Algorithms for Jumbled Pattern Matching in Strings

Péter Burcsi[1], Ferdinando Cicalese[2], Gabriele Fici[3], and Zsuzsanna Lipták[4]

[1] Department of Computer Algebra, Eötvös Loránd University, Hungary
`bupe@compalg.inf.elte.hu`
[2] Dipartimento di Informatica ed Applicazioni, University of Salerno, Italy
`cicalese@dia.unisa.it`
[3] I3S, UMR6070, CNRS et Université de Nice-Sophia Antipolis, France
`fici@i3s.unice.fr`
[4] AG Genominformatik, Technische Fakultät, Bielefeld University, Germany
`zsuzsa@cebitec.uni-bielefeld.de`

Abstract. The Parikh vector $p(s)$ of a string s over a finite ordered alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ is defined as the vector of multiplicities of the characters, $p(s) = (p_1, \ldots, p_\sigma)$, where $p_i = |\{j \mid s_j = a_i\}|$. Parikh vector q occurs in s if s has a substring t with $p(t) = q$. The problem of searching for a query q in a text s of length n can be solved simply and worst-case optimally with a sliding window approach in $O(n)$ time. We present two novel algorithms for the case where the text is fixed and many queries arrive over time.

The first algorithm only *decides* whether a given Parikh vector appears in a binary text. It uses a linear size data structure and decides each query in $O(1)$ time. The preprocessing can be done trivially in $\Theta(n^2)$ time.

The second algorithm finds all occurrences of a given Parikh vector in a text over an arbitrary alphabet of size $\sigma \geq 2$ and has sub-linear expected time complexity. More precisely, we present two variants of the algorithm, both using an $O(n)$ size data structure, each of which can be constructed in $O(n)$ time. The first solution is very simple and easy to implement and leads to an expected query time of $O(n(\frac{\sigma}{\log \sigma})^{1/2} \frac{\log m}{\sqrt{m}})$, where $m = \sum_i q_i$ is the length of a string with Parikh vector q. The second uses wavelet trees and improves the expected runtime to $O(n(\frac{\sigma}{\log \sigma})^{1/2} \frac{1}{\sqrt{m}})$, i.e., by a factor of $\log m$.

Keywords: Parikh vectors, permuted strings, pattern matching, string algorithms, average case analysis, text indexing, non-standard string matching

1   Introduction

Parikh vectors of strings count the multiplicity of the characters. They have been reintroduced many times by different names (compomer [5], composition [3], Parikh vector [22], permuted string [7], permuted pattern [10], and others). They are natural objects to study, due to their numerous applications; for instance, in computational biology, they have been applied in alignment algorithms [3], SNP discovery [5], repeated pattern discovery [10], and, most naturally, in interpretation of mass spectrometry data [4]. Parikh vectors can be seen as a generalization of strings, where we view two strings as equivalent if one can be turned into the other by permuting its characters; in other words, if the two strings have the same Parikh vector.

The problem we are interested in here is answering the question whether a query Parikh vector q appears in a given text s (decision version), or where it occurs (occurrence version). An occurrence of q is defined as an occurrence of a substring t of s with Parikh vector q. The problem can be viewed as an approximate pattern matching problem: We are looking for an occurrence of a jumbled version of a query string t, i.e. for the occurrence of a substring t' which has the same Parikh vector. In the following, let n be the length of the text s, m the length of the query q (defined as the length of a string t with Parikh vector q), and σ the size of the alphabet.

The above problem (both decision and occurrence versions) can be solved with a simple sliding window based algorithm, in $O(n)$ time and $O(\sigma)$ additional storage space. This is worst case optimal with respect to the case of one query. However, when we expect to search for many queries in the same string, the above approach leads to $O(Kn)$ runtime for K queries. To the best of our knowledge, no faster approach is known. This is in stark contrast to the classical exact pattern matching problem, where all *exact* occurrences of a query pattern of length m are sought in a text of length n. In that case, for one query, any naive approach leads to $O(nm)$ runtime, while quite involved ideas for preprocessing and searching are necessary to achieve an improved runtime of $O(n + m)$, as do the Knuth-Morris-Pratt [17], Boyer-Moore [6] and Boyer-Moore-type algorithms (see, e.g., [2, 14]). However, when many queries are expected, the text can be preprocessed to produce a data structure of size linear in n, such as a suffix tree, suffix array, or suffix automaton, which then allows to answer individual queries in time linear in the length of the pattern (see any textbook on string algorithms, e.g. [23, 18]).

1.1   Related work

Jumbled pattern matching is a special case of approximate pattern matching. It has been used as a filtering step in approximate pattern matching algorithms [15], but rarely considered in its own right.

The authors of [7] present an algorithm for finding all occurrences of a Parikh vector in a runlength encoded text. The algorithm's time complexity is $O(n' + \sigma)$, where n' is the length of the runlength encoding of s. Obviously, if the string is not runlength encoded, a preprocessing phase of time $O(n)$ has to be added. However, this may still be feasible if many queries are expected. To the best of our knowledge, this is the only algorithm that has been presented for the problem we treat here.

An efficient algorithm for computing all Parikh fingerprints of substrings of a given string was developed in [1]. Parikh fingerprints are Boolean vectors where the k'th entry is 1 if and only if a_k appears in the string. The algorithm involves storing a data point for each Parikh fingerprint, of which there are at most $O(n\sigma)$ many. This approach was adapted in [10] for Parikh vectors and applied to identifying all repeated Parikh vectors within a given length range; using it to search for queries of arbitrary length would imply using $\Omega(P(s))$ space, where $P(s)$ denotes the number of different Parikh vectors of substrings of s. This is not desirable, since, for arbitrary alphabets, there are non-trivial strings of any length with quadratic $P(s)$ [8].

1.2 Results

In this paper, we present two novel algorithms which perform significantly better than the simple window algorithm, in the case where many queries arrive.

For the binary case, we present an algorithm which answers *decision queries* in $O(1)$ time, using a data structure of size $O(n)$ (Interval Algorithm, Sect. 3). The data structure is constructed in $\Theta(n^2)$ time.

For general alphabets, we present an algorithm with expected sublinear runtime which uses $O(n)$ space to answer *occurrence queries* (Jumping Algorithm, Sect. 4). We present two different variants of the algorithm. The first one uses a very simple data structure (an inverted table) and answers queries in time $O(\sigma J \log(\frac{n}{J} + m))$, where J denotes the number of iterations of the main loop of the algorithm. We then show that the expected value of J for the case of random strings and patterns is $O(\frac{n}{\sqrt{m}\sqrt{\sigma \log \sigma}})$, yielding an expected runtime of $O(n(\frac{\sigma}{\log \sigma})^{1/2}\frac{\log m}{\sqrt{m}})$, per query

The second variant of the algorithm uses wavelet trees [13] and has query time $O(\sigma J)$, yielding an overall expected runtime of $O(n(\frac{\sigma}{\log \sigma})^{1/2}\frac{1}{\sqrt{m}})$, per query. (Here and in the following, log stands for logarithm base 2.)

