Proceedings of the Prague Stringology Conference 2012

Edited by Jan Holub and Jan Žďárek

August 2012

Prague Stringology Club
http://www.stringology.org/

Proceedings of the Prague Stringology Conference 2012
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, Praha 6, 160 00, Czech Republic.
URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811
Printed by Česká technika – Nakladatelství ČVUT, Thákurova 550/1, Praha 6, 160 41, Czech Republic

ISBN 978-80-01-05095-8

Conference Organisation

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
Maxime Crochemore, *Co-chair*	(King's College London, United Kingdom)
František Franěk	(McMaster University, Canada)
Jan Holub, *Co-chair*	(Czech Technical University in Prague, Czech Republic)
Costas S. Iliopoulos	(King's College London, United Kingdom)
Shunsuke Inenaga	(Kyushu University, Japan)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Thierry Lecroq	(Université de Rouen, France)
Bořivoj Melichar, *Honorary chair*	(Czech Technical University in Prague, Czech Republic)
Yoan J. Pinzón	(Universidad Nacional de Colombia, Colombia)
Marie-France Sagot	(INRIA Rhône-Alpes, France)
William F. Smyth	(McMaster University, Canada)
Bruce W. Watson	(FASTAR Group (Stellenbosch University and University of Pretoria, South Africa))
Jan Žďárek	(Czech Technical University in Prague, Czech Republic)

Organizing Committee

Miroslav Balík, *Co-chair*	Jan Janoušek	Bořivoj Melichar
Jan Holub, *Co-chair*		Jan Žďárek

External Referees

Loek Cleophas	Arnaud Lefebvre	German Tischler
Peter Clote	Ludovic Mignot	Fritz Venter

Preface

The proceedings in your hands contains the papers presented in the Prague Stringology Conference 2012 (PSC 2012) which was devoted to 70th birthday of prof. Bořivoj Melichar, the founder of the Prague Stringology Club at the Czech Technical University in Prague, which organizes the event. The conference was held on August 27–28, 2012 and it focused on stringology and related topics. Stringology is a discipline concerned with algorithmic processing of strings and sequences.

The papers submitted were reviewed by the program committee. Ten papers were selected, based on originality and quality, as regular papers for presentations at the conference. This volume contains not only these selected papers but also an abstract of one invited talk "Correctness-by-construction in stringology".

The Prague Stringology Conference has a long tradition. PSC 2012 is the seventeenth event of the Prague Stringology Club. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2011 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on web pages of the Prague Stringology Club. Selected contributions were published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, and the International Journal of Foundations of Computer Science.

The Prague Stringology Club was founded in 1996 as a research group in the Czech Technical University in Prague. The goal of the Prague Stringology Club is to study algorithms on strings, sequences, and trees with emphasis on automata theory. The first event organized by the Prague Stringology Club was the workshop PSCW'96 featuring only a handful of invited talks. However, since PSCW'97 the papers and talks are selected by a rigorous peer review process. The objective is not only to present new results in stringology and related areas, but also to facilitate personal contacts among the people working on these problems. As a recognition of the conference, Elsevier B.V. decided to index the conference proceedings by Scopus collection. The main product derived from this collection is Scopus.com.

I would like to thank all those who had submitted papers for PSC 2012 as well as the reviewers. Special thanks go to all the members of the program committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC 2012. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2012

Jan Holub

Table of Contents

Correctness-by-Construction in Stringology

Bruce W. Watson

FASTAR Research Group
Stellenbosch University
South Africa
`bruce@fastar.org`

Correctness-by-construction (CbC) is an algorithm derivation technique in which the algorithm is co-developed with its correctness proof. Starting with a specification (most often as a pre- and post-condition), 'derivation steps' are made towards a final algorithm. Critically, each step in the derivation is a *correctness-preserving* one, meaning that the composition of the derivation steps *is* the correctness proof.

In this talk, I will present several stringological derivations to illustrate the usefulness of CbC – with a particular focus on *exploratory algorithmics*[1] (see [10] for an example of a new CbC-derived algorithm) and weak points of other algorithm derivations.

Correctness proofs in stringology algorithms (and in related fields such as compression and arbology) are particularly important for a few reasons:

– Many stringology problems arise in so-called *infrastructure software*, such as network routers, security, operating systems, compilers, computational linguistics, etc. All of these areas, are performance- and correctness-critical, with a low tolerance for bugs – unlike many user-level or web-applications.
– For many stringology algorithms, the *devil is in the details:* correctly defining, using and precomputing various lookup tables often proceeds via case analysis – a technique which is not always convincing or water-tight.
– The broad usefulness of these algorithms makes them ideal and central to many computing science curricula – where convincing correctness proofs are important.
– The multitude of stringology algorithms (take, for example, exact keyword pattern matching) is difficult to oversee (though several works successfully present the breadth of the field [7,1]) and taxonomies play an important role in bringing order.

For showing an algorithm's correctness, CbC has significant advantages over other approaches, namely:

– *Testing:* Edsger Dijkstra famously said "Testing shows the presence, not the absence of bugs". It follows that testing is a poor replacement for proper correctness proofs.
– *A postiori proof:* The majority of new algorithms are *presented* first and followed by a correctness proof. This usually leaves a large gap between the algorithm (how was it arrived at?) and the proof, or the proof remains just a sketch, where the correctness of some parts of the algorithm are left for the reader to work out.

[1] Exploratory algorithmics is the invention of new algorithms by exploring gaps and previously unexplored options amongst the existing algorithms for a field.

– *Automated proof:* Alongside the algorithm itself, a *model* of the algorithm is created; the model is then verified (using an automated theorem prover or a model-checker). This, of course, depends on a close correspondence between the algorithm and its model. Very few stringology derivations use this technique.

In CbC, at every derivation point there are often several possible derivation steps – which begs the question of how to choose the *right* step? Good derivations have an aspect of beauty and simplicity to them – properties which very often lead to the most efficient algorithms [3] – though writing down a good derivation requires practice and is an iterative process. Additionally, in many cases there are several possible 'next steps', making CbC an ideal technique for deriving entire *families* of algorithms. Such multiple-derivations can function both as a *taxonomy* (useful in teaching, for illustrating the commonalities and differences between related algorithms) and also for *exploratory algorithmics* in which new algorithms are invented [8,9].

CbC was 'invented' by Edsger Dijkstra in the late 1960's [2] with no small amount of input from his contemporaries such as Tony Hoare, Robert Floyd, Niklaus Wirth and Donald Knuth and also Dijkstra's colleagues in Eindhoven and Austin. Several Turing Awards (in particular Dijkstra's) were awarded for CbC-related research. David Gries and Carroll Morgan wrote two of the best follow-on text-books in the 1980's [4,6]. Despite the fact that some of those books are out of print, CbC remains alive and well as a successful and appropriate techniques for inventing and deriving new algorithms. The most recent book on this topic is by Kourie & Watson [5].

Acknowledgement: I would like to thank Nanette Watson-Saes and Derrick Kourie for proof-reading this extended abstract.

References

1. M. A. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific Publishing Company, 2003.
2. E. W. Dijkstra: *A Discipline of Programming*, Prentice Hall, 1976.
3. W. Feijen, A. van Gasteren, D. Gries, and J. Misra, eds., *Beauty is Our Business*, Springer-Verlag, 1990.
4. D. Gries: *The Science of Computer Programming*, Springer-Verlag, second ed., 1980.
5. D. G. Kourie and B. W. Watson: *The Correctness-by-Construction Approach to Programming*, Springer-Verlag, 2012.
6. C. Morgan: *Programming from Specifications*, Prentice Hall, second ed., 1998.
7. W. F. Smyth: *Computing Patterns in Strings*, Addison-Wesley, 2003.
8. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, Sept. 1995.
9. B. W. Watson: *Algorithms for Constructing Minimal Acyclic Deterministic Finite Automata*, PhD thesis, Department of Computer Science, University of Pretoria, South Africa, 2011.
10. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in Proceedings of the International Workshop on Combinatorial Algorithms (IWOCA), 2012.

Similarity Based Deduplication with Small Data Chunks

Lior Aronovich[1], Ron Asher[2], Danny Harnik[2], Michael Hirsch[2], Shmuel T. Klein[3], and Yair Toaff[2]

[1] IBM
Toronto, Canada
aronovic@ca.ibm.com

[2] IBM – Diligent
Tel Aviv, Israel
{ronasher,dannyh,hirschm,yairtoaff}@il.ibm.com

[3] Department of Computer Science
Bar Ilan University, Ramat Gan, Israel
tomi@cs.biu.ac.il

Abstract. Large backup and restore systems may have a petabyte or more data in their repository. Such systems are often compressed by means of deduplication techniques, that partition the input text into chunks and store recurring chunks only once. One of the approaches is to use hashing methods to store fingerprints for each data chunk, detecting *identical* chunks with very low probability for collisions. As alternative, it has been suggested to use *similarity* instead of identity based searches, which allows the definition of much larger chunks. This implies that the data structure needed to store the fingerprints is much smaller, so that such a system may be more scalable than systems built on the first approach.
This paper deals with an extension of the second approach to systems in which it is still preferred to use small chunks. We describe the design choices made during the development of what we call an approximate hash function, serving as the basic tool of the new suggested deduplication system and report on extensive tests performed on an variety of large input files.

Keywords: approximate hash scheme, deduplication, compression

1 Introduction and Motivation

Huge amounts of data have to be processed daily and the current trend suggests that these amounts will continue being ever-increasing in the foreseeable future. An efficient way to alleviate the problem is by using *deduplication*: large parts of the available data is copied again and again and forwarded without any change; the idea underlying a deduplication system is to locate repeated data and store only its first occurrence. Subsequent copies are replaced by pointers to the stored occurrence, which significantly reduces the storage requirements if the data is indeed repetitive [2].

Several approaches have been proposed to solve the problem, each concentrating on another aspect of the input characteristics. One of the approaches, based on hashing, can be schematically described as follows [11,13,14].

The available data is partitioned into parts called chunks C_i. These chunks can be of fixed or variable size, and the (average) size of a chunk can be small, say 4–8 KB, up to quite large, say, about 16 MB. A cryptographically strong hash function h is applied to these chunks, meaning that if $h(C_i) = h(C_j)$, it can be assumed, with very low error probability, that the chunks C_i and C_j are identical. The set S of different hash values, along with pointers to the corresponding chunks, is kept in a data structure D allowing fast access and easy update, typically a hash table or a B-tree. For each new chunk to be treated, its hash value is searched for in D, and if it appears there, one may assume that the given chunk is a duplicate. It is thus not stored again, rather, it is replaced by a pointer to its earlier occurrence. If the hash value is not in D, the given chunk is considered new, so it is stored and its hash value is adjoined to the set S.

The suggested methods mainly differ in the way they define the chunk boundaries, and in the suggested size of the chunks. The chunk size may indeed have a major impact on the performance: if it is too small, the number of different chunks may be so large as to jeopardize the whole approach, because the data structure D might not fit into RAM, so the system might not be scalable. On the other hand, if the chunk size is chosen too large, the probability of getting identical chunks decreases: many instances of chunks might exist, that could have been deduplicated had the chunk size been chosen smaller, but which, for the larger chunk size, have to be kept.

A possible solution to this chunk size dilemma has been suggested in [1]. The main idea there is to look for similar rather than identical chunks. If such a similar chunk is located, only the difference is recorded, which is generally much smaller than a full chunk. This allows the use of much larger chunks than in identity based systems.

For many applications, such as data backups and archiving, data is more fine-grained, and much better deduplication can be performed if one can use significantly smaller chunks. A simple generalization of the above system in which the chunk size would be reduced from 16 MB to 8 KB, that is, by a factor of 2000, without changing anything else in the design, would imply a 2000 fold increase of the size of the index, from 4 GB to about 8 TB. This cannot be assumed to fit into RAM in the near future. Moreover, keeping the definition of the notion of similarity and reducing the size of the chunks will lead to an increased number of collisions, which may invalidate the approach altogether.

The idea of the current work is to implement the required similarity by what we call an *approximate hash* scheme. This is an extension of the notion of locality-sensitive hashing introduced in [8]. The basic idea is that such an approximate hash function is not sensitive to “small” changes within the chunk, and yet behaves like other hash functions as far as the close to uniform distribution of its values is concerned. As a consequence, one can handle the set of approximate hash values as is usually done in hash applications (using a hash table, or storing the values in a B-Tree), but detect also similar, and not only identical chunks. If a given chunk undergoes a more extended, but still minor, update, its new hash value might be close to the original one, which suggests that in the case of a miss, the values stored in the vicinity of the given element in the hash table should be checked. Such vicinity searches are useless in a regular hash approach.

An approximate hash could be defined by a property that reminds the definition of a continuous function: let A and B be data chunks of fixed size, and let $d(x, y)$ be some distance function to be defined on the set of chunks; a hash function ah will be called an ε-approximate hash if

$$\exists\delta > 0 \quad d(A, B) < \delta \longrightarrow |ah(A) - ah(B)| < \varepsilon.$$

Note the difference with the common continuity definition, in which we would have $\forall\varepsilon\exists\delta$, implying that we can get function values as close as wanted (ε can tend to 0) if we start from close enough arguments. In our case, it would be exaggerated to impose such a property, and we can relax it to find two bounds δ and ε such that if the distance between chunks is bounded by the first, then the distance between the hash values of these chunks is bounded by the second, for reasonably chosen small values of ε.

Actually, even this definition could be too restrictive, and we should allow a small number of exceptions for certain extreme chunks. This leads to a probabilistic version of the above definition: a hash function ah will be called an ε-approximate hash with probability p if

$$\exists \delta > 0 \quad d(A,B) < \delta \longrightarrow Pr\left(|ah(A) - ah(B)| > \varepsilon\right) < 1 - p,$$

where the probability is taken over a uniform selection of the possible chunks A and B.

There are several possibilities to define the distance function d. A simple solution would be the Hamming distance, defined either on bits (number of 1 bits in A XOR B) or on characters (number of differing characters), but this requires the chunks to be of the same length. A more significant, yet more involved, function could be the edit distance: the minimal number of single character insert, delete and substitute operations needed to transform A into B.

The challenge is now to find such a function ah, giving a tradeoff between how well it can be adapted to reflect the approximate nature described above, and how long it takes to evaluate it. We should still bear in mind that one of the most basic requirements of a hash function is that it should not require too much CPU time.

The general algorithm for storing the repository will then be as follows. The number k of bits in the signature will be chosen in advance, and a hash table H with 2^k entries will be used as basic data structure. During the building process, each chunk C will be assigned its approximate hash value $ah(C)$, and the index, or address, of the chunk will be stored at $H[ah(C)]$, the entry in H indexed by the hash value of the chunk. If the location in the table is not free, it is overwritten. This may happen in case the new chunk is identical or very similar to a previously encountered chunk, in which case we prefer to store the address of the more recent chunk for potential later reference; but a collision may also be the result of two completely different chunks hashing to the same value, and then the pointer to the older chunk that has been overwritten will be lost.

In the next section, we describe the details leading to the design of the approximate hash function, and then report on extensive tests in Section 3, showing a noticeable improvement of the suggested method over identity based deduplication with small data chunks.

2 An approximate hashing function

Once it has been decided to base the system on approximate hashes according to the ideas stated above, the problem reduces to devising an appropriate function. This is the main thrust of the present suggestion.

Classical hashing functions have been studied for decades and many good solutions are known [6]. The major challenge in the design of an *approximate* hash function is finding the right balance between the following three competing criteria:

– Uniformity: the function should yield a distribution of values as close as possible to uniform, so as to minimize the number of collisions (false matches);
– Simplicity: the function should be easy and fast to calculate;
– Sensitivity: small changes in the chunk should not, or only slightly, affect the corresponding approximate hash value.

The first two are properties that are common to all hashing functions, the third one, sensitivity, is proper to the approximate version suggested herein. For standard hashing schemes, just the contrary is required: even very small changes in the chunk should lead to extensive changes in the hash value, otherwise the uniformity would be hurt. Some works dealing with similarity rather than identity can be found in [4,5,10]. Our approach is different and will be described below.

The value produced by a hash function is, in a certain sense, a summarization of the information contained in the data on which the function has been applied. This is reminiscent of similar functions, like the Fourier Transform with its many applications, or the Discrete Cosine Transform, used in JPEG image compression. Such transforms allow to recode the information of the given data into a different form, which may be more useful for certain applications, for example, being more compressible. Similarly, we would like to recode compactly much of the information contained within a given data chunk under the constraint that this recoding should be immune to small fluctuations.

This lead to the decision of using the *distribution* of the various characters that appear in the data as the basis for the suggested approximate hash. The data will be partitioned into relatively small chunks C of fixed or variable length, with (average) size of about 8–16 K. Each such chunk will be analyzed as to the distribution of the bytes forming it and their frequencies. We define the sequence of different bytes, ordered by their frequency of occurrence in the chunk, as the *c-spectrum* of C, and the corresponding sequence of frequencies as the *f-spectrum* of C. In addition, we consider also the sequence of different byte pairs, ordered by their frequency of occurrence in the chunk, and call it the *p-spectrum* of C. The suggested approximate hash function $ah(C)$ will be a combination of certain elements of these spectra. The reasoning behind the decision of relying on these distributions is that on the one hand, they usually behave like fingerprints, and it will be rare that essentially different chunks will exhibit the same distributions, but on the other hand, small perturbations in the data will often have no, or just a minor, impact on the corresponding spectra. This is the goal we wish to achieve in designing an approximate hash.

The size of the hash values will be fixed in advance, so as to exploit the space of the allocated hash table. For example, one could decide that the table will have about 4 billion entries, which corresponds to a hash value of 32 bits. A much larger hash value using $k > 32$ bits could be prohibitive, since the corresponding hash table would then have 2^k entries. On the other hand, a small value of k limits the number of elements of the spectra that can be chosen to be a part of the definition of the signature. To overcome this limitation, the chosen elements of the spectra, and more precisely, only a part of their bits, will be arranged appropriately by shifting them to the desired positions, and all these bit strings will be XORed. By using different indents for the different elements, the final value will not only depend on each of the building blocks, but also on their internal order. Figure 1 is a schematic representation of a possible layout of these elements. The columns correspond to the 32 bit positions, and each

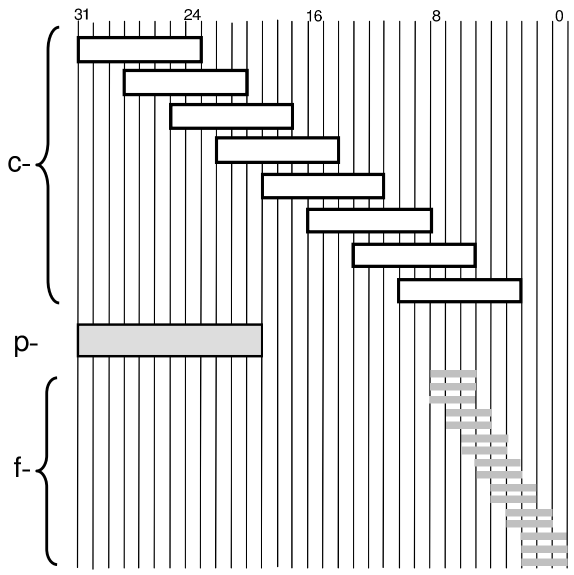


Figure 1. Schematic representation of the building blocks of a signature

rectangle stands for one of the elements, with the upper elements corresponding to the c-spectrum, the lowest elements corresponding to the f-spectrum, and the element in the middle corresponding to the p-spectrum, as detailed below. As can be seen, each bit position of the final signature is influenced by several elements.

We do not claim that the suggested layout is the best possible, not even for the sample data on which it has been tested. Rather, it is brought as an illustration of the ideas leading to its design. The specific values of the various parameters (lengths and shifts) shown in this example have been set empirically by iterating experiments to locally optimize the performance.

2.1 Using elements of the c-spectrum

Let $a_1, a_2, \ldots, a_n$ be the sequence of different bytes in the chunk, or, more precisely, the numerical value of these bytes, ordered by non-increasing frequency in the chunk. Ties are broken by sorting bytes with identical frequency by their numerical value. Let $f_1, f_2, \ldots, f_n$ be, respectively, the corresponding frequencies. The number n of different bytes in the chunk can vary between 1 (for chunks of identical bytes, like all zeroes or blanks) and $|C|$, the size of the chunk. As this size is mostly much larger than the maximum numerical value of a byte, one may assume that $1 \leq n \leq 256$.

A first attempt would be to consider each byte on its own as one of the building blocks of Figure 1, but this might result in a function that is too sensitive to noise. It will often happen that frequencies of certain bytes may be equal or very close. In such a case, a small perturbation might change the order of the bytes and yield a completely different hash value, contrarily to our goal of the approximate hash function being immune to small changes. To circumvent this problem, the a_i will be partitioned into *blocks*, gathering several bytes together and treating them symmetrically. The representation of all the elements in a block will be aligned with the same offset and will be XORed together, so that the internal order within the blocks may be arbitrary, since the XOR operation is commutative.

The sizes of the blocks should not be fixed in advance, but depend on the values themselves. Consider the sizes d_i of the *gaps* between the frequencies, $d_i = f_i - f_{i+1}$, for $i = 1, \ldots, n-1$. The boundaries between the blocks should be chosen according to the largest gaps, however, sorting according to d_i alone would strongly bias the definition of the gaps towards inducing blocks with single elements, since the largest gaps will tend to occur between the largest values. We should therefore normalize the size of the gaps by dividing by an appropriate weight. We chose harmonic weights $\frac{1}{i}$ for $i \geq 1$ according to Zipf's law [15]. The gaps are therefore sorted with respect to $i \times d_i = i \times (f_i - f_{i+1})$, which has the advantage of requiring only integer arithmetic.

For a given parameter ℓ, the $\ell - 1$ gaps with largest weights are chosen and the ℓ sets of consecutive elements delimited by the beginning of the sequence, these $\ell - 1$ gaps, and the end of the sequence, are defined as the blocks. Figure 2 is a schematic representation of an example partition into blocks with $\ell = 8$. The squares represent elements a_i, the arrows stand for weighted gaps $i\ (f_i - f_{i+1})$, and the numbers under the arrows are the indices of the weighted gaps in non-increasing order. In this example, the induced blocks would consist of 3, 1, 3, 2, 4,... bytes, respectively.

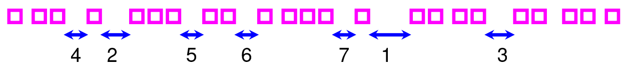


Figure 2. Schematic representation of the gaps

The number of elements forming the last block is limited, if necessary, to include at most a predetermined number of bytes, say 10, otherwise the speed of calculation could be hurt, and spurious bytes that appear possibly only once or twice in the chunk would have too strong of an influence. For the same reason, there are also lower bounds on the number of occurrences of a byte to be considered at all (for example, 15) and on the size d_i of a gap (for example, 5). If after these adjustments, the number of blocks in a given chunk is smaller than the selected value of ℓ, a different layout will be chosen that is adapted to the given number of blocks. In any case, one has to prepare layouts also for the possibility of having any number of blocks between 1 and ℓ, since certain extreme chunks may contain only a small number of different bytes.

Each block taken from the c-spectrum will be represented by a string of 8 bits, using the full representation of the corresponding bytes. The strings are depicted in Figure 1 as white rectangles. Each of these rectangles is shifted as indicated in Figure 1, where they are listed in order top down. That is, the first rectangle is shifted by 24 bits to the left (in fact, to get it left justified in the 32-bit layout), the next rectangle is shifted 21 bits, then 18, 15, 12, 9, 6 and 3 bits.

2.2 Using elements of the f-spectrum

The elements of the f-spectrum are incorporated into the signature independently from the partition into blocks of the corresponding bytes. For each frequency value, which can be an integer between 1 and $|C|$, consider first its standard binary representation (say, in 16 bits), and extend this string by 8 additional zeros to the right. We thus assign to each frequency f_i a 24-bit string F_i, for example, if $f_i = 5$, then F_i = 00000000 00000101 00000000. We further define D_i as the substring of

F_i of length m bits, for some predetermined small integer m, starting at the position immediately following the most significant 1-bit, for our case 00000000 000001**01** **0**0000000, the bits forming D_i for $m = 3$ appear bold faced. To give another example with a value of more than 8 bits, consider $f_i = 759$; 0000001**0** **11**110111 00000000 then displays both F_i and D_i. In the example of Figure 1, the elements included in the layout are the D_i, and the size m of all the elements is chosen as 3 bits. We experimented also with other values of m, from 1 to 8, but got better results with $m = 3$. The 8 bit padding allows values of m up to 8. The offsets of the chosen elements are as indicated, this time bottom up, with the largest frequency being depicted as the lowest element in the figure: 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 and 6. The idea behind this choice of bits is to select those with highest variability so as to get a broad spread of values, but to ignore, for the larger frequencies, the lowest bits, which are those most influenced by small fluctuations.

2.3 Using elements of the p-spectrum

Though much of the information of a chunk is already contained in the c- and f-spectrum, we decided to adjoin also elements of the p-spectrum and got empirical evidence that this improved the performance. When the maximal number ℓ of blocks could be used, a single element based on the p-spectrum was sufficient. The corresponding rectangle, depicted in the center of Figure 1, is of length 12 bits and will be placed left-justified in the layout. It is defined as follows: order the pairs by non-increasing frequencies and consider those indexed 5, 6, 7, 8 and 9 in this ordering. The reason for not choosing the most frequent pairs as we did for the individual bytes is that their distribution is much more biased, with the pairs (0,0) and (255,255) appearing as the most frequent in an overwhelming majority of the cases we tested. On the other side, there was already a great variability in the pairs in positions 5 to 9.

For each of the 5 pairs, the following bitstring is constructed. Given the 2 bytes $A = a_7 \cdots a_0$ and $B = b_7 \cdots b_0$, we rotate A cyclically to the left by 3 bits, and B cyclically to the right by 3 bits. The bytes are aligned so that the rightmost 4 bits of A overlap with the leftmost 4 bits of B, and then the strings are XORed. Formally, the 12 resulting bits are now

$$a_4,a_3,a_2,a_1,a_0 \oplus b_2, a_7 \oplus b_1, a_6 \oplus b_0, a_5 \oplus b_7,b_6,b_5,b_4,b_3,$$

where the $\oplus$ operator stands for bitwise XOR. Note that the most and least significant bits of both A and B are in the overlapping part, so if their distribution is biased, they have an additional chance to correct the bias by the additional XOR. This is important for special cases, for example, when the chunk only contains printable text. The representation of all the bytes would then start with the same one to three bits, which could have a negative effect on the uniformity we seek.

2.4 Putting it all together

Finally, all the elements of the layout are XORed, yielding a 32 bit string, representing a number between 0 and $2^{32} - 1$ that will act as the hash value of the given chunk C.

The geometry of the layout of the signature has been chosen on purpose as given in Figure 1, with the most frequent bytes being placed left-justified, thereby influencing the most significant (highest) bits, and the lowest elements of the f-spectrum appearing in the area influencing the least significant (lowest) bits. The intention was that in case of small fluctuations in the frequencies, the order of the most frequent characters might remain the same, so only some low order bits would change, yielding just a small difference in the signature values. Minor changes affecting even lower frequencies may go undetected, either because the corresponding frequencies are not among those chosen, or because the change is in the lower order bits that are not recorded in the signature.

3 Experimental Results

We performed a series of tests to assess the usefulness of the approach. A first concern was to verify that the proposed approximate hash indeed spreads its values evenly. Once this has been confirmed, we have to check that this uniformity does not come at the price of sensitivity, as it would for a standard hashing scheme. We thus checked the impact of the signature scheme in some artificial perturbation and clustering tests, described below. Finally, we bring examples of applying the whole deduplication process in comparison with an identity based approach.

As testbed, a subset of an Exchange database (EXC) of about 27 GB has been chosen, as well as the entire operating system of one of our computers (OS), a file of about 5 GB. The first set of tests was done with chunks of fixed length 8 K. These tests were then repeated for variable length sized chunks, the boundary of a chunk being defined by applying a simple Rabin-Karp rolling hash on the d rightmost bytes of the chunk under consideration. If this hash value equals some predefined constant c, the chunk is truncated after these d bytes; otherwise, the following byte is adjoined and the test with the rolling hash is repeated. In the test, $d = 25$, $c = 2718$ and the hash function is $RK(x) = x \bmod P$, where $P = 2^{48} - 257$ is a prime number. To avoid extreme values for the chunk lengths, a lower limit of 2 K and an upper limit of 64 K has been imposed. The average size of a chunk was around 12 K on our test databases.

	# blocks	Average Excess		St.Dev Excess		Avg # occ
expected		1/2		$1/\sqrt{12}$		
EXC	3,300,000	0.5050	1.0%	0.2991	3.6%	1.0033
OS	594,969	0.5085	1.7%	0.2858	-2%	1.0996

Table 1. Some statistics on the test databases and signatures

3.1 Uniformity

Table 1 summarizes some statistics about the test databases, the number of 8K blocks, the average signature value (normalized to the [0,1] range), the standard deviation of these normalized values, as well as the deviation from the expected results for a uniform distribution. As can be seen, the values are very close to the expected ones. On the EXC database, the chunk containing only zeros appeared 1756 times, but beside the corresponding signature, all the others appeared mostly only once, some appeared twice, etc. No signature appeared more than 45 times. The last column of Table 1 gives the average number of occurrences for each signature.

For a more precise evaluation, inspecting each individual bit, the graph of Figure 3 shows the probability of getting a 1-bit in each of the 32 positions of the signatures.

Note that these probabilities, for all bit positions, are very close to the expected value of 0.5 for a random distribution.

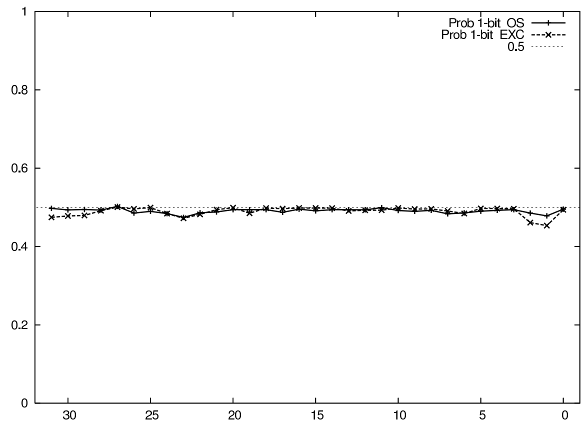


Figure 3. Bit distribution on example data

3.2 Perturbation tests

We now turn to observing the properties of the signature when introducing perturbations. Recall that the challenge was to reconcile two contradicting demands: on the one hand, the function is required, similarly to usual hash functions, to spread its values as much as possible, so as to minimize the number of collisions; on the other hand, we want small perturbations to yield only slight differences, if at all, in the corresponding signature values, a property one explicitly prohibits for classical hashing.

To simulate real life changes, the modified bytes did not get a random value, but rather another randomly chosen byte from within the same chunk was copied into the location to be modified. Thus the perturbations were introduced as follows: a random position i between 1 and $|C|$, the size of the chunk, was chosen, and the character from position $|C| - i + 1$ was copied to position i, overwriting the current one. The idea was to change the chunk slightly, but without introducing any new characters that are not already present in the chunk. Obviously, there is a small chance that this "perturbation" is in fact void (when overwriting a character by itself), but the corresponding probability is small enough so as not to bias the overall statistics. The signature function was then applied to the modified chunk and compared to the signature of the original chunk. In many cases, we got the same signature, meaning that changing a single byte in the chunk did not change the function, contrarily to what would be expected from a real hash function.

The above perturbation procedure has then been repeated, and the signature was reevaluated after $2, 3, \ldots, 10, 20, 30, \ldots, 100, 110$ byte changes. The changes were cumulative, that is, each test added one (or 10) more perturbations. Table 2 is a sample of some consecutive lines of the corresponding table.

One could define the distance between two signature values as the absolute value of their difference, reflecting the intention of the design of the signature layout to yield changes in the low order bits of the signature as result of small changes in the chunk. However, in the intended application to a deduplication system, one cannot afford too many search attempts in the vicinity of the hash value. More precisely, suppose

signature	1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90	100	110
1144762526	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2	12	12	12	12	12	12	12	12	12
127187251	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	14	14	14	14	14	14	14	14	14	14	14	14
4244827393	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	10	10	13	13	14	13	9	6	5
1818305692	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	14	17	17	15	18	18	18	18	18	20	20
1354737651	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	8	10	10	10	10	6	5	10	12	14
33724058	**0**	**0**	6	6	6	6	6	6	6	6	6	6	2	2	8	8	6	6	6	6
1392679006	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1
59007581	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	8	7	14	16	12	16	16	16	16
1343544922	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	11	0	7	7	7	11	0	0
1077804921	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
142372494	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2	2	2	2	2	2	2
1076507414	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2	2
583838910	**0**	1	2	2	2	2	2	2	2	3	3	9	9	3	0	6	6	6	6	6
2214783602	**0**	**0**	**0**	**0**	**0**	**0**	**0**	10	0	0	9	8	10	8	8	8	8	8	8	8
2217595617	1	1	1	1	1	0	0	1	1	1	0	6	3	2	2	1	1	4	3	3
2198134340	5	5	5	5	5	5	5	5	5	5	6	5	4	4	4	4	4	7	13	10
1073872964	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
3233873385	**0**	1	1	1	1	1	1	1	1	1	2	7	7	7	2	3	1	1	1	1
2155372916	**0**	**0**	8	8	8	8	8	8	8	2	0	2	2	12	12	4	2	2	2	2
4277376398	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	3	3	1	2	4	14	14	12	12	12
4264726240	**0**	**0**	**0**	**0**	**0**	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3
2248374342	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	7	7	7	13	15	16	8	8	8

Table 2. Hamming distance with original chunk after $1, 2, \ldots, 110$ perturbations

a chunk C is given. We would evaluate $ah(C)$ and check whether there is a pointer to a chunk D at the address $H[ah(C)]$ in the hash table. If so and indeed $D = C$, the newly arrived chunk C can be deduplicated by pointing to D. But if $D \neq C$, the intention was to look for pointers to chunks identical to C at the neighboring locations of the hash table. But each trial is costly, so the number of trials will have to be restricted. It might possibly only be reasonable to check at $H[ah(C)]$, $H[ah(C) + 1]$ and $H[ah(C) - 1]$. In that case, we can as well restrict ourselves to the *Hamming distance* between signatures, i.e., the number of differing bits in their standard binary representation, rather than their arithmetic difference. These are the values displayed in Table 2.

The first column gives the original signature (as a decimal number) before applying any perturbation, then in the column headed i is the Hamming distance between the original and the new signatures when i perturbations have been applied. Note that this distance is not always an increasing function of the number of perturbations, indicating that there might be quite a few cases in which the signature tends to "correct itself" when there are many changes; however, the overall trend is clearly increasing, as can be seen in the graphs below.

The plot in Figure 4 shows the average number of changed bits as a function of the number of perturbations, for both the Exchange and the OS databases. The average Hamming distance was between 0.3 and 5 to 6. There are slight differences between the databases, but the trend is the same.

Note that the distances, at least for the small number of perturbations, are quite low, and very often even zero, meaning that very small changes often yield the same signature as before. This is in sharp contrast with regular hashing schemes, for which the corresponding graph is expected to be a straight line (that is, independent of the number of changes as long as this number is > 0) at the level of about 16 (that is, about half of the bits are expected to change).

To verify this fact, we devised a control experiment in which a regular hash function (modulo $P = 2^{32} - 17 = 4,294,967,279$), was applied the same blocks, and then

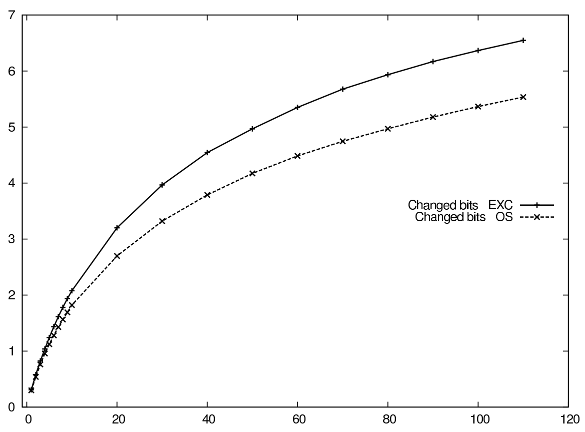


Figure 4. Average number of changed bits as function of the number of perturbations for the suggested signature

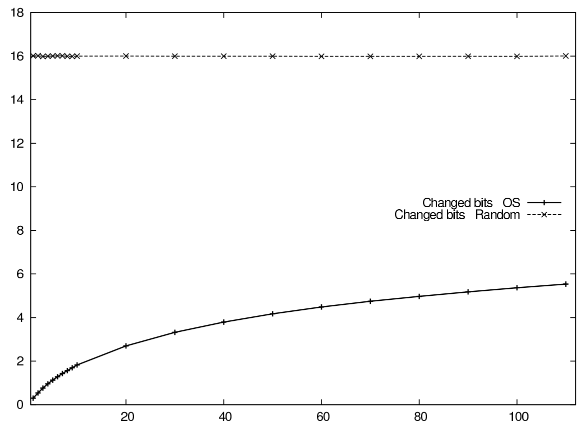


Figure 5. Average number of changed bits as function of the number of perturbations for a real hash function

performed the same perturbation tests as for our function, recording the Hamming distance between the original and perturbed signatures. Note that P is a prime number (actually the largest one fitting into our unsigned 32 bit signature). As expected, the number of changed bits was indeed around 16, as can be seen on the plot in Figure 5, more precisely, the average values were in the range from 15.988 to 16.018. Figure 5 also displays again the graph for the OS database, for comparison.

For our function, even if there are more than 100 bytes changed, this implies, at the average, only a change in about 5 to 6 bits. The resulting signature might thus be very different (depending on the position of those 6 changed bits), but the change is clearly not as radical as if a regular hash had be applied. In any case, this is just a noteworthy observation as in the intended application, there is no intention to look for similar chunks so far away.

To get a feeling on how far one can insert perturbations without yet changing the signature value, consider the plot in Figure 6, giving for each number i of perturbations, the probability of getting a non-zero value in the column headed i of the perturbation table. The plots are again given for both the Exchange and the OS databases. The probability of a non-zero value for a single perturbation is just about 0.06, and we see a clear ascending trend, reaching probability about 0.8 for

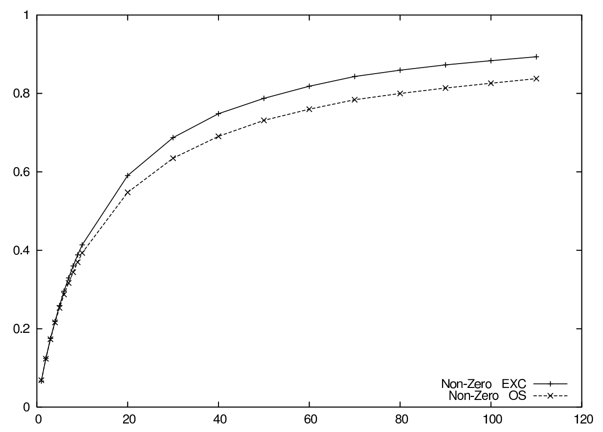


Figure 6. Probability of getting non-zero Hamming distance

more than 100 changes. For the control test with the real hashing function, we again got practically always non-zero values, more precisely, the probability for getting a non-zero for each of the columns was between 0.99986 and 1.00000.

3.3 Clustering

After testing that the proposed signature indeed has the properties required from an approximate hash function, that is: it gives a uniform spread, yet preserves locality in the sense that similar blocks give similar signatures, we turn now to a more general clustering test, which in fact checks the transpose of the above implication, that similar signatures also imply similar blocks.

For each of the tested databases, N centroïd chunks have been chosen (we used $N = 11$), so that they were mutually not similar. This is achieved by choosing the chunks in a random sequence, and checking for each new candidate that it is different enough from all the preceding chosen chunks in the sequence. X and Y are said to be different enough if $LD(X, Y) \geq T$, where T is some independently chosen threshold (we used 1000), and LD defines the Levenshtein distance [12].

Each of the centroïds is then used to generate a number M of perturbation chunks (we used $M = 10$), which are obtained by either changing a predetermined number K of bytes of the map to a random value, or by copying to each of these K bytes the value of another, randomly chosen, byte value from within the same chunk. The number of perturbations K has been chosen to vary from 2 to 1024, doubling in each step. Finally, the approximate hashing is applied to each of the generated chunks, and the whole set of $N \cdot M$ signatures is then sent to a clustering procedure, which partitions the set of signatures, and thereby the set of corresponding chunks, into subset of similar chunks. The number of hits, that is, correctly assigned correlations between a generated chunk and its generating centroïd, is recorded as a function of the number K of perturbations.

Three different alternatives have been considered to perform the clustering: the hierarchical Tree-method (repeatedly choosing the pair of closest chunks among the set of remaining subsets and dynamically updating the sets), K-means (minimizing the within-cluster sum of squares) [9], and simply checking the distance from every generated chunk to each of the centroïds and choosing that with minimal distance. The results were similar, with the first method consistently giving slightly better performance.

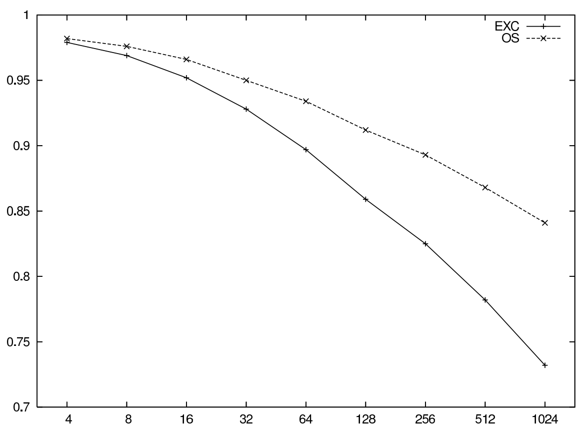


Figure 7. Probability of guessing the correct cluster as function of the number of perturbations

Each experiment was repeated 10 times and the values averaged. The results for our test databases of the hit ratio as function of the number of perturbations are displayed in Figure 7. As can be seen, the success rate is indeed decreasing with increasing K, and for a small number of perturbations, the number of successful assignments may be as large as 95%.

