Approximating LZ77 via Small-Space Multiple-Pattern Matching

Johannes Fischer[1], Travis Gagie[2⋆], Paweł Gawrychowski[3⋆⋆], and Tomasz Kociumaka[3⋆⋆⋆]

[1] TU Dortmund, Germany, johannes.fischer@cs.tu-dortmund.de
[2] University of Helsinki, Finland, travis.gagie@cs.helsinki.fi
[3] Institute of Informatics, University of Warsaw, Poland
{gawry,kociumaka}@mimuw.edu.pl

Abstract. We generalize Karp-Rabin string matching to handle multiple patterns in $\mathcal{O}(n \log n + m)$ time and $\mathcal{O}(s)$ space, where n is the length of the text and m is the total length of the s patterns, returning correct answers with high probability. As a prime application of our algorithm, we show how to approximate the LZ77 parse of a string of length n in $\mathcal{O}(n \log n)$ time. If the optimal parse consists of z phrases, using only $\mathcal{O}(z)$ working space we can return a parse consisting of at most $2z$ phrases in $\mathcal{O}(n \log n)$ time, and a parse of at most $(1+\varepsilon)z$ phrases in $\mathcal{O}(n \log^2 n)$ for any constant $\varepsilon > 0$. As previous quasilinear-time algorithms for LZ77 use $\Omega(n/\mathrm{poly}\log n)$ space, but z can be exponentially small in n, these improvements in space are substantial.

1 Introduction

Multiple-pattern matching, the task of locating the occ occurrences of s patterns of total length m in a single text of length n, is a fundamental problem in the field of string algorithms. The famous algorithm by Aho and Corasick [1] solves this problem using $\mathcal{O}(n+m)$ time and $\mathcal{O}(m)$ working space in addition to the space needed for the text and patterns. To list all occurrences rather than, e.g., the leftmost ones, extra $\mathcal{O}(\mathsf{occ})$ time is necessary. When the space is limited, we can use a compressed Aho-Corasick automaton [8]. In extreme cases, one could apply a linear-time constant-space single-pattern matching algorithm sequentially for each pattern in turn, at the cost of increasing the running time to $\mathcal{O}(n \cdot s + m)$. Well-known examples of such algorithms include those by Galil and Seiferas [6], Crochemore and Perrin [4] and Karp and Rabin [10] (see [2] for a recent survey).

It is easy to generalize Karp-Rabin matching to handle multiple patterns in $\mathcal{O}(n+m)$ expected time and $\mathcal{O}(s)$ working space provided that all patterns are

⋆ Supported by Academy of Finland grant 268324.
⋆⋆ Work done while the author held a post-doctoral position at Warsaw Center of Mathematics and Computer Science.
⋆⋆⋆ Supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

of the same length [7]. To do this, we store the fingerprints of the patterns in a hash table, and then slide a window over the text maintaining the fingerprint of the fragment currently in the window. Using the hash table, we can then check if the window corresponds to an occurrence of a patterns and if so, report it, updating the hash table so that every pattern is reported at most once. This is a very simple and actually applied idea [15], but it is not clear how to extend it for patterns with many distinct lengths. As far as we know, this question has not been addressed in the literature so far. In this paper we develop a method which works for any set of patterns in $\mathcal{O}(n \log n + m)$ time and $\mathcal{O}(s)$ working space. Since multiple-pattern matching is a very widely used tool, we believe that such generalization is of major interest.

As a prime application, we show how to approximate the Lempel-Ziv 77 (LZ77) parse [16] of a text of length n, while using $\mathcal{O}(n \log n)$ time and working space proportional to the number of phrases. Computing the LZ77 parse in small space is an issue of high importance, with space being a frequent bottleneck of today's systems. Moreover, LZ77 is useful not only for data compression, but also as a way to speed up algorithms [12]. We are only aware of one non-trivial sublinear space algorithm for computing the LZ77 parse due to Kärkkäinen et al. [9], which runs in $\mathcal{O}(nd)$ time and uses $\mathcal{O}(n/d)$ space for any parameter d. We present two solutions to this problem, both of which work in $\mathcal{O}(z)$ space for inputs admitting LZ77 parsing with z phrases. The first one produces a parse consisting of at most $2z$ phrases in $\mathcal{O}(n \log n)$ time, while the other for any positive $\varepsilon < 1$ in $\mathcal{O}(\varepsilon^{-1} n \log^2 n)$ time generates a factorization with no more than $(1+\varepsilon)z$ phrases. To the best of our knowledge, all previous algorithms use either significantly more time or significantly more space when the parse is small relative to n; see [5] for a recent discussion.

Sect. 2 introduces terminology and recalls several known concepts. This is followed by the description of our pattern-matching algorithm. In Sect. 3 we show how to process patterns of length at most s and in Sect. 4 we handle longer patterns, with different procedures for repetitive and non-repetitive ones. Finally, in Sect. 5, we apply the matching algorithm to construct the approximations of the LZ77 parsing.

Our algorithms are designed for the word-RAM with $\Omega(\log n)$-bit words and assume integer alphabet of polynomial size. The usage of Karp-Rabin fingerprints makes them Monte Carlo, despite they are otherwise deterministic. However, it is not difficult to make them Las Vegas as explained in Appendix C. We assume read-only random access to the text and the patterns, and we do not include their sizes while measuring space consumption.

2 Preliminaries

We consider finite words over an integer alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$, where $\sigma = \text{poly}(n + m)$. For a word $w = w[1] \ldots w[n] \in \Sigma^n$, we define the *length* of w as $|w| = n$. For $1 \leq i \leq j \leq n$, a word $u = w[i] \ldots w[j]$ is called a *subword* of w. By $w[i..j]$ we denote the occurrence of u at position i, called a *fragment* of w. A

fragment with $i = 1$ is called a *prefix* (and often denoted $w[..j]$) and a fragment with $j = n$ is called a *suffix* (and denoted $w[i..]$).

