String Matching in Lempel-Ziv Compressed Strings*

Martin Farach†
Rutgers University

Mikkel Thorup‡
University of Copenhagen

Abstract

String matching and Compression are two widely studied areas of computer science. The theory of string matching has a long association with compression algorithms. Data structures from string matching can be used to derive fast implementations of many important compression schemes, most notably the Lempel-Ziv (LZ77) algorithm. Intuitively, once a string has been compressed – and therefore its repetitive nature has been elucidated – one might be tempted to exploit this knowledge to speed up string matching. The *Compressed Matching Problem* is that of performing string matching in a compressed text, without uncompressing it. More formally, let $\mathcal{T}$ be a text, let $\mathcal{Z}$ be the compressed string representing $\mathcal{T}$, and let $\mathcal{P}$ be a pattern. The Compressed Matching Problem is that of deciding if $\mathcal{P}$ occurs in $\mathcal{T}$, given only $\mathcal{P}$ and $\mathcal{Z}$. Compressed matching algorithms have been given for several compression schemes such as LZW.

In this paper, we give the first non-trivial compressed matching algorithm for the classic adaptive compression scheme, the LZ77 algorithm. In practice, the LZ77 algorithm is known to compress more than other dictionary compression schemes, such as LZ78 and LZW, though for strings with constant per bit entropy, all these schemes compress optimally in the limit. However, for strings with $o(1)$ per bit entropy, while it was recently shown that the LZ77 gives compression to within a constant factor of optimal, schemes such as LZ78 and LZW may deviate from optimality by an exponential factor. Asymptotically, compressed matching is only relevant if $|\mathcal{Z}| = o(|\mathcal{T}|)$, i.e. if the compression ratio $|\mathcal{T}|/|\mathcal{Z}|$ is more than a constant. These results show that LZ77 is the appropriate compression method in such settings.

We present an LZ77 compressed matching algorithm which runs in time $O(\mathrm{N} \log^2 \mathrm{U}/\mathrm{N} + \mathrm{P})$ where $\mathrm{N} = |\mathcal{Z}|$, $\mathrm{U} = |\mathcal{T}|$, and $\mathrm{P} = |\mathcal{P}|$. Compare with the naïve "decompresion" algorithm, which takes time $\Theta(\mathrm{U} + \mathrm{P})$ to decide if $\mathcal{P}$ occurs in $\mathcal{T}$. Writing $\mathrm{U} + \mathrm{P}$ as $\mathrm{N} \cdot \mathrm{U}/\mathrm{N} + \mathrm{P}$, we see that we have improved the complexity, replacing the compression factor U/N by a factor $\log^2 \mathrm{U}/\mathrm{N}$. Our algorithm is *competitive* in the sense that $O(\mathrm{N} \log^2 \mathrm{U}/\mathrm{N} + \mathrm{P}) = O(\mathrm{U} + \mathrm{P})$, and *opportunistic* in the sense that $O(\mathrm{N} \log^2 \mathrm{U}/\mathrm{N} + \mathrm{P}) = o(\mathrm{U} + \mathrm{P})$ if $\mathrm{N} = o(\mathrm{U})$ and $\mathrm{P} = o(\mathrm{U})$.

---

*A preliminary version of the paper appeared in STOC '95.

†Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. (*farach@cs.rutgers.edu;http://www.cs.rutgers.edu/~farach*). Part of this work was done while the author was visiting the University of Copenhagen; Supported in part by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648.

‡Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Kbh. Ø, Denmark. (*mthorup@diku.dk, http://www.diku.dk/~mthorup*). This work was done while visiting DIMACS.

1 Introduction

String matching and Compression are two widely studied areas of computer science. The String Matching Problem is that of detecting if pattern $\mathcal{P}$ occurs in text $\mathcal{T}$. The Compression Problem is that of developing schemes which can be used to give succinct representations for large files. Each of these fields has its own very rich theory. The theory of string matching has a long association with compression algorithms. Data structures from string matching can be used to derive fast implementations of many important compression schemes [6, 13, 14], most notably the Lempel-Ziv algorithm [11]. The reason for this connection is clear; string algorithms and data structures most commonly detect repetitions of various sorts within strings. The elimination of repetition is the key to compression.

Intuitively, once a string has been compressed – and therefore its repetitive nature has been elucidated – one might be tempted to exploit this knowledge to speed up string matching. This idea is inherent in the work of Amir and Benson [2], where they introduced the *Compressed Matching Problem*. They were motivated by the practical consideration that, increasingly, files are stored in a compressed format, and that current string matching technology requires the files be uncompressed before string matching takes place.

The Compressed Matching Problem

Instance: A compressed string $\mathcal{Z}$ and a pattern $\mathcal{P}$.

Question: Does $\mathcal{P}$ occur in the text $\mathcal{T}$ represented by $\mathcal{Z}$?

Assuming $|\mathcal{Z}| \ll |\mathcal{T}|$, we hope for substantial savings over the naïve $\Omega(|\mathcal{T}| + |\mathcal{P}|)$ algorithm which first decompresses, and then performs standard string matching. An *optimal* algorithm takes time $\Theta(|\mathcal{Z}| + |\mathcal{P}|)$. Of course, the nature of the compression algorithm determines the difficulty of performing efficient compressed matching. For the remainder of the paper, let $\mathtt{U}$ be the size of the uncompressed text, $\mathtt{N}$ the size of its compressed representation, and $\mathtt{P}$ the size of the pattern. Note that compressed matching is only relevant if the *compression ratio* is more than a constant, i.e. if $\mathtt{U}/\mathtt{N} = \omega(1)$; otherwise the naïve decompressing algorithm is itself optimal for compression schemes which are decodable in linear time. We will say that a compressed matching algorithm is *competitive* if its running time is $O(\mathtt{U}+\mathtt{P})$, that is, if it is never worse than the naïve "decompression" algorithm. We will say that an algorithm is *opportunistic* if its running time is $o(\mathtt{U} + \mathtt{P})$ whenever $\mathtt{N} = o(\mathtt{U})$ and $\mathtt{P} = o(\mathtt{U})$. Note that any optimal algorithm is both competitive and opportunistic. Generally a competitive and opportunistic algorithm has a time complexity of the form $O(\mathtt{N}o(\mathtt{U}/\mathtt{N}) + \mathtt{P})$. Thus, in the absence of optimal algorithms, it is natural to look for a compressed matching algorithm with time complexity $O(\mathtt{N}(\text{polylog } \mathtt{U}/\mathtt{N}) + \mathtt{P})$, and we will refer to such algorithms as *pseudo-optimal*.

In this paper, we give a pseudo-optimal compressed matching algorithm for the classic adaptive compression scheme, the so-called LZ77[1] algorithm [11]. To the best of our knowledge this is the first non-trivial LZ77 compressed matching algorithm, and the search for such an algorithm was advocated in [3], in which strings compressed by LZW are considered. In practice, the LZ77 algorithm is known to compress more than other dictionary compression schemes, such as LZ78 and LZW, though for strings with constant per bit entropy, all these schemes compress optimally

[1] The LZ77 algorithm is sometimes called LZ1 and the LZ78 algorithm is sometimes called LZ2

in the limit. However, for strings with $o(1)$ per bit entropy, while Kosaraju [9] recently showed that the LZ77 gives compression to within a constant factor of optimal, schemes such as LZ78 and LZW can deviate from optimality by an exponential factor.

It should be noted that for most of todays applications of general compression, the compression ratio is not large enough to justify an assymptotic analysis. However, our study of compressed matching should be seen as an approach to developing general techniques for dealing with low-entropy strings. Consider, for example, a long text $\mathcal{T}$ which is essentially just a repetition of a some rather long patterns, but where various error-patterns may occur sporadically. The text $\mathcal{T}$ will then have low entropy, and hence, space-wise, we will benefit from compressing it into a compressed string $\mathcal{Z}$, where $|\mathcal{Z}| \ll |\mathcal{T}|$ if the repeated patterns are long. If we later want to look for some error pattern $\mathcal{P}$, compressed matching finds it directly from $\mathcal{Z}$, capitalizing on the small size of $\mathcal{Z}$ both in time and space. Alternatively, if one knew the patterns that $\mathcal{T}$ was made up from, one could easily make a specialized compression facilitating fast matching. Indeed, without general compressed matching routines, if we wanted to facilitate matching, we would have been forced to make a specialized compression, and this may exactly be the type of reason why general compression is not more frequently used on low-entropy texts. The point of general compressed matching is to avoid specialized programming; the more operations we can perform directly on generally compressed text, the less specialized programming is needed in dealing with low-entropy objects.

