Indexing a sequence for mapping reads with a single mismatch

Maxime Crochemore, Alessio Langiu, M. Sohel Rahman

Indexing a sequence for mapping reads with a single mismatch

Maxime Crochemore[1,2,†], **Alessio Langiu**[1,3,†], **M. Sohel Rahman**[1,4,‡]

[1]*Department of Informatics, King's College London, London WC2R 2LS, UK*
[2]*Laboratoire d'informatique Gaspard-Monge, Université Paris-Est, Paris, France*
[3]*Department of Mathematics and Informatics, University of Palermo, Palermo, Italy*
[4]*AlEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh*
{Maxime.Crochemore, Alessio.Langiu, Sohel.Rahman}@kcl.ac.uk

Mapping reads against a genome sequence is an interesting and useful problem in Computational Molecular Biology and Bioinformatics. In this paper, we focus on the problem of indexing a sequence for mapping reads with a single mismatch. We first focus on a simpler problem where the length of the pattern is given beforehand during the data structure construction. This version of the problem is interesting in its own right in the context of the Next Generation Sequencing (NGS). In the sequel we show how to solve the more general problem. In both cases, our algorithm can construct an efficient data structure in $O(n \log^{1+\varepsilon} n)$ time and space and can answer subsequent queries in $O(m \log\log n + \mathcal{K})$ time. Here, n is the length of the sequence, m is the length of the read, $0 < \varepsilon < 1$ and $\mathcal{K}$ is the optimal output size.

Key words: algorithms, genome sequence, indexing, mapping reads, mismatch, pattern matching.

1. Introduction

In the classical string matching problem we are given a text of length n and a pattern of length m and we need to answer the query whether the pattern exists in the text (*existence query*) as a factor or substring and further to provide the start (or equivalently end) positions of the corresponding occurrences (*occurrence query*). Using the famous KMP algorithm, due to Knuth, Morris and Pratt [36], the classical string matching problem can be solved in optimal $O(n + m)$

†Supported by EPSRC grand EP/J017108/1.
‡Supported by a Commonwealth Fellowship.

Philosophical Transactions A 1–24; doi: 10.1098/rspa.00000000

time. For a review on KMP algorithm as well as various other (exact) string matching algorithms the readers are referred to [11].

In many application settings, a number of patterns are searched in a particular text which gives rise to the *indexing version* of the problem. In this version, the text is given beforehand for preprocessing with a goal to construct an index data structure. Subsequently, a number of patterns are queried against the text. Clearly, this indexing variant can also be solved using KMP algorithm; however, if we have k patterns (each of length m), the running time will be $O(k(n+m))$. Since all the queries are done against a single text, it is only natural to construct an index data structure for it once, preferably in $O(n)$ time and then answer the subsequent queries in $O(m)$ time each, which gives a total of $O(km+n)$ running time. And, indeed there exist efficient data structures (e.g., suffix trees [19, 43, 54, 55] and suffix arrays [2, 34, 37]) that can be constructed in $O(n)$ time and are capable of answering an existence query in $O(m)$ time and an occurrence query in $O(m+|\text{Occ}|)$ time, where Occ is the set of output.

The string matching problem has tremendous applications in different branches of science including, but not limited to, Computational Molecular Biology, Bioinformatics, Computer Vision, Information retrieval, Computational Musicology, Data Mining, Network Security etc. However, in many, if not most, practical settings, instead of the exact matching some approximate matching schemes become more relevant and useful. The requirement of such approximate matching schemes is more prominent in Computational Molecular Biology and Bioinformatics as discussed below. Notably, in the context of Computational Biology, the string matching problem is essential for mapping *reads* (i.e., patterns) to a reference *biological sequence* (i.e., a text).

While an increasing number of the biology labs are using dedicated high-throughput equipments to produce many DNA sequences on a daily basis, the need for automatic annotation and content analysis is greater everyday. Unfortunately, even with the tremendous advancements of the current state of the art technology, the quality of the automatically obtained sequences is sometimes questionable due to the intrinsic limitations of the equipments (for instance, [44] evaluated the substitution error rate of the control genome PhiX174 to be in the range 0.11% - 0.28% excluding uncalled bases: all positions were affected by substitution errors). Moreover, the re-sequencing methods are affected by the natural polymorphism that can be observed between individual samples (e.g., a Single Nucleotide Polymorphism, that is an isolated mutation, can either stop the translation of a mRNA into a protein sequence, or create a binding site for a protein complex that will prevent the complete formation of the functional protein [10, 38]). Analysing such uncertain sequences is therefore much more complicated than the traditional pattern matching problem. This gives the computer scientists, in particular the stringology researchers, the challenge to solve the string matching problem where in the given text or pattern some positions may be uncertain in some sense. To capture this phenomenon of uncertainty, the idea of mismatches and of gaps was introduced. Additionally, a popular and useful framework of don't care pattern matching was introduced under this approximate matching scheme, where

a pattern and/or text may contain don't care characters that match with any character in the underlying alphabet.

The use of don't care paradigm to capture the approximate pattern matching scenario mentioned above is not new. It was introduced and solved efficiently using convolutional methods in [23]. Slightly tighter solutions have been presented in [12, 14, 32, 33]. There exist some solutions avoiding the convolution method as well [8, 47, 49]. A number of solutions exists in the literature that consider the problem of text indexing with don't cares [13, 39, 52, 53]. Notably, in the literature, the don't cares are also referred to as wildcards. Some work has also been done on a generalised model of the don't care paradigm known as the degenerate or indeterminate string model in the literature [28, 29, 50, 51]; in this model, some positions of a string may contain more than one letters and don't care is essentially modelled as a position that contains all the letters of the alphabet.

Another popular approximate model in the literature is where some $k \geq 0$ mismatches are allowed while doing the pattern matching. There exists a number of results on this problem [3, 6, 7, 18, 25, 26, 40]. However, from indexing point of view the results are only few (see for instance [5, 13, 15]. Notably, the k mismatch model can be represented/captured in the don't care model by assuming k don't care characters at all possible permutations of the positions in the pattern (and/or the text).

In this paper, we focus on the indexed version of the pattern matching problem with restricted number of mismatches and/or gaps. In particular, we are interested in the pattern matching problem when at most one mismatch or gap is allowed. Here gap refers to consecutive mismatches. To the best of our knowledge the only work that deals with this problem directly is the work of Amir et al. [5]. We will give a brief review of the work of [5] and compare their results with ours in a subsequent section.

The contribution of this current paper is twofold. First we attack a restricted version of the problem in hand where we assume that the size of the pattern is fixed, i.e., we are given the size of the pattern to be queried against our data structure beforehand. As discussed below, this particular version of the problem has strong motivation from Computational Biology and Bioinformatics, especially in the context of the Next Generation Sequencing. In the sequel we consider the general version of the problem where this restriction is lifted. The solution we propose makes use of suffix arrays and range search data structures borrowed from Computational Geometry literature. In particular, in order to present an efficient data structure for our problem, we utilise a reduction from our problem at hand to the range search problem in geometry. As will be further discussed in later sections, similar reduction has also been employed in [5] to solve this particular problem and in different other papers (e.g., [17, 30, 31, 41]) to solve some other interesting problems. Note however that the reduction itself is not enough to get a good solution. As will be clear later, we need to do some non-trivial work based on some useful observations and lemmas to achieve an efficient running time for the queries.

The motivation of our work comes both from the field of Stringology as well as from that of Computational Biology and Bioinformatics. Firstly, a solution to the approximate pattern matching with a single mismatch would be useful as a theoretical advancement in the context of the general variant where k mismatches are considered. Secondly, approximate pattern matching problem with at most a single mismatch is a useful problem on its own right. This is specially true in Computational Molecular Biology, where with the advent of new state of the art technologies, the chance of experimental mistakes has become much lower than it was before. Also, the existence of single mismatches, called SNPs, occurring at consecutive positions is rare (see [45, Table 1] for experimental results carried out on 10 individual human genomes from the 1000 Genome Project). Thirdly, because of the recent high throughput sequencing technologies we are particularly interested to provide a fast solution to the restricted version of the problem where the pattern size is fixed. In the so-called Next Generation Sequencing (NGS) technologies, millions of short reads are generated. Usually, the first region of these reads, called the seeds, contains almost no sequencing error. The seed region is followed by a region having very small possibilities of error and hence from mapping point of view, only a single mismatch or a gap (i.e., consecutive mismatches) are expected. Now, in some NGS platforms, the generated sequences, which are usually several millions in number, are of the same size; this size depends on the technologies used (e.g., Illumina etc.). When sequencing platforms generate variable-length reads, they can be reduced to prefix seeds of fixed length. As a result, the fixed pattern length version of the problem is of particular interest in this specific scenario.

