Longest Common Abelian Factors and Large Alphabets

Golnaz Badkobeh[1], Travis Gagie[2], Szymon Grabowski[3], Yuto Nakashima[4,5], Simon J. Puglisi[2], and Shiho Sugimoto[4]

[1] University of Warwick, Department of Computer Science, Conventry, United Kingdom
g.badkobeh@warwick.ac.uk
[2] Helsinki Institute for Information Technology
University of Helsinki, Department of Computer Science
Helsinki, Finland
{gagie, puglisi}@cs.helsinki.fi
[3] Lodz University of Technology, Institute of Applied Computer Science, Łódź, Poland
sgrabow@kis.p.lodz.pl
[4] Kyushu University, Department of Informatics, Kyushu, Japan
{shiho.sugimoto, yuto.nakashima}@inf.kyushu-u.ac.jp
[5] Japan Society for the Promotion of Science, Tokyo, Japan

Abstract. Two strings X and Y are considered Abelian equal if the letters of X can be permuted to obtain Y (and vice versa). Recently, Alatabbi et al. (2015) considered the *longest common Abelian factor problem* in which we are asked to find the length of the longest Abelian-equal factor present in a given pair of strings. They provided an algorithm that uses $O(\sigma n^2)$ time and $O(\sigma n)$ space, where n is the length of the pair of strings and σ is the alphabet size. In this paper we describe an algorithm that uses $O(n^2 \log^2 n \log^* n)$ time and $O(n \log^2 n)$ space, significantly improving Alatabbi et al.'s result unless the alphabet is small. Our algorithm makes use of techniques for maintaining a dynamic set of strings under split, join, and equality testing (Melhorn et al., Algorithmica 17(2), 1997).

1 Introduction

Two strings X and Y are considered to be *Abelian equal* if the letters of X can be permuted to obtain Y (and vice versa). At the String Masters 2013 meeting, Thierry Lecroq and Arnaud Lefebvre, posed the *longest common Abelian factor problem* in which we are asked to find the length of the longest Abelian-equal factor present in a given pair of strings.

The problem is a variant on the classic longest common factor (LCF) problem, the colorful history of which has been recently chronicled by Apostolico et al. [3]. The LCF of two strings can be computed in time linear in the length of

the strings via suffix tree construction, and indeed the drive for a linear-time LCF algorithm was the reason the suffix tree was unearthed when it was.

To our knowledge, the only work on the LCAF problem was presented very recently by Alatabbi et al. [1]. They describe an algorithm that runs in $O(\sigma n^2)$ worst-case time, using $O(\sigma n)$ working space[6], where n is the length of the strings and $\sigma = |\Sigma|$ is the alphabet size.

Our main result in this paper is an algorithm that uses $O(n^2 \log^2 n \log^* n)$ time and $O(n \log^2 n)$ space, significantly improving Alatabbi et al.'s result unless the alphabet is $o(\log^2 n \log^* n)$. To obtain this result, we make use of techniques for maintaining a set of strings under split, join, and equality testing by Melhorn, Sundar, and Uhrig [8].

We also show how to reduce the space requirements of Alatabbi et al.'s algorithm from $O(\sigma n)$ to $O(n)$, without affecting their running time. Before getting to these new results, however, in Section 3 we highlight a link between the LCF and LCAF problems that provides an alternative path to Alatabbi et al.'s upperbound.

2 Preliminaries

Let $S = S[1..n]$ be a string of length n over an alphabet Σ of size $\sigma = |\Sigma|$. For $1 \leq i \leq j \leq n$, $S[i]$ denotes the ith symbol of S, and $S[i..j]$ the contiguous sequence of symbols (or *factor* or *substring*) $S[i]S[i+1] \dots S[j]$. We will use the same notation for arrays. String $S[i..j]$, where $j - i + 1 = \ell$, will also be called an ℓ-gram from S. Throughout we assume that $\sigma = O(n)$ and $\Sigma = \{1, 2, \dots, \sigma\}$. If this is not the case, we can first remap the alphabet for both input strings in $O(n \log n)$ time and using $O(n)$ extra space.