3.4 Comparison of similarity with identity

As has been mentioned earlier, the ultimate aim of these hash based systems is to perform deduplication. One approach is to use standard hashing, even with cryptographically strong hash functions that reduce the probability of false alarms to almost zero, but can thereby detect only identical chunks. The alternative suggested in this paper is the *approximate hash*, which could be able of locating also similar and not necessarily identical data chunks.

It might not be possible to quantify the relative improvement caused by shifting from a system based on identity to one based on similarity: the results will be extremely data dependent, based on the nature of the data and its repetitiveness. It obviously makes no sense to simulate the system's behavior on random data, as is done for many other applications, since truly random data is not compressible. On the other hand, also compressed files cannot be compressed even further, but they may be able to take advantage of deduplication, for example when several copies of such a file appear in the database.

We therefore decided, by lack of what could be agreed on as being "typical" data, to test the performance in tests on publicly available files and report the results just as examples, without claiming that these results are representative. Indeed, on different input data, the figures could be higher or lower, depending on the data at hand.

file name	size (MB)	identity	similarity	gain
`centos-5.3-i386-server`	2816	6.58	7.10	7.3%
`freebsd-6.4-i386`	949	3.88	4.02	3.5%
`fedora-fc6-i386`	265	5.84	6.20	5.8%

Table 3. Comparing identity with similarity based systems

distance	-5	-4	-3	-2	-1	0	1	2	3	4	5
probability	0.6	2.4	1.3	2.1	2.7	81.7	2.9	2.4	1.3	1.9	0.6

Table 4. Distribution of distances from the approximate hash value

The files we chose were images of virtual machines obtainable from the web, a sample of which is presented in Table 3 The sizes are given in MB, and the columns headed identity and similarity list the corresponding compression ratios. The compression ratio is defined as the size of the original file divided by the size of the compressed file. For identity, we used the SHA1 secure hash function [7] and second and subsequent copies of identical chunks were removed. For similarity, we used our approximate hashing scheme, and in case a similar chunk has been found, the new chunk was delta-encoded using VCDIFF [3]. Fixed length chunks of size 8 K have been used for both parts of the experiment. The final column lists the relative gain, in percent, of using similarity instead of identity.

Table 4 gives a more specific insight in the distribution of where the matching chunks have been located by our system. We checked first at $H[ah(C)]$, and if this entry did not contain a pointer to C, we also checked $H[ah(C) \pm i]$, for $i = 1, 2, \dots, 5$. On our example data, in the overwhelming majority of cases among those where the chunk could indeed be deduplicated, the pointer was found at $H[ah(C)]$ itself. But in 18% of the cases, it was found nearby. As could be expected, the probability of locating the chunk decreases with the distance from $ah(C)$, but interestingly, the decrease is not monotonic: the values for ± 4 are larger than for ± 3. Clearly, this is due to the fact that a difference of 4 means that only one bit is different in the signature, while for a difference of 3, there are two differing bits.

4 Conclusion

We have presented the main ideas leading to the design of a similarity rather than identity based deduplication system working with relatively small data chunks. Similarity has been explored earlier in this context [1], but the performance depended critically on the fact that the chunk size could be chosen large enough, in the MB range, which reduced the size of the required data structures. The current work is a first attempt to adapt the similarity approach also to systems in which a more fine grained resolution is required, with data chunks typically in the KB range.

The tests we performed suggest that the proposed approximate hash function indeed combines quite contradicting properties, like uniformity and sensitivity as required, though this can only be empirically tested on chosen examples, and not quantitatively checked in controlled statistical experiments. The scalability of the system will obviously depend on the amount of duplicate data it contains.

References

1. L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and T. S. Klein: The Design of a Similarity Based Deduplication System, *Proc. of the SYSTOR'09 Conference*, Haifa, (2009) 1–14.
2. D. R. Bobbarjung, D. Jagannathan, and C. Dubnicki: Improving duplicate elimination in storage systems, *ACM Transactions on Storage*, **2**(4) (2006) 424–448.
3. J. L. Bentley and M. Douglas McIlroy: Data Compression Using Long Common Strings, *Proc. Data Compression Conference*, Snowbird, Utah, (1999) 287–295.
4. A. Z. Broder: Identifying and Filtering Near-Duplicate Documents, *Proc. Combinatorial Pattern Matching Conference, CPM'00*, (2000) 1–10.
5. A. Z. Broder: On the resemblance and containment of documents, *Proc. of Compression and Complexity of Sequences*, IEEE Computer Society, (1997) 21–29.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest: *Introduction to Algorithms*, MIT Press, 1990.
7. N. Ferguson, B. Schneier, and T. Kohno: *Cryptography Engineering*, John Wiley & Sons, (2010).
8. P. Indyk and R. Motwani: Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality, *Proc. of the ACM Symposium on the Theory of Computing STOC'98*, (1998) 604–613.
9. T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu: An efficient K-means clustering algorithm: Analysis and implementation, *IEEE Trans. Pattern Analysis and Machine Intelligence* **24** (2002) 881–892.
10. U. Manber: Finding Similar Files in a Large File System, *Proc. of the USENIX Winter 1994 Technical Conference*, (1994) 17–21.
11. G. H. Moulton and S. B. Whitehill: Hash file system and method for use in a commonality factoring system, U.S. Pat. No. 6,704,730, issued March 9, 2004.
12. G. Navarro: A guided tour to approximate string matching, *ACM Computing Surveys* **33**(1) (2001) 31–88.
13. S. Quinlan and S. Dorward: Venti: A New Approach to Archival Storage, *Proc. of the USENIX Conference on File And Storage Technologies (FAST)*, (2002) 89–101.
14. B. Zhu, K. Li, and H. Patterson: Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, *Proc. of the USENIX Conference on File And Storage Technologies (FAST)*, (2008) 279-292.
15. G. K. Zipf: *The Psycho-Biology of Language*, Boston, Houghton (1935).

LZW Data Compression on Large Scale and Extreme Distributed Systems

Sergio De Agostino

Computer Science Department, Sapienza University, 00198 Rome, Italy

Abstract. Results on the parallel complexity of Lempel-Ziv data compression suggest that the sliding window method is more suitable than the LZW technique on shared memory parallel machines. When instead we address the practical goal of designing distributed algorithms with low communication cost, sliding window compression does not seem to guarantee robustness if we scale up the system. The possibility of implementing scalable heuristics is instead offered by LZW compression. In this paper we present two implementations of the LZW technique on a large scale and an extreme distributed system, respectively. They are both derived from a parallel approximation scheme of a bounded memory version of the sequential algorithm.

Keywords: compression, factorization, distributed system, scalability

1 Introduction

Lempel-Ziv compression [18], [19], [22] is based on string factorization. Two different factorization processes exist with no memory constraints. With the first one (LZ1) [19], each factor is independent from the others since it extends by one character the longest match with a substring to its left in the input string (sliding window compression). With the second one (LZ2 or LZW) [22], each factor is instead the extension by one character of the longest match with one of the previous factors. This computational difference implies that while sliding window compression has efficient parallel algorithms [6], [11], [12], [3], LZW compression (a practical implementation of the LZ2 method [21]) is hard to parallelize [5]. This difference is maintained when the most effective bounded memory versions of Lempel-Ziv compression are considered [15], [2]. There are several heuristics for limiting the work-space of the LZW compression procedure in the literature. The most effective is the "least recently used" strategy (LRU). Hardness results inside Steve Cook's class (SC) have been proved for this approach [15], implying the likeliness of the non-inclusion of the LZW-LRU compression method in Nick Pippenger's class (NC). Completeness results in SC have also been obtained for a relaxed version of the LRU strategy (RLRU) [15]. RLRU was shown to be as effective as LRU in [8], [9]. Therefore, RLRU is the most efficient among the bounded memory versions of LZW compression. A simpler heuristic which is still effective is the RESTART strategy. Differently from LRU and RLRU, LZW-RESTART is parallelizable [15]. Moreover, parallel decompression is possible (this is true also for the unbounded memory version) [6], [7], [11], [12].

When we address the practical goal of designing distributed algorithms with low communication cost sliding window compression does not seem to guarantee robustness when we scale up the system [10], [11], [12], [2]. The possibility of implementing scalable heuristics is instead offered by LZW-RESTART compression [11], [12], [13]. Traditionally, the scale of a system is considered large when the number of nodes has the order of magnitude of a thousand. Modern distributed systems may nowadays consist of hundreds of thousands of nodes, pushing scalability well beyond traditional scenarios (extreme distributed systems). In this paper we present two implementations of the LZW technique on a large scale and an extreme distributed system, respectively. They are both derived from a parallel approximation scheme of the bounded memory version of the sequential algorithm presented in [13]. The approach for extreme distributed systems could be applied to arbitrarily smaller scale systems as well, but the alternative implementation we propose is simpler.

In Section 2 we describe Lempel-Ziv data compression while the bounded memory versions are given in Section 3. Section 4 briefly describes past work on the study of the parallel complexity of Lempel-Ziv methods since it is somehow consistent with the practical results on the distributed implementation of LZW compression shown in Section 5. Conclusion and future work are given in Section 6.

2 Lempel-Ziv Data Compression

Lempel-Ziv compression is a dictionary-based technique. In fact, the factors of the string are substituted by *pointers* to copies stored in a dictionary. LZ1 (LZ2) compression is also called the sliding (dynamic) dictionary method.

2.1 LZ1 Compression

Given an alphabet A and a string S in A^* the LZ1 factorization of S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where the factor f_i is the shortest substring which does not occur previously in the prefix $f_1 f_2 \cdots f_i$ for $1 \leq i \leq k$. With such a factorization, the encoding of each factor leaves one character uncompressed. To avoid this, a different factorization was introduced (LZSS factorization) where f_i is the longest match with a substring occurring in the prefix $f_1 f_2 \cdots f_i$ if $f_i \neq \lambda$, otherwise f_i is the alphabet character next to $f_1 f_2 \cdots f_{i-1}$ [20]. f_i is encoded by the pointer $q_i = (d_i, \ell_i)$, where d_i is the displacement back to the copy of the factor and ℓ_i is the length of the factor (LZSS compression). If $d_i = 0$, l_i is the alphabet character. In other words the dictionary is defined by a window sliding its right end over the input string, that is, it comprises all the substrings of the prefix read so far in the computation. It follows that the dictionary is both *prefix* and *suffix* since all the prefixes and suffixes of a dictionary element are dictionary elements.

2.2 LZ2 Compression

The LZ2 factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where the factor f_i is the shortest substring which is different from one of the previous factors. As for LZ1 the encoding of each factor leaves one character uncompressed. To avoid this a different factorization was introduced (LZW factorization) where each factor f_i is the longest match with the concatenation of a previous factor and the next character [21]. f_i is encoded by a pointer q_i to such concatenation (LZW compression). LZ2 and LZW compression can be implemented in real time by storing the dictionary with a trie data structure. Differently from LZ1 and LZSS, the dictionary is only prefix.

2.3 Greedy versus Optimal Factorization

The pointer encoding the factor f_i has a size increasing with the index i. This means that the lower one is the number of factors for a string of a given length the better is the compression. The factorizations described in the previous subsections are produced by greedy algorithms. The question is whether the greedy approach is always optimal, that is, if we relax the assumption that each factor is the longest match can we do better than greedy? The answer is negative with suffix dictionaries as for LZ1 or LZSS compression. On the other hand, the greedy approach is not optimal for LZ2 or LZW compression. However, the optimal approach is NP-complete [16] and the greedy algorithm approximates with an $O(n^{\frac{1}{4}})$ multiplicative factor the optimal solution [14].

3 Bounded Size Dictionary Compression

The factorization processes described in the previous section are such that the number of different factors (that is, the dictionary size) grows with the string length. In practical implementations instead the dictionary size is bounded by a constant and the pointers have equal size. While for sliding window compression this can be simply obtained by bounding the match and window lengths (therefore, the left end of the window slides as well), for the LZW compression the dictionary elements are removed by using a deletion heuristic.

3.1 The Deletion Heuristics

Let $d + \alpha$ be the cardinality of the fixed size dictionary where α is the cardinality of the alphabet. With the FREEZE deletion heuristic, there is a first phase of the factorization process where the dictionary is filled up and "frozen". Afterwards, the factorization continues in a "static" way using the factors of the frozen dictionary. In other words, the LZW factorization of a string S using the FREEZE deletion heuristic is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the longest match with the concatenation of a previous factor f_j, with $j \leq d$, and the next character.

The shortcoming of the FREEZE heuristic is that after processing the string for a while the dictionary often becomes obsolete. A more sophisticated deletion heuristic is RESTART, which monitors the compression ratio achieved on the portion of the input string read so far and, when it starts deteriorating, restarts the factorization process. Let $f_1 f_2 \cdots f_j \cdots f_i \cdots f_k$ be such a factorization with j the highest index less than i where the restart operation happens. Then, f_j is an alphabet character and f_i is the longest match with the concatenation of a previous factor f_h, with $h \geq j$, and the next character (the restart operation removes all the elements from the dictionary but the alphabet characters). This heuristic is used by the Unix command "compress" since it has a good compression effectiveness and it is easy to implement. Usually, the dictionary performs well in a static way on a block long enough to learn another dictionary of the same size. This is what is done by the SWAP heuristic. When the other dictionary is filled, they swap their roles on the successive block.

The best deletion heuristic is the LRU (last recently used) strategy. The LRU deletion heuristic removes elements from the dictionay in a "continuous" way by deleting at each step of the factorization the least recently used factor, which is not a proper prefix of another one. In [15] a relaxed version (RLRU) was introduced. RLRU partitions the dictionary in p equivalence classes, so that all the elements in each class are considered to have the same "age" for the LRU strategy. RLRU turns out to be as good as LRU even when p is equal to 2 [8], [9]. Since RLRU removes an arbitrary element from the equivalence class with the "older" elements, the two classes (when p is equal to 2) can be implemented with a couple of stacks, which makes RLRU slightly easier to implement than LRU in addition to be more space efficient. SWAP is the best heuristic among the "discrete" ones.

3.2 Compression with Finite Windows

As mentioned at the beginning of this section, bounded size dictionary compression can also be obtained by sliding a fixed length window and by bounding the match length. The window length is usually several thousands kilobytes. The compression tools of the Zip family, as the Unix command "gzip" for example, use a window size of at least 32 K.

3.3 Greedy versus Optimal Factorization

Greedy factorization is optimal for compression with finite windows since the dictionary is suffix. With LZW compression, after we fill up the dictionary using the FREEZE, RESTART or SWAP heuristic, the greedy factorization we compute with such dictionary is not optimal since the dictionary is not suffix. However, there is an optimal semi-greedy factorization which is computed by the procedure of figure 1 [17], [4]. At each step, we select a factor such that the longest match in the next position with a dictionary element ends to the rightest. Since the dictionary is prefix, the factorization is optimal. However, greedy factorizations are very close to optimal in practice even if they approximate the optimal solution with a multiplicative factor equal to the maximum match length in the worst case.

```
$j$:=0; $i$:=0
repeat forever
    for $k = j + 1$ to $i + 1$ compute
        $h(k)$: $x_k ... x_{h(k)}$ is the longest match in the $k^{th}$ position
    let $k'$ be such that $h(k')$ is maximum
    $x_j ... x_{k'-1}$ is a factor of the parsing; $j := k'$; $i := h(k')$
```

Figure 1. The semi-greedy factorization procedure

4 Lempel-Ziv Compression on a Parallel System

LZSS (or LZ1) compression can be efficiently parallelized on a PRAM EREW [6], [2], [3], that is, a parallel machine where processors access a shared memory without reading and writing conflicts. On the other hand, LZW (or LZ2) compression is P-complete [5] and, therefore, hard to parallelize. Decompression, instead, is parallelizable for both methods [7]. The asymmetry of the pair encoder/decoder between LZ1 and LZ2 follows from the fact that the hardness results of the LZ2/LZW encoder depend on the factorization process rather than on the coding itself.

As far as bounded size dictionary compression is concerned, the "parallel computation thesis" claims that sequential work space and parallel running time have the same order of magnitude giving theoretical underpinning to the realization of parallel algorithms for LZW compression using deletion heuristic. However, the thesis concerns unbounded parallelism and a practical requirement for the design of a parallel algorithm is a limited number of processors. A stronger statement is that sequential logarithmic work space corresponds to parallel logarithmic running time with a polynomial number of processors. Therefore, a fixed size dictionary implies a parallel algorithm for LZW compression satisfying these constraints. Realistically, the satisfaction of these requirements is a necessary but not a sufficient condition for a practical parallel algorithm since the number of processors should be linear. The SC^k-hardness and SC^k-completeness of LZ2 compression using, respectively, the LRU and RLRU deletion heuristics and a dictionary of polylogarithmic size show that it is unlikely to have a parallel complexity involving reasonable multiplicative constants [15]. In conclusion, the only practical LZW compression algorithm for a shared memory parallel system is the one using the FREEZE, RESTART or SWAP deletion heuristics. Unfortunately, the SWAP heuristic does not seem to have a parallel decoder. Since the FREEZE heuristic is not very effective in terms of compression, RESTART is a good candidate for an efficient parallel implementation of the pair encoder/decoder even on a distributed system. We will see these arguments more in detail in the next section.

5 Lempel-Ziv Compression on a Distributed System

Distributed systems have two types of complexity, the interprocessor communication and the input-output mechanism. While the input/output issue is inherent to any parallel algorithm and has standard solutions, the communication cost of the computational phase after the distribution of the data among the processors and before the output of the final result is obviously algorithm-dependent. So, we need to limit the interprocessor communication and involve more local computation to design a practical algorithm. The simplest model for this phase is, of course, a simple array of processors with no interconnections and, therefore, no communication cost. Such array of processors could be a set of neighbors linked directly to a central node (from which they receive blocks of the input) to form a so called *star* network (a rooted tree of height 1). In an *extended* star each node adjacent to the central one has a set of leaf neighbors (a rooted tree of height 2). Such extension is useful in practice when we scale up the system.

For every integer k greater than 1 there is an $\mathrm{O}(kw)$ time, $\mathrm{O}(n/kw)$ processors distributed algorithm factorizing an input string S with a cost which approximates the cost of the LZSS factorization within the multiplicative factor $(k+m-1)/k$, where n, m and w are the lengths of the input string, the longest factor and the window respectively [2]. As far as LZW compression is concerned, if we use a RESTART deletion heuristic clearing out the dictionary every ℓ characters of the input string we can trivially parallelize the factorization process with an $\mathrm{O}(\ell)$ time, $\mathrm{O}(n/\ell)$ processors distributed algorithm. This could also be done with the LRU or SWAP deletion heuristic. However, with the RESTART deletion heuristic scalable compression and decompression algorithms are possible on a tree architecture. The parallel encoder, after a dictionary is filled for each block of length ℓ, produces a factorization of S with a cost approximating the cost of the optimal factorization within the multiplicative factor $(k+1)/k$ in $O(km)$ time with $O(n/km)$ processors [13]. These algorithms provide approximation schemes for the corresponding factorization problems since the approximation factors converge to 1 when km and kw converge to ℓ and to n, respectively. In the following subsections, we first discuss sliding window compression and then propose two improved new versions of the LZW distributed algorithm suitable on large scale and extreme distributed systems.

5.1 Sliding Window Compression on a Distributed System

We simply apply in parallel sliding window compression to blocks of length kw. It follows that the algorithm requires $O(kw)$ time with n/kw processors and the approximation factor is $(k+m-1))/k$ with respect to any parsing. In fact, the number of factors of an optimal (greedy) factorization on a block is at least kw/m while the number of factors of the factorization produced by the scheme is at most $(k-1)w/m+w$. As shown in figure 2, the boundary might cut a factor (sequence of plus signs) and the length w of the initial full size window of the block (sequence of w's) is the upper bound to the factors produced by the scheme in it. Yet, the factor cut by the boundary might be followed by another factor (sequence of x's) which covers the remaining part of the initial window. If this second factor has a suffix to the right of the window, this suffix must be a factor of the sliding dictionary defined by it (dotted line) and the multiplicative approximation factor follows.

```
           +++++(+++)xxxxxxxxxxx
 ——————————————/——————————————————————
                wwwwwwwwwww
                             .....
```

Figure 2. The making of the surplus factors

Making the order of magnitude of the block length greater than the one of the window length largely beats the worst case bound on realistic data. Since the compression tools of the Zip family use a window size of at least 32 K, the block length in our parallel implementation should be about 300 K and the file size should be about one third of the number of processors in megabytes. Therefore, the approximation scheme is suitable only for a small scale system unless the file size is very large.

5.2 LZW Compression on a Distributed System

LZW compression was originally presented with a dictionary of size 2^{12}, clearing out the dictionary as soon as it is filled up [21]. The Unix command "compress" employs a dictionary of size 2^{16} and works with the RESTART deletion heuristic. The block length needed to fill up a dictionary of this size is approximately 300 K.

As previously mentioned, the SWAP heuristic is the best deletion heuristic among the discrete ones. After a dictionary is filled up on a block of 300 K, the SWAP heuristic shows that we can use it efficiently on a successive block of about the same dimension where a second dictionary is learned. A distributed compression algorithm employing the SWAP heuristic learns a different dictionary on every block of 300 K of a partitioned string (the first block is compressed while the dictionary is learned). For the other blocks, block i is compressed statically in a second phase using the dictionary learned during the first phase on block $i-1$. But, unfortunately, the decoder is not parallelizable since the dictionary to decompress block i is not available until the previous blocks have been decompressed. On the other hand, with RESTART we can work on a block of 600 K where the second half of it it is compressed statically. We wish to speed up this second phase though, since LZW compression must be kept more efficient than sliding window compression. In fact, it is well-known that sliding window compression is more effective but slower. If both methods are applied to a block of 300 K, but LZW has a second static phase to execute on a block of about the same length it would no longer have the advantage of being faster. We show how to speed up this second phase on a very simple tree architecture as the extended star in time $O(km)$ with $O(n/km)$ processors.

During the input phase, the central node broadcasts a block of length 600 K to each adjacent processor. Then, for each block the corresponding processor broadcasts to the adjacent leaves a sub-block of length $m(k+2)$ in the suffix of length 300 K, except for the first one and the last one which are $m(k+1)$ long. Each sub-block overlaps on m characters with the adjacent sub-block to the left and to the right, respectively (obviously, the first one overlaps only to the right and the last one only to the left). Every processor stores a dictionary initially set to comprise only the alphabet characters.

The first phase of the computation is executed by processors adjacent to the central node. The prefix of length 300 K of each block is compressed while learning the dictionary. At each step of the LZW factorization process, each of these processors sends the current factor to the adjacent leaves. They all adds such factor to their own dictionary. After compressing the prefix of length 300 K of each block, all the leaves have a dictionary stored which has been learned by their parents during such compression phase.

Let us call a *boundary match* a factor covering positions of two adjacent sub-blocks stored by leaf processors. Then, the leaf processors execute the following algorithm to compress the suffix of length 300 K of each block:

– for each block, every corresponding leaf processor but the one associated with the last sub-block computes the boundary match between its sub-block and the next one ending furthest to the right, if any;

– each leaf processor computes the optimal factorization from the beginning of its sub-block to the beginning of the boundary match on the right boundary of its sub-block (or the end of its sub-block if there is no boundary match).

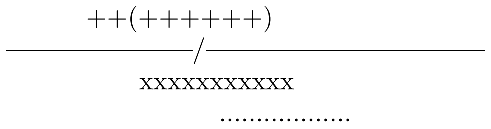


Figure 3. The making of a surplus factor

Stopping the factorization of each sub-block at the beginning of the right boundary match might cause the making of a surplus factor, which determines the approximation factor $(k+1)/k$ with respect to any factorization. In fact, as it is shown in figure

3, the factor in front of the right boundary match (sequence of x's) might be extended to be a boundary match itself (sequence of plus signs) and to cover the first position of the factor after the boundary (dotted line).

In [1], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorizarion is about the same as the sequential one. Results were presented for static prefix dictionary compression but they are valid for dynamic compression using the LZW technique with the RESTART deletion heuristic. In fact, experiments were proposed compressing similar files in a collection using a dictionary learned from one of them. This is true even if the second step is greedy, since greedy is very close to optimal in practice. Moreover, with the greedy approach it is enough to use a simple trie data structure for the dictionary rather than the modified suffix tree data structure of [17] needed to implement the semi-greedy factorization in real time. Therefore, after computing the boundary matches the second part of the parallel approximation scheme can be substituted by the following procedure:

– each leaf processor computes the static greedy factorization from the end of the boundary match on the left boundary of its sub-block to the beginning of the boundary match on the right boundary.

Considering that typically the average match length is 10, one processor can compress down to 100 bytes independently. Then compressing 300 K involves a number of processors up to 3000 for each block. It follows that with a file size of several megabytes or more, the system scale has a greater order of magnitude than the standard large scale parameter making the implementation suitable for an extreme distributed system. We wish to point out that the computation of the boundary matches is very relevant for the compression effectiveness when an extreme distributed system is employed since the sub-block length becomes much less than 1 K.

With standard large scale systems the sub-block length is several kilobytes with just a few megabytes to compress and the approach using boundary matches is too conservative for the static phase. In fact, a partition of the second half of the block does not effect on the compression effectiveness unless the sub-blocks are very small since the process is static. In conclusion, we can propose a further simplification of the algorithm for standard small, medium and large scale distributed systems.

Let $p_0 \cdots p_n$ be the processors of a distributed system with an extended star topology. p_0 is the central node of the extended star network and $p_1 \cdots p_m$ are its neighbors. For $1 \leq i \leq m$ and $t = (n - m)/m$ let the processors $p_{m+(i-1)t+1} \cdots p_{m+it}$ be the neighbors of processor i.

$B_1 \cdots B_m$ is the sequence of blocks of length 600 K partitioning the input file. Denote with B_i^1 and B_i^2 the two halves of B_i for $1 \leq i \leq m$. Divide B_i^2 into t sub-blocks of equal length.

The input phase of this simpler algorithm distributes for each block the first half and the sub-blocks of the second half in the following way:

– broadcast B_i^1 to processor p_i for $1 \leq i \leq m$

– broadcast the j-th sub-block of B_i^2 to processor $p_{m+(i-1)t+j}$ for $1 \leq i \leq m$ and $1 \leq j \leq t$

Then, the computational phase is:

in parallel for $1 \leq i \leq m$

– processor p_i applies LZW compression to its block, sending the current factor to its neigbors at each step of the factorization

– the neighbors of processor p_i compress their blocks statically using the dictionary received from p_i with a greedy factorization

5.3 Decompression

To decode the compressed files on a distributed system, it is enough to use a special mark occurring in the sequence of pointers each time the coding of a block ends. The input phase distributes the subsequences of pointers coding each block among the processors. If the file is encoded by an LZW compressor implemented with one of the two procedures described in the previous section, a second special mark indicates for each block the end of the coding of a sub-block. The coding of the first half of each block is stored in one of the neighbors of the central node while the coding of the sub-blocks are stored into the corresponding leaves. The first half of each block is decoded by one processor to learn the corresponding dictionary. Each decoded factor is sent to the corresponding leaves during the process, so that the leaves can rebuild the dictionary themselves. Then, the dictionary is used by the leaves to decode the sub-blocks of the second half.

6 Conclusion

We presented an approach to the parallel implementation of LZW data compression which is suitable for small and large scale distributed systems. Some blocks are compressed independently providing information for a second phase where the remaining portions of the input string are encoded in parallel with a higher granularity. In order to push scalability beyond what is traditionally considered a large scale system a more involved approach distributes overlapping sub-blocks of these remaining portions to compute boundary matches. These boundary matches are relevant to maintain the compression effectiveness on a so-called extreme distributed system. We wish to implement these ideas on real systems with the appropriate architecture to experiment how the communication cost effects on the speed-up. If we have a relatively small scale system available, the approach with no bounadary matches can be used. Moreover, if the system has an architecture with a simple star topology rather than an extended one we could still experiment on a file with size between 500 K and one megabyte. During the input phase, the central node brodcasts the sub-blocks of the second half of the file to the neighbors. Then, it applies LZW compression to the first half providing the dictionary to the other nodes for the compression of the second half. The parallel running time of this implementation could be compared with the sequential time of the sliding compression method applied to each of the two halves of the file (the higher one would be considered). In this way, it can be seen how the running times of the two parallel implementations relate to each other.

References

1. D. Belinskaya, S. D. Agostino, and J. A. Storer: *Near optimal compression with respect to a static dictionary on a practical massively parallel architecture*, in Proceedings IEEE Data Compression Conference, 1996, pp. 172–181.
2. L. Cinque, S. DeAgostino, and L. Lombardi: *Scalability and communication in parallel low-complexity lossless compression.* Mathematics in Computer Science, 3 2010, pp. 391–406.
3. M. Crochemore and W. Rytter: *Efficient parallel algorithms to test square-freeness and factorize strings.* Information Processing Letters, 38 1991, pp. 57–60.
4. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, 2003.
5. S. DeAgostino: *P-complete problems in data compression.* Theoretical Computer Science, 127 1994, pp. 181–186.
6. S. DeAgostino: *Parallelism and dictionary-based data compression.* Information Sciences, 135 2001, pp. 43–56.
7. S. DeAgostino: *Almost work-optimal PRAM EREW decoders of LZ-compressed text.* Parallel Processing Letters, 14 2004, pp. 351–359.
8. S. DeAgostino: *Bounded size dictionary compression: Relaxing the LRU deletion heuristic*, in Proceedings Prague Stringology Conference, 2005, pp. 135–142.
9. S. DeAgostino: *Bounded size dictionary compression: Relaxing the LRU deletion heuristic.* International Journal of Foundations of Computer Science, 17 2006, pp. 1273–1280.
10. S. DeAgostino: *Parallel implementations of dictionary text compression without communication*, 2009.
11. S. DeAgostino: *Lempel-Ziv data compression on parallel and distributed systems*, in Proceedings Data Compression, Communications and Processing Conference, 2011, pp. 193–202.
12. S. DeAgostino: *Lempel-Ziv data compression on parallel and distributed systems.* Algorithms, 4 2011, pp. 183–199.
13. S. DeAgostino: *LZW versus sliding window compression on a distributed system: Robustness and communication*, in Proceedings INFOCOMP, 2011, pp. 125–130.
14. S. DeAgostino and R. Silvestri: *A worst case analisys of the LZ2 compression algorithm.* Information and Computation, 139 1997, pp. 258–268.
15. S. DeAgostino and R. Silvestri: *Bounded size dictionary compression:* SC^k*-completeness and* nc *algorithms.* Information and Computation, 180 2003, pp. 101–112.
16. S. DeAgostino and J. A. Storer: *On-line versus off-line computation for dynamic text compression.* Information Processing Letters, 59 1996, pp. 169–174.
17. A. Hartman and M. Rodeh: *Optimal parsing of strings*, 1985.
18. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 22 1976, pp. 75–81.
19. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
20. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of ACM, 29 1982, pp. 928–951.
21. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
22. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.

Failure Deterministic Finite Automata

Derrick G. Kourie[1], Bruce W. Watson[2], Loek Cleophas[1,3], and Fritz Venter[1]

[1] University of Pretoria
[2] Stellenbosch University
[3] Eindhoven University of Technology
{dkourie,bruce,loek,fritz}@fastar.org

Abstract. Inspired by failure functions found in classical pattern matching algorithms, a failure deterministic finite automaton (FDFA) is defined as a formalism to recognise a regular language. An algorithm, based on formal concept analysis, is proposed for deriving from a given deterministic finite automaton (DFA) a language-equivalent FDFA. The FDFA's transition diagram has fewer arcs than that of the DFA. A small modification to the classical DFA's algorithm for recognising language elements yields a corresponding algorithm for an FDFA.

Keywords: failure arcs, DFA, formal concept analysis

1 Introduction

It is well-known that there is a mapping between deterministic finite automata (DFAs) and regular languages. Let $\mathcal{L}(\mathcal{D}) \subseteq \Sigma^*$ denote the regular language associated with DFA $\mathcal{D}$, the DFA being defined on an alphabet Σ and having δ as its transition function. The transition function maps a state / symbol pair to a new state, i.e. $\delta(q, a) = p$ where $q, p \in Q$, the DFA's set of states, and $a \in \Sigma$.

Given an arbitrary finite-length string $x \in \Sigma^*$, there is a classical algorithm to test whether $x \in \mathcal{L}(\mathcal{D})$. The algorithm uses δ to transition from state to state as it processes x on a character by character basis. It starts from the DFA's start state and terminates once all characters in x have been processed. Only if a final state has been reached at termination does the algorithm affirm that $x \in \mathcal{L}(\mathcal{D})$.

Such an algorithm takes time $\mathcal{O}(|x|)$, assuming that $\delta(q, a)$ is computed in constant time. It uses $\mathcal{O}(|\Sigma| \times |Q|)$ space, as δ has to be stored. Applications of the algorithm vary widely, with the underlying DFA possibly involving millions of states and transitions. Consequently, research efforts have been directed at improving on the algorithm's space or time efficiency. Examples include DFA minimisation [22], hard-coding and cache manipulation [10], various automata transformations [6,3], various strategies for storing sparse matrices [21,8], and other strategies to reduce representation sizes [5].

Here we focus on improving on space efficiency by relying on *failure* DFAs (FDFAs). The formalism derives from the failure functions found in classical pattern matching algorithms [1,11,4]. Recall, for example, the Aho-Corasick algorithm [1] which takes a finite set of patterns and identifies all their occurrences in a text. One version uses a DFA, while a second version uses a trie DFA [9] with a so-called failure function. The latter version removes arcs that do not contribute to the definition of patterns, replacing them judiciously with arcs derived from a failure function. The result is a trie DFA, decorated by various failure arcs. The standard acceptance algorithm is adapted to use this automaton. The total number of trie and failure arcs is significantly less than the number of arcs in the DFA version of the algorithm.

Benchmarks reported in [22] suggest that the gain in space efficiency comes at the cost of about 20% reduction in processing speed.

In addition to their use in the Knuth-Morris-Pratt and Aho-Corasick keyword pattern matching algorithms, there has been some work in broadening the use of failure functions, including:

- Kowaltowski, Lucchesi and Stolfi [16] present failure functions (and algorithms for computing them) in the restricted case of *acyclic automata* – especially as used in various natural language processing applications, such as spell-checking.
- Mohri [18] presents algorithms for the restricted case of constructing a failure function and manipulating $\mathcal{D}$ such that the resulting FDFA accepts language $\Sigma^* \cdot \mathcal{L}(\mathcal{D})$. The resulting compact representation is primarily useful in pattern matching for the language $\mathcal{D}$ *somewhere* in an input string x, as the Σ^* matches the prefix of x before the match.
- Crochemore and Hancart [4] illustrate how failure arc placement can sometimes be further optimised, but they do not give a general construction algorithm for deriving an FDFA from a DFA.

Our work takes up the failure arc idea and generalises it. Below we describe an ordered approach to deriving a language-equivalent FDFA from *any complete* DFA. This generalisation brings several issues to the fore.

The starting point to address these issues is the provision of a formal definition of an FDFA and its associated language. In terms of this definition, a DFA can be viewed as a degenerate FDFA. The right language of an FDFA state is recursively defined, and this provides a formal definition of an FDFA's language. Starting with the DFA (seen as a degenerate FDFA), we then show how to build incrementally a sequence of language-equivalent FDFAs. At each increment a set of arcs is replaced with a single failure arc while preserving the right languages of all states involved in such a transformation. As a consequence, the language recognised by FDFAs produced from transformation to the next remains invariant.

The matter of which set of arcs to select for transformation at each next iteration step is non-trivial. In general, there will be many possibilities, different selections leading to different FDFAs. One of the complications is that so-called divergent cycles of failure arcs have to be avoided (although non-divergent cycles may be tolerated).

To ensure that all candidate arcs for transformation are identified, we turn to formal concept analysis (FCA) – a domain of study in which a so-called (formal) concept lattice identifies clusters of objects that share common attributes [2]. We show how information about a complete DFA can be encapsulated in what we call a state/out-transition concept lattice. The state/out-transition lattice isolates arc sets that could potentially be replaced by failure arcs, and the arc redundancy measure is used to prioritise which sets to first select for such replacement. In this sense, we approximate a greedy algorithm. We also indicate how to proceed in order to render the algorithm a strictly greedy one, at some cost to the algorithm's efficiency. Since the greediness does not guarantee optimality, finding an efficient algorithm for deriving an arc-minimal language-preserving FDFA remains an open problem.

In summary, then, Section 2 provides the necessary formal material about FDFAs, while Section 3 introduces the reader to the FCA theory about concept lattices that is needed in this paper. Section 4 then shows how to build a state/out-transition concept lattice from a DFA. It also provides an algorithm which uses such a lattice to derive a language equivalent FDFA. Because the resulting FDFA retains all the original DFA states, the algorithm is characterised as a DFA-homomorphic algorithm. In Section 5 we point to additional work on this theme that is currently on our agenda. This includes algorithms under development which introduce failure arcs to new states derived directly from the lattice. Because of this, these may be described as lattice-homomorphic algorithms. However, full elaboration of these algorithms will be provided in subsequent research contributions.

2 Failure Deterministic Finite Automata

In defining and discussing an FDFA below, we rely on the following conventions and general notation.

- Where convenient, a function will be regarded as a set of pairs, the first element being from its domain and the second from its range. A function which is not guaranteed to be total but may be, is called a *possibly partial* function.
- The domain of any function f is denoted by $\operatorname{dom} f$. If $q \notin \operatorname{dom} f$ for the possibly partial function f, then this is denoted by $f(q) = \bot$.
- A DFA denoted by $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ is considered, where Q is the DFA's set of states; Σ is its alphabet; δ is the transition function mapping state / symbol pairs to a new state; s is the start state; and F is the set of final states.
- We use $\Sigma_q = \{a : \delta(q, a) \neq \bot\}$ to denote symbols labeling out-transitions of state q, and $\not\Sigma_q$ for $\Sigma \setminus \Sigma_q$. A complete DFA (sometimes called a total DFA, because δ is a total function) is characterised by the fact that $\Sigma_q = \Sigma$ for all $q \in Q$. Note that any DFA can easily be converted to a language-equivalent complete DFA by simply introducing an arc to a common sink state for every state q and symbol a such that $\delta(q, a) = \bot$.
- We will also use the function $\mathit{head} : \Sigma^+ \to \Sigma$ where $\mathit{head}(av) = a$; and the function $\mathit{tail} : \Sigma^+ \to \Sigma^*$ where $\mathit{tail}(av) = v$.
- We define the extended transition function $\delta^* : Q \times \Sigma^* \to Q$ by $\delta^*(q, w) = q$ if $w = \varepsilon$ and by $\delta^*(q, w) = \delta^*(\delta(q, \mathit{head}(w)), \mathit{tail}(w))$ otherwise.
- Given δ^*, the language of $\mathcal{D}$ is defined by $\mathcal{L}(\mathcal{D}) = \{w \mid \delta^*(s, w) \in F\}$.
- If $L \subseteq \Sigma^*$ and $u \in \Sigma$ then $u \cdot L$ denotes the prefixing of all elements in L by the symbol u, i.e. $u \cdot L = \{uw : w \in L\}$. Of course, $u \cdot \emptyset = \emptyset$.

Definition 1 (FDFA). *$\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ is an FDFA if $\mathfrak{f} : Q \to Q$ is a possibly partial function and $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ is a DFA.*

We shall call $\mathcal{D}$ the *embedded DFA* of $\mathcal{F}$ and $\mathfrak{f}$ the *failure function* of $\mathcal{F}$. If $q \in \operatorname{dom} \mathfrak{f}$, then q is called a *failure state.*

Definition 2 (Right language of an FDFA's state). *The right language of state q in FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$, denoted by $\overrightarrow{\mathcal{L}}(\mathcal{F}, q)$, is defined as the smallest language such that $\overrightarrow{\mathcal{L}}(\mathcal{F}, q) = \overrightarrow{\mathcal{L}}_\delta(\mathcal{F}, q) \cup \overrightarrow{\mathcal{L}}_\mathfrak{f}(\mathcal{F}, q)$, where*

$$\overrightarrow{\mathcal{L}}_\delta(\mathcal{F}, q) = \left(\bigcup_{b \in \Sigma_q} b \cdot \overrightarrow{\mathcal{L}}(\mathcal{F}, \delta(q, b)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

$$\overrightarrow{\mathcal{L}}_\mathfrak{f}(\mathcal{F}, q) = \begin{cases} \overrightarrow{\mathcal{L}}(\mathcal{F}, \mathfrak{f}(q)) \cap (\not\Sigma_q \Sigma^*) & \text{if } \mathfrak{f}(q) \neq \bot \\ \emptyset & \text{otherwise} \end{cases}$$

Thus, the right language of an FDFA in state q, written $\overrightarrow{\mathcal{L}}(\mathcal{F}, q)$, consists of three components: (1) all strings that can be generated from that state by making a conventional DFA transition to the next state on one of the out-transition symbols in Σ_q; (2) ε if q is final; and (3) those words in $\overrightarrow{\mathcal{L}}(\mathcal{F}, \mathfrak{f}(q))$ (the right language of the next state as determined by the failure function at q) that begin with a symbol *not* in Σ_q, because any word beginning with a symbol in Σ_q would already have caused a conventional DFA transition from q.

(Such a recursive definition of right language is well-formed. The above definition essentially gives rise to a finite set of equations with variables $\overrightarrow{\mathcal{L}}(\mathcal{F}, q)$, $\overrightarrow{\mathcal{L}}_\delta(\mathcal{F}, q)$ or $\overrightarrow{\mathcal{L}}_\mathfrak{f}(\mathcal{F}, q)$ (for all states q) on the left-hand side. All of those equations are either right-linear or chain-rules, and Gaussian elimination/substitution can be used to partially solve them, leaving a limited number of self- or mutually-recursive equations. Those equations are solvable (as a regular language) using Arden's lemma [20, Lemma 2.9, page 100]. See [20, Section 4.3.1, page 133] for a detailed example resembling this one.)

