Journal of Discrete Algorithms 25 (2014) 34–41

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Simple and efficient LZW-compressed multiple pattern matching

Paweł Gawrychowski [a,b,1]

[a] Institute of Computer Science, University of Wrocław, Wrocław, Poland
[b] Max-Planck-Institut für Informatik, Saarbrücken, Germany

ARTICLE INFO

Article history:
Available online 31 October 2013

Keywords:
Multiple pattern matching
Lempel–Ziv–Welch compression

ABSTRACT

We consider a natural variant of the classical multiple pattern matching problem: given a Lempel–Ziv–Welch representation of a string and a collection of (uncompressed) patterns, does any of them occur in the text?

As shown by Kida et al. [15], extending the single pattern algorithm of Amir, Benson and Farach [2] gives a running time of $\mathcal{O}(n + M^2)$ for the more general case, where n is the number of codewords in the compressed representation of the text and M is the sum of the length of all patterns. We prove that in fact it is possible to achieve $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ complexity. While not linear, running times of our solutions match the single pattern bounds achieved by the previously known solutions [2,17] in a more structured and unified manner, and without using any combinatorics on words. The only nontrivial components of our method are suffix arrays, constant time range minimum queries, and balanced binary search trees.

1. Introduction

Pattern matching is the most natural problem concerning processing text data. It has been thoroughly studied, and many different linear time solutions are known, starting from the well-known Knuth–Morris–Pratt algorithm [16]. While it might seem that the existence of linear time, constant space [10], and constant delay [9] solutions means that the question is completely solved, this is not quite the case. Whenever we must store a lot of text data, we store it in a compressed representation. This suggests a natural research direction: could we process this compressed representation without wasting time (and space) to uncompress it? Or, in other words, can we use the high compression ratio to accelerate the computation?

It turns out that for pattern matching and some compression methods, the answer is yes. For the case of Lempel–Ziv–Welch [18] compressed text, there are two algorithms given by Amir, Benson, and Farach [2]: one with a $\mathcal{O}(n+m^2)$ running time, and one with $\mathcal{O}(n \log m + m)$, where m is the length of the pattern and n the size of the compressed representation of a text $t[1..N]$. Farach and Thorup [7] considered the more general case of Lempel–Ziv compression, and developed a (randomized) $\mathcal{O}(n \log^2 \frac{N}{n} + m)$ time algorithm. When the compression used is Lempel–Ziv–Welch, their complexity reduces to $\mathcal{O}(n \log \frac{N}{n} + m)$. In a recent paper we proved that in fact it is possible to achieve a (deterministic) linear running time for this case [12], even if both the pattern and the text are compressed [11]. Another interesting generalization is constructing compressed full-text self-indices, where we want to augment a compressed representation of a text with some small additional data so that later we can detect an occurrence of any pattern efficiently. This setting is different than ours, though, because of how the size of the additional data is measured. For instance, the best currently known LZ78 (which is the same

E-mail address: gawry@cs.uni.wroc.pl.
[1] Supported by MNiSW grant number N N206 492638, 2010–2012 and START scholarship from FNP.

1570-8667/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jda.2013.10.004

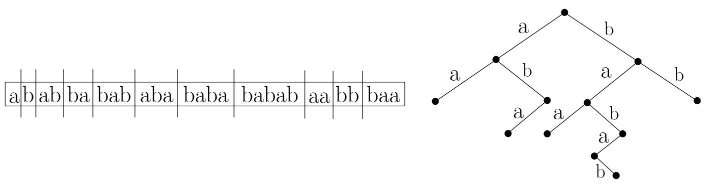

Fig. 1. An example of an LZW-compressed text and a trie storing all of its codewords.

as Lempel–Ziv–Welch from our point of view) based self-index [3] uses $(3+\epsilon)NH_k(t)+o(N\log\sigma)$ space, where $H_k(t)$ is the k-th order empirical entropy of the text and σ is the size of the alphabet, and locates an occurrence of a pattern in time $\mathcal{O}(m\log N)$. A closer inspection of the construction shows that $o(N\log\sigma)$ hides $\mathcal{O}(N\frac{\log\log N}{\log_\sigma N})$, which is probably acceptable from a purely practical point of view, but in theory can be almost quadratic in terms of the input size n. This can be avoided if we are satisfied with an $\mathcal{O}(\frac{m^2}{\epsilon}+m\log N)$ query time, but this (again) can be quadratic in theory.