A positive integer p is called a period of w whenever $w[i] = w[i + p]$ for $1 \leq i \leq |w| - p$. In this case, the prefix $w[..p]$ is often also called a period of w. The length of the shortest period of a word w is denoted as $\mathrm{per}(w)$. A word w is called *periodic* if $\mathrm{per}(w) \leq \frac{|w|}{2}$ and *highly periodic* if $\mathrm{per}(w) \leq \frac{|w|}{3}$. The well-known periodicity lemma says that if p and q are both periods of w, and $p + q \leq |w|$, then $\gcd(p, q)$ is also a period of w.

2.1 Fingerprints

Our randomized construction is based on Karp-Rabin fingerprints, see [10]. Fix a word $w[1..n]$ over an alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$, a prime number $p > \max(\sigma, n^{c+4})$, and choose $x \in \mathbb{Z}_p^*$ uniformly at random. We define the fingerprint of a subword $w[i..j]$ as $\Phi(w[i..j]) = w[i] + w[i+1]x + \ldots + w[j]x^{j-i} \bmod p$. With high probability (that is, probability at least $1 - \frac{1}{n^c}$), no two distinct subwords of the same length have equal fingerprints. The situation when this happens for some two subwords is called a *false-positive*. From now on when stating the results we assume that there are no false-positives to avoid repeating that the answers are correct with high probability. For pattern matching, we apply this construction for $w = TP_1 \ldots P_s$, the concatenation of the text with the patterns. Fingerprints let us easily locate many patterns of the same length, as mentioned in the introduction. Here, we give a solution avoiding hash tables.

Theorem 1. *Given a text T of length n and patterns $P_1, \ldots, P_s$, each of length exactly ℓ, we can compute the the leftmost occurrence of every pattern P_i in T using $\mathcal{O}(n + s\ell + s \log s)$ total time and $\mathcal{O}(s)$ space.*

Proof. We calculate the fingerprint $\Phi(P_j)$ of every pattern. Then we build in $\mathcal{O}(s \log s)$ time [13] a deterministic dictionary $\mathcal{D}$ with an entry mapping $\Phi(P_j)$ to j. For multiple identical patterns we create just one entry, and at the end we copy the answers to all instances of the pattern. Then we scan the text T with a sliding window of length ℓ while maintaining the fingerprint $\Phi(T[i..i + \ell - 1])$ of the current window. Using $\mathcal{D}$, we can find in $\mathcal{O}(1)$ time an index j such that $\Phi(T[i..i + \ell - 1]) = \Phi(P_j)$, if any, and update the answer for P_j if needed (i.e., if there was no occurrence of P_j before). If we precompute x^{-1}, the fingerprints $\Phi(T[i..i + \ell - 1])$ can be updated in $\mathcal{O}(1)$ time while increasing i. □

2.2 Tries

A trie of a collection of patterns $P_1, \ldots, P_s$ is a rooted tree whose nodes correspond to prefixes of the patterns. The root represents the empty word and the edges are labeled with single characters. The node corresponding to a particular prefix is called its *locus*. In a *compacted* trie unary nodes that do not represent any pattern are *dissolved* and the labels of their incidents edges are concatenated. The dissolved nodes are called *implicit* as opposed to the *explicit* nodes,

which remain stored. The locus of a string in a compacted trie might therefore be explicit or implicit. All edges outgoing from the same node are stored on a list sorted according to the first character, which is unique among these edges.

Consider two compacted tries $\mathcal{T}_1$ and $\mathcal{T}_2$. We say that (possibly implicit) nodes $v_1 \in \mathcal{T}_1$ and $v_2 \in \mathcal{T}_2$ are *twins* if they are loci of the same string. Note that every $v_1 \in T_1$ has at most one twin $v_2 \in T_2$.

Lemma 2. *Given two compacted tries $\mathcal{T}_1$ and $\mathcal{T}_2$ of size s_1 and s_2, respectively, in $\mathcal{O}(s_1 + s_2)$ total time and space we can find for each explicit node $v_1 \in \mathcal{T}_1$ a node $v_2 \in \mathcal{T}_2$ such that if v_1 has a twin in $\mathcal{T}_2$, then v_2 is the twin.*

Proof. We recursively traverse both tries while maintaining a pair of nodes $v_1 \in \mathcal{T}_1$ and $v_2 \in \mathcal{T}_2$, starting with the root of $\mathcal{T}_1$ and $\mathcal{T}_2$ satisfying the following invariant: either v_1 and v_2 are twins, or v_1 has no twin in T_2. If v_1 is explicit, we store v_2 as the candidate for its twin. Next, we list the (possibly implicit) children of v_1 and v_2 and match them according to the edge labels with a linear scan. We recurse on all pairs of matched children. If both v_1 and v_2 are implicit, we simply advance to their immediate children. The last step is repeated until we reach an explicit node in at least one of the tries, so we keep it implicit in the implementation to make sure that the total number of operations is $\mathcal{O}(s_1 + s_2)$. If a node $v \in T_1$ is not visited during the traversal, for sure it has no twin in T_2. Otherwise, we compute a single candidate for its twin. □

3 Short Patterns

To handle the patterns of length not exceeding a given threshold ℓ, we first build a compacted trie for those patterns. Its edge labels are stored as pointers to the respective subwords of the pattern so that the whole trie takes $\mathcal{O}(s)$ space. Construction is easy if the patterns are sorted: we insert them one by one into the compacted trie first naively traversing the trie from the root, then potentially partitioning one edge into two parts, and finally adding a leaf if necessary. The proof of the following technical result is provided in Appendix A.

Lemma 3. *One can lexicographically sort strings $P_1, \ldots, P_s$ of total length m in $\mathcal{O}(m + \sigma^\varepsilon)$ time using $\mathcal{O}(s)$ space, for any $\varepsilon > 0$.*

Next, we partition T into $\mathcal{O}(\frac{n}{\ell})$ overlapping blocks $T_1 = T[1..2\ell]$, $T_2 = T[\ell + 1..3\ell]$, $T_3 = T[2\ell + 1..4\ell], \ldots$. Notice that each substring of length at most ℓ is completely contained in some block. Thus, we may seek for occurrences of the patterns in each block separately.

