Discovering Frequent Patterns from Strings (Draft)

Jaak Vilo
Department of Computer Science
P.O.Box 26
FIN-00014 University of Helsinki
Finland;
phone: +358 9 708 44193, fax: +358 9 708 44441
e-mail: vilo@cs.helsinki.fi

Abstract

We have developed a new algorithm for discovering frequently occurring patterns from strings. Our algorithm generates systematically a pattern trie while maintaining information about the occurrences of each pattern. Patterns are constructed incrementally by expanding the prefixes of the frequent-enough patterns. Only the patterns that really occur in input strings and are frequent enough, are generated and analyzed. Hence, the higher the minimum frequency threshold, the faster the algorithm works. Pattern classes that can be generated in this way include the substring patterns, substring patterns containing group characters, and patterns containing wildcard positions. By a small modification the algorithm is able to take as input a set of strings that are classified as positive and negative examples, and to construct the patterns while tracking simultaneously the occurrences in each input set. The algorithm has been successfully used for analyzing the full genome of the yeast for predicting certain regulatory elements.

Keywords: pattern discovery algorithms, suffix tree, analysis of DNA and protein sequences

1 Introduction

Pattern discovery from biosequences aims at the discovery of *a priori* unknown patterns that share some interesting properties defined by different goodness or ranking criteria. According to the three-step paradigm for pattern discovery [BJEG98], one has to choose the pattern class (the class of grammars), that is, the hypothesis space; design the rating function for the given pattern class in respect to the input examples; and develop an algorithm that is able to efficiently construct the best patterns according to the rating function.

We will present a method for discovering the patterns that occur frequently in a string or in a set of strings. We study the substring patterns and patterns that can contain group characters or wildcards. One of the main application areas of the pattern discovery algorithms is the molecular bioinformatics, where the primary sequences of the DNA and protein molecules are the primary source of data. Systematic analysis of the *a priori* unknown frequently occurring patterns can assist in understanding the classificational, structural, or functional properties of these macromolecules. For a survey about pattern discovery methods for molecular biology, see for example [BJEG98, BJVU98b].

Our pattern discovery algorithm is based on the *suffix trie* data structure. It is a simplified but more resources demanding version of the well-known data structure suffix tree [Wei73, McC76, Ukk95] (for a textbook presentation, see for example [Gus97]). We construct the suffix trie for the set of input strings S. Our construction procedure is inspired by the lazy algorithm of [GK95] for generating suffix trees. The resulting trie represents all the patterns (in the chosen class of patterns) that are present in some string in S. The nodes of the trie are labeled with symbols from the pattern representation alphabet: the individual characters, wildcards, or character groups (all nonempty subsets of the alphabet). Labels on the path from the root to an internal node spell out the pattern associated with that node. Thus we call the trie the *pattern trie*.

At each node we maintain an *occurrence list* which gives all the positions of input string where the corresponding pattern matches it. The trie is generated starting from the root. The root corresponds to the empty pattern λ whose occurrence list contains all positions of the input string. The trie is extended by generating the nodes in the trie in a systematic order. At each step the children of some of the current leaves are generated and inserted into the trie. For a node n with associated pattern p, every legal extension pc is generated by inserting a new child with label c under the node n. The occurrence list of pc is computed from the occurrence list of p by checking for each occurrence of p in the input string if it can be extended to an occurrence of pc.

The time and space requirement of our algorithm for generating all different substrings of the n characters long input string S is $O(n^2)$. By restricting ourselves to patterns occurring at least K times we can speed up the algorithm in practice. The larger is K, the faster is the algorithm. For more complex pattern classes with many wildcards and group characters the complexity of the algorithm grows exponentially as the pattern space grows very fast.

It is possible to use various restrictions on the patterns generated by our algorithm. The nodes and corresponding subtrees that lead to patterns not satisfying the restrictions will not be generated. Some of the helpful restrictions are, for example: construct only the patterns having at least K occurrences in the input strings; construct only patterns matching at least K different input strings; construct only the patterns that match at least K of the input strings corresponding to positive strings (and output those that minimize the occurrences in negative strings).

The paper is organized as follows. At first we define the input data and pattern classes that we use in our algorithm. Next we briefly introduce the suffix tree data structure that has influenced our algorithm. We start presenting our algorithm from the substring patterns, then we add group characters and wildcards. In the end we describe an application where we have successfully used our method.

2 Definitions

We will use letter Σ to denote a finite set of characters. The *size* of the character set Σ is denoted by $|\Sigma|$. Any sequence $S = a_1 a_2 \ldots a_n$, such that $n \geq 0$ and each a_i is in Σ, is called a *string* (or sequence) over character set Σ. The *length* $|S|$ of string S is n. The set of all strings over Σ is denoted by Σ^*.

String S starts from *position* 1 (the first character in a string is at position 1) and ends at position $|S|$. Character positions i of string S are thus in the range $1 \leq i \leq |S|$. Character a_i at the position i can also be denoted by $S[i]$. The consecutive characters $a_i \ldots a_j$ (or $S[i], S[i+1], \ldots, S[j]$) form a *substring* of S that starts from position i and ends at position j. We denote it by $S[i..j]$, where $1 \leq i \leq j \leq |S|$. The substring $S[i..i]$ has length 1 and corresponds to the character a_i at position i. Another way to characterize substring x of S is to say, that $S = yxz$ for some strings y and z. The substring $x = a_i a_{i+1} \ldots a_j$ of S *occurs* at the *position* or *location* i of the string S. Alternatively the occurrence can sometimes be identified by the position j. Substring x can occur in many different positions of S.

