A *really* simple approximation of smallest grammar

Artur Jeż

[a] *Institute of Computer Science, University of Wrocław, 50-383 Wrocław, Poland.*

Abstract

In this paper we present a *really* simple linear-time algorithm constructing a context-free grammar of size $4g \log_{3/2}(N/g)$ for the input string, where N is the size of the input string and g the size of the optimal grammar generating this string. The algorithm works for arbitrary size alphabets, but the running time is linear assuming that the alphabet Σ of the input string can be identified with numbers from $\{1, \ldots, N^c\}$ for some constant c. Algorithms with such an approximation guarantee and running time are known, however all of them were non-trivial and their analyses were involved. The here presented algorithm computes the LZ77 factorisation and transforms it in phases to a grammar. In each phase it maintains an LZ77-like factorisation of the word with at most ℓ factors as well as additional $\mathcal{O}(\ell)$ letters, where ℓ was the size of the original LZ77 factorisation. In one phase in a greedy way (by a left-to-right sweep and a help of the factorisation) we choose a set of pairs of consecutive letters to be replaced with new symbols, i.e. nonterminals of the constructed grammar. We choose at least 2/3 of the letters in the word and there are $\mathcal{O}(\ell)$ many different pairs among them. Hence there are $\mathcal{O}(\log N)$ phases, each of them introduces $\mathcal{O}(\ell)$ nonterminals to a grammar. A more precise analysis yields a bound $\ell + 4\ell \log(N/\ell)$. As $\ell \leq g$, this yields the desired bound $g + 4g \log(N/g)$.

Keywords: grammar-based compression, construction of the smallest grammar, SLP, compression, LZ77
2000 MSC: 68W01, 68W05, 68W25, 68W40, 68Q42, 68Q45

1. Introduction

Grammar based compression

In the grammar-based compression text is represented by a context-free grammar (CFG) generating exactly one string. Such an approach was first considered by Rubin [24], though he did not mention CFGs explicitly. In general, the idea behind this approach is that a CFG can compactly represent the structure of the text, even if this structure is not apparent. Furthermore, the natural hierarchical definition of the context-free grammars makes such a representation suitable for algorithms, in which case the string operations can be performed on the compressed representation, without the need of the explicit decompression [6, 11, 15, 23, 7, 2].

The grammar-based compression was introduced with practical purposes in mind and the paradigm was used in several implementations [17, 16, 21]: intuitively, in many cases large data have relatively simple inductive definition, which results in a grammar representation of small size. On the other hand, it turned out that grammar compression is useful in more theoretical considerations: for instance, unveiling the repetitive structure of the text can be used to estimate

text similarity [18, 4, 20]. Another group of applications was made possible due to the aforementioned algorithms that operate directly on the compressed representations: In this approach "regular" data is compressed and the processed in this compressed form, often yielding more efficient algorithms. A recent survey by Lohrey[19] gives a comprehensive description of several areas of theoretical computer science where such an approach was successfully applied, such as word equations, computations in groups, computational topology and others.

The main drawback of the grammar-based compression is that producing the smallest CFG for a text is *intractable*: given a string w and number k it is NP-hard to decide whether there exist a CFG of size k that generates w [27]. Furthermore, the size of the smallest grammar for the input string cannot be approximated within some small constant factor [2].

Previous approximation algorithms

The first two algorithms with an approximation ratio $\mathcal{O}(\log(N/g))$ were developed simultaneously by Rytter [25] and Charikar et al. [2]. They followed a similar approach, we first present Rytter's approach as it is a bit easier to explain.

Rytter's algorithm [25] applies the LZ77 compression to the input string and then transforms the obtained LZ77 representation to an $\mathcal{O}(\ell \log(N/\ell))$ size grammar, where ℓ is the size of the LZ77 representation. It is easy to show that $\ell \leq g$ and as $f(x) = x \log(N/x)$ is increasing, the bound $\mathcal{O}(g \log(N/g))$ on the size of the grammar follows (and so a bound $\mathcal{O}(\log(N/g))$ on the approximation ratio). The crucial part of the construction is the requirement that the derivation tree of the intermediate constructed grammar satisfies the AVL condition. While enforcing this requirement is in fact easier than in the case of the AVL search trees (as the internal nodes do not store any data), it remains involved and non-trivial. Note that the final grammar for the input text is also AVL-balanced, which makes is suitable for later processing.

Charikar et al. [2] followed more or less the same path, with a different condition imposed on the grammar: it is required that the derivation tree is length-balanced, i.e. for a rule $X \to YZ$ the lengths of words generated by Y and Z are within a certain multiplicative constant factor from each other. For such trees efficient implementation of merging, splitting etc. operations were given (i.e. constructed from scratch) by the authors and so the same running time as in the case of the AVL grammars was obtained. Since all the operations are defined from scratch, the obtained algorithm is also quite involved and the analysis is even more non-trivial.

Sakamoto [26] proposed a different algorithm, based on RePair [17], which is one of the practically implemented and used algorithms for grammar-based compression. His algorithm iteratively replaces pairs of different letters and maximal repetitions of letters (a^ℓ is a *maximal repetition* if it cannot be extended by a to either side). A special pairing of the letters was devised, so that it is 'synchronising': if u has 2 disjoint occurrences in w, then those two occurrences can be represented as $u_1u'u_2$, where $|u_1|, |u_2| = \mathcal{O}(1)$, such that both occurrences of u' in w are paired and replaced in the same way. The analysis was based on considering the LZ77 representation of the text and proving that due to 'synchronisation' the factors of LZ77 are compressed very similarly as the text to which they refer. Constructing such a pairing is involved and the analysis non-trivial.