A natural research direction is to consider *multiple pattern matching*, where instead of just one pattern we are given a collection of patterns $p_1, p_2, \ldots, p_\ell$ (which, for example, can be a set of forbidden words from a dictionary), and we should check if any of them occurs in the text. It is known that extending one of the algorithms given by Amir et al. results in a $\mathcal{O}(n+M^2)$ running time for multiple Lempel–Ziv–Welch-compressed pattern matching, where $M=\sum_i |p_i|$ [15]. It seems realistic that the set of the patterns is very large, and hence M^2 addend in the running time might be substantially larger than n. In this paper we prove that in fact it is possible to achieve $\mathcal{O}(n\log M+M)$ or $\mathcal{O}(n+M^{1+\epsilon})$ complexity for this problem, with the space usage being $\mathcal{O}(n+M)$ and $\mathcal{O}(n+M^{1+\epsilon})$, respectively. While such running time is not linear, it matches the best bounds for the single pattern case developed by Amir et al. [2] and Kosaraju [17] in a unified and structured manner, and is achieved using rather simple means.

The main tool in our algorithms is reducing the problem to simple-to-state purely geometrical questions on an integer grid. Similar approach can be found, for example, in [5], but instead of using a black-box solution for 2D range reporting, we look more carefully at the properties of the resulting problem, and develop a hand-tailored (but elementary) solution. Many of the previous solutions for similar problems used combinatorics on words, but such methods seem difficult to generalize when we want to operate on multiple patterns. For example, an important tool in LZW-compressed single pattern matching is the *border tree* [13], but when we move to multiple patterns it is not clear how to generalize the definition so that the depth of the tree is still logarithmic.

2. Preliminaries

Let $M=\sum_{i=1}^{\ell}|p_i|$ be the sum of the lengths of all patterns. We assume that the alphabet Σ is either constant (which is the simple case) or consists of integers which can be sorted in linear time (in other words, polynomial in n and M). In the latter case we renumber the letters and assume the alphabet to be $\{0,1,\ldots,M-1\}$. We consider strings of length N over Σ given in a Lempel–Ziv–Welch compressed form which are represented as a sequence of n *codewords* (sometimes called *blocks*) where a codeword is either a single letter, or a previously occurring codeword concatenated with a single character (using both n and N might be confusing, but it is enough to remember that big N stands for the big original size, while small n refers to the hopefully small compressed size; similarly, big M denotes the big original size of all patterns). This additional character is not given explicitly: we define it as the first character of the next codeword, and initialize the set of codewords to contain all single characters in the very beginning (this is a technical detail distinguishing LZW from LZ78, which is not important to us). The resulting compression method enjoys a particularly simple encoding/decoding process, but unfortunately requires outputting at least $\Omega(\sqrt{N})$ codewords. Still, its simplicity and good compression ratio achieved on real life instances make it an interesting model to work with. For the rest of the paper we will use LZW when referring to Lempel–Ziv–Welch compression. An example of an LZW-compressed text, together with a trie storing all codewords, is shown in Fig. 1.

We use the following notion for a string w: prefix(w) is the longest prefix of w which is a suffix of some pattern, suffix(w) is the longest suffix of w which is a prefix of some pattern, and w^r is its reversal.

To prove the main theorem we need to design a few data structures. To simplify the exposition we use the notion of an $\langle f(M), g(M)\rangle$ *structure* meaning that after an $f(M)$ time preprocessing we are able to execute one query in $g(M)$ time. If such structure is *offline*, we are able to execute a sequence of t queries in total $f(M)+tg(M)$ time. Similarly, an $\langle f(M), g(M)\rangle$ *dynamic structure* allows updates in $f(M)$ time and queries in $g(M)$ time. It is *persistent* if updating creates a new copy instead of modifying the original data. The notion of persistence is well-studied, see for example [6].

We will extensively use the suffix tree T and the suffix array built for concatenation of all patterns separated by a special character \$ (which does not occur in either the text or any pattern, and is smaller than any original letter) which we call A:

$$A=p_1\$p_2\$\ldots\$p_{\ell-1}\$p_\ell$$

Both structures can be constructed in linear time [7,14]. Similarly, T^r is the suffix tree built for the reversed concatenation A^r:

$$A^r = p^r_\ell\$p^r_{\ell-1}\$\ldots\$p^r_2\$p^r_1$$

In both suffix trees we additionally make all nodes corresponding to suffixes of the patterns explicit. Both suffix arrays are enriched with range minimum query structures enabling us to compute the longest common prefix and suffix of any two substrings in constant time.