The suffix tree of each block T_i takes $\mathcal{O}(\ell \log \ell)$ time [14] and $\mathcal{O}(\ell)$ space to construct and store (the suffix tree is discarded after processing the block). We apply Lemma 2 to the suffix tree and the compacted trie of patterns; this takes $\mathcal{O}(\ell + s)$ time. For each pattern P_j we obtain a node such that the corresponding substring is equal to P_j provided that P_j occurs in T_i. We compute the leftmost occurrence $T_i[\ell..r]$ of the substring, which takes constant time if we store additional data at every explicit node of the suffix tree, and then we check whether

$T_i[\ell..r] = P_j$ using fingerprints. For this, we precompute the fingerprints of all patterns, and for each block T_i we precompute the fingerprints of its prefixes in $\mathcal{O}(\ell)$ time and space, which allows to determine the fingerprint of any of its substrings in constant time.

In total, we spend $\mathcal{O}(m + \sigma^\varepsilon)$ for preprocessing and $\mathcal{O}(\ell \log \ell + s)$ for each block. Since $\sigma = (n+m)^{\mathcal{O}(1)}$, for small enough ε this yields the following result.

Theorem 4. *Given a text T of length n and patterns $P_1, \ldots, P_s$ of total length m, using $\mathcal{O}(n \log \ell + s\frac{n}{\ell} + m)$ total time and $\mathcal{O}(s + \ell)$ space we can compute the leftmost occurrences in T of all patterns P_j of length at most ℓ.*

4 Long Patterns

To handle patterns of longer than a certain threshold, we first distribute them into groups according to the value of $\lfloor \log_{4/3} |P_j| \rfloor$. Patterns longer than the text can be ignored, so there are $\mathcal{O}(\log n)$ groups. Each group is handled separately, and from now on we consider only patterns P_j belonging to the i-th group. Let $\ell = \lceil (4/3)^{i-1} \rceil$ and define α_j and β_j be the prefix and suffix of length ℓ, respectively, of P_j. Since $\frac{2}{3}(|\alpha_j| + |\beta_j|) = \frac{4}{3}\ell \geq |P_j|$, the following lemma, proved in Appendix B, yields the classification of the patterns into three classes: either P_j is highly periodic, or α_j is not highly periodic, or β_j is not highly periodic.

Lemma 5. *Suppose x and y are a prefix and a suffix of a word w such that $|w| \leq \frac{2}{3}(|x| + |y|)$. If w is not highly periodic, then x or y is not highly periodic.*

To assign every pattern to the appropriate class, we compute the periods of P_j, α_j and β_j using small space. Roughly the same result has been proved in [11], but for completeness we provide the full proof in Appendix B.

Lemma 6. *Given a read-only string w one can decide in $\mathcal{O}(|w|)$ time and constant space if w is periodic and if so, compute* $\mathrm{per}(w)$.

4.1 Patterns without Long Highly Periodic Prefix

Below we show how to deal with patterns with non-highly periodic prefixes α_j. Patterns with non-highly periodic suffixes β_j can be processed using the same method after reversing the text and the patterns.

Lemma 7. *Let ℓ be an arbitrary integer. Suppose we are given a text T of length n and patterns $P_1, \ldots, P_s$ such that $\ell \leq |P_j| < \frac{4}{3}\ell$ and $\alpha_j = P_j[1..\ell]$ is not highly periodic for every j. We can compute the leftmost and the rightmost occurrence of every pattern in T in $\mathcal{O}(n + s(1 + \frac{n}{\ell}) \log s + s\ell)$ total time using $\mathcal{O}(s)$ space.*

The algorithm scans the text T with a sliding windows of length ℓ. Whenever it encounters a substring equal to the prefix α_j of some P_j, it creates a *request* to verify whether the corresponding suffix β_j of length ℓ occurs at the respective position. The request is processed when the sliding window reaches that position.

This way the algorithm detects the occurrences of all the patterns. In particular, we may store the leftmost and rightmost of occurrences of each pattern.

We use the fingerprints to compare the substrings of T with α_j and β_j. To this end, we precompute $\Phi(\alpha_j)$ and $\Phi(\beta_j)$ for each j. We also build a deterministic dictionary $\mathcal{D}$ [13] with an entry mapping $\Phi(\alpha_j)$ to j for every pattern (if there are multiple patterns with the same value of $\Phi(\alpha_j)$, the dictionary maps a fingerprint to a list of indices). These steps take $\mathcal{O}(s\ell)$ and $\mathcal{O}(s \log s)$, respectively. Pending requests are maintained in a priority queue $\mathcal{Q}$, implemented using a binary heap, as pairs containing the pattern index (as a value) and the position where the occurrence of β_j is anticipated (as a key).

Algorithm 1: Processing phase for patterns with non-highly periodic α_j.

```
1 for i = 1 to n − ℓ + 1 do
2 |   h := Φ(w[i..i + ℓ − 1])
3 |   foreach j : Φ(α_j) = h do
4 |   |   add a request (i + |P_j| − ℓ, j) to Q
5 |   foreach request (i, j) ∈ Q at position i do
6 |   |   if h = Φ(β_j) then
7 |   |   |   report an occurrence of P_j at i + ℓ − |P_j|
8 |   |   remove (i, j) from Q
```

Algorithm 1 provides a detailed description of the processing phase. Let us analyze its time and space complexities. Due to the properties of Karp-Rabin fingerprints, line 2 can be implemented in $\mathcal{O}(1)$ time. Also, the loops in lines 3 and 5 takes extra $\mathcal{O}(1)$ time even if the respective collections are empty. Apart from these, every operation can be assigned to a request, each of them taking $\mathcal{O}(1)$ (lines 3 and 5-6) or $\mathcal{O}(\log |\mathcal{Q}|)$ (lines 4 and 8) time. To bound $|\mathcal{Q}|$, we need to look at the maximum number of pending requests.

Fact 8. *For any pattern P_j there are $\mathcal{O}(1+\frac{n}{\ell})$ requests and at any time at most one of them is pending.*

Proof. Note that there is a one-to-one correspondence between requests concerning P_j and the occurrences of α_j in T. The distance between two such occurrences must be at least $\frac{1}{3}\ell$, because otherwise the period of α_j would be at most $\frac{1}{3}\ell$, thus making α_j highly periodic. This yields the $\mathcal{O}(1+\frac{n}{\ell})$ upper bound on the total number of requests. Additionally, any request is pending for at most $|P_j| - \ell < \frac{1}{3}\ell$ iterations of the main **for** loop. Thus, the request corresponding to an occurrence of α_j is already processed before the next occurrence appears. □

Consequently, the scanning phase uses $\mathcal{O}(s)$ space and takes $\mathcal{O}(n + s(1 + \frac{n}{\ell}) \log s)$ time. Adding the complexity of the initial preprocessing, we get the bounds from the statement of Lemma 7.

