Finding Missing Patterns

Shunsuke Inenaga, Teemu Kivioja, and Veli Mäkinen

Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
{inenaga,kivioja,vmakinen}@cs.helsinki.fi

Abstract. Consider the following problem: Find the shortest *pattern* that does not occur in a given *text*. To make the problem non-trivial, the pattern is required to consist only of characters that occur in the text. This problem can be solved easily in linear time using the suffix tree of the text. In this paper, we study an extension of this problem, namely the *missing patterns problem*: Find the shortest *pair of patterns* that do not occur close to each other in a given text, i.e., the distance between their occurrences is always greater than a given threshold α. We show that the missing patterns problem can be solved in $O(\min(\alpha n \log n, n^2))$ time, where n is the size of the text. For the special case where both pairs are required to have the same length, we give an algorithm with time complexity $O(\alpha n \log\log n)$. The problem is motivated by optimization of multiplexed nested-PCR.

1 Introduction

For a decade, *pattern discovery* has played a central role in bioinformatics [15]. Especially extracting surprising and useful patterns is a core of knowledge discovery from textual data [4, 13]. One extreme example of surprising patterns is *missing patterns*, namely, patterns that do *not* appear in a given text T are sought. Amir et al. [1] introduced a generalized version of the missing pattern problem in such a way that pattern P may 'approximately' occur in T. They call this problem the *inverse pattern matching* problem. Some improvements for this inverse problem appeared in [5]. Another related work is the *farthest substring problem* by Lanctot et al. [10], where a set of text strings is considered as input.

In this paper, we explore another type of extension of the missing pattern problem: given a text T and threshold value α, find the *shortest pair of patterns* such that the distance between their occurrences in T is *always greater than* α (see Fig. 1). We show that this problem is solvable in $O(\min(\alpha n \log n, n^2))$ time, where n is the length of T. For the case that the lengths of the two patterns have to be the same, we present a simpler and slightly more efficient solution when α is small; we achieve time complexity $O(\alpha n \log\log n)$. Not only is our missing patterns problem interesting in theory, but it is also well-motivated in practice. Indeed, we will show how missing patterns can be used to optimize the sensitivity of PCR.

The rest of the paper is organized as follows: In Section 2 we give definitions, and introduce our biological motivations and data structures. Section 3 presents our $O(\min(\alpha n \log n, n^2))$-time algorithm for finding a missing patterns pair in general cases, and in Section 4 we develop specialized algorithms for cases where the two patterns are required to be of the same length. In section 5 we make some preliminary experimental observations.

2 Preliminaries

2.1 Definitions

A *string* $T = t_1 t_2 \cdots t_n$ is a sequence of *characters* from an ordered *alphabet* Σ of size σ. A *substring* of T is any string $T_{i \ldots j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A substring of length k is called *k-mer*. A *suffix* of T is any substring $T_{i \ldots n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1 \ldots j}$, where $1 \leq j \leq n$. Suffixes and prefixes can be identified by their starting and ending positions, respectively. A *pattern* is a short string over the alphabet Σ. We say that pattern $P = p_1 p_2 \cdots p_k$ *occurs* at position j of *text* string T iff $p_1 = t_j, p_2 = t_{j+1}, \ldots, p_k = t_{j+k-1}$. Such positions j are called the occurrence positions of P in T.

A *missing pattern* P (with respect to text T) is such that P is not a substring of T, i.e., P does not occur at any position j of T. Let $\alpha > 0$ be a threshold parameter. A *missing pattern pair* (A, B) is such that if A (resp. B) occurs at position j of text T, then B (A) does not occur at any position j' of T, such that $j - \alpha \leq j' \leq j + \alpha$. If (A, B) is a missing pair, we say that A and B do not occur *α-close* in T. These notions are illustrated in Fig. 1.

