On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching*

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. Given a string s, the Parikh vector of s, denoted $p(s)$, counts the multiplicity of each character in s. Searching for a match of Parikh vector q (a "jumbled string") in the text s requires to find a substring t of s with $p(t) = q$. The corresponding decision problem is to verify whether at least one such match exists. So, for example for the alphabet $\Sigma = \{a, b, c\}$, the string $s = abaccbabaaa$ has Parikh vector $p(s) = (6, 3, 2)$, and the Parikh vector $q = (2, 1, 1)$ appears once in s in position $(1, 4)$. Like its more precise counterpart, the renown Exact String Matching, Jumbled Pattern Matching has ubiquitous applications, e.g., string matching with a dyslectic word processor, table rearrangements, anagram checking, Scrabble playing and, *allegedly*, also analysis of mass spectrometry data. We consider two simple algorithms for Jumbled Pattern Matching and use very complicated data structures and analytic tools to show that they are not worse than the most obvious algorithm. We also show that we can achieve non-trivial efficient average case behavior, but that's less fun to describe in this abstract so we defer the details to the main part of the article, to be read at the reader's risk... well, at the reader's discretion.

1 Prologue

Last month, I happened to organize a workshop at my university. We ended up being 20 people, so, for a little social dinner, I decided to call a friend who owns a restaurant in town, and asked him to prepare a table for 20 people.

My friend, who always likes to make jokes, decided to tease me a bit and on arrival at the restaurant we found a table laid for 100 people. However, instead of having exactly one fork, one knife, and one spoon at each plate, my friend

* Part of this work was done while F.C. and Zs.L. were visiting the Alfréd Rényi Institute of Mathematics in Budapest, Hungary, within the EU Marie Curie Transfer of Knowledge project "Hungarian Bioinformatics (HUBI)."

P. Boldi (Ed.): FUN 2010, LNCS 6099, pp. 89–101, 2010.

had put all the 300 pieces of cutlery 3 by 3 but completely at random. I was about to faint. What would my scholarly friends think of me! What a horrible impression they would get of my country's hospitality! So I hurried to my friend and told him to solve the problem immediately. I said, "unless you find a way to have us all sit next to each other, we are all going to McDonald's!"

My friend got pale at the prospect of losing 20 customers, but did not lose his spirit. He knew we were computer scientists, so, "in order to speed up things," he said, "please quickly check whether there are 20 consecutive places where you can find 20 knives, 20 forks and 20 spoons, and I will proceed to rearrange them properly." Of course we found the task amusing, and we ended up spending the rest of the evening discussing the following article ....

2 Definitions and Problem Statement

Caveat: Since this is a *fun* paper on jumbled pattern matching, the paper itself is also jumbled. Readers who prefer a more classic structure are advised to read Section 6 first, which details motivations and related work.

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 [11]. Such space usage is unacceptable 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 [10], an algorithm (Interval Algorithm) was presented which solved the decision problem on binary alphabets in constant time per query, using linear storage space. However, it needed $\Theta(n^2)$ time for the preprocessing phase. In this section, we improve that preprocessing time to $O(n^2/\log n)$. We first recall the Interval Algorithm, which makes use of the following property of binary strings:

Lemma 1 ([10], Lemma 3). *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 $x_1 \leq y \leq x_2$.*

This means that the Parikh vectors of substrings of s of length m build a set of the form $\{(x, m-x) \mid x = \mathrm{pmin}(m), \mathrm{pmin}(m)+1, \ldots, \mathrm{pmax}(m)\}$ for appropriate $\mathrm{pmin}(m)$ and $\mathrm{pmax}(m)$. The algorithm computes these values in a preprocessing step; then, when a query $q = (x, y)$ arrives, it answers *yes* if and only if $x \in [\mathrm{pmin}(x + y), \mathrm{pmax}(x + y)]$. The query time is $O(1)$. We now show how to reduce the preprocessing problem to a (min,+)-convolution problem, giving subquadratic running time.