We will follow closely the definitions of [BJVU98b] in defining the generalized regular patterns.

Let $\Sigma = \{a_1, \ldots, a_m\}$ be a *basic alphabet* and let $L(a_i) = \{a_i\}$ be the *language* defined by a_i. Let $g_1, \ldots, g_n$ be nonempty subsets of Σ such that each subset contains more than one element. Let $\Gamma = \{b_1, \ldots, b_n\}$ be another alphabet disjoint from Σ. We define language $L(b_i) = g_i$, and call characters b_i the *group characters*. Let $* \notin \Sigma \cup \Gamma$ define the language $L(*) = \Sigma^*$. We call $*$ the *wild-card of unrestricted length*. We define a *flexible wild-card of length between p and q* as an expression of type $*(p,q)$ and the language $L(*(p,q)) = \cup_{i=p}^{q} \Sigma^i$ as the set of all words over Σ with the length between p and q characters. Let X be the set $\{*(p,q) \bigm| 0 \leq p \leq q \leq \infty\}$ of flexible wildcards.

We define a *generalized regular pattern* π as a string over the alphabet $\Sigma \cup \Gamma \cup \{*\} \cup X$. We define the language $L(\pi)$ of patterns $\pi = c_1 \ldots c_r$, where $c_i \in \Sigma \cup \Gamma \cup \{*\} \cup X$, as

$$L(\pi) = \{\zeta_1 \ldots \zeta_r \;\bigm|\; \zeta_1 \in L(c_1), \ldots, \zeta_r \in L(c_r)\}.$$

Pattern π *matches* the string α if there exists a substring β of α such that $\beta \in L(\pi)$. Note that in this way we assume that the pattern π can be written also as $*\pi*$. This is not a restriction as we can modify our algorithms and forbid the patterns which start or end with unrestricted wildcards.

We define the *substring patterns* as $\pi = c_1 \ldots c_r$, where each $c_i \in \Sigma$. An example of such pattern is `AAGA`. We define the *substring patterns with group characters* as $\pi \in (\Sigma \cup \Gamma)^*$. One example of such pattern is `AA[GT]A`. The character groups in Γ do not usually contain all possible groups formed from characters in Σ, but only the groups of somehow related symbols from Σ. In biosequences, i.e. the DNA and amino acid sequences, the relatedness is usually defined by the physico-chemical properties of corresponding nucleotides or amino acids.

2.1 Suffix trees and substrings of the string

Every string S of length n has exactly $n + (n-1) + \ldots + 1 = \frac{n^2+n}{2} = O(n^2)$ substrings $S[i..j]$, where $1 \leq i \leq j \leq |S|$. Depending on S, some of these substrings can be equal to each other. Surprisingly, it is possible to enumerate and index all of them in $O(n)$ time by constructing a data structure called suffix tree. There are three different linear-time algorithms for constructing suffix trees ([Wei73, McC76, Ukk95]); the textbook ([Gus97]) gives a clear exposure of these methods. The following definition is from [Gus97]:

Definition 2.1 *A suffix tree T for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at the position i. That is, it spells out $S[i..m]$.* □

The size of the suffix tree for S is linear in $|S|$. This can be seen by the following argumentation. By introducing a special end-of-string character \$, all suffixes of $S\$$ end in different leaves. As there are exactly n leaves and every internal node is a branching node, there are altogether $O(n)$ nodes in the tree. Compact version of the suffix tree can be implemented so that each node and edge has size $O(1)$ (represent each edge label with two pointers to the endpoints of the corresponding substring of S) and hence the overall size of the suffix tree is $O(n)$.

We will use the simplified, but quadratic-size version of the suffix tree called *suffix trie*. Suffix trie differs from the suffix tree in the way that every edge-label has length exactly one. Hence we loose the condition that every node has to have at least two children. This is not true in the case of suffix tries. Therefore, the size of the suffix trie can be quadratic in $|S|$.

Figure 1 gives the suffix tree and suffix trie for the string $S =$ `ATACATA`.

2.2 Construction of frequent substrings and patterns

We want to discover the generalized regular patterns consisting of single character, group character, and wildcard positions, which occur frequently in the input string S. First we will describe our algorithm for generating the substring patterns, and then add other pattern classes. The main idea of the algorithm is to start from the most frequent patterns and to specialize them in a systematic way so that unfrequent patterns are discarded as early as possible.

The algorithm can be generalized for a set of strings, and for a set of strings belonging to two or more classes. For each generated pattern the exact occurrences (positions) are calculated. Therefore it is possible to count the number of different strings containing the pattern, or the number of strings from each input class containing the pattern. Based on these counts we can define various *selection functions* to output only the patterns that correspond to the predefined *fitness criteria*.