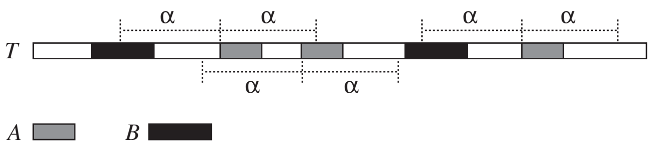

Fig. 1. Missing pattern pair (A, B).

We study the following problem:

Problem 1 *(**Missing Patterns Problem**) Given a text T and threshold α, find patterns A and B of minimum total length, such that (A, B) is a missing pattern pair with respect to T, i.e., A and B do not occur α-close in T.*

2.2 Biological Motivation

The missing patterns problem is biologically motivated by finding good adapters for primers used in a polymerase chain reaction (PCR). PCR is a standard technique for producing many copies of a region of DNA [3]. It is routinely used for example in medicine to detect infections and in forensic science to identify individuals even from tiny samples. In PCR a pair of short fragments of DNA called primers is specifically designed for the amplified region so that each of them is complementary to the 3' end of one of two strands of the region (see also Fig. 2). Given single-stranded DNA molecules, the primers hybridize to their binding sites flanking the target region. An enzyme called DNA polymerase adds nucleotides after each primer using the other strand as template and thus builds a copy of the strand started by the primer. The molecules are made single-stranded again by heating and the process is repeated many times (20-30) resulting in an exponential blow-up in the number of copies of the target region.

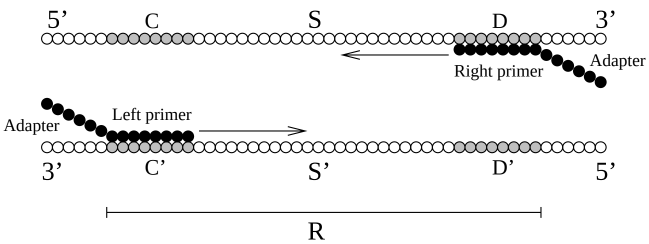

Fig. 2. PCR.

PCR is a sensitive method: in theory one can detect a couple of DNA molecules in a complex mixture of different molecules by using 20-30 amplification cycles but in practice the achieved sensitivity is lower. In particular, it is difficult to combine specificity of amplification with sensitivity. The sample can include homologous sequences that need to be separated from each other, such as sequences from several bacterial species, and the specific primers can thus bind to other sites than their intended target sites and start amplification of an incorrect region of DNA. The problem can be alleviated by using stringent conditions in PCR reaction but that can also decrease the amplification efficiency of the correct target leading to lower sensitivity. In order to achieve ultrasensitive detection, repeated PCR with nested primers, so-called nested PCR, is used.

Today, detection tests are performed preferably in a multiplexed fashion. In PCR, it means that several regions are amplified simultaneously in the same assay. Selecting specific primer pairs for multiplex PCR is a hard computational problem [12] but outside the scope of this paper.

In this work we describe adapters for specific PCR primers that facilitate the set-up of multiplexed nested-PCR assays. An adapter-primer pair is designed to further amplify any fragment amplified by the specific primers. Adapter primers work by binding to short sequences called adapters attached to the ends of all specific primers (see Fig. 2). Further amplification by adapter primers increases sensitivity of detection. On the other hand, using the same pair of primers for all fragments in the latter step of nested PCR simplifies multiplexing.

A crucial requirement for a good adapter primer pair is that it should only bind to the adapters and should not amplify anything directly from the sample. However, since the adapter sequences can be chosen freely, we have a huge number of options available. Motivated by the demand for ultrasensitive and multiplexed detection, we aim for optimal selection of adapter primers.