4.2 Highly Periodic Patterns

Lemma 9. *Let ℓ be an arbitrary integer. Given a text T of length n and a collection of highly periodic patterns $P_1, \ldots, P_s$ such that for every j we have*

$\ell \le |P_j| < \frac{4}{3}\ell$, *we can compute the leftmost occurrence of every* P_j *in* T *using* $\mathcal{O}(n + s(1 + \frac{n}{\ell})\log s + s\ell)$ *total time and* $\mathcal{O}(s)$ *space.*

The solution is basically the same as in the proof of Lemma 7, except that the algorithm ignores certain *shiftable* occurrences. An occurrence of x at position i of T is called *shiftable* if there is another occurrence of x at position $i - \mathrm{per}(x)$. The remaining occurrences are called *non-shiftable*. Notice that the leftmost occurrence is always non-shiftable, so indeed we can safely ignore some of the shiftable occurrences of the patterns. Because $2\,\mathrm{per}(P_j) \le \frac{2}{3}|P_j| \le \frac{8}{9}\ell < \ell$, the following fact (proved in Appendix B) implies that if an occurrence of P_j is non-shiftable, then the occurrence of α_j at the same position is also non-shiftable.

Fact 10. *Let* y *be a prefix of* x *such that* $|y| \ge 2\,\mathrm{per}(x)$. *Suppose* x *has a non-shiftable occurrence at position* i *in* w. *Then, the occurrence of* y *at position* i *is also non-shiftable.*

Consequently, we may generate requests only for the non-shiftable occurrences of α_j. In other words, if an occurrence of α_j is shiftable, we do not create the requests and proceed immediately to line 5. To detect and ignore such shiftable occurrences, we maintain the position of the last occurrence of every α_j. However, if there are multiple patterns sharing the same prefix $\alpha_{j_1} = \ldots = \alpha_{j_k}$, we need to be careful so that the time to detect a shiftable occurrence is $\mathcal{O}(1)$ rather than $\mathcal{O}(k)$. To this end, we build another deterministic dictionary, which stores for each $\Phi(\alpha_j)$ a pointer to the variable where we maintain the position of the previously encountered occurrence of α_j. The variable is shared by all patterns with the same prefix α_j.

It remains to analyze the complexity of the modified algorithm. First, we need to bound the number of non-shiftable occurrences of a single α_j. Assume that there is a non-shiftable occurrence α_j at positions $i' < i$ such that $i' \ge i - \frac{1}{2}\ell$. Then $i - i' \le \frac{1}{2}\ell$ is a period of $T[i'..i + \ell - 1]$. By the periodicity lemma, $\mathrm{per}(\alpha_j)$ divides $i - i'$, and therefore α_j occurs at position $i' - \mathrm{per}(\alpha_j)$, which contradicts the assumption that the occurrence at position i' is non-shiftable. Consequently, the non-shiftable occurrences of every α_j are at least $\frac{1}{2}\ell$ characters apart, and the total number of requests and the maximum number of pending requests can be bounded by $\mathcal{O}(s(1+\frac{n}{\ell}))$ and $\mathcal{O}(s)$, respectively, as in the proof of Lemma 7. Taking into the account the time and space to maintain the additional components, which are $\mathcal{O}(n + s\log s)$ and $\mathcal{O}(s)$, respectively, the final bounds remain the same.

4.3 Summary

Theorem 11. *Given a text* T *of length* n *and patterns* $P_1, \ldots, P_s$ *of total length* m, *using* $\mathcal{O}(n\log n + m + s\frac{n}{\ell}\log s)$ *total time and* $\mathcal{O}(s)$ *space we can compute the leftmost occurrences in* T *of all patterns* P_j *of length at least* ℓ.

Proof. The algorithm distributes the patterns into $\mathcal{O}(\log n)$ groups according to their lengths, and then into three classes according to their repetitiveness,

which takes using $\mathcal{O}(m)$ time and $\mathcal{O}(s)$ space in total. Then, it applies either Lemma 7 or Lemma 9 on every class. It remains to show that the running times of all those calls sum up to the claimed bound. Each of them can be seen as $\mathcal{O}(n)$ plus $\mathcal{O}(|P_j| + (1 + \frac{n}{|P_j|}) \log s)$ per every pattern P_j. Because $\ell \leq |P_j| \leq n$ and there are $\mathcal{O}(\log n)$ groups, this sums up to $\mathcal{O}(n \log n + m + s\frac{n}{\ell} \log s) = \mathcal{O}(n \log n + m + s\frac{n}{\ell} \log s)$. □

Using Thm. 4 for all patterns of length at most $\min(n, s)$, and (if $s \leq n$) Thm. 11 for patterns of length at least s, we obtain our main theorem.

Theorem 12. *Given a text T of length n and patterns $P_1, \ldots, P_s$ total length m, we can compute the leftmost occurrence of every P_i in T using $\mathcal{O}(n \log n + m)$ total time and $\mathcal{O}(s)$ space.*

5 Approximating LZ77 in Small Space

A non-empty fragment $T[i..j]$ is called a *previous fragment* if the corresponding subword occurs in T at a position $i' < i$. A *phrase* is either a previous fragment or a single letter not occurring before in T. The LZ77-factorization of a text $T[1..n]$ is a greedy factorization of T into z *phrases*, $T = f_1 f_2 \ldots f_z$, such that each f_i is as long as possible. To formalize the concept of LZ77-approximation, we first make the following definition.

Definition 13. *Let $w = g_1 g_2 \ldots g_a$ be a factorization of w into a phrases. We call it c-optimal if the fragment corresponding to the concatenation of any c consecutive phrases $g_i \ldots g_{i+c-1}$ is not a previous fragment.*

A c-optimal factorization approximates the LZ77-factorization in the number of factors, as the following observation states. However, the stronger property of c-optimality is itself a useful in certain situations.

