



# The Maximum Equality-Free String Factorization Problem: Gaps vs. No Gaps

Radu Stefan Mincu<sup>1</sup>(✉) and Alexandru Popa<sup>1,2</sup>

<sup>1</sup> Department of Computer Science, University of Bucharest, Bucharest, Romania  
{mincu.radu, alexandru.popa}@fmi.unibuc.ro

<sup>2</sup> National Institute for Research and Development in Informatics,  
Bucharest, Romania

**Abstract.** A factorization of a string  $w$  is a partition of  $w$  into substrings  $u_1, \dots, u_k$  such that  $w = u_1 u_2 \dots u_k$ . Such a partition is called equality-free if no two factors are equal:  $u_i \neq u_j, \forall i, j$  with  $i \neq j$ . The *maximum equality-free factorization problem* is to decide, for a given string  $w$  and integer  $k$ , whether  $w$  admits an equality-free factorization with  $k$  factors.

Equality-free factorizations have lately received attention because of their application in DNA self-assembly. Condon et al. (CPM 2012) study a version of the problem and show that it is  $\mathcal{NP}$ -complete to decide if there exists an equality-free factorization with an upper bound on the length of the factors. At STACS 2015, Fernau et al. show that the maximum equality-free factorization problem with a lower bound on the number of factors is  $\mathcal{NP}$ -complete. Shortly after, Schmid (CiE 2015) presents results concerning the Fixed Parameter Tractability of the problems.

In this paper we approach equality free factorizations from a practical point of view i.e. we wish to obtain good solutions on given instances. To this end, we provide approximation algorithms, heuristics, Integer Programming models, an improved FPT algorithm and we also conduct experiments to analyze the performance of our proposed algorithms.

Additionally, we study a relaxed version of the problem where gaps are allowed between factors and we design a constant factor approximation algorithm for this case. Surprisingly, after extensive experiments we conjecture that the relaxed problem has the same optimum as the original.

**Keywords:** String factorization · Equality-free · String algorithms · Heuristics

## 1 Introduction

To factorize a string (or word) means to obtain a partitioning of non-overlapping substrings that reconstitute the original string when concatenated in order. More

This work was supported by project PN19370401 “New solutions for complex problems in current ICT research fields based on modelling and optimization”, funded by the Romanian Core Program of the Ministry of Research and Innovation (MCI), 2019–2022.

© Springer Nature Switzerland AG 2020

A. Chatzigeorgiou et al. (Eds.): SOFSEM 2020, LNCS 12011, pp. 531–543, 2020.

[https://doi.org/10.1007/978-3-030-38919-2\\_43](https://doi.org/10.1007/978-3-030-38919-2_43)

exactly, a *factorization* of a string  $w$  is a tuple of strings  $(u_1, u_2, \dots, u_k)$  such that  $w = u_1 u_2 \cdots u_k$ .

Despite its simple definition, word factorization has a wide number of applications. For instance, finding an occurrence of a string  $v$  in a text  $t$  can be formulated as  $t$  admitting a factorization  $t = uvw$ . A string  $v$  is a prefix of another string  $t$  if  $t = vw$  and it is a suffix of  $t$  if  $t = uv$ . Moreover, many string problems can be seen as word factorization problems [5] such as: SHORTEST COMMON SUPERSTRING, LONGEST COMMON SUBSEQUENCE and SHORTEST COMMON SUPERSEQUENCE, to name a few. Another example of word factorization problem is the MINIMUM COMMON STRING PARTITION [1], a problem concerned with identifying factorizations for two strings such that the sequence of factors for one word is the permutation of the other's.

In this paper we focus on the equality-free factorization, a special case of word factorization in which all factors are distinct. The problem is motivated by an application in DNA synthesis [3]. More specifically, it is possible to produce short DNA fragments that will self-assemble into the wanted DNA structure. However, to obtain the desired structure, it is required that no two fragments are identical. Since the fragments must be short, one approach is to split the target DNA sequence into as many distinct pieces as possible.

PREVIOUS WORK. The equality-free factorization problem was first introduced by Condon, Mañuch and Thachuk [3] where it was presented as the *string partitioning problem*. The string partitioning problem asks for a factorization into distinct factors such that each factor is at most of a certain length. The problem was studied in a more general setting where the measure of collision between two factors is either equality or one is a prefix/suffix of the other. Condon et al. showed that these variants are  $\mathcal{NP}$ -complete. More recently, Fernau, Manea, Mercas and Schmid [4] presented a similar problem that imposes a lower bound on the number of factors instead of an upper bound on factor length. Fernau et al. showed that this variant is also  $\mathcal{NP}$ -complete. Afterwards, Schmid [5] studied the Fixed-Parameter Tractability of the two problems. Henceforth, we use the notation of Schmid and refer to the problem variant with a lower bound on the number of factors as MAXIMUM EQUALITY-FREE FACTORIZATION SIZE (MaxEFF-s).