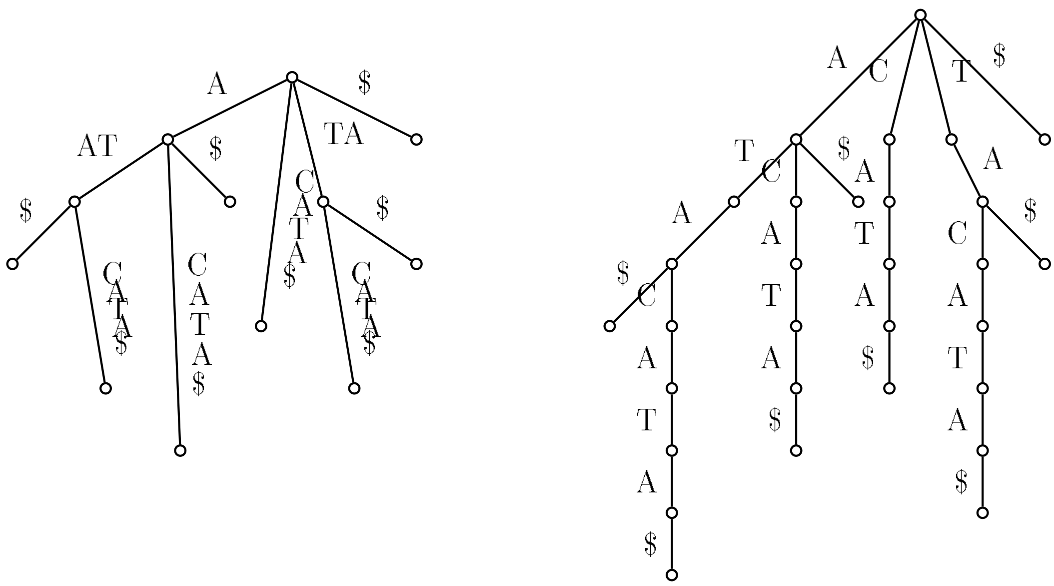

Figure 1: Suffix tree and suffix trie for the string S = `ATACATA$`. The last character, `$` is added to make all suffixes of S different from each other.

3 Frequent substrings

All substrings of the string can be discovered by constructing a suffix tree and traversing it in linear time. We want to find only the frequently occurring substrings of the string, or in other words – to solve the following problem.

Problem: Find all substrings of string S that occur in at least K different locations of S.

In the traditional solution we can construct the suffix tree in $O(n)$ time and extract the frequently occurring substrings as follows. Make a depth-first traversal over the tree and calculate for each node the number of leaves in the corresponding subtree. This number equals to the number of different occurrences of the substring corresponding to each node. Finally, output the substrings that have at least K occurrences.

Analysis of this method is straightforward. The construction of the suffix tree takes $O(n)$ time. The depth-first traversal to collect the number of occurrences of each substring can be done in $O(n)$ time. The output of the answer takes linear time too, but not in the length of the string S, but in the size of the output that can be quadratic in n. The solution seems optimal, as it should always take at least time $O(n)$ to generate the solution (we have at least to read the input data once), plus the time proportional to the size of the answer to output it.

From the practical point of view the above solution has the weakness that the space required for storing the full suffix tree can be too large. In an efficient implementation the size of the tree is in average about 10-15 times the size of S. Also, due to non-locality properties of linear-time suffix tree generation algorithms, it is hard to avoid memory paging problems [GK95].

We are interested only in the patterns that occur at least K times in S. It seems that unnecessary work has been done by constructing the subtrees with less than K leaves. Thus, there might be better algorithms that do not build the full suffix tree but only the part that contains the most frequent substrings of S.

If we knew the length l of the longest substring that occurs at least K times in S, we could use the modification of the suffix tree construction algorithm that builds the tree up to only some fixed depth l, i.e. the length of the longest frequent substring. But there does not exist any restrictive enough upper bound for the maximum depth l that could guarantee the bounded-depth suffix tree to contain all frequent substrings. Even very long substrings can occur frequently in S.

We aim at a solution that does not rely on the assumptions of randomness of S, is faster for larger values of K, keeps the space requirement low, and is simple to understand and implement. The solution is based on the previously known lazy suffix tree construction algorithm ([GK95]). Later we extend it for generating patterns from more complex pattern classes that can contain group and wildcard characters.

We will describe the algorithm for the case of suffix trie instead of the compact version of suffix tree. The compact form of suffix tree representation could be introduced as well, but this would unnecessarily complicate the representation of our algorithms.

We introduce some more notations. For identification of an individual node N in the trie we will use the string α over the trie label alphabet that spells out the labels on the path from the root to the node N. We denote that node N by $N(\alpha)$. Hence, $N(\texttt{ABC})$ is the node identified by substring `ABC`, and $N(\alpha\texttt{C})$ is the child of $N(\alpha)$ with character label `C`. Every node in the trie has some internal structure for representing the additional information about the relation between other nodes in the tree, and the substring corresponding to the node. We will use the dot-notation to represent subfields, e.g. *N.parent*, *N.child*, *N.char* and *N.sibling*. The substring α is spelled out by the character labels *N.char* along the path from the root to the node

$N(\alpha)$, $N(\alpha X).char = X$, and $N(\alpha X).parent = N(\alpha)$. Given the node N, we denote its children by $N.child(c)$ meaning the child P of node N such that $P.char = c$. The siblings of node N can be identified by shorthand notation $N.sibling(c)$, where $N.sibling(c)$ is actually $N.parent.child(c)$. Note that $N.sibling(c)$ is the same as N if $N.char = c$.

