Theoretical Computer Science 525 (2014) 130–137

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Approximating the maximum consecutive subsums of a sequence

Ferdinando Cicalese [a], Eduardo Laber [b], Oren Weimann [c,*], Raphael Yuster [d]

[a] *Department of Computer Science, University of Salerno, Italy*
[b] *Department of Informatics, PUC-Rio, Rio de Janeiro, Brazil*
[c] *Department of Computer Science, University of Haifa, Israel*
[d] *Department of Mathematics, University of Haifa, Israel*

ARTICLE INFO

Dedicated to Professor Gad M. Landau, on the occasion of his 60th birthday

Keywords:
Parikh vectors
Permutation pattern matching
Approximate pattern matching
Maximum subsequence sum

ABSTRACT

We present a novel approach for computing all maximum consecutive subsums in a sequence of positive integers in near-linear time. Solutions for this problem over binary sequences can be used for reporting existence of Parikh vectors in a bit string. Recently, several attempts have been made to build indexes for all Parikh vectors of a binary string in subquadratic time. However, no algorithm is known to date which can beat by more than a polylogarithmic factor the naive $\Theta(n^2)$ procedure. We show how to construct a $(1+\epsilon)$-approximate index for all Parikh vectors of a binary string in $O(n\frac{\log^2 n}{\log(1+\epsilon)})$ time, for any constant $\epsilon > 0$. Such index is approximate, in the sense that it leaves a small chance for false positives (no false negatives are possible). However, we can tune the parameters of the algorithm so that we can strictly control such a chance of error while still guaranteeing strong subquadratic running time.

1. Introduction

Let $s = s_1, \ldots, s_n$ be a sequence of non-negative integers. For each $\ell = 1, \ldots, n$, we denote with m_ℓ the maximum sum over a consecutive subsequence of s of size ℓ:

$$m_\ell = \max_{i=1,\ldots,n-\ell+1} \sum_{j=i}^{i+\ell-1} s_j.$$

The Maximum Consecutive Subsums Problem (MCSP) asks for computing m_ℓ for each $\ell = 1, \ldots, n$.

Since an obvious implementation of the above formula allows to compute m_ℓ for a single value of ℓ in $O(n)$ time, it follows that there exists a trivial $O(n^2)$ procedure to accomplish the above task. The interesting question is then to find a subquadratic procedure for MCSP. Notwithstanding the effort recently devoted to the problem—particularly in the case of binary strings (see the applications below)—no algorithm is known which is significantly better than the natural $\Theta(n^2)$.

In this paper we show that in near-linear time, we can closely approximate the values m_ℓ within an approximation factor as close to 1 as desired. More precisely, we are interested in the following variant of MCSP:

* Corresponding author.
E-mail addresses: cicalese@dia.unisa.it (F. Cicalese), laber@inf.puc-rio.br (E. Laber), oren@cs.haifa.ac.il (O. Weimann), raphy@math.haifa.ac.il (R. Yuster).

0304-3975/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.tcs.2013.05.032

Approximate Maximum Consecutive Subsums Problem (AMCSP): Given a sequence s of non-negative integers and some $\epsilon > 0$, build an index on s that can quickly report a $(1+\epsilon)$-approximation $\tilde{m}_\ell$ of m_ℓ (for any $\ell = 1, \ldots, n$), i.e., such that $\tilde{m}_\ell \leqslant (1+\epsilon)m_\ell$.

We show two solutions for the AMCSP. The first one constructs an index of size $O(n^\eta)$ in $O(n^{1+\eta})$ time (for any constant $\eta > 0$), and guarantees $O(1/\eta)$ query time. The second solution has a better construction time, namely $O(n\frac{\log^2 n}{\log(1+\epsilon)})$ but in order to guarantee $O(1)$ query time, requires to store the complete list of values $\tilde{m}_\ell$, namely the index built is of size $\Theta(n)$. Alternatively, this latter solution can be used to build an index of only $O(\log_{1+\epsilon} n)$ size but the query time becomes $O(\log_2 \log_{1+\epsilon} n)$. More precisely, we prove

Theorem 1. *For any constants $\epsilon, \eta > 0$, there exists an index for* AMCSP *that is of size $O(k_{\epsilon,\eta} \cdot n^\eta)$, and can be constructed in time $O(k_{\epsilon,\eta} \cdot n^{1+\eta})$, where $k_{\epsilon,\eta}$ is a constant only depending on ϵ and η. The index can be used to report, for any ℓ, a $(1+\epsilon)$-approximation $\tilde{m}_\ell$ for m_ℓ, in $O(1/\eta)$ time.*

Theorem 2. *For any $\epsilon > 0$, there exists an index for* AMCSP *that is of size $O(n)$, and can be constructed in time $O(n\log^2 n/\log(1+\epsilon))$. The index can be used to report, for any ℓ, a $(1+\epsilon)$-approximation $\tilde{m}_\ell$ for m_ℓ, in $O(1)$ time. The same construction can be used to build an index of size $O(\log n/\log(1+\epsilon))$ with $O(\log\log n/\log(1+\epsilon))$ query time.*

MCSP arises in several scenarios of both theoretical and practical interest, in what follows we describe some of them. We start with our main motivation for studying the AMCSP, the following Parikh vector matching problem.

