a few properties of the k-trie which can lead to pruning heuristics.

Let \( N \) be the set of nodes in \( T_k \) and let \( str_{T_k} : N \to \Sigma^* \) be a function which takes the path from the root to a node \( n \) in \( T_k \) and returns the string obtained by concatenating the symbols associated to the nodes of the path. Note that \( str \) is an injective function since different nodes in \( T_k \) correspond to different strings. For the sake of simplicity let us denote all the nodes at level \( j \) with \( lev[j] \) and let \( son_of \) be a function which, when applied to a node, produces all its sons. For a node \( n \) we denote with \( c(n) \) the value of the counter associated to it. Then the following properties hold in a k-trie.

1. **The sum of the counters associated to all the nodes in level \( j \) is almost equal to the length of the text:**

   \[
   m - (k - 1) \leq \sum_{n \in lev[j]} c(n) \leq m, \text{ for any } j \in [1, k].
   \]

   Actually, for level 1 the sum is exactly \( m \), for level 2 is \( m - 1 \), \ldots, for level \( j \) is \( m - j + 1 \), \ldots, and for level \( k \) is \( m - k + 1 \).

   This is due to the fact that \( k \leq m \) and that the first \( k - 1 \) symbols in \( T \) will not be counted in all the levels of the tree. However, note that when \( k \) is much smaller than \( m \), the interval \([m - k, m]\) is also very small, hence we can say that the sum of the counters associated to all the nodes at any level \( j \) approximates the length of the text.

   Nevertheless, when a k-trie represents \( l \) texts, \( T_1, T_2, \ldots, T_l \), this effect can become more relevant since it is increased by a factor \( l \):

   \[
   m - (k - 1)l \leq \sum_{n \in lev[j]} c(n) \leq m, \text{ for any } j \in [1, k].
   \]
where \( m = \sum_{i=1}^{l} |T_i| \). For each level \( j \) the sum of the counters is exactly \( m - (j - 1)l \).

2. The counter values are not increasing along a path. In particular, if a node \( n_j \) can be reached from the root through a path \( n_1 n_2 \ldots n_j \), then

\[
c(n_h) \geq c(n_l), \quad \text{for all } 1 \leq h < l \leq j.
\]

This is due to the fact that the nodes along the path \( n_1 \ldots n_j \) correspond to prefixes of the substring \( str_{T_k}(n_j) \) associated to the path itself, and clearly smaller prefixes of a substring are more frequent in \( T \) than longer ones.

3. The counter value of a node \( n \) at level \( j \) is greater or equal to the sum of the counters associated to its sons:

\[
c(n) \geq \sum_{n_k \in \text{son of } (n)} c(n_k), \quad \text{for any } j \in [1, k - 1].
\]

In fact, the sons of \( n \) represent all the possible substrings in \( T \) which are concatenation of the substring \( s = str_{T_k}(n) \) and of a symbol \( a \in \Sigma \). Clearly, the occurrences of any string \( sa \) in the text \( T \) cannot exceed the occurrences of the string \( s \) itself.

These properties suggest a way to reduce the size of the k-trie \( T_k \), while keeping the substrings in the text \( T \) which have an "interesting" frequency. Basically we could apply a pruning strategy by following the steps:

1. determine a threshold with respect to which the "interesting" frequency is shown;
2. prune the tree depending on a local or global pruning condition.

By local we mean a pruning condition depending only on the currently visited node. In contrast, by global we mean a pruning condition depending on the path of the visit or on some (statistical) relation among substrings.

When pruning the tree for reducing its complexity, we do not want to produce isolated nodes. Hence, a node is always pruned with all its descendants. In this way we obtain a pruned k-trie \( T'_k \). A simple procedure for pruning the tree top-down is given below. It is applied to all the sons of the root of \( T_k \).

**Prune_tree(n);**

{ n is the node from where the top-down pruning starts }

if Test(n) then Prune(n)

{ if test is positive the node is pruned with all its descendants }

else for all \( n_i \) son_of(n) do Prune_tree(n_i);

{ son_of applied to a leaf produces an empty set of sons}

end;

In the case of a local pruning condition, the function Test depends only on some input parameters (such as a threshold and a tolerance) and on the current node
For a global condition, \textsc{Test} will be more complex and it could require to keep track of previous evaluations done during the pruning of the tree. Clearly, this has to be tuned with respect to the particular application: actually the pruning strategy is the most critical step in the analysis of \( T \).

For example we could know or assume how the symbols in \( \Sigma \) should be distributed in \( T \). For simplicity let us assume a uniform distribution. This means that each symbol in the alphabet should appear \( \delta_1 = m/|\Sigma| \) times at each node at level 1, \( \delta_2 = (m - 1)/|\Sigma|^2 \) times at each node at level 2, .. and generally \( \delta_j = (m - j + 1)/|\Sigma|^j \) times at each node at level \( j \). If we have \( l \) texts, \( T_1, T_2, \ldots, T_l \) and \( m = \sum_{i=1}^{l} |T_i| \), we would have \( \delta_j = (m - (j - 1)l)/|\Sigma|^j \) at each node at level \( j \). On this basis we can derive a uniform pruning strategy by using the value \( \delta_j \) as a threshold for pruning the tree at level \( j \). This can be obtained by traversing the tree in any order and cutting all nodes \( n \) of level \( j \) satisfying the pruning condition:

\[
\begin{align*}
\phi(n) &\leq \delta_j \cdot \rho_{\text{high}}, \quad \text{if we are interested in over-represented substrings, or} \\
\phi(n) &\geq \delta_j \cdot \rho_{\text{low}}, \quad \text{if we are interested in under-represented substrings,}
\end{align*}
\]

where \( \rho_{\text{low}} \) and \( \rho_{\text{high}} \) are some positive tolerance values dependent on the application. This is clearly a local pruning condition. Since the node \( n \) is pruned with all its descendants, pruning \( n \) can eliminate many substrings and in particular some "interesting" ones. Special care has to be put in avoiding indirect pruning of such "interesting" substrings, for example \textsc{Test}, in procedure \textsc{Prune Tree}, could also check the descendants of \( n \) and give a warning when some of them are "interesting".
Example 2. Let $\Sigma$ be \{A, T, C, G\} and $m = 640$ for one single text $T$. We are interested in substrings of $T$ which occur more often than expected, i.e. for which $c(n) > \delta_j \cdot \rho_{\text{high}}$ at each level $j$. Let $T_k$ be the tree in the previous figure, of which we show only the first two levels.

Let us prune the tree with the uniform pruning strategy, we are interested in keeping only the substrings which are over the threshold, where $\delta_1 = 640/4 = 160$, $\delta_2 = 639/16 = 39.9$ and tolerance $\rho_{\text{high}} = 1.2$. At level one the pruning eliminates all the nodes with a counter less or equal to 192 and all their descendants. When considering the remaining nodes at level two, it keeps the ones with a counter greater than 47.9. Hence we obtain the following pruned tree where no “interesting” node has been pruned.

With the same assumption of a uniform distribution of the symbols we could obtain a tuned pruning strategy by applying the previous local pruning condition at level 1 and the following pruning condition to the other levels:

\[ c(n_i) \leq \frac{c(n)}{|\Sigma|} \cdot \rho_{\text{high}} \quad \text{or} \quad c(n_i) \geq \frac{c(n)}{|\Sigma|} \cdot \rho_{\text{low}} \]

where $n$ is the father of node $n_i$. This is a global pruning strategy which can be used in order to point out discontinuities in the distribution of the symbols in the substrings (for example to determine the length of interesting substrings).

Example 3. Let us prune the tree $T_k$ of Example 2 with a tuned pruning strategy. Let us assume to be interested in discontinuities in symbol distribution when they exceed the expected number of symbols, i.e. the pruning condition is $c(n_i) \leq \frac{c(n)}{|\Sigma|} \cdot \rho_{\text{high}}$, with $\rho_{\text{high}} = 1.2$. We obtain the following tree.

If we are interested in discontinuities in the distribution of the symbols when the symbols are less than expected, i.e. the pruning condition is $c(n_i) \geq \frac{c(n)}{|\Sigma|} \cdot \rho_{\text{low}}$, with $\rho_{\text{low}} = 0.8$, we obtain the following tree.
Actually the pruning could be only virtual; an alternative could be to sort (into increasing or decreasing order of the node counters) the paths in the tree by levels. Instead of pruning the nodes we could simply mark the ones which are not pruned. Then the visit of the tree can be guided and limited by the paths ordering and the pruning conditions. This allows one to try various hypotheses corresponding to different thresholds and pruning conditions without modifying the initial tree.

**Example 4.** Let us consider again the tree \( T_k \) of Example 2. We sort its paths into decreasing order by level, thus obtaining the following tree.

![Tree Diagram]

Given the same threshold of Example 2, we can traverse the tree (only partially thanks to the ordering and the pruning conditions which stop the traversal) and mark the nodes which are not pruned.

Both a local pruning strategy and sorting the k-trie has a time complexity, in the worst case, which is linear in the dimension of the tree. For a global pruning strategy we should analyze the complexity case by case.

A further observation regards how to give in input our thresholds \( \delta_j \). We can give them as tables \( \Sigma \times \text{Lev} \), where \( \text{Lev} = [1, k] \). This clearly allows one to express the dependency of such thresholds on the specific symbol at a specific level, but not a dependency on adjacent symbols in the substring.

Another possibility is that we know (or assume) the expected frequencies of k-substrings. In this case the thresholds can be associated to the leaves of the

---

**Diagram Description:**

- The tree diagram represents a binary tree structure with nodes labeled with symbols and node counts.
- The tree is rooted and has a hierarchical structure with branches labeled by symbols and their counts.
- The nodes are marked to indicate whether they are pruned or not.

---

**Note:** The diagram should be interpreted as a visual representation of the tree described in the text. The actual tree structure is not transcribed here but is implied by the description.
tree and we could use a bottom-up pruning strategy. First the thresholds are 
used to prune the border of the tree, since the leaves at level \( k \) represent exactly 
the substrings of length \( k \), then some global pruning condition can be used to 
propagate the pruning in a bottom-up way. The propagation condition could 
simply mean to prune each node (father) which has no son anymore, or it could 
be some more sophisticated condition.

4 Searching the tree with an abstract string or an 
abstract alphabet

A typical operation we want to do on the k-trie is to search it for interesting 
substrings. In the following we will denote with \( T_k \) both the original and the 
pruned tree since we assume that both of them have the same data structure 
and similar properties. Pruning is then somehow orthogonal to searching and 
analyzing the tree.

We want to search \( T \) in a blind way, that is we assume to ignore which are 
the interesting substrings and we want to try various possibilities by allowing 
some symbol abstractions. There are basically two ways for doing this:

1. we could search for abstractions of a substring, obtained by abstracting some 
symbols in the substring;
2. we could abstract the alphabet \( \Sigma \), for example by assuming that two or more 
symbols are similar (or indistinguishable) for the purpose of our search.

We discuss the two techniques in general.

4.1 Abstracting the string to be searched

Let us assume that we are searching for a substring containing a symbol \( a \) in 
a specified position. In our search we could wish to "make more abstract" the 
substring and to allow also some variants for \( a \) in that position. All the possible 
abstractions of a symbol \( a \in \Sigma \) are given by \( \text{Abs}_\Sigma(a) = \{S\mid S \subseteq \Sigma \text{ and } a \in S\} \).

For example let \( \Sigma \) be \{\( A, T, C, G \)\} and consider the symbol \( A \). Then \( \text{Abs}_\Sigma(A) = \{[A], [AT], [AC], [AG], [ACG], [ACT], [AGT], [ACGT]\} \), where for simplicity we 
denote sets of symbols in \( \Sigma \) with square brackets and without commas, that is \( [ACT] \) stands for \{\( A, C, T \)\}. Note that \( [ACT] \) can be written also as a regular 
expression: \( A + C + T \). The abstractions of an unknown symbol in \( \Sigma \) are then 
given by \( \mathcal{P}(\Sigma) \setminus \emptyset \). Such abstractions can be partially ordered in a natural way 
by set inclusion. We say that a symbol abstraction \( S \) is strictly more abstract 
than another symbol abstraction, \( S' \), iff \( S \supset S' \) and we denote it as \( S > S' \), or 
we can also say that \( S' \) is strictly more precise than \( S \). In the previous example: 
\( [A] < [AT] \), \( [ACG] > [AC] \) and \( [ACGT] \) is strictly more abstract than any other 
abstraction of \( A \).