Definition 3 (Language of an FDFA). *The language of an FDFA $\mathcal{F}$ is denoted by $\mathcal{L}(\mathcal{F})$ and is defined as $\overrightarrow{\mathcal{L}}(\mathcal{F}, s)$, where s denotes the start state of $\mathcal{F}$.*

Definition 4 (FDFA equivalence). *An FDFA $\mathcal{D}$ (which may possibly be a DFA) is said to be* (language) equivalent *to an FDFA $\mathcal{F}$ iff $\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{D})$. This will be denoted by $\mathcal{F} \equiv \mathcal{D}$.*

Clearly, the embedded DFA of an FDFA is not, in general, equivalent to the FDFA, but it is in the degenerate case, i.e. when $\mathfrak{f} = \emptyset$. As previously noted, a DFA can therefore be seen as a special case of an FDFA – it is an FDFA that has a degenerate failure function.

Note that for a given FDFA, there could be many equivalent DFAs and vice versa. It is well known that the regular languages partition the set of DFAs into equivalence classes. Thus, each regular language $\mathcal{R}$ defines a class $\mathcal{E}_\mathcal{D}(\mathcal{R}) = \{\mathcal{D} \mid \mathcal{D} \text{ is a DFA} \wedge \mathcal{L}(\mathcal{D}) = \mathcal{R}\}$ that is disjoint from every other such class. Similarly, the regular languages also induce equivalence classes of FDFAs so that for regular language $\mathcal{R}$ there is a unique and partitioning set of FDFAs $\mathcal{E}_\mathcal{F}(\mathcal{R}) = \{\mathcal{F} \mid \mathcal{F} \text{ is an FDFA} \wedge \mathcal{L}(\mathcal{F}) = \mathcal{R}\}$. Since every DFA is a degenerate FDFA, $\mathcal{E}_\mathcal{D}(\mathcal{R}) \subseteq \mathcal{E}_\mathcal{F}(\mathcal{R})$.

The algorithm proposed in Section 4 may be thought of as starting off with $\mathcal{D} \in \mathcal{E}_\mathcal{D}(\mathcal{R})$ and deriving a sequence of $\mathcal{F}_i \in \mathcal{E}_\mathcal{F}(\mathcal{R})$, terminating when there are no further opportunities for removing elements of δ while adding elements of $\mathfrak{f}$.

The FDFA $\mathcal{F}$ produced by that algorithm can be used for recognising whether a string x is a member of $\mathcal{L}(\mathcal{F})$. Algorithm 1 shows how this can be done. It assumes FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ is given.

In this text, the Guarded Command Language (GCL) is used to specify algorithms. This minimalist and easy to use specification language was invented by Dijkstra [7] and remains widely in use because of its conciseness and precision [14].

We rely on GCL's multiple guarded command format for the loop. In this form, the loop comprises of two guarded commands of the form $G \rightarrow S$ where G is a boolean expression and S is a command. All guards are evaluated at each iteration, and a statement is non-deterministically selected among those whose guards evaluate to **true**. If no guard evaluates to **true**, the loop terminates[1].

[1] **cand** and **cor** stand for "conditional and" and "conditional or" respectively, i.e. the equivalent of the short circuit operators && and || in C++, Java, etc.

The virtue of this multiple guarded command loop format is that it highlights the symmetry with standard DFA acceptance. The standard algorithm is identical to Algorithm 1, but with the second loop guard absent.

Algorithm 1 (Test for string membership of an FDFA's language)

```
$y, q := x, s;$
{ ***Invariant:*** *y is untested and the current state is q* }
do $(y \neq \varepsilon)$ cand $(\delta(q, head(y)) \neq \bot) \rightarrow q, y := \delta(q, head(y)), tail(y)$
[] $(y \neq \varepsilon)$ cand $((\delta(q, head(y)) = \bot) \wedge (\mathfrak{f}(q) \neq \bot)) \rightarrow q := \mathfrak{f}(q)$
od;
{ $((y = \varepsilon)$ cor $((\delta(q, head(y)) = \bot) \wedge (\mathfrak{f}(q) = \bot)))$ }
$accept := (y = \varepsilon) \wedge (q \in F)$
{ post $(accept \Leftrightarrow x \in \mathcal{L}(\mathcal{F}))$ }
```

However, Algorithm 1 embodies a potential complication that does not arise in its DFA counterpart. The presence of cycles in the failure function could lead to complications. In order to understand the meaning and consequences of such cycles, we begin by defining the notion of a failure path.

Definition 5 (Failure path and failure alphabet). *A sequence of FDFA states, $\langle p_0, p_1, \ldots, p_n\rangle$ of length $n > 0$ is called a* failure path *from p_0 to p_n, written $p_0 \overset{\mathfrak{f}}{\rightsquigarrow} p_n$, iff $\forall i \in [0, n) : \mathfrak{f}(p_i) = p_{i+1}$. For such a failure path, $\Sigma_{p_0 \overset{\mathfrak{f}}{\rightsquigarrow} p_n} = \not\Sigma_{p_0} \cap \not\Sigma_{p_1} \cap \cdots \cap \not\Sigma_{p_{n-1}}$ is its* failure alphabet*.*

Where convenient, $p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_j$ will be used as a predicate to assert that the FDFA under consideration has a failure path from state p_i to state p_j.

In Algorithm 1, the transition which occurs on symbols in Σ_q is determined by δ, and if q is a failure state then the transition to occur on symbols in $\not\Sigma_q$ is determined by the failure function $\mathfrak{f}$. The failure alphabet of a failure path is therefore the set of symbols, each of which is guaranteed to cause failure transitions from the start of the failure path to its end. This insight becomes important in distinguishing between failure paths that form cycles. We shall simply call a failure path that forms a cycle a *failure cycle* and designate it by $p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i$, where p_i is any state in the cycle.

Definition 6 (Divergent failure cycle). *A failure cycle $p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i$ is* divergent *iff $\Sigma_{p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i} \neq \emptyset$.*

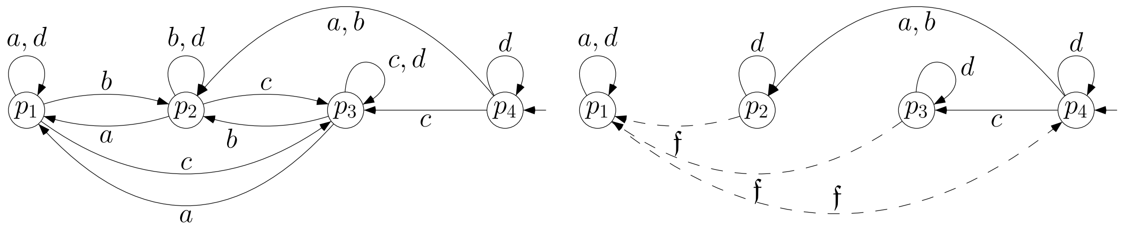
The term *divergent* is inspired by its use in process algebras. In that domain, a divergent concurrent system is one that is trapped into a cycle of non-productive state changes [19]. A divergent failure cycle in an FDFA would cause analogous behaviour in Algorithm 1. If the algorithm is examining symbol a in state p_i, where $a \in \Sigma_{p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i}$, then the algorithm will cycle non-productively through the divergent failure cycle without ever consuming symbol a. Clearly, therefore, it is advisable to avoid divergent failure cycles when constructing an FDFA. On the other hand, cycles which are not divergent (i.e. where $\Sigma_{p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i} = \emptyset$) are harmless, since it is guaranteed that at some state in the cycle, a symbol will *eventually* be consumed.

Definition 1 of an FDFA is as general as possible. It does not preclude failure cycles, whether or not they are divergent. It allows for useless states and transitions, including useless failure transitions. For example, a failure arc from state q where $\Sigma_q = \Sigma$ serves no purpose, but is not prohibited in the FDFA definition. We do not consider such cases in detail here, but ensure that they are avoided in the FDFA construction algorithm to be described.

Note in passing that the failure function description in [4] also allows for failure arcs in a general DFA setting, but no algorithm for constructing FDFAs in general is presented, and all failure arc cycles are prohibited there, even if they are not divergent. From what has been discussed above, this would seem to be an overly strict requirement.

Algorithm 1 operates in $\mathcal{O}(|w|)$ time in the best case, but in the worst case it has to traverse the path of an entire failure cycle before having a symbol of w consumed. Since the longest possible non-divergent cycle is $|Q|-1$, the algorithm's worst case performance is described by $\mathcal{O}(|w| \times (|Q|-1))$. The corresponding DFA string membership algorithm operates in $\mathcal{O}(|w|)$.

However, there is a potential savings in arc storage if an FDFA is used instead of a DFA. For example, consider the DFA depicted in Figure 1a. It has a total of sixteen arcs. (Doubly labelled arcs are counted twice, because storage is required to represent each transition.) The FDFA in Figure 1b is language equivalent to the DFA



(a) Initial DFA: $|\delta| = 16, |\mathfrak{f}| = 0$

(b) FDFA after two iterations of Algorithm 2: $|\delta| = 8, |\mathfrak{f}| = 3$

Figure 1: Initial DFA and an equivalent FDFA. All states are considered final

in Figure 1a. The FDFA has only eight arcs, and three failure transitions (represented by dashed arcs). This saving in arcs is possible because a conventional DFA sometimes contains redundancies, i.e. it may have transitions to the same state from several destinations, all on the same symbol[2]. For example, in Figure 1a, all states make a transition to state p_1 on a, the transition from p_4 on a being an exception. All states transition to state p_2 on b, and all states transition to state p_3 on c.

The FDFA in the Figure 1b is designed to handle transitions that are unique at each state, and to *fail over* to another state if the transition to be made on a set of symbols is shared with other states. For example, in state p_2, a transition on d is determined locally, yet on all other symbols, a failure transition is made to p_1, since on those symbols the behaviour from the states is the same.

[2] DFA *minimization* relies on such redundancy, but only works in case of right language *equality* between states, vs. *containment* in the FDFA case.

Thus, to recognise the string *abca*, the following DFA transitions are made in Figure 1a

$$p_4 \xrightarrow{a} p_2 \xrightarrow{b} p_2 \xrightarrow{c} p_3 \xrightarrow{a} p_1$$

However, in the case of the FDFA in Figure 1b the transitions made are as follows

$$p_4 \xrightarrow{a} p_2 \xrightarrow{\mathfrak{f}} p_1 \xrightarrow{\mathfrak{f}} p_4 \xrightarrow{b} p_2 \xrightarrow{\mathfrak{f}} p_1 \xrightarrow{\mathfrak{f}} p_4 \xrightarrow{c} p_3 \xrightarrow{\mathfrak{f}} p_1 \xrightarrow{a} p_1$$

An FDFA therefore needs at most $\mathcal{O}(|Q| \times (\Sigma + 1))$ to store δ and $\mathfrak{f}$. However, the actual storage will be decreased from this worst case estimate to the extent that δ can be minimized when constructing the FDFA. The challenge taken up here, therefore, is to derive from a DFA (seen here as a degenerate FDFA) say $\mathcal{F}' = (Q', \Sigma, \delta', \emptyset, s', F')$, an equivalent FDFA, say $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, s, F)$ such that $|\delta'| - (|\delta| + |\mathfrak{f}|)$ is as large as possible. Because of the benchmarking results reported in [22], we conjecture that the time penalty will be about 20%.

For the purposes of the algorithm, we assume $\mathcal{F}'$ to be a complete DFA, i.e. for every state q, $\Sigma_q = \Sigma$. Furthermore, we regard the various states as constants (i.e. $Q = Q', s = s'$ and $F = F'$). The algorithm thus preserves the originating DFA's shape, and will, for this reason, be called a DFA-homomorphic algorithm. In the algorithm δ and $\mathfrak{f}$ are variables whose values change from their initial values δ' and $\mathfrak{f}'$. The algorithm also ensures that at every step the right language of every state remains unchanged.

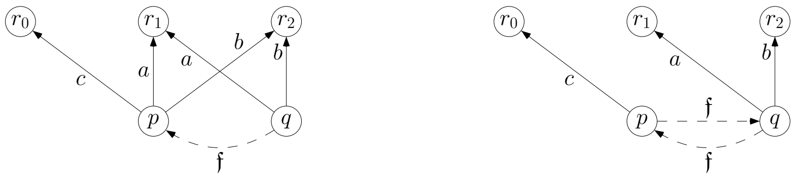
A theorem which relies on a predicate $\mathrm{FailPred}(P, q, X)$ indicates conditions under which the right language is preserved. The predicate is defined as follows.

Definition 7 (FailPred(P, q, X)). *For $P \cup \{q\} \subseteq Q$ and $X \subseteq \Sigma$, FailPred(P, q, X) is defined by*

$$\begin{aligned}
&\forall p \in P : (\\
&\qquad (\Sigma_p = \Sigma) && (1)\\
&\qquad \wedge\ (\mathfrak{f}(p) = \bot) && (2)\\
&\qquad \wedge\ (\forall (a \in X) : (\delta(p, a) = \delta(q, a))) && (3)\\
&\qquad \wedge\ (q \overset{\mathfrak{f}}{\rightsquigarrow} p \Rightarrow (\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap X = \emptyset)) && (4)\\
&\qquad)
\end{aligned}$$

A scenario in which this predicate holds is sketched in Figure 2a, where it is assumed that $P = \{p\}$, $\Sigma = \{a, b, c\}$ and $X = \{a, b\}$. Notice that the scenario in the figure complies with the first three conjuncts of Definition 7, i.e. $(\Sigma_p = \Sigma)$, complying with conjunct 1; $(\forall a \in X : (\delta(p, a) = \delta(q, a))$, complying with conjunct 3 ; and $(\mathfrak{f}(p) = \bot)$, complying with conjunct 2. Furthermore, the figure shows that $\mathfrak{f}(q) = p$, and thus $q \overset{\mathfrak{f}}{\rightsquigarrow} p$. Clearly, $\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} = \not\Sigma_q = \{c\}$ and since $\{c\} \cap X = \emptyset$, conjunct 4 holds as well. Thus, Figure 2a depicts a scenario in which $\mathrm{FailPred}(P, q, X)$ holds.

Figure 2b shows the result of removing the a and b transitions from p, and providing a failure transition from p to q. Note that this can be done without disturbing the right languages of any of the states in the figures. Note also that because conjunct 4 holds, we can be sure that a divergent cycle has not been created. Theorem 8 generalises these observations.



(a) Before (b) After

Figure 2: Theorem 8 applied, where $X = \{a, b\}$

Theorem 8 (A transformation that preserves right languages and does not introduce any failure cycle). *Let $\mathcal{F}$ be an FDFA such that $P \cup \{q\} \subseteq Q$, $X \subseteq \Sigma$ and FailPred(P, q, X) holds. Then the following transformation on each $p \in P$ leaves the right languages of all states unchanged and does not introduce any failure cycle:*

> *Delete from δ all transitions from p on each symbol in X, and add a failure arc from p to q.*

Applying such a transformation to FDFA $\mathcal{F}'$ results in an FDFA $\mathcal{F}$ such that $\mathcal{F}' \equiv \mathcal{F}$, in which $|\delta|$ has decreased by $|X|$ and $|\mathfrak{f}|$ has increased by 1. Theorem 8 may be applied repeatedly, as long as P, q and X satisfying the definition above can be found. In Section 4 we will show how formal concept lattices, introduced in the next section, can be used to identify such P, q and X.

3 State / Out-Transition Formal Concept Lattices

A *formal concept lattice* can be defined in a domain of discourse consisting of a set of objects, and a set of attributes that the various objects possess. In such a domain, a *concept* is considered to be a pair of two sets: a set of objects, the concept's *extent*; and a set of attributes, the concept's *intent*. All objects in the concept's extent have in common all and only the attributes in the intent. Furthermore, the extent is maximal over the objects: there may not be any object outside of the concept's extent which also possesses all the attributes in the intent.

In the theory known as formal concept analysis, such concepts are considered to be partially ordered: if c_i and c_j are two arbitrary concepts in the domain, and if $ext(c)$ denotes concept c's extent, then $c_i \leq c_j \Leftrightarrow ext(c_i) \subseteq ext(c_j)$. Equality holds if and only if $i = j$. Furthermore, it can be shown that there is a duality in the role of objects and attributes, such that if $int(c)$ denotes concept c's intent, then $c_i \leq c_j \Leftrightarrow int(c_j) \subseteq int(c_i)$. The relationship between objects and attributes in a given domain can be presented as a cross table known as a *context*. An example is shown in Table 1. The rows represent the objects $p_1, \ldots, p_4$ and the columns represent attributes designated $\langle a, p_1\rangle$, $\langle a, p_2\rangle$, $\langle b, p_2\rangle$, $\ldots$, $\langle d, p_4\rangle$. (We discuss the reason for these rather strange attributes later.) An entry in a cell indicates that the relevant object has the indicated attributes. E.g. object p_4 has attributes $\{\langle a, p_2\rangle, \langle b, p_2\rangle, \langle c, p_3\rangle, \langle d, p_4\rangle\}$.

It can be shown that the partial ordering over all possible concepts implied by such a context, constitutes a lattice. Various lattice construction algorithms have been devised to extract all possible concepts from a given context and to arrange them in a graph structure that reflects their parent/child relationships [17,12]. Figure 3 shows a line diagram, generated from the context in Table 1, showing the ordering of concepts in the lattice. Concepts have been labelled $c1, c2, c3, c4, c123$ and $c1234$.

	$\langle a, p_1\rangle$	$\langle a, p_2\rangle$	$\langle b, p_2\rangle$	$\langle c, p_3\rangle$	$\langle d, p_1\rangle$	$\langle d, p_2\rangle$	$\langle d, p_3\rangle$	$\langle d, p_4\rangle$
p_1	1		1	1	1			
p_2	1		1	1		1		
p_3	1		1	1			1	
p_4		1	1	1				1

Table 1: The state/out-transition context of DFA in Figure 1a

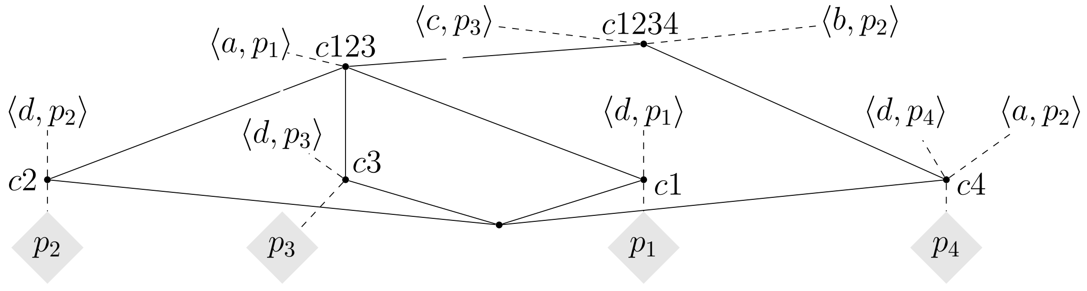


Figure 3: State/out-transition formal concept lattice of DFA in Figure 1a

Consider concept $c123$. Its extent is given by $ext(c123) = \{p_1, p_2, p_3\}$, while its intent is $int(c123) = \{\langle a, p_1\rangle, \langle c, p_3\rangle, \langle b, p_2\rangle\}$. Thus, concept $c123$ indicates that objects p_1, p_2 and p_3 share all and only the attributes $\langle a, p_1\rangle$, $\langle c, p_3\rangle$ and $\langle b, p_2\rangle$.

Concept $c123$ illustrates that the extent of a concept is the union of the extents of its children, together with any of its so-called "own objects". In this case, $c123$ does not have any own objects. Its children are $c1, c2$ and $c3$, and their respective extents correspond to their own objects, which are explicitly shown in the diagram – i.e. their extents are $\{p_1\}$, $\{p_2\}$ and $\{p_3\}$ respectively. Dually, concept $c123$ also illustrates the fact that the intent of a concept is the union of the intents of all its parents, together with any of its so-called "own attributes". It has $\langle a, p_1\rangle$ as its single own attribute, and its only parent, $c1234$, adds its intent, $\{\langle c, p_3\rangle, \langle b, p_2\rangle\}$, to that of $c123$.

Information in a DFA's transition graph can be represented in a context, and hence in a formal concept lattice. Here we propose one particular way to do so and call the result a *state/out-transition (formal concept) lattice*. For a DFA D, we denote this lattice by $\mathcal{SO}(\mathcal{D})$. The set of objects in $\mathcal{SO}(\mathcal{D})$ is simply the set of states in D, namely Q. Each attribute is a pair consisting of the label of an out-transition from some state, and the corresponding destination state. Formally, $\langle b, p\rangle$ is an attribute in $\mathcal{SO}(\mathcal{D})$ iff $\exists : q \in Q : \delta(q, b) = p$. In this case, $\langle b, p\rangle$ is an attribute of object q. The context in Table 1 was derived from the DFA in Figure 1a in precisely this way, and hence Figure 3 shows $\mathcal{SO}(\mathcal{D})$.

The space and time requirements for building the lattice's context table are determined by the size of δ, i.e. they are $\mathcal{O}(|Q|^2 \times |\Sigma|)$. An SO-lattice is a constrained lattice in the sense that its objects are constrained to each have exactly one attribute from each of $|\Sigma|$ classes, each class having $|Q|$ attributes. In [13] it is shown that the number of concepts for such a lattice is bound from above by $min((1 + |\Sigma|)^{|Q|}, \frac{|Q|}{(1+|\Sigma|)} 2^{1+|\Sigma|})$.

For convenience, we shall denote this expression by $\mathcal{LB}(\Sigma, Q)$. This means that for a fixed alphabet, an upper bound of the lattice size eventually becomes linearly dependent on the number of states.

4 A DFA-Homomorphic Algorithm

Consider an arbitrary concept c in $\mathcal{SO}(\mathcal{D})$, the state/out-transition lattice for a complete DFA $\mathcal{D}$. By definition, all states in $ext(c)$ share all and only the out-transitions in $int(c)$. (Of course, each state in $ext(c)$ may have other out-transitions.) For convenience, let $m = |ext(c)|$ and $n = |int(c)|$. Let q be any state in $ext(c)$, let $P = ext(c) \setminus \{q\}$ and let $X = \operatorname{dom} int(c)$. Thus $X \subseteq \Sigma$ is the set of symbols on which transitions to common states are made from all states in $ext(c)$.

We argue that $\mathrm{FailPred}(P, q, X)$ is true because the following holds for each $p \in P$. Conjunct 1 of the predicate is true because the DFA is assumed to be complete. Conjuncts 2 and 4 of the predicate hold because a DFA has no failure arcs. Conjunct 3 holds by the construction of $\mathcal{SO}(\mathcal{D})$ and by the definition of a concept in a concept lattice. Therefore Theorem 8 may be applied to produce an equivalent FDFA. This entails the following arc changes:

For each $p \in P$ remove all outgoing arcs represented in $int(c)$. Thus, the number of arcs removed from $\mathcal{D}$ is $n(m-1)$.
For each $p \in P$ install an outgoing failure arc to q. Thus, the number of arcs added to $\mathcal{D}$ is $(m-1)$.

As a result of these steps, $(n-1)(m-1)$ arcs will be removed from the initial structure.

For a given concept, c, we will call $(n-1)(m-1)$ its *arc redundancy*, denoted by $ar(c)$. For example, $ar(c123) = (3-1)\times(3-1) = 4$ since $|int(c123)| = |ext(c123)| = 3$.

If the above steps to construct a failure arc are applied to a concept c for which $ar(c) = 0$, there will be no decline in the number of arcs. Conversely, the 'maximum' decline is obtained if one selects from all the concepts, the one for which $ar(c)$ is 'maximal'. (Note that 'maximal' is used here in terms of the initially computed $ar(c)$ – it may not necessarily be maximal in terms of the current arc redundancy values, as we will discuss below.) This suggests the following 'greedy' algorithm for constructing FDFA $\mathcal{F}$ from DFA $\mathcal{D}$, assuming that $\mathcal{SO}(\mathcal{D})$ is available.

```
**Algorithm 2**
$\mathfrak{f}, O := \emptyset, Q$;
{ *Assume that AR is set of concepts with non-zero arc redundancy* }
{ *Invariant:* $(\operatorname{dom}\mathfrak{f} = Q \setminus O) \wedge$ (*Concepts in AR have not been processed*) }
do $((O \neq \emptyset) \wedge (AR \neq \emptyset)) \rightarrow$
    $c := maxcar(AR)$;
    $AR := AR \setminus \{c\}$;
    let $q \in ext(c)$;
    $P := ext(c) \setminus \{q\}$;
    for each $(p \in P \cap O) \rightarrow$
        if $\neg(q \overset{\mathfrak{f}}{\rightsquigarrow} p)$ COR $((\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \operatorname{dom} int(c)) = \emptyset) \rightarrow$
            for each $(\langle a, r\rangle \in int(c)) \rightarrow$
                $\delta := \delta \setminus \{\langle p, a, r\rangle\}$
            rof;
```

```
                $\mathfrak{f}(p), O := q, O \setminus \{p\};$
            $\|$  $(q \overset{\mathfrak{f}}{\rightsquigarrow} p)$ $\boldsymbol{CAND}$ $((\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \operatorname{dom} int(c)) \neq \emptyset) \to$ **skip**
            **fi**
        **rof**
**od**
```

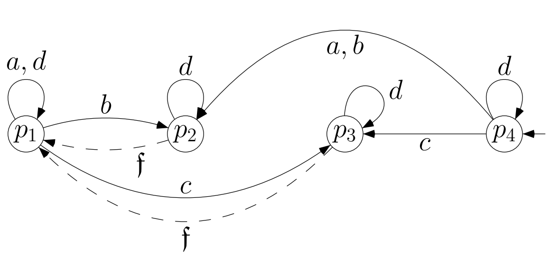
The set, AR, of concepts with non-zero arc redundancy is easily computed, and is assumed to be available to the algorithm. The algorithm initialises and maintains the set O of states which do not originate failure transitions, i.e. O is defined by $\operatorname{dom} \mathfrak{f} = Q \setminus O$. A function $maxcar : \mathcal{P}(\mathcal{SO}(\mathcal{D})) \to \mathcal{SO}(\mathcal{D})$ is assumed which selects from AR a concept, c with the maximum arc redundancy as initially determined.

Note that, as stated above, q is arbitrarily selected from $ext(c)$ to act as the target for failure arcs. The outer **for each** loop identifies states remaining in $ext(c)$ which may serve as sources of failure arcs. Such states have to be in O (to ensure compliance with conjunct 2 of Definition 7). The **if** statement then ensures that δ arcs are replaced by $\mathfrak{f}$ (within the inner **for each** loop) if and only if conjunct 4 of Definition 7 holds[3], thus ensuring that divergent cycles are never produced.

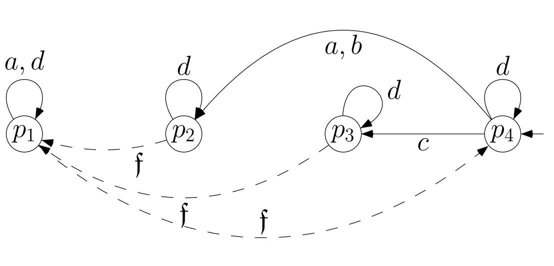
Applying the algorithm to the DFA in Figure 1a, and making use of the state/out-transition lattice shown in Figure 3 yields the FDFA shown in Figure 4a after the first iteration of the outer **do** loop. To see that this is so, note that upon entering the loop, $AR = \{c123, c1234\}$ where $ar(c123) = 4$, $ar(c1234) = 3$ and $O = \{p_1, p_2, p_3, p_4\}$. Thus, in the first iteration $maxcar(AR)$ returns concept $c123$ and the algorithm removes $c123$ from AR. Choosing $q = p_1$ (any element of $P = \{p_1, p_2, p_3\}$ could have been chosen) as the destination of all failure nodes in this iteration, the **for each** loop removes the following 6 arcs (δ mappings) from the DFA in Figure 1a:

$$\{\langle p_2, a, p_1\rangle, \langle p_2, b, p_2\rangle, \langle p_2, c, p_3\rangle, \langle p_3, a, p_1\rangle, \langle p_3, b, p_2\rangle, \langle p_3, c, p_3\rangle\}$$

Thereafter, it inserts two failure transitions: $\{\langle p_2, p_1\rangle, \langle p_3, p_1\rangle\}$. It also changes O to $\{p1, p4\}$. As a result, the number of arcs has been reduced by 4 – as predicted by $c123$'s arc redundancy.



(a) After first iteration: $|\delta| = 9, |\mathfrak{f}| = 2$



(b) After second iteration: $|\delta| = 8, |\mathfrak{f}| = 3$

Figure 4: FDFA's as Algorithm 2 progresses

[3] The guard of the **if** statement, namely $\neg(q \overset{\mathfrak{f}}{\rightsquigarrow} p)$ **COR** $(\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \operatorname{dom} int(c) = \emptyset))$, is logically equivalent to $(q \overset{\mathfrak{f}}{\rightsquigarrow} p) \Rightarrow (\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \operatorname{dom} int(c) = \emptyset)$, which in turn corresponds to conjunct 4 in Definition 7 in which $\operatorname{dom} int(c)$ takes the role of X.

After the second iteration of the outer **do** loop the FDFA in Figure 4b is obtained. (It is a copy of Figure 1b, reproduced here for convenience.) Upon entering the loop for a second time, $AR = \{c1234\}$. $maxcar(AR)$ therefore returns concept $c1234$. At this point one of the states in $ext(c1234) = \{p_1, p_2, p_3, p_4\}$ has to be selected as the destination of all failure nodes in this iteration.

The **for each** loop removes from the FDFA in Figure 4a the arcs $\langle p_1, b, p_2\rangle$ and $\langle p_1, c, p_3\rangle$ and inserts failure transition $\{\langle p_1, p_4\rangle\}$, reducing the number of arcs by 1.

We offer the following reflections, based on the algorithm and the example just given.

1. Out-transitions from p_2 and p_3 are not considered in the outer **for each** loop, since these became failure states in the previous iteration and were removed from set O. To have more than one failure arc emanating from a state would mean that we could no longer speak of a failure *function*, and we would not know under which circumstances which failure arc should be selected. This, of course, is the reason for conjunct 2 in Definition 7.
2. Suppose that instead of selecting p_4 from $ext(c1234)$ as the failure arc destination, p_2 had been chosen. In this case, p_1 and p_4 would be candidate source states for failure arcs to p_2. (Again, p_3 would be eliminated because it is already a failure state.) The **if** statement within the **for each** loop would discover that a failure arc $\langle p_1, p_2\rangle$ would result in a divergent failure cycle. Consequently, only failure transition $\langle p_4, p_2\rangle$ would be added, and arcs $\langle p_4, c, p_3\rangle$ and $\langle p_4, b, p_2\rangle$ would be removed.
3. It can easily be verified that the reduction in the number of arcs in the second iteration is by 1, no matter which member of $ext(c1234)$ is selected for the failure arc destination. This does *not* correspond to the initially computed value of $ar(c1234)$, namely 3. This is to be expected, because the algorithm as given above computes concept arc redundancy only once – at the start of the algorithm. Consequently, the algorithm in its above format naïvely ignores the fact that whenever states are removed from O, the concept arc redundancies may change for those concepts whose extents contain removed states. By implication, therefore, *maxcar* is no longer guaranteed to choose as "greedily" as it might have. This potential selection inefficiency could easily be overcome at the cost of recomputing concept arc redundancy whenever s is removed from O. In such a case, the arc redundancy of a concept whose extent contains s should account for the fact that s is not a candidate for being the *source* of an second failure arc.
4. The way in which the *target* state for failure arcs, q, is selected in Algorithm 2 could also be optimised to enhance the algorithm's greediness. Instead of selecting an arbitrary state in $ext(c)$, preference should be given to failure states (i.e. states already in $\operatorname{dom} \mathfrak{f}$). The reason for this heuristic is clear: when a state becomes a source state (i.e. a failure state), its δ arcs are removed, but when a state becomes a target state, no δ arcs are removed. Since an existing failure state is no longer a candidate for becoming a source state, and therefore cannot contribute to the removal of δ arcs, it might as well serve as the target of newly installed failure arcs, thus allowing more states to become source states and thus allowing more states to shed some of their δ arcs. If this heuristic is to be applied, then the recomputation of arc redundancy mentioned in point 3 above should be suitably adjusted.
5. The test to be carried out in the **if** statement of Algorithm 2 entails determining whether a divergent cycle will arise if a failure arc is installed from state p to

state q. Since cycle-determination is a well known task in general data structure theory, details here are unnecessary. In the present context, a cycle cannot be longer than $|Q|$. Furthermore cycles in a given FDFA have to be disjoint, since $\mathfrak{f}$ is a function. This considerably simplifies the task of identifying divergent cycles.

6. Consider the implications of providing a sink node to render a partial DFA complete so that it may serve as input to Algorithm 2 to produce an FDFA. Of course, there can be no guarantee that the number of arcs in the FDFA will be less than in the originating partial DFA. However, it may be possible to remove some of the inserted arcs from the FDFA. In particular, all arcs from non-failure states to the sink state may safely be removed. This applies, even if the non-failure state serves the target state of one or more failure states. Furthermore, noting that the sink state will have loops back to itself on all symbols (as part of the completeness requirement), it is possible that the sink state becomes the target of failure arcs from other states. Such failure arcs could be removed as well. Notwithstanding these few brief observations, the matter of converting partial DFAs to FDFAs requires further study.

Additional details relating to algorithmic enhancements mentioned in points 3 and 4 above are briefly taken up in [15], and an example is also given to illustrate the points that are made.

Recall from the previous section that the number of concepts in lattice $\mathcal{SO}(\mathcal{D})$ is bound from above by $\mathcal{LB}(\Sigma, Q)$, giving a very rough upper bound for $|AR|$ in our algorithm. We expect the actual bound to be much lower than this, since it is not clear that a state/out-transition lattice can reach the upper bound mentioned, and many concepts may have no arc redundancy and thus not end up in AR. Nevertheless, using that bound, and noting that $O \subseteq Q$, the outer **do** loop is executed at most $\mathcal{LB}(\Sigma, Q)$ times under the assumption that AR is never recomputed and under the unrealistic assumption that all concepts initially have a positive arc redundancy. This bound also ignores the fact that the loop terminates when O becomes the empty set.

The outer **for each** loop is executed at most $|Q|$ times, as both P and O are subsets of Q. The complexity of computing the value of the guards of the **if** statement is bounded by the maximum of $|Q|$ (for failure path tracing) and $|\Sigma|$ (for checking intersection), while the inner **for each** loop has complexity at most $|\Sigma|$. Combining this gives $\mathcal{LB}(\Sigma, Q) \times |Q| \times max(|Q|, |\Sigma|) \times |\Sigma|)$ as a very coarse upper bound on the algorithm's time complexity.

5 The Next Steps

The foregoing has stimulated a number of future research questions and ideas relating to FDFAs, their properties, their relation to DFAs, and their construction. They include investigating transition and state minimality properties of FDFAs compared to their DFA counterparts; directly constructing an FDFA from a regular expression; handling partial DFAs; and constructing a DFA from a given FDFA. Additionally, we are investigating alternative construction algorithms for producing an FDFA that is language-equivalent to a given DFA. These also rely on a state/out-transition lattice, but allow for the generation of new states that are derived from lattice concepts. In this sense, they can be regarded as "lattice-homomorphic". Refinements of the DFA-homomorphic algorithm of Section 4 have also been developed. They relate to the recomputation of arc redundancy and to optimised selection strategies for target state of a failure transition. The tradeoff between FDFA storage size reduction versus processing speed is currently under empirical investigation. We refer the reader to [15] for details regarding this and other future work.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) June 1975, pp. 333–340.
2. C. Carpineto and G. Romano: *Concept Data Analysis: Theory and Applications*, John Wiley & Sons, England, 2004.
3. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size.* IJFCS, 20(6) 2009, pp. 1069–1086.
4. M. Crochemore and C. Hancart: *Automata for matching patterns*, Springer-Verlag New York, Inc., New York, NY, USA, 1997, pp. 399–462.
5. J. Daciuk and D. Weiss: *Smaller representation of finite state automata*, in Proceedings of the Conference on Implementation and Application of Automata (CIAA), 2011, pp. 118–129.
6. N. de Beijer, L. Cleophas, D. G. Kourie, and B. W. Watson: *Improving automata efficiency by stretching and jamming*, in Proceedings of the Prague Stringology Conference (PSC), 2010, pp. 9–24.
7. E. W. Dijkstra: *Guarded commands, nondeterminacy and formal derivation of programs.* Commun. ACM, 18(8) Aug. 1975, pp. 453–457.
8. K. Driesen and U. Hölzle: *Minimizing row displacement dispatch tables*, in Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '95, New York, NY, USA, 1995, ACM, pp. 141–155.
9. E. Fredkin: *Trie memory.* Communications of the ACM, 3(9) 1960, pp. 490–499.
10. E. Ketcha Ngassam: *Towards cache optimization in finite automata implementations*, PhD thesis, University of Pretoria, 2007.
11. D. E. Knuth, J. James H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
12. D. G. Kourie, S. A. Obiedkov, B. W. Watson, and D. van der Merwe: *An incremental algorithm to construct a lattice of set intersections.* Science of Computer Programming, 74(3) 2009, pp. 128–142.
13. D. G. Kourie and G. D. Oosthuizen: *Lattices in machine learning: Complexity issues.* Acta Informatica, 35 1998, pp. 269–292.
14. D. G. Kourie and B. W. Watson: *The Correctness-by-Construction Approach to Programming*, Springer Verlag, 2012.
15. D. G. Kourie, B. W. Watson, T. Strauss, F. Venter, and L. Cleophas: *Failure deterministic finite automata*, Tech. Rep. 2012.1.0, FASTAR Research Group, 2012.
16. T. Kowaltowski, C. L. Lucchesi, and J. Stolfi: *Minimization of binary automata*, in Proceedings of the First South American String Processing Workshop, 1993, pp. 105–116.
17. S. O. Kuznetsov and S. A. Obiedkov: *Comparing performance of algorithms for generating concept lattices.* Journal of Experimental & Theoretical Artificial Intelligence, 14(2-3) 2002, pp. 189–216.
18. M. Mohri: *String-matching with automata.* Nordic Journal of Computing, 4 1997, pp. 217–231.
19. A. W. Roscoe: *The Theory and Practice of Concurrency*, Prentice Hall, 1997.
20. J. Sakarovitch: *Elements of Automata Theory*, Cambridge University Press, 2009.
21. R. E. Tarjan and A. C.-C. Yao: *Storing a sparse table.* Communications of the ACM ACM, 22(11) November 1979, pp. 606–611.
22. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, September 1995.

An Efficient Parallel Determinisation Algorithm for Finite-state Automata

Thomas Hanneforth and Bruce W. Watson

[1] Universität Potsdam, Germany
[2] Stellenbosch University, South Africa
Thomas.Hanneforth@uni-potsdam.de Bruce@fastar.org

Abstract. Determinisation of non-deterministic finite automata (NFA) is an important operation not only for optimisation purposes, but also the prerequisite for the complementation operation, which in turn is necessary for creating robust pattern matchers, for example in string replacement and robust parsing. In the paper, we present an efficient parallel determinisation algorithm based on a message-passing graph approach. In a number of experiments on a multicore machine we show that the parallel algorithm behaves very well for acyclic and cyclic NFAs of different sizes, especially in the worst case, where determinisation leads to an exponential blow-up of states.

Keywords: finite-state automata, determinisation, parallel algorithms, message passing, flow graphs, Kahn process networks, replacement rules

1 Introduction

Given a nondeterministic finite automaton (an NFA), *determinisation* is the construction of an equivalent deterministic finite automaton (DFA), where 'equivalence' means that the NFA and DFA accept the same language. Many real-life applications involve the relatively straightforward construction and manipulation initially of NFA's, for example when compiling regular expressions, regular grammars, or other descriptive formalisms (such as replacement rules in computational linguistics) to finite automata. While NFAs are often very compact and easily manipulated, several situations motivate the subsequent construction of a DFA:

– The standard approach for considering *equivalence of two automata* [4] is to minimize their respective equivalent DFA's and compare those (thanks to the uniqueness-modulo-isomorphism of minimal DFA's per language).
– Effective *complementation* of regular languages requires the construction of a DFA (cf. [4]).
– Complementation is also the key for *robust* natural language processing applications based on finite-state automata, e.g. shallow parsing systems etc. Many of these systems are build upon regular conditional [6] and unconditional replacement rules [7] which heavily rely on complementation to ensure robust application.
– The end-goal of constructing automata is often to apply it to a string, e.g. for pattern matching, network security applications, and computational linguistics. *Determinism* of the DFA means that only a single 'current state' needs to be tracked while processing input. By contrast, in the worst case, all of an NFA's states may become active while processing input – an enormous computational overhead as each symbol is processed, and usually impractical. In all of those applications, a DFA is essential [1].

The classical 'subset construction' algorithm[1] follows directly from Rabin & Scott's proof of NFA/DFA equivalence, and also shows that an equivalent DFA can be exponentially larger than the NFA in the worst-case [4]. Most real-life implementations combine reachability with the subset construction, which can subsequently be tuned quite effectively. In addition to tuning for memory and speed performance, various toolkits also implement *incremental* determinisation in which the DFA is constructed on-the-fly while processing an input string. Recent work by van Glabbeek & Ploeger [3] presents five determinisation algorithms, classifying them in lattice (based on the resulting DFA size), and giving benchmarking results.

Despite these algorithmic advances, there has been little work on parallel determinisation. Clock-speeds of modern processors and memory have plateaued and Moore's Law advances in silicon chip production are now devoted to more processor cores, enabling cheap multi-threading, with the caveat that parallel algorithms are less well-known and much more difficult to get correct. This paper presents one of the first such parallel algorithms.

