Threshold Approximate Matching in Grammar-Compressed Strings

Alexander Tiskin[⋆]

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
tiskin@dcs.warwick.ac.uk

Abstract. A grammar-compressed (GC) string is a string generated by a context-free grammar. This compression model captures many practical applications, and includes LZ78 and LZW compression as a special case. We give an efficient algorithm for threshold approximate matching on a GC-text against a plain pattern. Our algorithm improves on existing algorithms whenever the pattern is sufficiently long. The algorithm employs the technique of fast unit-Monge matrix distance multiplication, as well as a new technique for implicit unit-Monge matrix searching, which we believe to be of independent interest.

1 Introduction

String compression is a standard approach to dealing with massive data sets. From an algorithmic viewpoint, it is natural to ask whether compressed strings can be processed efficiently without decompression. For a recent survey on the topic, see Lohrey [14]. Efficient algorithms for compressed strings can also be applied to achieve speedup over ordinary string processing algorithms for plain strings that are highly compressible.

We consider the following general model of compression.

Definition 1. *Let t be a string of length n (typically large). String t will be called a* grammar-compressed string (GC-string)*, if it is generated by a context-free grammar, also called a* straight-line program (SLP)*. An SLP of length $\bar{n}$, $\bar{n} \leq n$, is a sequence of $\bar{n}$* statements*. A statement numbered k, $1 \leq k \leq \bar{n}$, has one of the following forms: $t_k = \alpha$, where α is an alphabet character, or $t_k = t_i t_j$, where $1 \leq i, j < k$.*

We identify every symbol t_r with the string it represents; in particular, we have $t = t_{\bar{n}}$. In general, the plain string length n can be exponential in the GC-string length $\bar{n}$. Grammar compression includes as a special case the classical LZ78 and LZW compression schemes by Ziv, Lempel and Welch [24,22].

Approximate pattern matching is a natural generalisation of both the ordinary (exact) pattern matching, and of the alignment score and the edit distance problems. Given a text string t of length n and a pattern string p of length $m \leq n$, the *approximate pattern matching problem* asks to find the substrings of the text that are locally closest to the pattern, i.e. that have the locally highest alignment score (or, equivalently, lowest edit distance) against the pattern. The precise definition of "locally" may vary in different versions of the problem.

Definition 2. *The* threshold approximate matching problem *(often called simply "approximate matching") assumes an alignment score with arbitrary weights, and,*

[⋆] Research supported by the Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, EPSRC award EP/D063191/1.

Alexander Tiskin: Threshold Approximate Matching in Grammar-Compressed Strings, pp. 124–138.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2 © Czech Technical University in Prague, Czech Republic

given a threshold score h, asks for all substrings of text t that have alignment score $\geq h$ against pattern p.

The substrings asked for by Definition 2 will be called *matching substrings.* The precise definition of "alignment score with weights" will be given in Subsection 2.5.

An important special case arises when the unweighted LCS score is chosen as the alignment score, and the pattern length m is chosen as the threshold h. This special case is known as the *local subsequence recognition problem.*

Approximate pattern matching on compressed text has been studied by Kärkkäinen et al. [11]. For a GC-text of length $\bar{n}$, an uncompressed pattern of length m, and an edit distance threshold k, the (suitably generalised) algorithm of [11] solves the threshold approximate matching problem in time $O(m\bar{n}k^2 + \mathit{output})$. In the special case of LZ78 or LZW compression, the running time is reduced to $O(m\bar{n}k + \mathit{output})$. Bille et al. [5] gave an efficient general scheme for adapting an arbitrary threshold approximate matching algorithm to work on a GC-text. In particular, when the algorithms by Landau and Vishkin [12] and by Cole and Hariharan [8] are each plugged into their scheme, the resulting algorithm runs in time $O(\bar{n} \cdot \min(mk, k^4 + m) + \bar{n}^2 + \mathit{output})$. In the special case of LZ78 or LZW compression, Bille et al. [4] show that the running time can be reduced to $O(\bar{n} \cdot \min(mk, k^4 + m) + \mathit{output})$.

For the special case of the local subsequence recognition problem on a GC-text, Cégielski et al. [6] gave an algorithm running in time $O(m^2 \log m \cdot \bar{n} + \mathit{output})$. In [21], we improved the running time to $O(m \log m\bar{n} + \mathit{output})$. Recently, Yamamoto et al. [23] improved the running time still further to $O(m\bar{n} + \mathit{output})$.

In this paper, we consider the threshold approximate matching problem on a GC-text against a plain pattern. We give an algorithm running in time $O(m \log m \cdot \bar{n} + \mathit{output})$, which improves on existing algorithms whenever the pattern is sufficiently long. The algorithm employs the technique of fast unit-Monge matrix distance multiplication [16], as well as a new technique for implicit unit-Monge matrix searching (Lemma 12).

This paper is a sequel to [21]; we encourage the reader to refer there for a warm-up to the result and the techniques presented in the current paper. Due to space constraints, some proofs and examples are omitted. They can be found in [17].

2 General techniques

We recall the framework developed in [18,19,20,16,21], and fully presented in [17].

2.1 Preliminaries

For indices, we will use either integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, or half-integers $\bigl\{\ldots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots\bigr\}$. For ease of reading, half-integer variables will be indicated by hats (e.g. $\hat{\imath}$, $\hat{\jmath}$). Ordinary variable names (e.g. i, j, with possible subscripts or superscripts), will normally denote integer variables, but can sometimes denote a variable that may be either integer, or half-integer.

It will be convenient to denote $i^- = i - \frac{1}{2}$, $i^+ = i + \frac{1}{2}$ for any integer or half-integer i. The set of all half-integers can now be written as $\bigl\{\ldots, (-3)^+, (-2)^+, (-1)^+, 0^+, 1^+, 2^+, \ldots\bigr\}$. We denote integer and half-integer *intervals* by $[i : j] = \{i, i+1, \ldots, j-1, j\}$, $\langle i : j \rangle = \bigl\{i^+, i + \frac{3}{2}, \ldots, j - \frac{3}{2}, j^-\bigr\}$. In both cases, the interval is defined by integer endpoints.

Given two index ranges I, J, it will be convenient to denote their Cartesian product by $(I \mid J)$. We extend this notation to Cartesian products of intervals:

$$[i_0 : i_1 \mid j_0 : j_1] = ([i_0 : i_1] \mid [j_0 : j_1]) \qquad \langle i_0 : i_1 \mid j_0 : j_1\rangle = (\langle i_0 : i_1\rangle \mid \langle j_0 : j_1\rangle)$$

Given index ranges I, J, a *vector over* I is indexed by $i \in I$, and a *matrix over* $(I \mid J)$ is indexed by $i \in I$, $j \in J$.

We will use the parenthesis notation for indexing matrices, e.g. $A(i, j)$. We will also use straightforward notation for selecting subvectors and submatrices: for example, given a matrix A over $[0 : n \mid 0 : n]$, we denote by $A[i_0 : i_1 \mid j_0 : j_1]$ the submatrix defined by the given sub-intervals. A star $*$ will indicate that for a particular index, its whole range is being used, e.g. $A[* \mid j_0 : j_1] = A[0 : n \mid j_0 : j_1]$. In particular, $A(*, j)$ and $A(i, *)$ will denote a full matrix column and row, respectively.