An index for constant time Parikh vector membership queries

Given a string t over an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, for each $c \in \Sigma$, let x_c be the number of occurrences of character c in t. The vector $(x_1, \ldots, x_\sigma)$ is called the Parikh vector of t.

In *Parikh vector pattern matching*, given a string t (the text) and a Parikh vector $p = (x_1, \ldots, x_\sigma)$ (the pattern) we ask for (all/one/existence) of occurrences of substrings s of t such that the Parikh vector of s equals p. Equivalently, we are asking for all the *jumbled* occurrences of a string s (the pattern) in a string t (the text), i.e., any occurrence of some permutation of s in t. Parikh vector matching arises in numerous applications where we search for pattern occurrences but we do not care about the order of the characters inside an occurrence. Most typically, applications are found in computational biology and regard the identification of compounds or complex molecules whose presence can be characterized by the presence of certain substructures and their occurrence within a relatively short distance, whilst the exact location of such substructures within the molecule is not significant [1,2,11,15].

More generally, the Parikh vector matching is an important type of approximate string matching [12,8] where we view two strings as equivalent if one can be turned into the other by permuting its characters. Indeed, like classical pattern matching, a Parikh vector matching can be found in $O(n)$ time. However, while classical pattern matching requires involved ideas like Knuth–Morris–Pratt [13] or Boyer–Moore [4], Parikh vector matching can be easily solved in $O(n)$ time with a simple sliding window based algorithm. The difficulty in Parikh vector matching is for the indexing problem (i.e., preprocess a text in order to efficiently answer Parikh vector queries). For indexing in classical pattern matching, the text can be preprocessed in $O(n)$ time to produce a data structure of size $O(n)$ (such as a suffix tree) that can answer queries efficiently. In contrast, for Parikh vector matching we do not currently have any $o(n^2)$ size data structure for answering queries in time linear in the pattern. This motivates the interest in indexes for *membership* queries which, given several queries over the same string, allows to at least filter out the "no" instances, before, eventually applying the trivial $O(n)$ procedure only to the "yes" instances.

In binary alphabets, it turns out that there is a simple data structure for membership queries that requires only $O(n)$ space and answers queries in $O(1)$ time. This data structure is exactly what MCSP computes. In particular, for a binary string t, knowing for each $\ell = 1, \ldots, n$ the minimum and maximum number of 1's found in a substring of length ℓ of t, we can answer membership queries in constant time. More precisely, the connection between MCSP and Parikh vector membership query problem is as follows.

Lemma 1. *(See [7].) Let s be a binary string and let $\mu_\ell^{\min}$ (resp. $\mu_\ell^{\max}$) denote the minimum (resp. maximum) number of ones in a substring of s of length ℓ. Then there exists a substring in s with Parikh vector $p = (x_0, x_1)$ if and only if $\mu_{x_0+x_1}^{\min} \leqslant x_1 \leqslant \mu_{x_0+x_1}^{\max}$.*

It follows that, after constructing the tables of $\mu^{\min}$'s and $\mu^{\max}$'s (which is equivalent to solving two instances of the MCSP) we can answer in constant time Parikh vector membership queries, i.e., questions asking: "Is there an occurrence of the Parikh vector (x_0, x_1) in s?".

Therefore, there has been recent interest in trying to solve the MCSP on binary sequences in subquadratic time. Currently, only logarithmic factor improvements are known. Namely, the best known constructions are as follows: Burcsi et al. in [7,6] showed a $O(n^2/\log n)$-time algorithm which is based on the $O(n^2/\log n)$ algorithm of Bremner et al. [5,9] for computing $(\min,+)$-convolution; Moosa and Rahman in [14] obtained the same result by a different use of $(\min,+)$-convolution, moreover, they show that an $O(n^2/\log^2 n)$ construction can be obtained assuming word-RAM operations.

Protein identification

The indexes for Parikh vector membership queries in binary strings have also been used in the following problem: Given a string s over an alphabet Σ together with a weight function $w : \Sigma \mapsto \mathbb{N}$. Construct an index on s such that for any $M \in \mathbb{N}$ we can quickly report whether there exists a substring t of s with total weight equal to M, i.e., such that $w(t) = \sum_{i=1}^{|t|} w(c_i)$, where c_i is the ith character of t.

Cieliebak et al. [10] considered this problem in the context of mass-spectrometry based protein identification. They showed that for binary alphabets, the index of maximal consecutive subsums can be used to provide an $O(\log|s|)$ solution to the above type of membership queries, independently of the weight function. However, the authors of [10] did not focus on the complexity of constructing the indexes, and their solution is to trivially construct them in $\Theta(n^2)$ time.