Recently, the author proposed another algorithm [10]. Similarly to the Sakamoto's algorithm it iteratively applied two local replacement rules (replacing pairs of different letters and replacing maximal repetitions of letters). Though the choice of pairs to be replaced was simpler, still the construction was involved. The main feature of the algorithm was its analysis based on the recompression technique, which allowed avoiding the connection of SLPs and LZ77 compression.

This made it possible to generalise this approach also to grammars generating trees [12]. On the downside, the analysis is quite complex.

Contribution of this paper

We present a very simple algorithm together with a straightforward and natural analysis. It chooses the pairs to be replaced in the word during a left-to-right sweep and additionally using the information given by an LZ77 factorisation. We require that any pair that is chosen to be replaced is either inside a factor of length at least 2 or consists of two factors of length 1 and that the factor of length at least 2 is paired in the same way as its definition. To this end we modify the LZ77 factorisation during the sweep. After the choice, the pairs are replaced and the new word inherits the factorisation from the original word. This procedure is repeated until a word of length 1 is obtained. This is indeed a grammar construction: when the pair ab is replaced by c we create a rule $c \to ab$. The approximation ratio of our algorithm is at most $1 + 4\log_{3/2}(N/g)$.

Note on computational model. The presented algorithm runs in linear time, assuming that we can compute the LZ77 factorisation in linear time. This can be done assuming that the letters of the input words can be sorted in linear time, which follows from a standard assumption that Σ can be identified with a continues subset of natural numbers of size $\mathcal{O}(N^c)$ for some constant c and the RadixSort can be performed on it. Note that such an assumption is needed for all currently known linear-time algorithms that attain the $\mathcal{O}(\log(N/g))$ approximation guarantee.

2. Notions

Sizes. By N we denote the size of the input word, by ℓ, g the sizes of the LZ77 factorisation and smallest grammar for the input string, both notions are defined in detail below.

LZ77 factorisation. An LZ77 factorisation (called simply *factorisation* in the rest of the paper) of a word w is a representation $w = f_1 f_2 \cdots f_\ell$, where each f_i is either a single letter (called *free letter* in the following) or $f_i = w[j\,..\,j + |f_i| - 1]$ for some $j \leq |f_1 \cdots f_{i-1}|$, in such a case f is called a *factor* and $w[j\,..\,j + |f_i| - 1]$ is called the *definition* of this factor. We do not assume that a factor has more than one letter though when we find such a factor we demote it to a free letter. The *size* of the LZ77 factorisation $f_1 f_2 \cdots f_\ell$ is ℓ. There are several simple and efficient linear-time algorithms for computing the smallest LZ77 factorisation of a word [1, 3, 5, 8, 13, 22, 9] and all of them rely on linear-time algorithm for computing the suffix array [14].

SLP. Straight Line Programme (*SLP*) is a CFG in the Chomsky normal form that generates a unique string. Without loss of generality we assume that nonterminals of an SLP are $X_1, \ldots, X_g$, each rule is either of the form $X_i \to a$ or $X_i \to X_j X_k$, where $j, k < i$. The *size* of the SLP is the number of its nonterminals (here: g).

The problem of finding smallest SLP generating the input word w is NP-hard [27] and the size of the smallest grammar for the input word cannot be approximated within some small constant factor [2]. On the other hand, several algorithms with an approximation ratio $\mathcal{O}(1 + \log(N/g))$, where g is the size of the smallest grammar generating w, are known [2, 25, 26, 10]. Most of those constructions use the inequality $\ell \leq g$ [25], where ℓ is the size of the smallest LZ77 factorisation for w. This bound is relatively easy to obtain: any SLP (of size k) defines a specific LZ77 factorisation (of size at most k), in particular, there is a factorisation of size g.

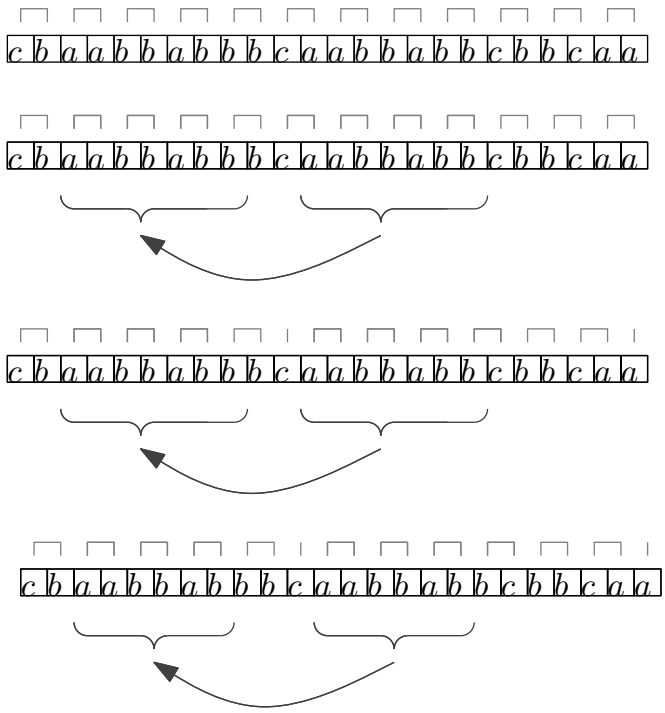

Figure 1: The pairings are presented over the words, one of the LZ77 factors is depicted below the word. On the top, the naive pairing is presented. On the second picture, the pairing is compared with the LZ77 factor as well as its definition; the factor and its definition are paired differently. On the third picture, we move the pairing so that it is consistent on the factor and its definition. This creates unpaired letters. On the bottom, we shorten the factor on the right, so that no pair is only partially within the factor.