The *Parikh vector* for string S, denoted as $P(S)[1 \dots \sigma]$, is defined as a vector (array) of size σ storing the number of occurrences of each alphabet symbol in S. Formally, $P(S)[c] = k$ iff $|\{i : S[i] = c\}| = k$, for any alphabet symbol c. For two strings S and T of equal length and over a common alphabet, we say that the Parikh vector $P(S)$ is (lexicographically) smaller than the Parikh vector $P(T)$, denoted as $P(S) < P(T)$, iff there exists an alphabet symbol c', $1 \leq c' \leq \sigma$, such that $P(S)[c] = P(T)[c]$ for all $c < c'$ and $P(S)[c'] > P(T)[c']$. The two Parikh vectors are equal, i.e., $P(S) = P(T)$, when $P(S)[c] = P(T)[c]$ for all symbols c.

3 LCAF via LCF

While Alatabbi et al.'s algorithm for computing the LCAF is simple, we note here that the same result can be immediately obtained by a reduction from the LCF problem.

Hui [6] showed that using a generalized suffix tree it is possible to find the LCF for a pair of strings of length n in $O(n)$ time. We use this algorithm n

[6] We express space usage in words, throughout.

times, for each factor length ℓ, replacing each ℓ-length factor by its Parikh vector followed with a unique terminator (e.g., for the factors taken from A the subsequent terminators can be -1, -2, ..., while for the factors taken from B they can be $-n-1$, $-n-2$, ...). The terminators prevent matches longer than σ. If there exists an LCF of length exactly σ, it must correspond to a pair of factors, one from A and one from B, of length ℓ. This takes $O(\sigma n)$ time for one value of ℓ, using $O(\sigma n)$ space, hence the total time, for all possible factor lengths, becomes $O(\sigma n^2)$ with $O(\sigma n)$ space (we build and discard the generalized suffix trees one at a time). In this way, we obtain the same time and space bounds as Alatabbi et al.'s solution.

4 Reducing space usage in Alatabbi et al.'s algorithm

Recently, Kociumaka et al. [7] showed that for any tradeoff parameter $1 \leq \tau \leq n$, the LCF problem can be solved in $O(\tau)$ space and $O(n^2/\tau)$ time. Applying this to the LCAF problem, we obtain an algorithm using $O(\tau\sigma n^2)$ time and $O(\sigma n/\tau)$ space, for any $1 \leq \tau \leq \sigma n$.

The specifics of LCAF, however, allow for a better result. We consider each factor length ℓ separately. For a given ℓ, we sort all $n-\ell+1$ factors of A according to their Parikh vectors, using LSD radix sort. Each factor is represented as its start position in A. There are σ passes of the radix sort and the problem seems to be accessing the keys' "digits". However, before each pass of the radix sort we scan A and for each ℓ-sized window collect the count of the corresponding symbol in it. More precisely, just before the ith pass of the radix sort, in which the keys will be distributed according to $P(\cdot)[\sigma - i + 1]$, we compute and store $P(A[j \ldots j+\ell-1])[\sigma-i+1]$ for each factor $A[j \ldots j+\ell-1]$, using $O(n)$ time and $O(n)$ extra space. This allows us to access a digit in the radix sort in constant time. After the ith pass, the $P(\cdot)[\sigma - i + 1]$ statistics are discarded. In this way, sorting of the ℓ-length factors of A takes $O(\sigma n)$ time and requires $O(n)$ working space, including for the output.

We sort the factors of B in the same way. Additionally, for every σth evenly sampled ℓ-length factor of A and B, we store explicitly its Parikh vector using $O(\sigma)$ space. More precisely, we compute and store the Parikh vectors for the factors $A[1 \ldots \ell], A[\sigma+1 \ldots \sigma+\ell], A[2\sigma+1 \ldots 2\sigma+\ell], \ldots$, and similarly for $B[1 \ldots \ell], B[\sigma+1 \ldots \sigma+\ell], B[2\sigma+1 \ldots 2\sigma+\ell], \ldots$. Because we scan the strings from left to right and compute the successive Parikh vectors incrementally (first making a copy of the previous vector), this phase takes $O(n + (n/\sigma)\sigma) = O(n)$ time and $O(n)$ space.

The computed Parikh vectors serve to speed up factor comparisons during the last phase, which is to intersect the lists of factors from A and B (similar to a two-way merge). By using the sampled Parikh vectors that we have kept at regular intervals of A and B, each factor comparison takes $O(\sigma)$ time, and the intersection therefore takes $O(\sigma n)$ time.