We recall the following definitions from [21,17]. We define two natural strict partial orders on points, called $\ll$- and $\gtrless$-*dominance*:

$$(i_0, j_0) \ll (i_1, j_1) \quad \text{if } i_0 < i_1 \text{ and } j_0 < j_1 \qquad (i_0, j_0) \gtrless (i_1, j_1) \quad \text{if } i_0 > i_1 \text{ and } j_0 < j_1$$

When visualising points, we will deviate from the standard Cartesian visualisation of the coordinate axes, and will use instead the matrix indexing convention: the first coordinate in a pair increases downwards, and the second coordinate rightwards. Hence, $\ll$- and $\gtrless$-dominance correspond respectively to the “above-left” and “below-left” partial orders. The latter order corresponds to the standard visual convention for dominance in computational geometry.

Definition 3. *Let D be a matrix over $\langle i_0 : i_1 \mid j_0 : j_1\rangle$. Its* distribution matrix D^Σ *over $[i_0 : i_1 \mid j_0 : j_1]$ is defined by $D^\Sigma(i, j) = \sum_{\hat\imath \in \langle i:i_1\rangle, \hat\jmath \in \langle j_0:j\rangle} D(\hat\imath, \hat\jmath)$ for all $i \in [i_0 : i_1]$, $j \in [j_0 : j_1]$.*

Definition 4. *Let A be a matrix over $[i_0 : i_1 \mid j_0 : j_1]$. Its* density matrix $A^\square$ *over $\langle i_0 : i_1 \mid j_0 : j_1\rangle$ is defined by $A^\square(\hat\imath, \hat\jmath) = A(\hat\imath^+, \hat\jmath^-) - A(\hat\imath^-, \hat\jmath^-) - A(\hat\imath^+, \hat\jmath^+) + A(\hat\imath^-, \hat\jmath^+)$ for all $\hat\imath \in \langle i_0 : i_1\rangle$, $\hat\jmath \in \langle j_0 : j_1\rangle$.*

Definition 5. *Matrix A over $[i_0 : i_1 \mid j_0 : j_1]$ will be called* simple*, if $A(i_1, j) = A(i, j_0) = 0$ for all i, j. Equivalently, A is simple if $A^{\square\Sigma} = A$.*

Definition 6. *Matrix A is called* totally monotone*, if $A(i, j) > A(i, j') \Rightarrow A(i', j) > A(i', j')$ for all $i \le i'$, $j \le j'$.*

Definition 7. *Matrix A is called* a Monge matrix*, if $A(i, j) + A(i', j') \le A(i, j') + A(i', j)$ for all $i \le i'$, $j \le j'$. Equivalently, matrix A is a Monge matrix, if $A^\square$ is nonnegative. Matrix A is called* an anti-Monge matrix*, if $-A$ is Monge.*

Definition 8. *A* permutation *(respectively,* subpermutation*)* matrix *is a zero-one matrix containing exactly one (respectively, at most one) nonzero in every row and every column.*

Definition 9. *0 Matrix A is called* a unit-Monge *(respectively,* subunit-Monge*)* matrix*, if $A^\square$ is a permutation (respectively, subpermutation) matrix. Matrix A is called* a unit-anti-Monge *(respectively,* subunit-anti-Monge*)* matrix*, if $-A$ is unit-Monge (respectively, subunit-Monge).*

A permutation matrix P of size n can be regarded as an implicit representation of the simple unit-Monge matrix P^Σ. Geometrically, a value $P^\Sigma(i,j)$ is the number of (half-integer) nonzeros in matrix P that are $\lessgtr$-dominated by the (integer) point (i,j). Matrix P can be preprocessed to allow efficient element queries on $P^\Sigma(i,j)$. Here, we consider *incremental queries*, which are given an element of an implicit simple (sub)unit-Monge matrix, and return the value of an adjacent element. This kind of query can be answered directly from the (sub)permutation matrix, without any preprocessing.

Theorem 10. *Given a (sub)permutation matrix P of size n, and the value $P^\Sigma(i,j)$, $i,j \in [0:n]$, the values $P^\Sigma(i \pm 1, j)$, $P^\Sigma(i, j \pm 1)$, where they exist, can be queried in time $O(1)$.*

Proof. Straightforward; see [17].

2.2 Implicit matrix searching

We recall a classical row minima searching algorithm by Aggarwal et al. [1], often nicknamed the “SMAWK algorithm”.

Lemma 11 ([1]). *Let A be an $n_1 \times n_2$ implicit totally monotone matrix, where each element can be queried in time q. The problem of finding the (say, leftmost) minimum element in every row of A can be solved in time $O(qn)$, where $n = \max(n_1, n_2)$.*

Proof (Lemma 11). Without loss of generality, let A be over $[0:n \mid 0:n]$. Let B be an implicit $\frac{n}{2} \times n$ matrix over $\left[0:\frac{n}{2} \mid 0:n\right]$, obtained by taking every other row of A. Clearly, at most $\frac{n}{2}$ columns of B contain a leftmost row minimum. The key idea of the algorithm is to eliminate $\frac{n}{2}$ of the remaining columns in an efficient process, based on the total monotonicity property.

We call a matrix element *marked* (for elimination), if its column has not (yet) been eliminated, but the element is already known not to be a leftmost row minimum. A column gets eliminated when all its elements become marked.

Initially, both the set of eliminated columns and the set of marked elements are empty. In the process of column elimination, marked elements may only be contained in the i leftmost uneliminated columns; the value of i is initially equal to 1, and gets either incremented or decremented in every step of the algorithm. The marked elements form a *staircase*: in the first, second, …, i-th uneliminated column, respectively zero, one, …, $i-1$ topmost elements are marked. In every iteration of the algorithm, two outcomes are possible: either the staircase gets extended to the right to the $i+1$-st uneliminated column, or the whole i-th uneliminated column gets eliminated, and therefore deleted from the staircase.

Let j, j' denote respectively the indices of the i-th and $i+1$-st uneliminated column in the original matrix (across both uneliminated and eliminated columns). The outcome of the current iteration depends on the comparison of element $B(i,j)$, which is the topmost unmarked element in the i-th uneliminated column, against element $B(i,j')$, which is the next uneliminated (and unmarked) element immediately to its right. The outcomes of this comparison and the rest of the elimination procedure are given in Table 1. By storing indices of uneliminated columns in an appropriate dynamic data structure, such as a doubly-linked list, a single iteration of this procedure can be implemented to run in time $O(q)$. The whole procedure runs in time $O(qn)$, and eliminates $\frac{n}{2}$ columns.

```
$i \leftarrow 0$; $j \leftarrow 0$; $j' \leftarrow 1$
while $j' \leq n$:
  case $B(i,j) \leq B(i,j')$:
    case $i < \frac{n}{2}$:  $i \leftarrow i+1$; $j \leftarrow j'$
    case $i = \frac{n}{2}$:  eliminate column $j'$
    $j' \leftarrow j'+1$
  case $B(i,j) > B(i,j')$:
    eliminate column $j$
    case $i = 0$:  $j \leftarrow j'$; $j' \leftarrow j'+1$
    case $i > 0$:  $i \leftarrow i-1$; $j \leftarrow \max\{k : k$ uneliminated and $< j\}$
```

Table 1. Elimination procedure of Lemmas 11 and 12.