PROBLEM DEFINITIONS. In this paper we consider the optimization version of the MaxEFF-s problem. Additionally, we consider a relaxed variant of the problem in which the factors do not necessarily cover the entire word, a so-called *gapped factorization*, that can be said to emerge from *generalized patterns* [2].

A *gapped factorization* of a string  $w$  over some alphabet  $\Sigma$  is a tuple of strings  $(u_1, u_2, \dots, u_k)$  such that  $w = \alpha_0 u_1 \alpha_1 u_2 \alpha_2 \cdots \alpha_{k-1} u_k \alpha_k$  with  $u_i \in \Sigma^+$  (non-empty substrings) and  $\alpha_i \in \Sigma^*$  (possibly empty substrings).

First, let us go over the base definitions:

1. *factorization* of  $w$  is a tuple of strings  $(u_1, u_2, \dots, u_k)$  s.t.  $w = u_1 u_2 \cdots u_k$ .
2. *equality-free factorization* is a factorization with all distinct factors.
3. *gapped factorization* of string  $w$  over alphabet  $\Sigma$  is a tuple  $(u_1, u_2, \dots, u_k)$  such that  $w = \alpha_0 u_1 \alpha_1 u_2 \alpha_2 \cdots \alpha_{k-1} u_k \alpha_k$ , with  $u_i \in \Sigma^+$  and  $\alpha_i \in \Sigma^*$ .

4. *size* of a factorization represents the number of factors.
5. *width* of a factorization represents the length of the longest factor.

Let us define the decision problem MaxEFF-s and its optimization version:

*Problem 1 (Maximum Equality-Free Factorization Size - Decision).* Does a given string admit an equality-free factorization of size at least  $k$ ?

*Problem 2 (Maximum Equality-Free Factorization Size - Optimization).* For a given string  $w$  find the largest integer  $k$ , such that  $w$  admits an equality-free factorization of size  $k$ .

In the rest of the paper we refer to Problem 2 as OptEFF-s.

In the relaxed variant we allow gaps between the factors of the string.

*Problem 3 (Maximum Gapped Equality-Free Factorization Size - Decision).* Does a given string admit a gapped equality-free factorization of size at least  $k$ ?

*Problem 4 (Maximum Gapped Equality-Free Factorization Size - Optimization).* For a given string  $w$  find the largest integer  $k$ , such that  $w$  admits a gapped equality-free factorization of size  $k$ .

To the best of our knowledge, the gapped version of the equality-free factorization problem has not been studied previously.

In the rest of the paper we refer to Problem 4 as OptGEFF-s.

**OUR RESULTS.** We provide heuristic algorithms for computing equality-free factorizations and we also give an approximation ratio guarantee. Additionally, in order to understand how well the algorithms perform it is necessary to compare the solutions of our algorithms with optimum solutions. For this purpose, we choose to build ILP models for OptEFF-s and OptGEFF-s, which we use with the state-of-the-art Gurobi solver to obtain optimum solutions on moderate sized instances.

The paper is organized as follows. In Sect. 2 we introduce our notations and present some observations. One such observation is used to improve the previously known best FPT algorithm for MaxEFF-s (see Sect. 3). Following that, we design a  $\frac{1}{2}$ -approximation algorithm for OptGEFF-s in Sect. 4. We would have liked to extend this result for OptEFF-s, since we conjecture that the optimum of the two problems is the same. In Sect. 5 we provide an ILP model for both OptEFF-s and OptGEFF-s that was successfully used to give optimum solutions using the Gurobi solver. It is with this model that we have discovered the same optimum for the two problems on each of nearly 300000 instances, leading to the conjecture that their optimum is the same. The design of our proposed heuristic algorithms (dubbed Greedy $k$ ) is presented in Sect. 6. We prove a  $\sqrt{OPT}$  approximation factor for Greedy1 and we give an example where Greedy1 has an approximation factor greater than  $\log n$ . We study the behavior of our algorithms on genomic data in Subsect. 6.3.

## 2 Preliminaries

We commonly use the notation  $S = s_1 s_2 \dots s_n$  for a string of length  $n$  over some alphabet  $\Sigma$ . A substring of  $S$  is identified by  $S[i..j] = s_i \dots s_j$  and has length  $j - i + 1$ .

Let there be a string  $w$  for which we are given an equality-free factorization of size  $k$ . Then, we can construct an equality-free factorization of size  $k - 1$  by concatenating one of the longest factors with one of its neighbors. This leads us to the following observation:

**Observation 1.** *If a string  $w$  admits an equality-free factorization of size  $k$ , then  $w$  admits equality-free factorizations of size  $i$ ,  $\forall i \in \{1, \dots, k\}$ .*

One of the implications of the previous observation is that, obtaining the solution of the optimization problem OptEFF-s for a given instance, provides the solutions for the decision problem MaxEFF-s on that respective instance and for all sizes.