The total cost of the described procedure, over all relevant factor lengths, becomes $O(\sigma n^2)$ and the required space is $O(n)$. This matches the time complexity of Alatabbi et al.'s solution, but reduced space usage by a factor of σ.

5 New algorithm based on dynamic string sets

To determine if A and B have a common factor of length ℓ, it suffices to be able to count the number of distinct Parikh vectors in a string. To see this, let $D_\ell(A)$, $D_\ell(B)$, and $D_\ell(A\$B)$ denote the number of distinct Parikh vectors in strings A, B, and $A\$B$, respectively, where \$ is a sentinel symbol not occurring in either A or B.

Clearly, if

$$D_\ell(A\$B) < D_\ell(A) + D_\ell(B) \tag{1}$$

then at least one Parikh vector is shared by A and B, and so the LCAF will have length at least ℓ. This gives us a simple algorithm for computing the LCAF: compute $D_\ell(A)$, $D_\ell(B)$, and $D_\ell(A\$B)$ for every $\ell \in [1, n]$; the largest ℓ for which (1) holds is the length of the LCAF.

For the remainder of this section we focus on how to compute $D_\ell(X)$ for a given string X and window length ℓ. Our main tool is a data structure due to Melhorn et al. [8] for maintaining a (dynamic) set of strings under split, join, and equality testing. More precisely, their data structure supports four operations:

(i) make_sequence(s, a_1): creates the sequence s equal to the symbol a_1.
(ii) equal(s_1, s_2): returns true iff the strings s_1 and s_2 are equal.
(iii) join(s_1, s_2, s_3): creates the sequence $s_3 = s_1 s_2$ without destroying s_1 and s_2.
(iv) split(s_1, s_2, s_3, i), which creates two new sequences, $s_2 = a_1 \dots a_i$ and $s_3 = a_{i+1} \dots a_n$, without destroying s_1.

All presented operations work with string identifiers (ids). For example, join takes as its parameters the ids of strings s_1 and s_2, and returns the id of the newly created s_3; if some string already in the collection is equal to s_3, their ids will be equal. Importantly, the ids in the collection are positive integers and their maximum value after m operations is m^3.

Two solutions were presented by Melhorn et al.: one deterministic and one randomized — the latter with slightly better expected times for the operations (ii)–(iv) — we only make use here of the deterministic solution. The corresponding four time complexities, for the mth operation in the lifecycle of the structure, are: $O(1)$ for make_sequence, $O(\log m)$ for equal, and $O(\log n(\log m \log^* m + \log n))$ for join and split. After m operations the space used is $O(m \log n(\log^* m + \log n))$.

Our use of Melhorn et al.'s data structure is to store Parikh vectors, which we will simultaneously treat as both strings and arrays of integers. In the context of our application, all we need to be able to do is support increment and decrement of elements of these Parikh vectors. This can be simulated with split and join

operations allowed by Melhorn et al.'s string collection data structure, as we now explain.

Consider the successive windows of length ℓ shifted over sequence X. For the first window, we calculate the corresponding Parikh vector in $O(\sigma + \ell) = O(n)$ time and add it to the string collection using a series of make_sequence and join operations. For any following window, starting at some valid position $i + 1$, the respective Parikh vector for $X[i+1 \dots i+\ell]$ is calculated from the Parikh vector for $X[i \dots i+\ell-1]$ by incrementing $P(X[i \dots i+\ell-1])[X[i+\ell]]$ and decrementing $P(X[i \dots i+\ell-1])[X[i]]$. For presentation clarity, let us implement this operation in two stages: first going from $P(X[i \dots i+\ell-1])$ to $P(X[i \dots i+\ell])$ and then going from $P(X[i \dots i+\ell])$ to $P(X[i+1 \dots i+\ell])$. Let $s_1 = P(X[i \dots i+\ell-1])$. Using the dynamic collection of strings, the first transition between the Parikh vectors boils down to the following sequence of steps:

split(s_1, s_2, s_3, $X[i+\ell]$),
split(s_2, s_4, s_5, $X[i+\ell]-1$),
make_sequence(s_6, $s_1[X[i+\ell]]+1$),
$s_7 =$ join(s_4, s_6),
$s_8 =$ join(s_7, s_3).