Let A' be the $\frac{n}{2} \times \frac{n}{2}$ matrix obtained from B by deleting the $\frac{n}{2}$ eliminated columns. We call the algorithm recursively on A'. This recursive call returns the leftmost row minima of A', and therefore also of B. It is now straightforward to fill in the leftmost minima in the remaining rows of A in time $O(qn)$. Thus, the top level of recursion runs in time $O(qn)$. The amount of work gets halved with every recursion level, therefore the overall running time is $O(qn)$.

Let us now restrict our attention to implicit unit-Monge matrices. An element of such a matrix (represented by an appropriate data structure, see [17]) can be queried in time $q = O(\log^2 n)$. A more careful analysis of the elimination procedure of Lemma 11 shows that the required matrix elements can be obtained, instead of standalone element queries, by the more efficient incremental queries of Theorem 10.

Lemma 12. *Let A be an implicit (sub)unit-Monge matrix over $[0 : n_1 \mid 0 : n_2]$, represented by the (sub)permutation matrix $P = A^{\square}$ and vectors $b = A(n_1, *)$, $c = A(*, 0)$, such that $A(i,j) = P^{\Sigma}(i,j) + b(j) + c(i) - b(0)$ for all i, j. The problem of finding the (say, leftmost) minimum element in every row of A can be solved in time $O(n \log\log n)$, where $n = \max(n_1, n_2)$.*

Proof (Lemma 12). First, observe that vector c has no effect on the positions (as opposed to the values) of any row minima. Therefore, we assume without loss of generality that $c(i) = 0$ for all i (and, in particular, $b(0) = c(n_1) = 0$). Further, suppose that some column $P(*, \hat{\jmath})$ is identically zero; then, depending on whether $b(\hat{\jmath}^-) \leq b(\hat{\jmath}^+)$ or $b(\hat{\jmath}^-) > b(\hat{\jmath}^+)$, we may delete respectively column $A(*, \hat{\jmath}^+)$ or $A(*, \hat{\jmath}^-)$ as it does not contain any leftmost row minima. Also, suppose that some row $P(\hat{\imath}, *)$ is identically zero; then the minimum value in row $A(\hat{\imath}^-, *)$ lies in the same column as the minimum value in row $A(\hat{\imath}^-, *)$, hence we can delete one of these rows. Therefore, we assume without loss of generality that A is an implicit unit-Monge matrix over $[0 : n \mid 0 : n]$, and hence P is a permutation matrix.

To find all the leftmost row minima, we adopt the column elimination procedure of Lemma 11 (see Table 1), with some modifications outlined below.

Let B be an implicit $n^{1/2} \times n$ matrix, obtained by taking a subset of $n^{1/2}$ rows of A at regular intervals of $n^{1/2}$. Clearly, at most $n^{1/2}$ columns of B contain a leftmost row minimum. We need to eliminate $n - n^{1/2}$ of the remaining columns.

Let B be over $\left[0 : \frac{n}{2} \mid 0 : n\right]$. Throughout the elimination procedure, we maintain a vector $d(i)$, $i \in [0 : n^{1/2} - 1]$, initialised by zero values. In every iteration, given a current value of the index j', each value $d(i)$ gives the count of nonzeros $P(s,t) = 1$ within the rectangle $s \in \left\langle n^{1/2} i : n^{1/2}(i+1)\right\rangle$, $t \in \langle 0 : j' \rangle$.

Consider an iteration of the column elimination procedure of Lemma 11 with given values i, j, j', operating on matrix elements $B(i,j)$, $B(i,j')$. For the iteration that follows the current one, the following matrix elements may be required:

- $B(i-1,j')$, $B(i+1,j')$. These values can be obtained respectively as $B(i,j)+d(i-1)$ and $B(i,j)-d(i)$.
- $B(i,j'+1)$, $B(i+1,j'+1)$. These values can be obtained respectively from $B(i,j')$, $B(i+1,j')$ by a rowwise incremental query of matrix P^{Σ} via Theorem 10, plus a single access to vector b.
- $B(i-1,\{k : k \text{ uneliminated and } < j\})$. This element was already queried in the iteration at which its column was first added to the staircase. There is at most one such element per column, therefore each of them can be stored and subsequently queried in constant time.

At the end of the current iteration, index j' may be incremented (i.e. the staircase may grow by one column). In this case, we also need to update vector d for the next iteration. Let $s \in \langle 0 : n \rangle$ be such that $P(s, j'-\frac{1}{2}) = 1$. Let $i = \lfloor s/n^{1/2} \rfloor$; we have $s \in \langle n^{1/2}i : n^{1/2}(i+1) \rangle$. The update consists in incrementing $d(i)$ by 1.

The total number of iterations in the elimination procedure is at most $2n$. This is because in total, at most n columns are added to the staircase, and at most n (in fact, exactly $n - n^{1/2}$) columns are eliminated. Therefore, the elimination procedure runs in time $O(n)$.

Let A' be the $n^{1/2} \times n^{1/2}$ matrix obtained from B by deleting the $n - n^{1/2}$ eliminated columns. Using incremental queries to matrix P, it is straightforward to obtain matrix A' explicitly in random-access memory in time $O(n)$. We now call the algorithm of Lemma 11 to compute the row minima of A', and therefore also of B, in time $O(n)$.

We now need to fill in the remaining row minima of matrix A. The row minima of matrix A' define a chain of $n^{1/2}$ submatrices in A at which these remaining row minima may be located. More specifically, given two successive row minima of A', all the $n^{1/2}$ row minima that are located between the two corresponding rows in A must also be located between the two corresponding columns. Each of the resulting submatrices has $n^{1/2}$ rows; the number of columns may vary from submatrix to submatrix. It is straightforward to eliminate from each submatrix all columns not containing any nonzero of matrix P; therefore, without loss of generality, we may assume that every submatrix is of size $n^{1/2} \times n^{1/2}$.

We now call the algorithm recursively on each submatrix to fill in the remaining leftmost row minima. The amount of work remains $O(n)$ in every recursion level. There are $\log\log n$ recursion levels, therefore the overall running time of the algorithm is $O(n \log\log n)$.

An even faster algorithm, running in optimal time $O(n)$ on the RAM model, has been recently suggested by Gawrychowski [9]. The algorithm of Lemma 12, which is thus suboptimal but has weaker model requirements, will be sufficient for the purposes of this paper.

2.3 Semi-local LCS

We will consider strings of characters taken from an alphabet. Two alphabet characters α, β *match*, if $\alpha = \beta$, and *mismatch* otherwise. In addition to alphabet characters, we introduce two special extra characters: the *guard character* ‘$’, which only matches itself and no other characters, and the *wildcard character* '?', which matches itself and all other characters.

It will be convenient to index strings by half-integer, rather than integer indices, e.g. string $a = \alpha_{0^+}\alpha_{1^+}\cdots\alpha_{m^-}$. We will index strings as vectors, writing e.g. $a(\hat\imath) = \alpha_{\hat\imath}$, $a\langle i : j\rangle = \alpha_{i^+}\cdots\alpha_{j^-}$. Given strings a over $\langle i : j\rangle$ and b over $\langle i' : j'\rangle$, we will distinguish between string *right concatenation* $\underline{a}b$, which is over $\langle i : j + j' - i'\rangle$ and preserves the indexing within a, and *left concatenation* $a\underline{b}$, which is over $\langle i' - j + i : j'\rangle$ and preserves the indexing within b. We extend this notation to concatenation of more than two strings, e.g. $a\underline{b}c$ is a concatenation of three strings, where the indexing of the second string is preserved. If no string is marked in the concatenation, then right concatenation is assumed by default.