Assume that we want to select such a pair of primers that they do not amplify a certain region R ($|R| < \alpha$) and let that region be flanked by patterns C and D in the sequence S (see Fig. 2). The sequence S denotes all the sequences that can be present in a sample, such as the human genome and the genomes of the bacteria or viruses that possibly are the cause of an infection. Let us denote the reverse complement of a sequence S by S' (the choice of one strand as S and the other as S' is arbitrary). The region R can be amplified by a primer pair only if the left primer binds to C' and the right primer binds to D. Thus, if we have found a missing pair (A, B) for the strand S, then (A', B) is such a pair of patterns that there is no such region of length less than α that A' is the binding site of the left primer and B is the binding site of the right primer. However, (A', B) can also work the other way round, i.e. there can be such a region that A' is a binding site of the right primer and B is the binding site of the left primer. But in that case the pair (A, B) would occur α-close in the strand S'. In conclusion, if (A, B) is a missing pair for both strands of the sample sequence, i.e. for the string SS', then (A', B) is not a pair of primer binding sites for any region of length less than α. A safe threshold α can be determined based on the speed of the polymerase reaction and duration of the PCR cycle.

Until now we have ignored the fact that a primer can bind and initiate the polymerase reaction even if the primer and the binding site are not exact complements. Especially, if the 3' end of the primer binds to a site, it can initiate the polymerase reaction even if the 5' end of the primer dangles freely. That is why we search for the shortest missing pair (A_{min}, B_{min}), where $|A_{min}| = |B_{min}| = k$. Then, we can take the pair (A'_{min}, B_{min}) as the 5' ends of the adapters and (A_{min}, B'_{min}) as the 3' primer ends of the adapter primers. Such a selection guarantees that if the first k nucleotides of the left primer match a binding site in the sample sequence, the right primer has at least one mismatch in the first k nucleotides and vice versa. In addition, no pair of primers satisfies this condition for $k-1$. Therefore, using the shortest missing pair in the construction of adapter primers minimizes the risk of any unwanted amplification. Notice that Problem 1 is a generalization of the objective described above; all of our algorithms can be adopted for this special case.

The adapter primers also have to satisfy other requirements. For example, the melting temperature of both primers should be within a specified range. The primers should not form stable hairpin loops or bind to each other. These additional requirements can be satisfied because there most probably are many missing pairs of the same length and the missing pairs are short compared to the length of the primer which typically is around 17..25 bases. We show in Section 5 that these assumptions are valid for the yeast genome. Thus, the 3' ends of the adapter primers can be chosen from several alternatives and the 5' ends can be chosen freely to satisfy the other requirements.

2.3 Data Structures

We use some well-known string data structures in our algorithms, such as *keyword tries* and *suffix trees*. Let us briefly recall these structures.

Definition 2 *(Adopted from [6]) The* keyword trie *for set $\mathcal{P}$ of patterns is a rooted directed tree $\mathcal{K}$ satisfying three conditions: (1) Each edge is labeled with exactly one character; (2) any two edges out of the same node have distinct labels; (3) every pattern P of $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on the path from the root of $\mathcal{K}$ to v spell out P, and every leaf of $\mathcal{K}$ is mapped to by some pattern in $\mathcal{P}$.*

Definition 3 *The* suffix trie *of text T is a keyword trie for set $\mathcal{S}$, where $\mathcal{S}$ is the set of all suffixes of T.*

Definition 4 *The* suffix tree *of text T is the* path-compressed *suffix trie of T, i.e., a tree that is obtained by representing each maximal non-branching path of the suffix trie as a single edge labeled by the catenation of the labels in the corresponding edges of the suffix trie. The labels of the edges of suffix tree correspond to substrings of T; each edge can be represented as a pair (l, r), such that $T_{l \ldots r}$ gives the label.*

Definition 5 *The* sparse suffix tree *of text T is a suffix tree built on a subset $\mathcal{S}'$ of suffixes of T, i.e., a path-compressed keyword trie for set $\mathcal{S}'$.*

The suffix tree of text $T = t_1 t_2 \cdots t_n$ takes $O(n)$ space, and it can be constructed in $O(n)$ time [16, 11, 14] (omitting the alphabet factors). A sparse suffix tree can be pruned from the (full) suffix tree in linear time [8, 2]. (Direct constructions of sparse suffix trees were also considered in [8, 2], but $O(n)$ time is enough for our purposes.)