For maintaining the information about the occurrences of each substring we will use the lists of positions of S where the substring occurs. The occurrences of the substring α are stored in the node $N(\alpha)$ and denoted by $N(\alpha).pos$.

Our algorithm works as follows. Build the suffix trie for the input string S in a systematic order, e.g. in the breadth-first order, level by level. For each node $N(\alpha)$ create the list of positions $N(\alpha).pos$ to each location of S where α occurs. To represent the occurrence that ends at position j of S we use a *pointer* to position $j+1$; this is just for technical convenience. To create the children of node $N(\alpha)$, find characters $\mathtt{a} \in \Sigma$ for which the substring $\alpha\mathtt{a}$ occurs in at least K different locations of S. This corresponds to counting which characters of Σ occur at least K times at the positions $N(\alpha).pos$ of S. This can be done by one traversal of the position list $N(\alpha).pos$ and creating simultaneously all the position lists for every character occurring at these positions in S. Only these nodes $N(\alpha\mathtt{a})$ are inserted into the trie, for which the character $\mathtt{a}$ occurs at least K times at positions $N(\alpha).pos$.

Algorithm 3.1 Frequent substring generation
Input: *String S of length n, minimum number K for occurrences of substrings*
Output: *Suffix trie containing substrings that occur at least K times in string S*
Method:

```
Root ← new node; Root.char ← λ
Root.pos ← (1, 2, ..., |S|)
enqueue(Q, Root)
while N ← dequeue(Q)
    // Group the positions according to characters in S
    foreach character c ∈ Σ
        Set(c) ← ∅
    foreach p ∈ N.pos
        add p + 1 to Set(S[p])
    // Insert new child nodes for substrings that are sufficiently frequent
    foreach character c ∈ Σ such that |Set(c)| ≥ K
        N.child(c) ← new node P with label P.char = c
        P.pos ← Set(c)
        enqueue(Q, P)
    delete N.pos
return Root
```

The tree is constructed by systematically extending the leaf nodes. Thus, the position lists are needed only for the leaves during the tree construction,

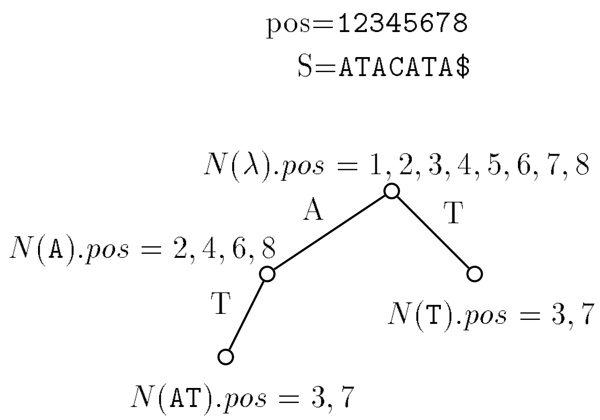

Figure 2: Discovering the substrings of string $S =$ ATACATA$ having at least 2 occurrences in S. First nodes generated into the trie represent the substrings λ, A, T, and AT.

while the position lists of internal nodes can be deleted[1]. Obviously, the total size of all position lists of the children of the same parent node can not exceed the size of the position list of the parent.

Lemma 3.2 *For any node N, $|N.pos| \geq \Sigma_{c \in \Sigma} |N.child(c).pos|$.*

No two substrings of the same length can occur at the same position in S. This means that the position lists of the nodes at depth l are disjoint. There are $n - l + 1$ possible locations for all substrings of length l. Hence, the total size of all position lists at any moment of the tree construction is at most $2n$. Moreover, it decreases during the tree construction. This means that at most $2n$ positions are kept in memory simultaneously. In practice that amount of memory is needed only closer to the root, when the tree itself is small and position lists are large.

According to Lemma 3.2 the size of the position lists along any path from the root down to leaves is non-increasing.

[1]One more advantage in constructing the tree in this way is, that all children of a node are inserted in one step. There is no need for multiple visits to internal nodes in different parts of the trie.

Lemma 3.3 *The sum of the sizes of the position lists of all current leaves of the tree is at most $n = |S|$.*

Lemma 3.4 *The total time used by Algorithm 3.1 is proportional to the number of all frequent substrings, and for each frequent substring to the number of its occurrences in S. The working space used by Algorithm 3.1 is proportional to the number of all frequent substrings plus the position lists of the leaves and their parents, i.e. less than $2n$ positions.*

At every depth l of the trie the work is proportional to the total size of the position lists of all nodes at that depth, i.e. $O(n)$. If $K = 1$, Algorithm 3.1 constructs the full suffix trie of depth n. Hence, the time requirement for Algorithm 3.1 is $O(n^2)$.

It may seem a bad idea to use this $O(n^2)$ algorithm while there are many $O(n)$ time algorithms available for constructing suffix trees. Surprisingly, the experiments of [GK95]) have shown, that the algorithm described above, when implemented to construct full suffix trees can still compete quite well with the theoretically faster linear-time algorithms, therefore making this quadratic time algorithm interesting as such.