Lemma 1. *(See [4].) Given an array of integers $t[1..n]$ we can build in linear time and space a range minimum/maximum query structure RMQ(t) which allows computing the minimum/maximum $t[k]$ over all $k \in \{i, i+1, \ldots, j\}$ for a given i, j in constant time.*

Lemma 2. *(See [14,4].) Array A can be preprocessed in linear time so that given any two fragments $A[i..i+k]$ and $A[j..j+k]$ we can find their longest common prefix LCP$(A[i..i+k], A[j..j+k])$ or longest common suffix LCS$(A[i..i+k], A[j..j+k])$ in constant time.*

A snippet is any substring of any pattern $p_i[j..k]$. We represent it as a triple (i, j, k). Given such triple, we would like to retrieve the corresponding (explicit or implicit) node in the suffix tree (or reversed suffix tree) efficiently. $\langle f(M), g(M)\rangle$ *locator* allows $g(M)$ time retrieval after an $f(M)$ time preprocessing.

Lemma 3. *$\langle\mathcal{O}(M), \mathcal{O}(\log M)\rangle$ locator exists.*

Proof. $\langle\mathcal{O}(M\log M), \mathcal{O}(\log M)\rangle$ is very simple to implement: for each vertex of the suffix tree we construct a balanced search tree containing all its ancestors sorted according to their depths. Constructing the tree for a vertex requires inserting just one new element into its parent tree (note that most standard balanced binary search trees can be made persistent so that inserting a new number creates a new copy and does not destroy the old one) and so the whole construction takes $\mathcal{O}(M\log M)$ time. This is too much by a factor of $\log M$, though. We use the micro–macro tree decomposition [8] to shave it. The suffix tree is partitioned into small subtrees by choosing at most $\frac{M}{\log M}$ macro nodes such that after removing them we get a collection of connected components of at most logarithmic size. Such partition can be easily found in linear time. Then for each macro node we construct a binary search tree containing all its macro ancestors sorted according to their depths. There are just $\frac{M}{\log M}$ macro nodes so the whole preprocessing is linear. To find the ancestor v at depth d we first retrieve the lowest macro ancestor u of v by following at most $\log M$ edges up from v. If none of the traversed vertices is the answer, we find the macro ancestor of u of largest depth not smaller than d using the binary search tree in $\mathcal{O}(\log M)$ time. Then retrieving the answer requires following at most $\log M$ edges up from u. □

To improve the query time in the above lemma we need to replace the balanced search tree. $\langle f(M), g(M)\rangle$ *dynamic dictionary* stores a subset S of $\{0, 1, \ldots, M-1\}$ so that we can add or remove elements in $\mathcal{O}(M^\epsilon)$ time, and check if a given x belongs to S (and if so, retrieve its associated information) or find its successor and predecessor in $\mathcal{O}(1)$ time.

Lemma 4. *$\langle\mathcal{O}(M^\epsilon), \mathcal{O}(1)\rangle$ persistent dynamic dictionary exists for any $\epsilon > 0$.*

Proof. Choose an integer $k \geqslant \frac{1}{\epsilon}$. The idea is to represent the numbers in base $B = M^{\frac{1}{k}}$ and store them in a trie of depth k. At each vertex of the trie we maintain a table child$[0..B-1]$ with the i-th element containing the pointer to the corresponding child, if any. This allows efficient checking if a given x belongs to the current set (we just inspect at most k vertices and at each of them use the table to retrieve the next one in constant time). Note that we do not create a vertex if its corresponding tree is empty. To find the successor (or predecessor) efficiently, we maintain at each vertex two additional tables next$[0..B-1]$ and prev$[0..B-1]$ where next$[i]$ is the smallest $j \geqslant i$ such that child$[j]$ is defined and prev$[i]$ is the largest $j \leqslant i$ such that child$[j]$ is defined. Using those tables the running time becomes $\mathcal{O}(k) = \mathcal{O}(1)$. Whenever we add or remove an element, we must recalculate the tables at all vertices from the traversed path. Its length is k and each table is of size B so the updates require $\mathcal{O}(kB) = \mathcal{O}(M^\epsilon)$ time. Note that the whole structure is easily made persistent as after each update we create a new copy of the traversed path and do not modify any other vertices. □

Lemma 5. *$\langle\mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1)\rangle$ locator exists for any $\epsilon > 0$.*

Proof. The idea is the same as in Lemma 3: for each vertex of the suffix tree we construct a structure containing all its ancestors sorted according to their depths. Note that the depths are smaller than M so we can apply Lemma 4. The total construction time is $\mathcal{O}(M \times M^\epsilon) = \mathcal{O}(M^{1+\epsilon})$ and answering a query reduces to one predecessor lookup. □

We assume the following preprocessing for both the suffix tree and the reversed suffix tree.