The rest of the paper is organised as follows. After the definitions and problem setting in Section 2, we give in Section 3 the general idea of the solution. It serves as a guideline for the development of algorithms. Section 4 solves a simpler problem, while Section 5 describes a solution for the complete problem. We put some relevant discussion and conclusion in Sections 6 and 7.

2. Preliminaries

A string is a sequence of zero or more symbols from an alphabet Σ. A string $\mathcal{T}$ of length n is denoted by $\mathcal{T}[1\,..\,n] = \mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$, where $\mathcal{T}_i \in \Sigma$ for $1 \le i \le n$. The length of $\mathcal{T}$ is denoted by $|\mathcal{T}| = n$. The string $\overleftarrow{\mathcal{T}}$ denotes the reverse of the string $\mathcal{T}$, i.e., $\overleftarrow{\mathcal{T}} = \mathcal{T}_n\mathcal{T}_{n-1}\ldots\mathcal{T}_1$.

A string w is a factor of $\mathcal{T}$ if $\mathcal{T} = uwv$ for $u, v \in \Sigma^*$; in this case, the string w occurs at position $|u|+1$ in $\mathcal{T}$. The factor w is denoted by $\mathcal{T}[|u|+1\,..\,|u|+|w|]$. A k-factor is a factor of length k. A prefix (or suffix) of $\mathcal{T}$ is a factor $\mathcal{T}[x\,..\,y]$ such that $x = 1$ $(y = n)$, $1 \le y \le n$ $(1 \le x \le n)$. We define the ith prefix to be the prefix ending at position i, i.e. $\mathcal{T}[1\,..\,i]$, $1 \le i \le n$. Dually, the ith suffix is the suffix starting at position i, i.e. $\mathcal{T}[i\,..\,n]$, $1 \le i \le n$.

The *Hamming distance* between two strings of equal length is the number of positions at which the corresponding letters are different. More formally, the Hamming distance between u and v is $\delta(u, v) = |\{i \mid u_i \ne v_i, 1 \le i \le |u| = |v|\}|$. Given two equal length strings $u, v \in \Sigma^*$, u is

said to match v (or equivalently v is said to match u) with at most k mismatches if $\delta(u, v) \leq k$. Essentially, the exact match is characterised by a zero Hamming distance.

Given a text $\mathcal{T}$ of length n and a pattern $\mathcal{P}$ of length m such that $m \leq n$, $\mathcal{P}$ is said to occur in $\mathcal{T}$ at position i (i.e., exact match) if and only if $\mathcal{P} = \mathcal{T}[i\,..\,i+m-1])$. The position i is said to be an occurrence of $\mathcal{P}$ in $\mathcal{T}$. We denote by Occ_P^T the set of occurrences of $\mathcal{P}$ in $\mathcal{T}$. As an extension, $\mathcal{P}$ is said to occur in $\mathcal{T}$ at position i with at most k mismatches if and only if $\delta(\mathcal{P}, \mathcal{T}[i\,..\,i+m-1]) \leq k$. We use $\mathrm{Occ}_P^T|_{\leq k}$ to denote the set of positions where $\mathcal{P}$ matches $\mathcal{T}$ with at most k mismatches. Similarly, we use $\mathrm{Occ}_P^T|_{=k}$ to denote the set of positions where $\mathcal{P}$ matches $\mathcal{T}$ with exactly k mismatches. Clearly, our interest is in calculating $\mathrm{Occ}_P^T|_{\leq 1}$.

In traditional full-text indexing problems one of the basic data structures used is the suffix array data structure. Others are suffix trees and suffix automata. In our indexing problem we make use of the suffix array data structure. A complete description of a suffix array is beyond the scope of this paper, and can be found in any textbook on stringology (e.g., [16, 27, 46]). Here we give a very concise definition of a suffix array. The suffix array $SA_{\mathcal{T}}[1\,..\,n]$ of a text $\mathcal{T}$ is an array of integers $j \in [1\,..\,n]$ such that $SA_{\mathcal{T}}[i] = j$ if, and only if, $\mathcal{T}[j\,..\,n]$ is the i-th suffix of $\mathcal{T}$ in (ascending) lexicographic order. Suffix arrays were first introduced in [42], where an $O(n \log n)$ construction algorithm and $O(m + \log n + |\mathrm{Occ}_{\mathcal{P}}^{\mathcal{T}}|)$ query time were presented. Later, linear time construction algorithms for space efficient suffix arrays were presented [34, 35, 37, 48]. The query time is also improved to optimal $O(m + |\mathrm{Occ}_{\mathcal{P}}^{\mathcal{T}}|)$ in [1] with the help of another array essentially storing the lengths of longest common prefixes between lexicographically consecutive suffixes. We recall that, the result of a query for a pattern $\mathcal{P}$ on a suffix array $SA_{\mathcal{T}}$ of $\mathcal{T}$, is given in the form of an interval $[s\,..\,e]$ such that $\mathrm{Occ}_{\mathcal{P}}^{\mathcal{T}} = \{SA_{\mathcal{T}}[s], SA_{\mathcal{T}}[s+1], \ldots, SA_{\mathcal{T}}[e]\}$. In this case, the interval $[s\,..\,e]$ is denoted by $Int_{\mathcal{P}}^{\mathcal{T}}$.

We remark that the query time of suffix array (and similar other data structures) always contains a hidden $O(\log \Sigma)$ factor. However, since in most of the cases the size of the underlying alphabet Σ is a constant, the trend in the literature is to omit the $O(\log \Sigma)$ factor from the running times. Indeed, this is all the more applicable in our case because we are focused on biological sequences where usually the underlying alphabet size is a small constant (e.g., 4 for DNA/RNA sequences and 20 for protein/amino acid sequences). Finally, we note that there are several linear time suffix array construction methods that works with integer alphabet as well (e.g., [35, 37]).

3. The Underlying Idea for a Solution

In this section, we discuss how we can efficiently compute $\mathrm{Occ}_{\mathcal{P}}^{\mathcal{T}}|_{\leq 1}$, given $\mathcal{T}$ and $\mathcal{P}$. In other words, we will in fact discuss a solution of our problem albeit not from the indexing point of view. In the sequel we will be using the underlying idea to construct the index data structures of our interest. Notably, the same idea was employed by Amir et al. in [5] to provide an indexing

Algorithm 1 Computing $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1}$

1: Set $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1} = \emptyset$
2: **for** $j \in [1\,..\,m]$ **do**
3: Compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$
4: Compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$
5: **for** $i \in \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$ **do**
6: $i = i - j$
7: **end for**
8: Set $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^{j}_{\leq 1} = \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \bigcap \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$
9: Set $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1} = \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1} \bigcup \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^{j}_{\leq 1}$
10: **end for**
11: **return** $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1}$

solution for the problem. We will discuss the problems of the solution in [5] and highlight the differences between their solution and the solution presented in this paper in a later section.

Suppose we know the position of the mismatch and assume that the position is j. In other words, we suppose that the mismatch is only allowed with $\mathcal{P}_j$. Let us call $\mathcal{P}|^j = \mathcal{P}[1\,..\,j-1] * \mathcal{P}[j+1\,..\,m]$ the pattern with a $*$ in position j, and let us use $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^{j}_{\leq 1}$ to denote the corresponding set of occurrences over $\mathcal{T}$. Then, we may proceed as follows. We compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$ and $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$. Then, clearly, our desired set of occurrences $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^{j}_{\leq 1}$ can be computed as follows:

$$\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^{j}_{\leq 1} = \{i \mid i \in \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \text{ and } (i+j) \in \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[j+1..m]}\}.$$

Now to lift the restriction we can simply run a loop on all possible values of j. This simple idea works perfectly and the steps are formally presented in Algorithm 1. An example of such algorithm is reported in Table 1 and Table 2. However, transforming this idea to handle the indexing version is a non-trivial task.

4. The Index Data Structure

In this section, we focus on constructing an index data structure for our problem. We however first consider a restricted variant of the original problem. In particular, for the time being, we assume that we are given the pattern-length m which we use during the index data structure construction. In other words, the index constructed will be m-specific, i.e., it will be able to handle patterns of length m only. Recall from Section 1 that this particular variant of the problem is of particular interest especially in the context of Next Generation Sequencing. The general version of the problem will be handled in a later section.