Let $\underline{x} = (x_0, x_1, \ldots, x_n)$ and $\underline{y} = (y_0, y_1, \ldots, y_n)$ be two real vectors. The (min,+)-convolution, minimum convolution or simply min-convolution of $\underline{x}$ and $\underline{y}$ is the vector $\underline{z} = \underline{x} \star \underline{y} = (z_0, z_1, \ldots, z_{2n})$ with $z_m = \min(x_i + y_{m-i})$, where the minimum is taken over all possible values of i . (Standard convolution or $(+, \cdot)$-convolution is obtained if min is replaced by $\sum$ and $x_i + y_{m-i}$ by $x_i \cdot y_{m-i}$.) We note that there are variants of the definition, e.g. the index $m - i$ is sometimes understood mod n or $\underline{z}$ is of the same length as $\underline{x}$ and $\underline{y}$. These are easily reducible to each other.

Min-convolution has first been used in optimization problems [3], but it also has applications in computer vision and signal processing [2], sequence alignment [13] and sequential data analysis [15]. The currently known best algorithm (in worst-case sense and in the RAM model) was introduced in [7] and runs in slightly subquadratic time: $O(n^2/\log n)$. This algorithm reduces min-convolution to a problem in computational geometry, that of finding *dominating pairs*, which is discussed and analyzed in [9].

We briefly describe how the problem of determining the values $\mathrm{pmin}(m)$ and $\mathrm{pmax}(m)$ for $m = 1, 2, \ldots, n$ can be reduced to min-convolution. Let s be a string of length n, and let x_i be the number of characters a in $pr(i)$, $y_{n-i} = -x_i$ for $i = 0, 1, \ldots, n$. Then

$$\mathrm{pmin}(m) = \min_{i=0}^{n-m}(x_{i+m} - x_i) = \min_{i=0}^{n-m}(x_{i+m} + y_{n-i}) = z_{n+m} \quad , \tag{1}$$

where $\underline{z} = \underline{x} \star \underline{y}$. The maximum can be calculated analogously.

In order to make the presentation self-contained, we also describe the problem of finding dominating pairs, and how min-convolution can be solved by it. For the analysis of running times, we refer to [9,7].

An instance of the dominating pairs problem consists of a red set and a blue set of points in d-dimensional space as input, and asks for all pairs of points (α, β), where α is red and β is blue and $\alpha \leq \beta$ componentwise, that is $\alpha_i \leq \beta_i$ for $i = 1, 2, \ldots, d$. Chan [9] solves the problem with the following divide-and-conquer algorithm. If there is only one point, there's nothing to do. If $d = 0$, then all red-blue pairs are dominating. Otherwise calculate the median ζ of the d-th coordinates of all points and divide red and blue points into left and right sets R_l, R_r and B_l, B_r according to the relationship of their d-th coordinates to ζ. Finally, solve the problem recursively for $R_l \cup B_l$, $R_r \cup B_r$ and for the projection of $R_l \cup B_r$ to the first $d-1$ coordinates. It turns out that for any $\varepsilon \in (0, 1)$ the running time of the algorithm is $O(c_\varepsilon^d n^{1+\varepsilon} + D)$, where n is the number of input points, D is the number of dominating pairs, and $c_\varepsilon = 2^\varepsilon/(2^\varepsilon - 1)$.

Let $\underline{x}$ and $\underline{y}$ be two vectors whose min-convolution has to be computed. Fix d (to be defined later). For each $\delta \in \{0, 1, \ldots, d-1\}$ we define a dominating pairs problem for the following set of red and blue points:

$$R = \{\alpha_i = (x_{i+\delta} - x_i, x_{i+\delta} - x_{i+1}, \ldots, x_{i+\delta} - x_{i+d-1}) \mid i = 0, d, \ldots, \lfloor n/d \rfloor d\}$$
$$B = \{\beta_j = (y_j - y_{j-\delta}, y_{j-1} - y_{j-\delta}, \ldots, y_{j-d+1} - y_{j-\delta}) \mid j = 0, 1, \ldots, n\}$$

For indices out of range, take the components of x and y to be ∞. We get a dominating pair (α_i, β_j) if and only if $x_{i+\delta} + y_{j-\delta} \leq x_{i+k} + y_{j-k}$ for $k = 0, 1, \ldots, d-1$. We collect all dominating pairs for all δ. Then, in the min-convolution calculation of z_{i+j} only those values $x_{i+\delta} + y_{j-\delta}$ have to be considered that come from a dominating pair. For each pair (i, j) (i a multiple of d and j arbitrary), only one such δ exists[1], therefore we gain a factor of d in the calculation of the minima. Choosing $d = \log_2 n/2$ gives an overall running time of $O(n^2/\log n)$.