Before we turn to the algorithm, we define the relevant technical notions in the next section. Then, Section 3 restates the standard reachability-based serial determinisation algorithm, before it develops an efficient parallel one. In Section 4, we give a short C++ code fragment which implements the parallel algorithm. Finally, in Section 5, we report on several experiments we conducted to compare serial and parallel determinisation.

2 Preliminaries

An alphabet Σ is a finite set of symbols. A string $x = a_1 \cdot a_2 \cdots a_n$ over Σ is a finite concatenation of symbols a_i taken from Σ (the concatenation operator $\cdot$ is normally omitted). The length of a string $x = a_1 \cdots a_n$ – symbolically $|x|$ – is n. The empty string is denoted by ε and has length zero. Let Σ^* denote the set of all finite-length strings (including ε) over Σ.
A *non-deterministic finite-state automaton* (NFA) A is a 5-tuple $\langle Q, \Sigma, q_0, \delta_{nd}, F\rangle$ with Q being a finite set of states; Σ, an *alphabet*, $q_0 \in Q$, the start state; $\delta_{nd} : Q \times \Sigma \mapsto 2^Q$, the transition function; and $F \subseteq Q$, the set of final states.
Define $\delta^*_{nd} : Q \times \Sigma^* \mapsto 2^Q$ as the reflexive and transitive closure of δ_{nd}:

- $\forall q \in Q, \delta^*_{nd}(q, \varepsilon) = \{q\}$ and
- $\forall q \in Q, a \in \Sigma, w \in \Sigma^* : \delta^*_{nd}(q, aw) = \bigcup_{p \in \delta_{nd}(q,a)} \delta^*_{nd}(p, a)$.

δ_{nd} may be a partial function. In case $\delta_{nd}(S, a)$ is undefined for some state set $S \subseteq Q$ and $a \in \Sigma$, we take $\delta_{nd}(S, a)$ be equal to $\emptyset$. The language of a NFA $A = \langle Q, \Sigma, q_0, \delta_{nd}, F\rangle$, symbolically $L(A)$, is defined as $L(A) = \{w \in \Sigma^* \mid \delta^*_{nd}(q_0, w) \cap F \neq \emptyset\}$.

A *deterministic finite-state automaton* (DFA) A is defined as a 5-tuple $\langle Q, \Sigma, q_0, \delta_d, F\rangle$ where A, Q, q_0, and F are the same as in the NFA case and δ_d is a (partial) function mapping $Q \times \Sigma$ to Q. The notions of δ^*_d and $L(A)$ are defined analogously to the ones in NFAs.

A state q is *reachable* if there exists a word $w \in \Sigma^*$ such that $\delta^*_d(q_0, w) = q$.

For every NFA, an equivalent DFA (with respect to the recognized language) can be constructed. The key idea is the *subset construction*:

[1] Sometimes known as the 'powerset construction', see the next section.

Definition 1 (Subset construction). *Let* $A = \langle Q, \Sigma, q_0, F, \delta_{nd}\rangle$ *be an NFA. Define* A'*, the equivalent DFA with* $L(A') = L(A)$ *as* $A' = \langle 2^Q, \Sigma, \{q_0\}, F', \delta_d\rangle$ *with:*

– $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$
– $\delta_d(S, a) = \bigcup_{q\in S} \delta_{nd}(q, a),\ \forall a \in \Sigma, \forall S \subseteq Q$

The next section describes serial and parallel determinisation algorithms based on the subset construction.

3 Determinisation algorithms

This section recapitulates the standard serial determinisation algorithm and introduces our parallel version of it.

3.1 Serial determinisation

A naïve NFA determinisation algorithm implementing Definition 1 directly would lead to worst-case behaviour in every case. In practice, it turns out that most of the states in the powerset of Q are not reachable from the start state of the DFA. Thus, their creation can be completely avoided by incorporating a reachability constraint into the algorithm. This leads directly to the queue-based version shown in Algorithm 1.

Algorithm 1: SERIAL NFA DETERMINISATION ALGORITHM

```
   Input: NFA $A = \langle Q, \Sigma, q_{0_{nd}}, \delta_{nd}, F\rangle$
   Output: DFA $A' = \langle Q', \Sigma, q_{0_d}, \delta_d, F'\rangle$
1  $R(\{q_{0_{nd}}\}) \leftarrow c \leftarrow q_{0_d} \leftarrow 0$
2  $L \leftarrow \emptyset$
3  $Q' \leftarrow F' \leftarrow \emptyset$
4  $Enqueue(L, \langle\{q_{0_{nd}}\}, q_{0_d}\rangle)$
5  while $L \neq \emptyset$ do
6      $\langle S, q\rangle \leftarrow Dequeue(L)$
7      $Q' \leftarrow Q' \cup \{q\}$
8      if $S \cap F \neq \emptyset$ then
9          $F' \leftarrow F' \cup \{q\}$
10     $C = \{\langle a, \bigcup_{p\in S} \delta_{nd}(p, a)\rangle \in \Sigma \times 2^Q \mid \exists r \in S : \delta_{nd}(r, a) \neq \emptyset\}$
11     for each $\langle a, S'\rangle \in C$ do
12         $p \leftarrow R(S')$
13         if $p = \top$ then
14             $c \leftarrow c + 1$
15             $R(S') \leftarrow p \leftarrow c$
16             $Enqueue(L, \langle S', p\rangle)$
17         $\delta_d(q, a) \leftarrow p$
```

Algorithm 1 uses several auxiliary data structures: First of all, $R : 2^Q \mapsto \mathbb{N} \cup \{\top\}$ is a *state register* mapping subsets of Q to natural numbers. If some set S is not in the register, $R(S)$ returns $\top$. Initially, the set containing q_{0_d} is mapped to zero. Furthermore, the algorithm maintains a queue L holding pairs $\langle S, q\rangle \in 2^Q \times \mathbb{N}$ and a global state counter c initially set to 0. In line 4, the initial pair $\langle\{q_{0_{nd}}\}, 0\rangle$ is added to the queue which is subsequently processed in the **while**-loop between lines 5 and 17. In line 6, a pair $\langle S, q\rangle$ is removed from L. If S contains a final state, q is added to the final states of the DFA. In line 10, a set C of *candidate states* is constructed. For this purpose, a set $\Sigma' \subseteq \Sigma$ is created such that a is in Σ' if $\delta_{nd}(r, a)$ is defined (that is, $\delta_{nd}(r, a) \neq \emptyset$) for some $r \in S$. Then, for each $a \in \Sigma'$, a new state set S' is assembled holding all the destination states $\delta_{nd}(p, a)$ for all $p \in S$. In the following, we will refer to this step as the *symbol indexing step*. The **for**-loop in lines 11–17 processes all pairs $\langle a, S' \rangle$. Line 12 looks up state set S' in the register R. If S' is not found in R, it is added to R by assigning it a new state number p by incrementing the global state counter c (line 14–15). Furthermore, the pair $\langle S', p \rangle$ is added to the queue L (line 16). In both cases, a new transition from q to p with a is added to δ_d (line 17).

By maintaining a queue L, the algorithm ensures that each state q added to Q' in line 7 is reachable from that start state q_{0_d}. Nevertheless, in the worst case, all subsets of Q are added to the queue resulting in a running time in $\mathcal{O}(|2^Q||\Sigma|)$.

Given an alphabet $\Sigma = \{a, b\}$, the worst case is exihibited by NFAs resulting from regular expressions $r(k)$ of the form $\Sigma^* a(a+b)^k$ which leads to DFA with 2^{k+1} states. Figure 1 shows an NFA constructed from $r(2)$, while Figure 2 shows the equivalent DFA. Note that DFA constructed from regular expressions $r(k)$ are also complete, that is, δ_d is a total function.

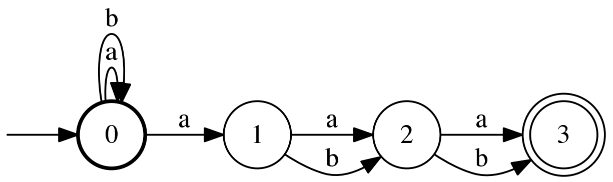


Figure 1. NFA created from regular expression $\Sigma^* a(a + b)^2$

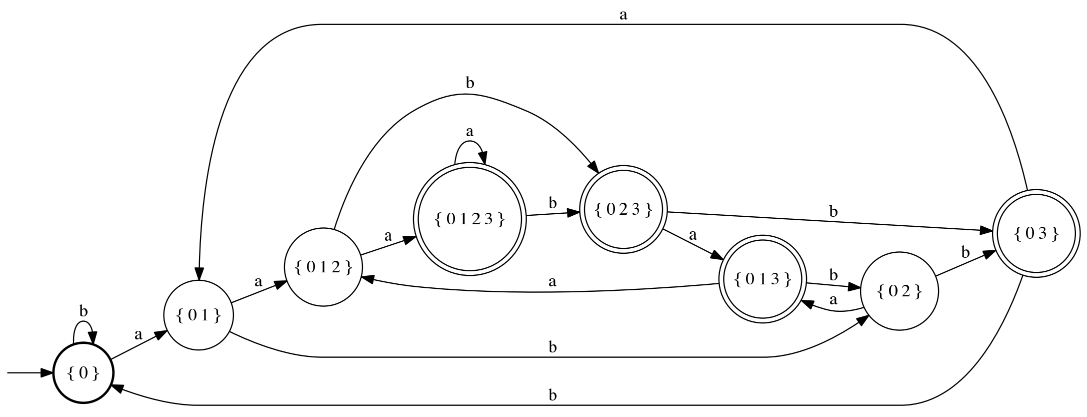


Figure 2. Equivalent DFA to the NFA of Figure 1

This worst case of exponential blow-up may not be so uncommon in practice as one might expect. Consider a *pattern matching* problem [1] where some finite set P of patterns is to be efficiently found in some given input text. In automata-theoretic terms, this amounts to constructing an NFA for $\Sigma^* \cdot P$, the infinite regular language consisting of all strings having some $p \in P$ as a suffix. If P has the form $a(a + b)^k$ or something similar, then determinisation is exponential.

3.2 Parallel determinisation

When looking at Algorithm 1 for parts which could be run in parallel and which can not, the following observations could be made:

– The **while**-loop between lines 5 and 17 is a good candidate for parallel processing, since several pairs $\langle S, q\rangle$ could be removed from the queue (line 6) and further processed in parallel.
– This is in particular the case for the symbol indexing step in line 10, since the creation of follower candidate states for each state set S is completely independent for all state sets S.
– The **for**-loop (lines 11–17) could in principle be parallelised, but the main actions in its body – querying/adding to the state register and adding new transitions to the DFA – must certainly be serialised.
– The same is true for adding final states to the DFA (line 8–9). Assuming a suitable data structure for the DFA, adding final states (line 9) and adding transitions (line 17) can certainly be done in parallel.
– Incrementing the state counter (line 14) must again be serialised.

A straightforward way to link parallel and serial components of the algorithm are the concepts of *Kahn Process Networks*, (cf. [5]) and *Labeled Transition Systems* ([2]).

Definition 2 (Labelled Transition System (LTS), cf. [2]). *Let a channel c be an unbounded FIFO-queue (first-in-first-out queue) with elements taken from some alphabet Σ_c. Let $Chan$ denote the set of all channels.*
An LTS is a tuple $\langle S, s_0, I, O, Act, \rightarrow\rangle$ consisting of a set S of states, an initial state $s_0 \in S$, a set $I \subseteq Chan$ of input channels, a set $O \subseteq Chan$ (distinct from I) of output channels, a set Act of actions consisting of input actions $\{c?a \mid c \in I, a \in \Sigma_c\} \subseteq Act$, output actions $\{c!a \mid c \in O, a \in \Sigma_c\} \subseteq Act$ and a labelled transition relation $\rightarrow \subseteq S \times Act \times S$.

Definition 3 (Kahn Process Network (KPN), cf. [2]). *A Kahn process network is a tuple $\langle P, C, I, O, Act, \{LTS_p \mid p \in P\}\rangle$ with the components as follows:*

– *P is a finite set of processes.*
– *C, I and O ($\subseteq Chan$) are finite and pairwise disjunct sets of internal channels, input channels and output channels, respectively.*
– *$Act = \{c?a, c!a \mid c \in C \cup I \cup O, a \in \Sigma_c\}$*
– *Every process $p \in P$ is defined by a* sequential *labelled transition system*[2] *$LTS_p = \langle S_p, s_{p_0}, I_p, O_p, Act, \xrightarrow{p}\rangle$, with $I_p \subseteq I \cup C$ and $O_p \subseteq O \cup C$.*
– *For every channel $c \in C \cup I$, there is exactly one process $p \in P$ that reads from it ($c \in I_p$) and for every channel $c \in C \cup O$, there is exactly one process $p \in P$ that writes to it ($c \in O_p$).*

Since KPNs are essentially graph structures, they admit an intuitive graphical representation. Figure 3 shows a KPN for the parallel version of the determinisation algorithm.

The start state labelled s_0 in Figure 3 starts the network by passing the initial pair $\langle\{q0_{nd}\}, q0_d\rangle$ to the state labelled *process state set*. This corresponds to enqueuing the initial pair in line 4 of Algorithm 1. State *process state set* – which basically implements the body of the **while**-loop in Algorithm 1 – is connected with 3 other nodes:

[2] Basically, a *sequential* LTS accepts at most one input/output operation at a given point in time.

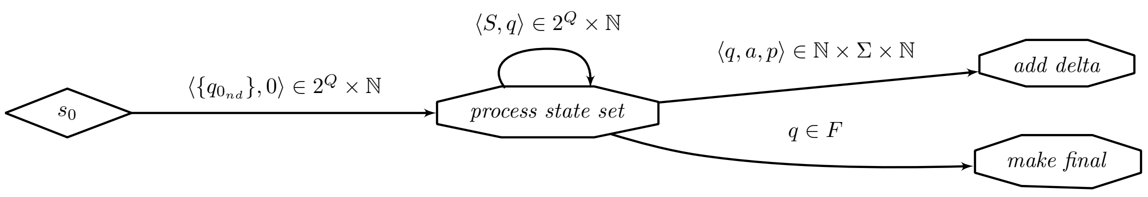

$\langle S, q\rangle \in 2^Q \times \mathbb{N}$
$\langle \{q_{0_{nd}}\}, 0\rangle \in 2^Q \times \mathbb{N}$
s_0
process state set
$\langle q, a, p\rangle \in \mathbb{N} \times \Sigma \times \mathbb{N}$
add delta
$q \in F$
make final

Figure 3. KPN for the parallel determinisation algorithm. Nodes with octogon shape are parallel nodes

1. with itself. This corresponds to line 15 of Algorithm 1: a state set S may lead to the creation of further state sets if these are not already present in the state register.
2. with state *add delta* which reflects line 17 of the serial algorithm and
3. with state *make final* which corresponds to line 9.

All states except s_0 work in principle in parallel, but they share three common resources: the state register, the state counter and the emerging DFA. Access to these resources must be synchronised by using appropriate locking mechanisms.

4 Implementation

The algorithm of Section 3.2 is implemented on the basis of the *flow graph* construct in Intel's *ThreadBuildingBlocks* C++ library (TBB, [8]). TBB defines a number of different graph nodes classes like `broadcast_node`, `function_node` and `multifunction_node`, which can be connected to each other by data flow edges.

Unlike instances of `function_node`, which are required to always compute a result, instances of `multifunction_node` are connected to other flow graph nodes by a tuple of channels, to which output actions are send. This is exactly what is required by state *process state set* in Figure 3, since a NFA state set currently processed may not lead to further new state sets.

The serial and parallel version of the determinisation algorithm are basically based on the same data structures. State sets of NFAs were implemented as sorted vectors. To allow an efficient test for equality of state sets and to speed up look-up in the state register, each state set also stores a permanent hash value. The δ-function of the class representing serial DFAs is based on a STL `hash_map`, while the one of the concurrent DFA uses TBB's `concurrent_hash_map`, a map data structure where the keys can be individually locked. The state registers for the serial and parallel algorithms are implemented in a similar fashion. Figure 4 states some of the relevant definitions of the parallel algorithm in C++. `ParallelDeterminizerBody`, `AddDeltaBody`, and `MakeFinalBody` are classes which implement the actions executed by the three parallel nodes of Figure 3.

5 Experiments

For the experiments, we choose two different types of NFAs:

1. Acyclic NFAs compiled from word lists and
2. Cyclic NFAs exhibiting the worst case along the lines of Figure 1 with various number of states and alphabet sizes.

```
typedef int                                                          STATE;
typedef StateSet<STATE>                                              NFAStateSet;
typedef std::pair<NFAStateSet,STATE>                                 NFAStateSetToState;
typedef tbb::concurrent_hash_map<NFAStateSet,STATE>                  StateRegister;
typedef std::tuple<NFAStateSetToDFAState,AddDelta,MakeFinal>         NFAStateSetToDFAStateTuple;
typedef tbb::flow::multifunction_node<NFAStateSetToState,NFAStateSetToStateTuple> ParDetWorkerNode;
typedef tbb::flow::function_node<AddDelta>                           AddDeltaNode;
typedef tbb::flow::function_node<MakeFinal>                          MakeFinalNode;

NFA nfa;
ConcurrentDFA dfa;
StateRegister state_register;
tbb::atomic<STATE> state_counter;
unsigned num_workers = tbb::flow::unlimited;

tbb::flow::graph g;
tbb::flow::broadcast_node<NFAStateSetToState> pardet_root_node(g);
ParallelDeterminizerBody pardet_worker_body(nfa,state_register,state_counter);
ParDetWorkerNode pardet_node(g, num_workers, pardet_worker_body);
AddDeltaNode add_transition_node(g, num_workers, AddDeltaBody(dfa));
MakeFinalNode make_final_node(g, num_workers, MakeFinalBody(dfa));
tbb::flow::make_edge(pardet_root_node, pardet_node);
tbb::flow::make_edge(tbb::flow::output_port<0>(pardet_node), pardet_node);
tbb::flow::make_edge(tbb::flow::output_port<1>(pardet_node), add_delta_node);
tbb::flow::make_edge(tbb::flow::output_port<2>(pardet_node), make_final_node);
```

Figure 4. C++ definitions of the parallel algorithm based on Intel's TBB framework

The acyclic NFAs derived from two different English words lists are maximally non-deterministic, that is, each word inserted into the NFA constitutes a separate chain from the start state to a distinct final state.

Table 1 summarises the different test automata.

NFA	$\lvert\mathbf{\Sigma}\rvert$	$\lvert\mathbf{Q_{nd}}\rvert$	$\lvert\delta_{\mathbf{nd}}\rvert$	$\lvert\mathbf{F_{nd}}\rvert$	$\lvert\mathbf{Q_d}\rvert$	$\lvert\delta_{\mathbf{d}}\rvert$	$\lvert\mathbf{F_d}\rvert$
NFA$_{dict1}$	56	681,719	681,718	49,999	271,194	271,193	49,999
NFA$_{dict2}$	45	994,676	994,675	128,972	270,411	270,410	128,972
NFA$_{r(k),2}$, $k \in [10 \dots 22]$	2	$k+1$	$2(k+1)+1$	1	2^{k+1}	$2 \cdot 2^{k+1}$	$\frac{2^{k+1}}{2}$
NFA$_{r(k),10}$, $k \in [10 \dots 19]$	10	$k+1$	$10(k+1)+1$	1	2^{k+1}	$10 \cdot 2^{k+1}$	$\frac{2^{k+1}}{2}$
NFA$_{r(k),100}$, $k \in [10 \dots 16]$	100	$k+1$	$100(k+1)+1$	1	2^{k+1}	$100 \cdot 2^{k+1}$	$\frac{2^{k+1}}{2}$

Table 1. Input NFA for the determinisation algorithms

All subsequent experiments were run on a Linux machine with 2 Intel-XEON 64bit-2.93 GHz 4-core-CPUs. Hyperthreading was turned on. In hyperthreaded architectures, each physical core is supplemented by a virtual core which takes over control if the physical one is currently stalled because it is waiting for CPU cache data etc. Virtual cores duplicate only certain sections of their physical counterpart, mainly those holding the current thread's state, but not the main computing resources.

Let us now turn to the experiments. In Figure 5, we compare the serial and the parallel version of the determinisation algorithm for NFA$_{r(k),2}$ on a logarithmic time scale. Unsuprisingly, the processing time for both versions grows exponentially with the number k of disjunctions[3] in the NFA. Starting with $2^{11+1} \approx 4,000$ DFA states, the parallel algorithm outperforms the serial one, with $k = 13$, it is already more than twice as fast. With bigger ks, the processing time of the parallel version converges at approximately one-third of the serial one.

The advantage of the parallel algorithm becomes even better when the alphabet size is increased. Figure 6 compares the serial and parallel determinisation of NFA$_{r(k)}$ with $|\Sigma| = 10$ and $|\Sigma| = 100$, respecitively. For an alphabet size of 10, the parallel algorithm is, depending on k, approximately 2 to 3.5 times faster than the serial one. For $|\Sigma| = 100$, the ratio is between 3.7 to 1 for $k = 10$ and 4.7 to 1 for $k = 16$.

[3] Also known as *alternations*.

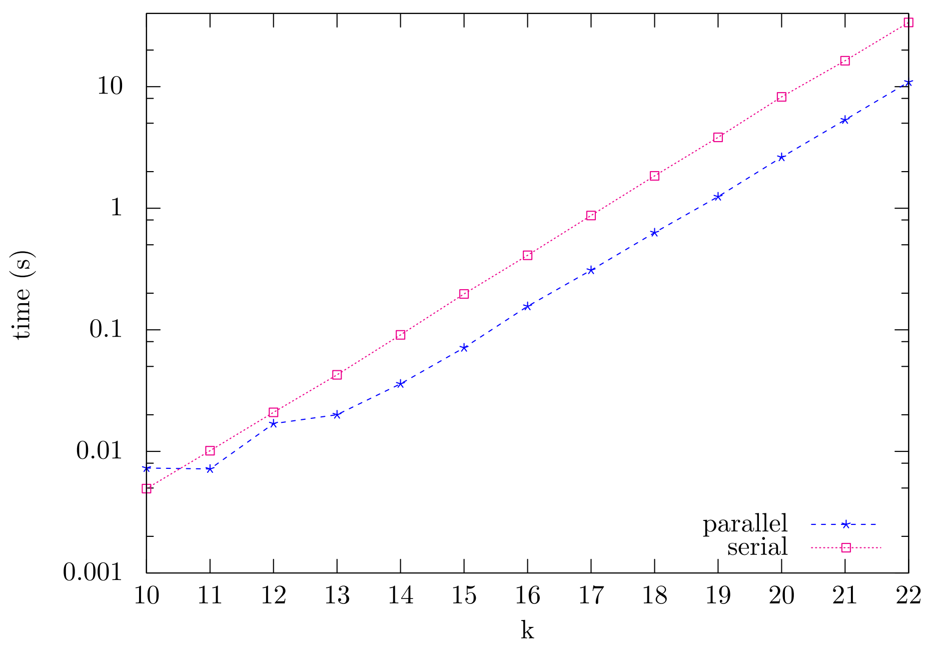


Figure 5. Serial vs. parallel determinisation of $\text{NFA}_{r(k),2}$

An explanation for the speedup for bigger alphabet sizes could be, that the number of DFA states depends only on k and thus the number of pairs $\langle S, q\rangle$ forwarded on the looping channel in Figure 3 is independent of the alphabet size. Furthermore, the state register shared between the parallel determinisation workers – even when queried $|\Sigma|$ times for each state set – doesn't seem to slow down processing very much.

To assess the amount of the contribution of the other shared resource – the DFA under construction – we ran a further experiment where we turned off the channels to the states *add delta* and *make final* in Figure 3 and made a similar move in the serial version. The results for the $\text{NFA}_{r(k),100}$ are shown in Figure 7.

Figure 7 shows that for the serial case whether DFA construction is turned on or off makes almost no difference with respect to processing time. But the situation is different for the parallel algorithm where the DFA construction contributes with more than 40% to the overall processing time.

Since both serial and concurrent DFA implementations rely on efficient hash maps, the difference must be explained with locking issues and the administrative overhead for implementing micro locks.

In our second-last experiment, we compare serial and parallel determinisation applied to acyclic NFAs, namely, the NFAs derived from the two word lists. Table 2 summarises the results.

NFA	serial	parallel
$dict_1$	0.465 s	0.197 s
$dict_2$	0.550 s	0.230 s

Table 2. Serial and parallel determinisation of the dictionary NFAs

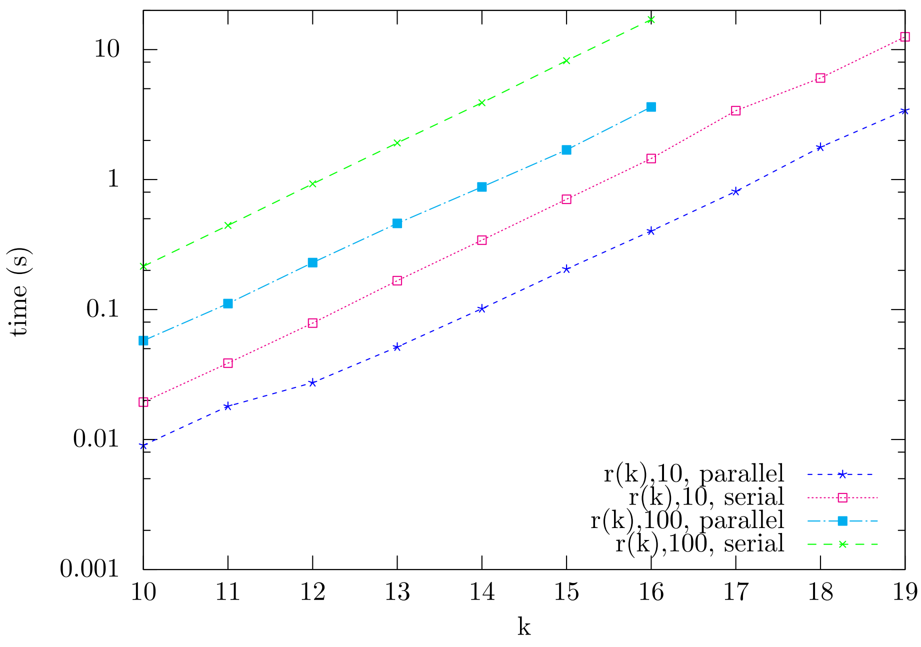


Figure 6. Serial vs. parallel determinisation of NFA$_{r(k),10}$ and NFA$_{r(k),100}$, resp.

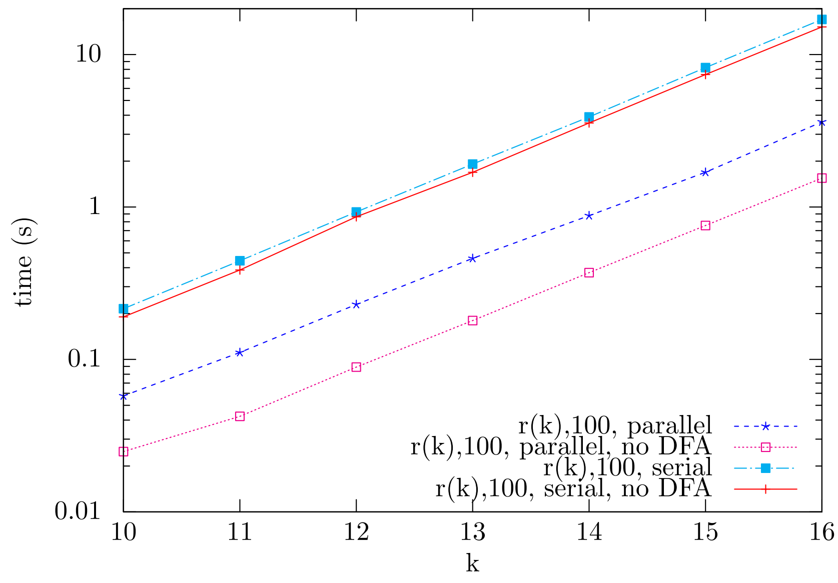


Figure 7. Serial and parallel determinisation of NFA$_{r(k),100}$, with DFA construction turned on and off

The parallel version exhibits a speed-up of a factor of approximately 2.4 compared to the serial algorithm. Even though the alphabet sizes are bigger, this is less than the speed-up in the cyclic case with an alphabet of size 2.

But, (serial) determinisation of acyclic NFA is very efficient anyway, since it is linear in the size of the NFA, so parallelising the algorithm is normally not worth the effort.

Intel's *ThreadBuildingBlocks* framework allows the control of the amount of parallelism by specifying the number of flow graph node copies concurrently active. A value of *unlimited* means that the framework chooses an optimal amount of concurrency.

Figure 8 shows the dependencies between the number of workers and the time consumed for two NFA ($r(17), 2$ and $dict_2$).

The relative plateau in both graphs for 8 to 16 workers could perhaps be explained with Intel's *hyperthreading* feature. Since the non-virtual cores are not idle when the parallel algorithm is executed, the virtual ones cannot take over control and thus do not contribute at all. Also apparent from the graphs is, that the parallel algorithm performs best if TBB's scheduler is allowed to control the amount of parallelism.

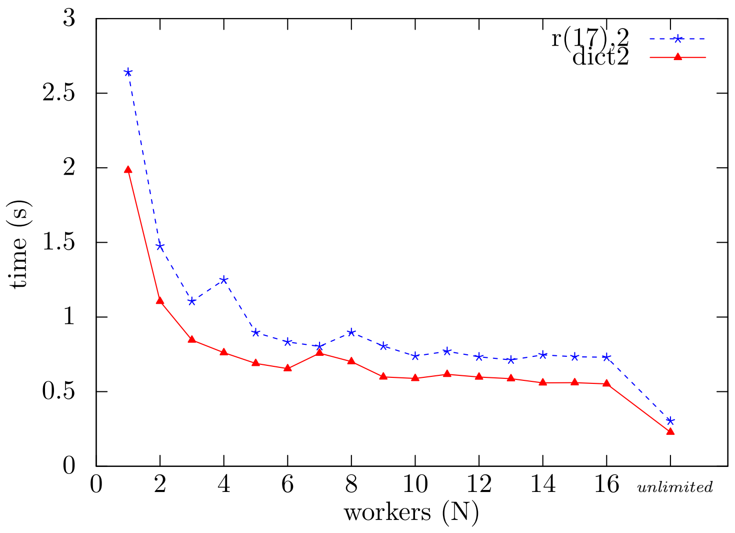


Figure 8. Dependency between the processing time and the number of workers for NFAs $r(17), 2$ and $dict_2$

6 Conclusion and further work

In the preceding sections, we developed an efficient parallel determinisation algorithm based on Kahn process networks. Experiments showed that the algorithm performed particularly well in cases of highly-cyclic result DFAs over realistically sized alphabets.

The worst-case pattern $\Sigma^* a(a+b)^k$ we choose for the cyclic test automata is not as artifical as it looks at first glance. For example, compiling replacement rules $\alpha \to \beta$ [7] relies on a *does-not-contain operator* $\overline{\Sigma^* \cdot \alpha \cdot \Sigma^*}$ to achieve robust behaviour by identity-mapping all strings to themselves which do not contain an instance of α. Since the standard complementation operation depends on a deterministic DFA for $\Sigma^* \cdot \alpha \cdot \Sigma^*$, choosing $a(a+b)^k$ for α creates the worst case.

In further work, we try to improve the algorithm in the following ways:

- Examine and profile the algorithm to reduce the number of locking situations,
- apply randomisation techniques to further reduce locking,

– make use of graph theoretic notions to divide the determinisation problem into largely independent subproblems which can be solved without making much use of shared resources, and
– study a redundant work approach where each parallel determinisation worker may process a limited number of state sets already processed by other workers to increase the relative independence of the workers from the shared resources.

References

1. A. V. Aho: *Algorithms for Finding Patterns in Strings*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. A, North-Holland, 1990, pp. 257–300.
2. M. Geilen and T. Basten: *Requirements on the Execution of Kahn Process Networks*, in Proc. of the 12th European Symposium on Programming, ESOP 2003, Springer Verlag, 2003, pp. 319–334.
3. R. Glabbeek and B. Ploeger: *Five Determinisation Algorithms*, in Proceedings of the 13th International Conference on Implementation and Applications of Automata, CIAA '08, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 161–170.
4. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.
5. G. Kahn: *The Semantics of a Simple Language for Parallel Programming.* Information Processing, 1974, pp. 471–475.
6. R. M. Kaplan and M. Kay: *Regular Models of Phonological Rule Systems.* Computational Linguistics, 20(3) 1994, pp. 331–378.
7. L. Karttunen: *The Replace Operator*, in ACL, 1995, pp. 16–23.
8. J. Reinders: *Intel Threading Building Blocks. Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly, 2007.

BlastGraph: Intensive Approximate Pattern Matching in Sequence Graphs and de-Bruijn Graphs

Guillaume Holley[1] and Pierre Peterlongo[1]*

Centre de recherche INRIA Rennes - Bretagne Atlantique, IRISA, Campus universitaire de Beaulieu, Rennes, France
`guillaumeholley@gmail.com, pierre.peterlongo@inria.fr`

Abstract. Many de novo assembly tools have been created these last few years to assemble short reads generated by high throughput sequencing platforms. The core of almost all these assemblers is a sequence graph data structure that links reads together. This motivates our work: BlastGraph, a new algorithm performing intensive approximate string matching between a set of query sequences and a sequence graph. Our approach is similar to blast-like algorithms and additionally presents specificity due to the matching on the graph data structure. Our results show that BlastGraph performances permit its usage on large graphs in reasonable time. We propose a Cytoscape plug-in for visualizing results as well as a command line program. These programs are available at `http://alcovna.genouest.org/blastree/`.

Keywords: sequence graph, de-Bruijn graph, string matching, high throughput sequencing, next generation sequencing, sequence assembly, Viterbi algorithm

1 Introduction

Compared to traditional Sanger technologies High Throughput Sequencing (HTS) technologies enable sequencing of biological material (DNA and RNA) at much higher throughput and a cost that is now affordable by most academic labs. They have revolutionized the field of genomics and medical research [5]. Sequencing became in a few years accessible to almost all biological labs while being able to produce sequences of full complex genomes in a few days.

HTS technologies do not output the entire sequence of a DNA or RNA molecule. Instead, they return small sequence fragments, called reads, whose length is usually ranging between 100 to 700 characters although some technologies produce longer reads. HTS produce overlapping reads, thus making possible to reconstruct the original sequence by assembling them. Over the last few years, many assemblers were created, such as Euler [2,3], Velvet [12] or Soapnovo [7] to cite a few among the most famous ones. They present different capacities and drawbacks, but all of them make use of a graph data structure storing sequences for organizing the reads. For assembly, the most used graph is the de-Bruijn graph, first proposed for assembly purposes by Pevzner, Tang and Waterman [9]. In a de-Bruijn graph a node represents a length-k substring (called a k-mer) and an edge connects nodes u and v if the two corresponding k-mers overlap over $k-1$ positions. Once the graph is created and usually after an error correction step, a traversal of the graph is performed for generating contiguous sequences called *contigs*.

* Corresponding author

In this paper, we present BlastGraph, a generic approach for aligning a (possibly large) set of query sequences on a graph storing sequences. The algorithm we propose applies on two kinds of graphs: 1/ any kind of graph storing sequences, called Sequence Graphs (SG); 2/ de-Bruijn graphs (DBG). Motivations for this work are multiple. For developers of assembly tools, it is of great interest to precisely detect query sequences in the graph, for instance while testing filter algorithms or correction algorithms. Biologically, checking the presence of approximate copies of a set of sequences in the graph, enables to detect homologies, to filter contaminants and to detect the presence of species. Avoiding the full assembly process presents two main advantages: first it avoids the time consuming contig generation phase, and second and more important, it avoids the usage of heuristics or statistical choices made while traversing the graph.

Note that the BlastGraph algorithm applies generically to any directed SG, and is also adapted to apply to a DBG. Given a directed sequence graph, a set of query sequences and a maximum edit distance, BlastGraph detects paths in the graph on which query sequences align at most at the given edit distance. Our approach is a blast-like algorithm [1]. The graph is indexed using seeds, this enables to decrease the request execution time. The main originality of our work stands in the fact that both seeds and mapped query sequences may be spread over several nodes of the graph.

This work presents similarities with the famous Viterbi algorithm [11]. In a few words, Viterbi is a dynamic programming algorithm for finding the most likely path in a rooted graph while reading a query sequence. The major fundamental differences with this work stand in the fact that:

– Viterbi nodes are composed by a unique symbol while in the BlastGraph framework, nodes store a full sequence, and their reverse complement in the DBG framework;
– In the Viterbi framework, the alignment is global: the full query sequence is aligned to the whole graph, starting from the root node, while in the BlastGraph algorithm, the alignment is semi global: the whole query sequence is aligned to any un-rooted sub-graph.

The next Section introduces preliminaries and definitions. In Section 3 we expose the BlastGraph algorithm when applied on a SG, while in Section 4 we show how BlastGraph is modified to apply on a DBG. We present some practical results in section 5.

2 Preliminaries

A *sequence* is composed by zero or more symbols from an alphabet Σ. A sequence s of length n on Σ is denoted also by $s[0]s[1]\cdots s[n-1]$, where $s[i] \in \Sigma$ for $0 \leq i < n$. The edit distance between two sequences is the minimal number of insertions, deletions and substitutions to transform one into the other. The length of s is denoted by $|s|$. We denote by $s[i,j]$ the *substring* $s[i]s[i+1]\cdots s[j]$ of s. In this case, we say that the substring $s[i,j]$ *occurs* at position i in s. We call *k-mer* a sequence of length k. If $s = u \cdot v$ for u and $v \in \Sigma^*$, we say that v is a *suffix* of s and that u is a *prefix* of s, the symbol "$\cdot$" designating the concatenation between two sequences. Let $s[i..]$ denote the suffix of s starting at position i (*i.e.* $s[i..] = s[i, |s|-1]$).

The symbol "$\overset{\circ}{k}$" designates the concatenation of two sequences, removing the first k symbols of the second. Formally, $u \overset{\circ}{k} v = u \cdot v[k..]$. In the DNA context, $\Sigma =$

$\{A, C, G, T\}$, and, given $s \in \Sigma^*$, $\overline{s}$ designates the reverse complement of s, that is s, read from right to left, switching characters A and T, and C and G.

2.1 Sequence Graphs (SG)

In a directed sequence graph $\mathcal{G}$, each node N stores a sequence s, denoted by $S(N)$. A node N_1 linked to a node N_2 denotes the fact that the sequence $S(N_1).S(N_2)$ is stored in $\mathcal{G}$. Example of a directed sequence graph is given Figure 1a.

2.2 De-Bruijn Graphs (DBG)

DBGs were first used in the context of genome assembly in 2001 by Pevzner *et al.* [9]. In 2007, Medvedev *et al.* [8] modified the definition to better model DNA as a double stranded molecule. In this context, given a fixed k value, a DBG is a bi-directed multigraph, each node N storing a k-mer s and its reverse complement $\overline{s}$. The sequence s, denoted by $F(N)$, is the forward sequence of N, while $\overline{s}$, denoted by $R(N)$, is the reverse complement sequence of N. An arc exists from node N_1 to node N_2 if the suffix of length $k-1$ of $F(N_1)$ or $R(N_1)$ overlaps perfectly with the prefix of $F(N_2)$ or $R(N_2)$. Each arc is labelled with a string in $\{FF, RR, FR, RF\}$. The first letter of the arc label indicates which of $F(N_1)$ or $R(N_1)$ overlaps $F(N_2)$ or $R(N_2)$, this latter choice being indicated by the second letter. Because of reverse complements, there is an even number of arcs in the DBG: if there is an arc from N_1 to N_2 then, necessarily, there is an arc from N_2 to N_1 (*e.g.* if the first arc has label FF then the second has label RR).

A DBG can be compressed without loss of information by merging *simple* nodes. A simple node denotes a node linked to at most two other nodes. Two adjacent simple nodes are merged into one by removing the redundant information. A valid path (see Definition 2) composed by $i > 1$ simple nodes is compressed into one node storing a sequence of length $k + (i-1)$ as each node adds one new character to the first node. Figure 1b represents a DBG (upper) and the corresponding compressed DBG (lower). In the remainder of the paper, we denote by cDBG a compressed DBG.

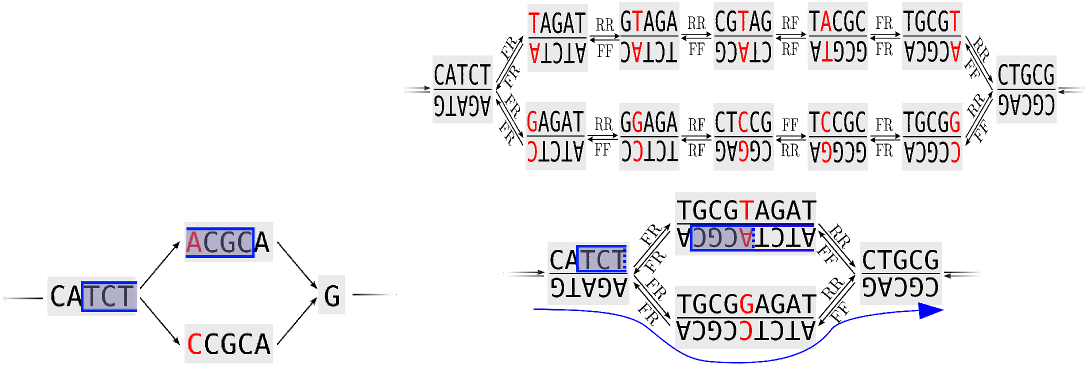
Definition 1 (Active strand of a node in a DBG). *The active strand of a node N in a DBG denotes which strand of the node, forward or reverse, is considered while traversing N.*

Definition 2 (Valid path). *The traversal of a node N is said to be valid if the rightmost label (F or R) of the arc used for entering the node is equal to the leftmost label of the arc used for leaving the node.*

A path in the graph is valid if for each node involved in the path, its traversal is valid, that is, each pair of adjacent arcs in the path are labelled, respectively, XY and YZ with $X, Y, Z \in \{R, F\}$.