We basically extend the idea of Algorithm 1. We maintain two suffix array data structures $SA_{\mathcal{T}}$ and $SA_{\overleftarrow{\mathcal{T}}}$. We use $SA_{\mathcal{T}}$ to find the occurrences of $\mathcal{P}[1\,..\,j-1]$. We can find the occurrences

of $\mathcal{P}[j+1\,..\,m]$ using $SA_{\mathcal{T}}$ as well. But we need to take a different approach because we have to "align" the occurrences of $\mathcal{P}[1\,..\,j-1]$ (line 5) with the occurrences of $\mathcal{P}[j+1\,..\,m]$ so that we can compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}|^j_{\leq 1}}$ by intersecting (line 8) them just as is done in Algorithm 1. However it is not as straightforward as Algorithm 1 because our aim is to maintain an index rather than finding a match for a particular pattern. We use the following trick, which has been applied to solve this and different other problems in the literature before (see for example [5, 17, 31]).

Our approach is as follows. We use the suffix array of the reverse string of $\mathcal{T}$, i.e. $SA_{\overleftarrow{\mathcal{T}}}$, to find the occurrences of $\overleftarrow{\mathcal{P}[j+1\,..\,m]}$. By doing so, in fact, we get the end positions of the occurrences of $\overleftarrow{\mathcal{P}[j+1\,..\,m]}$ in $\mathcal{T}$. However we still have to do a bit more "shifting" for the intersection to work because from $SA_{\mathcal{T}}$, we get the start positions of the occurrences of $\mathcal{P}[1\,..\,j-1]$. This is where the information of a fixed pattern-length, i.e. m, comes handy. To achieve the "shifting" effect automatically, we appropriately arrange the entries of $SA_{\overleftarrow{\mathcal{T}}}$ as follows. For all $1 \leq i \leq n$,

j	1	2	3	4	
$\mathcal{P}\|^j$	`*gat`	`c*at`	`cg*t`	`cga*`	
$\mathcal{P}[1\,..\,j-1]$	–	`c`	`cg`	`cga`	
$\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$	$\{1\,..\,19\}$	$\{1,3,8,12,16\}$	$\{1,12,16\}$	$\{12,16\}$	line 3
$\mathcal{P}[j+1\,..\,m]$	`gat`	`at`	`t`	–	
$\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$	$\{5,13\}$	$\{6,10,14\}$	$\{4,7,11,15\}$	$\{1\,..\,19\}$	line 4
$\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$	$\{4,12\}$	$\{4,8,12\}$	$\{1,4,8,12\}$	$\{-3\,..\,17\}$	line 5
$\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}\|^j_{\leq 1}}$	$\{4,12\}$	$\{8,12\}$	$\{1,12\}$	$\{12,16\}$	line 8
$\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}\|_{\leq 1}}$	$\{4,12\}$	$\{4,8,12\}$	$\{1,4,8,12\}$	**{1,4,8,12,16}**	line 9

Table 1. An example of the steps of Algorithm 1 for the text $\mathcal{T}[1\,..\,19] =$ `cgctgatcaatcgatcgag` and the pattern $\mathcal{P}[1\,..\,4] =$ `cgat`. The output set $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}|_{\leq 1}}$ is reported in bold font and a dash is used in place of the empty word.

i	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
$\mathcal{T}_i$	c	g	c	t	g	a	t	c	a	a	t	c	g	a	t	c	g	a	g
$\mathcal{P}$ = cgat												–	–	–	–				
$\mathcal{P}\|^1$ = *gat				*	–	–	–												
$\mathcal{P}\|^2$ = c*at								–	*	–	–								
$\mathcal{P}\|^3$ = cg*t	–	–	*	–															
$\mathcal{P}\|^4$ = cga*																–	–	–	*

Table 2. A graphical representation of matching of the pattern $\mathcal{P}[1\,..\,4] =$ `cgat` and its one error variants $\mathcal{P}|^j$, $1 \leq j \leq |\mathcal{P}|$, over the text $\mathcal{T}[1\,..\,19] =$ `cgctgatcaatcgatcgag`. A dash stands for a match and a star stands for a mismatching letter. $\mathcal{P}|^j$ matches in position 12 are not reported.

if originally $SA_{\overleftarrow{\mathcal{T}}}[i] = j$, we assign $SA'_{\overleftarrow{\mathcal{T}}}[i] = n - (j-1) - (m-1) = n - j - m + 2$. It is easy to see that this will effectively transform the position of each of the occurrences of $\mathcal{P}[j+1\,..\,m]$ to the appropriate position that will facilitate the intersection. How will be clarified later, the candidate starting positions of an occurrence of a pattern of length m over a text of length n are those in the range $[1\,..\,n-m+1]$. Obviously, any solution working for a generic range $[1\,..\,n]$ is applicable as well to a subrange $[1\,..\,n-m+1]$.

Now, it remains to show how we can perform the intersection (line 8) efficiently in the context of indexing. Clearly we now have two arrays, namely, $SA_{\mathcal{T}}[1\,..\,n]$ and $SA_{\overleftarrow{\mathcal{T}}}[1\,..\,n]$. And we also have two intervals, namely, $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$ and $Int^{\overleftarrow{\mathcal{T}}}_{\overleftarrow{\mathcal{P}[j+1..m]}}$. And, now our problem is to find the intersection of the positions within these two intervals. This problem is a known problem in geometry and is called the *Range Set Intersection* Problem.

PROBLEM 1. *[Problem RSI] Let $V[1\,..\,n]$ and $W[1\,..\,n]$ be two permutations of $[1\,..\,n]$. Preprocess V and W to answer the following form of queries.*
Query: Find the intersection of the elements of $V[i\,..\,j]$ and $W[k\,..\,\ell]$, $1 \le i \le j \le n, 1 \le k \le \ell \le n$.

In order to solve the above problem we reduce it to the well-studied *Range Search Problem* on a Grid.

PROBLEM 2. *[Problem RSG] Let A be a set of n points on the grid $[0\,..\,U] \times [0\,..\,U]$. Preprocess A to answer the following form of queries.*
Query: Given a query rectangle $q \equiv (a,b) \times (c,d)$ find the set of points of A contained in q.

We can see that Problem RSI is just a different formulation of the Problem RSG. This can be realised as follows. We set $U = n$. Since V and W in Problem RSI are permutations of $[1\,..\,n]$, every number in $[1\,..\,n]$ appears precisely once in each of them. We define the coordinates of every number $i \in [1\,..\,n]$ to be (x, y), where $V[x] = W[y] = i$. Thus we get at most n points on the grid $[1\,..\,n] \times [1\,..\,n]$, i.e., the array A of Problem RSG. The query rectangle q is deduced from the two intervals $[i\,..\,j]$ and $[k\,..\,\ell]$ as follows: $q \equiv (i,k) \times (j,\ell)$. It is easy to verify that the above reduction is correct and hence we can solve Problem RSI using the solution of Problem RSG. So, this completes our description for constructing the index data structure for our problem. Algorithm 2 formally states the steps to build our data structure while Table 3 and Table 4 show an example.

4.1. Analysis

Let us analyse the running time of Algorithm 2, i.e., the data structure construction. The computational effort spent for lines 1, 2 and the `for` loop at line 3 is $O(n)$. In line 7, we construct the set A of points in the grid $[1\,..\,n] \times [1\,..\,n]$ on which we will apply the range search. This step can also be done in $O(n)$ as follows. We construct $SA^{-1}_{\mathcal{T}}$ such that $SA^{-1}_{\mathcal{T}}[SA_{\mathcal{T}}[i]] = i$. Similarly, we can construct $SA'^{-1}_{\overleftarrow{\mathcal{T}}}$. Then, it is easy to construct A in $O(n)$. A detail is that in our case

Algorithm 2 Building the index data structure for the fixed-length pattern case

1: Build a suffix array $SA_{\mathcal{T}}$ of $\mathcal{T}$.
2: Build a suffix array $SA_{\overleftarrow{\mathcal{T}}}$ of $\overleftarrow{\mathcal{T}}$.
3: **for** $i = 1$ *to* n **do**
4: Set $j = SA_{\overleftarrow{\mathcal{T}}}[i]$
5: Set $SA'_{\overleftarrow{\mathcal{T}}}[i] = n - j - m + 2$
6: **end for**
7: **for** $i = 1$ to n **do**
8: **if** there exists (x, y) such that $SA_{\mathcal{T}}[x] = SA'_{\overleftarrow{\mathcal{T}}}[y] = i$ **then**
9: $A[i] = (x, y)$
10: **end if**
11: **end for**
12: Preprocess A for Range Search on the grid $[1\,..\,n] \times [1\,..\,n]$.