Our simulations on random strings and real biological strings confirm the sublinear behavior of the algorithms in practice. This is a significant improvement over the simple window algorithm w.r.t. expected runtime, both for a single query and repeated queries over one string.

The Jumping Algorithm is reminiscent of the Boyer-Moore-like approaches to the classical exact string matching problem [6, 2, 14]. This analogy is used both in its presentation and in the analysis of the number of iterations performed by the algorithm.

2 Definitions and problem statement

Given a finite ordered alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}, a_1 \leq \ldots \leq a_\sigma$. For a string $s \in \Sigma^*$, $s = s_1 \ldots s_n$, the *Parikh vector* $p(s) = (p_1, \ldots, p_\sigma)$ of s defines the multiplicities of the characters in s, i.e. $p_i = |\{j \mid s_j = a_i\}|$, for $i = 1, \ldots, \sigma$. For a Parikh vector p, the *length* $|p|$ denotes the length of a string with Parikh vector p, i.e. $|p| = \sum_i p_i$. An *occurrence* of a Parikh vector p in s is an occurrence of a substring t with $p(t) = p$. (An occurrence of t is a pair of positions $0 \leq i \leq j \leq n$, such that $s_i \ldots s_j = t$.) A Parikh vector that occurs in s is called a sub-Parikh vector of s. The prefix of

length i is denoted $pr(i) = pr(i, s) = s_1 \ldots s_i$, and the Parikh vector of $pr(i)$ as $prv(i) = prv(i, s) = p(pr(i))$.

For two Parikh vectors $p, q \in \mathbb{N}^\sigma$, we define $p \leq q$ and $p + q$ component-wise: $p \leq q$ if and only if $p_i \leq q_i$ for all $i = 1, \ldots, \sigma$, and $p + q = u$ where $u_i = p_i + q_i$ for $i = 1, \ldots, \sigma$. Similarly, for $p \leq q$, we set $q - p = v$ where $v_i = q_i - p_i$ for $i = 1, \ldots, \sigma$.

> **Jumbled Pattern Matching (JPM).** Let $s \in \Sigma^*$ be given, $|s| = n$. For a Parikh vector $q \in \mathbb{N}^\sigma$ (the query), $|q| = m$, find all occurrences of q in s. The *decision version* of the problem is where we only want to know whether q occurs in s.

We assume that K many queries arrive over time, so some preprocessing may be worthwhile.

Note that for $K = 1$, both the decision version and the occurrence version can be solved worst-case optimally with a simple window algorithm, which moves a fixed size window of size m along string s. Maintain the Parikh vector c of the current window and a counter r which counts indices i such that $c_i \neq q_i$. Each sliding step costs either 0 or 2 update operations of c, and possibly one increment or decrement of r. This algorithm solves both the decision and occurrence problems and has running time $\Theta(n)$, using additional storage space $\Theta(\sigma)$.

Precomputing, sorting, and storing all sub-Parikh vectors of s would lead to $\Theta(n^2)$ storage space, since there are non-trivial strings with a quadratic number of Parikh vectors over arbitrary alphabets [8]. Such space usage is inacceptable in many applications.

For small queries, the problem can be solved exhaustively with a linear size indexing structure such as a suffix tree, which can be searched down to length $m = |q|$ (of the substrings), yielding a solution to the decision problem in time $O(\sigma^m)$. For finding occurrences, report all leaves in the subtrees below each match; this costs $O(M)$ time, where M is the number of occurrences of q in s. Constructing the suffix tree takes $O(n)$ time, so for $m = o(\log n)$, we get a total runtime of $O(n)$, since $M \leq n$ for any query q.

3   Decision problem in the binary case

In this section, we present an algorithm for strings over a binary alphabet which, once a data structure of size $O(n)$ has been constructed, answers decision queries in constant time. It makes use of the following nice property of binary strings.

Lemma 1. *Let $s \in \{a, b\}^*$ with $|s| = n$. Fix $1 \leq m \leq n$. If the Parikh vectors $(x_1, m - x_1)$ and $(x_2, m - x_2)$ both occur in s, then so does $(y, m - y)$ for any $x_1 \leq y \leq x_2$.*

Proof. Consider a sliding window of fixed size m moving along the string and let (p_1, p_2) be the Parikh vector of the current substring. When the window is shifted by one, the Parikh vector either remains unchanged (if the character falling out is the same as the character coming in), or it becomes $(p_1 + 1, p_2 - 1)$ resp. $(p_1 - 1, p_2 + 1)$ (if

they are different). Thus the Parikh vectors of substrings of s of length m build a set of the form $\{(x, m - x) \mid x = \text{pmin}(m), \text{pmin}(m) + 1, \ldots, \text{pmax}(m)\}$ for appropriate $\text{pmin}(m)$ and $\text{pmax}(m)$. □

Assume that the algorithm has access to the values $\text{pmin}(m)$ and $\text{pmax}(m)$ for $m = 1, \ldots, n$; then, when a query $q = (x, y)$ arrives, it answers YES if and only if $x \in [\text{pmin}(x + y), \text{pmax}(x + y)]$. The query time is $O(1)$.

The table of the values $\text{pmin}(m)$ and $\text{pmax}(m)$ can be easily computed in a preprocessing step in time $\Theta(n^2)$ by scanning the string with a window of size m, for each m. Alternatively, lazy computation of the table is feasible, since for any query q, only the entry $m = |q|$ is necessary. Therefore, it can be computed on the fly as queries arrive. Then, any query will take time $O(1)$ (if the appropriate entry has already been computed), or $O(n)$ (if it has not). After n queries of the latter kind, the table is completed, and all subsequent queries can be answered in $O(1)$ time. If we assume that the query lengths are uniformly distributed, then this can be viewed as a coupon collector problem where the coupon collector has to collect one copy of each length m. Then the expected number of queries needed before having seen all n coupons is $nH_n \approx n \ln n$ (see e.g. [11]). The algorithm will have taken $O(n^2)$ time to answer these $n \ln n$ queries.