Finding large empty regions in data sets

Another interesting application of MCSP comes from a problem in the analysis of statistical data. Bergkvist and Damaschke in [3] used it for speeding up heuristics for the following problem: Given a sequence of positive real numbers $x_1, \dots, x_n$, called items, and integers $s \geqslant 1$ and $p \geqslant 0$, find s pairwise disjoint intervals with a total of $s + p$ items and maximum total length. Here an interval is a set of consecutive items $x_i, x_{i+1}, \dots, x_j$ $(i \leqslant j)$ and its length is $x_i + x_{i+1} + \cdots + x_j$. This problem (aka Disjoint Intervals of Maximum Length) and its density variant where we are interested in intervals of maximum density [9] rather than absolute length, are motivated by the problem of finding large empty regions (big holes) in data sets. Several other motivating applications can be found in [3] and [9].

By employing a geometric argument, in [3] a heuristic procedure is presented for the MCSP which can solve the problem in $O(n^{3/2})$ time in the best case, but whose worst case remains $\Theta(n^2)$.

Organization of the paper

In Section 2 we provide the proof of Theorem 1. We present the algorithm as a sequence of refinements, starting with a simple way to compute a $(\frac{1+\sqrt{5}}{2})$-approximation in total $O(n^{3/2})$ time and then describe how to improve both the approximation and the time complexity by using a recursive argument. Finally we show how to obtain $O(1)$ query time with a sublinear index size. In Section 3 we provide the proof of Theorem 2. In Section 4 we apply our results to the problem of Parikh vector membership queries. Section 5 concludes the paper with some observations on the results presented here and open problems for future research.

2. An index of sublinear size, near-linear construction, and constant query

In this section we will prove Theorem 1. We first describe an algorithm for computing a $(\frac{1+\sqrt{5}}{2})$-approximation $\tilde{m}_\ell$ of m_ℓ for all $\ell = 1, \dots, n$ in total time $O(n^{3/2})$. We then show how to refine the approach to achieve the desired $(1+\epsilon)$-approximation in the same $O(n^{3/2})$ time. Then, by employing a recursive argument, we show how to achieve the same approximation in time $O(k_{\epsilon,\eta} \cdot n^{1+\eta})$, for any constant $\eta > 0$, where $k_{\epsilon,\eta}$ is a constant that only depends on ϵ and η. Finally, we show that it is enough to store only $O(k_{\epsilon,\eta} \cdot n^{\eta})$ values in an index and we can still guarantee query time $O(1/\eta)$.

For ease of presentation, in the following we neglect rounding necessary to preserve the obvious integrality constraints. The reader can assume that, when necessary, numbers are rounded to the closest integer. It will always be clear that these inaccuracies do not affect the asymptotic results. On the other hand, this way, we gain in terms of much lighter expressions. We will use the following simple facts.

Fact 1. For each $1 \leqslant i < j \leqslant n$ it holds that $m_i \leqslant m_j$.

Fact 2. For each $\ell \in [n]$ and positive integers i, j such that $i + j = \ell$, it holds that $m_\ell \leqslant m_i + m_j$.

Fact 1 directly follows from the non-negativity of the elements in the sequence. Fact 2 is a consequence of the following easy observation: For a fixed ℓ, let $s_r, \dots, s_{r+\ell-1}$ be a subsequence achieving $s_r + \cdots + s_{r+\ell-1} = m_\ell$. By definition we have that, for any i, j such that $i + j = \ell$ it holds that $s_r + \cdots + s_{r+i-1} \leqslant m_i$ and $s_{r+i} + \cdots + s_{r+\ell-1} \leqslant m_j$, from which we obtain the desired inequality.

The following lemma, that follows easily from the above two facts, will be a key tool in the analysis of our algorithm. Informally, the lemma asserts that, for any integer $g > 1$, if we look at the n/g values $m_1, m_g, m_{2g}, m_{3g}, \dots, m_n$ then there cannot be many consecutive pairs m_{jg} and $m_{(j+1)g}$ that differ by a large factor. We will then be able to focus only on these few pairs. For all other pairs m_{jg} and $m_{(j+1)g}$ that do not differ by a large factor, we will be able to approximate m_ℓ for any $jg \leqslant \ell \leqslant (j+1)g$ by returning $m_{(j+1)g}$.

Lemma 2. *Let $k \geqslant 1$ be an integer and $\tilde{\alpha}$ be the positive real solution of the equation $\alpha = 1 + 1/\alpha^k$. Fix an integer $g > 1$ and let $1 \leqslant j_1 < j_2 < \cdots < j_r \leqslant n/g$ be indices such that $m_{(j_i+1)\times g}/m_{j_i\times g} > \alpha$. Then $r \leqslant k$.*

Proof. The proof is by contradiction. Assume that $r > k$. For each $i = 1, \dots, k$ we have that $m_{(j_i+1)\times g}/m_{j_i\times g} > \alpha$. Since $j_1 \geqslant 1$ and $j_r \geqslant j_k + 1$, by Fact 1 we have that

$$\frac{m_{j_r\times g}}{m_g} > \alpha^k. \tag{1}$$

Let us now consider the ratio $m_{(j_r+1)\times g}/m_{j_r\times g}$. We have

$$\frac{m_{(j_r+1)\times g}}{m_{j_r\times g}} \leqslant \frac{m_{j_r\times g} + m_g}{m_{j_r\times g}} = 1 + \frac{m_g}{m_{j_r\times g}} < 1 + \frac{1}{\alpha^k} = \alpha \tag{2}$$

where the first inequality follows from Fact 2, the second inequality follows from (1), and the last equality by the definition of α. Therefore, by assuming $r > k$, we have (2) which contradicts the hypothesis that $m_{(j_r+1)\times g}/m_{j_r\times g} > \alpha$. Hence it must hold that $r \leqslant k$. □