For abstracting a string we use a particular kind of regular expression corre-
spounding to a finite set of strings.
Definition 5. A pattern on an alphabet $\Sigma$ is a finite concatenation of non-empty subsets of $\Sigma$

\[ p = S_1S_2..S_h, \text{ where } S_j \subseteq \Sigma, S_j \neq \emptyset \text{ and } j \in [1, h]. \]

The length of the pattern is the length of the concatenation.

Searching such a pattern in a text corresponds to a pattern matching operation where more than one symbol is allowed in each position. In the pattern discovery literature such patterns are often called rigid patterns since their length is fixed.

Example 6. Let us consider $\Sigma = \{A, T, C, G\}$ and $p = [AT][AC][CG]$. $p$ is a pattern of length 3 corresponding to the regular expression $(A + T)(A + C)(C + G)$. Hence, the strings abstracted by $p$ are all the strings in the regular set $\{AAC, AAG, ACC, ACG, TAC, TAG, TCC, TCG\}$.

Let $p$ be a pattern on $\Sigma$ of length $h$, that is $p = S_1S_2..S_h$, where $S_j \subseteq \Sigma$ and $j \in [1, h]$. $p$ is a regular expression and it defines a (regular) set of strings on $\Sigma$. How can we determine if $p$ occurs in a k-trie $T_k$ and how many occurrences it has?

Definition 7. Let $T_k$ be a k-trie of a text $T$ on an alphabet $\Sigma$ and $p$ be a pattern on $\Sigma$ of length $h$, with $h \leq k$.

An occurrence of $p$ in $T_k$ is a path $n_1n_2..n_h$ from the root in $T_k$ such that $str_{T_k}(n_h)$ belongs to the regular set defined by $p$.

We say that $p$ occurs in $T_k$ if and only if there exists at least one occurrence of $p$ in $T_k$.

In order to determine how many occurrences $p$ has in $T_k$, we need to traverse all the paths of length $h$ in the tree which correspond to strings in the set $p$, and to sum up the counters associated to the last nodes of such paths. Let $c(p)$ be the number of occurrences of $p$ in $T_k$ then

\[ c(p) = \sum_{str_{T_k}(n_h) \in p} c(n_h). \]

In order to determine if a pattern occurs in the tree, in the worst case we have to traverse all the tree. If we try many patterns, we will have many traversals. When searching in a blind way we would like to try all the abstractions of a substring or even all the possible abstractions of any substring. The set of all the possible abstractions of any string of length $k$ in $\Sigma$ is $(P(\Sigma) - \emptyset)^k$, which means $(2^{|\Sigma|} - 1)^k$ patterns. In general this is an unfeasible number to deal with.

One possibility for reducing this complexity is to use the information in the (pruned) k-trie $T_k$ to reduce the number of patterns to search to those which are likely to be found in $T_k$. For example, if we have $\Sigma = \{A, T, C, G\}$ and only the symbols $A$ and $C$ are in the first level of $T_k$, then the first symbol in the
substrings can only be abstracted as \([A]\), \([C]\) or \([AC]\). This allows us to discard 12 of the \((2^4 - 1) = 15\) possibilities of abstracting the symbol in the first position of the searched substring. All the other possibilities for the first symbol of a pattern would either lead to a failure in the search in \(T_k\) or contain redundant symbols.

From this observation we can define a fast technique to discard patterns which cannot be found in the tree, thus allowing for a significant reduction in complexity by avoiding useless searches.

Remark 8 (Necessary condition for occurring). Let \(T_k\) be a (pruned) \(k\)-trie and let \(p = S_1S_2..S_h\) be a pattern, with \(h \leq k\). For any \(j = [1, h]\), let \(U_j\) be the set of the symbols of all the nodes in \(lev[j]\).

If \(p\) occurs in \(T_k\), then \(U_j \cap S_j \neq \emptyset\) for any \(j \in [1, h]\).

From this remark, we could derive a sufficient condition for a pattern not to occur in \(T_k\). But since we want to exclude as many patterns as possible from our search, we focus on patterns which occur in \(T_k\) and somehow are minimal w.r.t. this property, that is we want to avoid searching for redundant symbols.

Definition 9. Let \(T_k\) be a (pruned) \(k\)-trie of a text \(T\) on an alphabet \(\Sigma\). A pattern \(p = S_1S_2..S_h\) is compatible with \(T_k\) iff for all \(j \in [1, h]\) and for any symbol \(s \in S_j\), there is at least one occurrence of \(p\) in \(T_k\), \(n_1n_2..n_h\), such that \(str_{T_k}(n_j) = s_1s_2..s_{j-1}s\).

Compatibility is a stronger property than occurring, namely if \(p\) is compatible with \(T_k\), then \(p\) occurs in \(T_k\). Moreover a pattern which is compatible does not have ”redundant” symbols w.r.t. \(T_k\), that is any symbol in \(p\) is present, in the specified position, at least in one occurrence of the pattern in \(T_k\).

Remark 10 (Necessary condition for compatibility). Let \(T_k\) be a (pruned) \(k\)-trie and let \(p = S_1S_2..S_h\) be a pattern, with \(h \leq k\). For any \(j = [1, h]\), let \(U_j\) be the set of the symbols of all the nodes in \(lev[j]\).

If \(p\) is compatible with \(T_k\), then \(U_j \supseteq S_j\) for any \(j \in [1, h]\).

From this remark we derive a sufficient condition for excluding patterns from our search.

Remark 11 (Sufficient condition for not being compatible). Let \(T_k\) be a (pruned) \(k\)-trie and let \(p = S_1S_2..S_h\) be a pattern, with \(h \leq k\). For any \(j = [1, h]\), let \(U_j\) be the set of the symbols of all the nodes in \(lev[j]\).

If there exists \(j \in [1, h]\) such that \(U_j \nsubseteq S_j\), then \(p\) is not compatible with \(T_k\).

Note that this condition is only sufficient for not being compatible. In fact, there could be patterns which are not compatible, but satisfy condition 11.

Example 12. Consider the 2-trie of Example 1. Consider now the union of the symbols at each level in the tree: \(U_1 = \{A, C, G\}\) and \(U_2 = \{A, C, G\}\).

– Let us consider the pattern of length 2, \(p = [AG][G]\). It occurs in \(T_k\) and it is also compatible with \(T_k\), since both \(AG\) and \(GG\) occur in \(T_k\). It does not satisfy condition 11.
– Let us consider the pattern \( p = [AG][AT] \). It occurs in \( T_k \) but it is not compatible with \( T_k \). In fact GA occurs in \( T_k \) but \( U_2 \not\supseteq \{A,T\} \), since \( T \notin U_2 \). The symbol \( T \) in the second position of the pattern is redundant w.r.t. the 2-trie. It satisfies condition 11.
– Let us consider now the pattern \( p = [AC][A] \). This pattern does not satisfy the sufficient condition 11 but it is not compatible with the 2-trie since it does not occur in it.

We can define a simple procedure \textsc{Check\_compatible} for checking the sufficient condition for not being compatible with \( T_k \) on a pattern \( p \).

\textsc{Check\_compatible}(T_k, \( p \));
\{ \( p = S_1S_2..S_h \); if \(|p| = h \) and \( h < k \), then \( S_{h+1} = .. = S_{k} = \emptyset \) \}
\textbf{for } \( j := 1 \textbf{ to } k \textbf{ do} \) 
\begin{align*}
& \text{Build}(U_j); \{ \text{build the set of all the symbols at level } j \} \\
& \text{if } U_j \not\supseteq S_j \text{ then exit-fail}; \{ \text{p is not compatible} \}
\end{align*}
\textbf{end;}

An efficient way to implement the procedure is to represent each subset \( S_i \) of the alphabet \( \Sigma \) as a memory word and each symbol in \( \Sigma \) as a bit. The bits corresponding to symbols in \( S_i \) are set to 1, the other ones are set to 0. Hence the check \( U_j \not\supseteq S_j \) becomes a logical operation on words: \( U_j \lor S_j \neq U_j \).

The k-trie can be abstractly represented by a sequence of \( k \) memory words, the \textit{most abstract pattern compatible with the k-trie}, which may be used to filter the patterns and discard the ones which are not compatible with it.

**Definition 13.** Let \( T_k \) be a (pruned) k-trie of a text \( T \) on an alphabet \( \Sigma \).
The most abstract pattern compatible with \( T_k \), denoted with \( \mathcal{U}(T_k) \), is a sequence of \( k \) sets of symbols, \( U_1U_2..U_k \), where each \( U_j \), for \( j \in [1,k] \), is the set of symbols at level \( j \) in \( T_k \).

Note that for a large text \( T \) and a small alphabet \( \Sigma \), each \( U_j \) will be close to \( \Sigma \). In case of a large alphabet or a small text, the most abstract pattern expressed by \( T_k \) will be more useful.

The sufficient condition 11 can be used also in a different way, namely to give a refinement technique for the pattern \( p \) we are searching for in the tree.

The partial order on symbol abstractions induces a partial order on patterns,

**Definition 14.** Let \( p = S_1S_2..S_h \) and \( p' = S'_1S'_2..S'_h \) be two patterns of the same length. \( p' \) is strictly more abstract than \( p \) iff for all \( j \in [1,h] \), \( S'_j \geq S_j \) and for at least one \( i \in [1,h] \), \( S'_i > S_i \).

Equivalently we may say that \( p \) is strictly more precise than \( p' \).
Given a pattern \( p \) to be searched in \( T_k \), in general there will be many patterns which are strictly more abstract than \( p \). Some of them will satisfy the sufficient condition 11, and introduce some redundancy, some will not satisfy it, thus adding only symbols which might produce further occurrences of the pattern.

We can exploit this observation for “tuning” a pattern \( p = S_1S_2..S_h \) to be searched. In fact, for any \( j \in [1, k] \), we can tune the set of symbols \( S_j \) in two different ways:

1. by removing redundant symbols signalled by the sufficient condition 11, namely symbols in \( S_j - U_j \);
2. by (incrementally) adding symbols which do not cause the sufficient condition 11 to be verified, namely symbols in \( U_j - S_j \), i.e. symbols which can produce further occurrences of the pattern.

The first operation for ”tuning” the pattern will produce a new pattern \( p' \) which is strictly more precise than \( p \).

The second operation will produce a new pattern \( p' \) which is strictly more abstract than \( p \) but does not introduce redundant symbols.

Finally we can also use the sufficient condition 11 as a simple guide in generating all the patterns to be searched. Let \( U(T_k) = U_1U_2..U_k \) be the most abstract pattern expressed by \( T_k \), then \( P(T_k) = \{ p \mid p = S_1S_2..S_h \text{ and } S_j \in (P(U_j) - \emptyset), \text{ for } j \in [1, j]\} \) is the set of all patterns which do not satisfy the sufficient condition 11. These patterns could be searched exhaustively. After a smart pruning of the tree, this set will hopefully be much smaller than the set of all patterns on \( \Sigma \).

### 4.2 Abstracting the alphabet

In this section we discuss a second possibility for using abstraction in a blind search: we could abstract the alphabet \( \Sigma \), for example by assuming that some symbols are similar (or undistinguishable) for the purpose of our search. This is a very reasonable assumption, in particular for large alphabets such as the one of the aminoacids. For example we could know that aminoacids with similar properties can sometimes be substituted one for each other in some protein, hence while looking in our sequence \( T \) for interesting substrings we could exploit this knowledge. More generally, in a blind search we could want to try different abstractions of \( \Sigma \).

Abstracting the alphabet \( \Sigma \) means translating it into a new alphabet \( \Delta \) that should reflect the knowledge or assumptions on symbols similarity. Such similarity information on \( \Sigma \) can be formalized by means of an equivalence relation \( R \), grouping together similar symbols. This induces a partition of \( \Sigma \) into equivalence classes which can be interpreted as the symbols of a new (abstract) alphabet \( \Delta \) isomorphic to \( \Sigma / R \). The alphabet translation is then a function \( \tau \) mapping each symbol in \( \Sigma \) to the corresponding symbol in \( \Delta \) representing the equivalent class it belongs to.
Example 15. Let $\Sigma = \{A,T,C,G\}$ and let us assume that we want to abstract it by expressing that $C$ and $G$ are similar for the purpose of our analysis. We define a relation $R$ which is reflexive, symmetric and transitive and such that $C \mathrel{R} G$. This produces the partition $\{(C,G), \{A\}, \{T\}\}$ in $\Sigma$. Hence $\Delta$ can be any set isomorphic to $\{[CG], [A], [T]\}$.