Example 2. Let the two characters of the alphabet be a and b, and let $s = aabba$. Then $\underline{x} = (0, 1, 2, 2, 2, 3)$, $\underline{y} = (-3, -2, -2, -2, -1, 0)$. We agree on breaking ties the following way: if $x_i + y_j = x_{i'} + y_{j'}$, $i + j = i' + j'$, then $x_i + y_j$ is considered smaller if and only if $i < i'$. For the presentation we choose $d = 3$. For $\delta = 0$ we list the red and blue sets $R = \{(0, -1, -2), (0, 0, -1)\}$ and $B = \{(0, \infty, \infty), (0, -1, \infty), (0, 0, -1), (0, 0, 0), (0, -1, -1), (0, -1, -2)\}$. The

[1] We assume that ties are broken in a consistent manner.

dominating pairs $(i+\delta, j-\delta)$ are $\{(0,0), (0,1), (0,2), (0,3), (0,4), (0,5), (3,0), (3,2), (3,3)\}$. For $\delta = 1$ the dominating $(i+\delta, j-\delta)$ are $\{(4,0), (4,3), (4,4), (4,5)\}$ and for $\delta = 2$, $\{(5,5)\}$. We then calculate $\underline{z}$ based on these pairs. For example, $z_5 = \min\{x_0 + y_5, x_3 + y_2\} = 0$ and $z_7 = \min\{x_4 + y_3\} = 0$. Thus, for the computation of $\text{pmin}(2) = z_7$, we needed to consider only 1 sum instead of 4 acc. to 1, namely $z_7 = \min\{x_2 + y_5, x_3 + y_4, x_4 + y_3, x_5 + y_2\}$. We get $\underline{z} = (-3, -2, -2, -2, -1, 0, 0, 0, 1, 2, 3)$. We conclude that $\text{pmin}(m) = 0, 0, 1, 2, 3$ for $m = 1, 2, 3, 4, 5$. Note that if needed, the corresponding positions can be obtained.

In [10] it was conjectured that no subquadratic algorithm for this preprocessing phase exists. Informally, the reason to believe so was that if one computes the minimum for a single value m, the optimal running time is trivially linear (because one has to read the string), and knowing the minimum for one value does not give any useful information for the others. There is, however, some locality in the problem: if the substring $s_i \dots s_{i+m-1}$ has at least $\text{pmin}(m)+2$ characters equal to a, then it is not the position for a minimal substring of length $m+1$ either. This is the kind of locality that is exploited in the above algorithm. The logarithmic gain is too small for most practical applications, however. It remains open if an $O(n^{2-\varepsilon})$ algorithm exist for the calculation of pmin and pmax.

4 General Alphabets

An algorithm for the general case was presented in [10], whose expected running time was shown to be $O(n(\frac{\sigma}{\log \sigma})^{1/2}\frac{\log m}{\sqrt{m}})$, using $O(n)$ space, with preprocessing time $O(n)$. In this section, we show how to use wavelet trees to implement this algorithm, and thus improve the average runtime to $O(n(\frac{\sigma}{\log \sigma})^{1/2}\frac{1}{\sqrt{m}})$, making it competitive with the window algorithm as soon as $m = \omega(\sigma/\log\sigma)$, i.e. in practically all cases. Space requirements and preprocessing time remain the same.

4.1 The Jumping Algorithm

We give a brief explanation of the algorithm. For an illustration and the pseudocode, refer to Figs. 1 and 2.