The assumption of the uniform length distribution may not be very realistic; however, even if it does not hold, we never take more time than $O(n^2 + K)$ for K many queries. Since any one query may take at most $O(n)$ time, our algorithm never performs worse than the simple window algorithm. Moreover, for those queries where the table entries have to be computed, we can even run the simple window algorithm itself and report all occurrences, as well. For all others, we only give decision answers, but in constant time.

The size of the data structure is $2n = O(n)$. The overall running time for either variant is $\Theta(K + n^2)$. As soon as the number of queries is $K = \omega(n)$, both variants outperform the simple window algorithm, whose running time is $\Theta(Kn)$.

Example 1. Let $s = ababbaabaabbbaaabbab$. In Table 1, we give the table of pmin and pmax for s. This example shows that the locality of pmin and pmax is preserved only in adjacent levels. As an example, the value $\text{pmax}(3) = 3$ corresponds to the substring *aaa* appearing only at position 14, while $\text{pmax}(5) = 4$ corresponds to the substring *aabaa* appearing only at position 6.

Table 1. An example of the linear data structure for answering queries in constant time.

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
pmin	0	0	0	1	2	2	3	3	4	4	5	5	6	7	7	8	8	9	9	10
pmax	1	2	3	3	4	4	4	5	5	6	7	7	7	8	8	9	9	9	10	10

4   The Jumping Algorithm

In this section, we introduce our algorithm for general alphabets. We first give the main algorithm and then present two different implementations of it. The first one, an inverted prefix table, is very easy to understand and to implement, takes $O(n)$ space and $O(n)$ time to construct (both with constant 1), and can replace the string. Then we show how to use a wavelet tree of s to implement our algorithm, which has the same space requirements as the inverted table, can be constructed in $O(n)$ time, and improves the query time by a factor of $\log m$.

4.1   Main algorithm

Let $s = s_1 \ldots s_n \in \Sigma^*$ be given. Recall that $prv(i)$ denotes the Parikh vector of the prefix of s of length i, for $i = 0, \ldots, n$, where $prv(0) = p(\epsilon) = (0, \ldots, 0)$. Consider Parikh vector $p \in \mathbb{N}^\sigma$, $p \neq (0, \ldots, 0)$. We make the following simple observations:

Observation 1   *1. For any $0 \leq i \leq j \leq n$, $p = prv(j) - prv(i)$ if and only if p occurs in s at position $(i+1, j)$.*
  2. If an occurrence of p ends in position j, then $prv(j) \geq p$.

The algorithm moves two pointers L and R along the text, pointing at these potential positions i and j. Instead of moving linearly, however, the pointers are updated in jumps, alternating between updates of R and L, in such a manner that many positions are skipped. Moreover, because of the way we update the pointers, after any update it suffices to check whether $R - L = |q|$ to confirm that an occurrence has been found (cf. Lemma 2 below).

We first need to define a function FIRSTFIT, which returns the smallest potential position where an occurence of a Parikh vector can end. Let $p \in \mathbb{N}^\sigma$, then

$$\text{FIRSTFIT}(p) := \min\{j \mid prv(j) \geq p\},$$

and set FIRSTFIT$(p) = \infty$ if no such j exists. We use the following rules for updating the two pointers, illustrated in Fig. 1.

Updating R: Assume that the left pointer is pointing at position L, i.e. no unreported occurrence starts before $L + 1$. Notice that, if there is an occurrence of q ending at any position $j > L$, it must hold that $prv(L) + q \leq prv(j)$. In other words, we must fit both $prv(L)$ and q at position j, so we update R to

$$R \leftarrow \text{FIRSTFIT}(prv(L) + q).$$

Updating L: Assume that R has just been updated. Thus, $prv(R) - prv(L) \geq q$ by definition of FIRSTFIT. If equality holds, then we have found an occurrence of q in position $(L+1, R)$, and L can be incremented by 1. Otherwise $prv(R) - prv(L) > q$, which implies that, interspersed between the characters that belong to q, there are some "superfluous" characters. Now the first position where an occurrence of q can start is at the beginning of a *contiguous* sequence of characters ending in R which all belong to q. In other words, we need the beginning of the longest suffix of $s[L+1, R]$
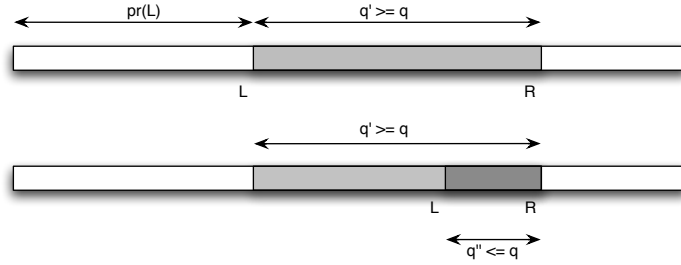
Fig. 1. The situation after the update of R (above) and after the update of L (below). R is placed at the first fit of $prv(L) + q$, thus q' is a super-Parikh vector of q. Then L is placed at the beginning of the longest good suffix ending in R, so q'' is a sub-Parikh vector of q.

with Parikh vector $\leq q$, i.e. the smallest position i such that $prv(R) - prv(i) \leq q$, or, equivalently, $prv(i) \geq prv(R) - q$. Thus we update L to

$$L \leftarrow \text{FIRSTFIT}(prv(R) - q).$$

Finally, in order to check whether we have found an occurrence of query q, after each update of R or L, we check whether $R - L = |q|$. In Figure 2, we give the pseudocode of the algorithm.

Algorithm *Jumping Algorithm*
Input: query Parikh vector q
Output: A set Occ containing all beginning positions of occurrences of q in s
1. set $m \leftarrow |q|; Occ \leftarrow \emptyset; L \leftarrow 0$;
2. **while** $L < n - m$
3.   **do** $R \leftarrow \text{FIRSTFIT}(prv(L) + q)$;
4.   **if** $R - L = m$
5.   **then** add $L + 1$ to Occ;
6. $L \leftarrow L + 1$;
7.   **else** $L \leftarrow \text{FIRSTFIT}(prv(R) - q)$;
8.   **if** $R - L = m$
9.   **then** add $L + 1$ to Occ;
10. $L \leftarrow L + 1$;
11. **return** Occ;

Fig. 2. Pseudocode of Jumping Algorithm

It remains to see how to compute the FIRSTFIT and *prv* functions. We first prove that the algorithm is correct. For this, we will need the following lemma.