Definition 3 (Sequence stored in a cDBG). *A valid path in a cDBG composed by ordered nodes $N_0, N_1, \ldots, N_l$, stores two sequences as following:*

1. *$s = F/R(N_0)_{\overset{\circ}{k}}F/R(N_1)_{\overset{\circ}{k}}\cdots_{\overset{\circ}{k}}F/R(N_l)$, the choice between R or F for node N_0 is equal the first label of the edge going from N_0 to N_1, while for $i \in [1, l]$, the choice between R or F for node N_i is equal the second label of the edge going from N_{i-1} to N_i.*
2. *$\overline{s}$.*



(a) SG (b) DBG

Figure 1: **(a)** Directed sequence graph. **(b)** Uncompressed (upper) and compressed (lower) de-Bruijn Graphs with $k = 5$. For each node, lower sequence is the reverse complement of the upper sequence, it should be read from right to left. **Boxes** both on (a) and (b): example of a seed of length 7 (TCTACGC) spread over 2 nodes. In the de-Bruijn Graph, the $k-1$ first characters of the second node are pruned due to overlap, and the reverse part of the second node is considered as the edge between the first (left) and the second (central) node is F**R**

For instance, the arrowed path on the cDBG presented Figure 1b, stores the sequences

$$\begin{aligned} s &= CATCT_{\mathring{k}}ATCTCCGCA_{\mathring{k}}CGCAG \\ &= CATCT.CCGCA.G \\ &= CATCTCCGCAG \end{aligned}$$

and

$$\bar{s} = CTGCGGAGATG$$

2.3 Approximate pattern matching in a graph (SG or cDBG)

Definition 4 (Approximate pattern matching in a graph). *Given a query Q, a graph $\mathcal{G}$ (SG or cDBG), and a parameter d, approximate pattern matching consists in finding all occurrences of Q in sequences stored in $\mathcal{G}$ within an edit distance of at most d.*

3 The BlastGraph algorithm

Blast like seed-based heuristics rest on the idea that if two sequences share some similarities, then there exists (at least) a common word (a seed) between these two sequences. Such algorithms consist in, first, anchoring the detection of similarities by exact matching of short sub-sequences, the seeds, and then, performing the similarity distance computation once sequences are anchored. The algorithm we propose applies these ideas between a graph (the bank) and a string (the query). It is divided into four main stages:

1. Index all seeds present in the graph $\mathcal{G}$.

2. Anchor query sequences to nodes of $\mathcal{G}$ using seeds. In the case of genomic data, reverse complement of query sequences may also be used as queries.
3. Align anchored query sequences on the left and right of the matched seeds.
4. Merge left and right alignments.

In the four following sections, we provide some more details for each of these four stages simply considering the graph as a SG. Then, in Section 4, we describe the modifications needed for applying the algorithm on a cDBG.

3.1 Stage 1: Indexing the seeds

Let n denote the length of the seeds. Each word of length n of the sequence of each node of $\mathcal{G}$ as well as those spread over several linked nodes are indexed using a hash table. The index contains for each seed a set of its occurrence positions.

Occurrence position in a graph: An occurrence position in the graph is defined as a couple (node identifier N, position on $S(N)$) indicating the starting position of the occurrence.

Seeds spread over more than one node: Any seed starting at less than n positions to the end of the sequence of a node is spread over more than one node. For instance, the seed $TCTACGC$ starting at position 2 on the leftmost node of Figure 1a, is spread over two distinct nodes. Seeds spread over more than one node are detected thanks to a depth first algorithm recursive approach.

In order to make a light index, the BLASTGRAPH algorithm only stores the starting position of a seed (node identifier N, position on $S(N)$) and not all possible nodes over which the seed is spread.

3.2 Stage 2: Anchoring query sequences to sequences of the graph

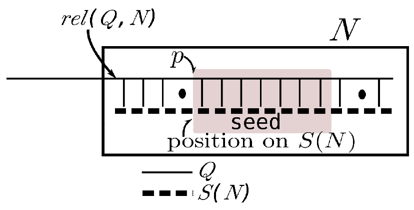


Figure 2: Value $rel(Q, N)$ while anchoring a query sequence Q on a node N with a seed

For each query sequence Q, all overlapping words of length n (seeds) are read. Let s be such a seed occurring at position p on Q, also having at least one occurrence in the graph. Then the index provides a set of couples (node N, position on $S(N)$). For each such couple, the query Q is anchored on the sequence $S(N)$, giving a relative position $rel(Q, N)$ of Q on $S(N)$. More precisely, $rel(Q, N) = p -$position on $S(N)$ (see Figure 2) is the position where Q aligns to $S(N)$. Note that $rel(Q, N)$ could be < 0 if a prefix of $S(N)$ is not aligned to Q. This is the case of $S(Q, L_1)$ in the example presented Figure 3.

Computing an alignment only once: If a node N and a sequence Q share more than one seed for the same alignment, each of them generate the same value $\{rel(Q,N)\}$. As this is a very usual case, in order to avoid computing several times the same alignments, while aligning sequence Q, the value $\{rel(Q,N)\}$ is stored in memory. Thus, the same alignment anchored at position $\{rel(Q,N)\}$ is computed only once.

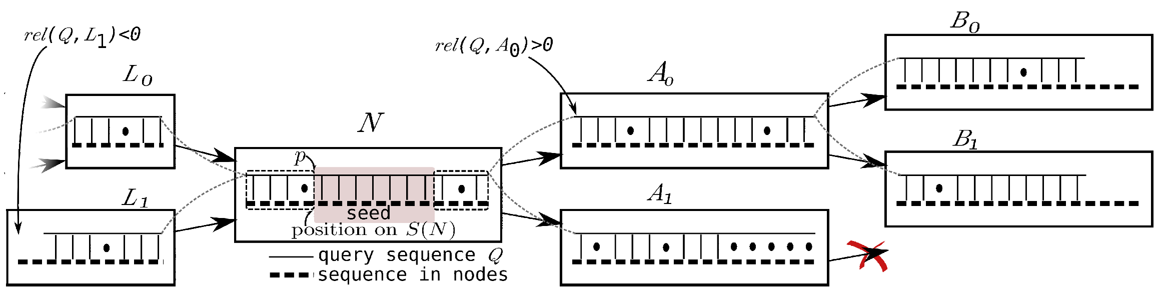


Figure 3: Overview of the alignment process. **Anchoring**: Using a seed, a query sequence Q is anchored to the node N. **Right alignment**: edit distance is computed between $Q[rel(Q,N)+i+k..]$ and $S(N)[i+k..]$ (right dotted square in node N), then between $Q[rel(Q,A_0)..]$ and $S(A_0)$, between $Q[rel(Q,B_0)..]$ and $S(B_0)$, between $Q[rel(Q,B_1)..]$ and $S(B_1)$, and so on. In this example, path using node A_1 presents an edit distance higher than the threshold; its children are not explored. **Left Alignment**: the same procedure is applied on the sequence on the left of the seed (left dotted square in node N), then on parents L_0, L_1 of node N, and so on…

3.3 Stage 3: Alignment between query sequence and sequence graph nodes

Given a query sequence Q anchored at position $\{rel(Q,N)\}$ in a node N of the graph, this stage computes all possible alignments (based on edit distance) between Q and all paths readable from node N (see Figure 3 for an example).

Computing the edit distance between two strings is a dynamic programming procedure that involves the usage of a matrix of size the product of the string lengths. However, in the particular case of this work, the user restricts the maximum edit distance for having a match. Consequently, the matrix computation is limited to a diagonal (see Figure 4 for an example) of width $\left\lfloor \frac{maximum\ edit\ distance}{cost\ indel} \right\rfloor \times 2$. Outside the diagonal, number of insertions or deletions becomes bigger than maximum number of insertions or deletions accepted equal to $\left\lfloor \frac{maximum\ edit\ distance}{cost\ indel} \right\rfloor$. Thus during this stage, the time and memory complexity for aligning query Q to one path of the graph is in $O(|Q|)$ considering *maximum edit distance* and *cost indel* as fixed parameters.

Right alignments The alignment is done between Q and $S(N)$ on the right of the matched seed. Additionally, as shown Figure 3, right extremity of the query sequence may finish after $S(N)$. In such a case the alignment has to be done on children $A_0, A_1, \ldots, A_n$ of node N. On each child A_i, the right extremity of the query sequence may finish after the $S(A_i)$, in this case, alignment continues on its children $B_0, B_1, \ldots, B_{n'}$, and so on. Thus, right part of sequence Q (starting after the anchored seed), may be aligned to $S(N \cdot A_i \cdot B_j \ldots)$. This is done via a recursive depth-first traversal of the graph, starting from N as long as the full right part of S is not aligned. An alignment between the sequence of a node and Q is never computed twice. For instance (Figure 3), if the alignment between Q and $S(N \cdot A_0 \cdot B_0)$ was computed,

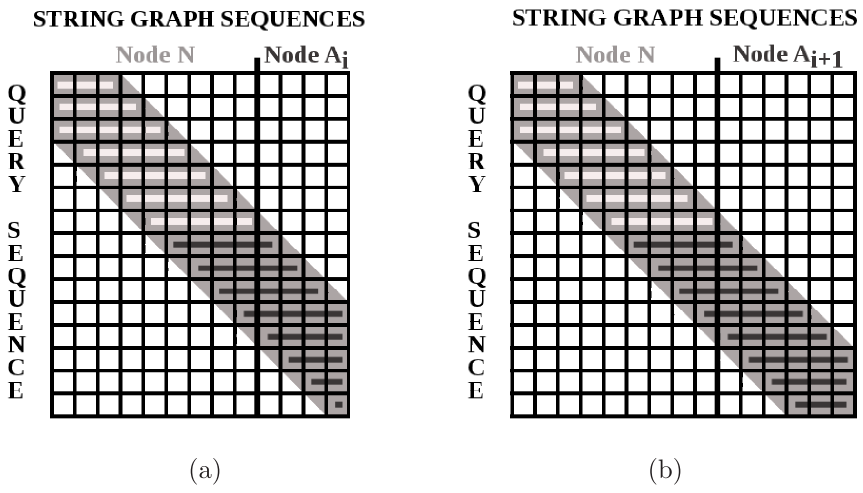


Figure 4: Dynamic programming matrix. Only the shadowed diagonal is computed. **(a)** distance computed between the query sequence S and $S(N.A_i)$. **(b)** distance computed between S and $S(N.A_{i+1})$. Lighter lines are not recomputed for computing matrix (b) if matrix (a) was already computed

the computation between Q and $S(N \cdot A_0 \cdot B_1)$ starts from the last full line of the alignment of Q with $S(N \cdot A_0)$. Thus the alignment between Q and $S(N)$ and $S(A_0)$ is never recomputed.

Left alignments Aligning the part of the sequence Q on the left of the seed to the graph is done using almost the same approach as the one previously described for right alignments. However, there are two main differences: 1) Sequences both from Q and from the nodes are reversed (read from right to left); 2) when the reversed query sequence is longer than the reversed sequence of a node N, the parents $L_0, L_1, ...$ of N are explored in depth first search approach (see Figure 3 for an example).

3.4 Joining left and right alignments

For a given aligned query sequence, each left alignment is compared to each right alignment. For each such couple whose sum of the cost of the alignments is below or equal the user defined maximum edit distance, the full alignment is reported.

4 BlastGraph on compressed de-Bruijn graphs

The three main differences between the SG and the cDBG are:

1. In the cDBG, the sequences of two connected nodes overlaps over $k-1$ characters. Thus, whatever the stage, the concatenation of the sequences of two nodes of the cDBG, has to be done removing the $k-1$ overlapping characters using the "$\underset{k}{\cdot}$" concatenation instead of the classical "$\cdot$" one.
2. In the cDBG, each node N stores a sequence ($F(N)$) and its reverse complement ($R(N)$).
3. Label of edges have to be considered while traversing the graph. Thus, in the cDBG, the general rule is the following: a node N is always traversed either as

forward ($F(N)$) or as reverse complement ($R(N)$), with F or R being its "active strand" (see Definition 1).
In the first case (resp. second case), accessing the children of the node is done following edges starting with the letter F (resp. R).
While following an edge, the active strand of the targeted node F (resp. R) is the second letter of the label of the edge.

Seeding in a DBG: The seeding approach is the same than the one applied on the SG. By convention, all seeds start on forward sequence of each node. This is done without loss of information as each query is considered both in its forward and its reverse complement directions.

Right extension in a DBG: The right extension in a DBG is the same as the one described for a SG. However, the algorithm takes into account some DBG specificities:
– query sequence is mapped on $F(N)$ (seeds are indexed only on forward strands);
– children of a node N are reached using only outgoing edges whose label first character corresponds to the active strand of N, and, once a child is reached, its active strand is the one corresponding to the second character of this label;
– concatenation of sequences of two linked nodes is done pruning the overlapping $k-1$ characters.

Left extension in a DBG: Left extensions in the DBG are done by right extending the reverse complement $\overline{Q}$ of the sequence Q to the DBG, starting from the reverse strand of the node N: $R(N)$.

5 Results

Two prototype versions of this algorithm are implemented. Under the CeCILL License, they can be downloaded here: `http://alcovna.genouest.org/blastree`.
A Java version is implemented in a Cytoscape plug-in. Cytoscape [10] is an open-source platform for visualization and interaction with complex graph, especially in bioinformatics. The second version is implemented in C and can be run under Unix platforms. In the two prototypes, while working on nucleotides, characters are coded in two bits.

The next section proposes a use case of the Java version, while section 5.2 proposes some results over the C prototype.

5.1 Use case

We present in this section a use case, on a toy example. We created a sequence graph containing five nodes (Figure 5). We searched for the sequence

$$ggcgTtcagac/cTatacgcatacgcagcagact/agCctacg,$$

spread over 3 nodes of this graph and containing two mismatches and one insertion. To help the reader, we indicated here substitutions and indel with an upper case letter and we indicated separations between nodes with a '/' character. Of course the practical query sequence is a raw un-annotated sequence. We fixed the cost of a mismatch to 1, the cost of an indel to 2 and the maximum edit distance to 4. BlastGraph (Cytoscape plug-in version)) found the correct path, as presented in Figure 5 where selected nodes are those in which alignment is found between the query and the graph.

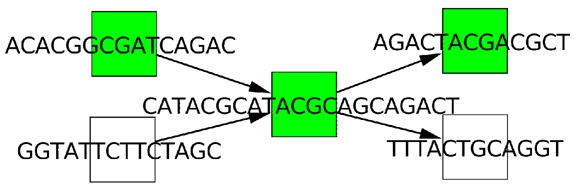


Figure 5: Cytoscape view of the selected nodes (green) in the sequence graph after the research of query sequence

5.2 Performances on DBG

We present results obtained in a typical use case while applying BlastGraph on a DBG graph. Results were obtained on a 64 bit 2×2.5 GHz dual-core computer with 3 MB cache and 4 GB RAM memory. From the NCBI Sequence Read Archive (SRA, `http://www.ncbi.nlm.nih.gov/Traces/sra`), we downloaded the DRR000096 Illumina run containing approximately 4 million reads and approximately 150 million nucleotides.

Increasing graph sizes Subsets of different sizes were generated by randomly sampling DRR000096 reads. For each subset, we constructed the de-Bruijn graph using $k = 31$. Table 1 reports the total number of nodes and nucleotides stored in some of these graphs.

No Reads	No Nodes	No nucleotides
10K	59K	1833K
100K	573K	17774K
150K	849K	26306K

Table 1: Total number of nodes and nucleotides stored in the graph with respect to the number of reads

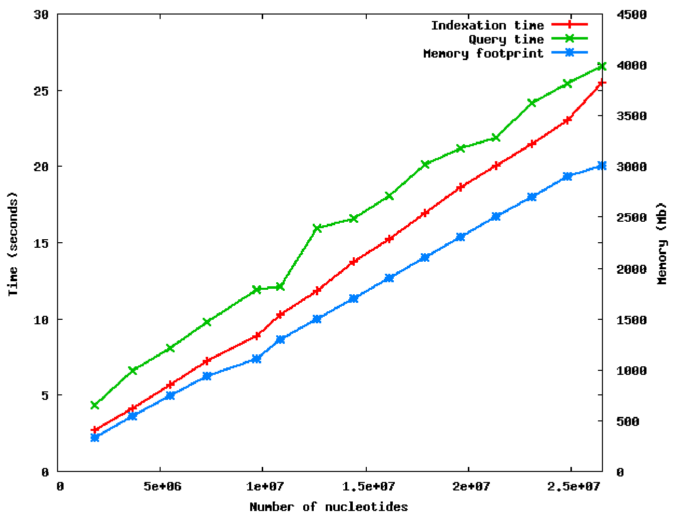


Figure 6: Time and memory consumption with respect to the number of nucleotides stored in the graph

On each graph, we applied the C version of BlastGraph, aligning a set of 10000 query sequences derived from the initial read set. We used seeds of length 19, a mismatch cost equal to 1 and an indel cost equal to 2 and a maximum edit distance equal to 5. We report in Figure 6 time and memory needed both for constructing the index and for performing the 10000 queries.

We can observe that memory footprint and both indexation and query execution times increase linearly with the quantity of information contained in the graph. While memory usage is the main bottleneck of this approach, the indexation and query time are acceptable. Even on the biggest tested graph (containing more than 26 million characters stored in approximately 849000 distinct nodes), indexation is done in 26 seconds and the 10000 queries are performed in less than 52 seconds.

Increasing number of queries In order to measure the impact of the number of queries on the execution time, we used the graph composed of 100000 reads from the DRR000096 data set using $k = 31$. We ran BlastGraph using queries dataset composed of $500, 1000, \ldots, 10000$ reads taken from the 100000 reads used for creating the graph. We report the query time (not including the indexation time) Figure 7. We note that, as expected, the query time increases slowly and linearly with the number of queries.

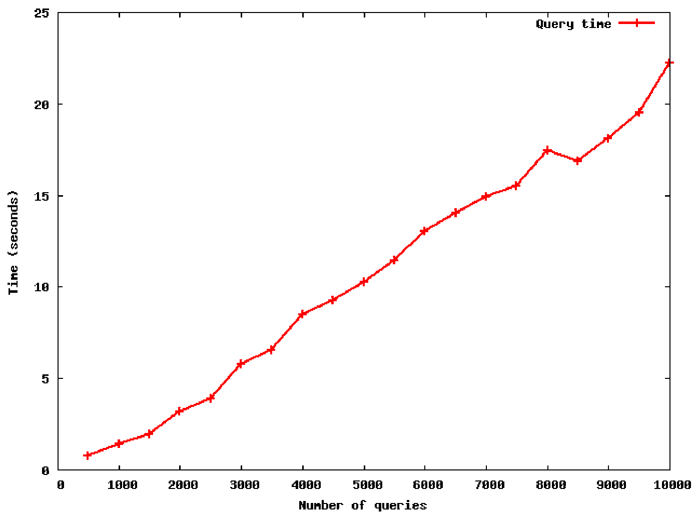


Figure 7: Query time with respect to the number of queries. Note that reported time do not include the indexation time equal to 16 seconds independently of the number of queries

6 Conclusion

We presented BlastGraph, a new algorithm for performing intensive approximate string matching between a set of query sequences and a directed sequence graph including the application to de-Bruijn graphs. This blast-like algorithm presents novelties with respect to “classical” blast-like approaches as seeds and alignments may be spread over several nodes and as the algorithm takes into account double stranded de-Bruijn graph features. Results showed that BlastGraph performances permit its usage on quite large graphs in reasonable time.

The main bottleneck of the approach comes from the memory footprint. Storing in memory graphs containing hundreds of millions of nucleotides together with seed index is challenging. Future work will include either an adaptation of BlastGraph to extremely light DBG representation [4] or a non indexed version of the algorithm, for instance based on KMP [6] algorithm. This will increase the query time, while decreasing the memory usage. Possible applications will exceed the frontiers of the current work as this problem is central in many algorithms associated to high throughput sequencing problems.

7 Acknowledgements

Authors warmly thank Vincent Lacroix and François Coste for their participation to discussions. This work was born from and supported by the Inria “action de recherche collaborative” ARC Alcovna `http://alcovna.genouest.org/`.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215(3) 1990, pp. 403–410.
2. M. J. Chaisson, D. Brinza, and P. A. Pevzner: *De novo fragment assembly with short mate-paired reads: Does the read length matter?* Genome Research, 19(2) 2009, pp. 336–346.
3. M. J. Chaisson and P. A. Pevzner: *Short read fragment assembly of bacterial genomes.* Genome Research, 18(2) 2008, pp. 324–330.
4. R. Chikhi and G. Rizk: *Space-efficient and exact de Bruijn graph representation based on a Bloom filter*, in WABI, 2012, p. to appear.
5. S. Feature: *Next-generation sequencing transforms today's biology.* Most, 5(1) 2008, pp. 16–18.
6. D. E. Knuth, J. James H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
7. R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang: *De novo assembly of human genomes with massively parallel short read sequencing.* Genome Research, 20(2) 2010, pp. 265–272.
8. P. Medvedev, K. Georgiou, G. Myers, and M. Brudno: *Computability of models for sequence assembly*, in WABI, 2007, pp. 289–301.
9. P. A. Pevzner, H. Tang, and M. S. Waterman: *An Eulerian path approach to DNA fragment assembly.* Proceedings of the National Academy of Sciences of the United States of America, 98(17) Aug. 2001, pp. 9748–53.
10. M. E. Smoot, K. Ono, J. Ruscheinski, P.-L. Wang, and T. Ideker: *Cytoscape 2.8: new features for data integration and network visualization.* Bioinformatics, 27(3) 2011, pp. 431–432.
11. A. Viterbi: *Error bounds for convolutional codes and an asymptotically optimum decoding algorithm.* Information Theory, IEEE Transactions on, 13(2) april 1967, pp. 260 –269.
12. D. R. Zerbino and E. Birney: *Velvet: Algorithms for de novo short read assembly using de Bruijn graphs.* Genome Research, 18(5) 2008, pp. 821–829.

A Multiobjective Approach to the Weighted Longest Common Subsequence Problem

David Becerra, Juan Mendivelso, and Yoan Pinzón

Universidad Nacional de Colombia
Facultad de Ingeniería
Department of Computer Science and Industrial Engineering
Research Group on Algorithms and Combinatorics (ALGOS-UN)
Carrera 30 No. 45-03. Edificio 453. Oficina 207. Bogotá, Colombia
{dcbecerrar,jcmendivelsom,ypinzon}@unal.edu.co

Abstract. Finding the Longest Common Subsequence in Weighted Sequences (WLCS) is an important problem in computational biology and bioinformatics. In this paper, we model this problem as a multiobjective optimization problem. As a result, we propose a novel and efficient algorithm that not only finds a WLCS but also the set of all possible solutions. The time complexity of the algorithm depends primarily on the number of length-1 common subsequences between the two input weighted sequences.

Keywords: longest common subsequence, weighted sequences, multiobjective optimization, bioinformatics

1 Introduction

Algorithmic studies over molecular data have allowed the concomitant development of valuable analysis in biological processes. Specifically, studies on comparative genomics have lead to the development of powerful data analysis tools that have been successfully applied in several contexts from gene functional annotation to phylogenomics and whole genome comparison [4].

Since the publication of the human genome in 2001 [11], weighted sequences, also called position weight matrices [13], have become a major area of research in computational biology. A weighted sequence can be defined as a sequence of character-sets where, at each position of the sequence, each character is associated to a weight (or frequency). Thus, weighted sequences allow a newer and more precise encoding paradigm that allows to model several biological processes. For instance, in a molecular weighted sequence, the characters can represent either nucleotides or amino acids, and the weight can model either the occurrence probability of a character or the stability contributed by the character to a molecular complex [9]. Weighted sequences can be used to represent a variety of sequence lengths from short sequences, like protein binding sites, to much larger sequences such as profiles of protein families or even a complete chromosome sequence [6].

In recent years, different research groups have been modeling several other biological processes through weighted sequences. In [14], weighted sequences were used to propose a simple modeling of the translation of gene expression and regulation. Later, new weighted sequence algorithms were given for DNA approximate matching [2]. Moreover, in [10], a novel data structure called weighted suffix tree was introduced. This structure can be used to compute repeats and covers, as well as to detect the longest common substring. More recently, weighted sequences were used to model gene expression data derived from DNA micro-array analysis [15]. Despite its wide range of utilities, it is important to consider that, in most sequence comparison methods, the quality of the results are significantly affected by small perturbations in the algorithmic methods and the data. Furthermore, there is a dearth of computational tools to compare sequences beyond a certain length and quality [3].

The Longest Common Subsequence (LCS) of a given set of sequences is one of the most used similarity measures in computational biology. Computing the LCS has been analytically shown to be intractable (NP- hard in the strong sense) even for sequences over a binary alphabet [12]; however, it can be solved for a fixed number of sequences in polynomial time via standard dynamic programming algorithms [7]. Then, the development of adequate algorithms for the variants of the LCS has become an increasing necessity, considering that efficient algorithms are an undeniable requirement for the analysis of high throughput sequencing data.

Particularly, the Weighted Longest Common Subsequence (WLCS) is a similarity measure between weighted sequences. The WLCS of two weighted sequences, X and Y, is the longest common subsequence such that the product of the weights associated to each character in the subsequence is greater or equal to given bounds for X and Y. It was proven that computing the WLCS is NP-hard for unbounded alphabets [1]; opposite to the case of the LCS, the tractability of the problem for a bounded alphabet is still an open problem. To the best of our knowledge, there are only two algorithms to solve the WLCS problem. Specifically, a $(1/|\Sigma|)$-approximation algorithm was presented in [1]; more recently, this algorithm was improved by means of a polynomial-time approximation scheme in [5]. In this paper, we propose an exact algorithm, based on a new concept of dominance, to find the WLCS of two weighted sequences over bounded alphabets (DNA). As an advantage, the proposed algorithm returns not only the longest common subsequence (and its length), but also the set of all dominant common subsequences.

The outline of the paper is as follows. The next section provides some definitions necessary to understand the essential features of this paper. Then, the framework to tackle the WLCS problem as a simple multiobjective problem is presented in §3. In §4, the proposed algorithm is described along with its time complexity analysis and an example. The concluding remarks are drawn in the last section.

2 Preliminaries

Let Σ be an alphabet of cardinality $\sigma = |\Sigma|$ which consists of a set of symbols.

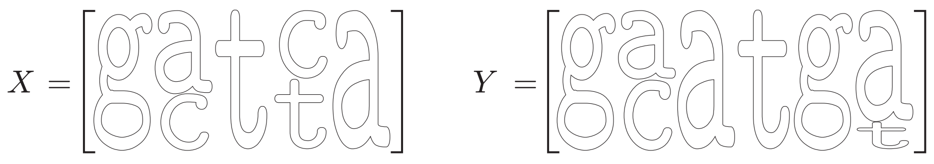
Definition 1 (LONGEST COMMON SUBSEQUENCE). *The* LCS *problem for two input strings x and y consists of finding the longest sequence p such that p is a subsequence of both x and y.*

Definition 2 (WEIGHTED SEQUENCE). *A* weighted sequence $X = X[1]\cdots X[n]$ *over the alphabet Σ is a sequence of n sets $X[i]$, $1 \leq i \leq n$. Each set $X[i]$ is comprised of pairs $(s_j, \pi_i^X(s_j))$, where $s_j \in \Sigma$ and $\pi_i^X(s_j)$ is the* occurrence probability *of the character s_j at location i. Additionally, $\sum_j \pi_i^X(s_j) = 1$ for every position $1 \leq i \leq n$. For convenience and brevity, we will refer to $X[i]$ as the set of characters occurring at position i with a probability greater than zero.*

For a finite alphabet $\Sigma = \{s_1, \ldots, s_\sigma\}$, we can view a length-$n$ weighted sequence X as a $|\Sigma| \times n$ matrix A, where $A[j,i] = \pi_i^X(s_j)$. The elements of this matrix represent the occurrence probability of each character in every position of the sequence. Thus, matrix A is comprised of values within the real interval [0,1]. As an example, two weighted sequences are illustrated in Fig. 1. Notice that for a given character the occurrence probability can be different at each position; however, for a given position, the occurrence probability of all characters must sum 1.

$$X = \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{ccccc} 0_{\mathtt{a}} & 0.6_{\mathtt{a}} & 0_{\mathtt{a}} & 0_{\mathtt{a}} & 1_{\mathtt{a}} \\ 0_{\mathtt{c}} & 0.4_{\mathtt{c}} & 0_{\mathtt{c}} & 0.5_{\mathtt{c}} & 0_{\mathtt{c}} \\ 1_{\mathtt{g}} & 0_{\mathtt{g}} & 0_{\mathtt{g}} & 0_{\mathtt{g}} & 0_{\mathtt{g}} \\ 0_{\mathtt{t}} & 0_{\mathtt{t}} & 1_{\mathtt{t}} & 0.5_{\mathtt{t}} & 0_{\mathtt{t}} \end{array}\right] \end{array} \qquad Y = \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \left[\begin{array}{cccccc} 0_{\mathtt{a}} & 0.5_{\mathtt{a}} & 1_{\mathtt{a}} & 0_{\mathtt{a}} & 0_{\mathtt{a}} & 0.8_{\mathtt{a}} \\ 0_{\mathtt{c}} & 0.5_{\mathtt{c}} & 0_{\mathtt{c}} & 0_{\mathtt{c}} & 0_{\mathtt{c}} & 0_{\mathtt{c}} \\ 1_{\mathtt{g}} & 0_{\mathtt{g}} & 0_{\mathtt{g}} & 0_{\mathtt{g}} & 1_{\mathtt{g}} & 0_{\mathtt{g}} \\ 0_{\mathtt{t}} & 0_{\mathtt{t}} & 0_{\mathtt{t}} & 1_{\mathtt{t}} & 0_{\mathtt{t}} & 0.2_{\mathtt{t}} \end{array}\right] \end{array}$$

(a)



(b)

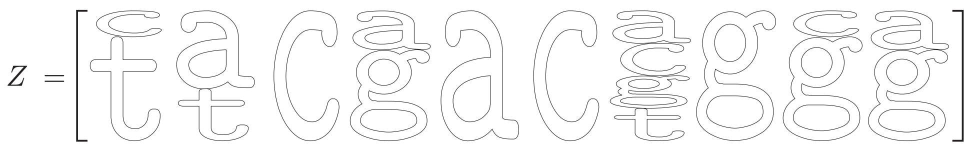
Figure 1. Example of 2 weighted sequences drawn from the alphabet $\Sigma_{\text{DNA}} = \{a, c, g, t\}$ shown in **(a)** as a matrix and in **(b)** as a 2-dimensional pictogram

Definition 3 (SUBSEQUENCE OF A WEIGHTED SEQUENCE). *For a given weighted sequence $X = X[1] \cdots X[n]$, $p = p_{i_1} \cdots p_{i_{|p|}}$ (where $p_{i_j} \in \Sigma$, $1 \leq j \leq |p|$ and $1 \leq i_1 < i_2 < \cdots < i_{|p|} \leq n$) is a* subsequence *of X iff $p_{i_j} \in X[i_j]$ for $1 \leq j \leq |p|$.*

Definition 4 (OCCURRENCE PROBABILITY OF A SUBSEQUENCE). *Let $p = p_{i_1} \cdots p_{i_{|p|}}$ be a subsequence of a weighted sequence $X = X[1] \cdots X[n]$. The* occurrence probability *of the subsequence p with respect to X, denoted as $\pi^X(p)$, is given by $\prod_{j=1}^{|p|} \pi^X_{i_j}(p_{i_j})$.*

See Fig. 2 for an example illustrating this last definition.

$$Z = \begin{array}{c} \begin{array}{cccccccccc} & i_1 & i_2 & i_3 & & & i_4 & & & i_5 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \left[\begin{array}{cccccccccc} 0_{\mathtt{a}} & 0.6_{\mathtt{a}} & 0_{\mathtt{a}} & 0.3_{\mathtt{a}} & 1_{\mathtt{a}} & 0_{\mathtt{a}} & 0.25_{\mathtt{a}} & 0_{\mathtt{a}} & 0_{\mathtt{a}} & 0.3_{\mathtt{a}} \\ 0.2_{\mathtt{c}} & 0_{\mathtt{c}} & 1_{\mathtt{c}} & 0_{\mathtt{c}} & 0_{\mathtt{c}} & 1_{\mathtt{c}} & 0.25_{\mathtt{c}} & 0_{\mathtt{c}} & 0.2_{\mathtt{c}} & 0_{\mathtt{c}} \\ 1_{\mathtt{g}} & 0_{\mathtt{g}} & 0_{\mathtt{g}} & 0.7_{\mathtt{g}} & 0_{\mathtt{g}} & 0_{\mathtt{g}} & 0.25_{\mathtt{g}} & 1_{\mathtt{g}} & 0.8_{\mathtt{g}} & 0.7_{\mathtt{g}} \\ 0.8_{\mathtt{t}} & 0.4_{\mathtt{t}} & 0_{\mathtt{t}} & 0_{\mathtt{t}} & 0_{\mathtt{t}} & 0_{\mathtt{t}} & 0.25_{\mathtt{t}} & 0_{\mathtt{t}} & 0_{\mathtt{t}} & 0_{\mathtt{t}} \end{array}\right] \end{array}$$



$p =$ `acata` with i_j=(2,3,4,7,10) $\qquad \pi^Z(p)$=0.6×1×0.3×0.25×0.3=0.0135

Figure 2. Example of a subsequence p extracted from a weighted sequence Z. Note that $p' = acata$ with $i_j = (2, 3, 5, 7, 10)$ is also a subsequence of Z but $\pi^Z(p') = 0.045$

Definition 5 (WEIGHTED LONGEST COMMON SUBSEQUENCE PROBLEM[1]). *Let $X = X[1] \cdots X[n]$ and $Y = Y[1] \cdots Y[m]$ be two weighted sequences. For two given constants α_1 and α_2 with $0 < \alpha_1, \alpha_2 \leq 1$, the* weighted longest common subsequence *is*

[1] Also known as the Longest Common Weighted Subsequence Problem with Two Thresholds.

the maximal integer length ℓ such that there is a common subsequence of length ℓ, $p = p_{i_1} \cdots p_{i_\ell}$, for which $\pi^X(p) \geq \alpha_1$ AND $\pi^Y(p) \geq \alpha_2$.

Definition 6 (DOMINANCE BETWEEN TWO COMMON SUBSEQUENCES). *Let $X = X[1] \cdots X[n]$ and $Y = Y[1] \cdots Y[m]$ be two weighted sequences where $n \leq m$. Also, let p and q be two length-h common subsequences of both X and Y where p and q occur in X at indices $(i_{x_1}, \ldots, i_{x_h})$ and $(j_{x_1}, \ldots, j_{x_h})$, respectively. Similarly, p and q occur at Y at indices $(i_{y_1}, \ldots, i_{y_h})$ and $(j_{y_1}, \ldots, j_{y_h})$, respectively. We say that p dominates q, denoted as $p \prec q$, iff:*

i) $\pi^X(p) \geq \pi^X(q)$
ii) $\pi^Y(p) \geq \pi^Y(q)$
iii) $i_{x_h} < j_{x_h}$
iv) $i_{y_h} < j_{y_h}$

Notice that if p dominates q, then the positions of the last character of p, in both X and Y, are lower than those of q. This fact is useful given that a common subsequence that ends at a lower index will have a better chance of being extended; thus, it may lead to longer common subsequences. On the other hand, the occurrence probabilities of p, with respect to X and to Y, are greater or equal to the ones of q. Consequently, if p dominates q, a possible LCS containing p will be at least as good as the LCS containing q.

Definition 7 (MULTIOBJECTIVE OPTIMIZATION PROBLEM – MOOP). *Find a vector $x = [x_1, x_2, \ldots, x_n]^T$ that:*

i) satisfies the r equality constraints $h_i(x) = 0,\ 1 \leq i \leq r$,
ii) is subject to the s inequality constraints $g_i(x) \geq 0,\ 1 \leq i \leq s$, and
iii) optimizes the vector function $f(x) = [f_1(x), \ldots, f_m(x)]^T$.

Then, it is clear that a MOOP problem focuses on searching for the optimal values of the decision variables (vector x) that minimize/maximize the objective function vector $f(x)$ while satisfying the constraints. The vector x is an n-dimensional decision vector or solution and $\mathcal{X}$ is the decision space, *i.e.*, the set of all expressible solutions. The objective vector $z = f(x)$ maps $\mathcal{X}$ into $\Re^m$, where $m \geq 2$ is the number of objectives. The image of $\mathcal{X}$ in the objective space, denoted as $\mathcal{Z}$, is the set of all attainable points (see Fig. 3).

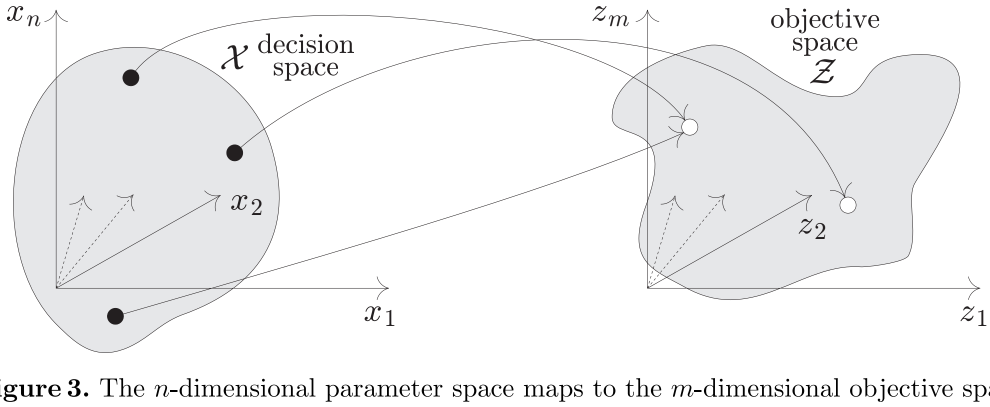


Figure 3. The n-dimensional parameter space maps to the m-dimensional objective space

3 WLCS as a Multiobjective Optimization Problem

In any multiobjective problem, two spaces should be defined: the *decision space* (the set of all expressible solutions), and the *objective space* (the space where the image of the solutions is the set of all attainable solutions). In general, to model any specific problem as a MOOP, three basic sets should be established: a set of objective functions, a set of decision variables and a set of equality/inequality constraints.

Particularly, for the WLCS problem, the length of the LCS should be maximized under two weight restrictions. Moreover, more than one possible direction of extending the substrings must be considered. Given two input weighted sequences, X and Y, and two constants $\alpha_1 > 0$ and $\alpha_2 > 0$, let $p = p_{i_1} \cdots p_{i_h}$ be a length-h common subsequence of X and Y that occurs at indices $(i_{x_1}, \ldots, i_{x_h})$ and $(i_{y_1}, \ldots, i_{y_h})$, respectively. Then, we define the elements of the MOOP model as follows:

– **Decision Space.** A vector with three components will be used as the decision vector. The first component is the list of symbols of the subsequence, *i.e.* $p_{i_1} \cdots p_{i_h}$. The second component is the list of indices at which the subsequence p occurs in X, *i.e.*, $(i_{x_1}, \ldots, i_{x_h})$. Finally, the third component is the list of indices at which the subsequence p occurs in Y, *i.e.*, $(i_{y_1}, \ldots, i_{y_h})$.

– **Objective space.** A vector with four components will be used as the objective function. The first objective, which will be maximized, is the occurrence probability of the subsequence with respect to X, *i.e.*, $\pi^X(p)$. The second objective, which will also be maximized, is the occurrence probability of the subsequence with respect to Y, *i.e.*, $\pi^Y(p)$. The third and fourth objectives, which will be minimized, are the positions of the last symbol of the subsequence in X, *i.e.* i_{x_h}, and in Y, *i.e.*, i_{y_h}, respectively. Then, the dominance relation between two common subsequences can be easily checked (*c.f.* see Definition 6).

– **Feasible regions.** The following constraints establish the feasible regions: the occurrence probabilitities of the subsequence p with respect to X and to Y must be greater or equal to α_1 and α_2, respectively. That is, $\pi^X(p) \geq \alpha_1$ and $\pi^Y(p) \geq \alpha_2$.

The proposed algorithm maximizes the length of the longest common subsequence by means of the objective functions. Its core idea is building different pareto optimum solutions, by using Definition 6, until the LCS is found in one of these optimum sets. Specifically, the algorithm iteratively finds new decision spaces as the result of the concatenation between the former pareto optimum set and the decision space of matches. The new decision space contains all the subsequences of length ℓ, while the former decision space contained all the subsequences of length $\ell - 1$. Then, these solutions are mapped to new objective spaces from which only the new pareto optimum set is extracted. The algorithm will continue until a new decision space can no longer be created (see Fig. 4, for an illustration of this process). In the next section, we describe in detail the proposed algorithm.