Recall that $prv(j) = p(s_1 \dots s_j)$ is the Parikh vector of the prefix of s of length j. The algorithm makes use of the simple observation that q has an occurrence at position $(i+1, j)$ if and only if $prv(j) - prv(i) = q$. Imagine moving two pointers L and R along s, which point to these potential positions i and j. We alternate in updating L and R: In each update, either L or R is moved to the right. The invariant is that $p(R-L) \geq q$ after each update of R, and $p(R-L) \leq q$ after each update of L (Lemma 2 of [10]). We need the following function:

$$\text{FIRSTFIT}(p) := \min\{j \mid prv(j) \geq p\}. \tag{2}$$

The update rules are as follows:

- update R: Both $prv(L)$ and q must fit before the new positions of R, so move R to the first index j s.t. $prv(j) \geq prv(L)+q$. So $R \leftarrow \text{FIRSTFIT}(prv(L)+q)$.
- update L: $L \leftarrow L+1$ if a match was found, else:
  Some characters of $p(R-L)$ are unnecessary and have to be accommodated before the new position of L: $L \leftarrow \text{FIRSTFIT}(prv(R)-q)$.

After each update of R or L, we check whether there is a match *by checking whether* $R-L=|q|$. This is correct due to the invariants above. So no matching or accessing the string s is ever done. The complexity of the algorithm depends on how the functions $prv(j)$ and FIRSTFIT are implemented.

Example 3. Consider the string $s = bbacaccababbabccaaac$ and query $q = (3,1,2)$, which has 3 occurrences in s, namely $(5,10)$, $(13,18)$, and $(14,19)$. R assumes the values $8, 10, 13, 18, 19$, and thus, the while-loop is executed 5 times (see Fig. 2).

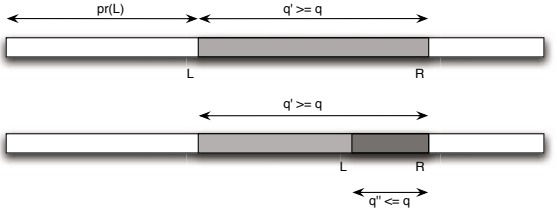

Fig. 1. The situation after the update of R (above) and after the update of L (below). R is placed at the first fit of $prv(L)+q$, thus q' is a super-Parikh vector of q. Then L is placed at the beginning of the longest good suffix ending in R, so q'' is a sub-Parikh vector of q.

4.2 Improved Running Time Using Wavelet Trees

In [10] we used an inverted table for computing the two functions $prv(j)$ and FIRSTFIT. It 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. Here we replace this data structure with a wavelet tree [17].

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 $p = (p_1, \ldots, p_\sigma)$, $\text{FIRSTFIT}(p) = \max_{k=1,\ldots,\sigma}\{select_k(p_k)\}$.

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.