The nodes of all the above-defined trees can be partitioned into two classes: (1) A node is *complete* if it has an edge $e(c)$ for each $c \in \Sigma$ such that the label of edge $e(c)$ starts with character c; (2) otherwise the node is *incomplete*. Let us denote by $label(v, s)$ the catenation of labels between two nodes v and s. With the *depth* of a node v we mean $|label(root, v)|$.

We sometimes refer to *implicit nodes* of the suffix tree, meaning, in addition to all (explicit) nodes of the suffix tree, also the positions on the edge labels of the suffix tree, as they all correspond to nodes of the corresponding suffix trie.

3 General Algorithm using Sparse Suffix Trees

Let us first describe how the one pattern case (as mentioned in the abstract) can be solved. That is, we wish to find a pattern of minimum length that does not occur in a text of length n. The solution is as follows. Build the suffix trie of the text. Among all incomplete nodes of the trie, select the one that has the minimum depth. Let that node be v and let the character that makes the node incomplete be c. Then the answer to the question is $label(root, v)c$. The size of the suffix trie can be $O(n^2)$, which makes this algorithm inefficient. The same algorithm can be simulated using the suffix tree, which reduces the running time to $O(n)$; instead of scanning through all implicit nodes of the suffix tree, we can check the explicit nodes for incompleteness and for each edge whose label is longer than 1, we know that the implicit node corresponding to the first letter on the label is incomplete.

3.1 Basic Properties

The topic of this paper, the two-pattern case defined in Problem 1, is more challenging. However, some aspects of the one-pattern solution can be exploited, as summarized in the following observation.

Observation 6 *(Monotony property) Let v be a node of the suffix tree of text T, and let e be an edge out of v labeled $L = l_1 l_2 \cdot l_p$. Then, string $label(root, v)l_1$ occurs in T exactly at the same positions as any string $label(root, v)L_{1...i}$, where $1 < i \leq p$.*

Before using the monotony property, we also mention the following simple but important lemma.

Lemma 7 *(Substring property) It holds that either (i) there is a solution to the missing patterns problem, say pair (A, B), such that both A and B are substrings of the text; or (ii) the solution is a single pattern.*

Proof. Let (A, B) be a solution to the missing pairs problem such that A is not a substring of the text. Then (A, ϵ) is also a missing pair. Since $|A| + |\epsilon| = |A| \leq |A| + |B|$, pair (A, B) can not be the shortest missing pair, unless B is an empty string, in which case A is a single pattern solution. □

The above lemma states that we can restrict to selecting both A and B as non-empty substrings of T. The case where one pattern is enough was considered at the beginning of this section.

3.2 Basic Algorithm

Let V be the set of all nodes of the suffix tree of text T, and let $\mathcal{P}$ be the set of strings obtained by adding to each $label(root, v)$, $v \in V$, all starting characters of labels on the out edges of v. It is easy to see that $|\mathcal{P}| \leq 2n - 1$; the size of $\mathcal{P}$ is bounded by the number of internal nodes in the tree. Finally, let $Occ(P)$ be the list of occurrences of pattern $P \in \mathcal{P}$ in T; it can be obtained in time $O(|Occ(P)|)$ from the suffix tree.

Recall that we are interested in finding a missing pair (A, B). Let us choose as A a string from $\mathcal{P}$. Our goal is to choose B so that (A, B) will be a missing pair. As A is now fixed, we try to choose B of minimum length. Let us, for now, assume that we have found pattern B of minimum length such that (A, B) is a missing pair. The crucial observation is that if we repeat this process for all $A \in \mathcal{P}$, we can choose among all the missing pairs found so far, the one where the sum $|A|+|B|$ is minimized. The correctness of this procedure follows directly from Observation 6 and Lemma 7.