Observation 14. *If $w = g_1 g_2 \ldots g_a$ is a c-optimal factorization of w into a phrases, and the LZ77-factorization of w consists of z phrases, then $a \leq c \cdot z$.*

5.1 2-Approximation Algorithm

Outline. Our algorithm is divided into 3 phases, each of which refines the factorization from the previous phase:

Phase 1. Create an $\mathcal{O}(\log n)$-optimal factorization of $T[1..n]$, stored implicitly as $\mathcal{O}(z)$ *chains* consisting of $\mathcal{O}(\log n)$ phrases each.

Phase 2. Try to merge phrases within the chains to produce an $\mathcal{O}(1)$-optimal factorization.

Phase 3. Try to merge adjacent factors as long as possible to produce the final 2-optimal factorization.

Every phase takes $\mathcal{O}(n \log n)$ time and uses $\mathcal{O}(z)$ working space. In the end, we get a 2-approximation of the LZ77-factorization. Phases 1 and 2 use the very simple multiple pattern matching algorithm for patterns of equal lengths developed in Thm. 1, while Phase 3 requires the general multiple pattern matching algorithm obtained in Thm. 12.

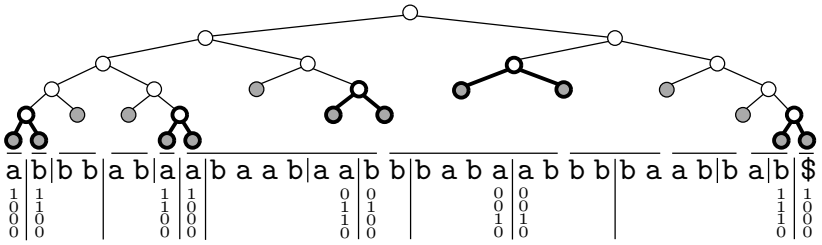

Fig. 1. An illustration of Phase 1 of the algorithm, with the "cherries" depicted in thicker lines. The horizontal lines represent the LZ77-factorization and the vertical lines depict factors induced by the tree. Longer separators are drawn between chains, whose lengths are written in binary with the least significant bits on top.

Phase 1. To construct the $\mathcal{O}(\log n)$-optimal factorization, we imagine creating a binary tree on top the text T of length $n = 2^k$ – see also Fig. 1 (we implicitly pad w with sufficiently many \$'s to make its length a power of 2). The algorithm works in $\log n$ rounds, and the i-th round works on level i of the tree, starting at $i = 1$ (the children of the root). On level i, the tree divides T into 2^i blocks of size $n/2^i$; the aim is to identify previous fragments among these blocks and declare them as phrases. (In the beginning, no phrases exist, so all blocks are unfactored.) To find out if a block is a previous fragment, we use Thm. 1 and test whether the leftmost occurrence of the corresponding subword is the block itself. The exploration of the tree is naturally terminated at the nodes corresponding to the previous fragments (or single letters not occurring before), forming the leaves of a (conceptual) binary tree. A pair of leaves sharing the same parent is called a *cherry*. The block corresponding to the common parent is *induced* by the cherry. To analyze the algorithm, we make the following observation:

Fact 15. *A block induced by a cherry is never a previous fragment. Therefore, the number of cherries is* $\mathcal{O}(z)$.

Proof. The former part of the claim follows from construction. To prove the latter, observe that the blocks induced by different cherries are disjoint and hence each cherry can be assigned a unique LZ77-factor ending within the block. □

Consequently, while processing level i of the tree, we can afford storing all cherries generated so far on a sorted linked list $\mathcal{L}$. The remaining already generated phrases are not explicitly stored. In addition, we also store a sorted linked list $\mathcal{L}_i$ of all still *unfactored* nodes on the current level i (those for which the corresponding blocks are tested as previous fragments). Their number is bounded by z (because there is a cherry below every node on the list), so the total space is $\mathcal{O}(z)$. Maintaining both lists sorted is easily accomplished by scanning them in parallel with each scan of T, and inserting new cherries/unfactored nodes at their correct places. Furthermore, in the i-th round we apply Thm. 1 to at most 2^i patterns of length $n/2^i$, so the total time is $\sum_{i=1}^{\log n} \mathcal{O}(n + 2^i \log(2^i)) = \mathcal{O}(n \log n)$.

Next, we analyze the structure of the resulting factorization. Let $h_{x-1}h_x$ and $h_y h_{y+1}$ be the two consecutive cherries. The phrases $h_{x+1} \dots h_{y-1}$ correspond to

the right siblings of the ancestors of h_x and to the left siblings of the ancestors of h_y (no further than to the lowest common ancestor of h_x and h_y). This naturally partitions $h_x h_{x+1} \dots h_{y-1} h_y$ into two parts, called an *increasing chain* and a *decreasing chain* to depict the behaviour of phrase lengths within each part. Observe that these lengths are powers of two, so the structure of a chain of either type is determined by the total length of its phrases, which can be interpreted as a bitvector with bit i' set to 1 if there is a phrase of length $2^{i'}$ in the chain. Those bitvectors can be created while traversing the tree level by level, passing the partially created bitvectors down to the next level $\mathcal{L}_{i+1}$ until finally storing them at the cherries in $\mathcal{L}$.

At the end we obtain a sequence of chains of alternating types, see Fig. 1. Since the structure of each chain follows from its length, we store the sequence of chains rather the actual factorization which could take $\omega(z)$ space. By Fact 15, our representation uses $\mathcal{O}(z)$ space and the last phrase of a decreasing chain concatenated with the first phrase of the consecutive increasing chain never form a previous fragment, which yields $\mathcal{O}(\log n)$-optimality.

Phase 2. In this phase we merge phrases within the chains. We describe how to process increasing chains; the decreasing are handled, mutatis mutandis, analogously. We partition the phrases within a chain into groups which are previous fragments but so that three consecutive groups never form previous fragments.