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 ← |q|; Occ ← ∅; L ← 0;
2.  while L < n − m
3.     do R ← FIRSTFIT(prv(L) + q);
4.        if R − L = m
5.          then add L + 1 to Occ;
6.               L ← L + 1;
7.          else L ← FIRSTFIT(prv(R) − q);
8.               if R − L = m
9.                 then add L + 1 to Occ;
10.                     L ← L + 1;
11. return Occ;
```

Fig. 2. Pseudocode of Jumping Algorithm

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. 3. 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 ([21,12,22]).

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 $x_u = \max\{select_0(B_u, x_{\text{left}}), select_1(B_u, x_{\text{right}})\}$. The desired value is equal to x_{root}.

Example 4. Let s = *bbacaccabaddabccaaac* (cp. Fig. 3). 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 $\max\{select_0(B_{a,b,c,d}, 6), select_1(B_{a,b,c,d}, 4)\} = \max\{9, 11\} = 11$.

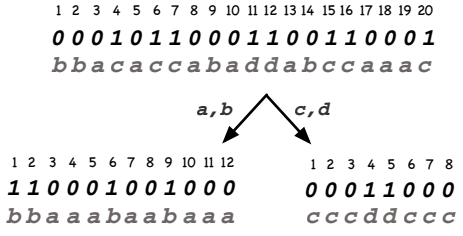

Fig. 3. The wavelet tree for string *bbacaccabaddabccaaac*. For clarity, the leaves have been omitted. Note also that the third line at each inner node (the strings over the alphabet $\{a, b, c, d\}$) are only included for illustration.

Analysis: Let J denote the number of times the while-loop in line 2 (see Fig. 2) is executed. The work done in each iteration is dominated by the computation of FIRSTFIT($prv(L + q)$) (line 3) and FIRSTFIT($prv(R - q)$) (line 7). Since both FIRSTFIT and prv can be computed in $O(\sigma)$ time, we have that the total runtime of the algorithm is $O(J\sigma)$. Since in every iteration, R is moved forward, $J = O(n)$; however, since there are cases where $J = n/2$, such as searching for $(2, 0)$ on string $(ab)^n$, we also have $J = \Theta(n)$. So worst-case running time of the Jumping Algorithm is $\Theta(\sigma n)$, i.e., slower than the window algorithm.

In [10], the expected value of J was shown to be $\mathbb{E}(J) = O(\frac{n}{\sqrt{m}\sqrt{\sigma \log \sigma}})$ on uniformly distributed strings and queries. This yielded an expected runtime of $O((\frac{\sigma}{\log \sigma})^{1/2} \frac{\log m}{\sqrt{m}} n)$ using the inverted prefix table. With the wavelet tree implementation, we get expected runtime $O((\frac{\sigma}{\log \sigma})^{1/2} \frac{1}{\sqrt{m}} n)$, i.e., we save a $\log m$ factor. Moreover, the new version is sublinear as soon as $m = \omega(\sigma / \log \sigma)$. Preprocessing is still linear in n, since the wavelet tree of s can be constructed in $O(n)$ time.

The space required by the wavelet tree is $\lceil \log \sigma \rceil (n + o(n))$ *bits*, since each level requires $n + o(n)$ bits. Our previous implementation needed $n \log n$ bits for the inverted prefix table. With the usual assumptions on the RAM model, namely that $\log n$ fits into a computer word, both are $O(n)$; however, on a bit level, we improve.

5 Variations: Sub-Parikh Vectors and Scrabble-Like Games

Consider the following game, which can be played using the Scrabble set you probably have at home. The only other thing you need is a random text, maybe from a newspaper or from the Internet. Each player draws 8 letters from the sack of letters. When it is her turn, she aligns a word made with her letters to a position in the text, trying to maximize the total score, which is the sum of the scores of the individual letters used. Then she draws new letters from the sack, until she has again 8 letters in front of her. Overlapping alignments are allowed, so the same position in the text can be matched more than once. The game ends when all letters finish, or when no player can move, and is won by the player with the highest total score.

If we refer to the player's current Parikh vector (the contents of the tray) as q, then the task is to find a jumbled match of a sub-Parikh vector q' of q to the text, under the constraint that the substring matched be a word of English. Moreover, we want to maximize the score of q'.

Note that in the game described above, maximizing the score in a single move does not necessarily result in an optimal game strategy, since you might save a letter for the next round if a lucky new letter allows for a much higher score. However, for our purposes, we will assume the simple strategy of maximizing in each move. Further, let us drop the constraint of the substring being a word of English—these simplifications will render the game more tractable (in the non-technical sense).