Lemma 2. *The following algorithm invariants hold:*

1. *After each update of R, we have $prv(R) - prv(L) \geq q$.*
2. *After each update of L, we have $prv(R) - prv(L) \leq q$.*
3. *$L \leq R$.*

Proof. *1.* follows directly from the definition of FIRSTFIT and the update rule for R. For *2.*, if an occurrence was found at (i, j), then before the update we have $L = i - 1$ and $R = j$. Now L is incremented by 1, so $L = i$ and $prv(R) - prv(L) = q - e_{s_i} < q$, where e_k is the k'th unity vector. Otherwise, $L \leftarrow$ FIRSTFIT$(prv(R) - q)$, and again the claim follows directly from the definition of FIRSTFIT. For *3.*, if an occurrence was found, then L is incremented by 1, and $R - L = m - 1 \geq 0$. Otherwise, $L =$ FIRSTFIT$(prv(R) - q) = \min\{\ell \mid prv(\ell) \geq prv(R) - q\} \leq R$. □

Theorem 1. *Algorithm Jumping Algorithm is correct.*

Proof. We have to show that (1) if the algorithm reports an occurrence, then it is correct, and (2) if there is an occurrence, then the algorithm will find it.

(1) If the algorithm reports an index i, then $(i, i + m - 1)$ is an occurrence of q: An index i is added to Occ whenever $R - L = m$. If the last update was that of R, then we have $prv(R) - prv(L) \geq q$ by Lemma 2, and together with $R - L = m = |q|$, this implies $prv(R) - prv(L) = q$, thus $(L + 1, R) = (i, i + m - 1)$ is an occurrence of q. If the last update was L, then $prv(R) - prv(L) \leq q$, and it follows analogously that $prv(R) - prv(L) = q$.

(2) All occurrences of q are reported: Let's assume otherwise. Then there is a minimal i and $j = i + m - 1$ such that $p(s[i, j]) = q$ but i is not reported by the algorithm. By Observation 1, we have $prv(j) - prv(i - 1) = q$.

Let's refer to the values of L and R as two sequences $(L_k)_{k=1,2,\ldots}$ and $(R_k)_{k=1,2,\ldots}$. So we have $L_1 = 0$, and for all $k \geq 1$, $R_k =$ FIRSTFIT$(prv(L_k) + q)$, and $L_{k+1} = L_k + 1$ if $R_k - L_k = m$ and $L_{k+1} =$ FIRSTFIT$(prv(R_k) - q)$ otherwise. In particular, $L_{k+1} > L_k$ for all k.

First observe that if for some k, $L_k = i - 1$, then R will be updated to j in the next step, and we are done. This is because $R_k =$ FIRSTFIT$(prv(L_k) + q) =$ FIRSTFIT$(prv(i - 1) + q) =$ FIRSTFIT$(prv(j)) = j$. Similarly, if for some k, $R_k = j$, then we have $L_{k+1} = i - 1$.

So there must be a k such that $L_k < i - 1 < L_{k+1}$. Now look at R_k. Since there is an occurrence of q after L_k ending in j, this implies that $R_k =$ FIRSTFIT$(prv(L_k) + q) \leq j$. However, we cannot have $R_k = j$, so it follows that $R_k < j$. On the other hand, $i - 1 < L_{k+1} \leq R_k$ by our assumption and by Lemma 2. So R_k is pointing to a position somewhere between $i - 1$ and j, i.e. to a position within our occurrence of q. Denote the remaining part of q to the right of R_k by q': $q' = prv(j) - prv(R_k)$. Since $R_k =$ FIRSTFIT$(prv(L_k) + q)$, all characters of q must fit between L_k and R_k, so the Parikh vector $p = prv(i) - prv(L_k)$ is a super-Parikh vector of q'. If $p = q'$, then there is an occurrence of q at $(L_k + 1, R_k)$, and by minimality of (i, j), this occurrence was correctly identified by the algorithm. Thus, $L_{k+1} = L_k + 1 \leq i - 1$, contradicting our choice of k. It follows that $p > q'$ and we have to find the longest good suffix of the substring ending in R_k for the next update L_{k+1} of L. But $s[i, R_k]$ is a good suffix because its Parikh vector is a sub-Parikh vector of q, so $L_{k+1} =$ FIRSTFIT$(prv(R_k) - q) \leq i - 1$, again in contradiction to $L_{k+1} > i - 1$. □
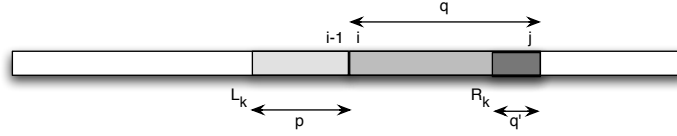
We illustrate the proof in Fig. 3.

Fig. 3. Illustration for proof of correctness.

4.2   Variant using an inverted table

Storing all prefix vectors of s would require $O(\sigma n)$ storage space, which may be too much. Instead, we construct an "inverted prefix vector table" I containing the increment positions of the prefix vectors: for each character $a_k \in \Sigma$, and each value j up to $p(s)_k$, the position in s of the j'th occurrence of character a_k. Formally, $I[k][j] = \min\{i \mid prv(i)_k \geq j\}$ for $j \geq 1$, and $I[k][0] = 0$. Then we have

$$\text{FIRSTFIT}(p) = \max_{k=1,\ldots,\sigma} \{I[k][p_k]\}.$$

We can also compute the prefix vectors $prv(i)$ from table I: For $k = 1, \ldots, \sigma$,

$$prv(j)_k = \max\{i \mid I[k][i] \leq j\}$$

The obvious way to find these values is to do binary search for j in each row of I. However, this would take time $\Theta(\sigma \log n)$; a better way is to use information already acquired during the run of the algorithm. By Lemma 2, it always holds that $L \leq R$. Thus, for computing $prv(R)_k$, it suffices to search for R between $prv(L)_k$ and $prv(L)_k + (R - L)$. This search takes time proportional to $\log(R - L)$. Moreover, after each update of L, we have $L \geq R - m$, so when computing $prv(L)_k$, we can restrict the search for L to between $prv(R)_k - m$ and $prv(R)_k$, in time $O(\log m)$. For more details, see Section 4.4.

Table I can be computed in one pass over s (where we take the liberty of identifying character $a_k \in \Sigma$ with its index k). The variables c_k count the number of occurrences of character a_k seen so far, and are initialized to 0.

Algorithm *Construct I*
1.   **for** $i = 1$ **to** n
2. $c_{s_i} = c_{s_i} + 1;$
3. $I[s_i][c_{s_i}] = i;$

Table I requires $O(n)$ storage space (with constant 1). Moreover, the string s can be discarded, so we have zero additional storage. (Access to $s_i, 1 \leq i \leq n$, is still possible, at cost $O(\sigma \log n)$.)