The translation function $\tau$, associated to $\Delta$, is defined by: $\tau(A) = [A], \tau(T) = [T], \tau(C) = [CG], \tau(G) = [CG]$. It extends to any string in $\Sigma^*$, for example $\tau(\text{ATCCGA}) = [A][T][CG][CG][CG][A]$. In this way we give a translation from a string in $\Sigma$ into an abstract string in $\Delta$.

In the context of pattern discovery, the proposed translation $\tau$ of the original alphabet $\Sigma$ into a more abstract alphabet $\Delta$ can be applied in two different ways:

- we can first translate the text $T$ into $\Delta$, then build the k-trie, possibly prune it, and then search it for (sub)strings in $\Delta$;
- we can first build the k-trie $T_k$, translate it and then search it for (sub)strings in $\Delta$.

The first alternative requires a linear scan of $T$ in order to produce the translation. For each translation, we have to build $T_k$, possibly prune it, and then search it.

The second alternative requires to translate all the symbols labelling the nodes in $T_k$, to compact the resulting tree, since in the translated tree each node can have multiple sons labelled with the same symbol, and then to search it. Translating and compacting the tree is linear in the number of nodes of the tree.

The main difference between the two alternatives regards the combination between pruning and abstraction. In the latter either we prune the original tree once and then give different translations of it, or we prune it after the translation. In the first alternative we can prune the tree only after each translation. Hence the choice is very much dependent on the application. The first alternative is simpler but, when the pruning is independent from the abstraction choice on $\Sigma$, the second alternative could be more convenient.

Note that abstracting the alphabet and abstracting the string to be searched can be usefully combined. After abstracting the alphabet, the search for substrings can be optimized with the ideas shown in Section 4.1.

5 Scoring the k-trie with respect to expected frequencies

In Section 3 we discussed how to prune the tree in order to make it more handy, while keeping it connected and still containing the more interesting substrings, that is the more (or less) frequent ones with respect to given thresholds. If we assume to know the expected frequency of any substring, we can also associate it to the corresponding node in the tree and define the "interest" of each node (substring) in terms of an associated relevance score, which compares its observed frequency to its expected frequency.
Please notice that in the following we use the terms expected frequency and observed frequency of a string in \( T \), but we actually intend expected occurrences and observed occurrences of the substrings in \( T \).

In the uniform pruning strategy shown in Example 2 we actually compare the observed frequency with the expected one in the very simple hypotheses that all the symbols in \( \Sigma \) have the same probability in \( T \) and that, in any substring, the symbols are independent. In fact, with such assumptions the expected frequency of a substring \( s = s_1 s_2 \ldots s_h \) in \( T \) is

\[
\text{Exp}(s) = (m - h + 1) \left( \frac{1}{|\Sigma|} \right)^h = (m - h + 1) / |\Sigma|^h
\]

where \( m = |T| \) and \( (1/|\Sigma|) \) is the probability associated to each symbol. \( \text{Exp}(s) \) corresponds exactly to the thresholds we used in Example 2 for pruning the tree.

In the literature many different ways to associate a relevance measure to strings have been proposed, see [18] for a recent survey. The simplest choices are obtained by assuming independence of the symbols in each substring and by assuming that the probability associated to each symbol in \( \Sigma \) does not vary in \( T \). In this case, the probability of the substring \( s = s_1 s_2 \ldots s_h \) is the product of the probability of each symbol in the substring:

\[
pr(s) = \prod_{i=1}^{h} pr(s_i)
\]

and the expected frequency is:

\[
\text{Exp}(s) = (m - h + 1) \prod_{i=1}^{h} pr(s_i).
\]

When we consider \( T_1, \ldots, T_l \) input texts of size \( m_1, \ldots, m_l \), respectively, the expected frequency of \( s \) generalizes to

\[
\text{Exp}(s) = (m_1 - h + 1) \prod_{i=1}^{h} pr(s_i) + \ldots + (m_l - h + 1) \prod_{i=1}^{h} pr(s_i) = \\
\sum_{j=1}^{l} (m_j - h + 1) \prod_{i=1}^{h} pr(s_i)
\]

The relevance of a substring can then be measured by various scores. The simplest ones are the following:

\[
z_1(s) = c(n) - \text{Exp}(s)
\]

\[
z_2(s) = \frac{c(n)}{\text{Exp}(s)}
\]
\[
    z_3(s) = \frac{(c(n) - \text{Exp}(s))^2}{\text{Exp}(s)}
\]

where \( n \) is the node corresponding to substring \( s \), that is \( \text{str}_{Tk}(n) = s \), \( c(n) \) is the observed frequency of \( s \), i.e. the number of its occurrences in \( T \), and \( \text{Exp}(s) \) is the expected frequency of \( s \). \( z_3(s) \) is the \textit{chi-square score}.

Depending on the application, we should be able to choose the relevance score which is more useful for identifying the substrings which are "interesting" for our purposes.

More generally, any \textit{relevance score} for a substring \( s \) associated to a node \( n \) will use a relation \( R \) for comparing the observed frequency \( c(n) \) of \( s \) with some expected frequency \( \text{Exp}(s) \) which will depend on our assumptions or knowledge on \( T \). For example, if we drop the simplifying assumption of independence of symbols in \( s \), \( \text{Exp}(s) \) could be estimated by analyzing a sample set of substrings (or texts) with an HMM.

If we now consider the k-trie annotated in each node with a relevance score, we may analyze it with respect to such a score. In general only sparse nodes in the tree will have high (low) scores. These correspond to the relevant substrings.

### 5.1 Frequency behaviour with respect to symbol concatenation

It is interesting to characterize how the observed and expected frequencies of substrings vary depending on their length. That is, we can state how symbol concatenation affects the expected frequency of substrings. Let us consider the two substrings \( \text{str}_{Tk}(n) = s_1s_2..s_{h-1}, \) associated to the node \( n \) at level \( h - 1 \) in the k-trie, and \( \text{str}_{Tk}(n') = s_1s_2..s_{h-1}s_{h}, \) associated to the node \( n' \), son of \( n \). Let \( |T| = m \). Then

\[
    \text{Exp}(n) = (m - h + 2) \prod_{i=1}^{h-1} \text{pr}(s_i)
\]

and

\[
    \text{Exp}(n') = (m - h + 1) \prod_{i=1}^{h} \text{pr}(s_i).
\]

Hence we can compute the expected frequency of a string incrementally along the paths of the k-trie by adding one symbol at a time. In fact

\[
    \text{Exp}(n') = \text{Exp}(n) \, \text{pr}(s_h) \frac{(m - h + 1)}{(m - h + 2)}
\]

This shows that, \textit{with our hypotheses, a traversal of the tree is sufficient for annotating it with the expected frequency of each substring}, that is we can add this information to each node in linear time w.r.t the number of nodes in the tree. The previous scores depend only on the observed and the expected frequency of each node and they can be evaluated in constant time. Hence \textit{also the score}
annotation can be associated to the nodes of the tree in the same traversal, that is in linear time w.r.t the number of nodes in the tree.

A deep investigation on how to annotate the nodes of a suffix tree with their expected values, variances and scores of significance with respect to various probabilistic models can be found in [1, 2].

Moreover, since \(0 \leq \text{pr}(s_h)^{(m-h+1)\over(m-h+2)} \leq 1\), we observe that with our hypotheses the expected frequency cannot increase along the paths of the k-trie. The same holds for the observed frequencies \(c(n)\) and \(c(n')\), as we noted in Section 3.

When we consider \(T_1, \ldots, T_l\) input texts of size \(m_1, \ldots, m_l\), respectively, we obtain:

\[
\text{Exp}(n) = \sum_{j=1}^{l} (m_j - h + 2) \prod_{i=1}^{h-1} \text{pr}(s_i)
\]

and

\[
\text{Exp}(n') = \sum_{j=1}^{l} (m_j - h + 1) \prod_{i=1}^{h} \text{pr}(s_i) =
\]

that is

\[
\text{Exp}(n') = \text{Exp}(n) \cdot \text{pr}(s_h)^{\sum_{j=1}^{l} (m_j - h + 1) \over \sum_{j=1}^{l} (m_j - h + 2)}
\]

Since \(0 \leq \text{pr}(s_h)^{\sum_{j=1}^{l} (m_j - h + 1) \over \sum_{j=1}^{l} (m_j - h + 2)} \leq 1\), also in this case the expected frequency cannot increase along the paths of the k-trie.

**Definition 16.** Consider an alphabet \(\Sigma\) and a binary operation on strings \(\text{op} : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*\). Let \(f\) be a function mapping strings to reals, \(f : \Sigma^* \rightarrow \mathbb{R}\). We say that:

- \(f\) is monotone non-increasing w.r.t. \(\text{op}\) iff \(f(s_1) \geq f(s_1 \text{ op } s_2)\),
- \(f\) is monotone non-decreasing w.r.t. \(\text{op}\) iff \(f(s_1) \leq f(s_1 \text{ op } s_2)\),

for all strings \(s_1\) and \(s_2\) in \(\Sigma^*\).

Then both the expected and the observed frequencies are monotone non-increasing functions w.r.t. symbol concatenation.

In the following, we analyze how the previous scores vary with respect to both string concatenation and string union, which is the basis for string abstraction. We denote with \(D(n)\) the difference between the observed and the expected frequency of the string \(\text{str}_{T_k}(n)\) corresponding to node \(n\) in \(T_k\), that is \(D(n) = c(n) - \text{Exp}(n)\). We say that the string is over-represented in \(T\) (resp. under-represented in \(T\)) if \(D(n) > 0\) (resp. \(D(n) < 0\)).
5.2 Score behaviour with respect to symbol concatenation

In this section we analyze how the considered relevance scores varies with respect to symbol concatenation.

Consider again the strings $str_{T_k}(n)$ and $str_{T_k}(n')$ and their corresponding nodes $n$, $n'$ in the tree, with $n'$ son of $n$. We use the following notation:

- $\Delta_h^c$ denotes the decrease of the observed frequency between the two strings, that is $\Delta_h^c = c(n) - c(n')$;
- $\Delta_h^E$ denotes the decrease of the expected frequency between the two strings, that is $\Delta_h^E = \text{Exp}(n) - \text{Exp}(n') = \text{Exp}(n)(1 - pr(s_h))$, where $pr(s_h)$ is the probability of symbol $s_h$.

Hence, $\Delta_h^c = z_1(n) - (\Delta_h^c - \Delta_h^E)$ and $\Delta_h^E = z_1(n) - (\Delta_h^c - \Delta_h^E)$. Therefore, if $\Delta_h^c > \Delta_h^E$, the score $z_1$ decreases by symbol concatenation, if $\Delta_h^c < \Delta_h^E$ it increases.

Then, the score decreases (resp. increases) if the concatenation of the symbol $s_h$ to $str_{T_k}(n)$ produces a bigger (resp. smaller) decrease in the observed frequency than in the expected one.

- The first score can have both a positive or a negative value, this depends respectively if the string is over- or under-represented in $T$.

We have

$z_1(n) = c(n) - \text{Exp}(n)$

and

$z_1(n') = c(n') - \text{Exp}(n') = (c(n) - \Delta_h^c) - (\text{Exp}(n) - \Delta_h^E) = (c(n) - \text{Exp}(n)) - (\Delta_h^c - \Delta_h^E) = z_1(n) - (\Delta_h^c - \Delta_h^E)$.

Hence, if $\Delta_h^c > \Delta_h^E$, the score $z_1$ decreases by symbol concatenation, if $\Delta_h^c < \Delta_h^E$ it increases.

For simplicity we exclude the limit case when $\Delta_h^c = 0$ and we consider the nodes $n$ and $n'$ instead of the corresponding substrings.

- The second score has always a positive value, it is greater than 1 for over-represented strings and less than 1 for under-represented ones.

We have $z_2(n) = \frac{c(n)}{\text{Exp}(n)}$ and

$z_2(n') = \frac{c(n')}{\text{Exp}(n')} = \frac{c(n) - \Delta_h^c}{\text{Exp}(n) - \Delta_h^E}$.

In this case, $z_2(n') \geq z_2(n) \iff \frac{c(n) - \Delta_h^c}{\text{Exp}(n) - \Delta_h^E} \geq \frac{c(n)}{\text{Exp}(n)}$.

that is iff

$(c(n) - \Delta_h^c)\text{Exp}(n) \geq c(n)(\text{Exp}(n) - \Delta_h^E)$.

After simplifying we get

$-\Delta_h^c \text{Exp}(n) \geq -\Delta_h^E c(n)$.
that is
\[
\frac{\Delta c}{\Delta E} \leq \frac{c(n)}{\text{Exp}(n)}
\]
hence if \(\frac{\Delta c}{\Delta E} < z_2(n)\), the score \(z_2\) increases by symbol concatenation, if \(\frac{\Delta c}{\Delta E} > z_2(n)\), it decreases.