2.1. Warm-up: A golden ratio approximation in $O(n^{3/2})$

Let $\alpha = \frac{1+\sqrt{5}}{2}$. Fix an integer $t \geqslant 2$ and set $g = n^{1/2}$. We will show how to compute $\tilde{m}_\ell$ for all $\ell = 1, \dots, n$, in total time $O(n^{3/2})$, such that $1 \leqslant \tilde{m}_\ell/m_\ell \leqslant \alpha$.

Here we use the expression "*compute* m_ℓ *exhaustively*" (for some fixed ℓ) to indicate the linear time computation attained by scanning the sequence s from left to right and computing the sum of all consecutive subsequences of size ℓ (i.e., a sliding window approach). This exhaustive computation for a single ℓ can be clearly achieved in $\Theta(n)$ time.

The basic idea is to compute exhaustively the value of $m_{j\times g}$ for each $j = 1, \dots, n/g$ and then use these values for approximating all the others. For each $j = 1, \dots, n/g$, we set $\tilde{m}_{j\times g} = m_{j\times g}$, i.e., our approximate index will contain the exact value.

Let ℓ be such that $j \times g < \ell < (j+1) \times g$ for some $j = 1, \dots, n/g - 1$. By Fact 1 we have that $m_{j\times g} \leqslant m_\ell \leqslant m_{(j+1)\times g}$. Therefore, if $m_{(j+1)\times g}/m_{j\times g} \leqslant \alpha$, by setting $\tilde{m}_\ell = m_{(j+1)\times g}$ we also have that $\tilde{m}_\ell$ is an α-approximation of the real value m_ℓ. What happens if the ratio between $m_{(j+1)\times g}$ and $m_{j\times g}$ is large? In this case we say that there is a large gap between $m_{(j+1)\times g}$ and $m_{j\times g}$.

If $m_{(j+1)\times g}/m_{j\times g} > \alpha$, our idea is to compute exhaustively m_ℓ for each $\ell = j \times g + 1, \dots, (j+1) \times g - 1$ (i.e., each m_ℓ between $m_{j\times g}$ and $m_{(j+1)\times g}$). The critical point here is that the "large" gap can only happen once! In fact, this is a consequence of Lemma 2 (with $k = 1$ and $g = n^{1/2}$). Therefore, after encountering the first large gap between $m_{j\times g}$ and $m_{(j+1)\times g}$ all the following gaps will be "small", hence we can safely set $\tilde{m}_\ell = m_{(i+1)\times g}$ for each $i > j$ and $i \times g < \ell \leqslant (i+1) \times g$. In fact, using again Fact 1 we have $\tilde{m}_\ell/m_\ell \leqslant m_{(i+1)\times g}/m_{i\times g}$ and since the ratio must be small (by the above observation) we are guaranteed that $\tilde{m}_\ell = m_{(i+1)\times g}$ is indeed an α-approximation of the exact m_ℓ.

2.2. As close to 1 as desired

We now refine the algorithm described in the previous section in order to show how to obtain a $(1+\epsilon)$-approximation for any $\epsilon > 0$.

Let k be the minimum positive integer such that $\tilde{\alpha} \leqslant 1 + \epsilon$, where $\tilde{\alpha}$ is the positive real solution of the equation $\alpha = 1 + 1/\alpha^k$. In an explicit way, $k = \lceil -\frac{\ln(\epsilon)}{\ln(1+\epsilon)} \rceil$. The value $\tilde{\alpha}$ defines the approximation of our solutions. Let us also set $g = n^{1/2}$.

We partition the list of values m_ℓ ($\ell = 1, \dots, n$) into n/g intervals, each of them with g consecutive values. Then, we proceed as follows:

1. Compute exhaustively the exact value for each m_ℓ in the first interval, i.e., for m_ℓ such that $\ell = 1, \dots, g$.
2. Compute exhaustively the exact value for the extremes of all intervals (i.e., for m_ℓ such that $\ell = 2g, 3g, \dots$).
3. Let $g, 2g, \dots$ be the extremes of the n/g intervals. We say that an extreme $i \times g$ is *relevant* if $m_{(i+1)\times g}/m_{i\times g} > \tilde{\alpha}$. In this case, the interval $[i \times g, (i+1) \times g]$ is a *relevant interval*. Compute exhaustively m_ℓ for every ℓ such that ℓ lies in a relevant interval.
4. The remaining m_ℓ's are approximated by the value of the right extreme in the interval where they lie (i.e., for $\ell \in \{jg+1, \dots, (j+1)g - 1\}$ such that $m_{(j+1)g}/m_{jg} \leqslant \alpha$ we set $\tilde{m}_\ell = m_{(j+1)g}$).