Lemma 6. *A suffix tree built for a text of length M can be preprocessed in linear time so that given an implicit or explicit vertex v we can retrieve its pre- and post-order numbers* (pre(v) *and* post(v), *respectively*) *in the uncompressed version of the tree* (*i.e., in the suffix trie*) *in constant time.*

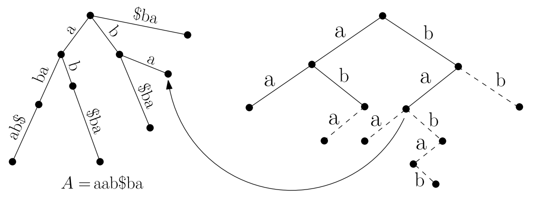

Fig. 2. Intersecting the trie from the example with the suffix tree constructed for $p_1 = aab$ and $p_2 = ba$. Edges which do not belong to the intersection are dashed.

Proof. For each explicit vertex, we store its pre- and post-order numbers in the suffix trie. To compute the numbers for an implicit vertex, we use the data stored at its lowest explicit ancestor. □

3. Overview of the algorithm

We are given a collection of patterns and a sequence of blocks, each block being either a single letter, or a previously defined block concatenated with a single letter. For each block we would like to check if the corresponding word occurs in any of the patterns, and if not, we would like to find its longest suffix (prefix) which is a prefix (suffix) of any of the patterns. First we consider all blocks at once and for each of them compute its longest prefix which occurs in some p_i.

Lemma 7. *Given an LZW-compressed text we can compute for all blocks the corresponding snippet (if any) and the longest prefix which is a suffix of some pattern in total linear time.*

Proof. The idea is the same as in the single pattern case [12]. We intersect the suffix tree and the trie defined by all blocks. More precisely, for every node of the trie we locate the corresponding node (implicit or explicit) of the suffix tree, if any, see Fig. 2. Assuming that the outgoing edges in both structures are sorted, this can be done in linear total time, as given a node of the trie and its corresponding node of the suffix tree we can simultaneously scan their outgoing edges, and hence determine for each child of the node of the trie its corresponding node of the suffix tree, if any. This gives the snippet for every block. To compute the longest prefix which is a suffix of some pattern, we only need to show how to determine for every block if it corresponds to a suffix of some pattern. For this we simply mark all nodes of the suffix tree corresponding to suffixes of the patterns. Then as soon as we have the corresponding node of the suffix tree for every node of the trie, we can determine if a given block corresponds to a suffix of some pattern by looking up its node. □

To compute the longest suffix which is a prefix of some pattern, we would like to use the Aho–Corasick automaton built for all $p_1, p_2, \dots, p_\ell$, which is a standard multiple pattern matching tool [1]. Recall that its state set consists of all unique prefixes $p_i[1..j]$ organized in a trie. Additionally, each state v stores the so-called *failure link* failure(v), which points to the longest proper suffix of the corresponding word which occurs in the trie as well. If the alphabet is of constant size, we can afford to compute and store the full transition function of such automaton. If the alphabet is $\{0, 1, \dots, M-1\}$, it is not clear if we can afford to store the transition $\delta(v, c)$ for every possible v and c, though. Nevertheless, storing the trie and all failure links is enough to navigate in amortized constant time per letter. This is not enough for our purposes, though, as we need a worst case bound. We start with building the trie and computing the failure links. This is trivial to perform in linear time after constructing the reversed suffix tree: each state is an (implicit or explicit) node of the tree with an outgoing edge starting with \$. Its failure link is simply the lowest ancestor corresponding to such node as well. Then depending on the allowed preprocessing we get two time bounds.