– The third score has always a positive value, the greater the more over-represented or under-represented is the string to which it is applied.

We have
\[
z_3(n) = \frac{D(n)^2}{\text{Exp}(n)} \quad \text{and} \quad z_3(n') = \frac{D(n')^2}{\text{Exp}(n')}.
\]
Since \(\text{Exp}(n) \geq \text{Exp}(n')\), we have a sufficient condition for score increasing, which is \(D(n)^2 \leq D(n')^2\).

This condition can be expressed also as \(|c(n) - \text{Exp}(n)| \leq |c(n') - \text{Exp}(n')|\).
The other case, \(D(n)^2 > D(n')^2\), can produce either score increasing or decreasing.

From all this discussion we realize that even these three simple relevance scores vary w.r.t. symbol concatenation in a rather sophisticated way. In particular, they are not monotone w.r.t. symbol concatenation, that is along the paths in the \(k\)-trie. This has two important consequences:

1. pruning the tree w.r.t. such relevance scores can be destructive since in general only sparse nodes will have high (low) scores. Hence, by just eliminating uninteresting nodes, one would loose the tree structure or produce isolated nodes;
2. the scoring can be computed incrementally along the paths of the tree; but the interesting strings cannot be determined incrementally since their relevance varies along the paths.

5.3 Score behaviour with respect to string union

In order to study how the scores are affected by string abstraction we recall that a pattern is just a particular set of strings and that the process of abstracting a string consists in applying a union operation with other strings satisfying some shape conditions. Hence, we need to analyze how the scores we consider are affected by adding a string to a set of strings, that is by the union operation.

First of all we extend appropriately the notion of monotonicity.

**Definition 17.** Consider an alphabet \(\Sigma\) and a binary operation \(\text{op} : \mathcal{P}(\Sigma^*) \times \Sigma^* \rightarrow \mathcal{P}(\Sigma^*)\). Let \(f\) be a function mapping sets of strings to reals, \(f : \mathcal{P}(\Sigma^*) \rightarrow \mathbb{R}\). We say that:

– \(f\) is monotone non-increasing w.r.t. \(\text{op}\) iff \(f(S) \geq f(S \text{ op } r)\),
– \(f\) is monotone non-decreasing w.r.t. \(\text{op}\) iff \(f(S) \leq f(S \text{ op } r)\),
for all sets of strings $S$ and all strings $r$ on $\Sigma$, with $r \notin S$.

We extend now the notions of probability, observed and expected frequencies from a single string to a set of strings of the same length.

Let us consider the set $S_h$ of all the strings in $\Sigma^*$ of length $h$, $S_h = \{ r_1 \ldots r_l \}$, where $l = |\Sigma|^h$, $r_j = s_{1,j} \ldots s_{h,j}$ and $1 \leq j \leq l$. The probabilities of the strings in $S_h$ sum up to 1. In fact $\sum_{j=1}^l \prod_{i=1}^h pr(s_{i,j})$ can be written as

$$\sum_{s_1 \in \Sigma} \ldots \sum_{s_h \in \Sigma} pr(s_1) pr(s_2) \ldots pr(s_h)$$

that is as

$$\sum_{s_1 \in \Sigma} \sum_{s_2 \in \Sigma} \ldots \sum_{s_h \in \Sigma} pr(s_1) pr(s_2) \ldots pr(s_h) = 1.$$

**Definition 18.** Let $T_k$ be a $k$-trie and let $S = \{ r_1 \ldots r_q \}$ be strings of length $h$ belonging to $T_k$. We define the probability and the observed and expected frequencies of $S$ as follows:

- $pr(S) = \sum_{i=1}^q pr(r_i)$,
- $c(S) = \sum_{i=1}^q c(r_i)$,
- $Exp(S) = \sum_{i=1}^q Exp(r_i)$.

From the above definition it follows that both the expected and the observed frequencies are compositional and monotone non-decreasing w.r.t. string union. In analogy to what we did before we can define also the abbreviation $D(S) = c(S) - Exp(S)$.

With the previous definitions we have $D(S) = \sum_{i=1}^q D(r_i)$.

Also the considered scores can be naturally extended to sets of strings:

- $z_1(S) = c(S) - Exp(S)$,
- $z_2(S) = \frac{c(S)}{Exp(S)}$,
- $z_3(S) = \frac{(c(S) - Exp(S))^2}{Exp(S)}$.

In order to analyze how the scores behave w.r.t. string union, let us consider, in a $k$-trie $T_k$, a set of strings $S$ having the same length $h$ and a further string $r$ of length $h$, associated to a node $n$, that is $r = str_{T_k}(n)$. Consider now the set given by the union $S' = S \cup r$, with $r \notin S$.

- For the first score we have
  $$z_1(S') = (c(S) + c(n)) - (Exp(S) + Exp(n)) = z_1(S) + z_1(n).$$

Hence the score $z_1$ is compositional w.r.t. string union.

Since $z_1$ can have both a positive or a negative value, $z_1(S)$ will increase or decrease by union with string $r$ depending on the score $z_1(r)$ which is added.
– For the second score we have 
\[
 z_2(S') = \frac{c(S) + (c(n) - (\text{Exp}(S) + \text{Exp}(n)))^2}{\text{Exp}(S) + \text{Exp}(n)}.
\]
In this case we can show that the score of the union \( S' \) is always greater or equal to the minimum between the score of the previous set \( S \) and the score of the added string \( r \) and it is always smaller or equal to the maximum of the two scores, that is \( \min\{z_2(S), z_2(r)\} \leq z_2(S') \leq \max\{z_2(S), z_2(r)\} \).
In fact, suppose \( z_2(r) \leq z_2(S) \).
We have to show that \( z_2(r) \leq z_2(S') \leq z_2(S) \).
Concerning the left inequality, we proceed as follows: since all quantities in
\[
\frac{c(n)}{\text{Exp}(n)} \leq \frac{c(S)}{\text{Exp}(S)}
\]
are positive, it follows that \( \frac{c(n)}{c(S)} \geq \frac{\text{Exp}(n)}{\text{Exp}(S)} \).
Hence, \( (1 + \frac{c(n)}{c(S)}) \geq (1 + \frac{\text{Exp}(n)}{\text{Exp}(S)}) \), that is \( \frac{c(n) + c(S)}{c(S)} \geq \frac{\text{Exp}(n) + \text{Exp}(S)}{\text{Exp}(n)} \) and finally
\[
\frac{c(n) + c(S)}{\text{Exp}(n) + \text{Exp}(S)} \geq \frac{c(S)}{\text{Exp}(n)}.
\]
For the right inequality we proceed in the same way: from \( \frac{c(n)}{\text{Exp}(n)} \leq \frac{c(S)}{\text{Exp}(S)} \)
it follows that \( \frac{c(n)}{c(S)} \leq \frac{\text{Exp}(n)}{\text{Exp}(S)} \). Hence, \( (1 + \frac{c(n)}{c(S)}) \leq (1 + \frac{\text{Exp}(n)}{\text{Exp}(S)}) \), that is \( \frac{c(n) + c(S)}{c(S)} \leq \frac{\text{Exp}(n) + \text{Exp}(S)}{\text{Exp}(n)} \) and finally
\[
\frac{c(n) + c(S)}{\text{Exp}(n) + \text{Exp}(S)} \leq \frac{c(S)}{\text{Exp}(n)}.
\]
Analogously, if \( z_2(S) \leq z_2(r) \), then we can prove that \( z_2(S) \leq z_2(S') \leq z_2(r) \).
Summarizing, the behaviour of the second score with respect to string union is the following:
\[
\begin{align*}
&\bullet \ z_2(S') > z_2(S) \text{ iff } z_2(S) < z_2(r); \\
&\bullet \ z_2(S') < z_2(S) \text{ iff } z_2(S) > z_2(r); \\
&\bullet \ z_2(S') = z_2(S) \text{ iff } z_2(S) = z_2(r).
\end{align*}
\]
– Let us now consider the third score,
\[
 z_3(S') = \frac{(c(n) + (c(n) - (\text{Exp}(S) + \text{Exp}(n)))^2}{\text{Exp}(S) + \text{Exp}(n)}.
\]
Let us compare \( z_3(S) \) to \( z_3(S') \), that is, by multiplying both expressions by \( \text{Exp}(S) + \text{Exp}(n) \), we compare
\( (c(S) - \text{Exp}(S))^2 (\text{Exp}(S) + \text{Exp}(n)) \) to
\( ((c(S) - \text{Exp}(S)) + (c(n) - \text{Exp}(n)))^2 \text{Exp}(S). \)
We simplify common subexpressions, then it is equivalent to compare
\( (c(S) - \text{Exp}(S))^2 \text{Exp}(n) \) to
\( (c(n) - \text{Exp}(n))^2 \text{Exp}(S) + 2(c(S) - \text{Exp}(S))(c(n) - \text{Exp}(n))\text{Exp}(S), \) that is, after dividing both expressions for \( \text{Exp}(n) \text{Exp}(S) \), we compare
\( z_3(S) \) to \( z_3(r) + \frac{2D_0D_s}{\text{Exp}(n)}. \)
Therefore, let \( \sigma(S, n) = \frac{2D_0D_s}{\text{Exp}(n)} \), we have:
\[
\begin{align*}
&\bullet \ z_3(S') \text{ is equal to } z_3(S) \text{ iff } z_3(S) = z_3(r) + \sigma(S, n); \\
&\bullet \ z_3(S') \text{ decreases w.r.t. } z_3(S) \text{ iff } z_3(S) > z_3(r) + \sigma(S, n); \\
&\bullet \ z_3(S') \text{ increases w.r.t. } z_3(S) \text{ iff } z_3(S) < z_3(r) + \sigma(S, n).
\end{align*}
\]
From the above analysis it follows that all the considered scores are not monotone w.r.t. string union. In fact, by adding a new string to a set of strings (or a pattern), the score of the resulting set (pattern) can either increase or decrease with respect to the one of the initial set (pattern). This has the important consequence that if we want to determine a set containing the most relevant substrings and we intend to compute it incrementally, we have to be extremely careful. In the next section we discuss how to do it in the case of \( z_3 \) score.
6 Determining the most relevant strings

In this section we show how to synthesize, from a set of interesting strings, an appropriate subset with its associated score, in order to point out the most relevant strings in the set. In doing this we have to be careful since, as shown in the previous section, the relevance scores we consider can decrease by string union.

Let us analyze in particular the chi-square score $z_3$. As discussed in Section 5.3, this score cannot distinguish over-represented strings in $T$ from under-represented ones. In order to get a better understanding of the behaviour of the score, let us consider the simple initial case in which $S = \{r_1\}$, namely the set contains just one single string $r_1$ associated to the node $n_1$, and we add to this set a further string $r_2$, associated to the node $n_2$. By applying the union operation, $S' = S \cup r_2$, we do not decrease the score, that is $z_3(S') \geq z_3(S)$, iff $z_3(S) \leq z_3(r_2) + \sigma(S, n_2)$, as shown before. We can analyze the $\sigma(S, n_2)$ component and distinguish three different cases:

1. $\sigma(S, n_2) = 0$. This can happen if either $D_{n_1} = 0$ or $D_{n_2} = 0$, that is if the expected frequency is equal to the observed frequency either for $r_1$ or for $r_2$;
2. $\sigma(S, n_2) > 0$. This happens if $D_{n_1}$ and $D_{n_2}$ have either both a positive value, that is $c(n_1) > \text{Exp}(n_1)$ and $c(n_2) > \text{Exp}(n_2)$, or both a negative value, namely the two strings are both over-represented or both under-represented in $T$;
3. $\sigma(S, n_2) < 0$. This happens if one of $D_{n_1}$ and $D_{n_2}$ has a positive value and the other has a negative value, that is one string is over-represented and the other is under-represented in $T$.

Note that, for determining the most relevant strings, we are not interested in the case $\sigma(S, n_2) = 0$, since it corresponds to strings which occur exactly as many times as expected.

When we compute the set of most relevant strings in an incremental way, $S'$ is the set of most relevant strings already accumulated and $r_2$ is the string we are trying to add at each step. Hence $z_3(S) \geq z_3(r_2)$ and the total score of $S \cup \{r_2\}$ does not decrease only if $\sigma(S, n_2) = \frac{2D_{n_1}D_{n_2}}{\text{Exp}(n_2)} > 0$, that is when we consider either over-represented or under-represented strings only. We deal with the two cases separately.