We need to prove that the $\tilde{m}_\ell$ values satisfy the desired $(1+\epsilon)$-approximation. Let $\mathcal{R} = \{j_1 < j_2 < \dots < j_r\}$ be the indices of all the relevant extremes. Note that for each $j \in \mathcal{R}$ the algorithm verified $m_{(j+1)\times g}/m_{j\times g} > \alpha$ and hence computed exhaustively m_ℓ in the interval $\ell = jg, \dots, (j+1) \times g$.

It is easy to see that $1 \leqslant \tilde{m}_\ell/m_\ell \leqslant \alpha$ for every ℓ. In fact, if ℓ is an extreme of an interval then $\tilde{m}_\ell = m_\ell$. If the left extreme of the interval where ℓ lies belongs to $\mathcal{R}$ then $\tilde{m}_\ell = m_\ell$. Finally, if the left extreme, say $j \times g$, of the interval where ℓ lies does not belong to $\mathcal{R}$ then $\tilde{m}_\ell = m_{(j+1)\times g} \leqslant \alpha \times m_{j\times g} \leqslant \alpha \times m_\ell$.

This concludes the proof that for each ℓ the $\tilde{m}_\ell$ values are indeed an α-approximation (and hence a $(1+\epsilon)$-approximation) of m_ℓ.

For the time bound, we observe that, by Lemma 2, we have $r \leqslant k$. Hence, in the above procedure, the number of times we use the exhaustive linear procedure is at most $(k+1) \times n^{1/2}$ for the full intervals of size g (items 1. and 3.) and $n^{1-1/2}$ for the extremes of the intervals (item 2.). Therefore the algorithm runs in time $O(k \cdot n^{3/2})$.

2.3. The last piece: A recursive argument

In the above procedure, for $g = n^{1/2}$ we first compute $m_g, m_{2g}, m_{3g}, \ldots, m_n$ in time $O(n^{3/2})$. Then, we identify (up to) $k+1$ relevant intervals. The first interval is $[1, g]$ and the other k intervals are all the ones of the form $[jg, (j+1)g]$ where $m_{(j+1)g}/m_{jg} > \alpha$. For each one of the $k+1$ relevant intervals we compute *exhaustively* all m_ℓ values where ℓ lies inside the interval, hence in total requiring time $O(k \times n^{3/2})$.

In order to reduce the time complexity, instead of computing all the values exhaustively in the relevant intervals we can recursively use in each interval the argument we use for the full "interval" $[1, m]$. In particular, this means subdividing each relevant interval into sub-intervals and computing exhaustively the values m_ℓ's only at the sub-intervals' extremes. By Lemma 2, there might be—altogether among the sub-intervals of all relevant intervals—$k+1$ relevant sub-intervals, where we recurse again.

As an example, let us explain how to obtain a $(1+\epsilon)$-approximation in time $O(n^{1+1/3})$. For this we choose $g = n^{2/3}$ and compute exhaustively $m_g, m_{2g}, \ldots, m_n$, so spending in total $O(n^{1+1/3})$ time. Then, we identify $k+1$ relevant intervals just as before. Suppose that $[jg, (j+1)g]$ is one of the $k+1$ relevant intervals, i.e., $m_{(j+1)g}/m_{jg} > \alpha$. We partition this interval into $n^{1/3}$ sub-intervals each of size $n^{1/3}$. Now, we first compute m_ℓ for every sub-interval extreme, which requires $n^{1/3} \times n = O(n^{1+1/3})$ time. Then, we compute m_ℓ exhaustively for each ℓ in the $k+1$ relevant sub-intervals, i.e., the first sub-interval and the ones for which the ratio of the values m_ℓ at the boundaries is greater than α. By Lemma 2 (with $g = n^{1/3}$), besides $[1, n^{1/3}]$ at most k sub-intervals can exist for which such condition is verified.

Overall, the number of values of ℓ for which we compute m_ℓ exhaustively are:

- The $n^{1/3}$ boundaries $g, 2g, \ldots$. This requires $O(n^{1/3})$ exhaustive computations of some m_ℓ.
- The $n^{1/3}$ boundaries of the sub-interval in the $k+1$ relevant intervals. This requires $O(kn^{1/3})$ exhaustive computations of some m_ℓ.
- Altogether, in the $k+1$ relevant interval, we may have up to $k+1$ relevant sub-intervals. This holds again by Lemma 2 with $g = n^{1/3}$. Each relevant sub-interval requires to compute $n^{1/3}$ new values m_ℓ's. In total this gives additional $O(kn^{1/3})$ exhaustive computations of some m_ℓ.

Therefore, the time complexity becomes $O(2kn^{1+1/3})$.

If we want to get $O(n^{1+1/4})$ running time we can choose $g = n^{3/4}$ and add one more level of recursion, i.e., partition the sub-intervals into sub-sub-intervals. This way the running time becomes $O(3kn^{1+1/4})$.

In general, given any fixed $\eta > 0$ we can set $t = \lceil 1/\eta \rceil$ and $g = n^{1-1/t}$. By using $t-1$ levels of recursion we then get a $(1+\epsilon)$-approximation $\tilde{m}_\ell$ of m_ℓ for all $\ell = 1, \ldots, n$, in total $O(tkn^{1+1/t}) = O(k_{\epsilon,\eta} n^{1+\eta})$ time, where $k_{\epsilon,\eta}$ is a constant depending only on ϵ and η.