1.1 Previous Results

The theory of compressed matching was initiated with the study of two dimensions run-length compression, the compression algorithm used for fax transitions. In [2, 4], an optimal $O(\mathtt{N} + \mathtt{P}^2)$ algorithm was derived for finding a $\mathtt{P} \times \mathtt{P}$ two dimensional pattern within a file of length $\mathtt{N}$.

In one dimension, the formerly most commonly used compression algorithm was the LZW algorithm [16] as implemented by the UNIX `compress` program [15]. It seems to have been replaced in popularity by a variant on LZ77, as implemented in GNU's `gzip`. In [3], it was shown that a length $\mathtt{P}$ pattern can be found in a length $\mathtt{N}$ text in time $O(\mathtt{N} + \mathtt{P}^2)$, or in $O(\mathtt{N} \log \mathtt{P} + \mathtt{P})$ in LZW compressed texts. In [10], Kosaraju improved this complexity to $O(\mathtt{N} + \mathtt{P}^{1+\epsilon})$. The techniques in [3] show that it is trivial to perform linear time compressed matching on LZW files if the pattern is of constant length. In contrast, it seems quite difficult to perform any non-trivial compressed matching on LZ77 files, *even for patterns of length 2!*

1.2 Our Results

We will present an algorithm for the LZ77 compressed matching problem showing:

Theorem 1.1 *Given an LZ77 compressed string $\mathcal{Z}$ of size $\mathtt{N}$ representing a text $\mathcal{T}$ of size $\mathtt{U}$, and given a pattern $\mathcal{P}$ of size $\mathtt{P}$ there is a randomized (Las Vegas) algorithm to decide if $\mathcal{P}$ occurs in $\mathcal{T}$ which runs in time $O(\mathtt{N} \log^2 \mathtt{U}/\mathtt{N} + \mathtt{P})$ w.h.p.*

Technical Issues and Contribution: The centerpiece of our algorithm is an entropy based potential function argument. In high entropy strings, the "information" can be thought of as being more or less evenly distributed through the string. For lower entropy strings, the information

tends to clump in certain regions. We take advantage of this clumpiness to efficiently uncompress only the relevant parts of the underlying text string, as described in Sections 3 and 4. The ideas presented therein can be used directly to arrive at a deterministic algorithm with running time $O(\mathrm{N}\log(\mathrm{U}/\mathrm{N})(\log(\mathrm{U}/\mathrm{N})+\log \mathrm{P}))$. Alternatively, combining with ideas from [3, 10], we get an $O(\mathrm{N}(\log^2 \mathrm{U}/\mathrm{N})+\mathrm{P}^{1+\epsilon})$ deterministic algorithm.

Neither of these deterministic bounds is fully competitive. They both perform worse than the naïve decompressing $\Theta(\mathrm{U}+\mathrm{P})$ algorithm if the pattern is large and the compression ratio is low. However, we give a novel application of Karp and Rabin's fingerprint method [8], which gives us a randomized $O(\mathrm{N}\log^2(\mathrm{U}/\mathrm{N})+\mathrm{P})$ time algorithm for LZ77 compressed matching. Thus we get an algorithm which is both opportunistic and competitive. In fact this last technique also applies to the best known compressed matching algorithm for LZW [3] to yield a $O(\mathrm{N}\log \mathrm{U}/\mathrm{N}+\mathrm{P})$ time algorithm. Thus a side-effect of this paper is the first competitive and opportunistic LZW compressed matching algorithm.

1.3 The LZ77 Algorithm

The LZ77 algorithm gives a very natural way of representing a string and is defined as follows. Given some alphabet Σ, an *LZ77 compressed string* is a string of the form: $\mathcal{Z} = (P_0, L_0, C_0)\cdots(P_i, L_i, C_i)\cdots(P_{\mathrm{N}-1}, L_{\mathrm{N}-1}, C_{\mathrm{N}-1}) \in (\mathbb{N}_0, \mathbb{N}_0, \Sigma)^{\mathrm{N}}$. Let $U_0 = 0$ and $U_i = U_{i-1} + L_{i-1} + 1$ for $i > 0$. For $\mathcal{Z}$ to be well-formed, it is required that, for all $L_i > 0, P_i < U_i$. Now, $\mathcal{Z}$ *represents* $\mathcal{T}$ if for $i = 0, \ldots, \mathrm{N}-1$, $\mathcal{T}[U_i, U_i + L_i - 1] = \mathcal{T}[P_i, P_i + L_i - 1]$, $\mathcal{T}[U_i + L_i] = C_i$. Here $\mathcal{T}[I]$ denotes the $(I+1)$th character of $\mathcal{T}$, and $\mathcal{T}[I, J]$ denotes the segment $\mathcal{T}[I]\cdots\mathcal{T}[J]$. Clearly, $\mathcal{T}$ is uniquely defined in terms of $\mathcal{Z}$. For example, $\mathcal{Z} = (0, 0, \mathtt{a})(0, n-2, \mathtt{a})$ represents $\mathcal{T} = \mathtt{a}^n$. For comparison, LZW can never compress a text to less than the square root of its original length.

Simple transformations to LZ77: First, note that each (uncompressed) text fragment $\mathcal{T}[U_i, U_i + L_i]$ consists of a region in which the string is unknown (without uncompressing) followed by a single known character. In order to facilitate a more uniform treatment, we simplify the scheme by the following trivial transformation. First, map Σ into $[1, |\Sigma|]$. Then take each triple (P_i, L_i, C_i) and replace it by the pairs $(P_i, L_i)(-C_i, 1)$. In the special case where $L_i = 0$, replace (P_i, L_i, C_i) with $(-C_i, 1)$ so all specified intervals have positive length. Now, given a compressed text $\mathcal{Z} = (P_0, L_0)\cdots(P_{\mathrm{N}-1}, L_{\mathrm{N}-1})$, the uncompressed text $\mathcal{T}$ is defined by

$$\begin{aligned} &\forall i = 0, \ldots, \mathrm{N}-1: \\ &\quad \mathcal{T}[U_i, U_i + L_i - 1] = \mathcal{T}[P_i, P_i + L_i - 1] \end{aligned} \tag{1}$$

where $U_i = \sum_{j<i} L_j$ and where, for any integer C representing a character, $\mathcal{T}[-C] = C$. Thus, we imagine that $\mathcal{T}$ has been prefixed by Σ in the negative positions. For example, $\mathtt{a}^n$ is compressed to "a"$(-1, 1)(0, n-1)$. For the moment, we will make the simplifying assumption that $\mathcal{Z}$ is not *self-referential* meaning that we assume $L_i \le U_i - P_i$ for all i. In practice, this means that if we decompress $\mathcal{Z}$ using Eq. 1, then $\mathcal{T}[U_i, U_i + L_i - 1]$ can be copied directly from the previously decompressed text $\mathcal{T}[0, U_i - 1]$. The same simplification is made by `gnuzip` and most other implementations of LZ77. In the worst case, it gives a blow-up in the size of the compressed text which is logarithmic in the compression factor. For example, our representation of $\mathtt{a}^n$ is changed to

"a"$(-1,1)(0,1)(0,2)(0,4)\cdots(0,2^{\lfloor \log n \rfloor -1})(0,n-2^{\lfloor \log n \rfloor}+1)$. However, in Section 7, we will show how self-reference can be dealt with directly, avoiding any increase in asymptotic complexity.

In Section 2, we give an outline of the algorithm and conclude with an outline of the rest of the paper.

2 General Algorithmic Outline

We can make the following simple observation which limits where we need to look for the pattern occurrence.

Observation 2.1 ([3]) *If u is the least position such that $\mathcal{P} = \mathcal{T}[u, u+|\mathcal{P}|-1]$, then $U_i \in [u, u+|\mathcal{P}|]$ for some $i > 0$.*

Algorithmically we can formulate Observation 2.1 as follows. Suppose for $i = 0, \ldots, \mathrm{N}$ we have:

1. The longest pattern substrings $\mathcal{P}_i^+$ which is a prefix of $\mathcal{T}[U_i, \mathrm{U}]$
2. The longest pattern substring $\mathcal{P}_i^-$ which is a suffix of $\mathcal{T}[0, U_i + L_i - 1]$.

Then the compressed pattern matching problem reduces to deciding whether $\mathcal{P}$ is a substring of $\mathcal{P}_i^- \mathcal{P}_{i+1}^+$ for some $i > 0$.