**Observation 2.** *OptEFF-s( $w$ )  $\leq$  OptGEFF-s( $w$ ): For any input string  $w$ , an equality-free factorization of size  $k$  gives is also a solution for a gapped factorization of size  $k$ .*

Indeed, for  $w = \alpha_0 u_1 \alpha_1 u_2 \alpha_2 \dots \alpha_{k-1} u_k \alpha_k$ , if we consider all the gaps  $\alpha_i$  to be the empty string  $\epsilon$  then all equality-free factorizations of size  $k$  are also gapped factorizations of size  $k$ .

**Observation 3.** *In a string of length  $n$  there exists an equality-free factorization of maximum size with width (i.e. length of the longest factor) at most  $\lceil \sqrt{2n} \rceil$ .*

The previous observation follows from the fact that, in the worst case, all the factors have different length:  $1, 2, \dots, \ell$ . This brings us to the well-known finite sum  $n = \ell(\ell+1)/2$ . Solving for  $\ell$  shows that  $width \leq \lceil (\sqrt{1+8n}-1)/2 \rceil \leq \lceil \sqrt{2n} \rceil$ . We bring to the attention of the reader that this result is important for two reasons. First, we can use it to reduce the number of variables in our proposed ILP model in Sect. 5 from  $O(n^2)$  to  $O(n\sqrt{n})$ , drastically improving solver computing speed. The second reason is that this result improves the previously known best FPT algorithm for MaxEFF-s.

## 3 A Better FPT Algorithm for MaxEFF-S

In [5] Schmid shows an FPT algorithm for deciding if a string of length  $n$  has an equality-free factorization with  $k$  factors with complexity  $O((\frac{k^2+k}{2} - 1)^k)$ . In this section we design another algorithm with running time  $O((k^2 + k)^{\frac{k}{2}})$ . The algorithm is similar to the one of Schmid, but uses Observation 3.

By Observation 3, there exists an optimum solution with width at most  $\lceil \sqrt{2n} \rceil$ . Thus, instead of an  $O(n^k)$  algorithm to verify if a string of length  $n$  has a

factorization with  $k$  factors, we obtain an algorithm with complexity  $O((2n)^{\frac{k-1}{2}})$ , by trying all the possible starting points of the  $k - 1$  factors (notice that the first factor always starts at position 1).

Finally, the running time of the FPT algorithm follows from the following observation:

**Observation 4.** *When  $n \geq k(k+1)/2$ , there always exists a factorization, which means that the problem has a trivial polynomial kernel.*

## 4 A $\frac{1}{2}$ -Approximation Algorithm for OptGeff-s

In this section we show that there exists a natural reduction from OptGeff-s to the problem JISP $k$  (the so-called Job Interval Selection Problem with  $k$  intervals per job). Moreover, this problem admits a  $\frac{1}{2}$ -approximation [6].

An instance of JISP $k$  is a set of  $n$   $k$ -tuples (also called jobs), containing time intervals. The intervals are of the form  $[a, b]$  with  $a, b$  integers. Two time intervals  $[a, b]$  and  $[c, d]$  are said to intersect if  $[a, b] \cap [c, d] \neq \emptyset$ .

*Problem 5 (JISP $k$ ).* Given  $n$  jobs containing  $k$  time intervals each, find the maximum number of intervals that can be selected such that (i) no two intervals intersect and (ii) at most one time interval is selected per job.

**Theorem 1.** *An instance of OptGeff-s can be transformed into an instance of JISP $n$ , with the same optimum solution.*

*Proof.* We proceed to construct an instance of JISP $n$  containing  $O(n^2)$  jobs from a string  $w$  of length  $n$ .

Consider the factors in a gapped equality-free factorization of a string  $w$  of length  $n$ . They are a set of non-overlapping and distinct substrings of  $w$ . For each distinct substring of  $w$  (which is a possible factor) we create a job in the corresponding JISP $n$  instance. For each job created from a substring  $s$ , we add as time intervals  $[a, b]$  the start and end indices of all the occurrences  $s = w[a \dots b - 1]$  of  $s$  in  $w$ . At this moment, we have created a set of jobs that are not  $n$ -tuples and therefore cannot be said to be a JISP $n$  instance.

To obtain a JISP $n$  instance, we simply pad each tuple by adding an appropriate number of duplicate intervals. This operation constructs an equivalent instance due to the observation that a JISP( $n + t$ ) instance with the same optimum as a JISP $n$  instance can be created by adding  $t$  duplicates of an arbitrarily selected interval within every job.