Given a string, we distinguish between its contiguous *substrings*, and not necessarily contiguous *subsequences*. Special cases of a substring are *a prefix* and *a suffix* of a string. Unless indicated otherwise, an algorithm's input is a string a of length m, and a string b of length n.

We recall the following definitions from [21,17].

Definition 13. *Given strings a, b, the* longest common subsequence (LCS) problem *asks for the length of the longest string that is a subsequence of both a and b. We will call this length the* LCS score *of strings a, b.*

Definition 14. *Given strings a, b, the* semi-local LCS problem *asks for the LCS scores as follows: a against every substring of b (the* string-substring LCS scores*); every prefix of a against every suffix of b (the* prefix-suffix LCS scores*); symmetrically, the* substring-string LCS scores *and the* suffix-prefix LCS scores*, defined as above but with the roles of a and b exchanged. The first three (respectively, the last three) components, taken together, will also be called the* extended string-substring *(respectively,* substring-string*)* LCS problem*.*

Definition 15. *A* grid-diagonal dag *is a weighted dag, defined on the set of nodes $v_{l,i}$, $l \in [0 : m]$, $i \in [0 : n]$. The edge and path weights are called* scores*. For all $l \in [0 : m]$, $\hat l \in \langle 0 : m\rangle$, $i \in [0 : n]$, $\hat\imath \in \langle 0 : n\rangle$, the grid-diagonal dag contains:*

- *the horizontal edge $v_{l,\hat\imath^-} \to v_{l,\hat\imath^+}$ and the vertical edge $v_{\hat l^-,i} \to v_{\hat l^+,i}$, both with score 0;*
- *the diagonal edge $v_{\hat l^-,\hat\imath^-} \to v_{\hat l^+,\hat\imath^+}$ with score either 0 or 1.*

Definition 16. *An instance of the semi-local LCS problem on strings a, b corresponds to an $m\times n$ grid-diagonal dag $\mathrm{G}_{a,b}$, called the* alignment dag *of a and b. A cell indexed by $\hat l \in \langle 0 : m\rangle$, $\hat\imath \in \langle 0 : n\rangle$ is called a* match cell*, if $a(\hat l)$ matches $b(\hat\imath)$, and a* mismatch cell *otherwise (recall that the strings may contain wildcard characters). The diagonal edges in match cells have score 1, and in mismatch cells score 0.*

Definition 17. *Given strings a, b, the corresponding* semi-local score matrix *is a matrix over $[-m : n \mid 0 : m+n]$, defined by $\mathrm{H}_{a,b}(i, j) = \max score(v_{0,i} \rightsquigarrow v_{m,j})$, where $i \in [-m : n]$, $j \in [0 : m + n]$, and the maximum is taken across all paths between the given endpoints $v_{0,i}$, $v_{m,j}$ in the $m \times (2m+n)$ padded alignment dag $\mathrm{G}_{a,?^m\underline{b}?^m}$. If $i = j$, we have $\mathrm{H}_{a,b}(i, j) = 0$. By convention, if $j < i$, then we let $\mathrm{H}_{a,b}(i, j) = j - i < 0$.*

Theorem 18. *Given strings a, b, the corresponding semi-local score matrix $\mathrm{H}_{a,b}$ is unit-anti-Monge. More precisely, we have $\mathrm{H}_{a,b}(i, j) = j-i-\mathrm{P}^{\Sigma}_{a,b}(i, j) = m-\mathrm{P}^{T\Sigma T}_{a,b}(i, j)$, where $\mathrm{P}_{a,b}$ is a permutation matrix over $\langle -m : n \mid 0 : m+n\rangle$. In particular, string a is a subsequence of substring $b\langle i : j\rangle$ for some $i, j \in [0 : n]$, if and only if $\mathrm{P}^{T\Sigma T}_{a,b}(i, j) = 0$.*

Definition 19. *Given strings a, b, the* semi-local seaweed matrix *is a permutation matrix* $\mathrm{P}_{a,b}$ *over* $\langle -m : n \mid 0 : m+n \rangle$, *defined by Theorem 18.*

When talking about semi-local score and seaweed matrices, we will sometimes omit the qualifier "semi-local", as long as it is clear from the context.

2.4 Seaweed submatrix notation

The four individual components of the semi-local LCS problem correspond to a partitioning of both the score matrix $\mathrm{H}_{a,b}$ and the seaweed matrix $\mathrm{P}_{a,b}$ into submatrices. It will be convenient to introduce a special notation for the resulting subranges of their respective index ranges $[-m : n \mid 0 : m+n]$ and $\langle -m : n \mid 0 : m+n \rangle$. This notation will be used as matrix superscripts, e.g. $\mathrm{H}^{◱}_{a,b} = \mathrm{H}_{a,b}[0 : n \mid 0 : n]$ denotes the matrix of all string-substring LCS scores for strings a, b. The notation for other subranges is as follows.

	$0 : n$	$n : m+n$
$-m : 0$	$\mathrm{H}^{◰}_{a,b}$	$\mathrm{H}^{◳}_{a,b}$
$0 : n$	$\mathrm{H}^{◱}_{a,b}$	$\mathrm{H}^{◲}_{a,b}$

and analogously for $\mathrm{P}_{a,b}$. Note that the four defined half-integer subranges of matrix $\mathrm{P}_{a,b}$ are disjoint, but the corresponding four integer subranges of matrix $\mathrm{H}_{a,b}$ overlap by one row/column at the boundaries.

Definition 20. *Given strings a, b, the corresponding* suffix-prefix, substring-string, string-substring and prefix-suffix score *(respectively,* seaweed*)* matrices *are the submatrices* $\mathrm{H}^{◰}_{a,b}$, $\mathrm{H}^{◳}_{a,b}$, $\mathrm{H}^{◱}_{a,b}$, $\mathrm{H}^{◲}_{a,b}$ *(respectively,* $\mathrm{P}^{◰}_{a,b}$, $\mathrm{P}^{◳}_{a,b}$, $\mathrm{P}^{◱}_{a,b}$, $\mathrm{P}^{◲}_{a,b}$*). The defined seaweed submatrices are all disjoint; the defined score submatrices overlap by one row/column at the boundaries. In particular, the global LCS score* $\mathrm{H}_{a,b}(0, n)$ *belongs to all four score submatrices.*

The nonzeros of each seaweed submatrix introduced in Definition 20 can be regarded as an implicit solution to the corresponding component of the semi-local LCS problem. Similarly, by considering only three out of the four submatrices, we can define an implicit solution to the extended string-substring (respectively, substring-string) LCS problem.

Definition 21. *Given strings a, b, we define the* extended string-substring *(respectively,* substring-string*)* seaweed matrix *over* $\langle -m : n \mid 0 : m+n \rangle$ *as*

$$\mathrm{P}^{▙}_{a,b} = \begin{bmatrix} \mathrm{P}^{◰}_{a,b} & \cdot \\ \mathrm{P}^{◱}_{a,b} & \mathrm{P}^{◲}_{a,b} \end{bmatrix} \qquad \mathrm{P}^{▜}_{a,b} = \begin{bmatrix} \mathrm{P}^{◰}_{a,b} & \mathrm{P}^{◳}_{a,b} \\ \cdot & \mathrm{P}^{◲}_{a,b} \end{bmatrix}$$

The extended string-substring seaweed matrix $\mathrm{P}^{▙}_{a,b}$ contains at least n and at most $\min(m+n, 2n)$ nonzeros. Note that for $m \geq n$, the number of nonzeros in $\mathrm{P}^{▙}_{a,b}$ is at most $2n$, which is convenient when m is large. Analogously, for $m \leq n$, the number of nonzeros in the extended substring-string matrix $\mathrm{P}^{▜}_{a,b}$ is at most $2m$, which is convenient when n is large.