What is left is to explain how to choose B of minimum length so that (A, B) will be a missing pair. This is done as follows. Let us define a set $Zone(A, \alpha)$:

$$Zone(A, \alpha) = \cup_{j \in Occ(A)} [j - \alpha, j + \alpha].$$

We have the following observation:

Observation 8 *If and only if B is a prefix of any suffix $T_{j' \ldots n}$ such that $j' \in Zone(A, \alpha)$, then pair (A, B) occurs α-close in T.*

Now, building the sparse suffix tree over suffixes $T_{j' \ldots n}$, $j' \in Zone(A, \alpha)$, we can choose B exactly as in the algorithm sketched at the beginning of this section: Among all incomplete implicit nodes of the sparse suffix tree, select the one that has the minimum depth. Let that node be u and let the character that makes the node incomplete be d. Then $B = label(root, u)d$. The algorithm is illustrated in Fig. 3.

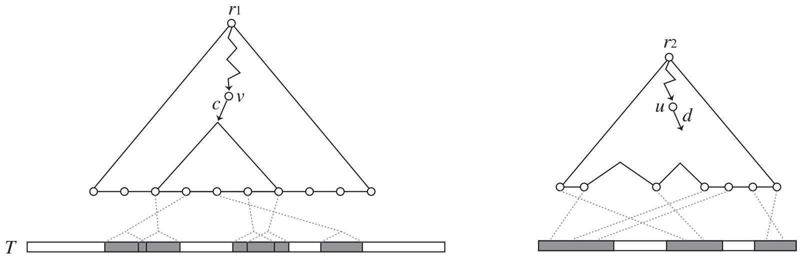

Fig. 3. Illustration of the algorithm to find a missing pair (A, B), where $A = label(r_1, v)c$, $B = label(r_2, u)d$, r_1 is the root of the full suffix tree, and r_2 is the root of the sparse suffix tree corresponding to A.

Theorem 9 *The missing patterns problem on text of length n can be solved in time $O(n^2)$ and space $O(n)$.*

Proof. The correctness of the algorithm should be clear from the above discussion. The time complexity follows from the facts that the size of $\mathcal{P}$ is at most $2n-1$, and for each $A \in \mathcal{P}$ we use $O(n)$ time for constructing the set $Zone(A, \alpha)$ and the corresponding sparse suffix tree; To construct $Zone(A, \alpha)$ in linear time, one should first mark in a bit-vector of length n all suffixes in $Occ(A)$. Then for each marked suffix j, one should mark in some other bit-vectors the starting point $j - \alpha$ and the end point $j + \alpha$ of the influence region. Finally, scanning from left to right one can maintain a counter to know at each text position j' whether it is inside some influence region or not, i.e., whether it should be included in the sparse suffix tree or not. As mentioned earlier, the sparse suffix tree can be obtained from the full suffix tree in $O(n)$ time. Overall, we have $O(n \times n) = O(n^2)$ time. At each phase of the algorithm, we use $O(n)$ space. □

3.3 Improved Algorithm

We will now improve Theorem 9 in the case where α is small. First, we observe that we can select pattern A near the root of the suffix tree because of the following lemma.

Lemma 10 *If $\sigma^k > n$, then there must be a missing pattern of length k.*

Proof. There are at most $n - k + 1$ different k-mers in T. Since σ^k is larger than this, there must be some $X \in \Sigma^k$ that is not a k-mer. □

Hence, we can restrict to the case $k \leq \log_\sigma n$, as there must otherwise be a single pattern solution, which can be found in linear time as explained at the beginning of this section.

Let $\mathcal{P}^{\leq q}$ be a subset of $\mathcal{P}$ such that all strings in $\mathcal{P}^{\leq q}$ are at most of length q. Now, we make the following observation:

Observation 11 *For each suffix j, there are at most $q = \log_\sigma n$ strings $A \in \mathcal{P}^{\leq q}$ such that $j \in Occ(A)$.*

A direct consequence of Observation 11 is that the overall size of sparse suffix trees corresponding to strings $A \in \mathcal{P}^{\leq q}$ is at most $O(\alpha n \log_\sigma n)$; each suffix can belong to at most $(2\alpha + 1)\log_\sigma n$ different sparse suffix trees, and the size of a sparse suffix tree is proportional to the number of suffixes it contains.