3. Intuition

Pairing. Relaxing the Chomsky normal form, let us identify each nonterminal generating a single letter with this letter. Suppose that we already have an SLP for w. Consider the derivation tree for w and the nodes that have only leaves as children (they correspond to nonterminals that have only letters on the right-hand side). Such nodes define a *pairing* on w, in which a letter can be paired with one of the neighbouring letters (such pairing is of course a symmetric relation). Construction of the grammar can be naturally identified with iterative pairing: for a word w_i we find a pairing, replace pairs of letters with 'fresh' letters (different occurrences of a pair ab can be replaced with the same letter though this is not essential), obtaining w_{i+1} and continue the process until a word $w_{i'}$ has only one letter. The fresh letters from all pairings are the nonterminals of the constructed SLP and its size is twice the number of different introduced letters. Our algorithm will find one such pairing using the LZ77 factorisation of a word.

Creating a pairing. Suppose that we are given a word w and know its factorisation. We try the naive pairing: the first letter is paired with second, third with fourth and so on, see Fig. 1. If we now replace all pairs with new letters, we get a word that is 2 times shorter so $\log N$ such iterations give an SLP for w. However, in the worst case there are $|w|/2$ different pairs already in the first pairing and so we cannot give any better bound on the grammar size than $\mathcal{O}(N)$.

A better estimation uses the LZ77 factorisation. Let $w = f_1 f_2 \cdots f_\ell$ be the LZ77 factorisation and consider a factor f_i. It is equal to $w[j\,..\,j + |f_i| - 1]$ and so all pairs occurring in f_i already

occur in $w[j\,..\,j+|f_i|-1]$ unless the parity is different, i.e. j and $|f_1 \cdots f_{i-1}|+1$ are of different parity, see Fig. 1. We want to fix this: it seems a bad idea to change the pairing in $w[j\,..\,j+|f_i|-1]$, so we change it in f_i: it is enough to shift the pairing by one letter, i.e. leave the first letter of f_i unpaired and pair the rest as in $w[j+1\,..\,j+|f_i|-1]$; in particular, we remove the first letter from the factor and make it a free letter. This increases the size of the LZ77 factorisation, but the number of factors stays the same (i.e. only the number of free letters increases). Note that the last letter in the factor definition may be paired with the letter to the right, which may be impossible inside f_i, see Fig. 1. As a last observation note that since we alter each f_i, instead of creating a pairing at the beginning and modifying it we can create the pairing while scanning the word from left to right.

There ar some issues: after the pairing we want to replace the pairs with fresh letters. This may make some of the the factor definitions improper: when f_i is defined as $w[j\,..\,j+|f_i|-1]$ it might be that $w[j]$ is paired with letter to the left. To avoid this situation, we replace the factor f_i with $w[j]f_i$ (so we shorten f_i by the first letter) and change its definition to $w[j+1\,..\,j+|f_i|-1]$. Similar operation may be needed at the end of the factor, see Fig. 1. Again, this increases the size of the LZ77 factorisation, but the number of factors stays the same (i.e. only the number of free letters increases). So far we did not consider the pairing of free letters: we do it in a greedy way: we pair two neighbouring free letters, whenever this is possible.

Another issue is that so far our algorithm cannot process factors that have a definition one letter before the factor. However, it is easy to see that in such a case the whole factor f is of a form $a^{|f|}$, where a is the preceding letter. Thus we can replace af with aaf and set the definition of the new factor two letters to the left.

Using a pairing. When the appropriate pairing is created, we replace each pair with a new letter. If the pair is within a factor, we replace it with the same symbol as the corresponding pair in the definition of the factor. In this way only pairs that are formed from free letters may contribute a fresh letter. As a result we obtain a new word together with a factorisation in which there are ℓ factors.

Analysis. The analysis is based on the observation that a factor f_i is shortened by a constant fraction, so it takes part in $\log|f_i|$ phases and in each of them it introduces $\mathcal{O}(1)$ free letters. Hence the total number of free letters introduced to the word is $\mathcal{O}(\sum_{j=1}^{\ell}\log|f_i|) = \mathcal{O}(\ell\log(N/\ell))$ (which is shown in details later on). As creation of a rule decreases the number of free letters in the instance by at least 1, we obtain that this is also an upper bound on the size of the grammar.

4. The algorithm

Stored data. The word is represented as a table of letters. The table *start* stores the information about the beginnings of factors: $start[i] = j$ means that $w[i]$ is the first letter of a factor and $w[j]$ is the first letter of its definition; otherwise $start[i] = false$. A bitvector *end* stores the information about the ends of factors: $end[i]$ has value *true*/*false*, that tells whether $w[i]$ is the last letter of a factor.

When we replace the pairs with new letters, we reuse the same tables, overwriting from left to the right. Additionally, a table $newpos[i]$ stores the position of the letter corresponding to $w[i]$ in the new word; note that when i and $i+1$ are paired then they have the same corresponding position.