4 MiCO: An Algorithm for the WLCS Problem

Given two input weighted sequences, $X = X[1] \cdots X[n]$ and $Y[1] \cdots Y[m]$, and two given constants α_1 and α_2, where $0 < \alpha_1, \alpha_2 \leq 1$, our algorithm computes the

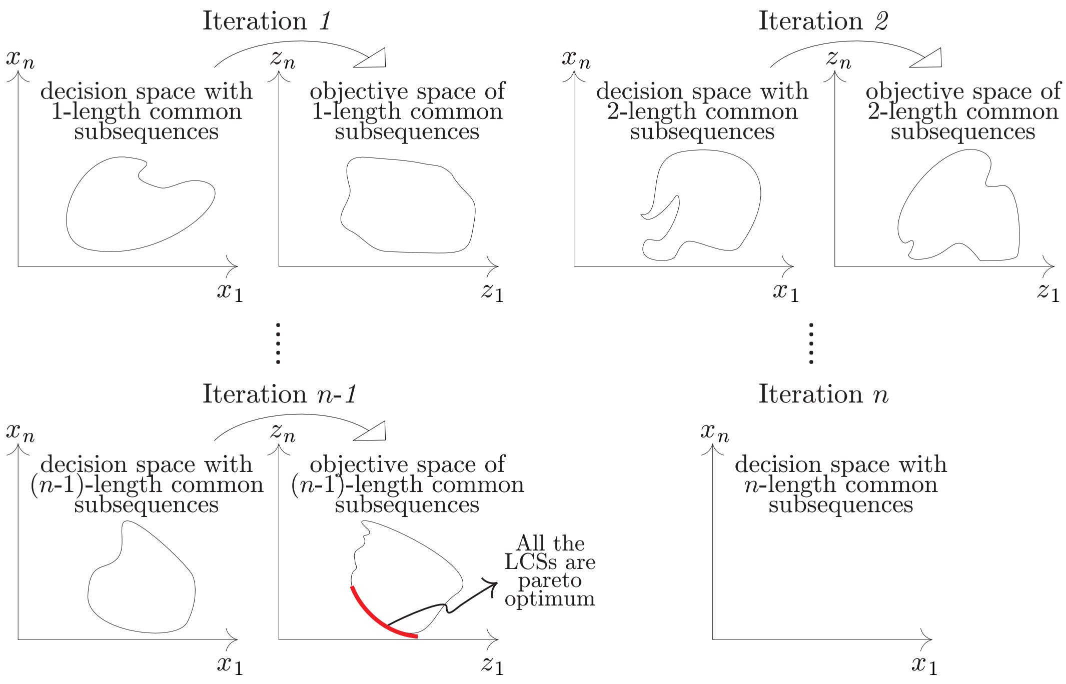

Iteration *1*
x_n
decision space with 1-length common subsequences
x_1
z_n
objective space of 1-length common subsequences
z_1
Iteration *2*
x_n
decision space with 2-length common subsequences
x_1
z_n
objective space of 2-length common subsequences
z_1
Iteration *n-1*
x_n
decision space with (n-1)-length common subsequences
x_1
z_n
objective space of (n-1)-length common subsequences
z_1
All the LCSs are pareto optimum
Iteration n
x_n
decision space with n-length common subsequences
x_1

Figure 4. Iterations of the algorithm between the decision and objective spaces

weighted longest common subsequence of X and Y, along with all dominant common subsequences. The algorithm's framework is as follows:

- **Step 1 [computing 1-length common subsequences]:** Find a set Δ of all length-1 common subsequences of X and Y.
- **Step 2 [finding dominant common subsequences]:** Compute the set $\mathcal{D}_i$ of all the length-i common subsequences of X and Y that are not dominated by any other common subsequence. First, the dominant subsequences from Δ are added to $\mathcal{D}_1$. Then, the following two phases are performed for each $\delta \in \Delta$:
  - **Phase 1 [inserting concatenations with δ]:** Updates $\mathcal{D}_i$ by inserting all the new length-i common subsequences, for $i \geq 2$, resulting from the concatenation between each $d \in \mathcal{D}_{i-1}$ and the given δ.
  - **Phase 2 [deleting dominated subsequences]:** Updates $\mathcal{D}_i$ by deleting all the elements that were dominated by the common subsequences inserted during Phase 1.

While Step 1 is straightforward to implement, Step 2 is perhaps the trickiest step and needs a bit more of attention. Both of these steps are described below in more detail.

Step 1: To achieve the goal of this step, we begin by considering a matrix like the matrix shown in Fig. 6**(a)**. In such matrix $X[i]$, $1 \leq i \leq n$, and $Y[j]$, $1 \leq j \leq m$, represent a column and a row, respectively. We now can proceed to compute all the length-1 common subsequences in a *row-wise* fashion from top to bottom, as follows: First, X is traversed (from left to right) and all the positions i for which $\pi_i^X(\gamma) \geq \alpha_1$, for each character γ, are inserted into $List_X[\gamma]$. Thereafter, Y is traversed (from top to bottom) and, for each position j in Y, all the lists $List_X[\gamma]$ for which $\pi_j^Y(\gamma) \geq \alpha_2$ are merged-sorted into a list Γ containing (ℓ, γ), where $\ell \in List_X[\gamma]$. Then, we add the triplet (j, ℓ, γ) to Δ for every $(\ell, \gamma) \in \Gamma$. Fig. 6**(b–c)** illustrates this operation on our running example.

Step 2: This is an iterative process that is performed once for each element $\delta \in \Delta$ computed in the previous step. Broadly, the main idea is to use Δ to compute the set $\mathcal{D}_i$ of the common subsequences with the following invariant condition of stability: *no element (common subsequence) in* $\mathcal{D}_i$ *dominates any other element in* $\mathcal{D}_i$. This basically means two things: *i*) a new common subsequence can be inserted into $\mathcal{D}_i$ iff it is not dominated by any other common subsequence currently in $\mathcal{D}_i$, and *ii*) all the common subsequences that are dominated by the new arriving subsequence should be deleted from $\mathcal{D}_i$.

```
MiCO Algorithm
Input: $X$, $Y$, $\Sigma$, $\alpha_1$, $\alpha_2$
Local Variables: $n \leftarrow |X|$, $m \leftarrow |Y|$, $\Delta \leftarrow \emptyset$
Output: $\mathcal{D}$
1  for $i \leftarrow 1$ to $n$ do
2    forall $\gamma \in \Sigma$ do
3        if $\pi_i^X(\gamma) \geq \alpha_1$ then    $List_X[\gamma]$.add($i$)
4  for $j \leftarrow 1$ to $m$ do
5    $\Gamma \leftarrow \emptyset$
6    forall $\gamma \in \Sigma$
7        if $\pi_j^Y(\gamma) \geq \alpha_2$ then $\Gamma$.mergeSort($List_X[\gamma], \gamma$)
8    forall $(\ell, \gamma) \in \Gamma$ do $\delta \leftarrow (j, \ell, \gamma)$, $\Delta$.add($\delta$)
9        if !Dominate($\delta, \mathcal{D}_1$) then
10           $\mathcal{D}_1$.add($\delta$)
11  $i \leftarrow 2$
12  forall $d \in \mathcal{D}_{i-1}$ do
13   forall $\delta \in \Delta$ do
14           if ( Concatenate($d, \delta$) and !Dominate($d\delta, \mathcal{D}_i$))then
15               $\mathcal{D}_i$.add($d\delta$)
16               Delete($d\delta, \mathcal{D}_i$)
17   if $\mathcal{D}_i = \emptyset$ then
18               return $\mathcal{D}_{i-1}$
19  $i \leftarrow i + 1$
```

Figure 5. MiCO Algorithm

The elements of $\mathcal{D}_1$ are the dominant length-1 subsequences extracted from Δ, while the elements of $\mathcal{D}_i$, for $i > 1$, are the dominant length-i subsequences generated by concatenating every $d \in \mathcal{D}_{i-1}$ with the elements $\delta \in \Delta$. When in Step 2 no subsequence is inserted, i.e. $\mathcal{D}_i = \emptyset$, this means that the length of the weighted LCS is $i-1$ and the algorithm returns all the set $\mathcal{D}_{i-1}$. In order to derive our final algorithm we need to define the following procedures:

- *Procedure* Concatenate(d, δ): Returns *true* if subsequence d can be concatenated with subsequence δ, (*i.e.* the positions of the row and the column of the last character of subsequence d are lower than the row and the column of δ, respectively), or *false* otherwise.
- *Procedure* Dominate($d\delta, \mathcal{D}$): Returns *true* if subsequence $d\delta$ is dominated by any element $d \in \mathcal{D}$, or *false* otherwise.

– *Procedure* Delete($d\delta, \mathcal{D}$): Deletes all the elements of $\mathcal{D}$ that are dominated by d.

The pseudocode for the MiCO algorithm is presented in Fig. 5. The time complexity analysis and an example are presented in §4.1 and §4.2.

Y \ X	1 1g	2 0.6a 0.4c	3 1t	4 0.5c 0.5t	5 1a
1 1g	●				
2 0.5a 0.5c		● ●		●	●
3 1a		●			●
4 1t			●	●	
5 1g	●				
6 0.8a 0.2t		●			●

(a)

Y \ X	1 1g	2 0.6a 0.4c	3 1t	4 0.5c 0.5t	5 1a
1 1g	δ_1 g				
2 0.5a 0.5c		δ_2 a δ_3 c		δ_4 c	δ_5 a
3 1a		δ_6 a			δ_7 a
4 1t			δ_8 t	δ_9 t	
5 1g	δ_{10} g				
6 0.8a 0.2t		δ_{11} a			δ_{12} a

(b)

$List_X$[a] = 2,5	for j = 1, {1}, δ_1=(1,1,g)
$List_X$[c] = 2,4	for j = 2, {2,2,4,5}, δ_2=(2,2,a), δ_3=(2,2,c), δ_4=(2,4,c), δ_5=(2,5,a)
$List_X$[g] = 1	for j = 3, {2,5}, δ_6=(3,2,a), δ_7=(3,5,a)
$List_X$[t] = 3,4	for j = 4, {3,4}, δ_8=(4,3,t), δ_9=(4,4,t)
	for j = 5, {1}, δ_{10}=(5,1,g)
	for j = 6, {2,5}, δ_{11}=(6,2,a), δ_{12}=(6,5,a)

(c)

Figure 6. **(a)** Matrix representation of weighted sequences X and Y, **(b)** computation trace of Step 1, **(c)** resulting length-1 common subsequences for $\alpha_1 = \alpha_2 = 0.2$

4.1 Time Complexity Analysis

The time complexity of Step 1 (see MiCO pseudo-code, lines 1–8) is bounded by $\mathcal{O}(\sigma n + \sigma m) = \mathcal{O}(\sigma m)$. This process can be done in $\mathcal{O}(m \log m)$ time for the case where the sequences are unweighted [8]. However, when this approach is used on weighted sequences, its time complexity becomes $\mathcal{O}(\sigma m \log(\sigma m))$ which is worse than the time complexity of the implementation proposed for Step 1. Step 2, computed in lines 12–19 (see MiCO pseudo-code), can be implemented in $O(\ell|\Delta||\mathcal{D}|)$, where $|\Delta|$ is the number of length-1 common subsequences between the two input sequences, $|\mathcal{D}|$ is the size of the sets resulting from the concatenations, and ℓ is the length of the computed WLCS. Clearly the overall time complexity of the algorithm is determined by Step 2. The space complexity is bounded by $O\ (|\Delta||\mathcal{D}|)$. It should be remarked that $|\Delta|$ will decrease as the algorithm runs; on the other hand, $|\mathcal{D}|$ will tend to increase during the first stages of the running time and decrease during the last ones.

4.2 Example

For the input weighted sequences X and Y shown in Fig. 1 and two constants $\alpha_1 = 0.2$ and $\alpha_2 = 0.2$, Fig. 6**(c)** shows how to calculate the set of length-1 common subsequences, Δ, using the procedure described in Step 1 (MiCO pseudo-code, lines 1–8). Then, Fig. 7 shows all the iterations during Step 2. The set $\mathcal{D}_1$ is calculated by calling !Dominate($\delta, \mathcal{D}_1$) (MiCO pseudo-code, line 9–10). In the first iteration of the for loops (MiCO pseudo-code, lines 12–13), the set $\mathcal{F}_2$ is built by calling

	Δ			$\mathcal{D}_1$
iteration 1	δ_1 `g` [1/1] (1, 1)	δ_5 `a` [2/5] (1, 0.5)	δ_9 `t` [4/4] (0.5, 1)	δ_1 `g`
	δ_2 `a` [2/2] (0.6, 0.5)	δ_6 `a` [3/2] (0.6, 1)	δ_{10} `g` [5/1] (1, 1)	δ_{10} `g`
	δ_3 `c` [2/2] (0.4, 0.5)	δ_7 `a` [3/5] (1, 1)	δ_{11} `a` [6/2] (0.6, 0.8)	
	δ_4 `c` [2/4] (0.5, 0.5)	δ_8 `t` [4/3] (1, 1)	δ_{12} `a` [6/5] (1, 0.8)	

	$\mathcal{F}_2 = \mathsf{Concatenate}(\Delta, \mathcal{D}_1)$		$\mathcal{D}_2$	
iteration 2	$\delta_1\delta_2$ `ga` [1,2/1,2] (0.6, 0,5)	$\delta_1\delta_8$ `gt` [1,4/1,3] (1, 1)	$\delta_1\delta_2$ `ga`	$\delta_1\delta_{11}$ `ga`
	$\delta_1\delta_3$ `gc` [1,2/1,2] (0.4, 0.5)	$\delta_1\delta_9$ `gt` [1,4/1,4] (0.5, 1)	$\delta_1\delta_3$ `gc`	$\delta_{10}\delta_{11}$ `ga`
	$\delta_1\delta_4$ `gc` [1,2/1,4] (0.5, 0.5)	$\delta_1\delta_{11}$ `ga` [1,6/1,2] (0.6, 0.8)	$\delta_1\delta_5$ `ga`	
	$\delta_1\delta_5$ `ga` [1,2/1,5] (1, 0.5)	$\delta_1\delta_{12}$ `ga` [1,6/1,5] (1, 0.8)	$\delta_1\delta_6$ `ga`	
	$\delta_1\delta_6$ `ga` [1,3/1,2] (0.6, 1)	$\delta_{10}\delta_{11}$ `ga` [5,6/1,2] (0.6, 0.8)	$\delta_1\delta_7$ `ga`	
	$\delta_1\delta_7$ `ga` [1,3/1,5] (1, 1)	$\delta_{10}\delta_{12}$ `ga` [5,6/1,5] (1, 0.8)	$\delta_1\delta_8$ `gt`	

	$\mathcal{F}_3 = \mathsf{Concatenate}(\Delta, \mathcal{D}_2)$		$\mathcal{D}_3$
iteration 3	$\delta_1\delta_2\delta_7$ `gaa` [1,2,3/1,2,5] (0.6, 0.5)	$\delta_1\delta_4\delta_7$ `gca` [1,2,3/1,4,5] (0.5, 0.5)	$\delta_1\delta_2\delta_7$ `gaa`
	$\delta_1\delta_2\delta_8$ `gat` [1,2,4/1,2,3] (0.6, 0.5)	$\delta_1\delta_4\delta_{12}$ `gca` [1,2,6/1,4,5] (0.5, 0.4)	$\delta_1\delta_2\delta_8$ `gat`
	$\delta_1\delta_2\delta_{12}$ `gaa` [1,2,6/1,2,5] (0.6, 0.4)	$\delta_1\delta_6\delta_8$ `gat` [1,3,4/1,2,3] (0.6, 1)	$\delta_1\delta_3\delta_8$ `gct`
	$\delta_1\delta_3\delta_7$ `gca` [1,2,3/1,2,5] (0.4, 0.5)	$\delta_1\delta_6\delta_{12}$ `gaa` [1,3,6/1,2,5] (0.6, 0.8)	$\delta_1\delta_6\delta_8$ `gat`
	$\delta_1\delta_3\delta_8$ `gct` [1,2,4/1,2,3] (0.4, 0.5)	$\delta_1\delta_8\delta_{12}$ `gta` [1,4,6/1,3,5] (1, 0.8)	$\delta_1\delta_8\delta_{12}$ `gta`
	$\delta_1\delta_3\delta_{12}$ `gca` [1,2,6/1,2,5] (0.4, 0.4)		

	$\mathcal{F}_4 = \mathsf{Concatenate}(\Delta, \mathcal{D}_3)$	$\mathcal{D}_4$
iteration 4	$\delta_1\delta_2\delta_8\delta_{12}$ `gata` [1,2,4,6/1,2,3,5] (0.6, 0.4)	$\delta_1\delta_2\delta_8\delta_{12}$ `gata`
	$\delta_1\delta_3\delta_8\delta_{12}$ `gcta` [1,2,4,6/1,2,3,5] (0.4, 0.4)	$\delta_1\delta_3\delta_8\delta_{12}$ `gcta`
	$\delta_1\delta_6\delta_8\delta_{12}$ `gata` [1,3,4,6/1,2,3,5] (0.6, 0.8)	$\delta_1\delta_6\delta_8\delta_{12}$ `gata`

iteration *5*
$\mathcal{F}_5 = \mathsf{Concatenate}(\Delta, \mathcal{D}_4) = \emptyset$

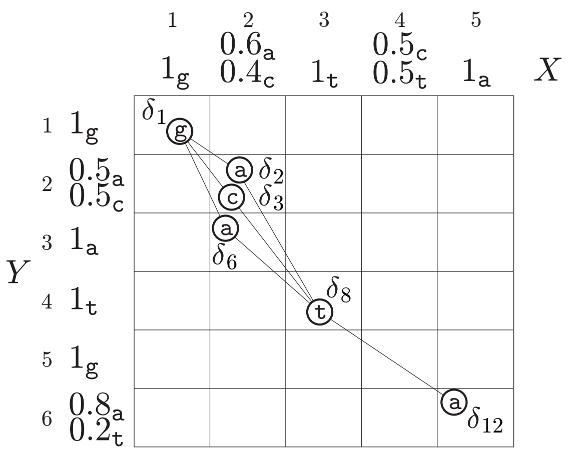
Figure 7. Computation trace of MiCO (Step 2). Each line represents a common subsequence p. The values [/] are the indices where p occurs in Y/X; (,) are $\pi^X(p)$ and $\pi^Y(p)$

Concatenate(d,δ) (MiCO pseudo-code, line 14), and the set $\mathcal{D}_2$ is established by calling !Dominate($d\delta, \mathcal{D}_1$) (MiCO pseudo-code, line 14). The algorithm performs the same procedure for the second and third iterations in which sets $\mathcal{D}_3$ and $\mathcal{D}_4$ are respectively computed. In the fifth iteration, the set $\mathcal{D}_5$ is empty after calling Concatenate(d,δ); thus, the set $\mathcal{D}_4$, consisting of three length-4 common subsequences of X and Y, is returned as the answer. Fig. 8 depicts, in a more schematic form, these longest common subsequences found by the proposed algorithm.

Notice that only the current set $\mathcal{D}$ needs to be stored; we do not need the additional data structure $\mathcal{F}_i$, it is only used for clarity purposes. Furthermore, there are some additional optimizations of the algorithm that were not illustrated in the example. For instance, the size of Δ can be decreased in each iteration and, therefore, we do not need to concatenate the elements $d \in \mathcal{D}_i$, at each iteration i, with all the elements δ in the initial set.

5 Conclusions

In this work we presented a new algorithm for the LCS problem applied to weighted sequences. This algorithm works under a multiobjective perspective, which allows tackling the LCS problem as an optimization problem. The time complexity of the algorithm depends on the total number of matches between the two input weighted



$$\pi^X(\delta_1\delta_2\delta_8\delta_{12}) = \pi^X(\texttt{gata}) = 0.6$$
$$\pi^Y(\delta_1\delta_2\delta_8\delta_{12}) = \pi^Y(\texttt{gata}) = 0.4$$

$$\pi^X(\delta_1\delta_3\delta_8\delta_{12}) = \pi^X(\texttt{gcta}) = 0.4$$
$$\pi^Y(\delta_1\delta_3\delta_8\delta_{12}) = \pi^Y(\texttt{gcta}) = 0.4$$

$$\pi^X(\delta_1\delta_6\delta_8\delta_{12}) = \pi^X(\texttt{gata}) = 0.6$$
$$\pi^Y(\delta_1\delta_6\delta_8\delta_{12}) = \pi^Y(\texttt{gata}) = 0.8$$

Figure 8. WLCS reported by MiCO

sequences, the number of dominant common subsequences detected during the computation and the length of the weighted longest common subsequence.

The main contributions of the paper can be summarized as follows: i) an algorithm that returns all the dominant LCSs between two weighted sequences, and ii) a framework to tackle the LCS problem as a multiobjective optimization problem.

References

1. A. Amir, Z. Gotthilf, and B. Shalom: *Weighted LCS.* Journal of Discrete Algorithms, 8 2010, pp. 273–281.
2. A. Amir, C. Iliopoulos, O. Kapah, and E. Porat: *Approximate matching in weighted sequences.* Lecture Notes on Computer Science, 4009 2006, pp. 365–376.
3. S. Bhowmick, M. Shafiullah, H. Rai, and D. Bastola: *A Parallel Non-Alignment Based Approach to Efficient Sequence Comparison using Longest Common Subsequences.* Journal of Physics: Conference Series, 256 2010, p. 012012.
4. P. Bonizzoni, G. Vedova, R. Dondi, G. Fertin, R. Rizzi, and S. Vialette: *Exemplar Longest Common Subsequence.* IEEE/ACM Transactions on Computational Biology and Bioinformatics, 4 2007, pp. 535–543.
5. M. Cygan, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen: *Polynomial-Time Approximation Algorithms for Weighted LCS Problem*, in Combinatorial Pattern Matching, R. Giancarlo and G. Manzini, eds., vol. 6661 of Lecture Notes in Computer Science, Springer Berlin - Heidelberg, 2011, pp. 455–466.
6. D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
7. W. Hsu and M. Du: *New algorithms for the LCS problem.* J. Computer and Systems Sciences, 19 1984, pp. 133–152.
8. J. Hunt and T. Szymanski: *A fast algorithm for computing longest common subsequences.* Communications of the ACM, 20(5) 1977, pp. 350–353.
9. C. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis: *Efficient algorithms for handling molecular weighted sequences.* Exploring New Frontiers of Theoretical Informatics, 155 2004, pp. 265–278.
10. C. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis: *The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications.* Fundamenta Informaticae, 71(2) 2006, pp. 259–277.
11. E. Lander: *Initial sequencing and analysis of the human genome.* Nature, 409 2001, pp. 860–921.
12. D. Maier: *The complexity of some problems on subsequences and supersequences.* Journal of the ACM (JACM), 25(2) 1978, pp. 322–336.

13. C. Makris and E. Theodoridis: *String Data Structures for Computational Molecular Biology.* Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications, 1 2011, pp. 3–27.
14. K. Perdikuri and A. Tsakalidis: *Motif extraction from biological sequences: Trends and contributions to other scientific fields.* Information Technology and Applications, 2005. ICITA 2005. Third International Conference on, 1 2005, pp. 453–458.
15. O. Saetrom, O. Snove Jr, and P. Saetrom: *Weighted sequence motifs as an improved seeding step in microRNA target prediction algorithms.* RNA, 11(7) 2005, p. 995.

New and Efficient Approaches to the Quasiperiodic Characterisation of a String

Tomáš Flouri[1], Costas S. Iliopoulos[2,3], Tomasz Kociumaka[4], Solon P. Pissis[1,5⋆], Simon J. Puglisi[2⋆⋆], William F. Smyth[2,3,6], and Wojciech Tyczyński[4]

[1] Heidelberg Institute for Theoretical Studies, 35 Schloss-Wolfsbrunnenweg, Heidelberg D-69118, Germany
{tomas.flouri,solon.pissis}@h-its.org

[2] King's College London, Dept. of Informatics, The Strand, London WC2R 2LS, UK
{c.iliopoulos,simon.puglisi}@kcl.ac.uk

[3] University of Western Australia, School of Mathematics and Statistics, 35 Stirling Highway, Crawley, Perth WA 6009, Australia

[4] University of Warsaw, Faculty of Mathematics, Informatics and Mechanics, Warsaw, Poland
{kociumaka,w.tyczynski}@mimuw.edu.pl

[5] University of Florida, Florida Museum of Natural History, 1659 Museum Road, Gainesville, FL 32611, USA

[6] McMaster University, Dept. of Computing and Software, 1280 Main St. West, Hamilton, Ontario L8S 4K1, Canada
smyth@mcmaster.ca

Abstract. A factor u of a string y is a *cover* of y if every letter of y lies within some occurrence of u in y; thus every cover u is also a *border* – both prefix and suffix – of y. A string y covered by u thus generalises the idea of a *repetition*; that is, a string composed of exact concatenations of u. Even though a string is coverable somewhat more frequently than it is a repetition, still a string that can be covered by a single u is rare. As a result, seeking to find a more generally applicable and descriptive notion of cover, many articles were written on the computation of a *minimum k-cover* of y; that is, the minimum cardinality set of strings of length k that collectively cover y. Unfortunately, this computation turns out to be NP-hard. Therefore, in this article, we propose new, simple, easily-computed, and widely applicable notions of string covering that provide an intuitive and useful characterisation of a string and its prefixes: the *enhanced cover* and the *enhanced cover array.*

Keywords: periodicity, quasiperiodicity, covers

1 Introduction

The notion of periodicity in strings and its many variants have been well-studied in many fields like combinatorics on strings, pattern matching, data compression, automata theory, formal language theory, and molecular biology (cf. [18]). Periodicity is of paramount importance in many applications – for example, periodic factors in DNA are of interest to genomics researchers [16] – as well as in theoretical studies in combinatorics on words. Not long ago the term *regularity* [10] was coined to cover such variants, and a recent survey [22] provides coverage of the exact regularities so far identified and the sequential algorithms proposed to compute them. In this article, in an effort to capture a more natural characterisation of a string in terms of its factors, we introduce a new form of regularity that is both descriptive and easy to compute.

⋆ Supported by the NSF-funded iPlant Collaborative (NSF grant #DBI-0735191).

⋆⋆ Supported by a Newton Fellowship.

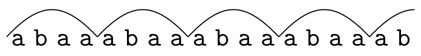


Figure 1. Periodicity in string `abaaabaaabaaabaaab`

A string y is a *repetition* if $y = u^k$ for some non-empty string u of length m and some integer $k \geq 2$; in this case, y has *period* m (see Fig. 1). But the notion of periodicity is too restrictive to provide a description of a string such as $x =$ `abaababaaba`, which is covered by copies of `aba`, yet not exactly periodic. To fill this gap, the idea of *quasiperiodicity* was introduced [1,2]. In a periodic string, the occurrences of the single periods do not overlap. In contrast, the quasiperiods of a quasiperiodic string may overlap. Quasiperiodicity thus enables the detection of repetitive structures that would be ignored by the classical characterisation of periods (see Fig. 2 in contrast with Fig. 3). The most well-known formalisation of quasiperiodicity is the cover of string. A factor u of length m of a string y of length n is said to be a *cover* of y if $m < n$, and every letter of y lies within some occurrence of u. Note that a cover of y must also be a *border* – both suffix and prefix – of y. Thus in the above example `aba` is a cover of $x =$ `abaababaaba`.

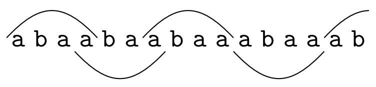


Figure 2. Quasiperiodicity in string `abaabaabaaabaaab`

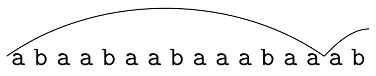


Figure 3. Periodicity in string `abaabaabaaabaaab`

In [3], Apostolico, Farach, and Iliopoulos described a recursive linear-time algorithm to compute the shortest cover of a string y of length n, if it has a cover; otherwise to report that no cover exists. Breslauer [4] introduced the *minimal cover array* C – an array of size n of integers such that $\mathsf{C}[i]$, for all $0 \leq i < n$, gives the shortest cover of $y[0\,..\,i]$, or zero if no cover exists. Moreover, he described an on-line linear-time algorithm to compute C. In [19,20], Moore and Smyth described a linear-time algorithm to compute *all* the covers of y. An $\mathcal{O}(\log(\log n))$-time parallel algorithm was given later by Iliopoulos and Park in [13]. Finally, Li and Smyth [17] introduced the *maximal cover array* C^{M} – an array of size n of integers such that $\mathsf{C}^{\mathrm{M}}[i]$ gives the longest cover of $y[0\,..\,i]$, or zero if no cover exists – and showed that, analogous to the border array [21], C^{M} actually specifies *all* the covers of every prefix of y. They then described a linear-time algorithm to compute C^{M}.

Still it remains unlikely that an arbitrary string, even on alphabet $\{a, b\}$, has a cover; for example, changing the above example x to $x' =$ `abaaababaaba` yields a string that not only has no cover, but whose every prefix also has no cover. Accordingly, in an effort to extend the descriptive power of quasiperiodicity, the notion of k-cover was introduced [14]: if for a given string y and a given positive integer k there exists a set $\mathcal{C}_k$ of factors of y, each of length k, such that every letter of y lies within some occurrence of some element of $\mathcal{C}_k$, then $\mathcal{C}_k$ is said to be a *k-cover* of y; a *minimal k-cover* if no smaller set has this property. Originally it was thought, incorrectly, that a minimal k-cover of a string y could be computed in time polynomial in n [14], but then later the problem was shown to be NP-complete for every $k \geq 2$ [8], even though an approximate solution could be computed in polynomial time [11].

Our contribution. We have seen that while the notion of cover captures very well the repetitive nature of extremely repetitive strings, nevertheless most strings, and particularly those encountered in practice, will have no cover, and so this measure of repetitiveness breaks down. More strings will have a useful k-cover, but this feature is hard to compute. Therefore we introduce a new and more natural and applicable form of quasiperiodicity.

- **Enhanced cover.** A border u of a string y is an *enhanced cover* of y, if the number of letters of y which lie within some occurrence of u in y is a maximum over all borders of y (see Fig. 5).

  This gives rise to the following data structure.

- **Enhanced cover array.** An array of size n of integers is the *enhanced cover array* of y, if it stores the length of the enhanced cover for every prefix of y.

In this article, we present efficient methods for computing all enhanced covers and the enhanced cover array of a string, and, in particular, the *minimal enhanced cover* and the *minimal enhanced cover array*. These methods are based on the maintenance of a new, simple but powerful data structure, which stores the number of positions covered by some prefixes of the string. This data structure allows us to compute the minimal enhanced cover of a string of length n in time $\mathcal{O}(n)$ and the minimal enhanced cover array in time $\mathcal{O}(n \log n)$.

The rest of this article is structured as follows. In Section 2, we present basic definitions and notation used throughout this article, and we also formally define the problems solved. In Section 3, we prove several combinatorial properties of the borders of y, which may be of independent interest. In Section 4, we show how to compute a few auxiliary arrays, which will be used for designing the proposed algorithms. In Section 5, we present an algorithm for computing the minimal enhanced cover of y. In Section 6, we present an algorithm for computing the minimal enhanced cover array of y. In Section 7, we present some experimental results. Finally, we briefly conclude with some future proposals in Section 8.

2 Definitions and notation

An *alphabet* Σ is a finite non-empty set whose elements are called *letters*. A *string* on an alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-letter sequence is called the *empty string*, and is denoted by ε. The *length* of a string x is defined as the length of the sequence associated with the string x, and is denoted by $|x|$. We denote by $x[i]$, for all $0 \leq i < |x|$, the letter at index i of x. Each index i, for all $0 \leq i < |x|$, is a position in x when $x \neq \varepsilon$. It follows that the ith letter of x is the letter at position $i-1$ in x, and that $x = x[0\,.\,.\,|x|-1]$.

The *concatenation* of two strings x and y is the string of the letters of x followed by the letters of y. It is denoted by xy. For every string x and every natural number n, we define the nth *power* of the string x, denoted by x^n, by $x^0 = \varepsilon$ and $x^k = x^{k-1}x$, for all $1 \leq k \leq n$. A string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$. A factor x of a string y is *proper* if $x \neq y$. Let the strings x, y, u, and v be such that $y = uxv$. If $u = \varepsilon$, then x is a *prefix* of y. If $v = \varepsilon$, then x is a *suffix* of y.

Let x be a non-empty string. An integer p, such that $0 < p \leq |x|$, is called a *period* of x if $x[i] = x[i+p]$, for all $0 \leq i < |x| - p$. Note that the length of a non-empty string is a period of this string, so that every non-empty string has at least one period. We define thus without any ambiguity *the period* of a non-empty string x as the smallest of its periods. It is denoted by $\mathsf{per}(x)$. A *border* of a non-empty string x is a proper factor of x (including the empty string) that is both a prefix and a suffix of x. We define *the border* of a non-empty string x as the longest border of x. By $\mathsf{border}(x)$, we denote the length of the border of x. The notions of period and of border are dual. It is a known fact (cf. [9]) that, for any non-empty string x, it holds $\mathsf{per}(x) + \mathsf{border}(x) = |x|$. The *border array* B of a non-empty string y of length n is the array of size n of integers for which $\mathsf{B}[i]$, for all $0 \leq i < n$, stores the length of the border of the prefix $y[0 \mathinner{..} i]$ of y – zero if none.

A non-empty string u of length m is a *cover* of a non-empty string y if both $m < n$, and there exists a set of positions $P \subseteq \{0, \ldots, n-m\}$ that satisfies both $y[i \mathinner{..} i+m-1] = u$, for all $i \in P$, and $\bigcup_{i \in P}\{i, \ldots, i+m-1\} = \{0, \ldots, n-1\}$. In other words, u is a cover of y, if every letter of y lies within some occurrence of u in y, and $u \neq y$.

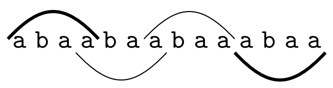


Figure 4. Cover of string abaabaabaaabaa

A string u is the *minimal cover* of string y if u is the shortest cover of y. The *minimal cover array* C of a non-empty string y of length n is the array of size n of integers for which $\mathsf{C}[i]$, for all $0 \leq i < n$, stores the length of the minimal cover of the prefix $y[0 \mathinner{..} i]$ of y – zero if none.

Definition 1. *A border u of a string y is an* enhanced cover *of y if the number of letters of y which lie within occurrences of u in y is a maximum over all borders of y.*

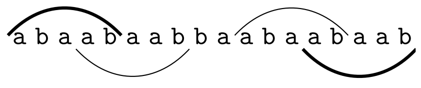


Figure 5. Enhanced cover of string abaabaabbaabaabaab

Definition 2. *We define as* minimal enhanced cover *the shortest enhanced cover of y.*

Definition 3. *The* minimal enhanced cover array MEC *of a non-empty string* y *of length* n *is the array of size* n *of integers for which* $\mathit{MEC}[i]$*, for all* $0 \leq i < n$*, stores the length of the minimal enhanced cover of the prefix* $y[0\,..\,i]$ *of* y *– zero if none.*

Example 4. Consider the string y = abaaababaabaaaababaa. The following table illustrates the border array B of y, the minimal cover array C of y, and the minimal enhanced cover array MEC of y. In this example, array C consists of only zeros, as a minimal cover does not exist for any of the prefixes of y. In contrast, array MEC is a more powerful data structure than array C, as apart from the minimal cover of every prefix, it also contains the minimal enhanced cover of every prefix in the case when a minimal cover does not exist. For instance, border abaa is the minimal enhanced cover of y, as 15 letters of y lie within some occurrence of abaa in y.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$y[i]$	a	b	a	a	a	b	a	b	a	a	b	a	a	a	a	b	a	b	a	a
B$[i]$	0	0	1	1	1	2	3	2	3	4	2	3	4	5	1	2	3	2	3	4
C$[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MEC$[i]$	0	0	1	1	1	2	3	2	3	4	2	3	4	1	1	2	3	2	3	4

We consider the following problems for a non-empty string y.

Problem 5. Compute the minimal enhanced cover of y.

Problem 6. Compute the minimal enhanced cover array MEC of y.

3 Combinatorial properties of non-periodic borders

Definition 7. *A string* w *is called* periodic *if it is non-empty and* $2\mathit{per}(w) \leq |w|$*. Otherwise it is called* non-periodic*.*

The efficiency of the algorithms in this article is a consequence of considering only non-periodic factors. The following fact explains why this restriction is valid.

Fact 1 *A periodic string always has a (proper) cover. As a consequence, the minimal enhanced cover is never periodic.*

Proof. Let w be a periodic string with the border u. As a factor of w, u has at least two occurrences: as a prefix and as a suffix. Since $2|u| \geq |w|$, these occurrences cover v. In particular, if w is an enhanced cover, then u is a shorter enhanced cover. Hence w cannot be the minimal enhanced cover. □

We prove two combinatorial properties of non-periodic borders, which are then used to prove the time complexities of our algorithms. The first one is simple fact, but, with corollaries, is the main reason behind the restriction to non-periodic borders.

Fact 2 *Let* u *and* v *be borders of* y*, such that* $|v| > |u|$*, and* v *is non-periodic. Then* $|v| > 2|u|$*.*

Proof. Clearly, u is a border of v, hence $|v| - |u|$ is a period of v. However v is non-periodic, so $2(|v| - |u|) > |v|$, i.e. $|v| > 2|u|$. □

Corollary 8. *The length of the* k*th shortest non-periodic border of* y *is at least* $2^k - 1$*. In particular, the total number of the non-periodic borders of* y *is at most* $\log n$*.*

Proof. Let b_k be the length of the kth shortest non-periodic border of y. From Fact 2, $b_{k+1} \geq 2b_k + 1$. Moreover, $b_1 \geq 1$. The first part of Corollary follows by induction. For the second part, it is enough to see that if there were $k > \log n$ non-periodic borders, then $b_k \geq 2^k - 1 > n - 1$, which is clearly a contradiction. □

Note that by Corollary 8, the total number of occurrences of the non-periodic prefixes of y is $\mathcal{O}(n \log n)$. This is because if a prefix v ends with an occurrence of u, then $u = v$ or u is a border of v. This induces a one-to-one correspondence between such occurrences – with exception of $\mathcal{O}(n)$ ones starting 0 – and the non-periodic borders of prefixes of y. It turns out that, if we consider just the occurrences of non-periodic borders of y, the number drops to $\mathcal{O}(n)$.

Before we proceed, let us introduce a notion, which we use across the proofs below. Let X be an array of size n of integers for which $\mathsf{X}[i]$, for all $0 \leq i < n$, stores the number of those non-periodic borders of the prefix $y[0\,..\,i]$ of y, which are simultaneously borders of y.

Lemma 9. *The total number of occurrences of the non-periodic borders of y is linear. More precisely*

$$\sum_{i=0}^{n-1} \mathsf{X}[i] \leq 2n.$$

Let us start with an auxiliary claim.

Claim. Let $0 \leq i, j < n$ be integers. If $\mathsf{X}[i] > k$ then $i \geq 3 \cdot 2^k - 2$. Moreover, if $i < j$, $\mathsf{X}[i] > k$ and $\mathsf{X}[j] > k$ then $j - i \geq 2^k$.

Proof. Clearly, if u and v are non-periodic borders of y, then the shorter one is a non-periodic border of the longer one. Thus $\mathsf{X}[i] > k$ if and only if the $k+1$th shortest non-periodic border of y is a border of $y[0\,..\,i]$. Let us denote this border by b. By Corollary 8, $|b| \geq 2^{k+1} - 1$. Any two occurrences of b in y must have their starting positions distant by at least $\frac{|b|+1}{2} \geq 2^k$. A pair of closer occurrences would induce a period of b no larger than $\frac{|b|}{2}$, which may not exist, since b is non-periodic.

For the first part of the Claim, consider the occurrence starting at 0 and the occurrence ending at i, i.e. starting at $i - |b| + 1$. These are different occurrences, so $i \geq |b| - 1 + 2^k \geq 3 \cdot 2^k - 2$. In the second part, there are occurrences of b ending at i and at j, which implies that $j - i \geq 2^k$. □

Proof (of Lemma 9). In the following proof, we use the Iverson bracket $[P]$. For a logical statement P, the Iverson bracket $[P]$ is by definition equal to 1 if P is satisfied, and 0 otherwise.

Clearly for a non-negative integer m we have $m = \sum_{k=0}^{\infty} [k < m]$. Hence

$$\sum_{i=0}^{n-1} \mathsf{X}[i] = \sum_{i=0}^{n-1} \sum_{k=0}^{\infty} [k < \mathsf{X}[i]] = \sum_{k=0}^{\infty} \sum_{i=0}^{n-1} [\mathsf{X}[i] > k].$$

Let us bound the single term of the outer sum. This sum counts positions i such that $\mathsf{X}[i] > k$. By Claim, the first of them is at least $3 \cdot 2^k - 2$, and the distance between any two such positions is at least 2^k. This means that if we write them all in increasing order, then the mth one is at least $(m+2) \cdot 2^k - 2$. In particular, for $m > \frac{n}{2^k}$, the

mth position would be at least $n + 2 \cdot 2^k - 2 \geq n$, which is clearly impossible. Hence, there cannot be more than $\frac{n}{2^k}$ such positions, i.e.

$$\sum_{i=0}^{n-1} [\mathsf{X}[i] > k] \leq \frac{n}{2^k}.$$

Therefore

$$\sum_{i=0}^{n-1} \mathsf{X}[i] = \sum_{k=0}^{\infty} \sum_{i=0}^{n-1} [\mathsf{X}[i] > k] \leq \sum_{k=0}^{\infty} \frac{n}{2^k} = 2n.$$

□

4 Auxiliary arrays

Our algorithms make use of a few auxiliary arrays. In this section, we show how to compute these arrays efficiently. Below we assume the availability of the border array B, which is computable in linear time (see, e.g. [15,9,21]), and an array CB of size n such that, for all $0 \leq i < n$, $\mathsf{CB}[i]$ is 1 if $i+1$ is a border of y, and 0 otherwise. CB is trivially computed from B as the borders of y are exactly the longest border of y and its borders.

Definition 10. *Given a string y of length n, the* pruned border array A *is an array of size n of integers for which $\mathsf{A}[i]$, for all $0 \leq i < n$, stores the length of the longest non-periodic border of $y[0\,..\,i]$ – zero if none.*

The pruned border array A of a string y can be computed by Algorithm 1 in linear time. Algorithm 1 loops through the prefixes of y and in each step considers two cases. If the longest border u of a prefix v is non-periodic, then u is the border of v we are looking for. Otherwise, the longest non-periodic border of v is the longest non-periodic border of u.