Let us analyze the time and space complexity of our algorithm for discovering the most frequent substrings. First notice, that the size of the trie structure is optimal in the sense that only the nodes $N(\alpha)$ corresponding to substrings α that occur at least K times in S, are inserted into the trie. Additional work has to be done for creating and maintaining the position lists. As we have noticed already, the total size of all position lists of the nodes at any depth of the suffix trie is $O(n)$. The worst-case time complexity of Algorithm 3.1 hence depends on the depth of the tree.

The size of the trie is not known before the trie is built. As we have noted earlier, the work that is done at any depth l in the trie is $O(n)$. Hence, the overall time complexity is $O(nd)$, where d is the depth of the tree.

Given a random string S with uniformly distributed characters over character set Σ, all patterns of the same length are equally probable to occur in S. The position lists of each node N in the trie have hence approximately $|N.parent.pos|/|\Sigma|$ elements. I.e. the pattern defined by $N.parent$ can be extended by all possible characters from Σ with equal probability. This divides the occurrence list of the parent into $|\Sigma|$ approximately equal subsets. Hence, for $l > log_{|\Sigma|}\frac{|S|}{K}$ the sizes of individual position lists of the nodes at depth l contain already less than K elements. Therefore, we can conlude the following theorem.

Theorem 3.5 *The overall time complexity of Algorithm 3.1 for constructing* *the frequently occurring (i.e. when $K > 1$) substrings of the random string S is $O(|S|(log_{|\Sigma|}|S| - log_{|\Sigma|}K))$.*

The overall time complexity of the algorithm is proportional to the size of the trie of frequently occurring substrings plus the total size of all the position lists for each of those frequent substrings. In the worst case, the algorithm can be quadratic in $|S|$ even for large K. One example of such worst case is the string $S = aaaaaa \ldots a$. Or in general, the strings where long substrings occur very frequently.

The order of the tree construction is determined by the implementation of the queue Q. If it is a standard FIFO queue, the construction is done breadth-first, level by level. This allows to output the result already during the search in a systematic order from shorter to longer patterns, and all the substrings of the same length ordered alphabetically.

The construction and/or output order can also be different. For example, if queue Q acted like a stack, the search order would be depth-first. Depth-first construction should be more efficient in the terms of memory-paging.

From our point of view it is important that the conceptual simplicity of this algorithm makes it suitable for modifications for finding frequent patterns that are more complex than the substrings.

4 Frequent substrings with group characters

Recall the character grouping and patterns of the type $\pi \in (\Sigma \cup \Gamma)^*$ from the Section 2. We will describe the solution to the following problem.

Problem: Find all patterns over the extended character set $\Sigma \cup \Gamma$ that occur in at least K different positions of S.

To solve this problem we will modify the Algorithm 3.1 to include the symbols from the extended character set Γ as follows. Patterns α corresponding to leaves of the trie are extended by symbols $c \in \Sigma \cup \Gamma$. New nodes $N(\alpha c)$ are inserted into the trie if pattern αc occurs at least K times in S. For $g \in \Gamma$ clearly $N(\alpha g).pos = \bigcup_{c \in g} N(\alpha c).pos$. Hence, it is not necessary to store $N(\alpha g).pos$ explicitly if all nodes $N(\alpha c)$, $c \in g$, are inserted into the trie with corresponding $N(\alpha c).pos$ even if $|N(\alpha c).pos| < K$, meaning that they are not extended later. Sets $N(\alpha c).pos$, $c \in \Sigma$ are disjoint and hence $|\bigcup_{c \in g} N(\alpha c).pos| = \Sigma_{c \in g}|N(\alpha c).pos|$. The nodes where the size of the set of positions is less than the threshold K are not extended.

Algorithm 4.1 Generation of frequent patterns with character group positions
Input: *String S of length n, character groups Γ, threshold K*
Output: *Pattern trie with nodes defining patterns over $\Sigma \cup \Gamma$ that occur at least K times in string S*
Method:

1. $Root \leftarrow$ *new node;* $Root.char \leftarrow \lambda$
2. $Root.pos \leftarrow (1, 2, \ldots, n)$
3. $enqueue(Q, Root)$
4. **while** $N \leftarrow dequeue(Q)$
5.   *// Construct the position list for pattern defined by node N*
6.   **if** $N.char \in \Sigma$ **then** $Pos \leftarrow N.pos$
7.   **else** $Pos \leftarrow \bigcup_{c \in N.char} N.sibling(c).pos$
8.   *// Group the positions according to characters in string S*
9.   **foreach** $c \in \Sigma$
10. $Set(c) \leftarrow \emptyset$
11.   **foreach** $p \in Pos$
12.   *add* $p + 1$ *to* $Set(S[p])$
13.   **foreach** *character* $c \in \Sigma$
14.   **if** $(\ |Set(c)| \geq K\ )$ **or**
15. $(\ \exists g \in \Gamma$ *s.t.* $c \in g$ **and** $\Sigma_{f \in g}|Set(f)| \geq K\ )\ )$
16.   **then**
17. $N.child(c) \leftarrow$ *new node P;* $P.char \leftarrow c$
18. $P.pos \leftarrow Set(c)$
19.   **if** $(\ |Set(c)| \geq K\ )$ $enqueue(Q, P)$
20.   **foreach** *character* $g \in \Gamma$
21.   **if** $\Sigma_{c \in g}|Set(c)| \geq K$ **then**
22.   **if** $\exists f$ *s.t.* $f \in (\Gamma \cup \Sigma), f \subset g, \Sigma_{c \in f}|Set(c)| = \Sigma_{c \in g}|Set(c)|$
23.   **then** *// More specific pattern covers already the same positions*
24.   **else**
25. $N.child(g) \leftarrow$ *new node P;* $P.char \leftarrow g$
26. $enqueue(Q, P)$
27.   **if** *all nodes* $N.sibling(c)$, $c \in \Sigma \cup \Gamma$ *have been expanded*
28.   *delete all* $N.sibling(c).pos$, *where* $c \in \Sigma$