Pairing. We are going to devise a pairing with the following (a bit technical) properties:

(P1) there are no two consecutive letters that are both unpaired;

(P2) if the first (last) letter of a factor f is paired then the other letter in the pair is within the same factor;

(P3) if $f = w[i\,..\,i+|f|-1]$ has a definition $w[start[i]\,..\,start[i]+|f|-1]$ then letters in f and in $w[start[i]\,..\,start[i]+|f|-1]$ are paired in the same way.

The pairing is found incrementally by a left-to-right scan through w: we read w and when we are at letter i we make sure that the word $w[1\,..\,i]$ satisfies (P1–P3). To this end we not only devise the pairing but also modify the factorisation a bit (by replacing a factor f with af or by fb, where a is the first and b the last letter of f). If during the sweep some f is shortened so that $|f| = 1$ then we can demote it to a free letter.

The pairing is recorded in a table: $pair[i]$ can be set to *first*, *second* or *none*, meaning that $w[i]$ is the *first*, *second* in the pair or it is unpaired, respectively.

Splitting letters. We mainly modify the factors by *splitting letters* from the factors: intutitively we replace the factor by a free letter and the rest of the factor, the free letter can be the first or the last one of the original factor. To be more precise when we split $w[i]$ which is the first letter of a factor, we set it as a free letter. If it was a unique letter in this factor, then the factor is removed (both *start* and *end* flags are cleared), otherwise $w[i+1]$ should be a new a new beginning of a factor, unless it is the last letter, in which case it is also made a free letter and the whole fragment is removed. Splitting the last letter from a factor is symmetrically done. In both cases we view the rest of the factor (except the split letter) as a modified original factor, in particular, when no factor is left, we say that it was removed.

Algorithm 1 Split(i)

```
1:  if start[i] and end[i] then                          ▷ This is a unique letter of a factor
2:      start[i] ← end[i] ← false                        ▷ Remove this factor
3:  if start[i] then                                     ▷ If this is a first letter of a factor
4:      if end[i + 1] then                               ▷ If the remaining letter is the only in the factor
5:          start[i] ← end[i + 1] ← false                ▷ We remove the factor
6:      else                                             ▷ If there are more letters in a factor
7:          start[i + 1] ← start[i] + 1   ▷ We make the letter to the right the new first letter of a
    factor
8:          start[i] ← false
9:  if end[i] then                                       ▷ If this is a last letter of a factor
10:     if start[i − 1] then                             ▷ If the remaining letter is the only in the factor
11:         end[i] ← start[i − 1] ← false                ▷ We remove it
12:     else                                             ▷ If there are more letters in a factor
13:         end[i − 1] ← true   ▷ We make the letter to the left the new last letter of a factor
14:         end[i] ← false
```

Note that the code of Split can be made more effective: when we apply it we know, whether

we split the first, or last letter. But as the actions are symmetric, it is easier for us to simply say that we split the letter.

Preprocessing. It is easy to see that one cannot devise a pairing satisfying (P1–P3) when a factor has a definition one letter earlier, i.e. when $start[j] = j - 1$, as in this case the pairing of each letter in this factor should be the same. However, this can be fixed in a simple preprocessing: we sweep through w; if for a factor f_i beginning at $w[j]$ we have $start[j] = j - 1$ then we split $w[j]$ from f_i. Additionally, if the factor survived, we set the definition of $w[j + 1]$ to $j - 1$.

Algorithm 2 Preproc

```
1: for i ← 1 .. |w| do
2:     if start[i] = i − 1 then        ▷ The factor is a^k, its definition begins one position to the left
3:         Split(i)                                                          ▷ Shorten the factor
4:         if start[i + 1] then                                              ▷ If the factor survived
5:             start[i + 1] ← i − 1                ▷ Move its definition one position to the left
```

Creation of pairing. We read w from left to right, suppose that we are at position i.

Algorithm 3 Pairing

```
1: pair[1] ← none
2: i ← 2
3: while i ≤ |w| do
4:     if start[i] then                                          ▷ w[i] is the first element of a factor
5:         if pair[start[i]] = pair[i − 1] = none or pair[start[i]] = second then
6:             Split(i)          ▷ The pairing of the definition of factor is bad: Split this letter
7:         else                                                                  ▷ Good factor
8:             j ← start[i]                                      ▷ Factor's definition begins at j
9:             repeat                                ▷ Copy the pairing from the factor definition
10:                pair[i] ← pair[j]
11:                i ← i + 1, j ← j + 1
12:            until end[i − 1]
13:            i ← i − 1
14:            if pair[i] = first then                           ▷ Bad pairing of the last letter
15:                Split(i), clear pair[i]                ▷ Split the letter and clear its pairing
16:    if not start[i] then          ▷ w[i] a free letter, perhaps due to processing above
17:        if pair[i − 1] = none then                           ▷ If previous letter is not paired
18:            Split(i − 1)                                                  ▷ May be within a factor
19:            pair[i − 1] ← first, pair[i] ← second                                  ▷ Pair them
20:        else
21:            pair[i] ← none                                    ▷ Leave the letter unpaired
22:    i ← i + 1
```

If i is a first letter of a factor then we check whether it is properly paired: if $w[start[i]]$ is a *second* or both $w[i - 1]$ and $w[start[i]]$ are *none*, then we split this letter. Otherwise, we copy

the pairing from the whole factor's definition to the factor starting at i and ending at i'. We now ensure (P2) for the last letter ($w[i']$): if $pair[i] = first$ then we split $w[i]$.