Example 2. Let $\Sigma = \{a, b, c\}$ and $s = cabcccaaabccbaacca$. The prefix vectors of s are given below. Note that the algorithm does not actually compute these.

pos.		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
s		c	a	b	c	c	c	a	a	a	b	c	c	b	a	a	c	c	a
# a's	0	0	1	1	1	1	1	2	3	4	4	4	4	4	5	6	6	6	7
# b's	0	0	0	1	1	1	1	1	1	1	2	2	2	3	3	3	3	3	3
# c's	0	1	1	1	2	3	4	4	4	4	4	5	6	6	6	6	7	8	8

The inverted prefix table I:

	0	1	2	3	4	5	6	7	8
a	0	2	7	8	9	14	15	18	
b	0	3	10	13					
c	0	1	4	5	6	11	12	16	17

Query $q = (3, 1, 2)$ has 4 occurrences, beginning in positions $5, 6, 7, 13$, since $(3, 1, 2) = prv(10) - prv(4) = prv(11) - prv(5) = prv(12) - prv(6) = prv(18) - prv(12)$. The values of L and R are given below:

k, see proof of Thm. 1	1	2	3	4	5	6	7
L	0	4	5	6	7	10	12
R	8	10	11	12	14	18	18
occurrence found?	−	yes	yes	yes	−	−	yes

4.3   Variant using a wavelet tree

A wavelet tree on $s \in \Sigma^*$ allows *rank, select,* and *access* queries in time $O(\log \sigma)$. For $a_k \in \Sigma$, $rank_k(s, i) = |\{j \mid s_j = a_k, j \leq i\}|$, the number of occurrences of character a_k up to and including position i, while $select_k(s, i) = \min\{j \mid rank_k(s, j) \geq i\}$, the position of the i'th occurrence of character a_k. When the string is clear, we just use $rank_k(i)$ and $select_k(i)$. Notice that

- $prv(j) = (rank_1(j), \ldots, rank_\sigma(j))$, and
- for a Parikh vector $p = (p_1, \ldots, p_\sigma)$, FIRSTFIT$(p) = \max_{k=1,\ldots,\sigma}\{select_k(p_k)\}$.

So we can use a wavelet tree of string s to implement those two functions. We give a brief recap of wavelet trees, and then explain how to implement the two functions above in $O(\sigma)$ time each.

A wavelet tree is a complete binary tree with $\sigma = |\Sigma|$ many leaves. To each inner node, a bitstring is associated which is defined recursively, starting from the root, in the following way. If $|\Sigma| = 1$, then there is nothing to do (in this case, we have reached

a leaf). Else split the alphabet into two roughly equal parts, Σ_{left} and Σ_{right}. Now construct a bitstring of length n from s by replacing each occurrence of a character a by 0 if $a \in \Sigma_{\text{left}}$, and by 1 if $a \in \Sigma_{\text{right}}$. Let s_{left} be the subsequence of s consisting only of characters from Σ_{left}, and s_{right} that consisting only of characters from Σ_{right}. Now recurse on the left child with string s_{left} and alphabet Σ_{left}, and on the right child with s_{right} and Σ_{right}. An illustration is given in Fig. 4. At each inner node, in addition to the bitstring B, we have a data structure of size $o(|B|)$, which allows to perform *rank* and *select* queries on bit vectors in constant time ([20, 9, 21]).

Now, using the wavelet tree of s, any *rank* or *select* operation on s takes time $O(\log \sigma)$, which would yield $O(\sigma \log \sigma)$ time for both $prv(j)$ and FIRSTFIT(p). However, we can implement both in a way that they need only $O(\sigma)$ time: In order to compute $rank_k(j)$, the wavelet tree, which has $\log \sigma$ levels, has to be descended from the root to leaf k. Since for $prv(j)$, we need all values $rank_1(j), \ldots, rank_\sigma(j)$ simultaneously, we traverse the complete tree in $O(\sigma)$ time.

For computing FIRSTFIT(p), we need $\max_k \{select_k(p_k)\}$, which can be computed bottom-up in the following way. We define a value x_u for each node u. If u is a leaf, then u corresponds to some character $a_k \in \Sigma$; set $x_u = p_k$. For an inner node u, let B_u be the bitstring at u. We define x_u by $x_u = \max\{select_0(B_u, x_{\text{left}}), select_1(B_u, x_{\text{right}})\}$, where x_{left} and x_{right} are the values already computed for the left resp. right child of u. The desired value is equal to x_{root}.

Example 3. Let $s = bbacaccabaddabccaaac$ (cp. Fig. 4). We demonstrate the computation of FIRSTFIT($2, 3, 2, 1$) using the wavelet tree. We have FIRSTFIT($2, 3, 2, 1$) $= \max\{select_a(s, 2), select_b(s, 3), select_c(s, 2), select_d(s, 1)\}$, where in slight abuse of notation we put the character in the subscript instead of its number. Denote the bottom left bitstring as $B_{a,b}$, the bottom right one as $B_{c,d}$, and the top bitstring as $B_{a,b,c,d}$. Then we get $\max\{select_0(B_{a,b}, 2), select_1(B_{a,b}, 3)\} = \max\{4, 6\} = 6$, and $\max\{select_0(B_{c,d}, 2), select_1(B_{c,d}, 1)\} = \max\{2, 4\} = 4$. So at the next level, we compute $\max\{select_0(B_{a,b,c,d}, 6), select_1(B_{a,b,c,d}, 4)\} = \max\{9, 11\} = 11$.


```
          1  2  3  4  5  6  7  8  9  10 11 12 13 14  15 16  17 18 19 20
          b  b  a  c  a  c  c  a  b  a  d  d  a  b  c  c  a  a  a  c
          0  0  0  1  0  1  1  0  0  0  1  1  0  0  1  1  0  0  0  1
                        a,b            c,d

   1 2 3 4 5 6 7 8 9 10 11 12          1  2  3  4  5  6  7  8
   b b a a a b a a b a a  a            c  c  c  d  d  c  c  c
   1 1 0 0 0 1 0 0 1 0 0  0            0  0  0  1  1  0  0  0
```
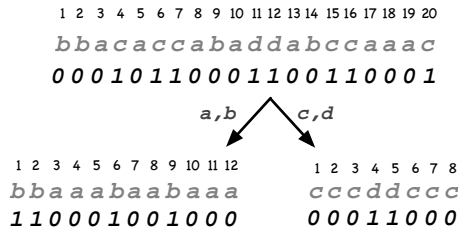
Fig. 4. The wavelet tree for string *bbacaccabaddabccaaac*. For clarity, the leaves have been omitted. Note also that the third line at each inner node (the strings over the alphabet $\{a, b, c, d\}$) are only included for illustration.