The brute force method first decompress the whole text $\mathcal{T}$ in time $\Theta(\mathrm{U})$ using Eq. 1, and then performs standard string matching in time $\Theta(\mathrm{P} + \mathrm{U})$. The conclusion we may draw from Observation 2.1 is that instead of uncompressing the whole text, we only need to uncompress the $2\mathrm{N}$ $\mathcal{P}_i^{+/-}$s. Note that we cannot afford to identify all the $\Omega(\mathrm{PN})$ individual characters in the $\mathcal{P}_i^{+/-}$s. However, each $\mathcal{P}_i^{+/-}$ can be represented by a beginning and ending pointer into the pattern, i.e. in constant space. The pattern will be preprocessed so that the $\mathcal{P}_i^{+/-}$s can be manipulated in terms of fast concatenations and truncations of pattern substrings in their succinct represention.

In order to focus on the decompression around the $\mathcal{P}_i^{+/-}$s, we will first use Eq. 1 in reverse finding out which positions the $\mathcal{P}_i^{+/-}$s correspond to earlier in the text and alphabet. Afterwards we know exactly which parts of the text it is relevant to decompress. Formally, we introduce what we will call *informers*. An informer x has a variable position $\mathrm{u}(x) \in \{0, \ldots, \mathrm{U}-1\}$ in the uncompressed text $\mathcal{T}$. When the compressed pattern matching starts, for $i = 0, \ldots, \mathrm{N}-1$, we have a *right informer* r_i placed at position U_i, and our goal is to load it with $\mathcal{P}_i^+$. Also, we have a *left informer* l_i placed at position $U_i + L_i - 1$ which we want to load with $\mathcal{P}_i^-$. The compressed pattern matching now proceeds in two phases.

The Winding Phase For $s := \mathrm{N}-1$ downto 0, we take all informers between U_s and $U_s + L_s - 1$ and shift them $U_s - P_s$ to the left. Since $\mathcal{Z}$ is not self-referential, $U_s - P_s \geq L_s$. Hence, inductively, we may conclude that step s does not leave any informers to the right of U_s. As a consequence, we may reformulate step s as, for all x such that $\mathrm{u}(x) \geq U_s$, set $\mathrm{u}(x) := \mathrm{u}(x) - (U_s - P_s)$. When the winding ends, all informers will have been shifted into the alphabet at negative positions. Immediately this allows us to identify the first character of $\mathcal{P}_i^+$ and the last character of $\mathcal{P}_i^-$, so in particular, the winding solves the compressed pattern matching problem for patterns of length 2. In each of the above N iterations, we risk shifting $\Omega(\mathrm{N})$ informers to the left. Nevertheless, as the cornerstone of this paper, in Section 3 we will implement the winding in $O(\mathrm{N} \log^2(\mathrm{U}/\mathrm{N}))$ total time.

Figure 2.1: In (a), the interval to the right of the cut has informers marked with a circle. All other informers are marked with a plus. At (b), we merge the informers. After (c), we have have bottomed out the computation and have partial information loaded in the informers. We un-merge the circle informers in (d) and extend the pattern substrings of all those that overlap the cut boundary. Above, it looks as if all the informers are both left and right informers, but this is to cover the case of overlap.

The Unwinding Phase In this phase we move the informers back, reversing shifts of the winding precisely. However, with each right informer, we will associate a variable pattern substring $\mathtt{t}(x)$ which, relative to the current position $\mathtt{u}(x)$, satisfies the condition that it is the longest pattern substring which is a prefix of the "known" part of $\mathcal{T}[\mathtt{u}(x), \mathtt{U}-1]$; after unwinding step s, we consider $\mathcal{T}[0, U_s+L_s-1]$ known. Similarly, if x is a left informer, $\mathtt{t}(x)$ is the longest pattern substring which is a suffix of $\mathcal{T}[0, \mathtt{u}(x)]$. Satisfying these invariants implies that when the informers come back to their original positions $\mathtt{t}(r_i) = \mathcal{P}_i^+$ and $\mathtt{t}(l_i) = \mathcal{P}_i^-$, for $i = 0, \ldots, \mathtt{N}-1$, as desired. If $\mathtt{u}(x) < 0$, the informer is still in the alphabet, and it is then just loaded with the single character it is positioned at.

More constructively, as soon as the winding has ended, for each informer x, set $\mathtt{t}(x)$ to the string consisting of the character numbered $-\mathtt{u}(x)$. Recall that in step s of the winding, we took all informers x at positions $\mathtt{u}(x) > U_s$ and shifted them $U_s - P_s$ to the left. Assume, for each s, that we have recorded the set X_s of informers that were shifted by step s. Then, for $s := 0$ to $\mathtt{N}-1$, we unwind the informers as follows. First, for all $x \in X_s$, set $\mathtt{u}(x) := \mathtt{u}(x) + U_s - P_s$, returning them to their position before winding step s. Concerning the loaded pattern substrings, for all right informers $x \in X_s$, truncate $\mathtt{t}(x)$ so that it does not extend beyond $U_s + L_s - 1$, i.e. if $\mathtt{u}(x) + |\mathtt{t}(x)| > U_s + L_s$, delete the last $\mathtt{u}(x) + |\mathtt{t}(x)| - U_s + L_s$ characters of $\mathtt{t}(x)$. Then $\mathtt{t}(x)$ satisfies that it is the longest pattern substring which is a prefix of $\mathcal{T}[\mathtt{u}(x), U_s + L_s - 1]$. In particular, this holds for the right informer r_s, which has now reached its original position U_s. For all the other right informers $x \notin X_s$, if $\mathtt{t}(x)$ extends to $U_s - 1$, i.e. if $\mathtt{u}(x) + |\mathtt{t}(x)| - 1 = U_s - 1$, make $\mathtt{t}(x)$ the longest pattern substring which is a prefix of $\mathtt{t}(x)\mathtt{t}(r_s)$. Now all right informers x satisfy that $\mathtt{t}(x)$ is the longest pattern substring which is a prefix of $\mathcal{T}[\mathtt{u}(x), U_s + L_s - 1]$.

Concerning the loading of the left informers, first notice that all left informers that are not shifted satisfy the invariant that $\mathtt{t}(x)$ is the longest pattern substring which is a suffix of $\mathcal{T}[0, \mathtt{u}(x)]$. In particular, this holds for l_{i-1} that has received its final loading, i.e. $\mathtt{t}(l_{i-1}) = \mathcal{P}_{i-1}^-$. For the shifted left informers $x \in X_s$, we need to do as follows. If $\mathtt{t}(x)$ extends back to U_s, i.e. if $\mathtt{u}(x) - |\mathtt{t}(x)| + 1 \leq U_s$, then first we delete the first $U_s - (\mathtt{u}(x) - |\mathtt{t}(x)| + 1)$ characters of $\mathtt{t}(x)$, second we make $\mathtt{t}(x)$ the longest pattern substring which is a suffix of $\mathtt{t}(l_{s-1})\mathtt{t}(x)$. Now all left informers x satisfy that $\mathtt{t}(x)$ is the longest pattern substring which is a suffix of $\mathcal{T}[0, \mathtt{u}(x)]$. Thus, we can proceed to the unwinding of winding step $s+1$. Figure 2.1, page 5, illustrates the work of the algorithm.

In the worst case, the above naïve unwinding could give rise to $\Theta(\mathtt{N}^2)$ operations on the loaded pattern substrings. However, many of these loaded pattern substrings can be deduced from the

others, and in Section 4 it will be shown that given the winding from Section 3, we only need to make $O(\mathrm{N}\log(\mathrm{U}/\mathrm{N}))$ operations on pattern substrings for the unwinding. Having applied an $O(\mathrm{P})$ pattern preprocessing technique from [7], each of these operations are done in $O(\log \mathrm{P})$ time, giving a total complexity for the compressed pattern matching problem of $O(\mathrm{P} + \mathrm{N}\log(\mathrm{U}/\mathrm{N})(\log(\mathrm{U}/\mathrm{N}) + \log \mathrm{P}))$. For the case where $\log \mathrm{P} = \omega(\log \mathrm{U}/\mathrm{N})$, in Section 5, we present a Monte Carlo randomized algorithm which solves the compressed pattern matching problem within the announced time bound of $O(\mathrm{P} + \mathrm{N}\log^2(\mathrm{U}/\mathrm{N}))$. In Section 6, we give a checker with which we convert our Monte Carlo algorithm into a Las Vegas one. Finally, in Section 7, we will show how we can deal with self-references.

3 Fast Winding

3.1 A fast algorithm

In this section, we will speed-up the naïve $\Omega(\mathrm{N}^2)$ winding from the last section to run in time $O(\mathrm{N}\log^2(\mathrm{U}/\mathrm{N}))$. Left and right informers will be wound separately. In particular this means that if two informers end up at the same position, then we can remove one of them as a duplicate, restoring it at the appropriate time of the unwinding. First consider the pseudo code for winding described in the last section.