The sequence labeled by s_8 corresponds to $P(X[i \dots i+\ell])$, hence the first stage is accomplished. The second stage is analogous so we omit it here. This procedure uses a constant number of split, join, and make_sequence operations and so has time complexity $O(\log n(\log m \log^* m + \log n))$.

We observe now that in $O(n \log n(\log m \log^* m + \log n))$ time (where $m = \Theta(n)$) we can obtain the Parikh vectors for all ℓ-grams from X, and the corresponding string ids in passing.

Our goal is to know the number of different string ids produced (which corresponds to the number of distinct Parikh vectors for the ℓ-length factors of X). With this in mind, at each step i we record the id produced in element i of an array of $n-\ell$ elements. Recall that the maximum id is upper-bounded by $\Theta(n^3)$, and so each id can be stored in $O(\log n)$ bits or $O(1)$ words of space. This in turn means the search tree requires $O(n)$ space overall. We then sort this array and then scan it to determine the number of distinct elements.

6 Concluding remarks

Finding the longest common Abelian factor is a recently posed problem, with a solution given in [1], achieving $O(\sigma n^2)$ worst-case time and needing $O(\sigma n)$ words of space. A significant weakness of that result is its space requirement, which may be unacceptable with a larger alphabet. We have improved this result in two ways.

The algorithm of Section 4 keeps the time complexity of Alatabbi et al.'s but reduces its space usage to $O(n)$. This is obtained by very simple means (the key component is LSD radix sort). Our second algorithm removes the dependency on σ and uses $O(n^2 \log^2 n \log^* n)$ time and $O(n \log^2 n)$ space. This algorithm is also

simple conceptually, exploiting a reduction of the problem to counting distinct Parikh vectors present in a string for different factor lengths.

We believe better algorithms for the LCAF problem are possible, and the discovery of one is the main open problem we leave; to be specific: is $O(n^2)$ time and $O(n)$ space possible? One obvious line of attack is to use word-level parallelism (in the word-RAM model) for Parikh vector comparisons. The anticipated speed-up factor however is only about $w/\log(n/\sigma)$, where w is the machine word size. Perhaps more interesting would be to attempt to share computations for different factor lengths to obtain faster algorithms. Hardness results, possibly following the 3SUM reduction for other Abelian problems by Amir et al. [2], would also be welcome. In another direction, we may be able to use rounding techniques described by Cicalese et al. [4] to trade off accuracy for time. We are currently working on sampling techniques that we hope can be combined with rounding to yield even faster algorithms.

Finally, we note that achieving $O(n^2)$ time and $O(n)$ space is possible if we are happy with answers that are sometimes incorrect. More precisely, we can use Karp-Rabin hashing in place of Melhorn et al.'s data structure in our algorithm (which is effectively acting as a rolling hash function). This gives a Monte Carlo algorithm that correctly computes the LCAF with high probability; and can be made Las Vegas fairly easily by applying techniques from [5]. We defer the details to the full version of this paper.

References

1. A. Alatabbi, C. S. Iliopoulos, A. Langiu, and M. S. Rahman. Algorithms for longest common abelian factors. *arXiv preprint arXiv:1503.00049*, 2015.
2. A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein. On hardness of jumbled indexing. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 8572, pages 114–125, 2014.
3. A. Apostolico, M. Crochemore, M. Farach-Colton, Z. Galil, and S. Muthukrishnan. 40 years of suffix trees. *Communications of the ACM*, 59(4):66–73, 2016.
4. F. Cicalese, T. Gagie, E. Giaquinta, E. S. Laber, Z. Lipták, R. Rizzi, and A. I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 56–63, 2013.
5. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. Lz77-based self-indexing with faster pattern matching. In *Proceedings of the 11th Latin American Theoretical Informatics Symposium (LATIN)*, LNCS 8392, pages 731–742, 2014.
6. L. C. K. Hui. Color set size problem with applications to string matching. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 644, pages 230–243. Springer, 1992.
7. T. Kociumaka, T. A. Starikovskaya, and H. W. Vildhøj. Sublinear space algorithms for the longest common substring problem. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA)*, LNCS 8737, pages 605–617. Springer, 2014.

8. K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.