Let us first consider the case in which we are interested in over-represented strings in $T$. Consider the set $\text{Over}_h$ of all the over-represented strings of length $h$ in $T$, that is the strings, corresponding to nodes at level $h$ in $T_k$, which are over-represented. We want to determine the set of the most interesting over-represented strings of length $h$ in $T$, that is the subset of strings in $\text{Over}_h$ having the highest score.

As before, for simplicity we identify a string and its corresponding node in the tree. Let $n_i$ be any string in $\text{Over}_h$, then $c(n_i) > \text{Exp}(n_i)$ and $D_{n_i} > 0$. 

Let us assume that the strings in $\text{Over}_h = \{r_1, r_2, \ldots, r_q\}$ are ordered in non-increasing order by score, namely $z_3(r_i) \geq z_3(r_j)$, when $i < j$.

Initially $S = \{r_1\}$, where $r_1$ is a string with the highest score among the over-represented strings of length $h$.

We add to $S$ another string, $r_i$, in $\text{Over}_h$ only if the union of $S$ and $r_i$ does not decrease the total score, that is only if $z_3(S) \leq z_3(r_i) + \sigma(S, n_i)$.

We recall that $z_3(S) \geq z_3(r_i)$, because the strings are ordered by score, and that $\sigma(S, n_i) > 0$, since all the strings are over-represented.

Then the set of strings in $\text{Over}_h$ with highest score can be built in an incremental way by following the order on strings; at each step, in order to include a further string $r_i$ in the set, the construction has to verify the condition $z_3(S) < z_3(r_i) + \sigma(S, n_i)$.

Note that condition $z_3(S) \leq z_3(r_i) + \sigma(S, n_i)$ depends on the partial set $S$ already built, that is on its score $z_3(S)$, and on $\sigma(S, n_i) = \frac{2D_S D_n}{\text{Exp}(n_i)}$, which also depends on $S$. Two important consequences come from this fact:

a. It is possible that a string $r_i$ decreases the score of $S$ by union, while a string $r_j$, with $i < j$, does not. In fact we could have

$$- \frac{D^2_{r_i}}{\text{Exp}(n_i)} \geq - \frac{D^2_{r_j}}{\text{Exp}(n_j)},$$

that is $z_3(r_i) \geq z_3(r_j)$;

b. It is possible that a string $r_i$ decreases the score of $S$ by union, although the same string does not decrease the score of a larger set $S' \supset S$. In fact we can have

$$- z_3(S) < z_3(S');$$

$$z_3(S') \leq z_3(r_i) + \frac{2D_S D_n}{\text{Exp}(n_i)},$$

and

$$z_3(S') \leq z_3(r_i) + \frac{2D_S D_n}{\text{Exp}(n_i)};$$

when $D_{S'}$ is much greater than $D_S$.

Because of the above facts, we may have to consider each string we want to add to the set more than one time. Hence in order to build incrementally the set $\text{MOS}(T, h)$ of Most Over-represented Strings of length $h$ in the text $T$ we have to follow the steps:

1. scan the strings in $\text{Over}_h$ not yet added to $\text{MOS}(T, h)$ in non-increasing score order;
2. add them to $\text{MOS}(T, h)$ if they do not decrease its score;
3. repeat (1) and (2) until $\text{MOS}(T, h)$ is stable
   (i.e., until no more strings can be added without decreasing $z_3(\text{MOS}(T, h))$).
We indicate the set $MOS(T, h)$ simply with $MOS$ whenever $T$ and $h$ are clear from the context. The above reasoning can be turned into the abstract procedure $BUILD\_MOS$.

$BUILD\_MOS(Over_h)$:

\{
Over_h = \{r_1, r_2, \ldots, r_q\} is ordered into non-increasing order w.r.t. the score $z_3$\}

$MOS := \{r_1\};$ Mark($r_1$);

\textbf{repeat}

for $r_i \in Over_h$ \textbf{and not} Marked($r_i$) \textbf{do}

\textbf{if} $z_3(MOS) \leq z_3(r_i) + \sigma(MOS, n_i)$ \textbf{then}

\begin{verbatim}
MOS := MOS \cup \{r_i\};
Mark(r_i);
\end{verbatim}

\textbf{end};

\textbf{until} $MOS$ is stable;

\textbf{end};

The result of this construction strongly depends on the initial ordering of $Over_h$ and on the associated initialization of $MOS$. We could choose a different ordering on $Over_h$, for example the one on $D_i$, namely $D(r_i) \geq D(r_j)$, when $i < j$, and we would obtain a different subset of $Over_h$. In this case the initialization of $MOS$ would be with a string most over-represented in $Over_h$.

Note that the set $Over_h$ could be very large, since all the strings of length $h$ could be $|\Sigma|^h$. Therefore considering all its strings for union and iterating the procedure until stability could be very expensive. A reasonable heuristic would be to stop the construction of the subset as soon as the cardinality of $MOS$ reaches a given number.

In the simple case in which we assume that all the symbols in the alphabet $\Sigma$ have the same probability, $1/|\Sigma|$, the computation of $MOS$ becomes much simpler and faster. In fact, in that case we have that $Exp(n_i) = \frac{(m-h+1)}{|\Sigma|^h}$, for all $i$, that is for all the strings in $Over_h$. Then, given two strings $r_i$ and $r_j$ in $Over_h$, $z_3(r_i) > z_3(r_j)$ implies

\begin{align*}
& - D_{n_i} > D_{n_j} \text{ and then also } \\
& - \frac{2D_{n_i} D_{n_j}}{Exp(n_i)} > \frac{2D_{n_i} D_{n_j}}{Exp(n_j)}.
\end{align*}

As a consequence, the case (a) previously described becomes impossible with this assumption. In fact $z_3(r_i) > z_3(r_j)$ implies $z_3(r_i) + \frac{2D_{n_i} D_{n_j}}{Exp(n_i)} > z_3(r_j) + \frac{2D_{n_i} D_{n_j}}{Exp(n_j)}$.

Hence in procedure $BUILD\_MOS$ if $r_i$ does not satisfy condition $z_3(MOS) < z_3(r_i) + \sigma(MOS, n_i)$, ensuring that the score of $MOS$ does not decrease by union, no other string $r_j$, with $j > i$, can satisfy it. Then we \textit{can stop considering the strings in $Over_h$ as soon as we find one string which decreases the score of MOS by union.}

Also the previous case (b) is no more a problem: at most one single scan of $Over_h$ guarantees that $MOS$ is the subset of strings of maximal score; each string
$r_i$ in $\text{Over}_h$ can be considered only once for union, either $r_i$ satisfies condition $z_3(MOS) \leq z_3(r_i) + \sigma(MOS, n_i)$, or it doesn’t. Hence, in the simple assumption of equiprobable symbols, the abstract procedure for computing $MOS$ becomes the following:

\begin{verbatim}
BUILD_SIMPLE_MOS(Over_h):
{Over_h = \{r_1, r_2, \ldots, r_q\} is ordered into non-increasing order w.r.t. the score z_3}\nMOS := \{r_1\};
while $r_i \in \text{Over}_h$ and $z_3(MOS) \leq z_3(r_i) + \sigma(MOS, n_i)$ do
   MOS := MOS $\cup \{r_i\}$;
end;
\end{verbatim}

For under-represented strings, the corresponding constructions are similar to the ones just described. If $n$ is an under-represented string, then $c(n) < \text{Exp}(n)$, that is $D_n < 0$. Let $\text{Under}_h = \{s_1, s_2, \ldots, s_q\}$ be the set of under-represented strings of length $h$ in $T$ (i.e. corresponding to nodes at level $h$ in $T_k$), ordered in non-increasing order by score. In order to obtain a subset of $\text{Under}_h$ containing the most interesting strings, that is the set $\text{MUS}(T, h)$ of Most Under-represented Strings of length $h$ in $T$, we apply the following steps:

1. scan the strings in $\text{Under}_h$ not yet added to $\text{MUS}(T, h)$ in non-increasing score order,
2. add them to $\text{MUS}(T, h)$ if they do not decrease its score;
3. repeat (1) and (2) until $\text{MUS}(T, h)$ is stable
   (i.e., until no more strings can be added without decreasing $z_3(\text{MUS}(T, h))$).

Let us conclude by observing that the set $MOS(T, h)$ ($\text{MUS}(T, h)$) is a set of annotated strings, namely each string has an observed frequency, an expected frequency and a chi-square score associated to it. This will be used also when abstracting them.

7 Determining the most relevant patterns

In this section we would like to combine the benefits of strings abstraction and scoring. We have already defined how to assign a relevance score to abstract substrings, since they are just sets of strings. Once the most interesting strings have been determined and stored into $MOS(T, h)$, we want to perform an abstraction step, in order to summarize such strings in terms of a sum of patterns, in the sense of Definition 5.

7.1 Determining a maximal abstraction of a set of strings

For pattern discovery it would be useful to have a concise representation of $MOS(T, h)$. The most natural choice is to define an abstraction of $MOS(T, h)$ as an equivalent sum of patterns which, as we already pointed out in Section 4.1, is just a particular kind of regular expression.
For simplicity’s sake in the following we omit the information associated to each string (i.e. observed frequency, expected frequency and chi-square score).

Let us consider the following simple example.

Example 19. Let $\Sigma$ be $\{A, T, C, G\}$ and let $MOS(T, 4)$ be the set:

$$\{ACGA, ACTA, TCTA, TATA, CATA, CAAT, CGAT, GGAT, GGCT, CAGA\}.$$

An abstraction of $MOS(T, 4)$ can be given in terms of the equivalent sum of 6 patterns, namely


This abstraction is not the best one. We can in fact further abstract $MOS(T, 4)$ as the sum of 5 patterns:


This second abstraction is clearly preferable since it is more concise.

The abstraction of $MOS$ is a sum of disjoint patterns, namely the patterns in the sum must be a partition of $MOS$.

First of all let us note that both a sum of strings and a sum of patterns on an alphabet $\Sigma$ can be viewed as a sum of strings on an extended alphabet $\Sigma'$ isomorphic to $\mathcal{P}(\Sigma) - \emptyset$. Namely, in each position in a pattern (string) there is a symbol in $\Sigma'$ corresponding to a set of symbols of $\Sigma$. Hence both the initial set of strings, $MOS(T, h)$, and its abstraction can be viewed as sets (sum) of strings on such an extended alphabet and we can deal with strings and patterns in the same way. For simplicity, the union of $s_1, s_2 \in \Sigma'$ will be denoted either with $[s_1s_2]$ or with the corresponding symbol in $\Sigma'$.

Let us define now the simplest case in which abstraction is possible.

**Definition 20.** Let $P_1, P_2, \ldots, P_j$, with $j > 1$, be disjoint patterns of length $h$ on an alphabet $\Sigma$. $P_1, P_2, \ldots, P_j$ have a simple abstraction iff they are identical except for a fixed position:

$$P_1 = \alpha s_1 \beta, P_2 = \alpha s_2 \beta, \ldots, P_j = \alpha s_j \beta$$

where $\alpha, \beta \in \Sigma'^* \text{ and } s_1, \ldots, s_j \in \Sigma'$.

Note that the previous definition only depends on the concepts of equality and inequality among symbols in $\Sigma'$ (i.e. sets of symbols in $\Sigma$).

**Proposition 21.** Let $P_1 = \alpha s_1 \beta, P_2 = \alpha s_2 \beta, \ldots, P_j = \alpha s_j \beta$, where $\alpha, \beta \in \Sigma'^* \text{ and } s_1, \ldots, s_j \in \Sigma'$, be a set of disjoint patterns which have a simple abstraction. Their sum is equivalent to a single pattern whose symbols are the union of the corresponding symbols in the patterns:

$$P = \alpha[s_1s_2 \ldots s_j] \beta \text{ and } P \equiv P_1 + P_2 + \ldots + P_j.$$
This is due to distributivity of concatenation w.r.t. sum in regular expressions. In the following we will call simple abstraction also the operation of transforming the disjoint patterns $P_1, P_2, \ldots, P_j$, which have a simple abstraction, into the equivalent pattern $P$ as stated in Proposition 21.

**Example 22.** Let us consider $MOS(T, 4)$ in the previous Example 19. The strings $ACGA$ and $ACTA$ have a simple abstraction, since they differ only in the third position. Their sum is equivalent to $AC[GT]A$.

The disjoint patterns $AC[GT]A$, $AC[GT]C$ and $AC[GT]T$ have a simple abstraction, since they differ only in the last position. They are equivalent to $AC[GT][ACT]$.