Algorithm A Winds a list $\mathcal{L}$, $|\mathcal{L}| = O(\mathrm{N})$, of informers sorted according to u-values.

A.1. For $s := \mathrm{N} - 1$ downto 0 do

A.1.1. Cut $\mathcal{L}$ at U_s into lists $\mathcal{L}_0$ and $\mathcal{L}_1$, i.e. $\mathcal{L} = \mathcal{L}_0\mathcal{L}_1$ and $x \in \mathcal{L}_0$ iff $\mathrm{u}(x) < U_s$.

A.1.2. Subtract $U_s - P_s$ from u for all informers in $\mathcal{L}_1$, corresponding to setting $\mathrm{u}(x) := \mathrm{u}(x) - U_s + P_s$ for all $x \in \mathcal{L}_1$.

A.1.3. Merge $\mathcal{L}_0$ with $\mathcal{L}_1$ into $\mathcal{L}$, removing informers that are duplicate with respect to u-values.

Using any standard balanced tree data structure [1] to implement the lists, all steps but the merge step (A.1.3) can be implemented in time $O(\log \mathrm{N})$, hence in time $O(\mathrm{N}\log \mathrm{N})$ over all iterations. However, for the merge step it is possible to construct inputs for which we have both $\mathcal{L}_0$ and $\mathcal{L}_1$ of length $\Omega(\mathrm{N})$ in $\Omega(\mathrm{N})$ iterations. Thus, it seems that the merging alone has running time $\Omega(\mathrm{N}^2)$. It turns out that what we need is an old folklore version of merging, below referred to as *segment merge* (Algorithm B). This version is normally abandoned because it has a bad worst-case performance for the isolated task of merging two lists (if the lists have lengths n_1 and n_2 it takes time $O((n_1 + n_2)\log(n_1 + n_2))$. Here we will resurrect segment merge, showing that in our context it has a very good amortized complexity. More precisely, with a potential function argument, we will show that segment merge takes $O(\mathrm{N}\log \mathrm{N}\log \mathrm{U})$ time in total, and later we will reduce this complexity to $O(\mathrm{N}\log^2(\mathrm{U}/\mathrm{N}))$. In the following algorithm, let $\mathtt{head}(L)$ be the first item on list L, and $\mathtt{tail}(L)$ be the the list with the first item removed.

Algorithm B `Segment-Merge`$(\mathcal{L}_0, \mathcal{L}_1)$ implements step A.1.3.

B.1. If one of the lists is empty, return the other.

B.2. For $i := 0, 1$, set $x_i := \mathtt{head}(\mathcal{L}_i)$.

B.3. If $\mathtt{u}(x_0) = \mathtt{u}(x_1)$, return
x_0 `Segment-Merge`$(\mathtt{tail}(\mathcal{L}_0), \mathtt{tail}(\mathcal{L}_1))$.

B.4. For $i := 0, 1, \bar{\imath} = 1 - i$, if $\mathtt{u}(x_i) < \mathtt{u}(x_{\bar{\imath}})$:

B.4.1. Cut $\mathcal{L}_i$ at $\mathtt{u}(x_{\bar{\imath}})$ into $\mathcal{L}'_i$ and $\mathcal{L}''_i$.

B.4.2. Return $\mathcal{L}'_i$ `Segment-Merge`$(\mathcal{L}''_i, \mathcal{L}_{\bar{\imath}})$.

In the following, by a *proper* call to `Segment-Merge` we refer to one where both lists are nonempty. Note that the number of improper calls is bounded by the number of calls to `Segment-Merge` from step A.1.3, hence by N. Hence we will only be concentrating on the proper calls to `Segment-Merge`. A single call `Segment-Merge`$(\mathcal{L}_0, \mathcal{L}_1)$ can give rise to $|\mathcal{L}_0| + |\mathcal{L}_1| - 1 = \Omega(\mathrm{N})$ recursive calls. Nevertheless we will show that from all the N direct calls to `Segment-Merge` from Algorithm A, we will only get $O(\mathrm{N} \log \mathrm{U})$ calls in total.

3.2 The potential function argument

Our argument is based on a simple potential function. In the above algorithm each informer is either in a list or deleted. For a "live" informer x, i.e. x has not been removed as a duplicate, in a list, we define $\mathtt{d}^-(x) = \mathtt{u}(x) - \mathtt{u}(x^-)$ if x^- is the predecessor of x, and $\mathtt{d}^-(x) = 2\mathrm{U}$ if x is the first element in the list. Similarly, $\mathtt{d}^+(x) = \mathtt{u}(x^+) - \mathtt{u}(x)$ if x^+ is the successor of x, and $\mathtt{d}^+(x) = 2\mathrm{U}$ if x is the last element in the list. Note that for any informer x in a list, $1 \le \mathtt{d}^{-/+}(x) \le 2\mathrm{U}$. Now the *local potential* $\pi(x)$ is defined as $1 + \log \mathtt{d}^-(x) + \log \mathtt{d}^+(x)$. If x has been deleted $\pi(x) = 0$. The *total potential* is $\Pi = \sum_x \pi(x) > 0$.

The idea behind Π is that if, during a merge, we insert a segment of a list between two elements x_0 and x_1 in another list, then we halve either $\mathtt{d}^+(x_0)$ or $\mathtt{d}^-(x_1)$. Thereby we decrease Π by at least 1. Thus, merges will be accounted for as decreases in Π. Before going into details about the decreases, first we will discuss the possible increases to Π.

Initially, the $\mathcal{L}$ contains all N informers, and $\Pi \le \mathrm{N}(1 + 2\log 2\mathrm{U}) = O(\mathrm{N} \log \mathrm{U})$. During the algorithm, Π will increase during the cut step A.1.1. More precisely, the cut will affect the last informer x_0 in $\mathcal{L}_0$, setting $\mathtt{d}^+(x_0) = 2\mathrm{U}$, and the first informer x_1 in $\mathcal{L}_1$, setting $\mathtt{d}^-(x_1) = 2\mathrm{U}$. Hence Π will increase by at most $2\log 2\mathrm{U}$, so in total Π is increased by $O(\mathrm{N} \log \mathrm{U})$ in step A.1.1. The subtraction step A.1.2 does not change Π. Hence, we prove the bound if we can show that the merge step A.1.3 decreases Π with a number proportional to the number of recursive calls to `Segment-Merge`, as defined in Algorithm B.

In order to study the total decrease in Π during a merge, we will see the process as divided into two phases. In the first phase we merge the elements, allowing multiplicity. Thus, the only effect of the first phase is to reduce $\mathtt{d}^-$ and $\mathtt{d}^+$ for some of the informers. In the second phase we delete the duplicate elements. Notice that neither phase increases the local potential of any informer.

There are now two cases to consider. If we go into the case in step B.3 with $\mathtt{u}(x_0) = \mathtt{u}(x_1)$, then this corresponds to the deletion of x_1 in the second phase. Since any live informer has $\pi \ge 1$, this step corresponds to a decrease in Π of at least 1.

Now consider the case in step B.4 where $\mathtt{u}(x_i) < \mathtt{u}(x_{\bar{\imath}})$. We will relate this to a decrease in Π during the first phase. Let $\mathcal{L}^o_{\bar{\imath}}$ be the list of $x_{\bar{\imath}}$ from the original (non-recursive) call to `Segment-Merge` from step A.1.3. Suppose that $x_{\bar{\imath}}$ is not the first element in $\mathcal{L}^o_{\bar{\imath}}$ and let $x^-_{\bar{\imath}}$ be its predecessor. Then the first phase of the merge is to insert the segment $\mathcal{L}'_i$ between $x^-_{\bar{\imath}}$ and $x_{\bar{\imath}}$,

thereby either halving $\mathtt{d}^+(x_{\bar{i}}^-)$ or $\mathtt{d}^+(x_{\bar{i}})$, in either case decreasing Π by at least 1. Note that if $x_{\bar{i}}^-$ or $x_{\bar{i}}$ are deleted it is not until the second phase. In the other case, where $x_{\bar{i}}$ is the first element of $\mathcal{L}_{\bar{i}}^o$, we are inserting $\mathcal{L}_i'$ before $x_{\bar{i}}$, thereby decreasing $\mathtt{d}^-(x)$ from $2\mathtt{U}$ to at most $\mathtt{U}$, again decreasing Π by at least 1. This completes the basic potential function argument.