there may exist i, $1 \le i \le n$, such that $SA'^{-1}_{\overleftarrow{\mathcal{T}}}[j] \neq i$ for all $1 \le j \le n$. This is because $SA'^{-1}_{\overleftarrow{\mathcal{T}}}$ is a permutation of $[-m+2\,..\,n-m+1]$ instead of $[1\,..\,n]$. Now, it is easy to observe that any $i \in SA'^{-1}_{\overleftarrow{\mathcal{T}}}$ such that $i > n$ or $i < 1$ is irrelevant in the context of our search. So we ignore any such $i \in SA'^{-1}_{\overleftarrow{\mathcal{T}}}$ while creating the set A. After A is constructed we perform line 12. Notice that from now on the shifted array $SA'_{\overleftarrow{\mathcal{T}}}$ as well as the inverted arrays SA^{-1} and SA'^{-1} are to be dismissed since they are not involved in the query process.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
$\mathcal{T}_i$	c	g	c	t	g	a	t	c	a	a	t	c	g	a	t	c	g	a	g	
$SA_{\mathcal{T}}$	9	18	6	14	10	8	16	12	1	3	19	17	5	13	2	7	15	11	4	line 1
$\overleftarrow{\mathcal{T}}_i$	g	a	g	c	t	a	g	c	t	a	a	c	t	a	g	t	c	g	c	
$SA_{\overleftarrow{\mathcal{T}}}$	10	11	6	2	14	19	17	8	4	12	1	18	7	3	15	9	5	13	16	line 2
$SA'_{\overleftarrow{\mathcal{T}}}$	7	6	11	15	3	-2	0	9	13	5	16	-1	10	14	2	8	12	4	1	line 3

i	1	2	3	4	5	6	7	8	
$A[i]$	(9, 19)	(15, 15)	(10, 5)	(19, 18)	(13, 10)	(3, 2)	(16, 1)	(6, 16)	line 7

i	9	10	11	12	13	14	15	16	
$A[i]$	(1, 8)	(5, 13)	(18, 3)	(8, 17)	(14, 9)	(4, 14)	(17, 4)	(7, 11)	line 7

Table 3. An example of the values computed by Algorithm 2 for the text $\mathcal{T}[1\,..\,19] =$ `cgctgatcaatcgatcgag`.

19								1										
18																	4	
17							12											
16						8												
15														2				
14				14														
13					10													
12																		
11							16											
10													5					
9														13				
8	9																	
7																		
6																		
5									3									
4																15		
3																	11	
2			6															
1															7			
$A[i]$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Table 4. A graphical representation of the index for Range Search Query built by Agorithm 2 in line 12 for the array $A[i] = (x, y), 1 \leq i \leq 16$, of Table 3.

There has been significant research work on Problem RSG. For example, we can use the data structure of Alstrup et al. [4]. This data structure can answer the query of Problem RSG in $O(\log \log n + k)$ time where k is the number of points contained in the query rectangle q. The data structure can be constructed in $O(n \log^{1+\varepsilon} n)$ time and space, for any constant $0 < \varepsilon < 1$. So, line 12 runs in $O(n \log^{1+\varepsilon} n)$ time and space, for any constant $0 < \varepsilon < 1$. Therefore, the overall time and space complexity of the index remains $O(n \log^{1+\varepsilon} n)$.

4.2. Query processing

Now, let us focus on the query processing. Again, for the sake of ease, let us suppose that we know the position of the mismatch and assume that the position is j. Then the query can be answered as follows. We first compute $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \equiv [s_\ell|^j \,..\, e_\ell|^j]$ using $SA_{\mathcal{T}}$. Then we compute $Int^{\overleftarrow{\mathcal{T}}}_{\overleftarrow{\mathcal{P}[j+1..m]}} \equiv [s_r|^j \,..\, e_r|^j]$ using the original (i.e., the unshifted) $SA_{\overleftarrow{\mathcal{T}}}$. Now, we find all the points in A that are inside the rectangle $q_j \equiv (s_\ell|^j, s_r|^j) \times (e_\ell|^j, e_r|^j)$. Let $B|^j$ is the set of those points. Then it is easy to verify that $\text{Occ}^{\mathcal{T}}_{\mathcal{P}|^j_{\leq 1}} = B|^j$.

Algorithm 3 Processing *occurrence query* with a single mismatch

1: $B = \emptyset$
2: **for** $j = 1$ *to* m **do**
3: Compute $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \equiv [s_\ell|^j \,..\, e_\ell|^j]$ using $SA_{\mathcal{T}}$.
4: Compute $Int^{\overleftarrow{\mathcal{T}}}_{\overleftarrow{\mathcal{P}[j+1..m]}} \equiv [s_r|^j \,..\, e_r|^j]$ using $SA_{\overleftarrow{\mathcal{T}}}$.
5: Set $B|^j = \{(x, y) \mid (x, y) \in A$ and (x, y) is contained in $q \equiv (s_\ell|^j, s_r|^j) \times (e_\ell|^j, e_r|^j)\}$
6: $B = B \bigcup B|^j$
7: **end for**
8: **return** $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1} = B$

Now, we can easily lift the restriction that j is the position of the mismatch. We simply, compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^j_{\leq 1}$, i.e., $B|^j$ for all values of j and compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1} = \bigcup_{j=1}^{m} B|^j$. The steps are formally presented in the form of Algorithm 3. Table 5 and Table 6 show an example of such algorithm.

The running time of the query processing is deduced as follows. Line 3 and line 4 can be done in $O(m)$ time. And line 5, i.e., the range search query, can be done in $O(\log\log n + K)$ time, where K is the number of points in the query rectangle. Then, the algorithm mostly consists of a loop. So, a straightforward analysis of Algorithm 3 leads to a running time of $O(m^2 + m\log\log n + \mathcal{K})$. Here, we assume that $\mathcal{K}$ is the size of the output returned by the algorithm. We will focus on $\mathcal{K}$ shortly. However, we can do far better than $O(m^2 + m\log\log n + \mathcal{K})$ as follows. Note that we can compute $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$ incrementally as we increment j from 1 to m. This means we can get all the intervals from $SA_{\mathcal{T}}$, namely, $[s_\ell|^j \,..\, e_\ell|^j], 1 \leq j < m$ spending $O(m)$ time in total. Similarly, we can get all the intervals from $SA_{\overleftarrow{\mathcal{T}}}$, namely $[s_r|^j \,..\, e_r|^j]$, $1 \leq j \leq m$, spending $O(m)$ time in total. Hence, we can compute all the intervals first and store them with a little book-keeping to implement line 5 for all j, $1 \leq j \leq m$, afterwards. This will give a much better running time of $O(m\log\log n + \mathcal{K})$.

Let us notice that the pattern matching part of our solution, namely, lines 3 and 4, where the intervals are provided by suffix arrays, is alphabet dependent while the geometrical part, that is the rectangle query in line 5, is not. Hence, in case of large (integer) alphabet the query time is dominated by the pattern matching time and the resulting query time is $O(m\log n + \mathcal{K})$.

4.3. Discussion on $\mathcal{K}$

Finally, a discussion on the value of $\mathcal{K}$ is in order. Since we are computing the occurrences with at least 1 mismatch, the occurrences of exact matches will also be reported. And in our algorithm when we compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^j_{\leq 1}$ for a particular value of j, it will also contain the exact occurrences. In other words, for all values of j, we have $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}} \subseteq \mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^j_{\leq 1}$. So, every exact occurrence will be reported m times. To make the value of $\mathcal{K}$ optimal we need to employ some further tricks. First we need the following easy lemma.