Since there are at most  $n$  occurrences of a substring in a string and there exist  $O(n^2)$  distinct substrings in any given string, we have shown that we can construct a JISP $n$  instance with  $O(n^2)$  jobs, from any string of length  $n$ .

Moreover, a solution for the JISP $n$  instance that is constructed in the manner described above immediately gives us a solution for OptGeff-s. Each interval selected from a job corresponds to the occurrence of a factor in the initial string. The intervals are not allowed to intersect and thus the factors are not allowed

to overlap. Only one interval may be selected per job and therefore only distinct factors may be selected because the jobs correspond to distinct substrings. As such, we conclude that we can reduce OptGeff-s to JISPn.  $\square$

It is known that JISPn has the following greedy  $\frac{1}{2}$ -approximation algorithm [6]: at each step, select the time interval with the lowest end time that does not intersect already selected intervals. Using Theorem 1 we have shown that:

**Theorem 2.** *OptGeff-s has a  $\frac{1}{2}$ -approximation algorithm.*

With the above results, we may now present a more tidy version of the greedy approximation algorithm for the OptGeff-s: for each position  $j = 1, 2, \dots, n$  in a string  $w$  of length  $n$ , select as a factor (if possible) any substring  $s$  of  $w$  that ends on position  $j$  such that (i)  $s$  does not overlap the previously selected factor and (ii)  $s$  is not equal to any previously selected factor.

### 5 ILP Formulations for OptEFF-S and OptGeff-S

We define an ILP model for the problems and then explain the notations:

$$\max \sum_{i=1}^n \sum_{j=1}^{n-i+1} x_{ij} \text{ subject to:} \tag{1a}$$

$$\sum_{j=1}^{n-i+1} x_{ij} \leq 1, \forall i = 1, 2, \dots, n \tag{1b}$$

$$x_{ij} + x_{k\ell} \leq 1, \forall i = 1, 2, \dots, n-1, \forall j = 1, 2, \dots, n-i+1 \tag{1c}$$

$$\forall k = i+1, i+2, \dots, i+j-1 \text{ and } k \leq n, \forall \ell = 1, 2, \dots, n-k+1$$

$$x_{i\ell} + x_{k\ell} \leq 1, \forall i, k, \ell, \text{ where } S[i..i+\ell] = S[k..k+\ell] \tag{1d}$$

$$x_{ij} - \sum_{\ell=1}^{n-i-j+1} x_{(i+j)\ell} \leq 0, \forall i = 1, 2, \dots, n-1, \forall j = 1, 2, \dots, n-i+1 \tag{1e}$$

$$\sum_{\ell=1}^n x_{1\ell} \geq 1 \text{ and } \sum_{i=1}^n x_{i(n-i+1)} \geq 1 \tag{1f}$$

$$x_{ij} \in \{0, 1\}, \forall i = 1, 2, \dots, n-1, \forall j = 1, 2, \dots, n-i+1 \tag{1g}$$

1. The binary variables  $x_{ij}$  (see 1g) represent the choice for a factor starting on position  $i$  of length  $j$ .
2. We need to maximize the number of factors i.e. sum of  $x_{ij}$  (see 1a).
3. Only one factor (regardless of length) may begin on any position (see 1b).
4. Factors cannot overlap i.e. begin inside each other (see 1c).
5. Distinct factors: only one of the occurrences of a factor may be selected (1d).

If we want an equality-free gapped factorization, conditions 1–5 are enough. To enforce factorizations without gaps, we add:

---

**Algorithm Greedy1:** reads the input string  $w$  left-to-right and builds an equality-free factorization  $F$  by greedily adding the next shortest factor not yet present in  $F$ .

*input:* string  $w[1..n]$ ; *output:* equality-free factorization  $F$ ;

---

```

1:  $last \leftarrow 1, F \leftarrow \emptyset$ ;
2: for  $i \leftarrow 1, n$  step 1 do
3:   if  $w[last..i] \notin F$  then
4:      $F \leftarrow F \cup \{w[last..i]\}, last \leftarrow i + 1$ ;
5: if  $last \neq n + 1$  then
6:    $F \leftarrow (F \setminus \{F[-1]\}) \cup \{F[-1] \cdot w[last..n]\}$ ;
7: return  $F$ ;

```

---

1. If a factor  $u_t$  is selected, then a factor  $u_{t+1}$  of any length that begins immediately after  $u_t$  ends must be selected (see 1e).
2. A factor starting on position 1 must be selected; a factor ending on the last position must be selected (see 1f).

Recall that by using Observation 3 we may reduce the number of variables in the formulation to  $O(n\sqrt{n})$  by discarding  $x_{ij}$  with  $j > \lceil \sqrt{2n} \rceil$ .