Let string a of length m be a concatenation of two fixed strings: $a = a'a''$, where a', a'' are nonempty strings of length m', m'' respectively, and $m = m' + m''$. A substring of the form $a\langle i' : i'' \rangle$ with $i' \in [0 : m'-1]$, $i'' \in [m'+1 : m]$ will be called a *cross-substring*. In other words, a cross-substring of a consists of a nonempty suffix of a' and a nonempty prefix of a''. A cross-substring that is a prefix or a suffix of a will be called a *cross-prefix* and a *cross-suffix*, respectively. Given string b of length n that is a concatenation of two fixed strings, $b = b'b''$, cross-substrings of b are defined analogously.

Definition 22. *Given strings $a = a'a''$ and b, the corresponding* cross-semi-local score matrix *is the submatrix* $\mathrm{H}_{a',a'';b} = \mathrm{H}_{a,b}[-m' : n \mid 0 : m''+n]$*.Symmetrically, given strings a and $b = b'b''$, the corresponding* cross-semi-local score matrix *is the submatrix* $\mathrm{H}_{a;b',b''} = \mathrm{H}_{a,b}[-m : n' \mid n' : m+n]$*.The* cross-semi-local seaweed matrices *are defined analogously:* $\mathrm{P}_{a',a'';b} = \mathrm{P}_{a,b}\langle -m' : n \mid 0 : m''+n\rangle$, $\mathrm{P}_{a;b',b''} = \mathrm{P}_{a,b}\langle -m : n' \mid n' : m+n\rangle$.

A cross-semi-local score matrix represents the solution of a restricted version of the semi-local LCS problem. In this version, instead of all substrings (prefixes, suffixes) of string a (respectively, b), we only consider cross-substrings (cross-prefixes, cross-suffixes). At the submatrix boundaries $\mathrm{H}_{a',a'';b}(*, m''+n)$ and $\mathrm{H}_{a',a'';b}(-m', *)$, cross-substrings of string a degenerate to suffixes of a' and prefixes of a''; in particular, cross-prefixes and cross-suffixes of a degenerate respectively to the whole a' and a''. The submatrix boundaries $\mathrm{H}_{a;b',b''}(*, n')$ and $\mathrm{H}_{a;b',b''}(n', *)$ correspond to similar degenerate cross-substrings of string b.

As before, the cross-semi-local seaweed matrix $\mathrm{P}_{a',a'';b}$ (respectively, $\mathrm{P}_{a;b',b''}$) gives an implicit representation for the corresponding score matrix $\mathrm{H}_{a',a'';b}$ (respectively, $\mathrm{H}_{a;b',b''}$).

Occasionally, we will use cross-semi-local score and seaweed matrices in combination with the superscript subrange notation, introduced earlier in this section. In such cases, the range of the resulting matrix will be determined by the intersection of the ranges implied by the superscript and the subscript. For example, matrix $\mathrm{H}^{◱}_{a;b',b''} = \mathrm{H}_{a,b}[0 : n' \mid n' : n]$ is the matrix of all LCS scores between string a and all cross-substrings of string $b = b'b''$.

2.5 Weighted scores and edit distances

The concept of LCS score is generalised by that of *(weighted) alignment score*. An *alignment* of strings a, b is obtained by putting a subsequence of a into one-to-one correspondence with a (not necessarily identical) subsequence of b, character by character and respecting the index order. The corresponding pair of characters, one from a and the other from b, are said to be *aligned*. A character that is not aligned against a character of another string is said to be aligned against a *gap* in that string. Each of the resulting character alignments is given a real *weight*:

- a pair of aligned matching characters has weight $w_{\mathsf{m}} \geq 0$;
- a pair of aligned mismatching characters has weight $w_{\mathsf{x}} < w_{\mathsf{m}}$;
- a gap-character or character-gap pair has weight $w_{\mathsf{g}} \leq \frac{1}{2}w_{\mathsf{x}}$; it is normally assumed that $w_{\mathsf{g}} \leq 0$ (i.e. this weight is in fact a penalty).

The intuition behind the weight inequalities is as follows: aligning a matching pair of characters is always better than aligning a mismatching pair of characters, which in its turn is never worse than leaving both characters unaligned (aligned against a gap).

Definition 23. *The* (weighted) alignment score *for strings a, b is the maximum total weight of character pairs in an alignment of a against b.*

We define the *semi-local (weighted) alignment score problem* and its component (string-substring, etc.) subproblems by straightforward extension of Definition 14. The concepts of alignment dag and score matrix can be naturally generalised to the weighted case. To distinguish between the weighted and unweighted cases, we will use a script font in the corresponding notation.

The weighted alignment of strings a, b corresponds to a *weighted alignment dag* $\mathcal{G}_{a,b}$, where diagonal match edges, diagonal mismatch edges, and horizontal/vertical edges have weight w_{m}, w_{x}, w_{g}, respectively. A semi-local alignment score corresponds to a boundary-to-boundary highest-scoring path in $\mathcal{G}_{a,b}$. The complete output of the semi-local alignment score problem is a *semi-local (weighted) score matrix* $\mathcal{H}_{a,b}$. This matrix is anti-Monge; however, in contrast with the unweighted case, it is not necessarily unit-anti-Monge.

Given an arbitrary set of alignment weights, it is often convenient to normalise them so that $0 = w_{\mathsf{g}} \leq w_{\mathsf{x}} < w_{\mathsf{m}} = 1$. To obtain such a normalisation, first observe that, given a pair of strings a, b, and arbitrary weights $w_{\mathsf{m}} \geq 0$, $w_{\mathsf{x}} < w_{\mathsf{m}}$, $w_{\mathsf{g}} \leq \frac{1}{2}w_{\mathsf{x}}$, we can replace the weights respectively by $w_{\mathsf{m}} + 2x$, $w_{\mathsf{x}} + 2x$, $w_{\mathsf{g}} + x$, for any real x. This weight transformation increases the score of every global alignment (top-left to bottom-right) path in $\mathrm{G}_{a,b}$ by $(m+n)x$. Therefore, the relative scores of different global alignment paths do not change. In particular, the maximum global alignment score is attained by the same path as before the transformation. By taking $x = -w_{\mathsf{g}}$, and dividing the resulting weights by $w_{\mathsf{m}} - 2w_{\mathsf{g}} > 0$, we achieve the desired normalisation. (A similar method is used e.g. by Rice et al. [15]).