LEMMA 1. *Suppose we are given a suffix array $SA_{\mathcal{T}}[1\,..\,n]$ of a text $\mathcal{T}[1\,..\,n]$ and a pattern $\mathcal{P}[1\,..\,m]$. Suppose $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \equiv [s|^j\,..\,e|^j]$ and $Int^{\mathcal{T}}_{\mathcal{P}} \equiv [s\,..\,e]$. If $1 \le j \le m$, we must have $s|^j \le s \le e \le e|^j$.*

Clearly, Lemma 1 tells us that the range identifying the occurrences of a pattern must be contained in or equal to the range identifying the occurrences of a prefix of that pattern. How do we use the lemma to ensure a optimal value of $\mathcal{K}$? We modify our query algorithm as follows. We only need to modify the part that uses $SA_{\mathcal{T}}$ and do some more work while we compute B_j. At the beginning (before the loop at line 2 of Algorithm 3), we compute $Int^{\mathcal{T}}_{\mathcal{P}} \equiv [s\,..\,e]$ using $SA_{\mathcal{T}}$. Now, suppose that we have computed $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \equiv [s_\ell|^j\,..\,e_\ell|^j]$ using $SA_{\mathcal{T}}$. By Lemma 1,

$\mathcal{T}[i\,..\,n]$	aatcgatcgag	ag	atcaatcgatcgag	atcgag	atcgatcgag	caatcgatcgag	cgag	cgatcgag	cgctgatcaatcgatcgag	ctgatcaatcgatcgag	g	gag	gatcaatcgatcgag	gatcgag	gctgatcaatcgatcgag	tcaatcgatcgag	tcgag	tcgatcgag	tgatcaatcgatcgag
$SA_{\mathcal{T}}$	9	18	6	14	10	8	16	12	1	3	19	17	5	13	2	7	15	11	4
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$							$Int^{\mathcal{T}}_{\texttt{cga}}$												
							$Int^{\mathcal{T}}_{\texttt{cg}}$												
							$Int^{\mathcal{T}}_{\texttt{c}}$												
							$Int^{\mathcal{T}}_{-}$												

j	1	2	3	4
$\mathcal{P}[1\,..\,j-1]$	$-$	c	cg	cga
$Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$	$[1\,..\,19]$	$[6\,..\,10]$	$[7\,..\,9]$	$[7\,..\,8]$
$\mathcal{P}[j+1\,..\,m]$	gat	at	t	$-$
$Int^{\overleftarrow{\mathcal{T}}}_{\overline{\mathcal{P}[1..j-1]}}$	$[17\,..\,18]$	$[16\,..\,18]$	$[16\,..\,19]$	$[1\,..\,19]$
q_j	$(1,17)\times(19,18)$	$(6,16)\times(10,18)$	$(7,16)\times(9,19)$	$(7,1)\times(8,19)$

Table 5. Graphical representation of intervals $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]}, 1 \le j \le |\mathcal{P}|$, in the suffix array $SA_{\mathcal{T}}$ of the pattern $\mathcal{P}$ =cgat over the text $\mathcal{T}$ =cgctgatcaatcgatcgag and corresponding q_j rectangles.

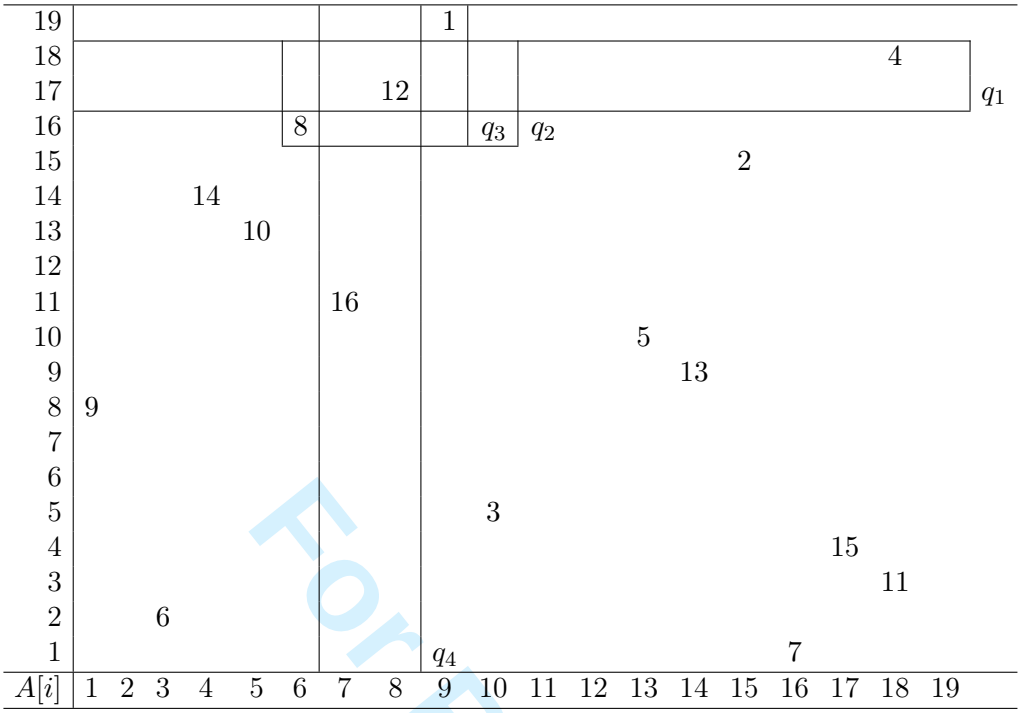

Table 6. Graphical representation of the Range Search Query index built over the text $\mathcal{T}$ =cgctgatcaatcgatcgag and used by Agorithm 3 on Line 5 in order to find $B|^j$ values, $1 \leq j \leq |\mathcal{P}|$, where $q_1 \equiv (1, 17) \times (19, 18)$, $q_2 \equiv (6, 16) \times (10, 18)$, $q_3 \equiv (7, 16) \times (9, 19)$, and $q_4 \equiv (7, 1) \times (8, 19)$, as reported in Table 5. We have $B|^1 = \{4, 12\}$, $B|^2 = \{8, 12\}$, $B|^3 = \{1, 12\}$, and $B|^4 = \{12, 16\}$.

we know that $s_\ell|^j \leq s \leq e \leq e_\ell|^j$. So, we divide the range into (at most) two ranges $[s_\ell|^j \,..\, s-1]$ and $[e+1 \,..\, e_\ell|^j]$. Then the computation of B_j is modified as follows:

$$B|^j = \{(x, y) \mid (x, y) \in A \text{ and } (x, y) \text{ is contained in } q_1 \text{ or in } q_2,$$
$$q_1 \equiv (s_\ell|^j, s_r|^j) \times (s-1, e_r|^j),\ q_2 \equiv (e+1, s_r|^j) \times (e_\ell|^j, e_r|^j)\}$$

In other words, what we are doing is that the interval of each of the prefixes $\mathcal{P}[1 \,..\, j-1], 1 \leq j \leq m$, of $\mathcal{P}$ is divided into at most two sub-intervals so as to remove the exact occurrences of $\mathcal{P}$. Hence, we finally need to include the exact occurrences of $\mathcal{P}$ before returning the set. In other words, instead of returning $B = \bigcup_{i=1}^{j} B|^j$ we need to return $B \bigcup \text{Occ}_{\mathcal{T}}^{\mathcal{P}}$. Clearly, now the output size $\mathcal{K}$ will be optimal.

Finally, what will be the query time of the augmented query algorithm? Clearly, now for each of the prefixes $\mathcal{P}[1 \,..\, j-1], 1 \leq j \leq m$, we would have at most two ranges (against one range of $\mathcal{P}[j+1 \,..\, m], 1 \leq j \leq m$). So, we need to make at most $2m$ range search queries in total keeping the total query time asymptotically the same as before. The results of this section can be summarized in the following theorem.

THEOREM 1. *Given a sequence $\mathcal{T}[1\,..\,n]$ over a constant alphabet and an integer m, we can construct a data structure in $O(n\log^{1+\varepsilon} n)$ time and space that, given a pattern $\mathcal{P}$ of length m as a query, can compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1}$ in $O(m\log\log n+\mathcal{K})$ time, where $\mathcal{K}=|\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1}|$.*

4.4. Exactly one mismatch

In some practical application, especially in Bioinformatics, the occurrences with exactly one mismatch is sought. In other words, in such cases, the exact occurrences are to be excluded, i.e., we are to compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{=1}$ instead of $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1}$. Clearly this can be achieved without any change in the query time. In this case, we simply need to return $B=\bigcup_{i=1}^{m} B|^{j}$ in the end in our augmented algorithm (without including $\mathrm{Occ}^{\mathcal{P}}_{\mathcal{T}}$).

5. General Case

In this section we relax the assumption that the pattern length m is given as part of the input. So, we cannot take advantage of using m during the data structure construction as we did in line 3 of Algorithm 2. For this case, we use the same idea but the shifting need be done in a different way so that we do not require the knowledge of m. This in the sequel requires a different query processing algorithm as will be discussed shortly. Below, we first discuss the index construction and then focus on to the query processing algorithm.