The actual index is built as follows: First, we store the values m_ℓ exhaustively computed at the boundaries of the $n^{1/t}$ intervals considered in the first level of recursion, in an array of size $n^{1/t}$. For each one of the at most $k+1$ relevant intervals we store the boundaries of the sub-intervals in a separate array of size $n^{1/t}$. For each level of recursion, we have at most $k+1$ additional arrays of sub-intervals boundaries for which the values m_ℓ are also stored. Hence, in total we store up to $(k+1)tn^{1/t}$ values.

In order to show that we can retrieve the approximate value for the maximum consecutive sum of any given length ℓ, we exploit the way we split the values $m_1, \ldots, m_n$ into regular size intervals before computing exhaustively their values only at the boundaries of these intervals. The key point here is that the intervals considered at each level of recursion have regular sizes, and, in fact, one specific size for each level of recursion. Therefore, when we look up for the approximate value for m_ℓ, it is easy to compute the interval in which ℓ lies, by a constant time computation.

More precisely, given $\ell \in \{1, \ldots, n\}$, in order to retrieve the approximate value $\tilde{m}_\ell$, we proceed as follows: We first compute the interval in which ℓ lies at the first level of recursion. Notice that at the first level of recursion, the algorithm stored the values $m_{n^{1-1/t}}, m_{2n^{1-1/t}}, m_{3n^{1-1/t}}, \ldots$. Thus, by computing $\ell_1 = \lfloor \ell / n^{1-1/t} \rfloor$ we can retrieve in constant time values $m_{(\ell_1+1)n^{1-1/t}} \geqslant m_\ell$ and $m_{\ell_1 n^{1-1/t}} \leqslant m_\ell$. If $m_{(\ell_1+1)n^{1-1/t}}/m_{\ell_1 n^{1-1/t}} \leqslant (1+\epsilon)$, then we return m_{ℓ_1+1}, which is a correct approximation of m_ℓ.

If $m_{\ell_1+1}/m_{\ell_1} > (1+\epsilon)$, then the algorithm recursed on the interval $[\ell_1 n^{1-1/t}, (\ell_1+1)n^{1-1/t}]$ we use the fact that at the second level of recursion, while building the index, the algorithm built an array of size $n^{1/t}$ containing the values $m_{\ell_1 n^{1-1/t}}, m_{\ell_1 n^{1-1/t}+n^{1-2/t}}, m_{\ell_1 n^{1-1/t}+2n^{1-2/t}}, \ldots, m_{(\ell_1+1)n^{1-1/t}}$.

Therefore—at retrieval time—with $\ell_2 = \lfloor (\ell - \ell_1 n^{1-1/t})/n^{1-2/t} \rfloor$, we look for values stored at positions ℓ_2 and $\ell_2 + 1$ in this array, which contain the values $mn^{1-1/t}\ell_1 + n^{1-2/t}\ell_2 \leqslant m_\ell$ and $mn^{1-1/t}\ell_1 + n^{1-2/t}(\ell_2+1) > m_\ell$, and check their ratio. Again, if the ratio is smaller than $(1+\epsilon)$, we can correctly report the larger value as a $(1+\epsilon)$-approximation for m_ℓ. Otherwise, we proceed as before by looking in the array containing the values computed at the third level of recursion, where the algorithm subdivided the interval $[\ell_1 n^{1-1/t} + \ell_2 n^{1-2/t}, \ell_1 n^{1-1/t} + (\ell_2+1)n^{1-2/t}]$.

Notice that these operations are repeated at most t times—once for each level of recursion of the algorithm—before we either find an interval whose extremes have ratio not larger than $(1+\epsilon)$ or we reach an array stored at the $(t-1)$th level of recursion which contains the exact value for m_ℓ. Therefore, the number of look up operations is bounded by $O(t) = O(1/\eta)$. This complete the proof of Theorem 1.

3. An index of linear size and constant query or logarithmic size and logarithmic query

In this section we will prove Theorem 2. The result will be a consequence of the following proposition.

Proposition 1. *For any constant ϵ, there exists an algorithm which in time $O(\frac{n\log^2 n}{\log(1+\epsilon)})$ computes values $\tilde{m}_1, \ldots, \tilde{m}_n$ such that $1 \leqslant \tilde{m}_\ell/m_\ell \leqslant 1+\epsilon$, for each $\ell = 1, \ldots, n$.*

The algorithm works into two phases, which we will analyze separately and refer to as Algo1 and Algo2, respectively.