Algorithm 1: PrunedBorderArray

```
Input  : The border array $\mathsf{B}$ of string $y[0\,..\,n-1]$
Output: The pruned border array $\mathsf{A}$
1 for $i \leftarrow 0$ to $n-1$ do
2     $b \leftarrow \mathsf{B}[i]$
3     if $b = 0$ or $2 \cdot B[b-1] < b$ then
4         $\mathsf{A}[i] \leftarrow b$
5     else
6         $\mathsf{A}[i] \leftarrow \mathsf{A}[b-1]$
```

Definition 11. *Given a string y of length n, let R be an array of size n of integers for which $\mathsf{R}[i]$, for all $0 \leq i < n$, stores the length of the longest non-periodic border of $y[0\,..\,i]$, which is also a border of y – zero if none.*

R can be computed by Algorithm 2 in linear time. For each prefix v of y we determine the longest non-periodic border u of v and consider the following cases. If u is a border of y (in particular if u is empty), then clearly $\mathsf{R}[i] = \mathsf{A}[i]$. Otherwise, the border we seek is a shorter non-periodic border of v, so the result for v is same as for u.

Algorithm 2: ARRAY R

```
Input  : The pruned border array A and array CB of string $y[0\,..\,n-1]$
Output: Array R

for $i \leftarrow 0$ to $n-1$ do
    $b \leftarrow \mathsf{A}[i]$
    if $b = 0$ or $CB[b-1] = 1$ then
        $\mathsf{R}[i] \leftarrow b$
    else
        $\mathsf{R}[i] \leftarrow \mathsf{R}[b-1]$
```

Definition 12. *Given a string y of length n, PCP is an array of size n of integers for which PCP$[i]$, for all $0 \le i < n$, stores the number of letters of y which lie within an occurrence of the non-periodic prefix of length $i+1$ having at least two occurrences in y – zero if the prefix is periodic or does not have two occurrences.*

The PCP array of string y can be computed by Algorithm 3. It takes as input the pruned border array A of y. We also maintain an array LO of size n of integers, for which LO$[i]$, for all $0 \le i < n$, stores the ending position of the last occurrence of the non-periodic prefix of length $i+1$ in y, not taking into account the occurrence as a prefix. Fields corresponding to periodic prefixes are never read or written.

Algorithm 3: POSITIONSCOVEREDBYPREFIXESARRAY

```
Input  : The pruned border array $\mathsf{A}[0\,..\,n-1]$ of string $y[0\,..\,n-1]$
Output: The PCP array

1  PCP $\leftarrow$ FILLWITHZEROS
2  for $i \leftarrow 0$ to $n-1$ do
3      $b \leftarrow \mathsf{A}[i]$
4      while $b > 0$ do
5          if $PCP[b-1] = 0$ then
6              $\mathsf{PCP}[b-1] \leftarrow \min(2b, i+1)$
7          else
8              $\mathsf{PCP}[b-1] \leftarrow \mathsf{PCP}[b-1] + \min(b, i - \mathsf{LO}[b-1])$
9          $\mathsf{LO}[b-1] \leftarrow i$
10         $b \leftarrow \mathsf{A}[b-1]$
```

The algorithm consists of an outer for loop, going through the pruned border array A, and an inner while loop, iterating through the non-periodic borders of prefix $y[0\,..\,i]$. If the first occurrence of some border of length b of $y[0\,..\,i]$ is found (line 5), we take the minimum between $2b$, that is in case $y[0\,..\,b-1]$ does not overlap with $y[i-b+1\,..\,i]$, and $i+1$, that is in case they overlap (line 6). If another occurrence of the same border is found (line 7), we update PCP$[b-1]$ by adding the minimum between b, that is in case $y[i-b+1\,..\,i]$ does not overlap with the last occurrence of the border, and $i - \mathsf{LO}[b-1]$, that is in case they overlap (line 8). Hence we obtain the following result.

Theorem 13. *The PCP array of a string of length n can be computed by Algorithm 3 in time $\mathcal{O}(n \log n)$.*

Proof. The algorithm consists of an outer for loop, going through the pruned border array A, and an inner while loop, iterating through the non-periodic borders of prefix $y[0 \mathinner{..} i]$. By Corollary 8, the number of non-periodic borders of each prefix is bounded by $\log n$. Hence, in overall, the time required is $\mathcal{O}(n \log n)$. $\square$

We define one more array, similar to PCP but restricted to borders of the whole string only.

Definition 14. *Given a string y of length n, PCB is an array of size n of integers for which PCB$[i]$, for all $0 \leq i < n$, stores the number of letters of y which lie within an occurrence of the non-periodic prefix of length $i+1$, which is a border of y – zero if the prefix is periodic or is not a border of y.*

Example 15. Consider the string y = aabaabaabbaaabaabaa. The following table illustrates the border array B of y, and the auxiliary arrays A, CB, R, PCP, and PCB of y.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$y[i]$	a	a	b	a	a	b	a	a	b	b	a	a	a	b	a	a	b	a	a
B$[i]$	0	1	0	1	2	3	4	5	6	0	1	2	2	3	4	5	6	7	8
A$[i]$	0	1	0	1	1	3	4	5	3	0	1	1	1	3	4	5	3	4	5
CB$[i]$	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
R$[i]$	0	1	0	1	1	0	1	5	0	0	1	1	1	0	1	5	0	1	5
PCP$[i]$	13	0	15	14	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PCB$[i]$	13	0	0	0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0

5 Minimal enhanced cover

In this section, we show how to compute the minimal enhanced cover of string y. The minimal enhanced cover of y can be computed by Algorithm 4. It takes as input the array R of y. The algorithm consists of an outer for loop, going through the array R, and an inner while loop, iterating through the non-periodic borders of prefix $y[0 \mathinner{..} i]$ which are also borders of y. The key idea is the on-line maintenance of the PCB array (lines 5–8). Notice that array R considers only borders of the prefixes of y which are also borders of y (line 3). The number of positions covered by the border of length b is given by PCB$[b-1]$ (line 10).

By Lemma 9, the total number of occurrences of the considered borders is bounded by $2n$. Hence we obtain the following result.

Theorem 16. *The minimal enhanced cover of a string of length n can be computed in time $\mathcal{O}(n)$.*

6 Minimal enhanced cover array

In this section, we show how to compute the minimal enhanced cover array MEC of string y. Array MEC of y can be computed by Algorithm 5. It takes as input the pruned border array A of y. Similarly as in the case of Algorithm PositionsCoveredByPrefixesArray, it goes through the pruned border array A, and iterates through the pruned set of borders of prefix $y[0 \mathinner{..} i]$. Thus we are able to maintain the PCP array on-line, and use it to compute array MEC. For each prefix $y[0 \mathinner{..} i]$, in addition to the maintenance of the PCP array, we store the maximum value of

Algorithm 4: MinimalEnhancedCover

```
Input  : The array R of string $y[0\,.\,.\,n-1]$
Output: The length $\ell$ of the minimal enhanced cover
PCB$[0\,.\,.\,n-1] \leftarrow$ FillWithZeros; $\delta \leftarrow 0$; $\ell \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
    $b \leftarrow$ R$[i]$
    while $b > 0$ do
        if PCB$[b-1] = 0$ then
            PCB$[b-1] \leftarrow \min(2b, i+1)$
        else
            PCB$[b-1] \leftarrow$ PCB$[b-1] + \min(b, i -$ LO$[b-1])$
        LO$[b-1] \leftarrow i$
        if PCB$[b-1] > \delta$ then
            $\delta \leftarrow$ PCB$[b-1]$
            $\ell \leftarrow b$
        else if PCB$[b-1] = \delta$ and $b < \ell$ then
            $\ell \leftarrow b$
        $b \leftarrow$ R$[b-1]$
```

Algorithm 5: MinimalEnhancedCoverArray

```
Input  : The pruned border array A$[0\,.\,.\,n-1]$ of string $y[0\,.\,.\,n-1]$
Output: The minimal enhanced cover array MEC
PCP$[0\,.\,.\,n-1] \leftarrow$ FillWithZeros
for $i \leftarrow 0$ to $n-1$ do
    $b \leftarrow$ A$[i]$
    $\ell \leftarrow 0$
    $\delta \leftarrow 0$
    while $b > 0$ do
        if PCP$[i] = 0$ then
            PCP$[i] \leftarrow \min(2b, i+1)$
        else
            PCP$[i] \leftarrow$ PCP$[i] + \min(b, i -$ LO$[b-1])$
        if PCP$[b-1] \geq \delta$ then
            $\delta \leftarrow$ PCP$[b-1]$
            $\ell \leftarrow b$
        LO$[b-1] \leftarrow i$
        $b \leftarrow$ A$[b-1]$
    MEC$[i] \leftarrow \ell$
```

PCP$[b-1]$, for each border of length b of that prefix, in a variable δ, and the length b in a variable ℓ (lines 11–13).

By Theorem 13, the PCP array can be computed in time $\mathcal{O}(n \log n)$. Hence we obtain the following result.

Theorem 17. *The minimal enhanced cover array of a string of length n can be computed in time $\mathcal{O}(n \log n)$.*

7 Experimental results

We were able to verify the runtime of the proposed algorithms in experiments.

Fig. 6 illustrates the maximal ratio of the total number of occurrences of the non-periodic borders of y, computed by Algorithm MinimalEnhancedCover, to the length n of string, for all strings on the binary alphabet of lengths 1 to 31. These ratios are known to be smaller than 2 by Lemma 9. However, values close to this bound are not observed for small word length.

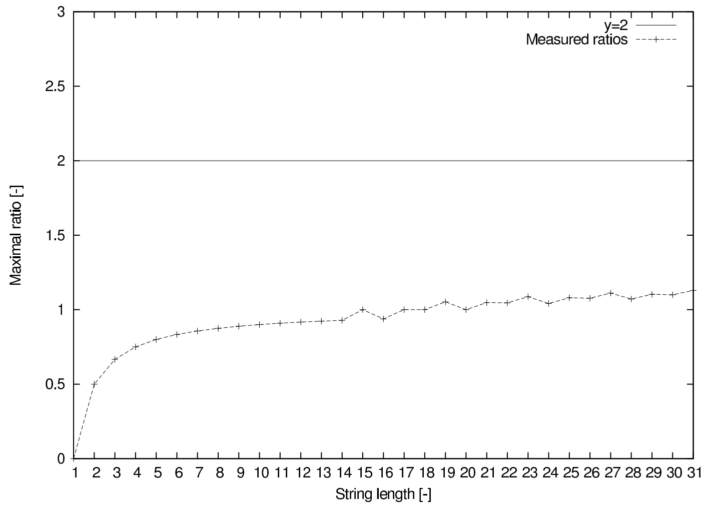


Figure 6. Maximal ratio of of the total number of occurrences of the non-periodic borders to the length n of string, for all strings on the binary alphabet

Fig. 7 and Fig. 8 illustrate the maximal ratio of the number of operations of Algorithm MinimalEnhancedCoverArray to the length n of string, for all strings on the binary alphabet of lengths 1 to 31, and the ratio of the number of operations to the length n of string, for Fibonacci strings f_3 to f_{45}, respectively. These ratios are known to be smaller than $\log n$ by Theorem 13.

The main observation from Fig. 7 is that, although the upper theoretical bound of these ratios is $\mathcal{O}(\log n)$, in practice, this is much less for strings on the binary alphabet. Fig. 8 strongly indicates that these ratios are probably constant for Fibonacci strings; something it would be interesting to show in the future.

In order to evaluate the performance of Algorithm MinimalEnhancedCover-Array with real datasets, we measured the ratio of the number of operations to the length of three DNA sequences: the single chromosome of *Escherichia coli* str. K-12 substr. MG1655; chromosome 1 of *Mus musculus* (laboratory mouse), Build 37.2; and chromosome 1 of *Homo sapiens* (human), Build 37.2. The measured ratios are 1.000006, 1.000000, and 1.001192, respectively, suggesting linear runtime of the proposed algorithms in practical terms.

The implementation of the proposed algorithms is available at a website (`http://www.exelixis-lab.org/solon/asc.html`) for further testing.

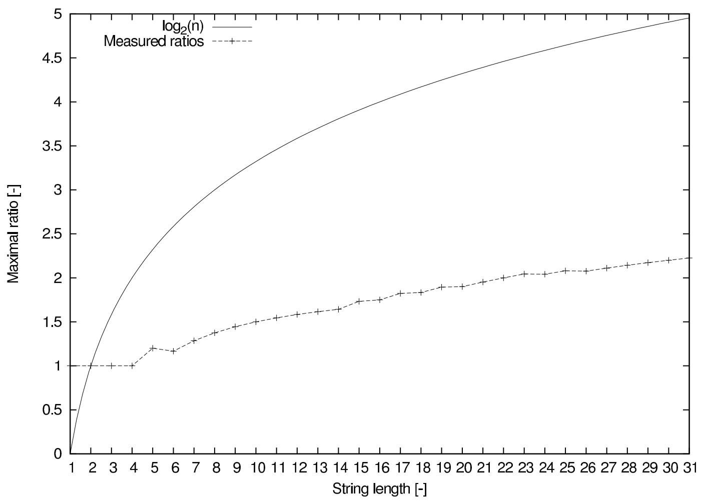


Figure 7. Maximal ratio of the number of operations of Algorithm 4 to the length n of string, for all strings on the binary alphabet

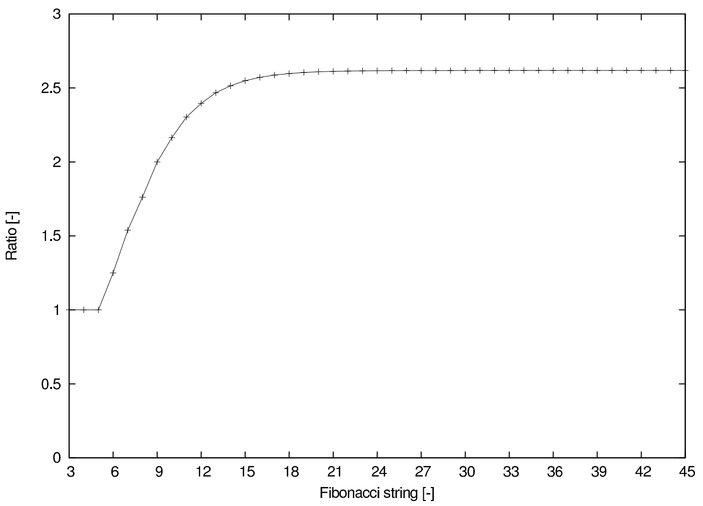


Figure 8. Ratio of the number of operations of Algorithm 4 to the length n of string, for Fibonacci strings

8 Concluding remarks

There are several directions for future work. Our immediate target is to investigate analogous data structures for other quasiperiodic notions such as the seed [12], the left seed [7], and the right seed [6] of a string. We will also consider the following problems for an array A of size n of integers.

Problem 18. Decide if A is the minimal enhanced cover array of some string.

Problem 19. When A is a valid minimal enhanced cover array, infer a string, whose minimal enhanced cover array is A.

For certain applications, the definition of the minimal enhanced cover might not be useful, since it primarily optimises the number of positions covered, while the length of the enhanced cover cannot be controlled. We can extend this notion by introducing the d-restricted enhanced cover of string y, which is the shortest border of y of length not exceeding d which covers the largest number of positions among borders no longer than d. The algorithm computing the minimal enhanced cover, with almost no extra computations, can compute the d-restricted enhanced covers for every positive integer $d < n$. Moreover, the algorithm computing the minimal enhanced cover array can be given an additional array D of size n of integers as input, and compute the $\mathsf{D}[i]$-restricted enhanced cover of $y[0\,..\,i]$, for all $0 \leq i < n$. This also requires no additional effort.

Another interesting open problem is to allow the enhanced cover to be any factor of the string – not only a border. A similar problem, though with different constraints on the occurrences of the cover, is considered in the context of grammar compression [5], but, to the best of our knowledge, no efficient solution for either problem has been published.

References

1. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings*, Tech. Rep. CSD-1R-I048, Purdue University, 1990.
2. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings.* Theoretical Computer Science, 119(2) 1993, pp. 247–265.
3. A. Apostolico, M. Farach, and C. S. Iliopoulos: *Optimal superprimitivity testing for strings.* Information Processing Letters, 39(1) 1991, pp. 17–20.
4. D. Breslauer: *An on-line string superprimitivity test.* Information Processing Letters, 44(6) 1992, pp. 345–347.
5. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat: *The smallest grammar problem.* IEEE Transactions on Information Theory, 51(7) 2005, pp. 2554–2576.
6. M. Christou, M. Crochemore, O. Guth, C. S. Iliopoulos, and S. P. Pissis: *On the right-seed array of a string*, in Proceedings of the seventeenth annual International Computing and Combinatorics Conference (COCOON 2011), B. Fu and D.-Z. Du, eds., vol. 6842 of Lecture Notes in Computer Science, USA, 2011, Springer, pp. 492–502.
7. M. Christou, M. Crochemore, C. S. Iliopoulos, M. Kubica, S. P. Pissis, J. Radoszewski, W. Rytter, B. Szreder, and T. Walen: *Efficient seed computation revisited*, in Proceedings of the twenty-second annual Symposium on Combinatorial Pattern Matching (CPM 2011), R. Giancarlo and G. Manzini, eds., vol. 6661 of Lecture Notes in Computer Science, Italy, 2011, Springer, pp. 350–363.
8. R. Cole, C. S. Iliopoulos, M. Mohamed, W. F. Smyth, and L. Yang: *The complexity of the minimum k-cover problem.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 641–653.
9. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, USA, 2007.
10. C. S. Iliopoulos, M. Mohamed, L. Mouchard, W. F. Smyth, K. G. Perdikuri, and A. K. Tsakalidis: *String regularities with don't cares.* Nordic Journal of Computing, 10(1) 2003, pp. 40–51.
11. C. S. Iliopoulos, M. Mohamed, and W. F. Smyth: *New complexity results for the k-covers problem.* Information Sciences, 181(12) 2011, pp. 2571–2575.
12. C. S. Iliopoulos, D. Moore, and K. Park: *Covering a string.* Algorithmica, 16(3) 1996, pp. 288–297.
13. C. S. Iliopoulos and K. Park: *A work-time optimal algorithm for computing all string covers.* Theoretical Computer Science, 164(1–2) 1996, pp. 299–310.
14. C. S. Iliopoulos and W. F. Smyth: *On-line algorithms for k-covering*, in Proceedings of the ninth Australasian Workshop on Combinatorial Algorithms (AWOCA 2008), Curtin University of Technology, 1998, pp. 64–73.
15. D. E. Knuth, J. H. M. Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.

16. R. Kolpakov, G. Bana, and G. Kucherov: *mreps: efficient and flexible detection of tandem repeats in DNA*. Nucleic Acid Research, 31(13) 2003, pp. 3672–3678.
17. Y. Li and W. F. Smyth: *Computing the cover array in linear time*. Algorithmica, 32(1) 2002, pp. 95–106.
18. M. Lothaire, ed., *Applied Combinatorics on Words*, Cambridge University Press, 2005.
19. D. Moore and W. F. Smyth: *An optimal algorithm to compute all the covers of a string*. Information Processing Letters, 50(5) 1994, pp. 239–246.
20. D. Moore and W. F. Smyth: *A correction to "An optimal algorithm to compute all the covers of a string"*. Information Processing Letters, 54(2) 1995, pp. 101–103.
21. W. F. Smyth: *Computing patterns in strings*, Addison-Wesley, 2003.
22. W. F. Smyth: *Computing regularities in strings: a survey*. European Journal of Combinatorics, 2012, (to appear).

The Number of Cubes in Sturmian Words

Marcin Piątkowski[1⋆] and Wojciech Rytter[2,1⋆⋆]

[1] Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University, Toruń, Poland
marcin.piatkowski@mat.umk.pl

[2] Department of Mathematics, Computer Science and Mechanics,
University of Warsaw, Warsaw, Poland
rytter@mimuw.edu.pl

Abstract. We design an efficient algorithm computing the number of distinct cubes in a standard Sturmian word given by its directive sequence (the special type of recurrences). The algorithm runs in linear time with respect to the size of the compressed representation (recurrences) describing the word, though the explicit size of the word can be exponential with respect to this representation. We give the explicit compact formula for the number of cubes in any standard word derived from the structural properties of runs (maximal repetitions). Fibonacci words are the most known subclass of standard Sturmian words. It is known that the ratio of the number of cubes to the size for Fibonacci words is asymptotically equal to $\frac{1}{\phi^3} \approx 0.2361$, where $\phi = \frac{\sqrt{5}+1}{2}$. We show a class of standard Sturmian words for which this ratio is much higher and equals $\frac{3\phi+2}{9\phi+4} \approx 0.36924841$. An extensive experimentation suggests that this value is optimal.

Keywords: standard Sturmian words, cubes, repetitions, algorithm

1 Introduction

Problems related to finding repetitions in strings are fundamental in combinatorics on words and have many practical applications (data compression, computational biology, pattern matching, etc.), see for instance [5], [8], [12] and [13]. The structure of repetitions is almost completely understood for the class of Fibonacci words, see [10], [11], [16], however it is not well understood for general words.

The most important type of repetitions are *runs* (maximal repetition), which form a compact representation of all repetitions in a word. Formally, a *run* in a word w is an interval $\alpha = [i..j]$ such that $w[i..j] = u^k v$ $(k \geq 2)$ is a nonempty periodic subword of w, where u is of the minimal length and v is a proper prefix (possibly empty) of u, that can not be extended (neither $w[i-1..j]$ nor $w[i..j+1]$ is a run with the period $|u|$).

In this paper we consider cubes: the nonempty words of the form $\alpha = x^3$. The length of x is called the *base* of the cube and denoted by $base(\alpha)$. A number i is a period of the word w if $w[j] = w[i+j]$ for all i with $i + j \leq |w|$. The minimal period (min-period, in short) of w will be denoted by $period(w)$.

⋆ The study is cofounded by the European Union from resources of the European Social Fund. Project PO KL “Information technologies: Research and their interdisciplinary applications”, Agreement UDA-POKL.04.01.01-00-051/10-00.

⋆⋆ Supported by the grant N206 566740 of the National Science Center

Example 1. Let $\alpha = (abab)^3$ be a cube. In this case we have:

$$base(\alpha) = 4, \quad period(\alpha) = 2.$$

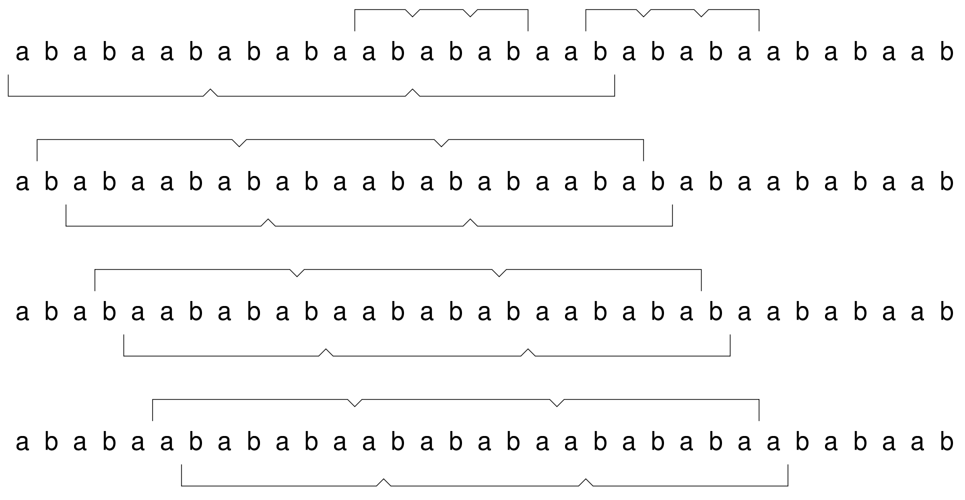

a b a b a a b a b a b a a b a b a b a a b a b a b a a b a b a a b

Figure 1. The structure of distinct cubes in the example binary word $w = ababaabababaabababaabababaababaab$

Observe that two different runs could correspond to the identical subwords, if we disregard their positions. Hence runs are also called the maximal *positioned* repetitions. In this paper we are interested in counting *distinct cubes*, hence we identify cubes with the same base, but perhaps multiple occurrences.

Example 2. Let w be as in Figure 1. There are 9 cubes:

$$\begin{array}{lll} ab \cdot ab \cdot ab, & ba \cdot ba \cdot ba, & ababaab \cdot ababaab \cdot ababaab, \\ babaaba \cdot babaaba \cdot babaaba, & & abaabab \cdot abaabab \cdot abaabab, \\ baababa \cdot baababa \cdot baababa, & & aababab \cdot aababab \cdot aababab, \\ abababa \cdot abababa \cdot abababa, & & bababaa \cdot bababaa \cdot bababaa. \end{array}$$

The *standard Sturmian* words are extensively studied in combinatorics on words. They are enough complicated to have many interesting properties and at the same time they are highly compressible. Due to their regularity, many problems are much easier for such strings compared with the general case. There are known exact formulas for the number of runs, cubic runs (i.e. runs in which the period repeats at least three times) and squares in standard words along with their *density ratio* (i.e. the asymptotic quotient of the maximal number of considered repetitions by the length of the word). See [2], [15] and [14] for details.

This paper is devoted to the investigation of the structure and the number of cubes in standard Sturmian words. Denote by $cubes(w)$ the number of cubes in a word w. We present exact formulas for $cubes(w)$ in any standard word w. We show also the algorithm, which computes the number of cubes in any standard word in linear time with respect to the size of its compressed representation – the directive sequence – hence in time logarithmic with respect to the length of the word. We show also a class of standard words reach in cubes and prove that for this class of strings the density ratio of distinct cubes equals $\frac{3\phi+2}{9\phi+4} \approx 0.36924841$, where $\phi = \frac{\sqrt{5}+1}{2}$. An extensive computer experimentation suggests that this value is optimal.

Some useful applets related to problems considered in this paper can be found on the web site: `http://www.mat.umk.pl/~martinp/stringology/applets/`

2 Standard Sturmian words

Standard Sturmian words (standard words in short) are one of the most investigated class of strings in combinatorics on words, see for instance [1], [4], [6], [12], [17], [18], [19] and references therein. They have very compact representations in terms of sequences of integers, which has many algorithmic consequences.

The *directive sequence* is the integer sequence: $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$, where $\gamma_0 \geq 0$ and $\gamma_i > 0$ for $i = 1, 2, \ldots, n$. The standard word corresponding to γ, denoted by $\mathrm{Sw}(\gamma)$, is described by the recurrences of the form:

$$x_{-1} = b, \qquad x_0 = a, \qquad \ldots, \qquad x_n = x_{n-1}^{\gamma_{n-1}} x_{n-2}, \qquad x_{n+1} = x_n^{\gamma_n} x_{n-1} \tag{1}$$

where $\mathrm{Sw}(\gamma) = x_{n+1}$. For simplicity we denote $q_i = |x_i|$.

The sequence of words $\{x_i\}_{i=0}^{n+1}$ is called the standard sequence. Every word occurring in a standard sequence is a standard word, and every standard word occurs in some standard sequence. We assume that the standard word given by the empty directive sequence is a and $\mathrm{Sw}(0) = b$. The class of all standard words is denoted by $\mathcal{S}$.

Example 3.
Consider the directive sequence $\gamma = (1, 2, 1, 3, 1)$. We have $\mathrm{Sw}(\gamma) = x_5$, where:

$$\begin{array}{llll}
x_{-1} = b & & & q_{-1} = 1 \\
x_0 = a & & & q_0 = 1 \\
x_1 = (x_0)^1 \cdot x_{-1} = a \cdot b & & & q_1 = 2 \\
x_2 = (x_1)^2 \cdot x_0 & = ab \cdot ab \cdot a & & q_2 = 5 \\
x_3 = (x_2)^1 \cdot x_1 & = ababa \cdot ab & & q_3 = 7 \\
x_4 = (x_3)^3 \cdot x_2 & = ababaab \cdot ababaab \cdot ababaab \cdot ababa & & q_4 = 26 \\
x_5 = (x_4)^1 \cdot x_3 & = ababaababaababaababaababa \cdot ababaab & & q_5 = 33
\end{array}$$

Without loss of generality we consider here the standard Sturmian words starting with the letter a, therefore we assume that $\gamma_0 > 0$. The words starting with the letter b can be considered similarly.

Remark 4.
The special kind of standard words are well known Fibonacci words. They are formed by repeated concatenation in the same way that the Fibonacci numbers are formed by repeated addition. By the definition Fibonacci words are standard words given by directive sequences of the form $\gamma = (1, 1, \ldots, 1)$ (n-th Fibonacci word F_n corresponds to a sequence of n ones).

The number $N = |\mathrm{Sw}(\gamma)|$ is the (real) size of the word, while $(n+1) = |\gamma|$ can be thought as its compressed size. Observe that, by the definition of standard words, N is exponential with respect to n. Each directive sequence corresponds to a *grammar-based compression*, which consists in describing a given word by a context-free grammar G generating this (single) word. The size of the grammar G is the total length of all productions of G. In our case the size of the grammar is proportional to the length of the directive sequence.

2.1 The structure of cubes in standard words

The main idea of the computation of distinct cubes in a standard word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ is the partition of them into separate categories depending on the length of their periods. In this section we define the concepts of the i-partition of standard words and the generative run, which will be crucial in cubes enumeration. The following fact is a direct consequence of recurrent definition of standard words.

Fact 1
Every standard word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ *can be represented as a sequence of concatenated words* x_i *and* x_{i-1}*, and has the form:*

$$x_i^{\alpha_1}\, x_{i-1}\, x_i^{\alpha_2}\, x_{i-1} \ldots x_i^{\alpha_s}\, x_{i-1}\, x_i \qquad \text{or} \qquad x_i^{\beta_1}\, x_{i-1}\, x_i^{\beta_2}\, x_{i-1} \ldots x_i^{\beta_s}\, x_{i-1},$$

where $\alpha_k,\ \beta_k \in \{\gamma_i,\ \gamma_i + 1\}$*, and* x_i *are as in equation (1).*

Such a decomposition of a standard word w is called the *i-partition* of w. The block x_i is then the *repeating block* and the block x_{i-1} – the *single block.*

Example 5. Recall the word $\mathrm{Sw}(1,2,1,3,1)$ from Example 3. We have then:

$\mathrm{Sw}(1,2,1,3,1)$	*ababaabababaababababaabababaababaab*
$1-$ partition	$x_1^2\, x_0\, x_1^3\, x_0\, x_1^3\, x_0\, x_1^3\, x_0\, x_1^2\, x_0\, x_1$
$2-$ partition	$x_2\, x_1\, x_2\, x_1\, x_2\, x_1\, x_2^2\, x_1$
$3-$ partition	$x_3^3\, x_2\, x_3$
$4-$ partition	$x_4\, x_3$

See Figure 2 for comparison.

The following facts characterize the possible bases of distinct cubes in standard words. Their thesis are consequence of the very special structure of the subword graphs (especially their compacted versions) of those words. For more information on the subword graphs of standard words see for instance [3] and [17].

Lemma 6 (See [9]).
Let $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ *be a standard Sturmian word and* v *be a factor of* w *such that* $|x_i| \le |v| < |x_{i+1}|$*, where* x_i *are as in equation (1). Then:*

1. *There is at most one position in* x_i *(respectively* x_{i-1}*) such that any occurrence of* v *in* w *which starts in some* x_i*-block (respectively* x_{i-1}*-block) of the i-partition of* w *has to start at this particular position in* x_i *(respectively* x_{i-1}*).*
2. *If* v *can start at position* k *in* x_i *and at position* l *in* x_{i-1} *(*k *and* l *are uniqye by 1), then we have* $k = l$*.*

Lemma 7.
The base of each cube in the standard word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ *has the length* $k \cdot |x_i|$*, where* $0 < k < \gamma_i$ *and* x_i*'s are as in equation (1). The min-period of each cube equals* q_i *for some* $0 \leq i \leq n$*.*

Proof. Let $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ be a standard word and $v = u^3$ be a cube in w such that $|x_i| \leq |u| < |x_{i+1}|$. We denote $v = u^{(1)} \cdot u^{(2)} \cdot u^{(3)}$ to be able to refer to each occurrence of u in v. Due to Lemma 6, the factors $u^{(1)}$, $u^{(2)}$ and $u^{(3)}$ start at the same (within the block) position l of some blocks of the i-partition of w. The distance between two consecutive l position could be either $k \cdot |x_i|$ or $k \cdot |x_i| + |x_{i-1}|$. Recall that every occurrence of x_{i-1} block is separated by γ_i or $\gamma_i + 1$ occurrences of the x_i block. Since $|v| < |x_{i+1}|$ and $|x_{i+1}| = \gamma_i |x_i| + |x_{i-1}|$ we have $k < \gamma_i$ and the only possible base of v is $|u| = k \cdot |x_i|$, for $0 < k < \gamma_i$. Moreover, every standard word x_i is primitive, hence the minimal period of v has the length $|x_i| = q_i$.

We say that a cube is of *type* i if its min-period equals q_i. The number of distinct cubes of the type i in the word $\mathrm{Sw}(\gamma)$ is denoted by $\pi_i(\gamma)$.

Example 8. Let $\mathrm{Sw}(1, 2, 1, 3, 1) = ababaababababaababababaabababaababaab$ be a standard word. We have 2 cubes of the type 1 and 7 cubes of type 3, see Figure 1 and Example 2 for comparison.

For each $0 \leq i \leq n$ let *gen-run*(i) be the value (as a word) of the longest run with minimal period equal to q_i. It is called a *generative run* of type i (see Figure 2 for an example).

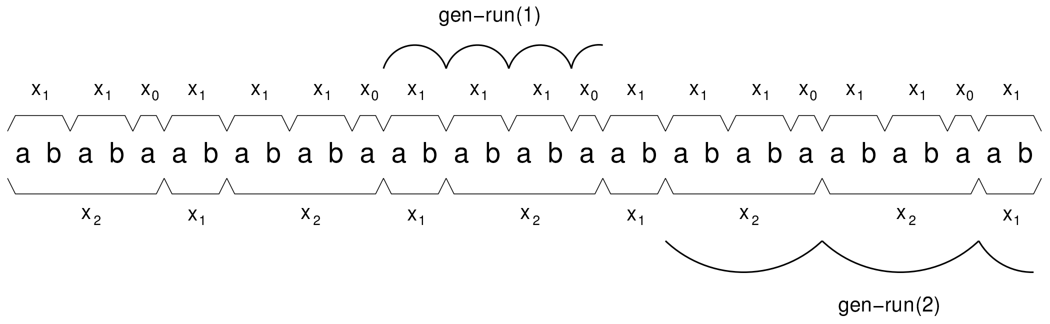

gen−run(1)
x1 x1 x0 x1 x1 x1 x0 x1 x1 x1 x0 x1 x1 x1 x0 x1 x1 x0 x1
a b a b a a b a b a b a a b a b a b a a b a b a b a a b a b a b a a b
x2 x1 x2 x1 x2 x1 x2 x2 x1
gen−run(2)

Figure 2. The 1-partition (above) and 2-partition (below) of the word $\mathrm{Sw}(1, 2, 1, 3, 1)$. We have *gen-run*$(1) = x_1^3 x_o$, *gen-run*$(2) = x_2^2 x_1$. The first generative run produces two different cubes, the second produces no cubes

Lemma 9 (See [3]).
Each generative run of the type i *is of the form:*

$$gen\text{-}run(i) \;=\; (x_i)^{\alpha} \cdot y,$$

where y *is a proper prefix of* x_i*.*

Example 10. Let $w = \mathrm{Sw}(1, 2, 1, 3, 1)$ (see Figure 1). The generative run of type 1 has the form *gen-run*$(1) = (x_1)^3 x_0$ and generates two cubes $(ab)^3$ and $(ba)^3$. On the other hand the generative run of type 2 has the form *gen-run*$(2) = (x_2)^2 x_1$ and does not generate any cube.

3 Formula and algorithm for counting the number of cubes

In this section we present and prove formulas for the number of distinct cubes in any standard word, that depend only on its compressed representation – the directive sequence. The following zero-one function for testing the value of the remainder of the division by 3 of a nonnegative integer x will be useful to simplify those formulas:

$$\mathbf{3}_k(x) \;=\; \begin{cases} 1 & \text{if } x \bmod 3 = k \\ 0 & \text{if } x \bmod 3 \neq k \end{cases}.$$

Recall that $q_i = |x_i|$ and π_i is the number of cubes of the type i in the word $\mathrm{Sw}(\gamma)$.

Theorem 11 (Main-Formulas).
The number of cubes in standard word $\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ *is given by the formula:*

$$cubes(\gamma_0, \gamma_1, \ldots, \gamma_n) \;=\; \sum_{i=0}^{n} \pi_i(\gamma_0, \gamma_1, \ldots, \gamma_n),$$

where:

$$\boxed{\begin{aligned}
&\textbf{(1)}\ (\ i \in [0, n-3]\ ) \ \Rightarrow\ \pi_i(\gamma) \;=\; \Big\lfloor \frac{\gamma_i + 1}{3} \Big\rfloor q_i + \mathbf{3}_1(\gamma_i) \cdot \Big(q_{i-1} - 1\Big) \\
&\textbf{(2)}\ \pi_{n-2}(\gamma) \;=\; \begin{cases} \left\lfloor \dfrac{\gamma_{n-2}+1}{3} \right\rfloor q_{n-2} + \mathbf{3}_1(\gamma_{n-2}) \cdot \Big(q_{n-3} - 1\Big) & if\ \ \gamma_n > 1 \\ \left\lfloor \dfrac{\gamma_{n-2}}{3} \right\rfloor \cdot q_{n-2} + \mathbf{3}_2(\gamma_{n-2}) \cdot \Big(q_{n-3} + 1\Big) & if\ \ \gamma_n = 1 \end{cases} \\
&\textbf{(3)}\ \pi_{n-1}(\gamma) \;=\; \left\lfloor \frac{\gamma_{n-1}}{3} \right\rfloor \cdot q_{n-1} + \mathbf{3}_2(\gamma_{n-1}) \cdot \Big(q_{n-2} - 1\Big) \\
&\textbf{(4)}\ \pi_n(\gamma) \;=\; \left\lfloor \frac{\gamma_n - 1}{3} \right\rfloor \cdot q_n + \mathbf{3}_0(\gamma_n) \cdot \Big(q_{n-1} + 1\Big)
\end{aligned}}$$

The proof of the above theorem is a matter of Section 4. Let us see some examples.

Example 12. Let $\mathrm{Sw}(1,2,1,3,1)$ be a standard word. Using formulas from Theorem 11 we have:

$$\pi_0(1,2,1,3,1) \;=\; \pi_2(1,2,1,3,1) \;=\; \pi_4(1,2,1,3,1) \;=\; 0$$
$$\pi_1(1,2,1,3,1) \;=\; 2 \qquad\qquad \pi_3(1,2,1,3,1) \;=\; 7$$

and finally

$$cubes(1,2,1,3,1) \;=\; 9.$$

See Example 8 and Figure 1 for comparison.

The number of cubes in Fibonacci words is given by the formula

$$cubes(F_n) \;=\; f_{n-3} - n + 2,$$

where f_k denotes the k-th Fibonacci number (see [7] for the proof). As the next example we derive this formula using results from Theorem 11.

Example 13. Recall that the n-th Fibonacci word F_n is defined as:

$$F_n = \mathrm{Sw}(\underbrace{1, 1, \ldots, 1}_{n}).$$

Hence

$$(\gamma_0, \gamma_1, \ldots, \gamma_{n-1}) \;=\; (1, 1, \ldots, 1),$$

and for each $i = 0, 1, \ldots, n-4$, we have

$$\pi_i(1, 1, \ldots, 1) \;=\; f_{i-1} - 1.$$

Moreover

$$\pi_{n-3}(1, 1, \ldots, 1) \;=\; \pi_{n-2}(1, 1, \ldots, 1) \;=\; \pi_{n-1}(1, 1, \ldots, 1) \;=\; 0.$$

Taking into account the identity

$$\sum_{i=-1}^{k} f_i \;=\; f_{k+2} - 1$$

we have

$$\begin{aligned} cubes(\underbrace{1, \ldots, 1}_{n}) &= \sum_{i=0}^{n-4}(f_{i-1} - 1) &&= \sum_{i=-1}^{n-5}(f_i - 1) \\ &= f_{n-3} - 1 - (n-3) &&= f_{n-3} - n + 2 \end{aligned}$$

Theorem 14.
The number of cubes in a standard word $\mathrm{Sw}(\gamma)$ *can be computed in linear time with respect to the length of the directive sequence* γ *(which is at least logarithmically smaller than the real length of the whole word* $\mathrm{Sw}(\gamma)$*).*

Proof.
The formulas for the number of cubes in a standard word $\mathrm{Sw}(\gamma)$ depend directly on the components of the directive sequence γ and the numbers q_i (namely $|x_i|$), see Theorem 11. Recall that, by the equation (1), we have

$$q_{i+1} \;=\; \gamma_i \cdot q_i + q_{i+1},$$

hence every number q_i can be computed by iteration of the equation (1) i times. We can compute the numbers q_0, q_1, $\ldots$, q_n consecutively and at each step i of the computation remember the number of cubes related to the value of q_i. The number of iterations performed by the algorithm corresponds directly to the length of the directive sequence, hence it has the time complexity $\mathrm{O}(|\gamma|)$. See Algorithm 1 for details.

Algorithm 1: *Cubes*(Sw(γ))

```
1 cubes ⟵ 0;
2 $q_{-1} \longleftarrow 1$;
3 $q_0 \longleftarrow 0$;
4 for $k := 0$ to $n$ do
5     $q_k \longleftarrow \gamma_k\, q_{k-1} + q_{k-2}$ ;
6     update cubes depending on the value of $\gamma_k$;
7 return cubes;
```

4 Proof of Theorem 11

Let us denote by $\widehat{w}$ the word w with two last letters removed and by $\widetilde{w}$ the word w with two last letters exchanged.

The following fact will be useful in proofs and can be shown by a simple induction, see for instance [12].

Lemma 15.
Let x_i be as in equation (1) and $i > 1$. Then:

(a) $x_{i-1} \cdot x_i \;=\; x_i \cdot \widetilde{x_{i-1}}$

(b) *The length of the longest prefix of $x_{i-1}x_i$ with period q_i equals $|x_i\widehat{x_{i-1}}|$.*

Example 16. Recall the word Sw(1, 2, 1, 3, 1) from Example 3. We have $x_2 = ababa$, $x_1 = ab$ and $\widetilde{x_1} = ba$. Therefore

$$x_1 \cdot x_2 \;=\; ab \cdot ababa \;=\; ababa \cdot ba \;=\; x_2 \cdot \widetilde{x_1}.$$

Let us fix throughout this section a standard word $w = \mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$. We show each point of Theorem 11 separately.