4.4   Algorithm Analysis

Let $\mathbb{A}_1(s,q)$ denote the running time of the Jumping Algorithm using inverted tables over a text s and a Parikh vector q, and $\mathbb{A}_2(s,q)$ that of the Jumping Algorithm using a wavelet tree. Further, let $J = J(s,q)$ be the number of iterations performed in the `while` loop in line 2, i.e., the number of jumps performed by the algorithm on the input q.

The time spent in each iteration depends on how the functions FIRSTFIT and prv are implemented (lines 3 and 7). In the wavelet tree implementation, as we saw before, both take time $O(\sigma)$, so the overall runtime of the algorithm is

$$\mathbb{A}_2(s,q) = O(\sigma J). \tag{1}$$

For the inverted table implementation, it is easy to see that computing FIRSTFIT takes $O(\sigma)$ time. Now denote, for each $i = 1, \ldots, J$, by $\hat{L}_i$, $\hat{R}_i$ the value of L and R after the i'th execution of line 3 of the algorithm, respectively.[5] The computation of $prv(\hat{L}_i)$ in line 3 takes $O(\sigma \log m)$: For each $k = 1, \ldots, \sigma$, the component $prv(\hat{L}_i)_k$ can be determined by binary search over the list $I[k][prv(\hat{R}_{i-1})_k - m], I[k][prv(\hat{R}_{i-1})_k - m+1], \ldots, I[k][prv(\hat{R}_{i-1})_k]$. By $\hat{L}_i \geq \hat{R}_{i-1} - m$, the claim follows.

The computation of $prv(\hat{R}_i)$ in line 7 takes $O(\sigma \log(\hat{R}_i - \hat{R}_{i-1} + m))$. Simply observe that in the prefix ending at position $\hat{R}_i$ there can be at most $\hat{R}_i - \hat{L}_i$ more occurrences of the k'th character than there are in the prefix ending at position $\hat{L}_i$. Therefore, as before, we can determine $prv(\hat{R}_i)_k$ by binary search over the list $I[k][prv(\hat{L}_i)_k], I[k][prv(\hat{L}_i)_k + 1], \ldots, I[k][prv(\hat{L}_i)_k + \hat{R}_i - \hat{L}_i]$. Using the fact that $\hat{L}_i \geq \hat{R}_{i-1} - m$, the desired bound follows.

The last three observations imply

$$\mathbb{A}_1(s,q) = O\left(\sigma J \log m + \sigma \sum_{i=1}^{J} \log(\hat{R}_i - \hat{R}_{i-1} + m)\right).$$

Notice that this is an overestimate, since line 7 is only executed if no occurrence was found after the current update of R (line 4). Standard algebraic manipulations using Jensen's inequality (see, e.g. [16]) yield $\sum_{i=1}^{J} \log(\hat{R}_i - \hat{R}_{i-1} + m) \leq J \log\left(\frac{n}{J} + m\right)$. Therefore we obtain

$$\mathbb{A}_1(s,q) = O\left(\sigma J \log\left(\frac{n}{J} + m\right)\right). \tag{2}$$

Average case analysis of J The worst case running time of the Jumping Algorithm, in either implementation, is superlinear, since there exist strings s of any length n and Parikh vectors q such that $J = \Theta(n)$: For instance, on the string $s = ababab \ldots ab$ and $q = (2, 0)$, the algorithm will execute $n/2$ jumps.

---

[5] The $\hat{L}_i$ and $\hat{R}_i$ coincide with the L_k and R_k from the proof of Theorem 1 almost but not completely: When an occurrence is found after the update of L, then the corresponding pair L_k, R_k is skipped here. The reason is that now we are only considering those updates that carry a computational cost.

This sharply contrasts with the experimental evaluation we present later. The Jumping Algorithm appears to have in practice a sublinear behavior. In the rest of this section we provide an average case analysis of the running time of the Jumping Algorithm leading to the conclusion that its expected running time is sublinear.

We assume that the string s is given as a sequence of i.i.d. random variables uniformly distributed over the alphabet Σ. According to Knuth *et al.* [17] "It might be argued that the average case taken over random strings is of little interest, since a user rarely searches for a random string. However, this model is a reasonable approximation when we consider those pieces of text that do not contain the pattern [. . .]". The experimental results we provide will show that this is indeed the case.

Let us concentrate on the behaviour of the algorithm when scanning a (piece of the) string which does not contain a match. According to the above observation we can reasonably take this as a measure of the performance of the algorithm, considering that for any match found there is an additional step of size 1, which we can charge as the cost of the output.

Let $E_{m,\sigma}$ denote the expected value of the distance between R and L, following an update of R, i.e. if L is in position i, then we are interested in the value ℓ such that $\text{FIRSTFIT}(prv(i) + q) = i + \ell$. Notice that the probabilistic assumptions made on the string, together with the assumption of absence of matches, allows us to treat this value as independent of the position i. We will show the following result about $E_{m,\sigma}$. For the sake of the clarity, we defer the proof of this technical fact to the next section.

Lemma 3. $E_{m,\sigma} = \Omega\left(m + \sqrt{m\sigma \ln \sigma}\right).$

At each iteration (when there is no match) the L pointer is moved forward to the farthest position from R such that the Parikh vector of the substring between L and R is a sub-Parikh vector of q. In particular, we can upper bound the distance between the new positions of L and R with m. Thus for the expected number of jumps performed by the algorithm, measured as the average number of times we move L, we have

$$\mathbb{E}[J] = \frac{n}{E_{m,\sigma} - m} = O\left(\frac{n}{\sqrt{m\sigma \ln \sigma}}\right). \tag{3}$$

Recalling (1) and (2), and using (3) for a random instance we have the following result concerning the average case complexity of the Jumping Algorithm.

Theorem 2. *Let $s \in \Sigma^*$ be fixed. Algorithm Jumping Algorithm finds all occurrences of a query q*

1. *in expected time $O(n(\frac{\sigma}{\log \sigma})^{1/2} \frac{\log m}{\sqrt{m}})$ using an inverted prefix table of size $O(n)$, which can be constructed in a preprocessing step in time $O(n)$;*
2. *in expected time $O(n(\frac{\sigma}{\log \sigma})^{1/2} \frac{1}{\sqrt{m}})$ using a wavelet tree of s of size $O(n)$, which can be computed in a preprocessing step in time $O(n)$.*

We conclude this section by remarking once more that the above estimate obtained by the approximating probabilistic automaton appears to be confirmed by the experiments.

The proof of Lemma 3 We shall argue asymptotically with m and according to whether or not the Parikh vector q is balanced, and in the latter case according to its degree of *unbalancedness*, measured as the magnitude of its largest and smallest components.