In Algorithm 3.1 for discovering the frequent substrings, all the position lists of the nodes at the same level in the trie were disjoint. Now we use group characters and hence different patterns of the same length can match at the same positions of S. It means that the total size of position lists at one level in concept trie can grow exponentially in the number of group positions in the patterns. We have to be careful in choosing which extensions are interesting when creating the frequent patterns.

We should never overgeneralize the patterns if it is not supported by data. If the only possible extension of pattern α is by character `A`, then it is unnecessary to construct the pattern α[`AB`] to the pattern trie as any of the patterns $\alpha A \beta$ and α[`AB`]β can only have exactly the same occurrences in string S. This overgeneralization is tested on line 22 of Algorithm 4.1. An extension of a pattern by a group character g has to give more matches in string S than all other extensions f, where $f \subset g$.

Lemma 4.2 *For all nodes N and symbols $c \in \Sigma \cup \Gamma$, $|N.pos| \geq |N.child(c).pos|$, i.e. the sizes of the position lists are nonincreasing along any path from the root down to leaves.*

Note, that the position lists $N(\alpha g).pos$, where $g \in \Gamma$ are stored only implicitly – they can be constructed from the lists $N(\alpha c).pos$, where $c \in \Sigma$.

There are $n - l + 1$ possible substrings of length l of string S, $|S| = n$. By allowing character group positions we essentially allow that some positions of substrings can be represented by a group of characters instead of one character. There are more potential occurrences of such patterns in S, of course. Let us consider possible patterns where for each character c of Σ there are p possible character groups that contain c. Or, in other words – there are p possible replacements for each character in a substring. If we allow at most x positions of a substring of length l to be replaced by one of these p character groups, there are $1 + pl + p^2 l(l-1) + \ldots + p^x l(l-1) \cdots (l-x+1) = O(xp^x l^x)$ different possible patterns that can match this substring. As the string S can contain $O(ln)$ substrings of length less than l, the total number of patterns that can match to some substring of S is $O(xnp^x l^{x+1})$. In Algorithm 4.1 the patterns are constructed incrementally by expanding shorter patterns one position at a time. The work used for creating each pattern is proportional to the number of occurrences of each pattern. In the worst case it is $O(n)$. Hence, the worst case time complexity of Algorithm 4.1 is $O(xn^2 p^x l^{x+1})$. In reality only the patterns that occur frequently in S, i.e. the patterns that match to at least K different substrings of S, are constructed.

Lemma 4.3 *The maximum number of patterns of length $\leq l$ over $\Sigma \cup \Gamma$ that may occur in string S of length n is $O(xnp^x l^{x+1})$, where x is the maximum number of group characters $g \in \Gamma$ allowed in the pattern, and for all $c \in \Sigma$, p is the maximum number of group characters g such that $c \in g$. The worst case time complexity of Algorithm 4.1 is $O(xn^2 p^x l^{x+1})$.*

In practice, it is easy to modify Algorithm 4.1 so that on each path from the root only x group characters are allowed at maximum. This can be done by adding a counter of group characters that have been used so far to each node. If the limit is exceeded, then only the single character extensions ($c \in \Sigma$) are allowed.

We do not have to store explicitly the position lists for patterns ending with a group character. If the search for patterns is done in depth-first order, only the position lists of the nodes (and the immediate siblings of these nodes) along one path from the root to the leaf, have to be kept in memory. The position lists of siblings do not overlap, hence at most n positions are stored in them. This means that during the construction of the trie of depth l, only $O(ln)$ positions are stored in the nodes along the path from the root to the leaf at depth l. Hence, the space requirement of Algorithm 4.1 is $O(T + ln)$, where T is the size of the trie of frequently occurring patterns and ln is the amount of memory used for storing the position lists along one path from the root to leaf at depth l (the length of the longest pattern).

5 Frequent subsequences

Next we will study the discovery of patterns that contain variable width wildcard positions. Variable width wildcard means unspecified number of any characters. If we use character $*$ for denoting the wildcards, our patterns are strings over alphabet $\Sigma \cup \{*\}$ with the restriction that first and last symbols of our patterns have to be from Σ. As earlier, we are interested in the patterns that occur frequently in the input string S. Note, that subsequences of the string can be considered as patterns containing variable-length wildcard symbols. Hence, we are interested in the frequently occurring subsequences of S.

Due to variable width wildcards we have to be more precise about how we define the frequently occurring patterns. We define that separate occurrences of the subsequence pattern have to end at different positions in the string.

Example: Some of the subsequences of string `ABAACA` are, for example, `AB*C`, `A*A*C`, and `A*A`. Thus `A*A` has three occurrences in `ABAACA` and `A*C` and `AB*A*C` only one.[2] The pattern `B` occurs once while pattern `B*A` has three occurrences.