So, given a weight function $w : \Sigma \to \mathbb{R}^+$ on the characters (the scores), we have the following problem:

> **Jumbled sub-Pattern Matching (JSPM).** Let $s \in \Sigma^*$ be given, $|s| = n$. For a Parikh vector $q \in \mathbb{N}^\sigma$ (the query), $|q| = m$, find all occurrences q' in s s.t. $q' \leq q$ with maximum weight.

Like the original JPM also this variant can be solved in linear time (in n) with the following simple variant of the window-algorithm. Slide a variable sized window over s, keeping the Parikh vector of the current window content in a vector c. Start with both pointers L and R pointing to the beginning of s, so $c = (0, 0, ..., 0)$. While $c \leq q$, move R towards the right one character at a time. Now make a note of $w(c)$, the weight of the current vector, increment R, and move L to the right until again $c \leq q$. Move again R as long as $c \leq q$. Now check the value of $w(c)$ against the previous value, replace if greater, and so on.

Clearly the above procedure finds all maximal (non-extendable) sub-Parikh vectors q' of q that occur in s, and thus also those with maximum $w(q')$.

Naturally, the reader might wonder whether the Jumping Algorithm might be also adapted to this optimization problem. In fact, it is not hard to come up with a simple variant of the Jumping Algorithm which solves JSPM.

First note that the movement of L is exactly as in the Jumping Algorithm, namely $L \leftarrow \text{FIRSTFIT}(prv(R) - q)$. Next, define $\text{TIFTSRIF}(p)$ as $\text{FIRSTFIT}(p)$ on the reverse string s^{rev}, and $vrp(i)$ as the prefix Parikh vector of position $n-i+1$ in the reverse string! Then, R is updated to $\text{TIFTSRIF}(vrp(L)-q)$. This is because R is now moved the same way as L usually is, but from the back: R is moved to the position furthest from L such that the current window is still a sub-Parikh vector of q. After each such jump of R, check $w(c)$ and keep track of maximum so far. If all occurrences are needed, make note of positions, too. So the loop is: $R{+}{+}$, update L, update R, check and maybe update $w(c)$.

For the implementation, we need a way to compute TIFTSRIF and vrp, which can be easily achieved without increasing time or space complexity by using a second wavelet tree for the reverse string. However, this still results in doubling the storage space. Instead, one can implement these functions using only the wavelet tree of the string itself by noting that $\text{TIFTSRIF}(vrp(L) - q) = \min_a\{select_a(prv(L)_a + q_a + 1)\} - 1$, i.e., R is updated to the left of the first position where the allowed number of characters a is exceeded for at least one a. We can compute $prv(L)$ first in $O(\sigma)$ time, and then compute the above minimum bottom-up, also in $O(\sigma)$ time, analogously to the maximum for FIRSTFIT.

More delicate is the question of whether the above variant of the Jumping Algorithm is also competitive in running time with the window algorithm. As we will immediately see, in the worst case there is again an additional factor σ.

Again each iteration of the loop needs $O(\sigma)$ time. So we have $O(\sigma J)$, with J denoting the number of iterations of the while-loop. Since R and L are always incremented, we have $J \leq n$, thus the running time is $O(\sigma n)$. In fact, it is not hard to see that this is also a lower bound: Consider the case $\Sigma = \{a, b, c\}$, with $s = a^{n-2}bc$ and $q = (0, 1, 1)$.

5.1 Average Case: Skewed Distributions and Skewed Patterns Help

Assume that each character of the string s is generated independently according to some fixed probability distribution $\mu = (\mu_1, \ldots, \mu_\sigma)$, where the probability of seeing character a_k is μ_k, for $k = 1, \ldots, \sigma$.

Let $q = (x_1 - 1, \ldots, x_\sigma - 1)$ be the pattern Parikh vector, thus $1 < x_k$, for each $k = 1, \ldots, \sigma$. Let T_j be the random variable which takes value i if the Parikh vector of the substring $s_{j+1}s_{j+2}\ldots s_{j+i-1}$ is a sub-Parikh vector of q but $p(s_{j+1}\ldots s_{j+i})$ is not. Note that this means that i is the first position where one of the entries of q is exceeded. Because of the i.i.d. model we chose for the generation of the string s, it follows that $T_i \sim T_j$ and thus $\mathbb{E}[T_i] = \mathbb{E}[T_j]$, for any i, j. So we can use $\mathbb{E}[T]$, making explicit the independence from the position in the string.