5.1. Index Construction

As has been mentioned previously, we will use the same basic idea and hence will depend on Algorithm 1. Let us suppose that we know the position of the mismatch and assume that the position is j. Similar to the previous approach we will maintain two suffix arrays $SA_{\mathcal{T}}$ and $SA_{\overleftarrow{\mathcal{T}}}$. However, we now reverse the role of the two suffix arrays as follows. We use $SA_{\mathcal{T}}$ to find the occurrences of $\mathcal{P}[j+1\,..\,m]$ (rather than $\mathcal{P}[1\,..\,j-1]$ as we did previously). We use the suffix array of the reverse string of $\mathcal{T}$, i.e. $SA_{\overleftarrow{\mathcal{T}}}$, to find the occurrences of $\overleftarrow{\mathcal{P}[1\,..\,j-1]}$. By doing so, in fact, we get the end positions of the occurrences of $\mathcal{P}[1\,..\,j-1]$ in $\mathcal{T}$. Now, note that we have the end positions of $\mathcal{P}[1\,..\,j-1]$ in $\mathcal{T}$ and the start positions of $\mathcal{P}[j+1\,..\,m]$ in $\mathcal{T}$. So, to get the desired alignment all we need to do is to consider the mismatch position and relabel accordingly. Since now the alignment is done at the start position of $\mathcal{P}[j+1\,..\,m]$ (instead of the start position of $\mathcal{P}[1\,..\,j-1]$), we do not need the knowledge of m. It is easy to verify that, now, each $i\in SA_{\overleftarrow{\mathcal{T}}}$ needs to be relabelled as $(n+1)-i+1+1=n-i+3$ to get the desired alignment. The rest of the steps are identical to Algorithm 2. Clearly, the running time of the index construction remains the same as before.

5.2. Query Processing

The query processing algorithm is built on the same principle as before. Let us suppose that we know the position of the mismatch and assume that the position is j. Then the query can be answered as follows. We first compute $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]} \equiv [s_r|^j \,..\, e_r|^j]$ using $SA_{\mathcal{T}}$. Then we compute $Int^{\overleftarrow{\mathcal{T}}}_{\overleftarrow{\mathcal{P}[1..j-1]}} \equiv [s_\ell|^j \,..\, e_\ell|^j]$ using $SA_{\overleftarrow{\mathcal{T}}}$. Now, we find all the points in A that are inside the rectangle $q \equiv (s_\ell|^j, s_r|^j) \times (e_\ell|^j, e_r|^j)$.

Let $B|^j$ is the set of those points. Now, note that we have done the alignment at the start position of $\mathcal{P}[j+1\,..\,m]$. So, we need to undo this shift to get the actual start position of the desired occurrence of the pattern $\mathcal{P}$. So we compute, $B'|^j = \{(SA_{\mathcal{T}}[x] - 1 - (j-1)) \mid (x,y) \in B|^j\}$. It is easy to verify that $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^j_{\leq 1} = B'|^j$.

Now, we can easily lift the restriction that j is the position of the mismatch. We simply, compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|^j_{\leq 1}$, i.e., $B|^j$ and $B'|^j$ for all values of j and compute $\mathrm{Occ}^{\mathcal{T}}_{\mathcal{P}}|_{\leq 1} = \bigcup_{j=1}^{m-1} B'|^j$.

5.3. Analysis of the Query Processing Algorithm

The analyses of the two algorithms, namely, the query processing algorithm presented in Section 4.2 and that presented in the previous section (Section 5.2) are similar but with a distinct difference that makes the running time of the latter worse. For the query processing of the fixed m case, we computed $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$ incrementally as we increment j from 1 to m and got all the intervals from $SA_{\mathcal{T}}$, namely, $[s_\ell|^j \,..\, e_\ell|^j], 1 \leq j \leq m$, spending $O(m)$ time in total. Similarly we computed all the intervals from $SA_{\overleftarrow{\mathcal{T}}}$, namely, $[s_r|^j \,..\, e_r|^j], 1 \leq j \leq m$, spending $O(m)$ time in total. For the general case unfortunately, we cannot apply the above trick readily. This is because, now, we need to compute $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$ incrementally as we increment j from 1 to m. This means that, unlike the previous case, where we were looking for intervals for prefixes of increasing sizes, now we have to compute intervals for suffixes of decreasing sizes. The same applies for the computation of intervals for $\overleftarrow{\mathcal{P}[1\,..\,j-1]}, 1 \leq j \leq m$, using $SA_{\overleftarrow{\mathcal{T}}}$. As a result the query time becomes $O(m^2 + m \log\log n + \mathcal{K})$.

5.4. Improved Query Time

Clearly the bottleneck is the computation of the appropriate intervals for $\mathcal{P}[j+1\,..\,m]$ and $\overleftarrow{\mathcal{P}[1\,..\,j-1]}$ for different values of j. To achieve the same query time of $O(m \log\log n + \mathcal{K})$ we need to be able to compute the intervals for the suffixes of decreasing sizes incrementally as we did for the prefixes of increasing sizes. This would ensure a $O(m)$ running time for the computation of the appropriate intervals and thereby keeping the desired running time intact. To achieve this improvement we resort to the famous backward search technique of Ferragina and Manzini [20, 21, 22] using their data structure popularly known as the FM-index. In particular, we use the following result which is the heart of the Backward Search Algorithm (BSA) of [20] using an FM-index.

LEMMA 2. *[Ferragina, Manzini [20]] Assume* $[s\,..\,e] = Int_{\mathcal{P}}^{\mathcal{T}}$ *is already computed. Then for any character* c*, the interval* $[s'\,..\,e'] = Int_{c\mathcal{P}}^{\mathcal{T}}$ *can be computed in* $O(1)$ *time.*

It is clear that with Lemma 2 at our disposal, we can compute the appropriate intervals for suffixes of decreasing sizes of a pattern $\mathcal{P}$ spending $O(|\mathcal{P}|)$ time in total. So, using BSA of [20, 21, 22] we are now able to compute the appropriate intervals for $\mathcal{P}[j+1\,..\,m]$ and $\overleftarrow{\mathcal{P}}[1\,..\,j-1]$ for different values of j spending a total time of $O(m)$. Therefore, the query time improves to $O(m \log\log n + \mathcal{K})$, where $\mathcal{K}$ is the output size.

5.5. Data Structure Construction with FM-Index

In the previous section we have used BSA (Lemma 2) to achieve the desired running time for the query. However, this means that we would need to include the FM-index in our index data structure. We do actually delegate the pattern matching part of our solution, that is the part in charge of computing the intervals of all the prefixes and the suffixes of a given pattern in the suffix arrays of the text and reverse text, to two FM-indexes: one for the text and another one for the reverse text. Notice that, since FM-index is a self-index, it does not need to access to the original text at query time. This leads to a practical decreasing of the space occupancy of the proposed solution due to the sublinear space occupancy of the FM-index for compressible texts. However, the space complexity of our solution remains dominated by the range search data structure.

A further analysis of the data structure construction time is in order. In what follows, we refer to the version of the FM-index presented in [22]. The FM-index and hence the BSA, make clever use of the so-called *Rank* query. Suppose we have a string X, $c \in \Sigma$ and f is an integer. Then, $Rank_X(c, f)$ is defined to be the number of occurrences of c (i.e., rank) in the prefix $X[1\,..\,f]$. FM-index uses a wavelet tree [24] to facilitate constant time *Rank* queries. It also requires an auxiliary array $\mathcal{C}$ such that $\mathcal{C}[c]$ stores the total number of occurrences of all $c' \leq c$, where '$\leq$' here means lexicographically smaller than or equal to. Finally, FM-index and BSA utilise the famous Burrows-Wheeler transformation (BWT) technique [9]. A complete description of BWT is out of the scope of and not required for our discussion. We just give a brief definition of BWT in relation to suffix array as follows. The BWT encoding of the string $\mathcal{T}$ is another string $\mathcal{B}_{\mathcal{T}}$ as defined below. Assume that the ith element of the suffix array of $\mathcal{T}$ is $SA_{\mathcal{T}}[i]$. Then $\mathcal{B}_{\mathcal{T}}[i] = \mathcal{T}[SA_{\mathcal{T}}[i] - 1]$ where $\mathcal{T}[0] = \mathcal{T}[n]$. The computation of $\mathcal{B}_{\mathcal{T}}$ can be done in $O(n)$ time. FM-index needs to preprocess $\mathcal{B}_{\mathcal{T}}$ for constant-time rank queries. Since wavelet trees can be constructed in linear time, this can also be achieved in $O(n)$ time. The auxiliary array $\mathcal{C}$ can also be prepared in $O(n)$ time. So overall the data structure construction time remains dominated by the range search data structure construction.

5.6. Optimality of $\mathcal{K}$

The problem with the optimality of $\mathcal{K}$ as discussed in Section 4.3 applies for the current algorithm as well. Unfortunately, The method employed in Section 4.3 to make $\mathcal{K}$ optimal cannot be used now. The technique employed in Section 4.3 was based on Lemma 1, which basically states that the interval provided by a suffix array for a pattern always lies within the interval of any of its prefixes. But this relation does not hold for suffixes. Therefore, to compute $\mathcal{K}$ we cannot use Lemma 1. Nevertheless, we apply another technique to achieve our goal as described below. We need the following lemma.