Now, if $w[i]$ is a free letter (perhaps due to splitting in the earlier processing) then we check, whether the previous letter ($w[i-1]$)is not paired. If so, then we split $w[i-1]$ (it may be a last letter in a a factor) and pair $w[i-1]$ and $w[i]$.

This is all formalised in Algorithm 3.

Algorithm 4 PairReplacement

$i \leftarrow i' \leftarrow 1$ ▷ i' is the position corresponding to i in the new word
while $i \leq |w|$ **do**
  if ***not*** $start[i]$ **then** ▷ $w[i]$ a free letter
    if $pair[i] = none$ **then**
      $w[i'] \leftarrow w[i]$ ▷ We copy the unpaired letter
      $newpos[i] \leftarrow i'$
      $i \leftarrow i+1$, $i' \leftarrow i'+1$ ▷ We move by this letter to the right
    else
      $w[i'] \leftarrow$ fresh letter ▷ Paired free letters are replaced by a fresh letter
      $newpos[i] \leftarrow newpos[i+1] \leftarrow i'$
      $i \leftarrow i+2$, $i' \leftarrow i'+1$ ▷ We move to the right by the whole pair
  if $start[i]$ **then** ▷ $w[i]$ is the first element of a factor
    $start[i'] \leftarrow j' \leftarrow newpos[start[i]]$ ▷ Factor in new word begins at the position corresponding to the beginning of the current factor
    $start[i] \leftarrow false$ ▷ Clearing obsolete information
    repeat
      $w[i'] \leftarrow w[j']$ ▷ Copy the letter according to new factorisation
      $newpos[i] \leftarrow i'$ ▷ Position corresponding to i
      $i \leftarrow i+1$ ▷ Move i by 1
      if $pair[i] = first$ **then** ▷ If we replace a pair
        $newpos[i] \leftarrow i'$ ▷ The same corresponding position
        $i \leftarrow i+1$ ▷ Movew by another letter
      $i' \leftarrow i'+1$, $j' \leftarrow j'+1$
    until $end[i-1]$ ▷ We processed the whole factor
    $end[i'-1] \leftarrow true$ ▷ End in the new word
    $end[i-1] \leftarrow false$ ▷ Clearing obsolete information

Using the pairing. When the pairing is done, we read the word w again (from left to right) and replace the pairs by letters. We keep two indices: i, which is the pointer in the current word (pointing at the first unread letter) and i', which is a pointer in the new word, pointing at the first free position. Additionally, when reading i we store (in $newpos[i]$) the position of the corresponding letter in the new word, which is always i'.

If $w[i]$ is a *first* letter in a pair and this pair consists of two free letters, in the new word we add a fresh letter and move two letters to the right in w (as well as one position in the new word). If $w[i]$ is unpaired and a free letter then we simply copy this letter to the new word, increasing both i and i' by 1. If $w[i]$ is a first letter of a factor, we look at the definition of this factor, so at $start[i]$, and copy the corresponding fragment of the new word (the first position is given by

newpos[*start*[i]]), moving i and i' in parallel: i' is always incremented by 1, while i is moved by 2 when it reads a *first* letter of a pair and by 1 when it reads an unpaired letter. Also, we store the new beginning and end of the factor in the new word: for a factor beginning at i and ending at i' we set *start*[*newpos*[i]] = *newpos*[*start*[i]] and *end*[*newpos*[i']] = *true*. Details are given in Algorithm 4.

Algorithm 5 TtoG

```
compute LZ77 factorisation of w
while |w| > 1 do
    make the preprocessing using Preproc
    compute a pairing of w using Pairing
    replace the pairs using PairReplacement
output the constructed grammar
```

Algorithm. TtoG first computes the LZ77 factorisation and then iteratively applies Preproc, Pairing and PairReplacement, until a one-word letter is obtained.

5. Analysis

Since we modify the factorisation, we should ensure that it is properly defined: we say that a factorisation is *proper*, if for every factor $f = w[i\,..\,i+k]$ with *start*[i] = j we have $w[i\,..\,i+k] = w[j\,..\,j+k]$ and $j < i$. We mostly modify the factorisation by splitting letters, which indeed returns proper factorisations.

Lemma 1. Split *runs in constant time. If it is applied to a proper factorisation then it returns a proper factorisation. It increases the number of free letters by* 1 *or* 2*, in the latter case it also removes one factor from the factorisation.*

If a factor survives the splitting, then the distance between its first letter and the first letter of its definition is unchanged.

The proof is obvious.

We can now show the basic properties of the preprocessing.

Lemma 2. *After* Preproc *returns a proper factorisation. It introduces at most* 1 *free letter per factor or* 2*, when this factor is removed. It runs in linear time. After* Preproc *for each factor (beginning at* i*) we have* $i - pocz[i] \geq 2$ *and the equality holds if a factor was modified.*

Proof. Any change is triggered only when *start*[i] = $i-1$, in which case either the factors has at most two letters and so it is replaced with free letters by Split, or it has more letters and so $w[i\,..\,i+|f|-1] = w[i-1\,..\,i+|f|-2]$. This implies that $f = a^{|f|}$, where $a = w[i] = w[i-1]$. Thus after Split we are left with a proper factor (by Lemma 1 we split at most 1 letter from the factor). It first letter is $w[i+1]$, but as $w[i-1\,..\,i+|f|-1] = a^{1+|f|}$, we can change the definition to $i-1$ as well. The bound on the number of removed letters follows from a similar bound on Split, see Lemma 1. For the running time, we spend only $\mathcal{O}(1)$ time per letter of the word.