Now, we can build the sparse suffix trees incrementally in linear time in their overall size as follows: make a depth-first search (DFS) on the full suffix tree limited to depth $\log_\sigma n$. Let SST_v be the sparse suffix tree corresponding to an internal node v; more formally, SST_v is the sparse suffix tree of suffixes $j \in Zone(A, \alpha)$, where $A = label(root, u)c$, u is the parent of v, and c is the first letter of the edge label from u to v. Let g be the child node of v to which we are proceeding in the DFS search. We make the observation that the sparse suffix tree SST_g corresponding to node g will contain a subset of suffixes represented by SST_v; we can prune SST_v to construct SST_g. To manage the incremental computation efficiently, we show in the next lemma that SST_g can be constructed from SST_v in linear time in the size of SST_v. To make this possible, we need to attach some additional information to the sparse suffix trees: We use *threaded* sparse suffix trees, where the leaves (suffixes) of the tree are linked together in a double linked list in increasing order of the suffix positions, and each leaf has a pointer to the corresponding leaf of the full suffix tree.

Lemma 12 *Let SST_v be the threaded sparse suffix tree corresponding to a node v of the full suffix tree (in the sense defined above). Then, the threaded sparse suffix tree SST_g corresponding to the child g of v can be constructed in linear time in the size of SST_v.*

Proof. The algorithm is as follows. We make a copy of SST_v and prune it (i.e. delete extra leaves) to construct SST_g. Let us simply use SST_v to denote the copy of it. The construction has three phases; (i) we mark all leaves (suffixes) of SST_v that are contained in the subtree of g in the full suffix tree, (ii) we mark all leaves of SST_v whose suffix positions are within α distance from the ones marked at phase (i), and (iii) we delete all unmarked leaves of SST_v to construct SST_g.

Phase (iii) is trivial; as a leaf is deleted (making some constant time local updates to the tree) we redirect the links between suffix positions to retain the threaded structure. In phase (ii) we extend the effect of the suffixes marked in phase (i) by scanning through the double linked list once from first to last and once from last to first. For phase (i) recall that the leaves of SST_v have pointers to the corresponding leaves of the full suffix tree. We reverse these pointers, so that we have pointers from some leaves of the full suffix tree to SST_v. Then we go through the leaves in the subtree of g, and follow the pointers from these leaves marking the corresponding leaves of SST_v. This concludes phase (i).

It is clear that after steps (i),(ii), and (iii), the remaining tree corresponds to SST_g, and the construction time is linear in the size of the tree SST_v. □

After noticing that the threaded version of the full suffix tree is easy to obtain in linear time in its size, we get by induction using Lemma 12 the following result.

Theorem 13 *The missing patterns problem on text of length n can be solved in time $O(\alpha n \log n)$ and space $O(n \log n)$ on a constant alphabet.*

Proof. Lemma 12 states that we use linear time in the size of the parent sparse suffix tree to construct the child sparse suffix tree. Each node of the full suffix tree can have at most σ children, and hence we can use time at most σ times the size of each sparse suffix tree. This gives the claimed time bound on a constant alphabet. The space usage follows from the fact that we need to store at most $\log_\sigma n$ different sparse suffix trees at the same time during the DFS to manage the incremental computation. □

The constant multiplicative factor σ occurring in the proof of the above theorem can be reduced to $\log \sigma$ by organizing the edges of each node of the full suffix tree in a balanced tree; we can build temporary sparse suffix trees for the nodes of each balanced tree. The overall size of the trees grows to $O(\log \sigma \alpha n \log_\sigma n)$, but each tree is scanned through only a constant number of times.

4 Algorithms for Patterns of Same Length

We now concentrate on the special case of the missing patterns problem, mentioned in Section 2.2 with a biological motivation, where the patterns are required to be of the same length. That is, we search for a missing pattern pair

(A, B) such that $|A| = |B| = k$. For this special case we give a slightly faster algorithm than the algorithm of the previous section based on the sparse suffix trees, when α is not too large; we obtain time complexity $O(\alpha n \log\log n)$ on a constant alphabet.