LEMMA 3. *Suppose we are given a suffix array $SA_{\mathcal{T}}[1\,..\,n]$ of a text $\mathcal{T}[1\,..\,n]$ and a pattern $\mathcal{P}[1\,..\,m]$. Suppose $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]} \equiv [s_1\,..\,e_1]$ and $Int^{\mathcal{T}}_{\mathcal{P}[j..m]} \equiv [s_0\,..\,e_0]$, where $1 \leq j < m$. Then the followings hold true:*

1. *The size of the interval $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$ is greater than or equal to the size of the interval $Int^{\mathcal{T}}_{\mathcal{P}[j..m]}$, i.e., $e_1 - s_1 + 1 \geq e_0 - s_0 + 1$.*
2. *We have $SA_{\mathcal{T}}[s_0] + 1 = SA_{\mathcal{T}}[p]$ for some $s_1 \leq p \leq e_1$.*
3. *Suppose, $SA_{\mathcal{T}}[s_0] + 1 = SA_{\mathcal{T}}[p]$. Then, $SA_{\mathcal{T}}[s_0 + i] + 1 = SA_{\mathcal{T}}[p + i]$ for all $1 \leq i \leq e_1 - s_1$.*

Now let us discuss how we can obtain the optimal $\mathcal{K}$ as follows. The basic idea is similar as before. We want to divide an interval into at most two subintervals such that the subinterval responsible for the exact occurrences of the complete pattern can be excluded when we do the intersection. Now, suppose we have computed $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$ and $Int^{\mathcal{T}}_{\mathcal{P}[j..m]}$. By Lemmas 3(2) and 3(3) we know that the occurrences of $\mathcal{P}[j+1\,..\,m]$ in some sense 'contain' the occurrences of $[j\,..\,m]$. This is because $\mathcal{P}[j\,..\,m] = \mathcal{P}_j\mathcal{P}[j+1\,..\,m]$.

Now, let us go back to our computation assuming that we know the position of the mismatch and assume that the position is j. So, we need to do the intersection between $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]} \equiv [s_r|^j\,..\,e_r|^j]$ (computed using $SA_{\mathcal{T}}$) and $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \equiv [s_\ell|^j\,..\,e_\ell|^j]$ (computed using $SA_{\overleftarrow{\mathcal{T}}}$). Note carefully that, here $\mathcal{P}_j$ corresponds to the mismatch position. So, if from the interval $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$ we can remove the positions which follows the occurrence of $\mathcal{P}_j$ in $\mathcal{T}$ we are done. Assuming that we have $Int^{\mathcal{T}}_{\mathcal{P}[j..m]}$ at our hand, we can easily identify those positions using Lemma 3 as follows. Following the hypothesis of Lemma 3, let us assume that $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]} \equiv [s_1\,..\,e_1]$ and $Int^{\mathcal{T}}_{\mathcal{P}[j..m]} \equiv [s_0\,..\,e_0]$. By Lemma 3(2), we have an $s_1 \leq p \leq e_1$ such that $SA_{\mathcal{T}}[s|^j] + 1 = SA_{\mathcal{T}}[p]$. We identify this p. How can we identify p efficiently? To do this efficiently, we slightly augment our index data structure by maintaining an auxiliary array $Next[1\,..\,n]$ of length n as follows. We store $Next[i] = j$ if and only if $SA_{\mathcal{T}}[i] + 1 = SA_{\mathcal{T}}[j]$. Clearly this will facilitate $O(1)$ time identification of the index p. Also, note that the computation of $Next[1\,..\,n]$ can be done in linear time during the index data structure construction.

Once p is identified, we compute $q = p + e_0 - s_0$. Now, we have two subintervals, namely, $[s_1\,..\,p-1]$ and $[q+1\,..\,e_1]$. It is not very difficult to realise that these two intervals, namely,

$[s_1 \,..\, p-1]$ and $[q+1 \,..\, e_1]$ give us the positions where $\mathcal{P}[j+1 \,..\, m]$ occurs in $\mathcal{T}$ but is not preceded by an occurrence of $\mathcal{P}_j$; this is exactly what we desired. So, now we finish off by modifying the computation of B_j appropriately. To show the modification of the computation of B_j we need to keep the notational convention followed so far. Note that the position of $\mathcal{P}[j+1 \,..\, m]$ is to the right with respect to the fixed mismatch position j. So, to keep the notational symmetry we will now rename s_1 and e_1 as $s_r|^j$ and $e_r|^j$ respectively. Now the modified computation of B_j is shown below.

$$B|^j = \{(x,y) \mid (x,y) \in A \text{ and } (x,y) \text{ is contained in } q_1 \text{ or in } q_2,$$
$$q_1 \equiv (s_\ell|^j, s_r|^j) \times (e_\ell|^j, p-1),\ q_2 \equiv (s_\ell|^j,\ q+1) \times (e_\ell|^j, e_r|^j)\}$$

Finally, we need to include the exact occurrences of $\mathcal{P}$ before returning the final output set. In other words, instead of returning $B = \bigcup_{i=1}^{j} B|^j$, we need to return $B \bigcup \mathrm{Occ}_{\mathcal{T}}^{\mathcal{P}}$. Clearly, now the output size $\mathcal{K}$ will be optimal. By similar argument as presented in Section 4.3, the query time remains asymptotically the same, i.e., $O(m \log\log n + \mathcal{K})$. The results of this section can be summarized in the following theorem.

THEOREM 2. *Given a sequence $\mathcal{T}$ of length n, we can construct a data structure in $O(n \log^{1+\varepsilon} n)$ time and space that, given a pattern $\mathcal{P}$ of length m as a query, can compute $\mathrm{Occ}_{\mathcal{P}}^{\mathcal{T}}|_{\leq 1}$ in $O(m \log\log n + \mathcal{K})$ time, where $\mathcal{K} = |\mathrm{Occ}_{\mathcal{P}}^{\mathcal{T}}|_{\leq 1}|$.*

6. Discussion on the solution of [5]

As has been mentioned above, the underlying idea used in our solution approach is identical to that of the solution proposed by Amir et al. [5]. In this section we present a detailed comparison between the two solutions and identify the shortcomings of the solution of [5]. In what follows, we will refer to the data structure of [5] as *DS_AKLLLR* (using the first letters of the authors' surnames). And we will use *DS_Fixed* and *DS_Gen* to refer to our data structures for the fixed m version and the general version, respectively. In both cases, we will use the names to refer to the corresponding algorithms as well.

DS_AKLLLR consists of two suffix trees, one for $\mathcal{T}$ and one for $\overleftarrow{\mathcal{T}}$. The role of these two suffix trees is exactly the same as the role of the two suffix arrays in our data structure *DS_Fixed*. Recall that in *DS_Gen* the roles of the two suffix arrays (FM-indexes, to be precise) are in fact reversed. The range search data structure used by *DS_AKLLLR* is similar but not identical to the one used by *DS_Fixed* and *DS_Gen*, as will be clear shortly. So, overall, the data structure construction is almost similar in both algorithms. The query algorithm however is drastically different as discussed below.

DS_AKLLLR query algorithm requires that the suffix trees are constructed using the Weiner construction [55]. According to Weiner's algorithm, given the text $\mathcal{T}$ of length n, the

suffix $\mathcal{T}[n\,..\,n]$ is first considered, followed by $\mathcal{T}[n-1\,..\,n]$, then $\mathcal{T}[n-2\,..\,n]$ and so on. Now similar to *DS_Fixed*, *DS_AKLLLR* needs to compute $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \equiv [s_\ell|^j\,..\,e_\ell|^j]$ using $ST_{\mathcal{T}}$ and $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]} \equiv [s_r|^j\,..\,e_r|^j]$ using $ST_{\overleftarrow{\mathcal{T}}}$. So with the suffix tree ST_T of $\mathcal{T}$ at its disposal, it proceeds as follows. *DS_AKLLLR* 'feeds' the pattern $\mathcal{P}$ to the suffix tree in some sense. In particular, it continues from $ST_{\mathcal{T}}$ and builds a suffix tree of $\mathcal{P}_1\mathcal{P}_2\ldots\mathcal{P}_m\#\mathcal{T}$, where $\# \notin \Sigma$. While doing this extended construction, *DS_AKLLLR* cleverly keeps track of the locus positions for each suffix of $\mathcal{P}$. And hence it can easily get the range $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]} \equiv [s_\ell|^j\,..\,e_\ell|^j]$. After the construction ends, it does 'undo' this to keep the suffix tree $ST_{\mathcal{T}}$ as before. Identical operation on $ST_{\overleftarrow{\mathcal{T}}}$ using $\overleftarrow{\mathcal{T}}$ gives the range $Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]} \equiv [s_r|^j\,..\,e_r|^j]$. Subsequently, all is needed is the application of an appropriate range search query on the range search data structure which is a part of *DS_AKLLLR*.