Concerning the distance between a factor and its definition, as the factorisation is proper, we have $i >$ *start*[i]. The case of $i =$ *start*[i]$+1$ is explicitly treated, and so $i -$ *start*[i] ≥ 2. Moreover, when we moved the definition, we moved it by exactly one letter to the left, so the equality holds for factors that were altered. □

We are now ready to show that Pairing produces a factorisation satisfying (P1–P3).

Lemma 3. Pairing *runs in linear time. It creates a proper factorisation and returns a pairing that satisfies (P1–P3) (for this new factorisation).*

Proof. For the running time analysis, note that each position of w is considered only once in the main loop and there are only $\mathcal{O}(1)$ operations peformed on it, each taking $\mathcal{O}(1)$ time.

Firstly, observe that all modifications to the factorisation are done by splitting, and so the obtained factorisation is proper. Moreover, as Pairing is applied after Preproc, we know that for each i that is a first letter of a factor we have factor $i - start[i] \geq 2$.

Let us make two preliminary observations:

- the considered position in the word is never decreased
- the pairing, once done, is not altered, with two exceptions: if $w[i]$ is the last letter in a factor and $pair[i] = \textit{first}$ then it is split from the factor and the pairing is cleared; if $w[i]$ is set to *none* then we can pair it, when we consider $i + 1$.

We show the second claim of the lemma by induction: when we processed $w[1\,..\,i-1]$ (i.e. we are at position i) then we have a partial pairing on $w[1\,..\,i-1]$, which differs from the pairing only in the fact that the position $i-1$ may be assigned as *first* in the pair and i is not yet paired.

If i is increased then we can create a new pair by pairing two unpaired letters (in line 19), which is fine, or by copying the pairing from factors definition (so in line 10), by induction assumption the letters there are paired correctly, the only potential problem is that i is assigned as *first* in the pair but there is no second element, but this is exactly the special case that the partial pairing allows. Lastly, we need to ensure that if $i-1$ was assigned as a *first* element in a pair then i will be assigned as the second in the pair (or the pairing on $i-1$ is cleared). Note that $i-1$ can be assigned in this way only when it is part of the factor, i.e. it gets the same status as some j. If i is also part of the same factor, then it is assigned the status of $j+1$, which by inductive assumption is paired with j, so is the second element in the pair (note that $j+1 < i$ and so the pairing of $j+1$ is already known). In the remaining case, if $i-1$ was the last element of the factor then we clear its pairing (in line 15).

We now show that the created partial pairing satisfies (P1–P3) restricted to $w[1\,..\,i-1]$; we begin with (P2): the pairing of the first letter is explicitly verified in line 5, if it is *second* then split $w[i]$ from the factor; for the last letter we similarly verify the condition in line 15.

Condition (P3) is explicitly enforced in loop in line 9, in which we copy the pairing from the definition of the factor. We can later remove the letters from the end of the factor, but this does not affect (P3) (as our factor only gets shorter).

Suppose that (P1) does not hold for $i-1, i$, i.e. they are both unpaired after processing i. It cannot be that they are both within the same factor, as then the corresponding $w[j-1]$ and $w[j]$ in the definition of the factor are also unpaired, by (P3), which contradicts the induction assumption. Similarly, it cannot be that one of them is in a factor and the other outside this factor: for the first letter this is explicitly verified in line 5 (in which case $w[i]$ is split from factor) and for the last letter this is considered when i is processed as a free letter in line 17 (and in this case $w[i-1]$ is split from factor). If both letters are free then this is verified when we consider the second of them, in line 17.

Finally, it is left to show that when we processed the whole w then we have proper pairing, i.e. that the last letter of w is not assigned as a *first* element of a pair. Consider, whether it is in a

factor or a free letter. If it is in a factor then clearly it is the last element of the factor and so it will be split and its pairing cleared in line 15. If it is a free letter observe that we only pair free letters in line 19, which means that it is paired with the letter on the next position, contradiction. □

We now show the bound on the number of letters that are made free during one application of Preproc and Pairing.

Lemma 4. *Consider an application of* Preproc *and* Pairing *to a factorisation of a word and a factor within this factorisation. Then there are at most* 4 *letters removed from the factor and made free plus perhaps one letter if the factor was removed entirely.*

Proof. Since the Split introduces (per application) one free letter (or two, when it removes the whole fragment), see Lemma 2, we count how many times it is applied to a single factor by Preproc and Pairing; clearly the former applies it at most once. Let a factor f begins (right before Preproc) at position i.

We claim that Pairing removes at most 2 letter from the beginning of the factor. Let j be the first letter of the definition of the factor (so $j = \mathit{start}[i]$). Clearly, if initially $\mathit{pair}[j] = \mathit{first}$ then no letter is split, as in line 5 its pairing is $\mathit{pair}[\mathit{start}[i]] = \mathit{first}$. So suppose that initially $\mathit{pair}[j] = \mathit{none}$ and consider $j+1$, which is smaller than i and so it has its pairing set. By (P1) the $\mathit{pair}[j+1] = \mathit{first}$ and such a pairing cannot be changed. Thus $w[i+1]$ is not removed from the factor (unless the whole factor is removed), as in line 5 its pairing is $\mathit{pair}[\mathit{pocz}[i+1]] = \mathit{first}$.