3.3 The final touch

We will now apply a bucketing argument, with which we reduce $O(\mathtt{N}\log\mathtt{U}\log\mathtt{N})$ to $O(\mathtt{N}\log^2(\mathtt{U}/\mathtt{N}))$. Fix $\mathtt{F} = \lceil \mathtt{U}/\mathtt{N} \rceil$. We have so far used a single list to manipulate all informers. However, we can partition these lists into $\mathtt{N}$ lists, each over a range of size $\mathtt{U}/\mathtt{N}$, by which we will reduce both the data structural time as well as tightening the analysis. We note for completeness that our complexity will become $O(\mathtt{N}\log(\mathtt{U}/\mathtt{N})\min\{\log\mathtt{N}, \log(\mathtt{U}/\mathtt{N})\})$. Specifically, before running Algorithm A we will preprocess the compressed string $\mathcal{Z}$ in two steps:

1. Insert cuts at all positions $k\,\mathtt{F} < \mathtt{U} - 1$ where $k \in \mathbb{N}_0$. That is, if a pair (P, L) in $\mathcal{Z}$ covers the interval $[U, U + L - 1]$, i.e. U is the sum of the lengths of the preceding pairs in $\mathcal{Z}$, and if $k\,\mathtt{F} \in [U + 1, U + L - 1]$, we replace (P, L) by the two pairs $(P, L')(P + L', L - L')$ where $L' = k\,\mathtt{F} - U$. Clearly, this step does not change the represented text, and the size of the compressed text is at most doubled by the at most $\lfloor \mathtt{U}/\lceil \mathtt{U}/\mathtt{F} \rceil \rfloor \leq \mathtt{N}$ new cuts.

2. For all pairs (P, L) where $k\mathtt{F} \in [P + 1, P + L - 1]$ for some $k \in \mathbb{N}_0$, cut (P, L) into the pairs $(P, L')(P + L', L - L')$ where $L' = k\,\mathtt{F} - P$.

After step 1 no length is greater than $\mathtt{F}$, so step 2 can only cut each pair once. Thus our new compressed text is still of size $O(\mathtt{N})$, and clearly it represents the same text $\mathcal{T}$.

Now suppose that both step 1 and step 2 have been applied to $\mathcal{Z}$ before the winding. A first advantage is that we can partition all our lists into buckets $B_k = [k\mathtt{F}, (k + 1)\mathtt{F} - 1]$. Step 1 above, ensures that for the cut step A.1.1, $\mathcal{L}_1$ is entirely contained in the bucket $B_{\lfloor U_i/\mathtt{F} \rfloor}$, and step 2 ensures that after the subtraction step A.1.2, $\mathcal{L}_1$ is entirely contained in $B_{\lfloor P_i/\mathtt{F} \rfloor}$. As there are only $\mathtt{F}$ possible informer positions in a bucket, all list operations are now on lists of length bounded by $\mathtt{F}$. Thus the complexity of each list operation is changed from $O(\log\mathtt{N})$ to $O(\log\mathtt{F}) = O(\log\mathtt{U}/\mathtt{N})$.

In fact the above preprocessing also allows us to tighten the potential function argument, reducing the number of merges from $O(\mathtt{N}\log\mathtt{U})$ to $O(\mathtt{N}\log\mathtt{U}/\mathtt{N})$. Our lists are now partitioned into buckets, and clearly, the maximum distance between two informers in the same bucket is bounded by $\mathtt{F}$. Thus, if we define $\mathtt{d}^-(x) = 2\mathtt{F}$ if x is the left-most informer in a bucket and $\mathtt{d}^+(x) = 2\mathtt{F}$ if x is the right-most informer in a bucket, then we get exactly the same potential function argument as before but with $\mathtt{U}$ replaced by $\mathtt{F}$. We may therefore conclude:

Theorem 3.1 *The winding takes $O(\mathtt{N}\log^2(\mathtt{U}/\mathtt{N}))$ time. Each of the lists involved has length $O(\mathtt{U}/\mathtt{N})$, and the total number of list operations is $O(\mathtt{N}\log\mathtt{U}/\mathtt{N})$.*

4 Unwinding

In this section, we will show how to unwind all left informers in $O(\mathtt{N}\log(\mathtt{U}/\mathtt{N})(\log\mathtt{U}/\mathtt{N} + \log\mathtt{P}))$ time. The right informers may be treated symmetrically, and they will not be discussed further. We are

now in the situation that we have shifted all the left informers into the alphabet, and we will now shift them back into their original positions, but loaded with information about the longest pattern substring extending to the left of their final position.

During the unwinding, if at some stage an informer is at position u, then it should be loaded with the longest suffix of $\mathcal{T}[0,u]$ which is a substring of $\mathcal{P}$. We will use a function LEFTAPPEND(α,β) which, given substrings α,β of the pattern, returns the longest suffix of $\alpha\beta$ which is a substring of the pattern. In [7], an algorithm for LEFTAPPEND was given which preprocesses a string of length P in $O(\mathrm{P})$ time and answers such queries in $O(\log \mathrm{P})$ time.

The basic idea is to log the winding of Algorithm A, so that we can reverse it directly in the unwinding. Thus, consider step s of the winding, where we first cut $\mathcal{L}$ at U_s into lists $\mathcal{L}_0$ and $\mathcal{L}_1$, second subtract $U_s - P_s$ from u for all informers in $\mathcal{L}_1$, and third merge $\mathcal{L}_0$ with $\mathcal{L}_1$ into $\mathcal{L}$, removing duplicates. In the unwinding of this step, our starting point is the list $\mathcal{L}$ with all informers at positions to the left of U_s, and each informer x being loaded with the longest suffix of $\mathcal{T}[0,\mathrm{u}(x)-1]$ which is a substring of $\mathcal{P}$. Moreover, we have the information that allows us to split $\mathcal{L}$ into $\mathcal{L}_0$ and $\mathcal{L}_1$. Being a bit more precise, we will see $\mathcal{L}$ as divided into elements $L_0,\ldots,L_{m_s-1}$ where each L_i is either a segment of $\mathcal{L}_0$, a segment of $\mathcal{L}_1$, or an informer that should be copied to both $\mathcal{L}_0$ and $\mathcal{L}_1$. From our potential function argument we know that the sum of the m_s over all steps is $O(\mathrm{N}\log \mathrm{U}/\mathrm{N})$. In order to use this bound, we need to be able to process each segment L_i in a time which is independent of the number of informers it contains. This independence is achieved by sparsification of the loading of the informers as follows.

Consider informers x_1,x_2,x_3 at positions u_1,u_2,u_3 where $u_1 < u_2 < u_3$. For $i=1,2,3$, let α_i be the longest suffix of $\mathcal{T}[0,u_i]$ which is a substring of $\mathcal{P}$. Suppose that $u_1 + |\alpha_3| \geq u_3$. Then $\alpha_2 = \text{LEFTAPPEND}(\alpha_1, \alpha_3[u_1 + |\alpha_3| - u_3, |\alpha_3| + u_2 - u_3 - 1])$, and then we say that x_1 and x_3 *cover* x_2. Now in such a case, we need not explicitly keep the information at x_2 up to date at all times since we can quickly recover such information as needed by one call to LEFTAPPEND. For a sequence of informers, we will then keep some informers loaded, that is, up to date, and other informers unloaded. We will say any such sequence is *proper* if each informer is either loaded or covered by loaded informers. If a sequence of informers is properly loaded and minimal, in the sense that no loaded informer is covered by two other loaded informers, then we will say the sequence is *economical*. In an economically loaded informer sequence, we will keep the positions of loaded informers in a balanced binary tree. This will allow us to update the list (with insertions, splits, concatenations, etc.) in $O(\log \mathrm{U}/\mathrm{N})$ time, since by the arguments above, no list will ever be of length greater than U/N. Also, each unloaded informer will be able to find its covering informers in time $O(\log \mathrm{U}/\mathrm{N})$. Thus, we will be able to load any informer with one tree lookup and one call to LEFTAPPEND.

The following lemmas allow us to manipulated economically loaded informer sequences.