## 6 Heuristic and Approximation Algorithms for OptEFF-S

In this section we present a family of heuristic greedy-based algorithms for the OptEFF-s problem. We begin with presenting the outline for the algorithms and then we evaluate the performance of the algorithms on datasets composed of randomly generated strings.

The proposed algorithms are based on building a factorization by reading the input left-to-right and greedily adding words to the incumbent solution.

### 6.1 Description of Greedy1

To illustrate the most basic strategy, consider starting “at the left” and adding the next shortest substring (distinct from the already selected factors) to the incumbent factorization at each step of the algorithm (see Algorithm Greedy1). The only issue is to define the behavior of the algorithm at the end of the string, where we may have a remainder that is already present as a factor in the working solution. We choose to simply concatenate this remainder to the last factor. We mention here that this algorithm is essentially identical to the well-known LZ’78 factorization procedure [7], excepting the handling of the last factor. To prove the correctness of our algorithm, the following property of Greedy1 is of interest:

*Property 1 (Prefix property).* Let there be two factors  $u_i$  and  $u_j$  in a factorization  $w = u_1u_2 \cdots u_n$  computed by Algorithm Greedy1. If  $u_i$  is a prefix of  $u_j$  then  $i < j$ . In other words, for any factor constructed by Greedy1, its prefixes precede it in the factorization.

**Theorem 3.** *Greedy1 yields an equality-free factorization in  $O(n)$  time.*

*Proof.* By adding the next distinct substring at each step, the equality-free condition of the factorization is satisfied. The only question is if the behavior of the algorithm at the end of the string is correct or may yield a duplicate factor. If there is a remainder  $r$  at the end of the sequence of operations, then it has a duplicate in the factorization. Concatenating this remainder to the last factor  $v$  always produces the equality-free factor  $vr$ . To prove this we use Property 1: the resulting factor  $vr$  must have a duplicate preceding  $v$  in the factorization for the procedure to be incorrect. However,  $v$  is a prefix of  $vr$  and must appear before  $vr$ , a contradiction. Therefore the factorization is equality-free.

In the implementation of Greedy1, if the incumbent factorization is a list of starting/ending positions of factors and we use a hash set structure to check for collisions, the average running time is  $O(nt)$ , with  $t$  being the average factor length  $t = \frac{n}{OPT}$  and  $OPT$  being the size of the optimum. This is because we compute a linear time string hash function  $n$  times (i.e. for each  $\notin$  operation; see Algorithm Greedy1, line 3). We can optimize by changing the way we compute the hash function (by not discarding previous partial results) or by using a modified insertion in a trie structure instead of using hash sets in order to bring the time down to  $O(n)$ .  $\square$

**Theorem 4.** *The Greedy1 algorithm is a  $\sqrt{OPT}$  approximation for OptEFF-s.*

*Proof.* A string with  $n$  characters can be factorized in at most  $n$  factors. The Greedy1 algorithm produces at least  $\sqrt{n}$  factors—the case when all the factors have different length.  $\square$

In the following paragraphs we focus on the tightness of Greedy1 as an approximation algorithm. In practice Greedy1 can offer very good solutions as can be seen in Fig. 1 from Subsect. 6.3. However, we show that there exists an instance for which the ratio between Greedy1 and  $OPT$  is  $\Omega(\log n)$ . Therefore, in order to obtain a constant factor approximation for OptEFF-s we need to design a different algorithm.

**Theorem 5.** *Let there be an alphabet  $\Sigma = \{x_1, x_2, \dots, x_n\}$ . We build a string  $s = X_1 \cdot X_2 \cdots X_n$  by concatenating in order strings  $X_i = x_1x_2 \cdots x_i$ . There exists a factorization of  $s$  with  $\Omega(n \log n)$  factors.*

*Proof.* A factorization of  $s$  is as follows. We begin with  $X_n$  and factorize it into  $n$  factors:  $x_1|x_2| \cdots |x_n$ . At each iteration  $1 \leq i \leq \lfloor n/2 \rfloor$  we factorize  $X_{n-i+1}$  into  $\lfloor (n-i+1)/i \rfloor$  factors of length  $i$  in left-to-right order:  $x_1 \cdots x_i|x_{i+1} \cdots x_{2i}| \cdots$ . If  $X_{n-i+1}$  is not a multiple of  $i$ , then we concatenate the remainder of length  $< i$  to the last factor. All of the factors added at iteration  $i$  are distinct, but a question remains about the correctness of the algorithm regarding the concatenated factor. Correctness is ensured because the new factor constructed in this manner contains  $x_{n-i+1}$ , symbol that cannot appear in subsequent iterations. In the end, the prefix  $X_1X_2 \cdots X_{\lfloor n/2 \rfloor}$  is transformed into one factor.