By the results in [19,20] (see also [16]) we have that, if $x_k >> \mu_k$ for $k = 1, \ldots, \sigma$, then (asympotically with $|q|$),

$$\mathbb{E}[T] \approx \min_{k=1,\ldots,\sigma} \frac{x_k}{\mu_k}. \tag{3}$$

We can recast the above problem in a generalization of the classical Problem of the Points, originating from a correspondence between Pascal and Fermat or, equivalently, of the Banach match-box problem: Given is a die with σ faces, where the k'th face appears with probability μ_k. There are σ players, and player k gets a point if the k'th face comes up and wins as soon as she accumulates x_k points. The question is to find the expected number of tosses before the game ends (i.e., some player wins). The distribution of the finishing time for each player, i.e., the time when she has accumulated the required number of points, follows the negative binomial distribution, with expectation given by $\frac{x_k}{\mu_k}$. The results mentioned above and summarized by Eq. (3) are to the effect that the expected time when some player wins is asymptotically equal to the minimum of the expected victory time of the individual players.

We can use this to provide a bound on the expected running time of our algorithm. First assume that the minimum in (3) is attained by exactly one k^*. Then, $k^* = \operatorname{argmin}_{k=1,\ldots,\sigma} \frac{x_k}{\mu_k}$, and we have that the expected position where the algorithm places the pointer R is given by $\mathbb{E}[T] = \frac{x_{k^*}}{\mu_{k^*}}$. Moreover, $s_R = a_{k^*}$ and $prv(R) - prv(L)$ contains exactly $x_{k^*} - 1$ many a_{k^*}'s.

Now pointer L is moved towards the right until the first new a_{k^*} is encountered. The average length of this displacement is $\frac{1}{\mu_{k^*}}$. Then the right pointer R is moved again to the furthest position from L such that the Parikh vector of the string between L and R is a sub-Parikh vector of q, and then incremented by 1. It is the difference between this new position of the right pointer and its previous position which is significant for our analysis. Because of the assumption of an i.i.d. model, the expected new position of the pointer R given by $\mathbb{E}[T_L]$ is equal to $\mathbb{E}[T] = \frac{x_{k^*}}{\mu_{k^*}}$. Thus the pointer R jumps by $\frac{1}{\mu_{k^*}}$. The same argument is clearly also valid for the following jumps.

If there is more than one index attaining the minimum in (3), then set $k^* = \operatorname{argmax}_l\{\mu_l \mid \frac{x_l}{\mu_l} = \min_{k=1,\ldots,\sigma} \frac{x_k}{\mu_k}\}$. Then the above analysis goes through as an upper bound on the expected number of jumps. The above discussion leads to:

Proposition 5. *Let s be a randomly generated string over $\Sigma = \{a_1, \ldots, a_\sigma\}$ such that $Pr(s_i = a_k) = \mu_k$, for each $i = 1, \ldots, n$ and $k = 1, \ldots, \sigma$. Let $q = (x_1 - 1, \ldots, x_\sigma - 1)$ be the query Parikh vector, and let $k^* = \operatorname{argmax}_l\{\mu_l \mid \frac{x_l}{\mu_l} = \min_{k=1,\ldots,\sigma} \frac{x_k}{\mu_k}\}$. Then the expected running time of the Jumping Algorithm on this instance of the JSPM problem is $O(n\sigma\mu_{k^*})$.*

6 Epilogue

Armed with our solutions, we quickly solved our table rearrangement problem, had a very satisfying dinner and FUN. The next day, a famous colleague who had enjoyed the restaurant's wine assortment the most, phoned me to communicate the following—I am quoting literally, since I haven't checked, trusting the everlasting "*In vino veritas*" [Pliny the Elder, *Naturalis historia* 14, 141].

Applications of our algorithms, apart from Scrabble and table cutlery arrangement, can be found in molecular biology, notably in interpretation of mass spectrometry data. The output of an experiment consists of the molecular masses

of sample molecules, whose molecular composition (e.g., the multiplicities of the different amino acids for proteins, or of nucleotides for DNA) can in certain cases be determined efficiently up to a few candidates [6]. In other words, several candidate Parikh vectors can be computed, and then those matched against a database of sequences. Parikh vectors have also been used in other bioinformatics applications, among them alignment [4] (there referred to as compositions), SNP discovery [5] (compomers), repeated pattern discovery [14] and gene clusters [23] (permuted patterns, πpatterns).