**Definition 23.** Let $S = \{s_1, \ldots, s_n\}$ be a set of strings of length $h$ on an alphabet $\Sigma$. An abstraction $A$ of $S$ is a sum of disjoint patterns of length $h$ on $\Sigma$ which is equivalent to $S$, that is

$$A = P_1 + \ldots + P_m, \quad P_i \cap P_j = \emptyset \text{ for all } i, j \in [1, m], \quad m \leq n \quad \text{and} \quad A \equiv S.$$  

The size of the abstraction $A$ is $m$, the number of patterns it consists of. $A$ is a maximal abstraction of $S$ iff no proper subset of $A$ has a simple abstraction.

Note that the size of a maximal abstraction of $S$ is always smaller or equal to $|S|$. It is equal to one for example when $S$ contains a single string, in this case the maximal abstraction of $S$ is the string itself, and also when $S$ contains all the strings of length $h$ on $\Sigma$, in this case the maximal abstraction of $S$ is the pattern of length $h$ with all the symbols in $\Sigma$ in each position.

In general the maximal abstraction of a set of strings of length $h$ on an alphabet $\Sigma$ is not unique. Moreover there can be maximal abstractions with different sizes for the same set $S$.

**Example 24.** Let $MOS(T, 3)$ be:

$$\{ACG, ACT, TCG, TCA\},$$

we can abstract it as $AC[GT] + TC[GA]$ but also as $[AT]CG + ACT + TCA$.

Both such abstractions of $MOS(T, 3)$ are maximal, the first one has size 2 and the second has size 3. The first one is a maximal abstraction of $MOS(T, 3)$ of minimal size.

One way to build an abstraction of $MOS(T, h)$ consists in building an abstract tree representing a set of disjoint patterns equivalent to the strings in $MOS(T, h)$. To reach this goal, we first transform the set $MOS(T, h)$ into the Most Over-represented Tree, $MOT(T, h)$. In such a tree each path represents a string in
Then we transform $MOT(T, h)$ into another tree, the *Abstract Most Over-represented Tree*, $AMOT(T, h)$, where each path represents a pattern.

To better clarify this technique, let us consider the following example.

**Example 25.** Let $T$ be a text on the alphabet $\Sigma = \{A, T, C, G\}$. Suppose that the interesting strings of length 2 in $T$ have been already collected into $MOS(T, 2)$:

$$\{AA, AC, CC, CG, GC, GT, TA, TC\}.$$ 

This corresponds to the following $MOT(T, 2)$ tree:

We can abstract the strings into disjoint patterns by transforming $MOT(T, 2)$ into the following abstract tree, $AMOT(T, 2)$:

In the abstract tree the paths from the root to the leaves represent patterns and they can be composed with the “+” operation.

This transformation on trees corresponds to transforming the set of strings $MOS(T, 2)$ into the following abstraction of $MOS(T, 2)$:


This regular expression is a maximal abstraction of $MOS(T, 2)$. 
Given $MOS(T, h)$, how can we obtain $MOT(T, h)$ and then $AMOT(T, h)$?

$MOT(T, h)$ can be built by invoking a procedure $\text{BUILD\_MOT}$ which builds a tree similar to an h-trie but containing only the strings in $MOS(T, h)$. The information on each string is associated to the leaf of the corresponding path.

The structure and the construction of the tree is similar to the one given in Section 2 for the k-trie. The node structure can be described by the following type.

```plaintext
type node = record
  symbol : string;
  counter : int;
  exp : real;
  score : real;
  sons : pointer(node);  \{ pointer to the sons \}
  level : pointer(node);  \{ pointer to the nodes of the same level \}
end;
```

The fields `counter`, `exp` and `score` are used only for the leaves of the tree. The field `symbol` is intended to store the set of symbols in $\Sigma$ associated to each node, that is a symbol in $\Sigma'$. An efficient way to implement this, could be to represent it as a memory word where each bit corresponds to a symbol in $\Sigma$.

In order to traverse the tree by levels we need an array of $h$ pointers to the levels of the tree:

```plaintext
lev = array [0, h] of pointer(node);
```

such pointers give access to the list of nodes at each level. All such pointers are initialized to `nil`, up to $lev[0]$ which points to the root. The algorithm $\text{BUILD\_MOT}$ makes use of $lev[1]$ to access the first level of the tree (note that $lev[0]$ does not contain any symbol) and of the procedure $\text{INSERT}(s, j, p1, p2)$. Such procedure looks for a node with the `symbol` field equal to $s$ in the list pointed by $p1$ at level $j$ in the tree ($p1$ points to a list of siblings). If $s$ is not in any node of the list, it adds a new node to the beginning of the list pointed by $p1$ and by $lev[j]$. If $p1 = nil$ ($lev[j] = nil$) it initializes the list by setting $p1$ ($lev[j]$) to point to the newly created node. If a new node is created, then it initializes its fields by setting `symbol` to $s$, `sons` to `nil` and `level` to $lev[j]$, that is the level of the tree to which the node is added. In either case it returns $p2$, a pointer to the node found or created for the symbol $s$.

$\text{BUILD\_MOT}(MOS(T, h, h));$

for all $(\text{string, counter, exp, score}) \in MOS(T, h)$ do

begin
  $p1 := lev[1]$; \{ beginning of a new path \}
  for $j := 1$ to $h$ do

    begin
      $\text{Insert}(\text{string}[j], j, p1, p2);$
    end;

end;
the symbol in position j in string is either found
or inserted in a node at level j in the list pointed by p1;
p2 points to the node thus found or inserted
p1 := p2.sons;
end;
Set_info(counter, exp, score, p2);
{information associated to string are stored into the leaf pointed by p2}
end;

Build_MOT is linear in the size of the set of strings (patterns) in input,
that is $O(hn)$, where $n = |MOS(T, h)|$.

The construction of $AMOT(T, h)$ from $MOT(T, h)$ is a little more tricky. In
Example 25 the abstract tree $AMOT(T, 2)$ is obtained as follows: nodes A and
T at level one of $MOT(T, 2)$ are composed because they satisfy two conditions:

– they have a common prefix, $\alpha$, that is they share the same father node in the
tree;
– their suffixes are the same, $\beta$, that is their subtrees are isomorphic except
for the root labels.

The same reasoning applies to the leaves, at level two: clearly all leaves sharing
the same father do have isomorphic subtrees, except for the root labels.

More generally, we can define a stepwise procedure $Abstract_tree$ which
takes $MOS(T, h)$ in input and produces $AMOT(T, h)$ in output.

Abstract_tree($MOS(T, h)$, h);
Build_MOT($MOS(T, h)$, h);
for $j := 1$ to h do
  for all $n \in lev[j - 1]$ do
    n.sons := Partition_and_fold(n.sons);
end;

The procedure $Partition_and_fold$ is used to transform the tree $MOT(T, h)$
built by the procedure $Build_MOT$. It considers the sons of node $n$ and their
corresponding subtrees, and partitions them into equivalence classes based on
tree isomorphism, but ignoring the root symbols. It folds the subtrees by keeping
only one representative for each class in the partition. The root symbol of such a
representative is the union of all the root symbols in the class and the information
associated to each leaf represents the information of all the corresponding leaves
in the class. For example, the observed frequency of leaves is the sum of all
observed frequencies of the corresponding leaves in the class. This sum may be
a useful indicator of their relevance.

Partition_and_fold($N$);
{ $N$ is a list of sibling nodes }
\[
\text{class} := \emptyset;
\]
\{ \text{class is a list of (roots of) subtrees, each one representing an equivalence class w.r.t. tree isomorphism} \}

for all \( n \in \mathbb{N} \)
begin
  \( \text{found} := \text{false}; \)
  while \( c \in \text{class} \) and not \( \text{found} \)
do
    if \( \text{isIsomorphic}(n, c) \) then
    begin
      \( \text{found} := \text{true}; \)
      \( c := \text{Fold}(n, c); \)
      \{ the subtree \( n \) is composed with the subtree \( c \} \)
    end
  end
  if not \( \text{found} \) then
  Add(\text{class}, n);
  \{ a new equivalence class for \( n \) is inserted into \text{class} \}
  return \text{class};
end;

The procedure \textsc{PartitionAndFold} is used to create a new abstract tree by incrementally folding all the subtrees in each equivalence class into a single one. The \textsc{Fold} procedure receives in input two isomorphic subtrees, \( n \) and \( c \), and composes them into a new isomorphic subtree where:

- node symbols are the union of the corresponding node symbols in \( n \) and \( c \), and
- the observed frequencies of the leaves are the arithmetic sum of the observed frequencies of the corresponding leaves in \( n \) and \( c \).

The time complexity of the procedure \textsc{AbstractTree} is exponential in the worst case. If all the strings of length \( h \) on \( \Sigma \) are in \( \text{MOS}(T, h) \), that is \(|\text{MOS}(T, h)| = |\Sigma|^h\), at each level all the subtrees are isomorphic and \textsc{PartitionAndFold} produces only trivial partitions consisting of one single class. In this case the size of the tree is quickly reduced. Hence the worst case is when not all the strings of length \( h \) are present, i.e. not all the subtrees are isomorphic, but the size of the tree is very large, that is \(|\text{MOS}(T, h)| \) is almost \(|\Sigma|^h\), then \textsc{AbstractTree} has a complexity that is less than \( \mathcal{O}(h|\Sigma|^{h+1}) \). In fact, in this case the time complexity is less than \( \sum_{j \in [1,h]} (|\Sigma|^{j-1}c|\Sigma|^2|\Sigma|^{h-j}) = c\sum_{j \in [1,h]} |\Sigma|^{h+1} \), where

- the external cycle on the tree levels gives the sum \( \sum_{j \in [1,h]} \);
- the internal cycle explores all nodes of level \( j - 1 \), which are at most \(|\Sigma|^{j-1};
- c|\Sigma|^2|\Sigma|^{h-j} \) is the worst time needed by procedure \textsc{PartitionAndFold} for building the \(|\Sigma|^{j-1} \) partitions with at most \(|\Sigma| \) classes.
In fact:
the external cycle of Partition_and_fold considers at most $|\Sigma|$ subtrees and the internal one at most $|\Sigma|$ classes, and
$c|\Sigma|^{h-j}$ is the worst time required by procedure is_isomorphic for visiting a subtree of depth $h-j$ and checking for tree isomorphism. The same complexity applies to procedure Fold for composing two isomorphic subtrees of depth $h-j$.

Clearly the worst case does not happen in practice since $MOS(T, h)$ is much smaller than $|\Sigma|^h$.

A simple optimization of the procedure Abstract_tree can be given by considering first the leaves and then all the other nodes. By scanning all the nodes at level $h-1$ and directly grouping their sons, which are leaves, we reduce the size of the subtrees to be considered in the procedure Partition_and_fold.

With procedure Abstract_tree we obtain $AMOT(T, h)$ in a deterministic way. It represents a sum of disjoint patterns which in general is not a maximal abstraction of $MOS(T, h)$. For each level $j$, the procedure Partition_and_fold performs a simple abstraction by grouping together regular expressions in the form $\alpha s_j \beta$, that is with the same prefix $\alpha$ and suffix $\beta$ and with a different symbol in position $j$, $s_j$, and by substituting them with a new equivalent pattern. But $\alpha$ and $\beta$ are captured through the structure of the tree $MOT(T, h)$ in different ways, that is $\alpha$ is the string (pattern) labelling the path from the root to the father node of $s_j$, while the suffix $\beta$ is captured through tree isomorphism and then in general it denotes a sum of strings (patterns). This asymmetry determines the choice of the particular abstraction of $MOS(T, h)$ built by Abstract_tree.

For simplicity we identify each path with the concatenation of symbols of its nodes. We can state the following lemma.

**Lemma 26.** The procedure Abstract_tree can build an abstract tree containing two paths $P_1$ and $P_2$, such that $P_1 = \alpha \gamma_1 \beta$ and $P_2 = \alpha \gamma_2 \beta$, with $\alpha, \beta \in \Sigma^*$ and $\gamma_1, \gamma_2 \in \Sigma'$. In such a case, the abstract tree contains also at least another path $P_3$, where either $P_3 = \alpha \gamma_1 \delta$ or $P_3 = \alpha \gamma_2 \delta$, with $\delta \in \Sigma^*$ and $\delta \neq \beta$.

**Proof.** In the abstract tree, having the two paths $P_1$ and $P_2$ with a common prefix $\alpha$, corresponds to have a unique path from the root in the tree to a node $n$ at level $|\alpha|$. From $n$, two branches reach two nodes, $n_1$ and $n_2$, with symbol $\gamma_1$ and $\gamma_2$ respectively. The two nodes are origin of a path $\beta \in \Sigma^*$. At each level of the initial tree, procedure Abstract_tree compacts the tree by composing the isomorphic (except for the root symbol) subtrees whose roots are sons of the same node.