So consider now the case that $\mathit{pair}[j] = \mathit{second}$ and again consider $j+1 < i$. If $\mathit{pair}[j+1] = \mathit{first}$ then as above we split only one letter ($w[i]$). If $\mathit{pair}[j+1] = \mathit{none}$ then by (P1) the $\mathit{pair}[j+2]$ is *first*, as long as the pairing is already done for $j+2$, i.e. when $j+2 < i$. In such a case we can split only $w[i]$ and $w[i+1]$ and we will not split $w[i+2]$, as claimed.

So consider the last case, in which $i = j+2$, $\mathit{pair}[i-2] = \mathit{first}$ and $\mathit{pair}[i-1] = \mathit{none}$. But then when we consider $w[i]$ we split it and pair it with $i-1$. So when we consider $w[i+1]$, its definition $w[i-1]$ is paired as *first* and we do not split $w[i+1]$.

For the last letter of a factor, note that Split can be applied only to the last letter, however, it could be then also applied to the previous letter, when it had status *none*. Thus at most two letters are removed. This estimation is too weak for the factors modified by Preprocwe obtained only an upper-bound of 5 letters removed from such a factor. We claim that such a factor can have at most one letter removed at its end: observe that after the moving of the definition in Preproc the factor has its definition two letters to the left, see Lemma 2. This cannot change due to further applications of Split, see Lemma 1. We claim that there are no unpaired letters within such a factor. Consider this factor and two preceding letter, by (P3) the two preceding letters and two first letters of the factor are paired in the same way. So by (P1) none of them is unpaired. By easy induction, no letters of this factor are unpaired. Which means that if Preproc was applied to a factor, then Pairing removes at most 1 letter from the end, which yields the claim. □

Now, we show that when we have a pairing satisfying (P1–P3) (so in particular the one provided by Pairing is fine, but it can be any other pairing satisfying (P1–P3)) then PairReplacement creates a word w' out of w together with a factorisation.

Lemma 5. *When a pairing satisfies (P1–P3) then* PairReplacement *runs in linear time and returns a word w' together with a factorisation; $|w'| \leq \frac{2|w|+1}{3}$ and the factorisation of w' has the same number of factors as the factorisation of w. If p fresh letters were introduced then w' has p less free letters than w.*

Proof. The running time is obvious as we make one scan through w.

Firstly, we show that when we erase the information about beginnings and ends of factors of w we do not erase the newly created information for w'. To this end it is enough to show that the corresponding position in the new word is strictly smaller, in terms of indices of the PairReplacement: $i > i'$. Whenever i' is incremented, i is incremented by at least the same amount, so it is enough to show that $i > i'$ when i is the first letter of a factor, in other words, there is at least one pair before this factor (as for pair we increase i by 2 and i' by 1). By Lemma 2 after Preproc the difference between i and $start[i]$ is at least 2 and by (P1) one of $start[i]$, $start[i]+1$ is paired and by (P2) this pair is wholly outside this factor.

Concerning the size of the produced word, by (P1) each unpaired letter (perhaps except the last letter of w) is followed by a pair, so there are at least $\frac{1}{3}(|w|-1)$ pairs in the word. As we remove one letter for a pair, at least $\frac{1}{3}(|w|-1)$ letters are removed from w, which yields the claim.

We should show that w' has a factorisation with the same amount of factors. Firstly, factorisation of w' is proper: we create a factor from compressed letters of an old factor and we explicitly copy the letters from its definition; by (P3) pairing of a factor is the same as the pairing of its definition and by (P2) if the first (last) letter of a factor is paired then it is paired within this factor. Secondly, as free letters are replaced with free letters, factors are created only in place of old factors. As a result, the number of factors remains the same.

Concerning the number of fresh letters introduced, suppose that ab is replaced with c. If ab is within some factor f then we use for the replacement the same letter as we use in the factor definition and so no new fresh letter is introduced. If both this a and b are free letters then each such a pair contributes one fresh letter. And those two free letters are replaced with one free letter, hence the number of free letters decreases by 1. The last possibility is that one letter from ab comes from a factor and the other from outside this factor, but this contradicts (P2). □

With properties of the subprocedures established, the main claims on TtoG can be shown.

Theorem 1. TtoG *runs in linear time and returns an SLP of size at most* $\min(n, \ell+4\ell\log_{3/2}(n/\ell))$*, In particular, its approximation ratio is* $1+4\log_{3/2}(n/g)$*, where* g *is the size of the optimal grammar.*

Proof. For the running time, the creation of the LZ77 factorisation takes linear time [1, 3, 5, 8, 13, 22]. In each phase the preprocessing, pairing and replacement of pairs takes linear time in the length of the current word. Thanks to (P1) the length of such a word is reduced by a constant fraction in each phase, hence the total running time is linear.

Concerning the size of the created grammar, note that there is a trivial upper bound on the size of the produced grammar: each new letter (nonterminal) replaces two old ones, so in total we can introduce at most n new letters, so the size of the produced grammar is at most n.

Due to Lemma 5 each introduction of a fresh letter reduces the number of free letters by 1. Thus to bound the number of different introduced letters it is enough to estimate the number of created free letters. Let us fix a factor f of the original factorisation, we show that in total it introduces at most

$$1 + 4\log_{\frac{3}{2}}|f| \tag{1}$$

free letters, during the whole run of the algorithm. Note, that this bound holds also for initial free letters, as for them $|f| = 1$ and they 'introduce' one free letter.