In the first phase, Algo1 performs some binary searches to find a sequence of representative indices $1 = t_1 < t_2 < \cdots < t_k = n$ such that for every $i \in \{1, \ldots, n\}$ there is an index t_j in the sequence that satisfies $1 \leqslant m_i/m_{t_j} \leqslant (1+\epsilon)$. In the second phase, Algo2 uses the values computed by Algo1 to compute an approximated value $\tilde{m}_\ell$ for every ℓ such that $1 \leqslant m_\ell/\tilde{m}_\ell \leqslant (1+\epsilon)$. Below is a simple pseudocode description of Algo1:

```
Algo1: Finding the representative indexes
Compute exhaustively $m_1$ and $m_n$
Store $m_1$ and $m_n$ as 1 and $n$
$t \leftarrow 1$
While $m_n/m_t > (1+\epsilon)$
    Do a binary search in $\{t+1, \ldots, n\}$ to find an index $j$ such that
            $\frac{m_j}{m_t} > (1+\epsilon)$ and $\frac{m_{j-1}}{m_t} \leqslant (1+\epsilon)$
    Store values $m_j$ and $m_{j-1}$; Store indices $j$ and $j-1$
    $t \leftarrow j$
End do
```

Notice that whenever the binary search evaluates an index i as a possible candidate for being the desired j, it computes exhaustively the value m_i in $O(n)$ time.

Let $1 = t_1 < t_2 < \cdots < t_k = n$ be the sequence of indices stored by Algo1. It is not difficult to see that for each $1 \leqslant \ell \leqslant n$ there is t_j such that $1 \leqslant m_\ell/m_{t_j} \leqslant (1+\epsilon)$. In fact, if ℓ belongs to the sequence then this clearly holds. Otherwise, let j and $j+1$ be such that $t_j < \ell < t_{j+1}$. It follows that $1 \leqslant m_{t_{j+1}}/m_{t_j} \leqslant (1+\epsilon)$ and from Fact 1 that $m_{t_j} \leqslant m_\ell \leqslant m_{t_{j+1}}$. Thus, $1 \leqslant m_\ell/m_{t_j} \leqslant (1+\epsilon)$.

The following lemma bounds the number of external loops executed by Algo1.

Lemma 3. *The number of external loops executed by* Algo1 *is at most $\frac{\log n}{\log(1+\epsilon)} + 1$.*

Proof. Let p be the number of loops executed by Algo1 and $h_1 < h_2 < \cdots < h_p$ be the sequence containing the largest of the two indexes stored at each loop. It follows from the algorithm's construction that $m_{h_j}/m_{h_{j-1}} > (1+\epsilon)$ for $j = 2, \ldots, p$. Thus, $m_{h_{p-1}} > (1+\epsilon)^{p-1} m_1$. We shall note that $m_{h_p}/m_1 \leqslant n$ because $m_{h_p} < \sum_{i=1}^n s_i$ and $m_1 = \max\{s_1, \ldots, s_n\}$. Thus, we can conclude that $p \leqslant \lceil \frac{\log n}{\log(1+\epsilon)} \rceil + 1$ so that the number of external loops is at most $\frac{\log n}{\log(1+\epsilon)} + 1$. □

Since the algorithm executes at most $\lceil \frac{\log n}{\log(1+\epsilon)} \rceil + 1$ binary searches, each of them computing m_ℓ for at most $\log n$ values of ℓ, it follows that Algo1 runs in $O(\frac{n\log^2 n}{\log(1+\epsilon)})$ time.

If we want to have an index of $O(n)$ size to answer approximate queries in $O(1)$ time, we may run Algo2, presented below.

```
Algo2: Building the index
Set $\tilde{m}_1 = m_1$; $t \leftarrow 1$
For $i = 2$ to $n$
    If $m_i$ has already been stored in Phase 1
        Set $\tilde{m}_i = m_i$; $t \leftarrow i$
    Else
        Set $\tilde{m}_i = m_t$
    End If
End For
```

Alternatively, we could avoid running Algo2 by storing only the sequence of representative indexes computed by Algo1 in a search tree. In this case, we have an index of $O(\frac{\log n}{\log(1+\epsilon)})$ size that can be queried in $O(\log(\frac{\log n}{\log(1+\epsilon)}))$ time.

4. Applying the index to Parikh vector pattern matching

We can now analyze the consequences of our result with respect to the connection between MCSP and the Parikh vector membership query problem. An immediate corollary of Proposition 1 is the following:

Corollary 1. *Let s be a binary string and for each $i = 1, \dots, n$ let $\mu_\ell^{\min}$ (resp. $\mu_\ell^{\max}$) denote the minimum (resp. maximum) number of ones in a substring of s of length ℓ. For any $\epsilon \in (0,1)$, we can compute in $O(n\frac{\log^2 n}{\log(1+\epsilon)})$ approximate values $\tilde{\mu}_\ell^{\min}$ (resp. $\tilde{\mu}_\ell^{\max}$) such that*

$$\mu_\ell^{\min} \geqslant \tilde{\mu}_\ell^{\min} \geqslant (1-\epsilon)\mu_\ell^{\min} \quad \text{and} \quad \mu_\ell^{\max} \leqslant \tilde{\mu}_\ell^{\max} \leqslant (1+\epsilon)\mu_\ell^{\max}.$$

Let s be a binary string of length n. Fix a tolerance threshold $\epsilon > 0$, and let $\tilde{\mu}_\ell^{\min}$ and $\tilde{\mu}_\ell^{\max}$ ($\ell = 1, \dots, n$) be as in Corollary 1. In terms of Parikh vector membership queries, we have the following necessary condition for the occurrence of a Parikh vector in s.