Suppose the phrases in the chain are $h_\ell \dots h_r$. For each chain we maintain an *active* group g_j, initially consisting of h_ℓ, and scan the remaining phrases in the left-to-right order. We either append a phrase h_i to the active group g_j, or we output g_j and make $g_{j+1} = h_i$ the new active group. We decide based on whether the fragment of length $2|h_i|$ starting at the same position as g_j is a previous fragment. Having processed the whole chain, we also output the last active group. Since the lengths of phrases form an increasing sequence of powers of two, we have $|g_j| \le |h_\ell \dots h_{i-1}| < |h_i|$, so $2|h_i| > |g_j h_i|$, and whenever we append h_i to g_j, we are guaranteed that they together form a previous fragment. Moreover, if we output a group g_j while processing h_i and there are further groups g_{j+1} and g_{j+2}, we know that $|g_j g_{j+1} g_{j+2}| > |g_{j+1} g_{j+2}| \ge |h_i h_{i+1}| > 2|h_i|$. Thus, our test assures that $g_j g_{j+1} g_{j+2}$ does not form a previous fragment. Consequently, the factorization is correct and satisfies the claimed optimality condition.

The procedure described above is executed in parallel for all chains, each of which only maintains the length of its active group. In the i-th round only chains containing a phrase of length 2^i participate (we use bit operations to verify which chains have length containing 2^i in the binary expansion). These chains provide fragments of length 2^{i+1} and Thm. 1 is applied to decide which of them are previous fragments. The chains modify their active groups based on the answers; some of them may output their old active groups. These groups form phrases of the output factorization, so the space required to store them is going to be amortized with the size of this factorization. As far as the running time is concerned, we observe that in the i-th round no more than $\min(z, \frac{n}{2^i})$ chains participate. Thus, the total running time is $\sum_{i=1}^{\log n} \mathcal{O}(n + \frac{n}{2^i} \log \frac{n}{2^i}) = \mathcal{O}(n \log n)$.

To bound the overall approximation guarantee, suppose there are 5 consecutive output phrases forming a previous fragment. By Fact 15, these fragments cannot contain a block induced by any cherry. Thus, the phrases are contained within two chains. However, no 3 consecutive phrases obtained from a single chain form a previous fragment. This contradiction proves 5-optimality.

Phase 3. The following lemma achieves the final 2-approximation:

Lemma 16. *Given a c-optimal factorization, one can compute a 2-optimal factorization using $\mathcal{O}(c \cdot n \log n)$ time and $\mathcal{O}(c \cdot z)$ space.*

Proof. The procedure consists of c iterations. In every iteration we first detect previous fragments corresponding to concatenations of two adjacent phrases. The total length of the patterns is up to $2n$, so this takes $\mathcal{O}(n \log n + m) = \mathcal{O}(n \log n)$ time and $\mathcal{O}(c \cdot z)$ space using Thm. 12. Next, we scan through the factorization and merge every phrase g_i with the preceding phrase g_{i-1} if $g_{i-1}g_i$ is a previous fragment and g_{i-1} has not been just merged with its predecessor.

We shall prove that the resulting factorization is 2-optimal. Consider a pair of adjacent phrases $g_{i-1}g_i$ in the final factorization and let j be the starting position of g_i. Suppose $g_{i-1}g_i$ is a previous fragment. Our algorithm performs merges only, so the phrase ending at position $j-1$ concatenated with the phrase starting at position j formed a previous fragment at every iteration. The only reason that these factors were not merged could be another merge of the former factor. Consequently, the factor ending at position $j-1$ took part in a merge at every iteration, i.e., g_{i-1} is a concatenation of at least c phrases of the input factorization. However, all the phrases created by the algorithm form previous fragments, which contradicts the c-optimality of the input factorization. □

5.2 Approximation Scheme

Lemma 17. *Given a text T of length n and a set of distinguished positions $i_1 < i_2 < \ldots i_s$ inside, using $\mathcal{O}(n \log^2 n)$ total time and $\mathcal{O}(s)$ space we can find for each j the longest previous fragment starting at i_j and fully contained inside $T[i_j..i_{j+1} - 1]$.*

Proof. For every j, we use binary search to compute the length ℓ_j of the resulting previous fragment. All binary searches are performed in parallel. That is, we proceed in $\mathcal{O}(\log n)$ phases, and in every phase we check, for every j, if $T[i_j..i_j + \ell_j - 1]$ is a previous fragment using Thm. 12, and then update every ℓ_j accordingly. The space complexity is clearly $\mathcal{O}(s)$ and, because the considered fragments are disjoint, the running time is $\mathcal{O}(n \log n)$ per phase. □

The starting point is a 2-optimal factorization into a phrases, which can be found in $\mathcal{O}(n \log n)$ time using the previous method. The text is partitioned into $\frac{\varepsilon}{2}a$ blocks corresponding to $\frac{2}{\varepsilon}$ consecutive phrases. Every block is then greedily factorized into phrases. The factorization is implemented using Lemma 17 to compute the longest previous fragments in parallel for all blocks. It requires

$\mathcal{O}(\frac{1}{\varepsilon} n \log n)$ time in total since every block can be surely factorized into $\frac{2}{\varepsilon}$ phrases by definition and the greedy factorization is optimal. To bound the approximation guarantee, observe that every phrase inside the block, except possibly for the last one, contains an endpoint of a phrase in the LZ77-factorization. Consequently, the total number of phrases is at most $z + \frac{\varepsilon}{2} a \leq (1+\varepsilon)z$.

Theorem 18. *Given a text T of length n whose LZ77-factorization consists of z phrases, we can factorize T into at most $2z$ phrases using $\mathcal{O}(n \log n)$ time and $\mathcal{O}(z)$ space. Moreover, for any positive $\varepsilon < 1$ in $\mathcal{O}(\varepsilon^{-1} n \log^2 n)$ time and $\mathcal{O}(z)$ space we can compute a factorization into no more than $(1+\varepsilon)z$ phrases.*