Lemma 4.1 *Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be lists of informers, each of which has economical coverings. Then we can produce an economical covering of their concatenation $\mathcal{L}_0\mathcal{L}_1$ by changing $O(1)$ informers.*

Proof: We may need to unload the last informer of $\mathcal{L}_0$ and the first in $\mathcal{L}_1$. But we can do so by checking to see if they are covered in the new concatenated list. So we change the status of at most 2 informers. ∎

Lemma 4.2 *Let $\mathcal{L}$ be a list of informers with an economical covering. Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be the lists produced by splitting $\mathcal{L}$ at u. Then we can produce economical coverings for $\mathcal{L}_0$ and $\mathcal{L}_1$ changing $O(1)$ informers.*

Proof: Let x be the first informer in $\mathcal{L}_1$. Suppose that x is not loaded. Load x using its covering in $\mathcal{L}$. Let x' be the loaded neighbor to the right of x, and let x'' be the loaded neighbor to the right of x'. If x' is covered by x and x'', unload x'. Finally, update the list of loaded informers. The last informers in $\mathcal{L}_0$ is dealt with symmetrically, and in conclusion, we only make a constant number of informer tree operations and calls to LEFTAPPEND in the processing of $\mathcal{L}_0$ and $\mathcal{L}_1$. ■

Let us return to our sequence $\mathcal{L}$ which was divided into elements $L_0, \ldots, L_{m_s-1}$ where each L_i is either a segment of $\mathcal{L}_0$, a segment of $\mathcal{L}_1$, or a single informer that should be copied to both $\mathcal{L}_0$ and $\mathcal{L}_1$. Using Lemma 4.2 we get an economical loading for each segment L_i, and using Lemma 4.1 we concatenate the L_is into $\mathcal{L}_0$ and $\mathcal{L}_1$. Thereby, we get a total of $O(m_s)$ informer load changes.

The next part of redoing step s is to add $U_s - P_s$ to $\mathtt{u}$ for the informers in $\mathcal{L}_1$. This should change the specified contents of some of the informers, but we claim that among the loaded informers, it is only the two leftmost that get affected. The loading of an informer x in $\mathcal{L}_1$ should only change if it extends to U_s or beyond after the addition to the $\mathtt{u}$-values, that is if $\mathtt{u}(x) - |\mathtt{t}(x)| + 1 \leq U_s$. In this case, recall that we already have a correctly loaded left informer y at position $U_s - 1$. Thus, in order to get a correct loading of x, we set $\mathtt{t}(x) := \text{LEFTAPPEND}(\mathtt{t}(y), \alpha)$ where α is the suffix of $\mathtt{t}(x)$ of length $\mathtt{u}(x) - U_s + 1$. Now, let x, x' be the two leftmost loaded informers in $\mathcal{L}_1$, and let x'' be any loaded informer further to the right. Since $\mathcal{L}_1$ is economic, x' is not covered by x and x'', so $\mathtt{u}(x'') - |\mathtt{t}(x'')| > \mathtt{u}(x) = U_s$. Thus the loading of x'' need not be changed.

The final part in redoing step s is to concatenate $\mathcal{L}_0$ and $\mathcal{L}_1$, again by Lemma 4.1. In conclusion, the unwinding is done with $O(\mathtt{N} \log \mathtt{U}/\mathtt{N})$ list operations and calls to LEFTAPPEND.

Theorem 4.3 *We can load all informers in time $O(\mathtt{N} \log(\mathtt{U}/\mathtt{N})(\log \mathtt{U}/\mathtt{N} + \log \mathtt{P}))$.*

In [7], an algorithm was given to compute if a pattern occurs within the concatenation of two of its substrings. This operation takes $O(\mathtt{P})$ preprocessing on a string of length $\mathtt{P}$ and answers such queries in $O(\log \mathtt{P})$ time. This result suffices for the following.

Corollary 4.4 *We can do LZ77 compressed matching on a string of length $\mathtt{N}$ which represents a string of length $\mathtt{U}$ with a pattern of length $\mathtt{P}$ in time $O(\mathtt{N} \log(\mathtt{U}/\mathtt{N})(\log \mathtt{U}/\mathtt{N} + \log \mathtt{P}))$.*

In the following section, we show what to do if $\mathtt{P}$ is so long that $\log \mathtt{P} = \omega(\log \mathtt{U}/\mathtt{N})$.

5 Unwinding Long Patterns

In this section, we will address the problem of unwinding for long patterns. We will consider the case where $\mathtt{P} \geq \mathtt{F}^5$. Note that Corollary 4.4 settles Theorem 1.1 without randomization for cases where $\mathtt{P} = O(\mathtt{F}^5)$.

Recall that in §3.3, step 2, we introduced an artificial cut at each position which is a multiple of $\mathtt{F}$. Define k such that $(k+1)\mathtt{F} \leq \mathtt{P} < (k+2)\mathtt{F}$. Consider some pattern occurrence $\mathcal{P} = \mathcal{T}[j; j+\mathtt{P}-1]$, and let i be minimal such that $i\mathtt{F} \geq j$, i.e. $i = \mathtt{F}\lceil j/\mathtt{F} \rceil$. Then $\mathcal{P}$ can be written as BME where

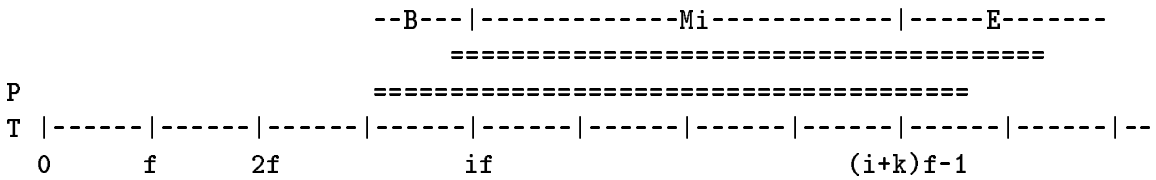

Figure 5.2:

$B = \mathcal{T}[j; i\mathtt{F} - 1]$, $M = \mathcal{T}[i\mathtt{F}; (i+k)\mathtt{F} - 1]$, and $E = \mathcal{T}[(i+k)\mathtt{F}, j + \mathtt{P} - 1]$. Note that $|B| < \mathtt{F}$ and $|E| < 2\mathtt{F}$. Also note that M only depends on i, hence, in the following, we will write it as M_i.

Let $\mathcal{P}^b$ denote the prefix of $\mathcal{P}$ of length $\mathtt{F} - 1$ and let $\mathcal{P}^e$ denote the suffix of $\mathcal{P}$ of length $2\mathtt{F} - 1$. Moreover, for each i, let B_i^* denote the longest substring of $\mathcal{P}^b$ which is a suffix of $\mathcal{T}[0, i\mathtt{F} - 1]$ and let E_i^* denote the longest substring of $\mathcal{P}^e$ which is a prefix of $\mathcal{T}[(i+k)\mathtt{F}, \mathtt{U}]$. Then with B and E depending on i as above, B is a suffix of B_i^* and E is a prefix of E_i^*, so $\mathcal{P}$ is a substring of $B_i^* M_i E_i^*$. Thus, we have reduced our task to finding each of the $O(\mathtt{N})$ values of B_i^*, M_i, and E_i^*, and to checking if their concatenations gives a match.

To find the B_i^*s we run our previous algorithm with pattern P^b, focusing on the left informers. Similarly, to find the E_i^*s we run the algorithm with pattern P^e, focusing on the right informers. By Corollary 4.4, since $|\mathcal{P}^b|, |\mathcal{P}^e| = O(\mathtt{F}) = O(\mathtt{U}/\mathtt{N})$, the B_i^*s and E_i^*s are found in $O(\mathtt{N} \log^2(\mathtt{U}/\mathtt{N}) + \mathtt{P})$ time. To compute if there is a match in $B_i^* M_i E_i^*$, we observe that there are only $O(\mathtt{F}^3)$ possible such combinations, and that each can be checked in $O(\mathtt{F}^2)$ time. So in $O(\mathtt{F}^5) = O(\mathtt{P})$ time, we can precompute all possible queries.

So the real question is how to compute the M_i, which we do as follows. Rather than compute the M_i themselves, we will show how to compute the *fingerprints* of the M_i. The notion of a string fingerprint was introduced by Karp and Rabin [8]. They showed that if a string over an alphabet of size σ is taken to be a base σ number, and further taken modulo some random prime $\wp$, then two distinct strings have the same such fingerprint with probability $1/\wp$. In our case, we will chose a random prime to be on the order of u^5, thus ensuring that the probability that two distinct substrings of $\mathcal{T}$ have identical fingerprints is polynomially small. The main significance of fingerprints in our context is that given the fingerprints of two strings, we can find the fingerprint of their concatenation in constant time. In the following we will denote the fingerprint modulo $\wp$ of $\mathcal{T}[x, y]$ as $\mathcal{F}[x, y]$.