Lemma 8. *Given an LZW-compressed text we can compute for all blocks the longest suffix which is a prefix of some pattern in total time $\mathcal{O}(n + M^{1+\epsilon})$ for any $\epsilon > 0$.*

Proof. For each state v we create an $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1)\rangle$ persistent dynamic dictionary storing the transitions $\delta(v, c)$ for all possible letters c. To create the dictionary for v we take the dictionary stored at failure(v) and update it by inserting all edges outgoing from v. There are at most M updates to all dictionaries, each of them taking $\mathcal{O}(M^\epsilon)$ time, and then any query is answered in constant time, resulting in the claimed bound. □

Lemma 9. *Given an LZW-compressed text we can compute for all blocks the longest suffix which is a prefix of some pattern in total time $\mathcal{O}(n \log M + M)$.*

Proof. Take a state v and consider the sequence of its ancestors failure(v), failure^2(v), failure^3(v), ... To retrieve the transition $\delta(v, c)$ we should find the first vertex in this sequence having an outgoing edge labelled by c. For each different character c we build a separate structure $S(c)$ storing all intervals [pre(v), post(v)] for v having an outgoing edge labelled by c, where pre(v) and post(v) are the pre- and post-order numbers of v in a tree defined by the failure links (i.e., failure(v)

is the parent of v there). Then to calculate $\delta(v, c)$ we should locate the smallest interval containing $\mathrm{pre}(v)$ in $S(c)$. Every $S(c)$ can be implemented as a sorted array containing the answer for each essentially different query. All arrays can be constructed in $\mathcal{O}(M)$ total time by first sorting all intervals and then scanning the intervals stored in every single $S(c)$ from left to right while maintaining all currently intersected intervals on a stack. Then to answer a single query we need just one binary search, hence the total time is as claimed. □

Hence we reduced the original problem to multiple pattern matching in a collection of sequences of snippets, with the total size of all collections linear in n. To solve the latter, we try to simulate the Knuth–Morris–Pratt algorithm on each of those sequences. Of course we cannot afford to process the snippets letter-by-letter, and hence must develop efficient procedures operating on whole snippets. A high level description of the algorithm is given in Multiple-pattern-matching, prefixer and detector are low-level procedures which will be developed in the next section.

Algorithm 1 Multiple-pattern-matching$(s_1, s_2, \dots, s_{n'})$.

```
1: $P \leftarrow \emptyset$
2: $c \leftarrow s_1$
3: for $k = 2, 3, \dots, n'$ do
4:     add $(c, s_k)$ to $P$
5:     $c \leftarrow \mathrm{prefixer}(c, s_k)$
6: end for
7: for all $(s, s') \in P$ do
8:     detector$(s, s')$
9: end for
```

Note that instead of constructing the set P we could call detector(c, s_k) directly but then its implementation would have to be online, and that seems difficult to achieve in the $\langle \mathcal{O}(M), \mathcal{O}(\log M) \rangle$ variant.

4. Multiple pattern matching in a sequence of snippets

An $\langle f(M), g(M) \rangle$ *prefixer* is a data structure which preprocesses the collection of patterns in $f(M)$ time so that given any two snippets we can compute the longest suffix of their concatenation which is a prefix of some pattern in $g(M)$ time.

Lemma 10. *$\langle \mathcal{O}(M), \mathcal{O}(\log M) \rangle$ prefixer exists.*

Proof. Let the two snippets be s_1 and s_2. First note that using Lemma 2 we can compute the longest common prefix of $s_2^r s_1^r$ and any given suffix of A^r in constant time. Hence we can apply binary search to find the (lexicographically) largest suffix of A^r which either begins with $s_2^r s_1^r$ or is (lexicographically) smaller in $\mathcal{O}(\log M)$ time. Given this suffix $A^r[i..|A^r|]$ we compute $d = |LCP(|s_2^r s_1^r|, A^r[i..|A^r|])|$ and apply Lemma 3 to retrieve the ancestor v of $A^r[i..|A^r|]$ at depth d in $\mathcal{O}(\log M)$ time. The longest prefix we are looking for corresponds to an ancestor u of v which has an outgoing edge starting with \$. Observe that such u must be explicit as there are no \$ characters on the root-to-v path. This means that we can apply a simple linear time preprocessing to compute such u for each possible explicit v in linear time. Then given a (possibly implicit) v we use the preprocessing to compute the u corresponding to the longest prefix in constant time, giving an $\mathcal{O}(\log M)$ total query time. □

Lemma 11. *$\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ prefixer exists for any $\epsilon > 0$.*