Acknowledgments. The authors would like to thank the participants of the Stringmasters 2015 workshop in Warsaw, where this work was initiated. We are particularly grateful to Marius Dumitran, Artur Jeż, and Patrick K. Nicholson.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Breslauer, D., Grossi, R., Mignosi, F.: Simple real-time constant-space string matching. Theor. Comput. Sci. 483, 2–9 (2013)
3. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. CUP (2007)
4. Crochemore, M., Perrin, D.: Two-way string matching. J. ACM 38(3), 651–675 (1991)
5. Fischer, J., I, T., Köppl, D.: Lempel Ziv Computation In Small Space (LZ-CISS) arXiv:1504.02605 (2015)
6. Galil, Z., Seiferas, J.I.: Time-space-optimal string matching. J. Comput. Syst. Sci. 26(3), 280–294 (1983)
7. Gum, B., Lipton, R.J.: Cheaper by the dozen: Batched algorithms. In: Kumar, V., Grossman, R.L. (eds.) Proc. SDM. pp. 1–11. SIAM (2001)
8. Hon, W., Ku, T., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed dictionary matching. Theor. Comput. Sci. 475, 113–119 (2013)
9. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lightweight Lempel-Ziv parsing. In: Proc. SEA. LNCS, vol. 7933, pp. 139–150. Springer (2013)
10. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260 (1987)
11. Kociumaka, T., Starikovskaya, T.A., Vildhøj, H.W.: Sublinear space algorithms for the longest common substring problem. In: Proc. ESA. LNCS, vol. 8737, pp. 605–617. Springer (2014)
12. Lohrey, M.: Algorithmics on SLP-compressed strings: A survey. Groups Complexity Cryptology 4(2), 241–299 (2012)
13. Ružić, M.: Constructing efficient dictionaries in close to sorting time. In: Proc. of 35th ICALP. vol. 5125, pp. 84–95 (2008)
14. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
15. Wikipedia: Rabin–Karp algorithm — Wikipedia, The Free Encyclopedia (2015), `http://en.wikipedia.org/w/index.php?title=Rabin-Karp_algorithm&oldid=646650598`, [online; accessed 22-April-2015]
16. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory 23(3), 337–343 (1977)

A Proof of Lemma 3

Lemma 3. *One can lexicographically sort strings $P_1, \ldots, P_s$ of total length m in $\mathcal{O}(m + \sigma^\varepsilon)$ time using $\mathcal{O}(s)$ space, for any $\varepsilon > 0$.*

Proof. We separately sort the $\sqrt{m} + \sigma^{\varepsilon/2}$ longest strings and all the remaining strings, and then merge both sorted lists.

Long strings are sorted using insertion sort. If the longest common prefixes between adjacent (in the sorted order) strings are computed and stored, inserting P_j can be done $\mathcal{O}(j+|P_j|)$ time. In more detail, let $S_1, S_2, \ldots, S_{j-1}$ be the sorted list of already processed strings. We start with $k := 1$ and increase k by one as long as S_k is lexicographically smaller than P_j while maintaining the longest common prefix between S_k and P_j, denoted ℓ. After increasing k by one, we update ℓ using the longest common prefix between S_{k-1} and S_k, denoted ℓ', as follows. If $\ell' > \ell$, we keep ℓ unchanged. If $\ell' = \ell$, we try to iteratively increase ℓ by one as long as possible. In both cases, the new value of ℓ allows us to lexicographically compare S_k and P_j in constant time. Finally, $\ell' < \ell$ guarantees that $P_j < S_k$ and we may terminate the procedure. Sorting the $\sqrt{m} + \sigma^{\varepsilon/2}$ longest strings using this approach takes $\mathcal{O}(m + (\sqrt{m} + \sigma^{\varepsilon/2})^2) = \mathcal{O}(m + \sigma^\varepsilon)$ time.

The remaining strings are of length at most $\sqrt{m}$ each, and if there are any, then $s \geq \sigma^{\varepsilon/2}$. We sort these strings by iteratively applying radix sort, treating each symbol from Σ as a sequence of $\frac{2}{\varepsilon}$ symbols from $\{0, \ldots, \sigma^{\varepsilon/2} - 1\}$. Then a single radix sort takes time and space proportional to the number of strings involved plus the alphabet size, which is $\mathcal{O}(s + \sigma^{\varepsilon/2}) = \mathcal{O}(s)$. Furthermore, because the numbers of strings involved in the subsequent radix sorts sum up to m, the total time complexity is $\mathcal{O}(m + \sigma^{\varepsilon/2}\sqrt{m}) = \mathcal{O}(m + \sigma^\varepsilon)$.

Finally, the merging takes linear time in the sum of the lengths of all the involved strings, so the total complexity is as claimed. □

B Missing Proofs of Sect. 4

Let us recall some basic facts from combinatorics of strings; see [3]. We say that word w is *primitive* if $\mathrm{per}(w)$ is not a proper divisor of $|w|$. Note that the shortest period $w[1..\,\mathrm{per}(w)]$ is always primitive.

Fact 19. *Suppose x and y are a prefix and a suffix of a word w, respectively. If $|x| + |y| \geq |w| + p$ and p is a period of both x and y, then p is a period of w.*

Proof. We need to prove that $w[i] = w[i + p]$ for all $i = 1, 2, \ldots, |w| - p$. If $i + p \leq |x|$ this follows from p being a period of x, and if $i \geq |w| - |y| + 1$ from p being a period of y. Because $|x| + |y| \geq |w| + p$, these two cases cover all possible values of i. □

Lemma 5. *Suppose x and y are a prefix and a suffix of a word w such that $|w| \leq \frac{2}{3}(|x| + |y|)$. If w is not highly periodic, then x or y is not highly periodic.*

Proof. Suppose that x and y are both highly periodic. Let z be the fragment of w common to x and y. Note that both $\mathrm{per}(x) \leq \frac{1}{3}|x|$ and $\mathrm{per}(y) \leq \frac{1}{3}|y|$ are periods of z. We have $\mathrm{per}(x)+\mathrm{per}(y) \leq \frac{1}{3}(|x|+|y|)$ and $|z| = |x|+|y|-|w| \geq \frac{1}{3}(|x|+|y|)$. Thus, the periodicity lemma implies that $d = \gcd(\mathrm{per}(x), \mathrm{per}(y))$ is a period of z. Consequently $\mathrm{per}(x) = \mathrm{per}(y)$ because shortest periods are primitive. Moreover, Fact 19 yields that d is also a period of the whole w, so we obtain a contradiction with the assumption that w is not highly periodic. □

Lemma 6. *Given a read-only string w one can decide in $\mathcal{O}(|w|)$ time and constant space if w is periodic and if so, compute* $\mathrm{per}(w)$.