Case 1. q is balanced, i.e., $q = (\frac{m}{\sigma}, \ldots, \frac{m}{\sigma})$. Then, from equations (7) and (12) of [19], it follows that

$$E_{m,\sigma} \approx m + \begin{cases} m2^{-m}\binom{m}{m/2} & \text{if } \sigma = 2, \\ \sqrt{2m\sigma \ln \frac{\sigma}{\sqrt{2\pi}}} & \text{otherwise.} \end{cases} \tag{4}$$

The author of [19] studied a variant of the well known coupon collector problem in which the collector has to accumulate a certain number of copies of each coupon. It should not be hard to see that by identifying the characters with the coupon types, the random string with the sequence of coupons obtained, and the query Parikh vector with the number of copies we require for each coupon type, the expected time when the collection is finished is the same as our $E_{m,\sigma}$. It is easy to see that (4) provides the claimed bound of Lemma 3.

Case 2. $q = (q_1, \ldots, q_\sigma) \neq (\frac{m}{\sigma}, \ldots, \frac{m}{\sigma})$. Assume, w.l.o.g., that $q_1 \geq q_2 \geq \cdots \geq q_\sigma$. We shall argue by cases according to the magnitude of q_1.

Subcase 2.1. Suppose $q_1 = \frac{m}{\sigma} + \Omega\left(\sqrt{\frac{m \ln \sigma}{\sigma}}\right)$. Let us consider again the analogy with the coupon collector who has to collect q_i copies of coupons of type i, with $i = 1, \ldots, \sigma$. Clearly the collection is not completed until the q_1'th copy of the coupon of type 1 has been collected. We can model the collection of these type-1 coupons as a sequence of Bernoulli trials with probability of success $1/\sigma$. The expected waiting time until the q_1'th success is σq_1 and from the previous observation this is also a lower bound on $E_{m,\sigma}$. Thus,

$$E_{m,\sigma} \geq \sigma q_1 = \sigma \left(\frac{m}{\sigma} + \Omega\left(\sqrt{\frac{m \ln \sigma}{\sigma}}\right) \right) = \Omega\left(m + \sqrt{m\sigma \ln \sigma}\right),$$

which confirms the bound claimed, also in this case.

Subcase 2.2. Finally, assume that $q_1 = \frac{m}{\sigma} + o\left(\sqrt{\frac{m \ln \sigma}{\sigma}}\right)$. Then, for the smallest component q_σ of q we have $q_\sigma \geq m - (\sigma - 1)q_1 = \frac{m}{\sigma} - o\left(\sqrt{m\sigma \ln \sigma}\right)$. Consider now the balanced Parikh vector $q' = (q_\sigma, \ldots, q_\sigma)$. We have that $q' \leq q$ and $|q'| = \sigma q_\sigma$. By the analysis of Case 1., above, on balanced Parikh vectors, and observing that collecting q implies collecting q' also, it follows that

$$E_{m,\sigma} \geq E_{\sigma q_\sigma, \sigma}$$
$$= \Omega \left(\sigma q_\sigma + \sqrt{\sigma^2 q_\sigma \ln \sigma} \right)$$
$$= \Omega \left(\sigma \left(\frac{m}{\sigma} - o\left(\sqrt{m\sigma \ln \sigma} \right) \right) + \sqrt{\sigma^2 \left(\frac{m}{\sigma} - o\left(\sqrt{m\sigma \ln \sigma} \right) \right) \ln \sigma} \right)$$
$$= \Omega \left(m - o\left(\sigma \sqrt{m\sigma \ln \sigma} \right) + \sqrt{m\sigma \ln \sigma - o\left(\sigma^2 \ln \sigma \sqrt{m\sigma \ln \sigma} \right)} \right),$$

in agreement with the bound claimed. This completes the proof.

4.5   Simulations

We implemented the Jumping Algorithm in C++ in order to study the number of jumps J. We ran it on random strings of different lengths and over different alphabet sizes. The underlying probability model is an i.i.d. model with uniform distribution. We sampled random query vectors with length between $\log n$ ($= \log_2 n$) and $\sqrt{n}$, where n is the length of the string. Our queries were of one of two types:

1. Quasi-balanced Parikh vectors: Of the form $(q_1, \ldots, q_\sigma)$ with $q_i \in (x - \epsilon, x + \epsilon)$, and x running from $\log n / \sigma$ to $\sqrt{n}/\sigma$. For simplicity, we fixed $\epsilon = 10$ in all our experiments, and sampled uniformly at random from all quasi-balanced vectors around each x.
2. Random Parikh vectors with fixed length m. These were sampled uniformly at random from the space of all Parikh vectors with length m.

The rationale for using quasi-balanced queries is that those are clearly worst-case for the number of jumps J, since J depends on the shift length, which in turn depends on FIRSTFIT($prv(L) + q$). Since we are searching in a random string with uniform character distribution, we can expect to have minimal FIRSTFIT($prv(L) + q$) if q is close to balanced, i.e. if all entries q_i are roughly the same. This is confirmed by our experimental results which show that J decreases dramatically if the queries are not balanced (Fig. 7, right).

We ran experiments on random strings over different alphabet sizes, and observe that our average case analysis agrees well with the simulation results for random strings and random quasi-balanced query vectors. Plots for $n = 10^5$ and $n = 10^6$ with alphabet sizes $\sigma = 2, 4, 16$ resp. $\sigma = 4, 16$ are shown in Fig. 6.

In Fig. 5 we show comparisons between the running time of the Jumping algorithm and that of the simple window algorithm. The simulations over random strings and Parikh vectors of different sizes appear to perfectly agree with the guarantees provided by our asymptotic analyses. This is of particular importance from the point of view of the applications, as it shows that the complexity analysis does not hide big constants.