The number of factors in this solution is  $n + \lfloor (n-1)/2 \rfloor + \lfloor (n-2)/3 \rfloor + \cdots + 1$  which is  $\Omega(n + (n/2 - 1) + (n/3 - 1) + \cdots + 1) = \Omega(n(1 + 1/2 + 1/3 + \cdots + 1/n)) = \Omega(n \log n)$ .  $\square$



On string  $s$ , Greedy1 produces  $n$  factors, i.e. the output factorization is  $X_1|X_2|\cdots|X_n$ . Using Theorem 5 we conclude that:

**Corollary 1.** *The approximation ratio of Greedy1 is  $\Omega(\log n)$ .*

### 6.2 Description of Greedy $k$

We generalize Algorithm Greedy1 into the family Greedy $k$  in the following way: instead of adding the next distinct substring to the incumbent factorization we consider adding  $k$  factors at the same time. In other words, we select the shortest substring that follows such that this substring admits a partition into  $k$  distinct factors that have not yet been selected (see Algorithms Greedy $k$  and Factorization). Again, the behavior of the algorithm needs to be defined when it is no longer possible to split a remaining substring into  $k$  factors and now it is a little more complicated because Greedy $k$  does not benefit from Property 1:

1. Greedy $k$  will first attempt to partition the remainder  $r$  into  $t$  distinct and not yet selected factors with  $t = k - 1, \dots, 1$ .
2. If this is not possible, then we try to discard the last  $k$  factors in the factorization  $u_i, u_{i-1}, \dots, u_{i-k+1}$  and re-partition the entire substring  $u_{i-k+1} \cdots u_i r$  into  $t$  valid factors,  $t = k, k - 1, \dots, 1$ . This step is added to the algorithm to ensure that Greedy $k$  will always produce an optimum solution when the optimum is known to be  $k$ .
3. Finally, if the above steps fail, we append the remainder  $r$  to the last factor and proceed to concatenate the last two factors in the solution until duplicates no longer exist.

---

**Function Factorization**( $T, F, start, end, w, k$ ): returns a factorization  $F$  with exactly  $k$  factors of the substring  $w[start..end]$ , or *null* if no such factorization is possible. Moreover, the factors in  $F$  must not already appear in the incumbent factorization  $T$ .  
*input:* factorizations  $T, F$  (pre-initialized), string  $w[1..n]$ , int  $start$ , int  $end$ , int  $k$ ;  
*output:* factorization  $F$  (modified from the input  $F$ );

---

```

1: if  $end - start + 1 < k$  then
2:   return null;
3: if  $k = 1$  and  $w[start..end] \notin T$  and  $w[start..end] \notin F$  then
4:    $F \leftarrow F \cup \{w[start..end]\}$ ;
5:   return  $F$ ;
6: for  $i \leftarrow start, end - 1$  step 1 do
7:   if  $w[start..i] \notin T$  and  $w[start..i] \notin F$  then
8:     solution  $\leftarrow$  Factorization( $T, F \cup \{w[start..i]\}, i + 1, end, w, k - 1$ );
9:     if solution  $\neq$  null then
10:      return solution
11: return null;
```

---

**Theorem 6.** *The Factorization( $T, F, start, end, w, k$ ) recursive function obtains a factorization of size  $k$ , if it exists in the specified substring of length  $\ell$ , in time  $O(\ell^k)$ .*

*Proof.* If we denote the length of the substring as  $\ell = \text{end} - \text{start} + 1$ , we can observe that we test all of the  $\binom{\ell-1}{k-1} = O(\ell^{k-1})$  partitions. The time necessary for the  $\notin$  operation depends on the structure used to check for substring collisions. Whether we employ a hash set and compute a hash function or use a trie, the time required is  $O(\ell)$  i.e. linear in substring size. Observe that when  $k = 1$  the function takes  $O(\ell)$  time and that by increasing  $k$  by 1, the time is multiplied by  $O(\ell)$ . By simple induction, the time complexity of the function is  $O(\ell^k)$ .  $\square$

**Theorem 7.** *Greedyk is correct and runs in  $O((n+k)\ell^k)$  average time, with  $\ell = k \frac{n}{OPT}$  and  $OPT$  being the value of the optimum solution.*

*Proof.* The equality-free condition of the factorization is satisfied when  $k$  distinct factors are added at each step (lines 2–5 in the pseudocode). When it is no longer possible, a substring  $r$  may remain:

1. First we try to split  $r$  into fewer than  $k$  factors (see lines 7–10). If we are successful, then the resulting factorization is equality-free and covers the entire string and is therefore correct.
2. Secondly, we try to discard the last  $k$  factors and refactorize the entire remaining substring into  $k$  factors or less (see lines 11–14). If we are successful, the resulting factorization is correct.