This means that procedure Abstract_tree in the first $|\alpha|$ iterations have composed (even trivially) a set of initial paths into a single path $\alpha$. At iteration $|\alpha| + 1$ the procedure could not compose the subtrees with roots $\gamma_1$ and $\gamma_2$. This implies that they were not isomorphic (even if we ignore the roots symbols).

On the other hand at iteration $|\alpha| + 2$ some subtrees, whose root is son of $n_1$, have been composed into a path $\beta$ and some subtrees, whose root is son of $n_2$, have been composed into another path $\beta$. 
Hence in $MOT(T, h)$ there exists at least one path with root in $n_1$ which does not correspond to any path with root in $n_2$ (or vice versa there exists at least one path with root in $n_2$ which does not correspond to any path with root in $n_1$), and then in the final abstract tree there must be at least another path $P_3 = \alpha \gamma_1 \delta$ (or $P_3 = \alpha \gamma_2 \delta$), with $\delta \neq \beta$.

The previous lemma shows that the procedure Abstract_tree can build an abstract tree which contains two paths such as $P_1 = \alpha \gamma_1 \beta$ and $P_2 = \alpha \gamma_2 \beta$, that is two disjoint patterns which have a simple abstraction, hence in general it does not produce a maximal abstraction.

One possibility for improving the abstraction is to require a weaker property than tree isomorphism for composing subtrees. We could require that two subtrees, with roots which are siblings, have a common path (still ignoring the root). Such common paths could be composed together. With this alternative technique, the structure of the resulting tree would be more complex.

Another possibility is to improve the abstraction incrementally by considering different trees derivable from the set of patterns in $AMOT(T, h)$. These can be obtained by first transforming the set of patterns.

Let us consider a permutation on strings. We recall that a permutation on a string $x$ of length $h$ can be represented by a bijective map $\tau : [1, h] \to [1, h]$ which specifies the reordering of the symbol positions in $x$. With $x^\tau$ we denote the string $y$ which results by reordering the symbols in $x$ as specified in $\tau$ and such that $y^\tau^{-1} = x$.

**Example 27.** Let $x = ACGT$ and $\tau$ be represented by the table

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

where the first row represents the domain and the second the range of $\tau$.

Then $ACGT^\tau = CTGA$ and $\tau^{-1}$ is represented by the table obtained by exchanging the rows in $\tau$ and by reordering the columns:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

hence $CTGA^\tau^{-1} = ACGT$.

**Definition 28.** Let $x = s_1 \ldots s_h$ be a string of length $h$ on an alphabet $\Sigma$. A rotation to the left of $x$ is a permutation $\rho$ such that $\rho(1) = h$ and $\rho(i) = i - 1$, for $i \in [2, h]$. A rotation to the right of $x$ is a permutation $\sigma$ such that $\sigma(i) = (i \mod h) + 1$, for $i \in [1, h]$.

We may apply compositions of permutations to strings. Note that a rotation to the left is equal to a composition of $h - 1$ rotations to the right and vice versa. In
particular we may apply compositions of rotations. By applying the composition of $h$ rotations to the left we obtain a complete rotation, equivalent to the identity permutation.

**Proposition 29.** The composition of simple abstraction of strings and permutation of strings is commutative, namely the result does not depend on the order of application.

This is due to a fact that we already noticed, i.e. simple abstraction depends only on equality and inequality of the corresponding symbols in the strings. But equality and inequality are preserved by permutation of strings.

As a consequence we may either apply a permutation to strings in $\text{MOS}(T, h)$ and then simple abstraction, or apply simple abstraction to $\text{MOS}(T, h)$ and then apply the same permutation to the resulting patterns; in both cases we obtain the same sum of patterns.

This observation leads us to the following procedure \textsc{Maximal Abstraction} for computing incrementally $\text{MAMOS}(T, h)$, a maximal abstraction of $\text{MOS}(T, h)$.

\textsc{Maximal Abstraction}(MOS($T$, $h$), $h$);
$I := \text{MOS}(T, h)$;
repeat
  for $j := 1$ to $h$ do \{it performs a complete rotation of the patterns\}
    begin
      Abstract\_tree($I$, $h$);
      \{it builds first the tree corresponding to $I$ and then its abstract tree\}
      $I := \{\text{patterns in the abstract tree rotated to the left}\}$;
    end;
until stable;
end;

This procedure is deterministic and it is able to compensate for the asymmetry determined by the tree structure of \textsc{AMOT}(T, h). In fact, because of string rotation, each position in $[1, h]$ is considered as a root for the tree.

**Example 30.** Let us consider the strings in Example 19. First iteration ($j=1$): the corresponding $\text{MOT}(T, 4)$ is the following.
We have indicated with a dotted line where composition of subtrees is possible.
After applying Abstract\textunderscore tree we obtain $AMOT_1(T,4)$, that is the following tree corresponding to the sum of 6 disjoint patterns.

We apply a rotation to the left to the patterns in $AMOT_1(T,4)$, thus obtaining the following tree.
Second iteration ($j=2$): no subtree can be composed, the application of Abstract_tree does not modify the tree, hence $AMOT_2(T, 4)$ is equal to the previous tree.

We apply a rotation to the left to the patterns in $AMOT_2(T, 4)$, thus obtaining the following tree.

Third iteration ($j=3$): after applying Abstract_tree we obtain $AMOT_3(T, 4)$, that is the following tree corresponding to the sum of 5 disjoint patterns.
We apply a rotation to the left to the patterns in $AMOT_3(T, 4)$, thus obtaining the following tree.

Fourth iteration ($j=4$): the application of Abstract_tree does not modify the tree, hence $AMOT_4(T, 4)$ is equal to the previous tree.

One further rotation to the left produces the final tree. In fact, in this example, by further rotations no new abstractions are possible.
The final tree corresponds to \( MAMOS(T, h) \), a maximal abstraction of \( MOS(T, 4) \), which is the following sum of 5 disjoint patterns:

\[
\]

**Proposition 31.** The procedure \textsc{Maximal\_abstraction} computes a sum of disjoint patterns, \( MAMOS(T, h) \), which is a maximal abstraction of \( MOS(T, h) \).

**Proof.** The internal cycle of \textsc{Maximal\_abstraction} performs a complete rotation of the patterns and, at each iteration, it produces an abstract tree corresponding to a sum of disjoint patterns.

If there are at least two disjoint patterns which have a simple abstraction, then they have the form \( P_1 = \alpha \gamma_1 \beta \) and \( P_2 = \alpha \gamma_2 \beta \), with \( \alpha, \beta \in \Sigma'^* \) and \( \gamma_1, \gamma_2 \in \Sigma' \). After at most \( |\alpha| + 1 \) further rotations to the left, the two patterns are abstracted, because the prefixes \( \beta \alpha \) correspond to a unique path and then the leaves are abstracted. Hence the size of the abstraction computed by the procedure decreases after a further iteration of the external cycle.

On the other hand the size of an abstraction of \( MOS(T, h) \) is finite and at least equal to one. Hence the size of the abstraction cannot continue to decrease, at a certain point there will be no more two patterns which have a simple abstraction. That is stability will be reached after a finite number of iterations.

The time complexity of \textsc{Maximal\_abstraction} is heavy. In fact for a given set \( MOS(T, h) \) we repeat at least \( h \) times the procedure \textsc{Abstract\_tree} and possibly iterate it until stability.

As shown in the proof of Proposition 31, we could introduce a testing condition after each rotation in order to guarantee stability and avoid iterating further. The condition for ensuring stability requires that all the abstract trees obtained in the \( h \) iterations of the internal cycle do not contain any pair of patterns which have a simple abstraction.

\( MAMOS(T, h) \), the maximal abstraction which is computed in this way, is not necessarily the one of minimal size, this depends on the symbol ordering.
in the strings of \( \text{MOS}(T, h) \). If we would like to \textit{compute all the maximal abstractions of MOS}(T, h), we should consider any permutation (except rotations or compositions of rotations) of our set of strings, as a starting point of the procedure MAXIMAL\_ABTRACTION.

\textit{Example 32.} Let us consider the following trivial set of strings, \( \text{MOS}(T, 3) \):

\[
\{\text{GCA, GCT, ACT, TCA}\}
\]

If we apply the procedure MAXIMAL\_ABTRACTION to it, we get at the first iteration (\( j=1 \)) the following tree \( \text{MOT}(T, 3) \):

\[
\begin{array}{c}
A \\
C \\
T \\
\end{array}
\begin{array}{c}
G \\
C \\
T \\
\end{array}
\begin{array}{c}
A \\
\end{array}
\]

which is abstracted into the following tree \( \text{AMOT}_1(T, 3) \).

\[
\begin{array}{c}
A \\
C \\
T \\
\end{array}
\begin{array}{c}
G \\
C \\
\text{[AT]} \\
\end{array}
\begin{array}{c}
T \\
C \\
A \\
\end{array}
\]

By iterating further we will not produce further abstraction, in fact the sum of three disjoint patterns corresponding to \( \text{AMOT}_1(T, 3) \):

\[
\text{GC}[\text{AT}] + \text{ACT} + \text{TCA}
\]

is already a maximal abstraction of \( \text{MOS}(T, 3) \).

Such a maximal abstraction is not unique. In order to find a different maximal abstraction of \( \text{MOS}(T, h) \), let us apply to the strings the permutation \( \tau \) (which is neither a rotation nor a composition of rotations) corresponding to the table

\[
\begin{array}{cccc}
1 & 2 & 3 \\
3 & 2 & 1 \\
\end{array}
\]
We get the following set of strings:

\{ACG, TCG, TCA, ACT\}

which is abstracted by the procedure \textsc{Maximal Abstraction} as the sum of disjoint patterns:

\[ AC[GT] + TC[GA] \]

Let us apply now the inverse permutation of \(\tau\), which is \(\tau\) itself, to the patterns, we get the sum of two disjoint pattern:

\[ [GT]CA + [GA]CT \]

which is also a maximal abstraction of the original set of strings \(MOS(T, 3)\). This is a maximal abstraction of minimal size.

In case we want to compute all the maximal abstractions of \(MOS(T, h)\) we should consider as a starting point any permutation of the strings, a part from rotations or compositions of rotations. Since the permutations of \(h\) symbols are \(h!\) and distinct compositions of rotations are \(h\) (including the identity), we should multiply the time complexity of \textsc{Maximal Abstraction} by \(h! - (h - 1)\).

From the previous discussion it is clear that the technique we propose for computing a maximal abstraction of a set of strings is rather costly and that \textit{computing all the maximal abstractions of a set of strings is unfeasible for large sets or large \(h\).} In order to attack this problem we intend to study different techniques and appropriate heuristics.

### 7.2 Abstracting the alphabet

A further simple possibility for combining abstraction and scoring is to abstract the alphabet \(\Sigma\). As discussed in Section 4.2, we can define an equivalence among symbols in \(\Sigma\) and translate the strings in \(MOS(T, h)\) into the resulting abstract alphabet. The translated \(MOS(T, h)\) is generally a multi-set of patterns and then we have to compact duplications, for example by summing up their observed frequencies. Clearly the sum of the observed frequencies, that is their occurrences, reflects somehow their relevance. In alternative we could evaluate the score associated to the set of the strings grouped into each pattern by the translation.

\textbf{Example 33.} Let \(\Sigma = \{A, T, C, G\}\) and let \(MOS(T, 4)\) be the following simple set of strings with their associated observed frequencies:

\{ACGA, 6; ACTA, 4; TCTA, 5; TATA, 10; CATA, 7; CAAT, 5; GGCT, 3; CAGA, 6\}.

Let us consider the abstract alphabet obtained by considering the symbol \(A\) equivalent to \(T\) and the symbol \(C\) equivalent to \(G\): \([AT], [CG]\).

Then \(MOS(T, 4)\) is translated into the multi-set of patterns:
which, after compacting duplications and summing up their observed frequencies, corresponds to the final set of disjoint patterns:

\[
\{ [AT][CG][CG][AT], 6; [AT][CG][AT][AT], 4; [AT][CG][AT][AT], 5; [AT][CG][AT][AT], 10; [CG][CG][CG][AT], 7; [CG][AT][AT][AT], 9; [AT][AT][AT][AT], 10; [CG][AT][CG][AT], 3; [CG][AT][CG][AT], 6 \}
\]

This approach to abstracting a set of strings has advantages and disadvantages. On one hand it is simple and efficient, in fact strings translation takes a time linear w.r.t. their size and compacting them requires also a linear time if we keep the patterns ordered. Hence the time complexity is equal to the complexity of a sorting algorithm. On the other hand, by abstracting the alphabet we superimpose a rigid interpretation to the set of interesting strings. This is not what we want to do in a completely blind search.