Proof. For each pattern p_i we consider all possibilities to cut it into two parts $p_i = p_i[1..j]p_i[j+1..|p_i|]$. For each cut we locate vertex u corresponding to $p_i[1..j]$ in the reversed suffix tree and vertex v corresponding to $p_i[j+1..|p_i|]$ in the suffix tree. By Lemma 5 it takes constant time and by Lemma 6 we can then compute $\mathrm{pre}(u)$, $\mathrm{pre}(v)$ and $\mathrm{post}(v)$. Then we add a horizontal segment $\{\mathrm{pre}(u)\} \times [\mathrm{pre}(v), \mathrm{post}(v)]$ with weight j to the collection. Now consider a query consisting of two snippets s_1 and s_2. First locate the vertex u corresponding to s_1 in the reversed suffix tree and vertex v corresponding to s_2 in the suffix tree. Then construct a vertical segment $[\mathrm{pre}(u), \mathrm{post}(u)] \times \{\mathrm{pre}(v)\}$ and observe that the query reduces to finding the heaviest horizontal segment in the collection it intersects (if there is none, we retrieve the lowest ancestor of v which has an outgoing edge starting with \$, which can be precomputed in linear time), see Fig. 3. Additionally, the horizontal segments are either disjoint or contained in each other. In the latter case, weight of the longer segment is bigger than weight of the shorter. To this end we prove that there exists an $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ structure for computing the heaviest horizontal segment intersected by a given vertical segment in such collection on an $M^2 \times M^2$ grid.

We sweep the grid from left to right maintaining a structure describing the currently active horizontal segments. The structure is based on the idea from Lemma 5 with $k \geqslant \frac{2}{\epsilon}$. Each leaf corresponds to a different y coordinate and stores all active horizontal segments with this coordinate on stack, with the most recently encountered segment on top (because weights of intersecting segments are monotone with their lengths, it is also the heaviest segment). Each inner vertex stores a table heaviest$[0..M^{\frac{2}{k}}]$ with the i-th element containing the maximum weight in the subtree corresponding to the i-th

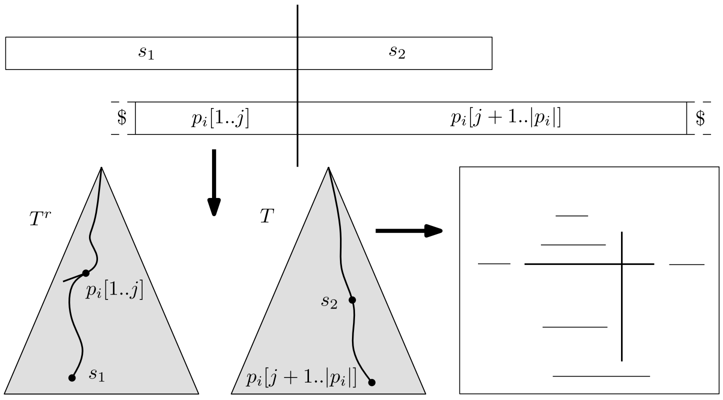

Fig. 3. Reducing prefixer queries to segments intersection.

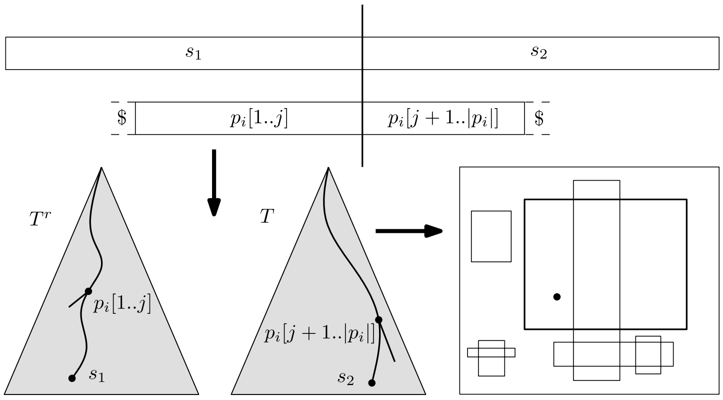

Fig. 4. Reducing detector queries to rectangles retrieval in a valid collection.

leaf, if any. Additionally, a range maximum query structure RMQ(heaviest) is stored so that given any two indices i, j we can compute the maximum heaviest$[k]$ over all $k \in \{i, i+1, \dots, j\}$ in constant time. Adding or removing an active segment requires locating the corresponding stack and either pushing a new element or removing the topmost element. Then we must update the tables at all ancestors of the corresponding leaf, which by Lemma 1 takes $k\mathcal{O}(M^{\frac{2}{k}}) = \mathcal{O}(M^{1+\epsilon})$ time. Given a query, we first locate the appropriate version of the structure. Then we traverse the trie and find the heaviest intersected segment by asking at most $2k$ range maximum queries. □