Definition 24. *Given original weights w_{m}, w_{x}, w_{g}, the corresponding* normalised weights *are $w^*_{\mathsf{m}} = 1$, $w^*_{\mathsf{x}} = \frac{w_{\mathsf{x}}-2w_{\mathsf{g}}}{w_{\mathsf{m}}-2w_{\mathsf{g}}}$, $w^*_{\mathsf{g}} = 0$. We call the corresponding alignment score the* normalised score. *The original alignment score h can be restored from the normalised score h^* by reversing the normalisation: $h = h^* \cdot (w_{\mathsf{m}} - 2w_{\mathsf{g}}) + (m+n) \cdot w_{\mathsf{g}}$.*

Thus, for fixed string lengths m and n, maximising the normalised global alignment score h^* is equivalent to maximising the original score h. However, more care is needed when maximising the alignment score across variable strings of different lengths, e.g. in the context of semi-local alignment. In such cases, an explicit conversion from normalised weights to original weights will be necessary prior to the maximisation.

Definition 25. *A set of character alignment weights will be called* rational, *if all the weights are rational numbers.*

Given a rational set of normalised weights, the semi-local alignment score problem on strings a, b can be reduced to the semi-local LCS problem by the following *blow-up* procedure. Let $w_{\mathsf{x}} = \frac{\mu}{\nu} < 1$, where μ, ν are positive natural numbers. We transform input strings a, b of lengths m, n into new *blown-up* strings $\tilde{a}$, $\tilde{b}$ of lengths $\tilde{m} = \nu m$, $\tilde{n} = \nu n$. The transformation consists in replacing every character γ in each of the strings by a substring $\$^{\mu}\gamma^{\nu-\mu}$ of length ν (recall that $\$$ is a special guard character, not present in the original strings). We have

$$\mathcal{H}_{a,b}(i,j) = \tfrac{1}{\nu} \cdot \mathrm{H}_{\tilde{a},\tilde{b}}(\nu i, \nu j)$$

for all $i \in [-m : n]$, $j \in [0 : m+n]$, where the matrix $\mathcal{H}_{a,b}$ is defined by the normalised weights on the original strings a, b, and the matrix $\mathrm{H}_{\tilde{a},\tilde{b}}$ by the LCS weights on the blown-up strings $\tilde{a}$, $\tilde{b}$. Therefore, all the techniques of the previous chapters apply to the rational-weighted semi-local alignment score problem, assuming that ν is a constant.

An important special case of weighted string alignment is the *edit distance problem.* Here, the characters are assumed to match "by default": $w_{\mathsf{m}} = 0$. The mismatches and gaps are penalised: $2w_{\mathsf{g}} \leq w_{\mathsf{x}} < 0$. The resulting score is always nonpositive. Equivalently, we regard string a as being transformed into string b by a sequence of weighted *character edits*:

- character insertion or deletion (*indel*) has weight $-w_{\mathsf{g}} > 0$;
- character *substitution* has weight $-w_{\mathsf{x}} > 0$.

Definition 26. *The* (weighted) edit distance *between strings a, b is the minimum total weight of a sequence of character edits transforming a into b. Equivalently, it is the (nonnegative) absolute value of the corresponding (nonpositive) alignment score.*

In the rest of this work, the edit distance problem will be treated as a special case of the weighted alignment problem. In particular, all the techniques of the previous sections apply to the semi-local edit distance problem, as long as the character edit weights are rational.

3 Threshold approximate matching in compressed strings

Given a matrix A and a threshold h, it will be convenient to denote the subset of entries above the threshold by $\tau_h(A) = \big\{(i,j), \text{ such that } A(i,j) \geq h\big\}$. The threshold approximate matching problem (Definition 2) corresponds to all points in the set $\tau_h(\mathcal{H}^{\text{◱}}_{p,t})$.

Using the techniques of the previous sections, we now show how the threshold approximate matching problem on a GC-text can be solved more efficiently, assuming a sufficiently high value of the edit distance threshold k. The algorithm extends Algorithms 1 and 2 of [21], and assumes an arbitrary rational-weighted alignment score. As in Algorithm 2 of [21], we assume for simplicity the constant-time index arithmetic, keeping the index remapping implicit. The details of index remapping used to lift this assumption can be found in Algorithm 1 of [21].

Algorithm 1 (Threshold approximate matching).
Parameters: character alignment weights w_{m}, w_{x}, w_{g}, assumed to be constant rationals.
Input: plain pattern string p of length m; SLP of length $\bar{n}$, generating text string t of length n; score threshold h.
Output: locations (or count) of matching substrings in t.
Description.
First phase. Recursion on the input SLP generating t.

To reduce the problem to an unweighted LCS score, we apply the blow-up technique described in Subsection 2.5. Consider the normalised weights (Definition 24), and define the corresponding blown-up strings $\tilde{p}$, $\tilde{t}$ of length $\tilde{m} = \nu m$, $\tilde{n} = \nu n$, respectively.
Recursion base: $n = \bar{n} = 1$, $\tilde{n} = \nu$. The extended substring-string seaweed matrix $\mathrm{P}^{\text{▜}}_{\tilde{p},\tilde{t}}$ can be computed by the seaweed algorithm [20,17] in ν linear sweeps of string $\tilde{p}$. This matrix can be used to query the LCS score $\mathrm{H}_{\tilde{p},\tilde{t}}(0,\nu)$ between $\tilde{p}$ and $\tilde{t}$. String t is matching, if and only if the corresponding weighted alignment score $\mathcal{H}_{p,t}(0,1)$ is at least h.

Recursive step: $n \geq \bar{n} > 1$, $\tilde{n} = \nu n$. Let $t = t't''$ be the SLP statement defining string t. We have $\tilde{t} = \tilde{t}'\tilde{t}''$ for the corresponding blown-up strings.

As in Algorithm 2 of [21], we obtain recursively the extended substring-string seaweed matrix $\mathrm{P}^{\blacksquare}_{\tilde{p},\tilde{t}}$ and the cross-semi-local seaweed matrix $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$ by an application of the fast algorithm for distance multiplication of unit-Monge matrices [16]. These two subpermutation matrices are typically very sparse: $\mathrm{P}^{\blacksquare}_{\tilde{p},\tilde{t}}$ contains at most $2\tilde{m} = 2\nu m$ nonzeros, and $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$ exactly $\tilde{m} = \nu m$ nonzeros.

In contrast to Algorithm 2 of [21], it is no longer sufficient to consider just the $\gtrless$-maximal nonzeros of $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$; we now have to consider all its $\tilde{m}$ nonzeros. Let us denote the indices of these nonzeros, in increasing order independently for each dimension, by

$$\hat{\imath}_{0^+} < \hat{\imath}_{1^+} < \cdots < \hat{\imath}_{\tilde{m}^-} \qquad \hat{\jmath}_{0^+} < \hat{\jmath}_{1^+} < \cdots < \hat{\jmath}_{\tilde{m}^-} \tag{1}$$

These two index sequences define an $\tilde{m} \times \tilde{m}$ non-contiguous permutation submatrix of $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$:

$$P(\hat{s}, \hat{t}) = \mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}(\hat{\imath}_{\hat{s}}, \hat{\jmath}_{\hat{t}}) \tag{2}$$

for all $\hat{s}, \hat{t} \in \langle 0 : \tilde{m} \rangle$.