We consider only patterns π over $\Sigma \cup \{*\}$ such that the first and last symbol of π are in Σ. Patterns of the form $\alpha*$ and $*\alpha$ are excluded because they occur at exactly the same locations as α.

Technically that is implemented by building the tree over character set $\Sigma \cup \{* \times \Sigma\}$ meaning that whenever we use the wildcard, we should also extend it immediately by some of the characters in Σ. If $\Sigma = \{$`A,C,G,T`$\}$ then we will build the pattern trie over the character set $\{$`A, C, G, T, *A, *C, *G, *T`$\}$.

Let S be a string $S \in \Sigma^*$ of length n. Let us consider the string $S' = S\#a^n$, s.t. $\# \notin \Sigma$, and $|S'| = 2n + 1$. Every pattern $\alpha * \# * a$, where α is a

[2] The subsequences `A*C` and `B*C` are considered two different patterns although they end at the same position.

substring of S occurs n times in S' even though $\alpha * \#$ can occur only once. Hence, the definition of frequency is not so intuitive any more.

In practice we are usually interested in the patterns that occur frequently in a set of many input strings $S^m = \{S_1, \ldots, S_m\}$, not just one string S. We will map the strings from S^m to one single string S by joining the individual strings from S^m to each other so that different strings are separated by a delimiter $\# \notin \Sigma$. I.e., $S = \#S_1\#S_2\# \ldots \#S_m\#$. Given the functions that are able to determine the delimiters from S and a mapping from the positions in S to the number of string in S^m (i.e. the mapping $\{1, \ldots, |S|\} \rightarrow \{1, \ldots, m\}$), we can use a slight modification of all our algorithms. In this case we are interested in the patterns that occur frequently in S^m, i.e. they occur in as many S_i as possible. We will denote by *setofstrings*(Set) the function that maps the list of positions Set of the string S into the list of the numbers of individual strings from S^m by in the time $O(|Set|)$.

Algorithm 5.1 Frequent subsequence generation
Input: *Input strings* $S^m = \{S_1, \ldots, S_m\}$ *, threshold* K
Output: *Pattern trie with nodes defining patterns over* $\Sigma \cup \{* \times \Sigma\}$ *that occur in at least* K *different strings from* S^m
Method:

```
S ← #S1# ... #Sm#
Root ← new node with empty label Root.char
Root.pos ← (1, 2, ..., |S|)
enqueue(Q, Root)
while N ← dequeue(Q)
    // Group the positions to separate sets by actual characters in string
    foreach p ∈ N.pos
        add p + 1 to Set(S[p])
        // Create position lists for '*c' extensions.
        w ← 0
        while S[p + w] ≠ '#' // Until the delimiter
            add p + w + 1 to Set('*S[p + w]')
            w ← w+1
    // Construct the nodes defining the frequent patterns
    foreach character c ∈ (Σ ∪ '*Σ') where |setofstrings(Set(c))| ≥ K
        N.child(c) ← new node P with P.char ← c
        P.pos ← Set(c)
    delete Set(c)
    delete N.pos
```

For practical reasons the algorithm can be implemented with the restriction that the wildcards have some maximal width $w \leq W$, and there is only restricted number of wildcards in each pattern. This can be implemented by adding a counter of wildcard characters to each node which tells how many wildcards have been used so far on the path from the root to current node.

If the limit of wildcard characters is exceeded, then only the fixed character extensions are allowed.

The $X(k, l)$ representation for flexible width wildcards can be computed by counting the actual lengths of the wildcard positions in text and replacing each $*$ by corresponding $X(k, l)$ symbol. Or, if the k and l were given as parameters, these values could be used during the pattern trie construction in order not to allow longer wildcards.

6 Frequent patterns with groups and wildcards

By combining all three algorithms, we can develop an algorithm that systematically generates and evaluates all possible patterns that occur most frequently in the string. The trie of patterns is built in the same manner as for earlier cases. For each pattern in a trie we keep track of its occurrences in S. Patterns are extended by counting directly from the data, which possible extensions of the pattern can still produce frequent patterns. The possible extensions for each pattern are single character positions, group characters, fixed or variable-width wild-cards followed by some single character position or character group.

There are no restrictions in which order the pattern trie should be generated and evaluated. In every node we have already the information about the occurrences of the corresponding pattern, and the extensions are computed from the actual characters in original string S. Therefore, we can build the tree in the order we like: the breadth-first, depth-first, best-first, or other. The order of the search and reporting of the patterns can depend on the application.

For example, if we define some goodness measure for every pattern, we can extend greedily the patterns that have the highest potential for best goodness measure and thus look as the best choices for further study. We can use efficient priority queue implementations to keep track which patterns should be studied next, and build the pattern trie as long as time and space allow.

We will generate the patterns in the order of their frequency (number of occurrences of the pattern). We will maintain a priority queue of patterns based on the frequency of the pattern occurrences. We will specialize the most frequent patterns and insert all the extended patterns to the same priority queue until there remains no patterns that can have the same number of occurrences. When all the patterns of, say, frequency M have been studied, all their children nodes define the patterns that can have at most $M - 1$ occurrences. All the patterns having frequency equal to M can be now reported in the order of the goodness measure (the goodness measure is up to application). This is needed for reporting only the best (based on some ranking criteria) patterns for each frequency. For example, the best 10 patterns covering all input strings, all minus one input strings, etc.