We will now show how to compute the fingerprints of all the M_is. During the unwinding, if a (left or right) informer is at position j, it will contain $\mathcal{F}[0; j]$. This will give us the desired result, for consider the situation when the unwinding is finished. Then for $i = 1, \ldots, \lfloor \mathtt{U}/\mathtt{F} \rfloor - 1$, we have a left informer at position $i\mathtt{F} - 1$ and a left informer at position $(i+k)\mathtt{F} - 1$, i.e. we know $\mathcal{F}[0; i\mathtt{F} - 1]$ and $\mathcal{F}[0; (i+k)\mathtt{F} - 1]$. But then $\mathcal{F}[if; (i+k)\mathtt{F} - 1]$ can be found in constant time since $\mathcal{F}[i\mathtt{F}; (i+k)\mathtt{F} - 1] = (\mathcal{F}[0; (i+k)\mathtt{F} - 1] - \mathcal{F}[0; i\mathtt{F} - 1])\sigma^{1-i\mathtt{F}} \bmod \wp$. Computing the needed powers of $\sigma \pmod{\wp}$ is standard (see [5, Chapter 33]).

We are now ready to describe the unwinding. Besides the above information, an informer at position j will contain the fingerprint of the character $\mathcal{T}[j]$. Thus for each informer x, in the unwinding we will maintain variables $\mathtt{pref}(x) = \mathcal{F}[0; \mathtt{u}(x)]$ and $\mathtt{char}(x) = \mathcal{F}[\mathtt{u}(x); \mathtt{u}(x)]$. Note that

following the definition of the compression scheme, $\texttt{char}(x)$ is not changed once it has been set.

Our task is to reverse steps A.1.1–A.1.3 in Algorithm A, or, in fact, we only need to change the contents of the informers in connection with steps A.1.1–A.1.2. Let $\mathcal{L}$, $\mathcal{L}_1$, $\mathcal{L}_2$, U_s and P_s be as in Algorithm A. Thus, we want to add $U_s - P_s$ to the positions of all informers in $\mathcal{L}_2$ and then set $\mathcal{L} := \mathcal{L}_1\mathcal{L}_2$. First, concerning the base case, where $\mathcal{L}_2$ consists of a single informer x which is moved from the alphabet at the negative position $-C$. Hence we set $\texttt{char}(x) := C \bmod \wp$. If $s = 0$, we set $\texttt{pref}(x) := \texttt{char}(x)$. Otherwise, we note that we have a left informer l_{s-1} in $\mathcal{L}_1$ at position $U_s - 1$, and hence we set

$$\texttt{pref}(x) := \texttt{pref}(l_s) + \texttt{char}(x)\sigma^{U_s-1} \bmod \wp.$$

We are now ready for the real case where $\mathcal{L}_2$ is a sequence of informers in the text with positive u-values. We will exploit that we have a left informer l_{s-1} at position $U_s - 1$ in $\mathcal{L}_1$ and that the right-most informer r_s of $\mathcal{L}_2$ is going to be put at position U_s. The needed transformation for each informer x in $\mathcal{L}_2$ is now:

- $\texttt{pref}(x) := \texttt{pref}(x) - \texttt{pref}(r_s) + \texttt{char}(r_s)\sigma^{P_s-1} \bmod \wp$.
- $\texttt{pref}(x) := \texttt{pref}(x)\sigma^{U_s-P_s} \bmod \wp$.
- $\texttt{pref}(x) := \texttt{pref}(l_{s-1}) + \texttt{pref}(x) \bmod \wp$.

This procedure depends on a data structure on trees which supports the following operations: in time $O(\log \texttt{F})$ it should be able to multiply or add to pref for all elements in a list of length F, and it should support cuts and concatenations on such lists. See cutting and linking trees in [12]. As a result, the work connected with maintaining pref and char is $O(\texttt{N} \log \texttt{F})$. Combining with Theorem 3.1 and Corollary 4.4, the whole compressed matching is done in time $O(\texttt{N} \log^2 \texttt{F} + \texttt{P})$, thus establishing Theorem 1.1 with a Monte Carlo algorithm for the case without self-references.

Finally, we need to show how to check our output, thus changing the Monte Carlo algorithm above to a Las Vegas algorithm. We also need to show how to deal with self-references. Once these technical steps are performed, we will have completed the proof of Theorem 1.1.

6 Checking the Answer

The randomized algorithm from Section 5 for the case of long patterns is of the Monte Carlo type, that is, it has a guaranteed running time but may sometimes report pattern occurrence where none exists. In order to guarantee the correctness of the output, at the risk of increasing the running time, we have to test any pattern occurrence found by the algorithm. A simple $O((\texttt{P} + \texttt{N}) \log^2(\texttt{U}/\texttt{N}))$ solution to the checking problem would be to place an informer on each of the P positions of the potential pattern occurrence. We could then wind these informers back into the alphabet in $O((\texttt{P} + \texttt{N}) \log^2(\texttt{U}/\texttt{N}))$ time, thereby checking each character of the potential occurrence. However, it is possible to get an $O(\texttt{P} + \texttt{N} \log^2(\texttt{U}/\texttt{N}))$ checking algorithm. Generally the idea is that we load the informers with pattern substrings corresponding to the potential pattern occurrence. Then we start winding. As the winding proceeds, some of these substrings will be cut and distributed over several informers, and others will start overlapping. In the latter case, we have to check that the

pattern substrings agree in the overlap. Such a check can be done in constant time, after an $O(\mathtt{P})$ time preprocessing, via standard suffix tree and least common ancestors techniques. At the end, the surviving informers will end up in the alphabet, where we have to check that the characters match.

To be more concrete, we want to check the long middle of a pattern occurrence $\mathcal{P}[j, j+k\mathtt{F}] = \mathcal{T}[i\mathtt{F}, (i+k)\mathtt{F}-1]$ suggested by technique from Section 5. First, before the winding, for $a = 1, \ldots, k$, we load the left informer x at position $(i+a)\mathtt{F}-1$ with $\mathtt{t}(x) = \mathcal{P}[j+(a-1)\mathtt{F}, j+a\mathtt{F}]$. No other left informers are loaded. Then we start winding following Algorithm A from Section 3. At the beginning of any iteration, we have a sequence $x_1, \ldots, x_l$ of loaded informers. Our suggested pattern occurence is correct if and only if $\mathtt{t}(x_i)$ is a suffix of $\mathcal{T}[0, \mathtt{u}(x_i)]$ for all i. The covering is kept *non-overlapping* in the sense that for all $i > 1$, $\mathtt{u}(x_i) - |\mathtt{t}(x_i)| \geq \mathtt{u}(x_{i-1})$.

Consider a cut in step A.1.1. Let x be the first informer in $\mathcal{L}_1$. Suppose $\mathtt{t}(x)$ extends past U_s, i.e. $\mathtt{u}(x) - |\mathtt{t}(x)| + 1 < U_s$. Since the covering is non-overlapping, this implies that the left informer y at position $U_s - 1$ is unloaded. Let $\alpha\beta = \mathtt{t}(x)$ where $|\alpha| = \mathtt{u}(x) - U_s + 1$. Set $\mathtt{t}(x) = \beta$ and $\mathtt{t}(y) = \alpha$. In this way, each cut loads at most one informer. The above cut is implemented in time $O(\log \mathtt{F})$, giving a total cost of $O(\mathtt{N} \log \mathtt{F})$.

The merging of $\mathcal{L}_0$ and $\mathcal{L}_1$ in step A.1.3, consists of a series of segment merges, where we take a segment $(x_a, \ldots, x_b)$ of loaded informers from one list, and insert after a loaded informer y from the other list. Now *match* x_a with y as follows. If $\mathtt{t}(x_a)$ extends beyond $\mathtt{u}(y)$, i.e. if $\mathtt{u}(x_a) - |\mathtt{t}(x_a)| < \mathtt{u}(y)$, let $\alpha\beta = \mathtt{t}(x_a)$ where $|\beta| = \mathtt{u}(x_a) - \mathtt{u}(y)$. Check in constant time that the shorter of α and $\mathtt{t}(y)$ is a suffix of the other. If not, report that the pattern occurence was incorrect. Otherwise, if α is shorter than $\mathtt{t}(y)$, set $\mathtt{t}(x_a) = \beta$. If α is longer than $\mathtt{t}(y)$, unload y and match x_a with y's loaded predecessor.

Ignoring the repeated unloading from the recursive matching, each segment merge is implemented in time $O(\log \mathtt{F})$, and from our potential function argument, we know that there are at most $O(\mathtt{N} \log \mathtt{F})$ segment merges, amounting to a total cost of $O(\mathtt{N} \log^2 \mathtt{F})$.

Attributing a cost of $O(\log \mathtt{F})$ to the segment merge, the repeated unloading can be done in time linear in the number of unloads. We start with $< \mathtt{P}$ loaded informers, and cuts result in the loading of $O(\mathtt{N})$ informers. Hence the number of unloadings is bounded by $O(\mathtt{P} + \mathtt{N})$, so total cost of merging is $O(\mathtt{P} + \mathtt{N} \log^2 \mathtt{F})$.