Let us show that (1) is indeed enough to show approximation guarantee for TtoG. Let the initial LZ77 factorisation have size ℓ. Summing over all those factors yields that at most

$$\begin{aligned}\sum_{i=1}^{\ell} 1 + 4\log_{3/2}|f_i| &= \ell + 4\log_{3/2}\left(\prod_{i=1}^{\ell}|f_i|\right)\\ &\leq \ell + 4\log_{3/2}\left(\prod_{i=1}^{\ell}(n/\ell)\right)\\ &= \ell + 4\ell\log_{3/2}\left(\frac{n}{\ell}\right)\end{aligned}$$

free letters were introduced to the word during all phases, where the inequality follows due to inequality between means and the fact that $\sum_{i=1}^{\ell}|f_i| = n$. Hence the number of nonterminals in the grammar introduced in this way is at most $\ell + 4\ell\log_{3/2}(n/\ell)$. We would like to claim the same bound with g replacing ℓ.

Consider the function $f(x) = x + 4x\log_{3/2}(n/x)$, we want to show that

$$\min(n, f(\ell)) \leq f(g) \ , \tag{2}$$

which yields the claim of the theorem. If $f(g) \geq n$ then clearly (2) holds, so consider the case in which $f(g) < n$. The derivative f' of f is

$$f'(x) = 1 + 4\log_{3/2}(n/x) - 4/\ln(3/2)$$

and it can be routinely checked that f' decreases on the whole interval $[0, n]$, $f'(0) > 0$, $f'(n) < 0$ and $f(n) = n$. In particular, the maximum value of f on the interval $[0, n]$ (which is the interval of possible arguments of f) is at least n. Thus the interval on which $f(x) < n$ corresponds to some initial fragment of the possible values of x, wand the maximum of f is not attained on this interval. Thus f is increasing on this interval and so $f(g) < n$ implies that $f(\ell) \leq f(g)$, which ends the proof of (2).

Let us return to the proof of (1). The '+1' in it stands for the last letter that can be introduced by a factor, when it is removed; so we disregard it in the following estimations. By Lemma 4 during Preproc and Pairing at most 4 new free letters are created (except the possible 1 that is created during the pairing). By (P1) the length of f drops almost by 1/3 in each Pairing: after each unpaired letter there are two paired ones. Thus factor of length m is turned into factor of length at most $\frac{2}{3}(m-1)+1 = \frac{2(m+1)}{3}$. However, if at least one letter was split from this factor, its length before Preproc was at least $m+1$, so indeed it was shortened by a fraction at least $\frac{1}{3}$ of its length. This yields the proof of (1) and concludes the proof of the theorem. □

Acknowledgements

Most of this work was carried out when the author was a a Postdoctoral Fellow at Max Planck Institute fuer Informatik, funded by the Humboldt Foundation.

[1] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, William F. Smyth, German Tischler, and Munina Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Comput. Surv.*, 45(1):5, 2012.

[2] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[3] Gang Chen, Simon J. Puglisi, and William F. Smyth. Fast and practical algorithms for computing all the runs in a string. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 307–315. Springer, 2007.

[4] Rudi Cilibrasi and Paul M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[5] Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *DCC*, pages 482–488. IEEE Computer Society, 2008.

[6] Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In Camil Demetrescu and Magnús M. Halldórsson, editors, *ESA*, volume 6942 of *LNCS*, pages 421–432. Springer, 2011.

[7] Leszek Gąsieniec, Marek Karpiński, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT*, volume 1097 of *LNCS*, pages 392–403. Springer, 1996.

[8] Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *DCC*, pages 133–142. IEEE, 2013.

[9] Keisuke Goto and Hideo Bannai. Space efficient linear time lempel-ziv factorization for small alphabets. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *DCC 2014*, pages 163–172. IEEE, 2014.

[10] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

[11] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, Jan 2015.

[12] Artur Jeż and Markus Lohrey. Approximation of smallest linear tree grammar. In Ernst W. Mayr and Natacha Portier, editors, *STACS*, volume 25 of *LIPIcs*, pages 445–457. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[13] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013.

[14] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[15] Marek Karpiński, Wojciech Rytter, and Ayumi Shinohara. Pattern-matching for strings with short descriptions. In *CPM*, pages 205–214, 1995.

[16] John C. Kieffer and En-Hui Yang. Sequential codes, lossless compression of individual sequences, and Kolmogorov complexity. *IEEE Transactions on Information Theory*, 42(1):29–39, 1996.

[17] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Data Compression Conference*, pages 296–305. IEEE Computer Society, 1999.

[18] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M. B. Vitányi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004.

[19] Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[20] Masaya Nakahara, Shirou Maruyama, Tetsuji Kuboyama, and Hiroshi Sakamoto. Scalable detection of frequent substrings by grammar-based compression. *IEICE Transactions*, 96-D(3):457–464, 2013.

[21] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.

[22] Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *CPM*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011.

[23] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In Jan van Leeuwen, editor, *ESA*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.

[24] Frank Rubin. Experiments in text file compression. *Commun. ACM*, 19(11):617–623, 1976.

[25] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[26] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.

[27] James A. Storer and Thomas G. Szymanski. The macro model for data compression. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *STOC*, pages 30–39. ACM, 1978.