---

**Algorithm Greedyk:** reads the input string  $w$  from left to right and builds a Factorization  $F$  by greedily adding  $k$  distinct, not yet selected factors at each step. We use the notations *newEmptyFactorization* to denote creation of a new factorization structure and  $F[-1]$  to refer to the last factor inside the factorization structure  $F$ .

*input:* string  $w[1..n]$ , *int*  $k$ ; *output:* Factorization  $F$ ;

---

```

1:  $prev \leftarrow 1, last \leftarrow 1, F \leftarrow \emptyset, solution \leftarrow \emptyset, lastsol \leftarrow \emptyset, sol2 \leftarrow \emptyset$ ;
2: for  $i \leftarrow 1, n$  step 1 do
3:    $solution \leftarrow Factorization(F, newEmptyFactorization, last, i, w, k)$ ;
4:   if  $solution \neq null$  then
5:      $F \leftarrow F \cup solution, prev \leftarrow last, last \leftarrow i + 1, lastsol \leftarrow solution$ ;
6: if  $last \neq n + 1$  then
7:   for  $j \leftarrow k - 1, 1$  step  $-1$  do
8:      $solution \leftarrow Factorization(F, newEmptyFactorization, last, n, w, j)$ ;
9:     if  $solution \neq null$  then
10:      return  $F \cup solution$ ;
11:   for  $j \leftarrow k, 1$  step  $-1$  do
12:      $sol2 \leftarrow Factorization(F \setminus lastsol, newEmptyFactorization, prev, n, w, j)$ ;
13:     if  $sol2 \neq null$  then
14:       return  $(F \setminus lastsol) \cup sol2$ ;
15:    $lastFactor \leftarrow F[-1], F \leftarrow F \setminus \{F[-1]\}, lastFactor \leftarrow lastFactor \cdot w[last..n]$ ;
16:   while  $lastFactor \in F$  do
17:      $lastFactor \leftarrow F[-1] \cdot lastFactor$ ;
18:      $F \leftarrow F \setminus \{F[-1]\}$ ;
19:    $F \leftarrow F \cup \{lastFactor\}$ ;
20: return  $F$ ;
```

---

- Thirdly we employ a fallback where we append the remainder  $r$  to the last factor and keep concatenating the last two factors until the factorization is equality-free (see lines 15–19).

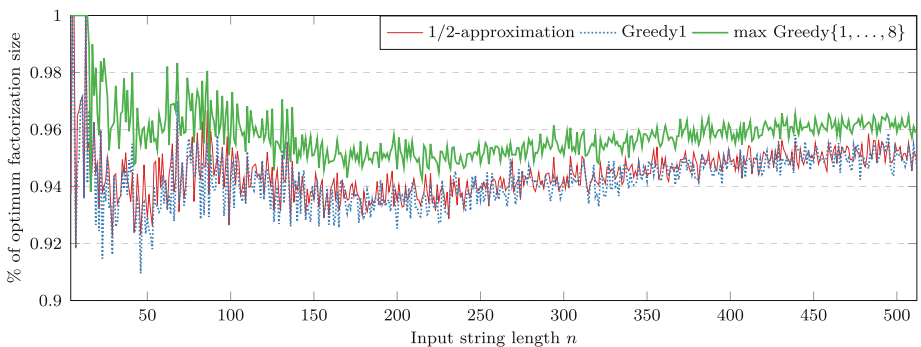
In the implementation of Greedy $k$ , we traverse the string and attempt to partition a substring of length  $\ell$  into  $k$  valid factors. This operation takes  $O(\ell^k)$  time (see Theorem 6). There are  $O(n + k)$  calls to the partitioning function, therefore the average running time for the algorithm is bounded by  $O((n+k)\ell^k)$ , with  $\ell = k\frac{n}{OPT}$ .

### 6.3 Experimental Results

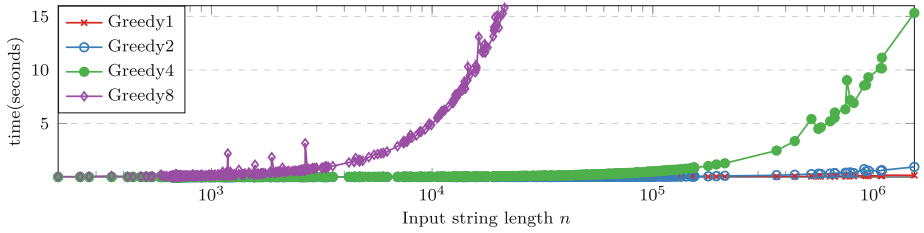
First, we want to determine the solution quality given by the Greedy $k$  algorithms in practice. The dataset we have selected for this experiment is the RNA string of *Saccharomyces cerevisiae* narnavirus 23S (obtained from [yeastgenome.org](http://yeastgenome.org)).