Corollary 2. *For any $p = (x_0, x_1)$ such that there exists a substring of s whose Parikh vector equals p, we have that*

$$\tilde{\mu}_{x_0+x_1}^{\min} \leqslant x_1 \leqslant \tilde{\mu}_{x_0+x_1}^{\max}.$$

We also have an "almost matching" sufficient condition, which also follows from Lemma 1 and Corollary 1.

Corollary 3. *Fix a Parikh vector $p = (x_0, x_1)$. If*

$$\frac{\tilde{\mu}_{x_0+x_1}^{\min}}{1-\epsilon} \leqslant x_1 \leqslant \frac{\tilde{\mu}_{x_0+x_1}^{\max}}{1+\epsilon}$$

then p occurs in s.

As a consequence, if we use the values $\tilde{\mu}^{\min}$ and $\tilde{\mu}^{\max}$ we can answer correctly to any membership query involving a Parikh vector which occurs in s. Moreover, we can also answer correctly any membership query involving a Parikh vector satisfying the condition in Corollary 3. In contrast, it might be that the approximate index makes us report false positives, when the membership query is about a Parikh vectors $p = (x_0, x_1)$ such that

$$\tilde{\mu}_{x_0+x_1}^{\min} \leqslant x_1 \leqslant \frac{\tilde{\mu}_{x_0+x_1}^{\min}}{1-\epsilon} \quad \text{or} \quad \frac{\tilde{\mu}_{x_0+x_1}^{\max}}{1+\epsilon} \leqslant x_1 \leqslant \tilde{\mu}_{x_0+x_1}^{\max}.$$

5. Some final observations and open problems

We presented two novel approaches to approximate all maximum consecutive subsums of a sequence. Our solutions can be directly used for obtaining indices for binary strings, which allow to give in constant time approximate answers to Parikh vector membership queries. For the *exact* case, the existence of a $o(n^{2-\delta})$ time solution, for some $\delta > 0$, remains a major open problem.

The computation of our indexes is easy and seems fast to implement, and, in contrast to the previous *exact* solutions presented in the literature [7,6,14], does not require any tabulation or convolution computation.

It would be interesting to investigate whether it is possible to obtain Las Vegas algorithms for the Parikh vector problem based on our approximation perspective. Another direction for future investigation regards the extension of our approach to cover the case of Parikh vector membership queries in strings over non-binary alphabets.

References

[1] A. Amir, A. Apostolico, G.M. Landau, G. Satta, Efficient text fingerprinting via Parikh mapping, J. Discrete Algorithms 1 (5–6) (2003) 409–421.
[2] G. Benson, Composition alignment, in: Proc. of the 3rd International Workshop on Algorithms in Bioinformatics, WABI'03, 2003, pp. 447–461.
[3] A. Bergkvist, P. Damaschke, Fast algorithms for finding disjoint subsequences with extremal densities, Pattern Recognit. 39 (2006) 2281–2292.
[4] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20 (10) (1977) 762–772.
[5] D. Bremner, T.M. Chan, E.D. Demaine, J. Erickson, F. Hurtado, J. Iacono, S. Langerman, P. Taslakian, Necklaces, convolutions, and $X + Y$, in: 14th Annual European Symposium on Algorithms, ESA'06, 2006, pp. 160–171.
[6] P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták, On approximate jumbled pattern matching, Theory Comput. Syst. 50 (1) (2012) 35–51.
[7] P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták, Algorithms for jumbled pattern matching in strings, in: 5th International Conference FUN with Algorithms, FUN 2010, 2010, pp. 89–101.

[8] A. Butman, R. Eres, G.M. Landau, Scaled and permuted string matching, Inform. Process. Lett. 92 (6) (2004) 293–297.
[9] Y.H. Chen, H.I. Lu, C.Y. Tang, Disjoint segments with maximum density, in: Proc. of the International Workshop on Bioinformatics Research and Applications, IWBRA 2005, in: Lecture Notes in Comput. Sci., vol. 3515, 2005, pp. 845–850.
[10] M. Cieliebak, T. Erlebach, Zs. Lipták, J. Stoye, E. Welzl, Algorithmic complexity of protein identification: Combinatorics of weighted strings, Discrete Appl. Math. 137 (1) (2004) 27–46.
[11] R. Eres, G.M. Landau, L. Parida, Permutation pattern discovery in biosequences, J. Comput. Biol. 11 (6) (2004) 1050–1060.
[12] P. Jokinen, J. Tarhio, E. Ukkonen, A comparison of approximate string matching algorithms, Softw. Practice Exper. 26 (12) (1996) 1439–1458.
[13] D.E. Knuth, J.H. Morris Jr., V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[14] T.M. Moosa, M.S. Rahman, Sub-quadratic time and linear size data structures for permutation matching in binary strings, J. Discrete Algorithms 10 (1) (2012) 5–9.
[15] L. Parida, Gapped permutation patterns for comparative genomics, in: Proc. of WABI 2006, 2006, pp. 376–387.