Algorithm 6.1 *Generating the patterns that occur frequently in the set of input strings*
Input: *Strings* $S^m = \{S_1, \ldots, \S_m\}$*, pattern class , threshold* K
Output: *Patterns from pattern class in the order of frequency.*
Method:

$S \leftarrow \#S_1\# \ldots \#S_m\#$
Root $\leftarrow$ *new node; Root.char* $\leftarrow \lambda$
Root.pos $\leftarrow (1, 2, \ldots, |S|)$
$Enqueue(Q, Root, size(Root.pos))$ *// Priority queue based on the set size*
$Size = size(Root.pos)$
while $N \leftarrow dequeue(Q)$
    if $size(N.pos) < Size$
        // Print best patterns occurring $Size$ *times (or in* $Size$ *strings in* S
        while $(PN = dequeue(PQ))$ *output pattern defined by* PN
    $Size = size(N.pos)$
    $Enqueue(PQ, N, cost(N))$
    // Generate all possible extensions P *to pattern* $label(N)$ *from position list* $N.pos$
    foreach *extension* P *of* N
        if *pattern* P *and position list P.pos fulfill the criteria*
        then
            if *Exists node* M *s.t. position list P.pos is equal to M.pos*
            then *join the subtrees* M *and* P
            else $N.sibling \leftarrow P$ **and** $Enqueue(Q, P, size(P.pos))$
// Print all the remaining patterns
while$(PN = dequeue(PQ))$ *output pattern defined by* PN

The function $size(N(\alpha).pos)$ used in Algorithm 6.1 can be either the number of positions where pattern α occurs, or the number of different strings from S^m matched by α. The order of frequency can hence be based either on the number of occurrences or the number of different strings that are matched by the pattern.

7 Discussion

We have used our algorithm for predicting regulatory elements from gene upstream regions of the full genome of yeast *Saccharomyces Cerevisiae*. The algorithm seems to be appropriate for large-scale analysis of real biosequence data [BJVU98a].

First we looked for patterns that occur more frequently in the gene upstream regions than in randomly chosen regions in the yeast genome. For each pattern present that is present in at least 10 sequences (from more than 12000) we calculated a score equal to the ratio of the number of upstream regions that contain the pattern divided by the number of random regions (of the same length and number) that contain the pattern, and rated the patterns according to this ratio.

In the second case we used information from the yeast genome expression data [DIB97] to cluster the genes according to their expression profiles. After clustering the upstream regions (treating the expression measurements as time-series) we selected "characteristic" clusters according to some rigorous criteria. We hypothesized that the genes in a cluster may contain binding sites for the same transcription factors or other common regulatory elements. We used our algorithm to look for patterns that are overrepresented in each cluster as compared to other upstream regions.

The ability to construct suffix tree in linear time and space makes it attractive starting point for further algorithmic developments. On the other hand, the full suffix tree construction is not needed for example in the case where we want the patterns to occur very frequently in input strings. Also, the efficient implementation of linear-time suffix tree construction algorithm can be quite tricky due to many implementation details and non-locality effects at the tree construction time. We adopted the lazy approach of [GK95] to suffix tree generation as it does not make any "unnecessary" work.

Variations of this conceptually simple pattern generation algorithm are possible. For instance, one needs to decide in which order the nodes are processed. If all nodes at level k are processed before the nodes at level $k+1$, then we know which patterns at level k can fulfill the requirements. This can be used to decide which nodes to explore at level $k+1$. For example, in order for the pattern `ATCG` to have at least K occurrences, both pattern `ATC` and pattern `TCG` need to have at least K occurrences. Another useful technique is based on detecting nodes with identical occurrence lists. Because the subtrees below these nodes will be identical, one can avoid duplicate work by generating and representing each subtree only once. We implemented this method by using a search tree data structure for finding identical occurrence lists. We eliminate equal subtrees and try to minimize the size of the search structures.

The implementation of the methods described in this paper is available for use at `http://www.cs.Helsinki.FI/~vilo/bio/`.

8 Acknowledgments

We would like to thank Esko Ukkonen, Inge Jonassen, and Alvis Brazma for valuable comments during the development of the pattern discovery algorithm and reading the manuscripts of the current paper.

References

[BJEG98] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to automatic discovery of patterns in biosequences. *Journal of Computational Biology*, 5(2):277–304, 1998.

[BJVU98a] A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen. Predictin gene regulatory elements *in silico* on a genomic scale. 1998. To be published in *Genome Research*.

[BJVU98b] Alvis Brāzma, Inge Jonassen, Jaak Vilo, and Esko Ukkonen. Pattern discovery in biosequences. In *Proceedings of Fourth International Colloquium on Grammatical Inference (ICGI-98)*. Springer, July 1998.

[DIB97] J. L. DeRisi, V. R. Iyer, and P. O. Brown. Exploring the metabolic and genetic control of gene expression on a genomic scale. *Science*, 278:680–686, 1997.

[GK95] R. Giegerich and S. Kurtz. A comparison of imperative and purely functional suffix tree constructions. *Science of Computer Programming*, 25(2–3):187–218, 1995.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, Cambridge, 1997.

[McC76] E. M. McCreight. A space–economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Conference Record, IEEE* 14^{th} *Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.