Index sequence $\hat{\imath}_{\hat{s}}$ (respectively, $\hat{\jmath}_{\hat{t}}$) partitions the range $[-\tilde{m} : \tilde{n}']$ (respectively, $[\tilde{n}' : \tilde{m} + \tilde{n}]$) into $\tilde{m} + 1$ disjoint non-empty intervals of varying lengths. Therefore, we have a partitioning of the cross-semi-local score matrix $\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}$ into $(\tilde{m} + 1)^2$ disjoint non-empty rectangular *H-blocks* of varying dimensions. Consider an arbitrary H-block

$$\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}\left[\hat{\imath}^{+}_{u^-} : \hat{\imath}^{-}_{u^+} \mid \hat{\jmath}^{+}_{v^-} : \hat{\jmath}^{-}_{v^+}\right] \tag{3}$$

where $u, v \in [0 : \tilde{m}]$. For the boundary H-blocks, some of the above bounds are not defined. In these cases, we let $\hat{\imath}_{0^-} = -\tilde{m}$, $\hat{\imath}_{\tilde{m}^+} = \tilde{n}'$, $\hat{\jmath}_{0^-} = \tilde{n}'$, $\hat{\jmath}_{\tilde{m}^+} = \tilde{m} + \tilde{n}$.

Since the boundaries of an H-block (3) are given by adjacent pairs of indices from (1), we have $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}(\hat{\imath}, \hat{\jmath}) = 0$ for all $\hat{\imath} \in \left\langle \hat{\imath}^{+}_{u^-} : \hat{\imath}^{-}_{u^+} \right\rangle$, $u \in [0 : \tilde{m}]$ and arbitrary $\hat{\jmath}$, as well as for arbitrary $\hat{\imath}$ and all $\hat{\jmath} \in \left\langle \hat{\jmath}^{+}_{v^-} : \hat{\jmath}^{-}_{v^+} \right\rangle$, $v \in [0 : \tilde{m}]$. Therefore, given a fixed H-block, all its points are $\gtrless$-dominated by some fixed set of nonzeros in $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$ (and hence also in $\mathrm{P}_{\tilde{p},\tilde{t}}$). The number of nonzeros in this set is

$$d = \mathrm{P}^{T\Sigma T}_{\tilde{p};\tilde{t}',\tilde{t}''}(\hat{\imath}^{-}_{u^+}, \hat{\jmath}^{+}_{v^-}) = \mathrm{P}^{T\Sigma T}_{\tilde{p},\tilde{t}}(\hat{\imath}^{-}_{u^+}, \hat{\jmath}^{+}_{v^-}) \tag{4}$$

where the H-block's bottom-left ($\gtrless$-minimal) point $(\hat{\imath}^{-}_{u^+}, \hat{\jmath}^{+}_{v^-})$ is chosen arbitrarily to be its reference point. Since the value of d is constant across the H-block, all its entries have identical value: we have

$$\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}(i, j) = \tilde{m} - d \tag{5}$$

for all $i \in [\hat{\imath}^{+}_{u^-} : \hat{\imath}^{-}_{u^+}]$, $j \in [\hat{\jmath}^{+}_{v^-} : \hat{\jmath}^{-}_{v^+}]$.

We now switch our focus from the blown-up strings $\tilde{p}$, $\tilde{t}'$, $\tilde{t}''$ back to the original strings p, t', t''. The partitioning of the LCS score matrix $\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}$ into H-blocks induces a partitioning of the alignment score matrix $\mathcal{H}_{p;t',t''}$ into $(\tilde{m} + 1)^2$ disjoint rectangular *$\mathcal{H}$-blocks* of varying dimensions. The $\mathcal{H}$-block corresponding to H-block (3) is

$$\mathcal{H}_{p;t',t''}\left[\hat{\imath}^{(+)}_{u^-} : \hat{\imath}^{(-)}_{u^+} \mid \hat{\jmath}^{(+)}_{v^-} : \hat{\jmath}^{(-)}_{v^+}\right] \tag{6}$$

where we denote $\hat\imath^{(-)} = \left\lfloor \frac{1}{\nu}\hat\imath \right\rfloor$, $\hat\imath^{(+)} = \left\lceil \frac{1}{\nu}\hat\imath \right\rceil$, for any $\hat\imath$. Note that, although an H-block (3) is by definition non-empty, the corresponding $\mathcal{H}$-block (6) may be empty: we have $\hat\imath^{(+)}_{u^-} > \hat\imath^{(-)}_{u^+}$ (respectively, $\hat\jmath^{(+)}_{v^-} > \hat\jmath^{(-)}_{v^+}$), if the interval $\left[\hat\imath^{+}_{u^-} : \hat\imath^{-}_{u^+}\right]$ (respectively, $\left[\hat\jmath^{+}_{v^-} : \hat\jmath^{-}_{v^+}\right]$) contains no multiples of ν.

Although all entries within an H-block (3) are constant, the entries within the corresponding $\mathcal{H}$-block (6) will typically vary. By (5) and Definition 24, we have

$$\mathcal{H}_{p;t',t''}(i,j) = \tfrac{\tilde m - d}{\nu} \cdot (w_{\mathsf{m}} - 2w_{\mathsf{g}}) + (m + j - i) \cdot w_{\mathsf{g}} \tag{7}$$

where $i \in \left[\hat\imath^{(+)}_{u^-} : \hat\imath^{(-)}_{u^+}\right]$, $j \in \left[\hat\jmath^{(+)}_{v^-} : \hat\jmath^{(-)}_{v^+}\right]$. Recall that $w_{\mathsf{g}} \leq 0$. Therefore, the score within an $\mathcal{H}$-block is maximised when $j - i$ is minimised, so the maximum score is attained by the block's bottom-left (i.e. $\gtrless$-minimal) entry $\mathcal{H}_{p;t',t''}\bigl(\hat\imath^{(-)}_{u^+}, \hat\jmath^{(+)}_{v^-}\bigr)$. If $w_{\mathsf{g}} < 0$, then this maximum is strict; otherwise, we have $w_{\mathsf{g}} = 0$, and all the entries in the $\mathcal{H}$-block have an identical value $\frac{\tilde m - d}{\nu} \cdot w_{\mathsf{m}}$.

Without loss of generality, let us now assume that all the $\mathcal{H}$-blocks (6) are non-empty. We are interested in the bottom-left entries attaining block maxima, taken across all the $\mathcal{H}$-blocks. The leftmost column and the bottom row of these entries (respectively $\mathcal{H}_{p;t',t''}\bigl(\hat\imath^{(-)}_{u^+}, n'\bigr)$ and $\mathcal{H}_{p;t',t''}\bigl(n', \hat\jmath^{(+)}_{v^-}\bigr)$ for all u, v) lie on the boundary of matrix $\mathcal{H}_{p;t',t''}$, and correspond to comparing p against respectively suffixes of t' and prefixes of t'', rather than cross-substrings of t. After excluding such boundary entries, the remaining block maxima form an $\tilde m \times \tilde m$ non-contiguous submatrix $H(u,v) = \mathcal{H}_{p;t',t''}\bigl(\hat\imath^{(-)}_{u^+}, \hat\jmath^{(+)}_{v^-}\bigr)$, where $u \in [0 : \tilde m - 1]$, $v \in [1 : \tilde m]$.

Consider the string-substring submatrix of H (i.e. the submatrix of entries that correspond to the comparison of a string against a cross-substring, as opposed to a prefix against a cross-suffix, or a suffix against a cross-prefix): $H^{\text{◰}} = \bigl(H(u,v) : \bigl(\hat\imath^{(-)}_{u^+}, \hat\jmath^{(+)}_{v^-}\bigr) \in [0 : n' \mid n' : n]\bigr)$. Since matrix $\mathcal{H}_{p;t',t''}$ is anti-Monge, its submatrices H and $H^{\text{◰}}$ are also anti-Monge.