An $\langle f(M), g(M)\rangle$ *detector* is a data structure which preprocesses the collection of patterns in time $f(M)$ so that given any two snippets we can detect an occurrence of a pattern in their concatenation in $g(M)$ time. Both implementations that we are going to develop are based on the same idea of reducing the problem to a purely geometric question on an integer grid, similar to the one from Lemma 11. For each pattern p_i we consider all possibilities to cut it into two parts $p_i = p_i[1..j]p_i[j+1..|p_i|]$. For each cut we locate in constant time vertex u corresponding to $p_i[1..j]$ in the reversed suffix tree and vertex v corresponding to $p_i[j+1..|p_i|]$ in the suffix tree, and add a rectangle $[\text{pre}(u), \text{post}(u)] \times [\text{pre}(v), \text{post}(v)]$ to the collection. Then given two snippets s_1 and s_2 detecting an occurrence in their concatenation reduces in constant time to retrieving any rectangle containing point $(\text{pre}(u), \text{pre}(v))$ where u is the vertex corresponding to s_1 in the reversed suffix tree and v is the vertex corresponding to s_2 in the suffix tree, see Fig. 4. Note that the x and y projections of any two rectangles in the collection are either disjoint or contained in each other. Assuming no pattern occurs in another, no two rectangles are contained in each other (if some p_i occurs in some p_j, which can be efficiently detected in the preprocessing stage, we can forget about p_j). We call a collection with such two properties *valid*.

Lemma 12. *$\langle \mathcal{O}(M), \mathcal{O}(\log M)\rangle$ offline detector exists.*

Proof. Recall that in the offline version we are given all queries in advance. We sweep the grid from left to right maintaining a structure describing currently intersected rectangles. At a high level the structure is just a full binary tree on M leaves corresponding to different y coordinates (and each inner vertex corresponding to a continuous interval of y coordinates). If we aim to achieve logarithmic time of both update and query, the implementation is rather straightforward. We want to achieve constant time update, though. Say that we encounter a new rectangle and hence need to insert an interval $[y_1, y_2]$ with $y_1 < y_2$ into the structure. We compute the lowest common ancestor v of the leaves corresponding to y_1 and y_2 in the tree (as the tree is full there exists a simple arithmetic formula for that) and call v *responsible* for $[y_1, y_2]$. v corresponds to an interval $[\alpha 2^\ell, (\alpha+2)2^\ell)$ such that $y_1 \in [\alpha 2^\ell, (\alpha+1)2^\ell)$ and $y_2 \in [(\alpha+1)2^\ell, (\alpha+2)2^\ell)$. For each inner vertex we store its *interval stack*. To insert $[y_1, y_2]$ we simply push it on the interval stack of the responsible vertex. Note that because the collection is valid, all intervals $I_1, I_2, \ldots, I_k$ stored on the same interval stack at a given moment are nested, i.e., $I_1 \subseteq I_2 \subseteq \cdots \subseteq I_k$. To remove an interval we locate the responsible vertex and pop the topmost element from its interval stack. The only nontrivial part is detecting an interval containing a given point x. First traverse the path starting at the corresponding leaf. This gives us a sequence of $\log M$ interval stacks. Observe that for a fixed interval stack it is enough to check if its bottom element (if any) contains x, hence $\mathcal{O}(\log M)$ query time follows. □

Lemma 13. *$\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1)\rangle$ detector exists for any $\epsilon > 0$.*

Proof. We sweep the grid from left to right maintaining a structure describing currently intersected rectangles. The structure should allow adding or removing intervals from $\{0, 1, \ldots, M-1\}$ and retrieving any interval containing a specified point. To implement the structure we again use the idea from Lemma 4. We represent all numbers in base $B = M^{\frac{1}{k}}$ and store them in a trie of depth k, where each vertex keeps two tables child$[0..B-1]$ and full$[0..B-1]$. Each such vertex corresponds in a natural way to an interval containing all of its leaves. To add a new interval, we first locate its endpoints in the trie. Then we decompose it into at most $2k$ smaller parts, where each part corresponds to a contiguous range of children of the same node of the trie. We increase all entries of the array full corresponding to these children by one. To remove an interval, we decrease the entries by one. This implementation is easily made persistent by making a new copy of each modified array and the whole traversed path so that every update takes $\mathcal{O}(kB) = \mathcal{O}(M^\epsilon)$ time. To answer a query, we traverse the corresponding path in the trie in $\mathcal{O}(1)$ time. If for any node on this path we descend to the i-th child and full$[i]$ is positive, we have an occurrence. □

By plugging either Lemma 10 and Lemma 12 or Lemma 11 and Lemma 13 into MULTIPLE-PATTERN-MATCHING we get the main theorem.