Proof. Let v be the prefix of w of length $\lceil \frac{1}{2}|w| \rceil$ and p be the starting position of the second occurrence of v in w, if any. The position p can be found in linear time by a constant-space pattern matching algorithm.

We claim that if $\mathrm{per}(w) \leq \frac{1}{2}|w|$, then $\mathrm{per}(w) = p - 1$. Observe first that in this case v occurs at a position $per(w) + 1$. Hence, $\mathrm{per}(w) \geq p - 1$. Moreover $p-1$ is a period of $w[..|v|+p-1]$ along with $\mathrm{per}(w)$. By the periodicity lemma, $\mathrm{per}(w) \leq \frac{1}{2}|w| \leq |v|$ implies that $\gcd(p-1, \mathrm{per}(w))$ is also a period of that prefix. Thus $\mathrm{per}(w) > p - 1$ would contradict the primitivity of $w[1..\mathrm{per}(w)]$.

The algorithm computes the position p. If it exists, it uses letter-by-letter comparison to determine whether $w[..p-1]$ is a period of w. If so, by the discussion above $\mathrm{per}(w) = p - 1$ and the algorithm returns this value. Otherwise, $2\,\mathrm{per}(w) > |w|$, i.e., w is not periodic. The algorithm runs in linear time and uses constant space. □

Fact 10. *Let y be a prefix of x such that $|y| \geq 2\,\mathrm{per}(x)$. Suppose x has a non-shiftable occurrence at position i in w. Then, the occurrence of y at position i is also non-shiftable.*

Proof. Note that $\mathrm{per}(y) + \mathrm{per}(x) \leq |y|$ so the periodicity lemma implies that $\mathrm{per}(y) = \mathrm{per}(x)$.

Let $x = \rho^k\rho'$ where ρ is the shortest period of x. Suppose that the occurrence of y at position i is shiftable, meaning that y occurs at position $i - \mathrm{per}(x)$. Since $|y| \geq \mathrm{per}(x)$, y occurring at position $i - \mathrm{per}(x)$ implies that ρ occurs at the same position. Thus $w[i - \mathrm{per}(x)..i + |x| - 1] = \rho^{k+1}\rho'$. But then x clearly occurs at position $i - \mathrm{per}(x)$, which contradicts the assumption that its occurrence at position i is non-shiftable. □

C Las Vegas Algorithms

By verifying the found occurrences, it is not difficult to modify our multiple-pattern matching algorithm so that it always returns the correct answers. The running time becomes then $\mathcal{O}(n \log n + m)$ with high probability, assuming that we are interested in finding just the leftmost occurrence of every pattern. This in turn immediately implies that we can approximate the LZ77-factorization so that the answer is always correct and the bounds on the running time hold with high probability.

First, we explain how to modify the solution for short patterns described in Thm. 4. For every pattern P_j such that no occurrence of P_j has been found so far, we naively verify that $T_i[\ell..r] = P_j$ in $\mathcal{O}(|P_j|)$ time. If a false-positive is detected, we reset the whole computation with a fresh random value of x, chosen independently from the previous ones. Because with high probability there are no false-positives, and the total verification time is $\mathcal{O}(\sum_j |P_j|)$, the modified algorithm always returns correct answers and, with high probability, works in $\mathcal{O}(n \log \ell + s\frac{n}{\ell} + m)$ time.

The solution for patterns without long highly periodic prefix described in Lemma 7 is modified similarly: if no occurrence of P_j has been found so far, before reporting an occurrence of P_j at $i + \ell - |P_j]$ we naively verify that $T[i + \ell - |P_j|..i + \ell - 1] = P_j$, and possibly reset the whole computation. Similarly, we verify the rightmost occurrence of every pattern (once the whole text is processed).

For highly periodic patterns, the situation is more complicated. The algorithm from Lemma 9 assumes that we are correctly detecting all occurrences of every α_j so that we can filter out the shiftable ones. Verifying these occurrences naively might take too much time, because it is not enough check just one occurrence of every α_j.

Recall that $\mathrm{per}(\alpha_j) \leq \frac{1}{3}\ell$. If the previous occurrence of α_j was at position $i' \geq i - \frac{1}{2}\ell$, we will check if $\mathrm{per}(\alpha_j)$ is a period of $T[i'..i + \ell - 1]$. If so, either both occurrences (at position i' and at position i) are false-positives, or none of them is, and the occurrence at position i' can be ignored. Otherwise, at least one occurrence is surely false-positive, and we reset the computation. To check if $\mathrm{per}(\alpha_j)$ is a period of $T[i'..i + \ell - 1]$, we partition T into overlapping blocks $T_1 = [1..\frac{2}{3}\ell], T_2 = [\frac{1}{3}\ell + 1..\frac{4}{3}\ell], \ldots$. Let $T_t = T[(t-1)\frac{1}{3}\ell + 1..(t+1)\frac{1}{3}\ell]$ be the rightmost such block fully inside $T[1..i + \ell - 1]$. We calculate the period of T_t using Lemma 6 in $\mathcal{O}(\ell)$ time and $\mathcal{O}(1)$ space, and then calculate how far the period extends to the left and to the right, terminating if it extends very far. Formally, we calculate the largest $r < (t+2)\frac{1}{3}\ell$ and the smallest $\ell > (t-2)\frac{1}{3}\ell - \frac{1}{2}\ell$ such that $\mathrm{per}(T_t)$ is a period of $T[\ell..r]$. This takes $\mathcal{O}(\ell)$ time for every t summing up to $\mathcal{O}(n)$ total time. Then, to check if $\mathrm{per}(\alpha_j)$ is a period of $T[i'..i + \ell - 1]$ we check if it divides the period of $\mathrm{per}(T_t)$ and furthermore $r \geq i + \ell - 1$ and $\ell \leq i'$. Finally, before reporting the leftmost occurrence of a pattern P_j, we naively verify it as explained before.

For all classes of patterns, the results are now always correct, and the bounds on the running times becomes the same, except that now they hold with high probability. Therefore, repeating the reasoning from Thm. 12, we indeed construct a Las Vegas algorithm.