The methodology for the experiment is as follows. For each integer  $\ell \in \{4, \dots, 512\}$  we randomly select 10 substrings of length  $\ell$  from the input RNA string (whose length is 2891). We compute the optimum solution for each substring using the ILP formulations in Sect. 5 and the Gurobi solver. We proceed to compute the value of the  $\frac{1}{2}$ -approximation algorithm for OptGEFF-s from Sect. 4, as well as Greedy $\{1, \dots, 8\}$ . Following that, we average the results among the 10 substrings for each length  $\ell$  and plot the fractions  $\frac{\frac{1}{2}\text{-approximation}}{OPT}$ ,  $\frac{\text{Greedy1}}{OPT}$  and  $\frac{\max(\{\text{Greedy1}, \dots, \text{Greedy8}\})}{OPT}$  in Fig. 1.

We are pleased to report Greedy1 situating within 91% of the optimum (alongside the  $\frac{1}{2}$ -approximation) and  $\max(\{\text{Greedy1}, \dots, \text{Greedy8}\})$  placing within 93% of the optimum. All solutions are within 94% of the optimum on lengths 350–512. In Fig. 2 we also display the running time of Greedy $k$  on longer strings (using whole genomes from [yeastgenome.org](http://yeastgenome.org) including *Saccharomyces bayanus* and *Saccharomyces cerevisiae*). The experiments demonstrate that the Greedy $k$  algorithms are fit for practical usage.



**Fig. 1.** Plot describing the values of the  $\frac{1}{2}$ -approximation algorithm for OptGEFF-s, Greedy1, and  $\max(\{\text{Greedy1}, \dots, \text{Greedy8}\})$  for all integer string lengths  $\in \{4, \dots, 512\}$ . All  $y$ -axis values are averages of 10 instances per  $x$ -axis point, as well as being divided by the average of the optima of the 10 instances.



**Fig. 2.** The running time of Greedy{1, 2, 4, 8}. The  $x$ -axis is logarithmic scaled.

During our testing, we have computed exact solutions using our ILP model for both OptEFF-s and OptGEFF-s on each test instance. We hypothesized that OptGEFF-s would have a higher value solution than OptEFF-s. However, after having evaluated some 300.000 random instances (using various lengths and alphabet size), we observed no difference between the two problems regarding the size of the exact solutions on the same instance. Thus, we believe that the two problems share the same optimum.

## 7 Conclusions and Open Problems

We have presented heuristic and approximation algorithms for the OptEFF-s and OptGEFF-s problems. Moreover, our experiments show insights into the nature of the problem and provide high quality solutions.

We leave as open problems the following conjectures:

*Conjecture 1 (Gaps = No Gaps).*  $\text{OptGEFF-s}(w) = \text{OptEFF-s}(w), \forall w.$

We strongly believe that the optimum for the two problems OptGEFF-s and OptEFF-s is one and the same. This result can be achieved if one of the following statements is proven:

1. Given a string, it is possible to transform an equality-free gapped factorization of maximum size into one without gaps and of the same size.
2. The number of factors in the maximum size equality-free factorization for an instance does not decrease if we insert a symbol anywhere in the string.

*Conjecture 2 (Greedy1 approximation ratio is tight).* There exists an instance for which the ratio between Greedy1 and OPT is  $\Theta(\sqrt{n})$ .

We conjecture that the analysis of the Greedy1 algorithm is tight. Nevertheless, we leave this as an open problem.

## References

1. Bulteau, L., Hüffner, F., Komusiewicz, C., Niedermeier, R.: Multivariate algorithms for NP-hard string problems. *Bull. EATCS* **114**, 295–301 (2014)
2. Clifford, R., Harrow, A.W., Popa, A., Sach, B.: Generalised matching. In: Karlgren, J., Tarhio, J., Hyvrö, H. (eds.) *SPIRE 2009*. LNCS, vol. 5721, pp. 295–301. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03784-9\\_29](https://doi.org/10.1007/978-3-642-03784-9_29)
3. Condon, A., Mañuch, J., Thachuk, C.: The complexity of string partitioning. *J. Discrete Algorithms* **32**, 24–43 (2015)
4. Fernau, H., Manea, F., Mercas, R., Schmid, M.L.: Pattern matching with variables: fast algorithms and new hardness results. In: *32nd International Symposium on Theoretical Aspects of Computer Science*, 4–7 March 2015, Garching, Germany, pp. 302–315 (2015)
5. Schmid, M.L.: Computing equality-free and repetitive string factorisations. *Theor. Comput. Sci.* **618**, 42–51 (2016)
6. Spieksma, F.: On the approximability of an interval scheduling problem. *J. Sched.* **2**(5), 215–227 (1999)
7. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* **24**, 530–536 (1978)