To see how our algorithm behaves on non-random strings, we downloaded human DNA sequences from GenBank [12] and ran the Jumping Algorithm with random quasi-balanced queries on them. We found that the algorithm performs 2 to 10 times
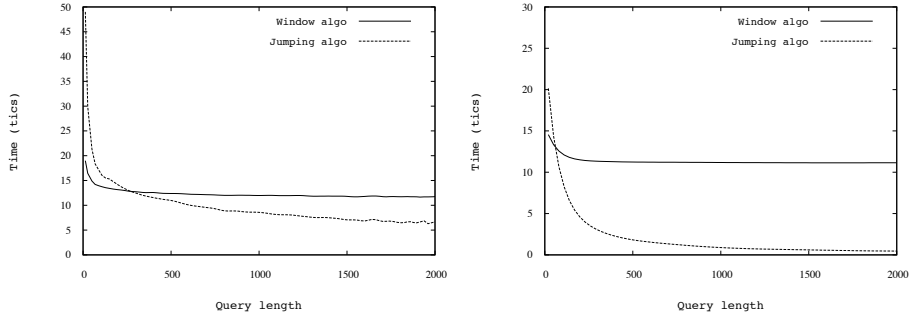
Fig. 5. Running time comparisons between the Jumping Algorithm and the window algorithm. The text is a random string (uniform i.i.d.) of size 9000000 from a four letter alphabet. Parikh vectors of different sizes between 10 and 2000 were randomly generated and the results averaged over all queries of the same size. On the left are the results for quasi-balanced Parikh vectors (cf. text). On the right are the results for random Parikh vectors.

fewer jumps on these DNA strings than on random strings of the same length, with the gain increasing as n increases. We show the results on a DNA sequence of 1 million bp (from Chromosome 11) in comparison with the average over 10 random strings of the same length (Fig. 7, left).
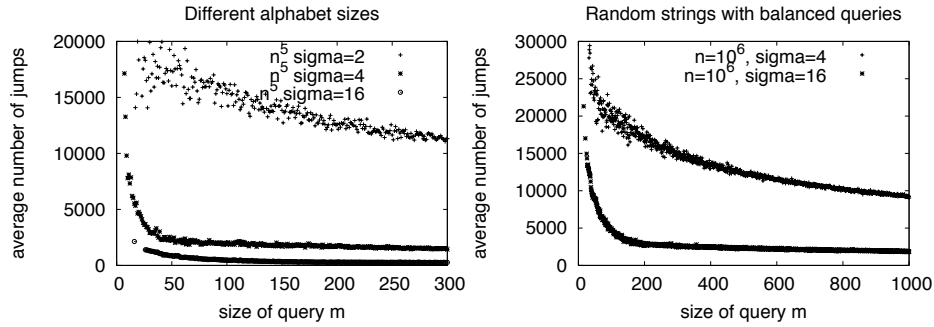

Fig. 6. Number of jumps for different alphabet sizes for random strings of size 100 000 (left) and 1 000 000 (right). All queries are randomly generated quasi-balanced Parikh vectors (cf. text). Data averaged over 10 strings and all random queries of same length.

5   Conclusion

Our simulations appear to confirm that in practice the performance of the Jumping Algorithm is well predicted by the average case analysis we proposed. A more precise
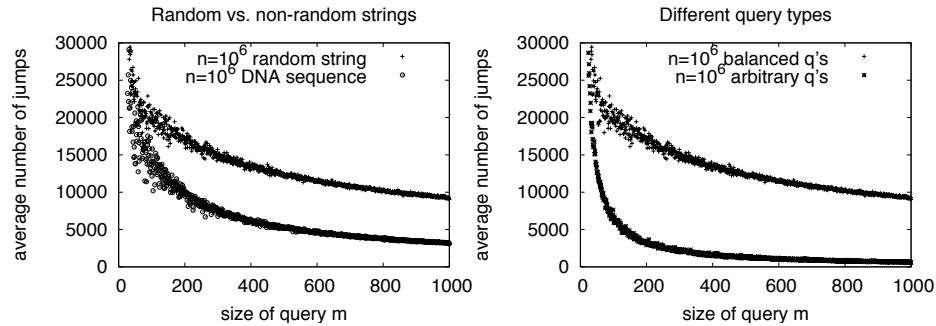
Fig. 7. Number of jumps in random vs. nonrandom strings: Random strings over an alphabet of size 4 vs. a DNA sequence, all of length 1 000 0000, random quasi-balanced query vectors. Data averaged over 10 random strings and all queries with the same length (left). Comparison of quasi-balanced vs. arbitrary query vectors over random strings, alphabet size 4, length 1 000 000, 10 strings. The data shown are averaged over all queries with same length m (right).

analysis is needed, however. Our approach seems unlikely to lead to any refined average case analysis since that would imply improved results for the intricate variant of the coupon collector problem of [19].

Moreover, in order to better simulate DNA or other biological data, random string models other than uniform i.i.d. should also be analysed, such as first or higher order Markov chains.

We remark that our wavelet tree variant of the Jumping Algorithm, which uses rank/select operations only, opens a new perspective on the study of Parikh vector matching. We have made another family of approximate pattern matching problems accessible to the use of self-indexing data structures [21]. We are particularly interested in compressed data structures which allow fast execution of rank and select operations, while at the same time using reduced storage space for the text. Thus, every step forward in this very active area can provide improvements for our problem.

Acknowledgements

References

1. A. Amir, A. Apostolico, G. M. Landau, and G. Satta. Efficient text fingerprinting via Parikh mapping. *J. Discrete Algorithms*, 1(5-6):409–421, 2003.
2. A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.
3. G. Benson. Composition alignment. In *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI'03)*, pages 447–461, 2003.

4. S. Böcker. Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt. *Journal of Computational Biology*, 11(6):1110–1134, 2004.
5. S. Böcker. Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. *Bioinformatics*, 23(2):5–12, 2007.
6. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
7. A. Butman, R. Eres, and G. M. Landau. Scaled and permuted string matching. *Inf. Process. Lett.*, 92(6):293–297, 2004.
8. M. Cieliebak, T. Erlebach, Zs. Lipták, J. Stoye, and E. Welzl. Algorithmic complexity of protein identification: combinatorics of weighted strings. *Discrete Applied Mathematics*, 137(1):27–46, 2004.
9. D. Clark. *Compact pat trees*. PhD thesis, University of Waterloo, Canada, 1996.
10. R. Eres, G. M. Landau, and L. Parida. Permutation pattern discovery in biosequences. *Journal of Computational Biology*, 11(6):1050–1060, 2004.
11. W. Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, 1968.
12. Website. http://www.ncbi.nlm.nih.gov/Genbank/.
13. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.
14. R. N. Horspool. Practical fast searching in strings. *Softw., Pract. Exper.*, 10(6):501–506, 1980.
15. P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.
16. S. Jukna. *Extremal Combinatorics*. Springer, 1998.
17. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
18. M. Lothaire. *Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, New York, NY, USA, 2005.
19. R. May. Coupon collecting with quotas. *Electr. J. Comb.*, 15, 2008.
20. J. I. Munro. Tables. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, pages 37–42, 1996.
21. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.
22. A. Salomaa. Counting (scattered) subwords. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 81:165–179, 2003.
23. W. F. Smyth. *Computing Patterns in Strings*. Pearson Addison Wesley (UK), 2003.