At the end of the winding, we have have a sequence $x_1, \ldots, x_l$ of informers in the alphabet. Recall that they are all loaded with sub-patterns of $\mathcal{P}$, and we now have to check that this loading match the alphabet. Since the aphabet does not repeat characters and since the covering is non-overlapping, the loadings must correspond to disjoint sub-strings of $\mathcal{P}$. In particular, this implies that if $\sum_i |\mathtt{t}(x_i)| > |\mathcal{P}|$, we may report that the suggested pattern occurence is incorrect. Otherwise, in time $O(\mathtt{P})$ it is trivial to check the loadings. That is, the last checking is done in time $O(\mathtt{P})$.

In conclusion, we have checked our pattern occurence in time $O(\mathtt{P} + \mathtt{N} \log^2 \mathtt{F}) = O(\mathtt{P} + \mathtt{N} \log^2(\mathtt{U}/\mathtt{N}))$, thus achieving a Las Vegas type randomization.

7 Self-reference

Referring to the symbols as they were defined in Sections 1–2, for $i = 0, \ldots, \mathtt{N}-1$, let $D_i = U_i - P_i$. So far we have required that our compressed text $\mathcal{Z}$ is non-referential, i.e. that $D_i \geq L_i$, but now

we will show how to deal with the general case without this restriction. Thus, our only requirement is that $D_i > 0$, as in the original definition of Lempel-Ziv compressed texts.

Note the following alternative but equivalent statement of Eq. 1 from Section 1:

$$\begin{aligned} &\forall i = 0, \ldots, \mathtt{N} - 1 : \\ &\mathcal{T}[U_i, U_i + L_i - 1] \text{ is a prefix of } \mathcal{T}[P_i, P_i + D_i - 1]^\infty \end{aligned} \tag{2}$$

Winding

With Eq. 2 in mind, we can easily generalize our winding algorithm to deal with the general case.

Algorithm C Winding algorithm allowing self-references.

C.1. For $s := \mathtt{N} - 1$ downto 0 do

C.1.1. While the last informer x in $\mathcal{L}$ has $\mathtt{u}(x) \geq U_s$ do:

C.1.1.1. Cut $\mathcal{L}$ at U_s into lists $\mathcal{L}_0$ and $\mathcal{L}_1$.

C.1.1.2. Let x be the first informer in $\mathcal{L}_1$, and set $k = \lfloor (\mathtt{u}(x) - U_s)/D_s \rfloor$.

C.1.1.3. Subtract $(k+1)D_s$ from u for all informers in $\mathcal{L}_1$.

C.1.1.4. Merge $\mathcal{L}_0$ with $\mathcal{L}_1$ into $\mathcal{L}$, removing informers that are duplicate with respect to u-values.

Above, if $k = 0$, we are following Eq. 1 directly. If $k > 0$, we are just skipping loops with superfluous merges where the first informer in $\mathcal{L}_1$ lands strictly before the last informer in $\mathcal{L}_0$. Extending the potential function argument from Section 3, we will show that with this little optimization, the general winding of Algorithm C is performed within the same time bounds as we had for the winding without self-references in Algorithm A.

For each of the $O(\mathtt{N})$ iterations of the outer for-loop (step C.1), we will show that Π increases by at most $O(\log \mathtt{F})$. Moreover, we will show that each iteration of the inner while-loop (step C.1.1) decreases Π by $\Omega(1)$. Since Π starts at $O(\mathtt{N} \log \mathtt{F})$, this will allow us to conclude that we have a total of $O(\mathtt{N} \log \mathtt{F})$ while-loop iterations.

Consider a specific for-loop iteration (step C.1), fixing some value of s. To facilitate the proof, we insert two auxiliary informers a and b where $\mathtt{u}(a) = P_s$ and $\mathtt{u}(b) = U_s - 1$. These will be removed again at the end of the for-loop, if they have not already been removed as duplicates. Note that b will always be the last informer in $\mathcal{L}_0$. Technically, we fix $\mathtt{d}^+(b) = 2\mathtt{F}$. Hence $\pi(b)$ is not affected when cutting $\mathcal{L}$ into $\mathcal{L}_0$ and $\mathcal{L}_1$. Inserting or removing an informer affects the potential of at most two other informers; namely its neighbors. Thus our insertion and removal of a and b increases Π by $O(\log \mathtt{F})$, and this will capture the whole increase to Π during the for-loop.

Consider an iteration of the while-loop (step C.1.1). Let x be the first informer in $\mathcal{L}_1$. Since $\mathtt{d}^+(b)$ is fixed, $\mathtt{d}^-(x)$ is the only $\mathtt{d}^{+/-}$ distance that could possibly increase during the while-loop. However, before the while-loop, $\mathtt{d}^+(x)$ is the distance to b, but the distance to a is at most $D_s - 1$ bigger, and step C.1.1.3 moves x a positive multiple of D_s closer to a. Thus $\mathtt{d}^+(x)$ is, in fact, decreased during the while-loop, that is, no informer inceases potential during the while-loop.

If x lands on another informer, an informer gets removed, and Π decreases by $\Omega(1)$, as desired. Otherwise, x lands between a and b. Let y and z, $\mathtt{u}(y) < \mathtt{u}(z)$, be its nearest neighbors from $\mathcal{L}_0$

after the merge. Then, either $\mathtt{d}^+(y)$ or $\mathtt{d}^-(z)$ is halved by the merge, so we conclude that Π is decreased by $\Omega(1)$ during the while-loop.

The argument from Section 3, that each recursive call to `Segment-Merge` decreases Π by $\Omega(1)$, still holds. Thus, the total number of while-loops and recursive calls to `Segment-Merge` is bounded by the initial value plus the total increase of Π, which is $O(\mathrm{N}\log\mathrm{F})$. As in Section 3, this again bounds the total number of list operations, each of which takes time $O(\log\mathrm{F})$. We may therefore conclude that the general winding of Algorithm C, which allows self-reference, satisfies the description in Theorem 3.1.

Unwinding

For the unwinding, we can essentailly follow the pattern from Section 4. Again, we are only focussing on left informers as right informers may be treated symmetrically. The only critical point is what happens to the loading of the first left informer x in $\mathcal{L}_1$ when redoing the shift in step C.1.1.3. Let s, x, $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}_1$, and k be fixed according to a specific iteration of the shift in step C.1.1.3.

As in Section 4, we already have a correctly loaded left informer y at position $U_s - 1$. Let $\mathtt{u}^o(x)$ and $\mathtt{u}^n(x)$ denote the position of x before and after the shift. If $|\mathtt{t}(x)| < \mathtt{u}^o(x) - P_s + 1$, $\mathtt{t}(x)$ does not need to be changed, so suppose that this is not the case. Let α be the suffix of $\mathtt{t}(x)$ of length $\mathtt{u}^o(x) - P_s + 1$. If $|\mathtt{t}(y)| < D_s$, as in Section 4, we set $\mathtt{t}(x) := \textsc{LeftAppend}(\mathtt{t}(y), \alpha)$. However, suppose $|\mathtt{t}(y)| \geq D_s$. Let $\gamma\beta = \mathtt{t}(y)$ where $|\beta| = D_s$. Then $\mathtt{t}(x) := \textsc{LeftAppend}(\gamma\beta^{k+1}, \alpha)$. The pattern preprocessing for LeftAppend from [7] allows queries in time $O(\log\mathrm{P})$ despite the β^{k+1} in the argument.

The unwinding of long paterns from Section 5 is changed similarly, so this completes the proof of Theorem 1.1.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] A. Amir and G. Benson. Efficient two dimensional compressed matching. *Proc. of the 2nd IEEE Data Compression Conference*, pages 279–288, Mar 1992.

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Proc. of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1994.

[4] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. *Proc. of 21st International Colloquium on Automata Languages and Programming*, 1994.

[5] Corman, Leiserson, and Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[6] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression. *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[7] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. *Proc. of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1994.

[8] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[9] S. Rao Kosaraju. On the entropy estimation of low entropy sources. Personal Communication, 1994.

[10] S. Rao Kosaraju. Pattern matching in compressed texts. *Proc. of the 10th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, 1995.

[11] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22:75–81, 1976.

[12] D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 24, 1983.

[13] L. Stauffer and D. Hirschberg. Pram algorithms for static dictionary compression. *8th International Parallel Processing Symposium*, 1994.

[14] J. Storer. *Data compression: methods and theory.* Computer Science Press, Rockville, Maryland, 1988.

[15] T. A. Welch. A technique for high-performance data compression. *IEEE Transactions on Computers*, 17:8–19, 1984.

[16] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.