Now, there are two important points that need be carefully noted as highlighted below.

- First, the discussion, in Section 4.3 and later again in Section 5.6, on the output size $\mathcal{K}$ applies to *DS_AKLLLR* as well. To make $\mathcal{K}$ optimal *DS_AKLLLR* employs a different technique than ours. In particular, it resorts to a higher dimensional range search query. This is where the range search data structure used by *DS_AKLLLR* differs from the one used by our data structure. We omit the details because this is not relevant. But the point is that, this technique required the data structure of 3D range search and 3D query. This makes both the data structure construction time and query time (slightly) inferior to that of ours.

- Secondly, and more importantly, the claim that the computation of $Int^{\mathcal{T}}_{\mathcal{P}[1..j-1]}$ ($Int^{\mathcal{T}}_{\mathcal{P}[j+1..m]}$) by 'feeding' $\mathcal{P}$ ($\overleftarrow{\mathcal{P}}$) to $ST_{\mathcal{T}}$ ($ST_{\overleftarrow{\mathcal{T}}}$) runs asymptotically in $O(m)$ time is somewhat flawed as follows. The linear time of the Weiner construction of suffix tree (and in fact all other linear time construction, e.g., [54]) depends on an amortised analysis. Hence, while we can certainly say that the construction of a suffix tree for the string $\mathcal{P}\#\mathcal{T}$ can be done in $O(|\mathcal{P}| + |\mathcal{T}|)$ time, given a suffix tree for $\mathcal{T}$ we cannot always claim that extending it for $\mathcal{P}\#\mathcal{T}$ can be done in $O(|\mathcal{P}|)$ time.

7. Conclusion

In this paper we have focused on the the problem of indexing sequences for mapping reads with a single mismatch. We have considered a simpler problem first, where the pattern length m is given beforehand along with the text $\mathcal{T}$ of length n for preprocessing. So, in this version, the patterns to be queried must be of the same length, m. This simpler problem is interesting in its own right especially in the context of the Next Generation Sequencing (NGS). Subsequently, we have discussed how to solve the more general problem, which can handle patterns of different lengths. In both cases, our algorithm can construct an efficient data structure in $O(n \log^{1+\varepsilon} n)$

time and space, which is able to answer subsequent queries in $O(m \log\log n + \mathcal{K})$ time, where $0 < \varepsilon < 1$ and $\mathcal{K}$ is the optimal output size.

References

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

[2] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact strring matching based on suffix arrays. In A. H. F. Laender and A. L. Oliveira, editors, *SPIRE*, volume 2476 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2002.

[3] K. R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

[4] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *FOCS*, pages 198–207, 2000.

[5] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.

[6] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.

[7] R. A. Baeza-Yates and G. H. Gonnet. Fast string matching with mismatches. *Inf. Comput.*, 108(2):187–199, 1994.

[8] P. Bille, I. L. Gørtz, H. W. Vildhøj, and S. Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 2013.

[9] M. Burrows and D. Wheeler. A block-sorting loss-less data compression algorithm. *SRC Research Report*, 124, 1994.

[10] L. Cartegni and A. Krainer. Disruption of an SF2/ASF-dependent exonic splicing enhancer in SMN2 causes spinal muscular atrophy in the absence of SMN1. *Nature Genetics*, 30:377–384, 2002.

[11] C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms*. Texts in Algorithmics. King's College London, London, UK, 2004.

[12] P. Clifford and R. Clifford. Self-normalised distance with don't cares. In B. Ma and K. Zhang, editors, *CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 63–70. Springer, 2007.

[13] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In L. Babai, editor, *STOC*, pages 91–100. ACM, 2004.

[14] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In J. H. Reif, editor, *STOC*, pages 592–601. ACM, 2002.

[15] M. Crochemore, C. Epifanio, A. Gabriele, and F. Mignosi. From nerode's congruence to suffix automata with mismatches. *Theor. Comput. Sci.*, 410(37):3471–3480, 2009.

[16] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.

[17] M. Crochemore, M. Kubica, T. Walen, C. S. Iliopoulos, and M. S. Rahman. Finding patterns in given intervals. *Fundam. Inform.*, 101(3):173–186, 2010.

[18] A. Dermouche. A fast algorithm for string matching with mismatches. *Inf. Process. Lett.*, 55(2):105–110, 1995.

[19] M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143, 1997.

[20] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000.

[21] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[22] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly fm-index. In A. Apostolico and M. Melucci, editors, *SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160. Springer, 2004.

[23] M. Fischer and M. Paterson. String matching and other products. *in Complexity of Computation, R.M. Karp (editor), SIAM AMS Proceedings*, 7:113–125, 1974.

[24] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006.

[25] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[26] Z. Galil and R. Giancarlo. Parallel string matching with k mismatches. *Theor. Comput. Sci.*, 51:341–348, 1987.

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

[28] J. Holub, W. F. Smyth, and S. Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008.

[29] C. S. Iliopoulos, L. Mouchard, and M. S. Rahman. A new approach to pattern matching in degenerate dna/rna sequences and distributed pattern matching. *Mathematics in Computer Science*, 1(4):557–569, 2008.

[30] C. S. Iliopoulos and M. S. Rahman. Indexing circular patterns. In S.-I. Nakano and M. S. Rahman, editors, *WALCOM*, volume 4921 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2008.

[31] C. S. Iliopoulos and M. S. Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009.

[32] P. Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *FOCS*, pages 166–173. IEEE Computer Society, 1998.

[33] A. Kalai. Efficient pattern-matching with don't cares. In D. Eppstein, editor, *SODA*, pages 655–656. ACM/SIAM, 2002.

[34] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Simple linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[35] D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2-4):126–142, 2005.

[36] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[37] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

[38] J. Kwoka, G. M. Hallidaybc, W. S. Brooksbd, G. Doliose, H. Laudonf, O. M. M. Halluppa, R. F. Badenhopac, J. Vickersh, R. Wange, J. Naslundf, A. Takashimag, S. Gandyi, and P. Schofieldacj. Presenilin-1 mutation l271v results in altered exon 8 splicing and alzheimer's disease with non-cored plaques and no neuritic dystrophy. *Journal of Biological Chemistry*, 278(9):6748–6754, 2003.

[39] T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-M. Yiu. Space efficient indexes for string matching with don't cares. In T. Tokuyama, editor, *ISAAC*, volume 4835 of *Lecture Notes in Computer Science*, pages 846–857. Springer, 2007.

[40] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986.

[41] M. Lewenstein. Orthogonal range searching for text indexing. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 267–302. Springer, 2013.

[42] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[43] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[44] A. E. Minoche, J. C. Dohm, and H. Himmelbauer. Evaluation of genomic high-throughput sequencing data generated on Illumina HiSeq and Genome Analyzer systems. *Genome Biology*, 12(11):1–15, 2011.

[45] J. C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, and K. Park. Suffix array of alignment: A practical index for similar data. In O. Kurland, M. Lewenstein, and E. Porat, editors, *SPIRE*, volume 8214 of *Lecture Notes in Computer Science*, pages 243–254. Springer, 2013.

[46] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

[47] R. Y. Pinter. Efficient string matching with don't-care patterns. *Combinatorial ALgorithms on Words, NATO ASI Series*, 12:11–29, 1985.

[48] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

[49] M. S. Rahman and C. S. Iliopoulos. Pattern matching algorithms with don't cares. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, F. Plasil, and M. Bieliková, editors, *SOFSEM (2)*, pages 116–126. Institute of Computer Science AS CR, Prague, 2007.

[50] W. F. Smyth and S. Wang. An adaptive hybrid pattern-matching algorithm on indeterminate strings. *Int. J. Found. Comput. Sci.*, 20(6):985–1004, 2009.

[51] W. F. Smyth, S. Wang, and M. Yu. An adaptive hybrid pattern-matching algorithm on indeterminate strings. In J. Holub and J. Zdárek, editors, *Stringology*, pages 95–107. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2008.

[52] A. Tam, E. Wu, T. W. Lam, and S.-M. Yiu. Succinct text indexing with wildcards. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *SPIRE*, volume 5721 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2009.

[53] C. Thachuk. Succincter text indexing with wildcards. In R. Giancarlo and G. Manzini, editors, *CPM*, volume 6661 of *Lecture Notes in Computer Science*, pages 27–40. Springer, 2011.

[54] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[55] P. Weiner. Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11. IEEE Computer Society, 1973.