Jumbled pattern matching is a special case of approximate pattern matching. It has been used as filtering step in approximate pattern matching algorithms [18], but rarely considered in its own right.

Three of the authors of the present paper described two algorithms for JPM in [10], Interval Algorithm and Jumping Algorithm, both of which have been crucially improved here. The authors of [8] presented 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 other algorithm that has been presented for the problem we treated 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 [14] 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 there are strings with quadratic $P(s)$ [11].

6.1 Postscriptum (for Those Who Read the Technical Parts)

On the agenda for future work is refining the analysis of the Jumping Algorithm and the preprocessing time for the Interval Algorithm.

We remark that our new implementation of the Jumping Algorithm using 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 [22]. We are in particular 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. Thanks to Gonzalo Navarro for fruitful discussions.

References

1. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via Parikh mapping. J. Discrete Algorithms 1(5-6), 409–421 (2003)
2. Babai, L., Felzenszwalb, P.F.: Computing rank-convolutions with a mask. ACM Trans. Algorithms 6(1), 1–13 (2009)
3. Bellman, R., Karush, W.: Mathematical programming and the maximum transform. Journal of the Soc. for Industrial and Applied Math. 10(3), 550–567 (1962)
4. Benson, G.: Composition alignment. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS (LNBI), vol. 2812, pp. 447–461. Springer, Heidelberg (2003)
5. Böcker, S.: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. Bioinformatics 23(2), 5–12 (2007)
6. Böcker, S., Lipták, Z.: A fast and simple algorithm for the Money Changing Problem. Algorithmica 48(4), 413–432 (2007)
7. Bremner, D., Chan, T.M., Demaine, E.D., Erickson, J., Hurtado, F., Iacono, J., Langerman, S., Taslakian, P.: Necklaces, convolutions, and $X + Y$. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 160–171. Springer, Heidelberg (2006)
8. Butman, A., Eres, R., Landau, G.M.: Scaled and permuted string matching. Inf. Process. Lett. 92(6), 293–297 (2004)
9. Chan, T.M.: All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. Algorithmica 50(2), 236–243 (2008)
10. Cicalese, F., Fici, G., Lipták, Z.: Searching for jumbled patterns in strings. In: Proc. of the Prague Stringology Conference 2009, pp. 105–117 (2009)
11. Cieliebak, M., Erlebach, T., Lipták, Z., Stoye, J., Welzl, E.: Algorithmic complexity of protein identification: combinatorics of weighted strings. Discrete Applied Mathematics 137(1), 27–46 (2004)
12. Clark, D.: Compact pat trees. PhD thesis, University of Waterloo, Canada (1996)
13. Eppstein, D.A.: Efficient algorithms for sequence analysis with concave and convex gap costs. PhD thesis, New York, NY, USA (1989)
14. Eres, R., Landau, G.M., Parida, L.: Permutation pattern discovery in biosequences. Journal of Computational Biology 11(6), 1050–1060 (2004)
15. Felzenszwalb, P.F., Huttenlocher, D.P., Kleinberg, J.M.: Fast algorithms for large-state-space HMMs with applications to web usage analysis. In: Thrun, S., Saul, L.K., Schölkopf, B. (eds.) NIPS. MIT Press, Cambridge (2003)
16. Goczyła, K.: The generalized Banach match-box problem: Application in disc storage management. Acta Applicandae Mathematicae 5, 27–36 (1986)
17. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850 (2003)
18. Jokinen, P., Tarhio, J., Ukkonen, E.: A comparison of approximate string matching algorithms. Software Practice and Experience 26(12), 1439–1458 (1996)
19. Mendelson, H., Pliskin, J., Yechiali, U.: Optimal storage allocation for serial files. Communications of the ACM 22, 124–130 (1979)
20. Mendelson, H., Pliskin, J., Yechiali, U.: A stochastic allocation problem. Operations Research 28, 687–693 (1980)
21. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
22. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. 39(1) (2007)
23. Parida, L.: Gapped permutation patterns for comparative genomics. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 376–387. Springer, Heidelberg (2006)