8 Comparing k-tries

In this section we briefly discuss how we can compare two texts \(T_1\) and \(T_2\) by means of our k-tries.

In the previous sections we have introduced the k-trie of a given text \(T\) over an alphabet \(\Sigma\), where \(T\) represents a biological sequence and \(\Sigma\) is a biological alphabet (i.e. the alphabet of nucleotides or amino acids). It is reasonable to assume that the length \(m\) of the biological sequence \(T\) will be much longer than the length \(k\) of the substrings we are interested in. Moreover, it is reasonable to assume that most of the symbols in \(\Sigma\) will be present in \(T\).

Under these hypotheses, if we have two distinct texts of the same size over the same alphabet, their k-tries might have quite similar structures and symbols, but their observed frequencies, that is substring counters, could be significantly different. Then, a comparison of the two trees has to take into account their structure and, for each two corresponding nodes, their symbols and counters. In particular, any difference on structure or symbols is worth to be highlighted, while differences on counters values have to be pointed out only up to a given range of values. A simple comparison of two k-tries can consist in finding their maximum common subtree starting from the roots of the two given trees.

Let \(T_1\) and \(T_2\) be two labeled trees. The general maximum common subtree of \(T_1\) and \(T_2\), \(T_{1,2}^{c}\), is a subtree of both \(T_1\) and \(T_2\) whose number of edges is maximum among all possible subtrees.

However, we are interested only in the maximum common subtree starting from the roots of the two given trees. This easier problem can be solved by visiting in parallel the two trees through a level-order visit.

In the following algorithm, we use the queue \(Q\) to support the parallel level-order visit and the typical queue operations: \textit{Enqueue} inserts the specified element in the queue, and \textit{Dequeue} removes and returns the value of the first
element in the queue. In our case, each element in the queue $Q$ is actually a pair of nodes: the left one is a node of $T_k^1$ and the right one is a node of $T_k^2$.

Initially, we enqueue together the roots of the two trees and then, each time a couple of nodes is dequeued, all their sons are taken into consideration and possibly enqueued. More precisely, we enqueue together any two nodes having the same symbol and with counter values ranging over a given interval of values $[\delta_{\text{min}}, \delta_{\text{max}}]$. In general we may compare two texts which have significantly different lengths, hence we have to compare the counters of corresponding nodes in a sensible way. One solution could be to normalize the counters w.r.t. the size of their text. In the following algorithm instead we normalize the counter value of each node w.r.t. the counter value of his father.

In order to highlight the maximum common subtree of the two given trees, we mark each visited node with a matching flag which can be stored in a single bit field added to each node. We assume that the matching flag of each node is initially set to zero, and that the list of sons of each node is sorted with respect to the alphabet symbols.

{the matching flags of all nodes are set to zero}
MATCHING_TREES($T_k^1, T_k^2$)
Enqueue($Q, (n_1, n_2)$);
{n1 is the root of $T_k^1$ and n2 is the root of $T_k^2$}
while not isEmpty($Q$) do
begin
($x, y$) := Dequeue($Q$);
for all $x'$ son of ($x$) and $y'$ son of ($y$) do
{use the order on symbols to optimize node comparison}
begin
if ($x'.\text{symbol} = y'.\text{symbol}$) and
($\delta_{\text{min}} \leq |x'.\text{counter}/x.\text{counter} - y'.\text{counter}/y.\text{counter}| \leq \delta_{\text{max}}$) then
begin
$x'.\text{matching flag} := 1$;
$y'.\text{matching flag} := 1$;
Enqueue($Q, (x', y')$);
end;
end;
end;
end;

The complexity of the described algorithm is the complexity of a level-order visit of the trees, that is $O(max(n_1, n_2))$, where $n_1$ (resp. $n_2$) is the number of nodes of $T_k^1$ (resp. $T_k^2$).

Note that the algorithm can accept in input trees having different depths. Moreover, observe that the admissible range of counter values $[\delta_{\text{min}}, \delta_{\text{max}}]$ is unique for the whole tree comparison: in alternative we could associate a suitable range with each level of the trees. This would allow, for example, a more strict comparison on the first symbols of each substring and a more weak one on the last symbols.
Example 34. Consider the 2-tries shown on the left and right of the following figure.

The result of applying the Matching Trees algorithm to such trees, using \([0, 0.3]\) as admissible range of values, is highlighted in both trees.

9 Related works

In this section we illustrate some other approaches which explore the problem of discovery unknown patterns in biological sequences and propose some (partial) solution techniques. Some approaches are closer to our proposal because they are based on suffix trees, other approaches are based on completely different techniques such as, for example, graph cliques (Winnover [20]) and random projections (Projection [5]).

An interesting classification of patterns in terms of which data regularities they can capture is presented in [16], as well as some applications of pattern discovery in molecular biology. The input data can be simple sequences over a given alphabet, as well as two-dimensional arrays of reals (e.g. the output of microarray experiments), or even graphs or networks. The author distinguishes various classes of patterns and, among them, substring patterns are rather similar to the type of patterns we consider. They are strings over a given alphabet, enriched with a don’t care symbol "." and they can have fixed or variable length. The input data are simple sequences, but also sequences of sets of symbols are considered and, correspondingly, patterns with a set of symbols in each position.

A recent survey on existing approaches, methodologies and tools for pattern discovery is presented in [18] by Pavesi, et. al. The focus is on discovering specific patterns, namely patterns describing transcription factor binding sites, and some of the considered methods are based on suffix trees [1, 17, 13].

In particular, Apostolico et. al. present in [1] a deep investigation on how to annotate the nodes of a suffix tree with their expected values, variances and
scores of significance, with respect to the simple probabilistic model in which sequences are produced by a random source emitting symbols from a known alphabet independently and according to a given distribution. The authors show how to perform tree annotations in an incremental and efficient way. More precisely, they prove that, given a text of length $m$, the full tree annotation can be obtained in optimal $O(m^2)$ worst case and $O(m \log(m))$ expected time and space. This result is achieved by expressing mean, variance and related scores of significance incrementally, thus allowing for their efficient computation. Note that we use the same hypotheses in our probabilistic models, so we may apply the same method for annotating the nodes of our k-trie. The same authors present in [2] a deep analysis on the monotonicity of some scores of significance w.r.t. string length. Monotonicity is in fact the key property allowing to bound the number of candidate over- and under-represented strings in a sequence and carry out the relative computations in efficient time and space.

The tool VERBUMCULUS [3] is based on the efficient techniques presented in [1]. It offers a choice among different probabilistic models and different scores of significance. It provides moreover the possibility to prune the tree according to some filters, such as limiting the size of the strings, giving a lower bound on the absolute value of the score, or giving a lower bound on the value of the expectation in order to avoid considering ”rare” words.

Other approaches based on suffix trees share the same underlying idea: they take advantage of the inherent compactness and efficiency of suffix trees to represent the input sequences and then propose methodologies and algorithms to find unknown patterns, usually potential transcription factor binding sites, by assuming some knowledge on their shape and structure, i.e. patterns with an upper bound on mismatches, patterns with gaps, structured motifs, etc.

Pavesi et. al. in [17] illustrate the Weeder algorithm for finding patterns of unknown length in DNA sequences. Starting from a suffix tree representing the input sequences, Weeder allows finding patterns with mismatches, that is patterns where only symbol substitutions are permitted. The exact length of the patterns to be found is not a needed input parameter. In order to overcome the combinatorial explosion of an exhaustive enumeration method, the algorithm imposes a restriction on the location of mismatches of the patterns to be found. The Weeder algorithm is implemented by the tool Weeder Web [19].

In [13] Marsan and Sagot describe two exact algorithms for extracting structured motifs using a suffix tree. A structured motif is composed of at least two patterns separated by a (variable) number of spacers. Each pattern represents a transcription factor binding site. Both algorithms proceed by visiting the suffix tree and looking for the defined structured motif. In order to ”skip” the spacers in the tree, the first algorithm adopt a jumping technique, while the second one temporarily modifies the tree by putting the patterns one beside the other.

Among the approaches and tools for pattern discovery which are not based on suffix trees, we mention Teiresias [21], Winnover [20], and Projection [5].

Rigoutus and Floratos propose in [21] the Teiresias algorithm and tool. It considers a specific kind of patterns, namely patterns consisting of characters
of a given alphabet $\Sigma$ plus a wild-card character ‘.’. Moreover allowed patterns must satisfy specific structure constraints: the number of wild-card characters in the pattern is limited by fixing the number of the concrete characters. The proposed algorithm is proved to be correct, that is to produce all and only the patterns meeting the required structural restrictions.

Winnower is a proposal by Pevzner and Sze [20] and it has been initially designed to solve the following problem:

**Planted $(m, d)$-motif problem**: suppose there is a fixed but unknown pattern $M$ of length $l$. The problem is to determine $M$, given $t$ input sequences each of length $n$, and each containing a planted variant of $M$ with exactly $d$ point substitutions.

An instance of this problem, where each input sequence contains exactly one planted variant of $M$, has been used as a challenge for pattern discovery methods: a planted $(15, 4)$-motif in $t = 20$ sequences each of length $n = 600$. Pevzner and Sze show that Winnower succeed in solving it, while a number of algorithms proposed in the literature fail or perform poorly.

Finally, Buhler and Tompa propose in [5] the Projection approach and tool, which is based on the use of a random projections algorithm to solve the planted $(m, d)$-motif problem. The key idea is to choose $k$ of the $m$ positions at random, consider all the substrings of length $m$ of the input sequence and use the selected positions of each substring $x$ as a hash function $h(x)$. When a sufficient number of substrings hash to the same bucket, they are selected for the planted motif $M$. The authors show that Projection can handle the $(15, 4)$, $(14, 4)$, $(16, 5)$ and $(18, 6)$ motif problem while Winnower fails all of them except for the $(15, 4)$ one.

We conclude this section by pointing out that both our work and most of the approaches presented in this section are trying to discover basically the same kind of patterns, that is patterns allowing for mismatches but having fixed length. This kind of patterns are often referred to in the literature as rigid patterns, in contrast with the so called flexible patterns allowing also for gaps and flexible length. However, while most of the approaches impose some restriction on the position and/or number of mismatches in the patterns to be discovered, our patterns can represent as many mismatches as possible. Besides, our patterns faithfully represent the interesting concrete substrings found in the original input sequence.

10 Concluding remarks

In this paper, we illustrate some techniques for pattern discovery, based on the definition, construction, annotation and manipulation of a tree data structure, the k-trie. The latter is essentially an enriched and cut suffix trie of one (or more) input sequence, representing and counting all substrings up to a fixed length $k$. We propose an on-line algorithm to build a k-trie, whose time complexity is linear with respect to the length of the input sequence.
Moreover, in order to filter out uninteresting substrings and at the same time reducing the space required by the tree, we propose some local and global pruning techniques which can be applied by performing a linear traversal of the tree.

We consider three simple scores of significance which can be associated to each node (substring) of a k-trie in order to measure its interest. For discovering which strings are the most interesting, we analyze the score behavior in terms of monotonicity w.r.t. two basic operations: symbol concatenation and string union. The first operation would allow us to select interesting strings in an incremental way while visiting the tree. The second operation is the basis for string abstraction.

Abstraction of subsequences, that is grouping similar subsequences and representing them in a compact way as patterns, seems particularly useful in the pattern discovery framework, as a mean to point out structural similarities among interesting subsequences. We propose some techniques for exploiting abstraction in two different contexts. The first one concerns searching a set of strings described by a pattern in the k-trie. The second one concerns the representation of interesting strings in a compact way, pointing out their structural similarities. This is achieved by combining selection of interesting strings and abstraction. Such techniques, however, are just a preliminary proposal: further effort is needed for a deep investigation and an efficient use of the abstraction concept. This is a main issue for our future work.

The last contribution of the paper is a simple technique for k-trie comparison. Its main motivation is to single out similarities and differences between input sequences coming for example from different DNA regions or different families of proteins by a direct comparison of the corresponding k-tries. Also this aspect, however, needs to be further investigated.

We plan to verify the real effectiveness of our approach through concrete applications and experiments on biological sequences. Currently, a software package implementing the main techniques and features of the approach is under development.

References

24. J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weight-