We now need to obtain the row maxima of matrix $H^{\text{◰}}$. Let

$$N(u,v) = \frac{\nu}{2w_{\mathsf{g}} - w_{\mathsf{m}}} H^{\text{◰}}(u,v) = P^{T\Sigma T}(u,v) - \tilde m + \frac{\nu\bigl(m + \hat\jmath^{(+)}_{v^-} - \hat\imath^{(-)}_{u^+}\bigr) w_{\mathsf{g}}}{2w_{\mathsf{g}} - w_{\mathsf{m}}} \qquad \text{(By (7), (4), (2))}$$

Since $2w_{\mathsf{g}} - w_{\mathsf{m}} < 0$, the problem of finding row maxima of $H^{\text{◰}}$ is equivalent to finding row minima of matrix N, or, equivalently, column minima of the transpose matrix N^T. This matrix (and therefore N itself) is subunit-Monge: we have $N^T(v,u) = N(u,v) = P^{T\Sigma}(v,u) + b(u) + c(v)$, where $b(u) = -\frac{\nu \hat\imath^{(-)}_{u^+} \cdot w_{\mathsf{g}}}{2w_{\mathsf{g}} - w_{\mathsf{m}}}$, $c(v) = -\tilde m + \frac{\nu\left(m + \hat\jmath^{(+)}_{v^-}\right) w_{\mathsf{g}}}{2w_{\mathsf{g}} - w_{\mathsf{m}}}$. Therefore, the column minima of N^T can be found by Lemma 12 (replacing row minima with column minima by symmetry).

The set $\tau_h(H^{\text{◰}})$ of all entries in $H^{\text{◰}}$ scoring above the threshold h, and therefore the set of all matching cross-substrings of a, can now be obtained by a local search in the neighbourhoods of the row maxima. (End of recursive step)

Second phase. For every SLP symbol, we now have the relative locations of its minimally matching cross-substrings. It is now straightforward to obtain their absolute locations and/or their count (as in Algorithm 2 of [21], substituting “matching” for “minimally matching”).

Cost analysis.

First phase. Each seaweed matrix multiplication runs in time $O(\tilde m \log \tilde m) = O(m \log m)$. The algorithm of Lemma 12 runs in time $O(\tilde m \log\log \tilde m) = O(m \log\log m)$.

Hence, the running time of a recursive step is $O(m \log m)$. There are $\bar{n}$ recursive steps in total, therefore the whole recursion runs in time $O(m \log m \cdot \bar{n})$.
Second phase. For every SLP symbol, there are at most $m - 1$ minimally matching cross-substrings. Given the output of the first phase, the absolute locations of all minimally matching substrings in t can be reported in time $O(m\bar{n} + \mathit{output})$.
Total. The overall running time is $O(m \log m \cdot \bar{n} + \mathit{output})$.

Algorithm 1 improves on the algorithm of [11], as long as $k = \omega\bigl((\log m)^{1/2}\bigr)$ in the case of general GC-compression, and $k = \omega(\log m)$ in the case of LZ78 or LZW compression. Algorithm 1 also improves on the algorithms of [4,5], as long as $k = \omega\bigl((m \log m)^{1/4}\bigr)$, in the case of both general GC-compression and LZ78 or LZW compression.

4 Conclusions

We have obtained a new efficient algorithm for threshold approximate matching between a GC-text and a plain pattern. Our algorithm is of interest not only in its own right, but also as a natural application of fast unit-Monge matrix multiplication, developed in our previous works. We have also demonstrated a new technique for incremental searching in an implicit totally monotone matrix, which which we believe to be of independent interest.

References

1. A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber: *Geometric applications of a matrix-searching algorithm.* Algorithmica, 2(1) 1987, pp. 195–208.
2. A. Apostolico and C. Guerra: *The longest common subsequence problem revisited.* Algorithmica, 2(1) 1987, pp. 315–336.
3. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in Proceedings of the 7th SPIRE, 2000, pp. 39–48.
4. P. Bille, R. Fagerberg, and I. L. Gørtz: *Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts.* ACM Transactions on Algorithms, 6(1) 2009, pp. 3:1–3:14.
5. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann: *Random access to grammar-compressed strings*, in Proceedings of the 22nd ACM-SIAM SODA, 2011, pp. 373–389.
6. P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich: *Window subsequence problems for compressed texts*, in Proceedings of CSR, vol. 3967 of Lecture Notes in Computer Science, 2006, pp. 127–136.
7. K.-M. Chao and L. Zhang: *Sequence Comparison: Theory and Methods*, vol. 7 of Computational Biology Series, Springer, 2009.
8. R. Cole and R. Hariharan: *Approximate string matching: A simpler faster algorithm.* SIAM Journal on Computing, 31(6) 2002, pp. 1761–1782.
9. P. Gawrychowski: *Faster algorithm for computing the edit distance between SLP-compressed strings*, in Proceedings of SPIRE, vol. 7608 of Lecture Notes in Computer Science, 2012, pp. 229–236.
10. D. Hermelin, G. M. Landau, S. Landau, and O. Weimann: *A unified algorithm for accelerating edit-distance computation via text-compression*, in Proceedings of the 26th STACS, 2009, pp. 529–540.
11. J. Kärkkäinen, G. Navarro, and E. Ukkonen: *Approximate string matching on Ziv–Lempel compressed text.* Journal of Discrete Algorithms, 1 2003, pp. 313–338.
12. G. M. Landau and U. Vishkin: *Fast parallel and serial approximate string matching.* Journal of Algorithms, 10(2) 1989, pp. 157–169.

13. V. Levenshtein: *Binary codes capable of correcting spurious insertions and deletions of ones.* Problems of Information Transmission, 1 1965, pp. 8–17.
14. M. Lohrey: *Algorithmics on SLP-compressed strings: a survey.* Groups Complexity Cryptology, 4(2) 2012, pp. 241–299.
15. S. V. Rice, H. Bunke, and T. A. Nartker: *Classes of cost functions for string edit distance.* Algorithmica, 18 1997, pp. 271–280.
16. A. Tiskin: *Fast distance multiplication of unit-Monge matrices.* Algorithmica, To appear.
17. A. Tiskin: *Semi-local string comparison: Algorithmic techniques and applications*, Tech. Rep. 0707.3619, arXiv.
18. A. Tiskin: *Semi-local longest common subsequences in subquadratic time.* Journal of Discrete Algorithms, 6(4) 2008, pp. 570–581.
19. A. Tiskin: *Semi-local string comparison: Algorithmic techniques and applications.* Mathematics in Computer Science, 1(4) 2008, pp. 571–603.
20. A. Tiskin: *Periodic string comparison*, in Proceedings of CPM, vol. 5577 of Lecture Notes in Computer Science, 2009, pp. 193–206.
21. A. Tiskin: *Towards approximate matching in compressed strings: Local subsequence recognition*, in Proceedings of CSR, vol. 6651 of Lecture Notes in Computer Science, 2011, pp. 401–414.
22. T. A. Welch: *A technique for high-performance data compression.* Computer, 17(6) 1984, pp. 8–19.
23. T. Yamamoto, H. Bannai, S. Inenaga, and M. Takeda: *Faster subsequence and don't-care pattern matching on compressed texts*, in Proceedings of CPM, vol. 6661 of Lecture Notes in Computer Science, 2011, pp. 309–322.
24. G. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.