In the sequel, we assume that the alphabet of the strings is $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. This makes the exposition easier, and is not a crucial assumption, since it takes $O(n \log \sigma)$ time to map any other (ordered) alphabet into Σ. This is negligible to the time required by the algorithms for the missing patterns problem.

Let us start with a trivial algorithm, and then proceed with improvements that result into an improved bound. We will first consider checking if there is a missing pair for a fixed length k.

4.1 Trivial Algorithm

For a fixed k, search the text for each possible pattern pair and check whether any occurrences are too close. There are σ^{2k} pattern pairs of correct length. It takes $O(k + n)$ time and $O(k)$ space to check one pair: Run e.g. two Knuth-Morris-Pratt algorithms [9] in parallel. So, the total time requirement to report a possible missing pair for fixed k is $O(\sigma^{2k}(k + n))$. The algorithm only needs $O(k)$ space.

4.2 Simple Algorithm

For each k-mer pair (C, D) of T such that C and D are α-close in T, insert the concatenated string CD into a keyword trie and search the tree for missing pairs. The size of the keyword trie is $O(k\alpha n)$ and the time requirement for inserting the concatenated strings is $O(k\alpha n \log \sigma)$. Checking whether there exists a string P of length $2k$ which is not in the keyword trie, takes $O(k\alpha n)$ time. Such a string $P = AB$ defines a missing pair (A, B), $|A| = |B| = k$.

Alternatively, one can use a bit-table of size σ^{2k}, as there is a bijective mapping from the strings CD to integers $0, 1, \ldots, \sigma^{2k} - 1$. For each string CD its entry in the table can be computed in constant time using the well-known technique of computing the entry of string Yb knowing the entry of string aY (see e.g. [7]). [1] Then, marking the entries corresponding to the α-close k-mer pairs (C, D) takes $O(\sigma^{2k} + n\alpha)$ time. An unmarked entry corresponds to a missing pair.

Analysis. Notice that $\sum_{i=0}^{k-1} \sigma^i < \sigma^k$, since $\sigma \geq 2$. The term σ^k is a multiplicative factor in the previous algorithms, and hence if we run those algorithms for each value $k = 1, 2, \ldots$, until we find a missing pair (A, B) with some length $|A| = |B| = k$, the total time complexity will be at most two times the complexity of the last step.

[1] The entry of aY is $v = a\sigma^{|Y|} + y_1\sigma^{|Y|-1} + y_2\sigma^{|Y|-2} + \cdots + y_{|Y|}$. The entry of Yb can be computed in constant time, as it is $\sigma(v - a\sigma^{|Y|-1}) + b$.

Before running any of the algorithms, we can first check in $O(n)$ time whether there is a single missing pattern using the suffix tree approach. If such pattern is not found, we know that $k \leq \log_\sigma n$ (due to the Lemma 10). Equivalently $\sigma^k \leq n$, which simplifies the bounds.

We notice also that we can bound σ^k with a function of n and α: If k is the smallest value such that $\sigma^{2k} > n\alpha$, then there must be a missing pair (A, B) such that $|A| = |B| = k$. Then equation $\sigma^{2k-2} \leq n\alpha$ gives an estimate $\sigma^{2k} \leq \sigma^2 n\alpha$.

Plugging these bounds to the complexity of the bit-table algorithm we get $O(\sigma^{2k} + n\alpha) = O(\sigma^2 n\alpha + n\alpha)$. This bound is for fixed k. We can search for the correct value of k using binary search among $1, 2, \ldots, \frac{\log_\sigma(n\alpha)}{2} + 1$. The overall work becomes $O(\sigma^2 n\alpha + n\alpha \log(\frac{\log_\sigma(n\alpha)}{2} + 1)) = O(\sigma^2 n\alpha + n\alpha \log\log_\sigma n)$, which gives the following result.