Theorem 1. *Multiple pattern matching in a sequence of n snippets can be performed in $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ time, where M is the combined size of all patterns.*

By adding Lemma 7 and either Lemma 9 or Lemma 8 we get the claimed total running time of the whole solution.

Theorem 2. *Multiple pattern matching in LZW-compressed texts can be performed in $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ time, where M is the combined size of all patterns and n is the size of the compressed representation. The space used by the solutions is $\mathcal{O}(n + M)$ and $\mathcal{O}(n + M^{1+\epsilon})$, respectively.*

5. Conclusions

We presented two efficient solutions for detecting if some pattern occurs in an LZW-compressed text. The first of our algorithms is linear in M, and the second is linear in n, hence they represent the two extremes of a possible trade-off. An interesting question is how this trade-off looks like.

A natural generalization is detecting all occurrences. If no pattern occurs in another, straightforward generalizations of Lemma 12 and Lemma 13 allow generating all *occ* occurrences in $\mathcal{O}(n \log M + M + occ)$ and $\mathcal{O}(n + M^{1+\epsilon} + occ)$ time, respectively. Without such assumption, there is one problem, though: we cannot assume that the collection of rectangles in Lemma 12 is valid, and $\mathcal{O}((n + M)\log M + occ)$ running time seems to be required. We leave decreasing this complexity as future work.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (1975) 333–340.
[2] A. Amir, G. Benson, M. Farach, Let sleeping files lie: pattern matching in Z-compressed files, in: SODA '94: Proceedings of the Fifth Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994, pp. 705–714.
[3] D. Arroyuelo, G. Navarro, K. Sadakane, Stronger Lempel–Ziv based compressed text indexing, Algorithmica 62 (1–2) (2012) 54–101.
[4] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of the 4th Latin American Symposium on Theoretical Informatics, LATIN '00, Springer-Verlag, London, UK, 2000, pp. 88–94.
[5] P. Bille, I. Gørtz, Substring range reporting, in: R. Giancarlo, G. Manzini (Eds.), Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 6661, Springer, Berlin/Heidelberg, 2011, pp. 299–308.
[6] J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan, Making data structures persistent, J. Comput. Syst. Sci. 38 (1) (1989) 86–124.
[7] M. Farach, M. Thorup, String matching in Lempel–Ziv compressed strings, in: STOC '95: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, ACM, New York, NY, USA, 1995, pp. 703–712.
[8] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. Syst. Sci. 30 (2) (1985) 209–221.

[9] Z. Galil, String matching in real time, J. ACM 28 (1) (1981) 134–149.
[10] Z. Galil, J. Seiferas, Time–space-optimal string matching (preliminary report), in: STOC '81: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, ACM, New York, NY, USA, 1981, pp. 106–113.
[11] P. Gawrychowski, Tying up the loose ends in fully LZW-compressed pattern matching, in: C. Dürr, T. Wilke (Eds.), STACS, in: LIPIcs, vol. 14, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012, pp. 624–635.
[12] P. Gawrychowski, Optimal pattern matching in LZW compressed strings, ACM Trans. Algorithms 9 (3) (2013) 25:1–25:17.
[13] M. Gu, M. Farach, R. Beigel, An efficient algorithm for dynamic text indexing, in: Proceedings of the Fifth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA '94, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994, pp. 697–704.
[14] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.
[15] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, S. Arikawa, Multiple pattern matching in LZW compressed text, in: Data Compression Conference, DCC'98, Proceedings, IEEE, 1998, pp. 103–112.
[16] D.E. Knuth, J.H. Morris Jr., V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[17] S.R. Kosaraju, Pattern matching in compressed texts, in: Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, London, UK, 1995, pp. 349–362.
[18] T.A. Welch, A technique for high-performance data compression, Computer 17 (6) (1984) 8–19.