Theorem 14 *The missing patterns problem on a text of length n for patterns of the same length can be solved in $O(\alpha n \log\log n)$ time and $O(n\alpha)$ space on a constant alphabet.*

Notice also that the algorithms for the general case can be used to solve this restricted case. In fact, the result of Theorem 13 already gives an $O(\alpha n \log n)$ time solution with better constant factors on the alphabet size. Moreover, that algorithm uses considerably less space than the bit-table algorithm.

5 Experiments

We have run some preliminary tests with the baker's yeast (*Saccharomyces cerevisiae*) genome using the bit-table version of the simple algorithm described in Section 4.2. We set the distance α to a realistic value 5000 and searched for shortest missing pattern pairs of the same length k. There were solutions for $k = 8$ (i.e. both patterns of the pair are of length 8), in fact there were over 16 million such pairs.

Ultimately our aim is to find short missing pairs of patterns for the human genome in order to construct good adapter primers for biomedical applications. The test results with the yeast genome suggest that there most likely are missing pattern pairs for the human genome that are short enough to make the approach attractive from the biochemical point of view. The human genome is about 250 times larger than the baker's yeast genome, the size which size is about 12Mb, but on the other hand increasing k by 2 increases the number of pattern pairs 256 times. In addition, the human genome has a lot of repetitive elements. Processing a text as large as the human genome will be challenging but, in our opinion, feasible. Moreover, the time complexity of the simple algorithm depends on σ^k, which is in practice (at least in our experiment with yeast) much smaller than the theoretical bounds we derived for it.

6 Acknowledgements

The authors would like to thank Mr. K. Kataja and Dr. R. Satokari from VTT Biotechnology. Mr. Kataja was the first one to bring the adapter primer selection problem to our attention and Dr. Satokari has advised us on the biotechnological issues. The discussions with Juha Kärkkäinen from the University of Helsinki and Jens Stoye, Sven Rahmann, and Sebastian Böcker from Bielefeld University led to better understanding of the problem. Especially we wish to thank Matthias Steinrücken from Bielefeld University, as he found a fundamental fault in one of the algorithms we had in an earlier version of this paper.

References

1. A. Amir, A. Apostolico, and M. Lewenstein. Inverse Pattern Matching. *J. Algorithms*, 24(2):325–339,1997.
2. A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.
3. B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell, fourth edition.* Garland Science, 2002.
4. A. Apostolico. Pattern discovery and the algorithmics of surprise. *Artificial Intelligence and Heuristic Methods for Bioinformatics*, pp. 111–127, 2003.
5. L. Gąsieniec, P. Indyk, P. Krysta. External Inverse Pattern Matching. In *Proc. Combinatorial Pattern Matching 97 (CPM'97)*, Springer-Verlag LNCS 1264, pp. 90–101, 1997.
6. D. Gusfield. *Algorithms on strings, trees and sequences: Computer science and computational biology.* Cambridge University Press, 1997.
7. R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.
8. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. Second Annual International Computing and Combinatorics Conference (COCOON '96)*, Springer-Verlag LNCS 1090, pp. 219–230, 1996.
9. D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
10. J. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. *Information and Computation*, 185(1):41–55, 2003.
11. E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23, pp. 262–272, 1976.
12. P. Nicodème and J.-M. Steyaert. Selecting optimal oligonucleotide primers for multiplex PCR. In *Proc. of the 5th International Conference on Intelligent Systems for Molecular Biology (ISMB'97)*, pp. 210–213, 1997.
13. A. Shinohara, M. Takeda, S. Arikawa, M. Hirao, H. Hoshino, and S. Inenaga. Finding Best Patterns Practically. In *Progress in Discovery Science (Final Report of the Japanese Discovery Science)*, Springer-Verlag LNAI 2281, pp. 307–317, 2002.
14. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
15. J. Wang, B. Shapiro, and D. Shasha. *Pattern Discovery in Biomolecular Data.*, Oxford University Press, 1999.
16. P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Annual Symposium on Switching and Automata Theory*, pp. 1–11, 1973.