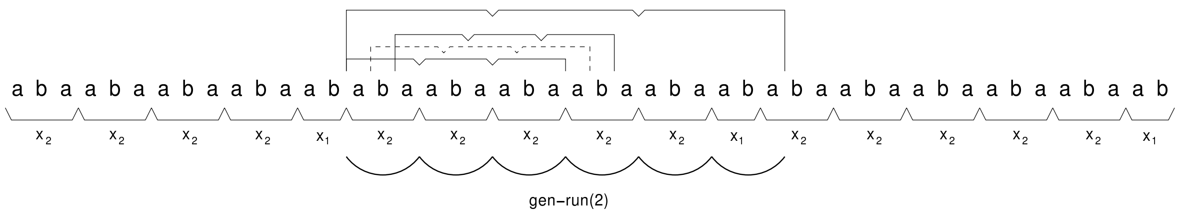

a b a a b a a b a a b a a b a b a a b a a b a a b a a b a a b a b a a b a a b a a b a a b a a b
x2 x2 x2 x2 x1 x2 x2 x2 x2 x2 x1 x2 x2 x2 x2 x2 x1
gen−run(2)

Figure 3. The illustration of Lemma 17: the structure of *gen-run*(2) and cubes of type 2 in the word Sw(1, 2, 4, 1, 2)

Lemma 17.

(a) $i \le n-3 \implies \textit{gen-run}(i) \;=\; (x_i)^{\gamma_i+2} \cdot \widehat{x_{i-1}}$

(b) *The point (1) from Theorem 11 is correct.*

Proof.

Point (a)

Let $w = \mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a standard word. Due to Fact 1 its i-partition has the form:

$$x_i^{\alpha_1}\, x_{i-1}\, x_i^{\alpha_2}\, x_{i-1} \ldots x_i^{\alpha_s}\, x_{i-1}\, x_i \qquad \text{or} \qquad x_i^{\beta_1}\, x_{i-1}\, x_i^{\beta_2}\, x_{i-1} \ldots x_i^{\beta_s}\, x_{i-1},$$

where $\alpha_k,\ \beta_k\ \in\ \{\gamma_i,\ \gamma_i + 1\}$. Let us consider the inner factor

$$v\ =\ (x_i)^{\gamma_i+1} \cdot x_{i-1} \cdot x_i.$$

Due to Lemma 15 the longest periodic prefix of v with period of the length $|x_i|$ (namely the generative run of type i) has the form:

$$(x_i)^{\gamma_i+2} \cdot \widehat{x_{i-1}}$$

and this concludes the proof of this point.

Point (b)

It is obvious that every cube of type i must be derived from the generative run of type i. Therefore, we have cubes with the bases: $q_i,\ 2 \cdot q_i,\ \ldots,\ \left\lfloor \frac{\gamma_i+1}{3} \right\rfloor \cdot q_i$. Each of them could be shifted to the right $q_i - 1$ times producing altogether q_i distinct cubes with the same base.

Moreover, if $\gamma_i \bmod 3\ =\ 1$, the subword $v = (x_i)^{\gamma_i+2}$ is also a cube. According to the structure of the generative run, v could be shifted to the right $q_{i-1} - 2$ times producing altogether $q_{i-1} - 1$ distinct cubes with the same base. See Figure 3 for an example of this case.

Finally the number of cubes of type i is given as:

$$\pi_i(\gamma)\ =\ \left\lfloor \frac{\gamma_i + 1}{3} \right\rfloor \cdot q_i + \mathbf{3}_1(\gamma_i) \cdot \Big(q_{i-1} - 1\Big).$$

This completes the proof of the lemma.

Lemma 18.

(a) *gen-run*$(n-2)$ $= \begin{cases} (x_{n-2})^{\gamma_{n-2}+2} \cdot \widehat{x_{n-3}} & \textit{for}\ \ \gamma_n > 1 \\ (x_{n-2})^{\gamma_{n-2}+1} \cdot x_{n-3} & \textit{for}\ \ \gamma_n = 1 \end{cases}$

(b) *The point (2) from Theorem 11 is correct.*

Proof.

Point (a)

The case of $\gamma_n > 1$ folows the same argumentation as in proof of Lemma 17, hence we can assume $\gamma_n = 1$. The standard word $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_{n-1}, 1)$ has the form:

$$w\ =\ \overbrace{(\underbrace{x_{n-2} \cdots x_{n-2}}_{\gamma_{n-2}} \cdot x_{n-3}) \cdots (\underbrace{x_{n-2} \cdots x_{n-2}}_{\gamma_{n-2}} \cdot x_{n-3})}^{\gamma_{n-1}} \cdot x_{n-2} \cdot (\underbrace{x_{n-2} \cdots x_{n-2}}_{\gamma_{n-2}} \cdot x_{n-3}).$$

The longest run with the period of the length q_i (namely the generative run of type i) is the suffix of w:

$$(x_{n-2})^{\gamma_{n-2}+1} \cdot x_{n-3}$$

and this concludes the proof of this point.

Point (b)

Similarly as in the proof of Point (a) we assume $\gamma_n = 1$. Every cube of type $n-2$ is derived from the generative run of type $n-2$. Therefore we have q_{n-2} cubes for each base length: q_i, $2 \cdot q_i$, …, $\lfloor \frac{\gamma_{n-2}}{3} \rfloor \cdot q_i$. Moreover, if $\gamma_{n-2} \bmod 3 = 2$, the factor $(x_{n-2})^{\gamma_{n-2}+1}$ is also a cube, which could be shifted q_{n-3} times. Hence we have $q_{n-3}+1$ additional cubes with the base $\frac{\gamma_{n-2}+1}{3} \cdot q_{n-2}$. See Figure 4 for an example of this case.

Finally we have

$$\pi_{n-2}(\gamma) \;=\; \left\lfloor \frac{\gamma_{n-2}}{3} \right\rfloor \cdot q_{n-2} + \mathbf{3}_2(\gamma_{n-2}) \cdot \Big(q_{n-3}+1\Big)$$

and the proof is complete.

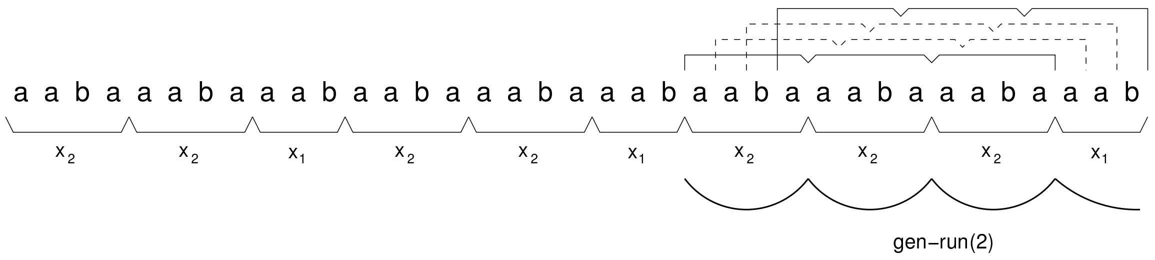

a a b a a a b a a a b a a b a a a b a a a b a a b a a a b a a a b a a a b
x2
x2
x1
x2
x2
x1
x2
x2
x2
x1
gen−run(2)

Figure 4. The illustration of the Lemma 18: the structure of *gen-run*(2) and cubes of the type 2 (i.e. type $n-2$) in the word $\mathrm{Sw}(2,1,2,2,1)$

Lemma 19.

(a) *gen-run*$(n-1) \;=\; (x_{n-1})^{\gamma_{n-1}+1} \cdot \widehat{x_{n-2}}$

(b) *The point (3) from Theorem 11 is correct.*

Proof.

Point (a)

By definition the word $w = \mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ has the form:

$$w \;=\; \overbrace{\Big(\underbrace{x_{n-1}\cdots x_{n-1}}_{\gamma_{n-1}} \cdot x_{n-2}\Big) \cdot \Big(\underbrace{x_{n-1}\cdots x_{n-1}}_{\gamma_{n-1}} \cdot x_{n-2}\Big) \cdots \Big(\underbrace{x_{n-1}\cdots x_{n-1}}_{\gamma_{n-1}} \cdot x_{n-2}\Big)}^{\gamma_n} \cdot x_{n-1}.$$

Due to Lemma 15 the longest periodic factor of w with period of the length $|x_{n-1}|$ (namely the generative run of type $n-1$) has the form:

$$(x_{n-1})^{\gamma_{n-1}+1} \cdot \widehat{x_{n-2}}$$

and this concludes the proof of this point.

Point (b)

According to the structure of *gen-run*$(n-1)$ we have q_{n-1} cubes for each base length: q_{n-1}, $2 \cdot q_{n-1}$, ..., $\lfloor \frac{\gamma_{n-1}}{3} \rfloor \cdot q_{n-1}$. Moreover, if $\gamma_{n-1} \bmod 3 \;=\; 2$, the factor $(x_{n-1})^{\gamma_{n-1}+1}$ is also a cube, which could be shifted $q_{n-1} - 2$ times. Hence we have $q_{n-2} - 1$ additional cubes with the base $\frac{\gamma_{n-1}+1}{3} \cdot q_{n-1}$. See Figure 5 for an example of this case.

Finally we have

$$\pi_{n-1}(\gamma_0, \gamma_1, \ldots, \gamma_n) \;=\; \left\lfloor \frac{\gamma_{n-1}}{3} \right\rfloor \cdot q_{n-1} + \mathbf{3}_2(\gamma_{n-1}) \cdot \Big(q_{n-2} - 1\Big).$$

and this concludes the proof.

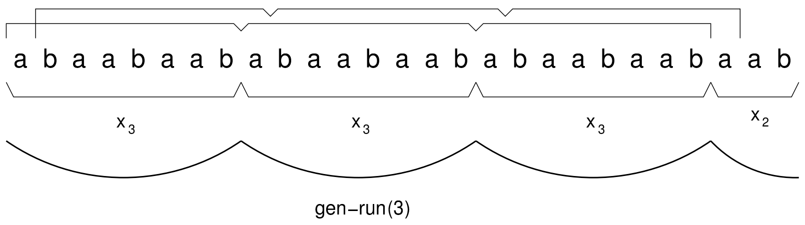


Figure 5. The illustration of the Lemma 19: the structure of *gen-run*(3) and cubes of the type 3 (i.e. type $n-1$) in the word $\mathrm{Sw}(1,1,2,2,1)$

Lemma 20.

(a) *gen-run*$(n) \;=\; (x_n)^{\gamma_n} \cdot x_{n-1}$

(b) *The point (4) from Theorem 11 is correct.*

Proof.

Point (a)

By definition the word $w = \mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ has the form:

$$w \;=\; \underbrace{x_n \cdot x_n \cdots x_n}_{\gamma_n} \cdot x_{n-1}.$$

Since x_{n-1} is the prefix of x_n, the value of generative run of type n is the whole word w.

Point (b)

According to the structure of *gen-run*(n) we have q_n cubes for each base length: q_n, $2 \cdot q_n$, ..., $\lfloor \frac{\gamma_n - 1}{3} \rfloor \cdot q_n$. Moreover, if $\gamma_n \bmod 3 \;=\; 0$, the factor $(x_n)^{\gamma_n}$ is also a cube, which could be shifted q_{n-1} times. Hence we have $q_{n-1} + 1$ additional cubes with the base $\frac{\gamma_n}{3} \cdot q_n$. See Figure 6 for an example of this case.

Finally we have

$$\pi_n(\gamma_0, \gamma_1, \ldots, \gamma_n) \;=\; \left\lfloor \frac{\gamma_n - 1}{3} \right\rfloor \cdot q_n + \mathbf{3}_0(\gamma_n) \cdot \Big(q_{n-1} + 1\Big)$$

and this completes the proof.

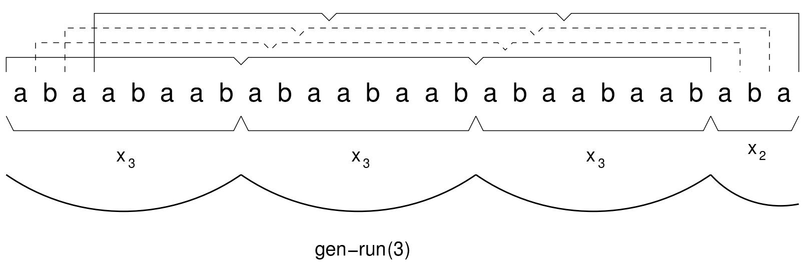


Figure 6. The illustration of Lemma 20: the structure of *gen-run*(3) and cubes of the type 3 (i.e. type n) in the word $\mathrm{Sw}(1, 1, 2, 3)$

Proof (of Theorem 11).
The sets of distinct cubes of type i and distinct cubes of type j are disjoint for $i \neq j$. Therefore, the thesis of Theorem 11 follows by summing up the formulas for number of cubes of all types from Lemma 17, Lemma 18, Lemma 19 and Lemma 20.

5 Standard words with large number of cubes

In this section we show the family of standard words rich in cubes. Experimental evidence shows that asymptotically this family achieves the highest ratio of the number of cubes to the length of the word.

Theorem 21.
Let $\gamma^k = (\underbrace{1, \ldots, 1}_{k}, 2, 3, 1)$ be a directive sequence and $w_k = \mathrm{Sw}(\gamma^k)$ be a standard word. We have:

$$\lim_{k\to\infty} \frac{cubes(w_k)}{|w_k|} = \frac{3\phi + 2}{9\phi + 4} \approx 0.36924841\ldots$$

where $\phi = \frac{\sqrt{5}+1}{2}$.

Proof. Denote by f_k the k-th Fibonacci numer:

$$f_{-1} = 1; \quad f_0 = 1; \quad f_1 = 2; \quad f_2 = 3; \quad f_4 = 5; \quad \ldots$$

By definition of standard words we have

$$|\mathrm{Sw}(\gamma_k)| = 9f_k + 4f_{k-1}.$$

According to Theorem 11 we have

$$\pi_i \;=\; f_{i-1} - 1 \quad \text{for} \quad i = 0, \ldots, k-1,$$

$$\pi_k \;=\; f_{k-1} + 1, \qquad \pi_{k+1} \;=\; 2f_k + f_{k-1}, \qquad \pi_{k+2} \;=\; 0.$$

Taking into account the well known identity

$$\sum_{i=-1}^{k} f_k = f_{k+2} - 1$$

we obtain

$$\sum_{i=0}^{k-1}(f_{i-1}-1) \;=\; \sum_{i=-1}^{k-2}(f_i-1) \;=\; f_k-k-1.$$

Altogether we have

$$cubes\Big(\mathrm{Sw}(\gamma_k)\Big) = 3f_k + 2f_{k-1} - k.$$

Denote by

$$\beta_k \;=\; \frac{f_k}{f_{k-1}}.$$

Then we have

$$\lim_{k\to\infty} \beta_k \;=\; \phi.$$

Therefore

$$\begin{aligned}\lim_{k\to\infty}\frac{cubes(w_k)}{|w_k|} \;&=\; \lim_{k\to\infty}\frac{3f_k+2f_{k-1}-k}{9f_k+4f_{k-1}}\\ &=\; \lim_{k\to\infty}\frac{3\beta_k+2-O(1)}{9\beta_k+4}\\ &=\; \frac{3\phi+2}{9\phi+4}\\ &\approx\; 0.36924841\ldots\end{aligned}$$

Remark 22. The extensive experimentation strongly suggests that the coefficient $\frac{3\phi+2}{9\phi+4}$ from the last theorem equals also the upper bound for the asymptotic ratio between the number of cubes and size of a standard words, see Table 1 for some examples.

Directive sequence	**Length**	**Cubes**	**Ratio**
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$	46368	10926	0.2356366459
$(3,3,3,3,3,3,3,3,3,3,1,1,1,1,1,1,1,1,1,2,3,1)$	138069388	50878017	0.3684959985
$(5,5,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,3,1)$	1028890	379883	**0.3692163399**
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,3,1)$	125574	46349	**0.3690971061**
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,3,2)$	222491	50529	0.2271058155
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,1)$	96917	28637	0.2954796372
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,4,1)$	154231	46349	0.3005167573
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,1)$	81790	28637	0.3501283775
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,3,1)$	169358	61475	0.3629884623
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,4,3,1)$	213142	72421	0.3397781761
$(\underbrace{1,1,1,1,1,1,1,1,1,1,\ldots,1,1,1,1,1,1,1}_{40},2,3,1)$	3073549228	1134903130	**0.3692483985**

Table 1. The example standard words with the number of cubes and cubes density

References

1. J. Allouche and J. Shallit: *Automatic Sequences. Theory, Applications, Generalizations*, Cambridge University Press, 2003.
2. P. Baturo, M. Piątkowski, and W. Rytter: *The number of runs in Sturmian words*, in Proceedings of the 13th international conference on Implementation and Applications of Automata, vol. 5148 of Lecture Notes in Computer Science, Springer, 2008, pp. 252–261.
3. P. Baturo, M. Piątkowski, and W. Rytter: *Usefulness of directed acyclic subword graphs in problems related to standard Sturmian words.* International Journal of Foundations of Computer Science, 20(6) 2009, pp. 1005–1023.
4. J. Berstel: *Sturmian and Episturmian words: a survey of some recent results*, in Proceedings of the 2nd international conference on Algebraic informatics, vol. 4728 of Lecture Notes in Computer Science, Springer, 2007, pp. 23–47.
5. J. Berstel and J. Karhumaki: *Combinatorics on words: a tutorial.* Bulletin of the EATCS, 79 2003, pp. 178–228.
6. J. Berstel, A. Lauve, C. Reutenauer, and F. Saliola: *Combinatorics on Words: Christoffel Words and Repetitions in Words*, CRM monograph series, Providence, R.I: American Mathematical Society, 2009.
7. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen: *On the maximal number of cubic runs in a string*, in Proceedings of the International Conference on Implementation and Applications of Automata, 2010, pp. 227–238.
8. M. Crochemore and W. Rytter: *Jewels of Stringology: Text algorithms*, World Scientific, 2003.
9. D. Damanik and D. Lenz: *The index of Sturmian sequences.* European Journal of Combinatorics, 23(1) 2002, pp. 23–29.
10. C. S. Iliopoulos, D. Moore, and W. F. Smyth: *A characterization of the squares in a Fibonacci string.* Theoretical Computer Science, 172(1–2) 1997, pp. 281–291.
11. R. M. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in Proceedings of 12th International Symposium on Fundamentals of Computation Theory, vol. 1684 of Lecture Notes in Computer Science, Springer, 1999, pp. 374–385.
12. M. Lothaire: *Algebraic Combinatorics on Words*, vol. 90 of Encyclopedia of mathematics and its application, Cambridge University Press, 2002.
13. M. Lothaire: *Applied Combinatorics on Words*, vol. 105 of Encyclopedia of Mathematics and its Application, Cambridge University Press, 2005.
14. M. Piątkowski and W. Rytter: *Computing the number of cubic runs in standard Sturmian words*, in Proceedings of the 16-th Prague Stringology Conference, Czech Technical University, 2011, pp. 106–120.
15. M. Piątkowski and W. Rytter: *Asymptotic behaviour of the maximal number of squares in standard Sturmian words.* International Journal of Foundations of Computer Science, 23(2) 2012, pp. 303–321.
16. W. Rytter: *The structure of subword graphs and suffix trees of Fibonacci words.* Theoretical Computer Science, 363(2) 2006, pp. 211–223.
17. M. Sciortino and L. Zamboni: *Suffix automata and standard Sturmian words*, in Proceedings of the 11th International Conference on Developments in Language Theory, vol. 4588 of Lecture Notes in Computer Science, Springer, 2007, pp. 382–398.
18. J. Shallit: *Characteristic words as fixed points of homomorphisms*, Tech. Rep. CS-91-72, University of Waterloo, Department of Computer Science, 1991.
19. H. Uscka-Wehlou: *Digital lines, Sturmian words, and continued fractions*, PhD thesis, Department of Mathematics, Uppsala University, 2009.

Quasi-linear Time Computation of the Abelian Periods of a Word

Gabriele Fici[1], Thierry Lecroq[2], Arnaud Lefebvre[2], Élise Prieur-Gaston[2], and William F. Smyth[3]

[1] I3S, CNRS & Université Nice Sophia Antipolis, France
Gabriele.Fici@unice.fr
[2] LITIS EA4108, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France
{Thierry.Lecroq,Arnaud.Lefebvre,Elise.Prieur}@univ-rouen.fr
[3] Department of Computing and Software,
McMaster University, Hamilton ON L8S 4K1, Canada
smyth@mcmaster.ca

Abstract. In the last couple of years many research papers have been devoted to Abelian complexity of words. Recently, Constantinescu and Ilie (Bulletin EATCS 89, 167–170, 2006) introduced the notion of *Abelian period*. In this article we present two quadratic brute force algorithms for computing Abelian periods for special cases and a quasi-linear algorithm for computing all the Abelian periods of a word.

Keywords: Abelian period, Abelian repetition, weak repetition, design of algorithms, text algorithms, combinatorics on words

1 Introduction

An integer $p > 0$ is a (classical) period of a word $\boldsymbol{w}$ of length n if $\boldsymbol{w}[i] = \boldsymbol{w}[i+p]$ for any $1 \leqslant i \leqslant n-p$. Classical periods have been extensively studied in combinatorics on words [16] due to their direct applications in data compression and pattern matching.

The Parikh vector of a word $\boldsymbol{w}$ enumerates the cardinality of each letter of the alphabet in $\boldsymbol{w}$. For example, given the alphabet $\Sigma = \{a, b, c\}$, the Parikh vector of the word $\boldsymbol{w} = aaba$ is $(3, 1, 0)$. The reader can refer to [6] for a list of applications of Parikh vectors.

An integer p is an *Abelian period* for a word $\boldsymbol{w}$ over a finite alphabet $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$ if $\boldsymbol{w}$ can be written as $\boldsymbol{w} = \boldsymbol{u}_0\boldsymbol{u}_1 \cdots \boldsymbol{u}_{k-1}\boldsymbol{u}_k$ where for $0 < i < k$ all the $\boldsymbol{u}_i$'s have the same Parikh vector $\mathcal{P}$ such that $\sum_{i=1}^{\sigma} \mathcal{P}[i] = p$ and the Parikh vectors of $\boldsymbol{u}_0$ and $\boldsymbol{u}_k$ are contained in $\mathcal{P}$ [11]. For example, the word $\boldsymbol{w} = ababbbabb$ can be written as $\boldsymbol{w} = \boldsymbol{u}_0\boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_3$, with $\boldsymbol{u}_0 = a$, $\boldsymbol{u}_1 = bab$, $\boldsymbol{u}_2 = bba$ and $\boldsymbol{u}_3 = bb$, and 3 is an Abelian period of $\boldsymbol{w}$ with Parikh vector $(1, 2)$ over $\Sigma = \{a, b\}$.

This definition of Abelian period matches that of *weak repetition* (also called *Abelian power*) when $\boldsymbol{u}_0$ and $\boldsymbol{u}_k$ are the empty word and $k > 2$ [12].

In the last couple of years many research papers have been devoted to Abelian complexity [13,1,8,3,14,2,4,20]. Efficient algorithms for Abelian Pattern Matching (also known as Jumbled Pattern Matching) have been designed [10,5,6,17,18,7].

Recently [15] gave algorithms for computing all the Abelian periods of a word of length n in time $O(n^2 \times \sigma)$. This was improved to time $O(n^2)$ in [9].

In this article we present a quasi-linear time algorithm for computing the Abelian periods of a word. In Section 2 we give some basic definitions and notation. Section 3 presents brute force algorithms while Section 4 presents our main contribution. Finally, Section 5 contains conclusions and perspectives.

2 Notation

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be a finite ordered alphabet of cardinality σ and Σ^* the set of words on alphabet Σ. We denote by $|\boldsymbol{w}|$ the length of the word $\boldsymbol{w}$. We write $\boldsymbol{w}[i]$ for the i-th symbol of $\boldsymbol{w}$ and $\boldsymbol{w}[i\,..\,j]$ for the factor of $\boldsymbol{w}$ from the i-th symbol to the j-th symbol, with $1 \leqslant i \leqslant j \leqslant |\boldsymbol{w}|$. We denote by $|\boldsymbol{w}|_a$ the number of occurrences of the symbol $a \in \Sigma$ in the word $\boldsymbol{w}$.

The *Parikh vector* of a word $\boldsymbol{w}$, denoted by $\mathcal{P}_{\boldsymbol{w}}$, counts the occurrences of each letter of Σ in $\boldsymbol{w}$; that is $\mathcal{P}_{\boldsymbol{w}} = (|\boldsymbol{w}|_{a_1}, \ldots, |\boldsymbol{w}|_{a_\sigma})$. Notice that two words have the same Parikh vector if and only if one word is a permutation of the other.

Given the Parikh vector $\mathcal{P}_{\boldsymbol{w}}$ of a word $\boldsymbol{w}$, we denote by $\mathcal{P}_{\boldsymbol{w}}[i]$ its i-th component and by $|\mathcal{P}_{\boldsymbol{w}}|$ the sum of its components. Thus for $\boldsymbol{w} \in \Sigma^*$ and $1 \leqslant i \leqslant \sigma$, we have $\mathcal{P}_{\boldsymbol{w}}[i] = |\boldsymbol{w}|_{a_i}$ and $|\mathcal{P}_{\boldsymbol{w}}| = \sum_{i=1}^{\sigma} \mathcal{P}_{\boldsymbol{w}}[i] = |\boldsymbol{w}|$.

Finally, given two Parikh vectors $\mathcal{P}, \mathcal{Q}$, we write $\mathcal{P} \subset \mathcal{Q}$ if $\mathcal{P}[i] \leqslant \mathcal{Q}[i]$ for every $1 \leqslant i \leqslant \sigma$ and $|\mathcal{P}| < |\mathcal{Q}|$.

Definition 1 ([11]). *A word $\boldsymbol{w}$ has an Abelian period (h, p) if $\boldsymbol{w} = \boldsymbol{u}_0\boldsymbol{u}_1 \cdots \boldsymbol{u}_{k-1}\boldsymbol{u}_k$ such that:*

- $\mathcal{P}_{\boldsymbol{u}_0} \subset \mathcal{P}_{\boldsymbol{u}_1} = \cdots = \mathcal{P}_{\boldsymbol{u}_{k-1}} \supset \mathcal{P}_{\boldsymbol{u}_k}$,
- $|\mathcal{P}_{\boldsymbol{u}_0}| = h$, $|\mathcal{P}_{\boldsymbol{u}_1}| = p$.

We call $\boldsymbol{u}_0$ and $\boldsymbol{u}_k$ resp. the *head* and the *tail* of the Abelian period. Notice that the length $t = |\boldsymbol{u}_k|$ of the tail is uniquely determined by h, p and $|\boldsymbol{w}|$, namely $t = (|\boldsymbol{w}| - h) \bmod p$.

The following lemma gives a bound on the maximum number of Abelian periods of a word.

Lemma 2 ([15]). *The maximum number of Abelian periods for a word of length n over the alphabet Σ is $\Theta(n^2)$.*

Proof. The word $(a_1a_2 \cdots a_\sigma)^{n/\sigma}$ has Abelian period (h, p) for any $p \equiv 0 \bmod \sigma$ and $h < p$. □

A natural order can be defined on the Abelian periods.

Definition 3. *Two distinct Abelian periods (h, p) and (h', p') of a word $\boldsymbol{w}$ are ordered as follows: $(h, p) < (h', p')$ if $p < p'$ or ($p = p'$ and $h < h'$).*

Definition 4 ([9]). *Let $\boldsymbol{w}$ be a word of length n. Then the mapping $pr : \Sigma \to A$, where A is the set of the first σ prime numbers, is defined by:*

$$pr(\sigma_i) = i\text{-th prime number.}$$

The P-signature of $\boldsymbol{w}$ is defined by:

$$P\text{-}signature(\boldsymbol{w}) = \Pi_{i=1}^{n} pr(\boldsymbol{w}[i]).$$

Definition 5 ([9]). *Let $\boldsymbol{w}$ be a word of length n. Then the mapping $s : \Sigma \to B$, where B is the set of the first $\sigma - 1$ powers of $n + 1$ and 0, is defined by:*

$$s(\sigma_i) = \begin{cases} 0 & \text{if } i = 1 \\ (n+1)^{i-2} & \text{otherwise.} \end{cases}$$

The S-signature of $\boldsymbol{w}$ is defined by:

$$S\text{-}signature(\boldsymbol{w}) = \sum_{i=0}^{n} s(\boldsymbol{w}[i]).$$

Observation 1 ([9]) *For a word $\boldsymbol{w}$ of length n the array Pr of n elements is defined by*

$$Pr[i] = \Pi_{j=1}^{i} pr(\boldsymbol{w}[j]),$$

then

$$P\text{-}signature(\boldsymbol{w}[k\,..\,\ell]) = \begin{cases} Pr[\ell]/Pr[k-1] & \text{if } k \neq 0 \\ Pr[\ell] & \text{otherwise.} \end{cases}$$

Observation 2 ([9]) *For a word $\boldsymbol{w}$ of length n the array S of n elements is defined by*

$$S[i] = \sum_{j=1}^{i} s(\boldsymbol{w}[j]),$$

then

$$S\text{-}signature(\boldsymbol{w}[k\,..\,\ell]) = \begin{cases} S[\ell] - S[k-1] & \text{if } k \neq 0 \\ S[\ell] & \text{otherwise.} \end{cases}$$

Example 6. $\boldsymbol{w}$ = `abaab`:

i	1	2	3	4	5
$\boldsymbol{w}[i]$	a	b	a	a	b
$pr(\boldsymbol{w}[i])$	2	3	2	2	3
$Pr[i]$	2	6	12	24	72

P-*signature*$(\boldsymbol{w}[3\,..\,5]) =$
P-*signature*(`aab`) $=$
$Pr[5]/Pr[2] = 72/6 = 12$

i	1	2	3	4	5
$\boldsymbol{w}[i]$	a	b	a	a	b
$s(i)$	0	1	0	0	1
$S[i]$	0	1	1	1	2

S-*signature*$(\boldsymbol{w}[3\,..\,5]) =$
S-*signature*(`aab`) $=$
$S[5] - S[2] = 2 - 1 = 1$

3 Brute Force Algorithms

We will first focus on the case where we consider periods without head nor tail.

In the remaining of the article we will write that a word $\boldsymbol{w}$ has Abelian period p whenever it has Abelian period $(0, p)$. When the tail is also empty, for a word $\boldsymbol{w}$ of length n an Abelian period p must divide n. We define:

– $P[i]$ is the set of Abelian periods of $\boldsymbol{w}[1\,..\,i]$;
– $V[i] = \mathcal{P}(\boldsymbol{w}[1\,..\,i])$ is the Parikh vector of $\boldsymbol{w}[1\,..\,i]$.

3.1 Abelian periods with neither head nor tail

In a first step we set $P[i] = \{i\}$ for all the divisors of n. Then we process the positions i of $\boldsymbol{w}$ in ascending order: if $j \in P[i]$ and $\mathcal{P}_{\boldsymbol{w}}[i+1\,..\,i+j] = \mathcal{P}_{\boldsymbol{w}}[1\,..\,j]$, then we add j to $P[i+j]$. This test can be done in $O(\sigma)$ time by precomputing the Parikh vectors of all the prefixes of $\boldsymbol{w}$ or in constant time using signatures. At the end of the process $P[n]$ contains all the Abelian periods of $\boldsymbol{w}$ with neither head nor tail (see algorithm in Figure 1).

```
AbelianPeriodsNoHeadNoTail($\boldsymbol{w}, n$)
 1  $V[i] \leftarrow \mathcal{P}(\boldsymbol{w}[1\,..\,i]), \forall\, 1 \le i \le n$
 2  $P[i] \leftarrow \emptyset, \forall\, 1 \le i \le n$
 3  for $i \leftarrow 1$ to $n/2$ do
 4      if $n \bmod i = 0$ then
 5          $P[i] \leftarrow \{i\}$
 6  for $i \leftarrow 1$ to $n-1$ do
 7      for $j \in P[i]$ do
 8          if $V[i+j] - V[i] = V[j]$ then
 9              $P[i+j] \leftarrow P[i+j] \cup \{j\}$
10  return $P[n]$
```

Figure 1. Compute the Abelian periods with no head and no tail of a word $\boldsymbol{w}$ of length n

```
AbelianPeriodsNoHeadWithTail($\boldsymbol{w}, n$)
 1  $V[i] \leftarrow \mathcal{P}(\boldsymbol{w}[1\,..\,i]), \forall\, 1 \leqslant i \leqslant n$
 2  $P[i] \leftarrow \{i\}, \forall\, 1 \leqslant i \leqslant n/2$
 3  $P[i] \leftarrow \emptyset, \forall\, n/2 < i \leqslant n$
 4  for $i \leftarrow 1$ to $n-1$ do
 5      for $j \in P[i]$ do
 6          if $i + j > n$ then
 7              if $V[n] - V[i+1] \le V[j]$ then
 8                  $P[n] \leftarrow P[n] \cup \{j\}$
 9          else if $V[i+j] - V[i] = V[j]$ then
10              $P[i+j] \leftarrow P[i+j] \cup \{j\}$
11  return $P[n]$
```

Figure 2. Compute the Abelian periods without head and with a possibly non-empty tail of a word $\boldsymbol{w}$ of length n

Example 7. $\boldsymbol{w} =$ `abaababbbabaabbabbaaabbababbaa`:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
$\boldsymbol{w}[i]$	a	b	a	a	b	a	b	b	b	a	b	a	a	b	b	a	b	b	a	a	a	b	b	a	b	a	b	b	a	a
P	{1}	{2}	{3}		{5}	{6}				{10}					{15}					{10}										{10}
						{3}																								

Theorem 8. *The algorithm* ***AbelianPeriodsNoHeadNoTail*** *computes all the Abelian periods with neither head nor tail of a word* $\boldsymbol{w}$ *of length* n *in time* $O(n^2 \times \sigma)$ *if the test in line 8 is performed by comparing Parikh vectors and in time* $O(n^2)$ *if the test in line 8 is performed by using S-signatures or P-signatures.*

3.2 Abelian periods without head with tail

Now we consider Abelian periods without head and with a possibly non-empty tail. We adapt the previous algorithm by setting $P[i] = \{i\}$ for $1 \leqslant i \leqslant n/2$ (see algorithm Figure 2).

Theorem 9. *The algorithm* ***AbelianPeriodsNoHeadWithTail*** *computes all the Abelian periods without head and with tail of a word* $\boldsymbol{w}$ *of length* n *in time* $O(n^2 \times \sigma)$ *if the tests in lines 7 and 1 are performed by comparing Parikh vectors and in time* $O(n^2)$ *if the test in lines 7 and 1 are performed by using S-signatures or P-signatures.*

4 Quasi-Linear Time Computation of Abelian Periods with neither Head nor Tail

In a linear-time preprocessing phase we compute $\mathcal{P}_{\boldsymbol{w}}[j]$, $j = 1, 2, \ldots, \sigma$, the components of the Parikh vector of the word $\boldsymbol{w}$. Also we compute

$$g = \gcd(\mathcal{P}_{\boldsymbol{w}}[1], \mathcal{P}_{\boldsymbol{w}}[2], \ldots, \mathcal{P}_{\boldsymbol{w}}[\sigma])$$

and $q = n/g$. Without loss of generality we suppose $\sigma \geq 2$ and $g > 1$. In $O(\sqrt{g})$ time we compute a stack D of all divisors $1 \leq d \leq g$ of g in ascending order.

Definition 10. *The word $\boldsymbol{w}$ is an **Abelian repetition** of **period** p and **exponent** e if $p \mid n$ and each of the e substrings*

$$\boldsymbol{w}[1 \,..\, p], \boldsymbol{w}[p+1 \,..\, 2p], \ldots, \boldsymbol{w}[n-p+1 \,..\, n]$$

contains $(p \times \mathcal{P}_{\boldsymbol{w}}[j])/n = \mathcal{P}_{\boldsymbol{w}}[j]/e$ occurrences of the letter $\sigma_j \in \Sigma$ for any j.

In other words, an Abelian repetition of period p and exponent e is the concatenation of e strings all having the same Parikh vector $\mathcal{P}$ of length p.

Observation 3 *The only possible Abelian periods p of $\boldsymbol{w}$ are of the form $p = d \times q$, where d is an entry in D. Thus the smallest period is $d \times q$, where d is the least such entry.(Note that the last element of D is g.)*

Definition 11 (Segment). *A factor $\boldsymbol{w}[i \,..\, j]$ is a **segment** of $\boldsymbol{w}$ if:*

1. *$i = k \times q + 1$ with $k \geqslant 0$;*
2. *$j - i + 1 = t \times q$ with $t \geqslant 1$;*
3. *$\mathcal{P}_{\boldsymbol{w}[i..j]}[k]/(j - i + 1) = \mathcal{P}_{\boldsymbol{w}}[k]/|\boldsymbol{w}|$ for every letter $\sigma_k \in \Sigma$;*
4. *there does not exist a $j' < j$ such that $j' - i + 1 = t' \times q$ and $\mathcal{P}_{\boldsymbol{w}[i..j']}[k]/(j' - i + 1) = \mathcal{P}_{\boldsymbol{w}}[k]/|\boldsymbol{w}|$ for every letter $\sigma_k \in \Sigma$.*

In other words segments:

- start at positions multiples of q plus one;
- are non-empty and of length multiple of q;
- have the same proportion of every letter as the whole word $\boldsymbol{w}$;
- are of minimal length.

Since we suppose that $\boldsymbol{w}$ has Abelian period $p \in 1 \,..\, n/2$, it follows that either $\boldsymbol{w}$ itself is a segment or else consists of a concatenation of segments. Note that a segment is a minimum-length substring of Abelian period p.

Lemma 12. *The word $\boldsymbol{w}$ has Abelian period $d \times q$ if and only if for every $k = 0, 1, \ldots, n/(d \times q) - 1$, $k \times d \times q + 1$ is the starting position of a segment of $\boldsymbol{w}$.*

We begin by computing the segments of $\boldsymbol{w}$ (see Figure 3), making use of the precomputed values q and $\mathcal{P}_{\boldsymbol{w}}$. We compute a Boolean array L of n elements: for $1 \leqslant i \leqslant n$, $L[i] = 1$ iff i is the starting position of a segment, $L[i] = 0$ otherwise.

Observation 4 *If p is an Abelian period of $\boldsymbol{w}$ with neither head nor tail and T is the length of the longest segment of $\boldsymbol{w}$ divided by q, then $p \geqslant T$.*

```
ComputeSegments($\boldsymbol{w}, n, q, \mathcal{P}_{\boldsymbol{w}}$)
 1  $(i, T) \leftarrow (1, 0)$
 2  $L \leftarrow 0^n$
 3  while $i \leqslant n$ do
        ▷ Start a new segment
 4      $(i_0, j, t, count) \leftarrow (i, 0, 0, 0^\sigma)$
 5      while $j \leqslant \sigma$ do
            ▷ See if $t$ partitions of length $q$ form a segment
 6          $t \leftarrow t + 1$
 7          for $k \leftarrow 1$ to $q$ do
 8              $j \leftarrow \boldsymbol{w}[i]$
 9              $count[j] \leftarrow count[j] + 1$
10              $i \leftarrow i + 1$
            ▷ Check counts of letters $1 \mathinner{..} j$ from position $i_0$
11          $j \leftarrow 1$
12          $t' \leftarrow t \times q$
13          while $j \leqslant \sigma$ and $count[j] = (t' \times \mathcal{P}_{\boldsymbol{w}}[j])/n$ do
14              $j \leftarrow j + 1$
        ▷ Update the array $L$ and the maximum segment length $T$
15      $L[i_0] \leftarrow 1$
16      $T \leftarrow \max\{T, t\}$
17  return $(L, T)$
```

Figure 3. Compute a Boolean array L of the starting positions of the segments of $\boldsymbol{w}$ ordered from left to right, also the maximum number T of factors of length q in any segment

The procedure that computes L visits each position i in $\boldsymbol{w}$ once, and corresponding to each i performs constant-time processing: the internal **while** loop updates j at most σ times corresponding to each partition of length $q \geqslant \sigma$.

Proposition 13. *The algorithm* ComputeSegments($\boldsymbol{w}, n, q, \mathcal{P}_{\boldsymbol{w}}$) *computes the segments of a word* $\boldsymbol{w}$ *of length* n *on an alphabet of size* σ *in time* $O(n)$.

Example 14. $\boldsymbol{w} =$ `abaababbbabaabbabbaaabbababbaa`: $n = 30$, $\mathcal{P}_{\boldsymbol{w}} = (15, 15)$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
$\boldsymbol{w}[i]$	a	b	a	a	b	a	b	b	b	a	b	a	a	b	b	a	b	b	a	a	a	b	b	a	b	a	b	b	a	a
$L[i]$	1	0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	0	0
T	0	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

$\boldsymbol{w}$ is thus a concatenation of segments: $\boldsymbol{w} =$ `ab` · `aababb` · `ba` · `ba` · `ab` · `ba` · `bbaa` · `ab` · `ba` · `ba` · `bbaa` and $T = 3$.

The procedure, given in Figure 4, scans all the multiples of the divisors $d \in D$, their number is equal to the sum of the divisors of g which is in $O(n \log\log n)$ [19].

In practice, the case where $d = 1$ is treated in lines 5 and 7. If $T = 1$, it means that w can be segmented into factors of length q: q is then an Abelian period of w. The case where $d = g$ is treated outside the main loop, at the end of the algorithm: it corresponds to the trivial case where the Abelian period is n.

Example 15. $\boldsymbol{w} =$ `abaababbbabaabbabbaaabbababbaa`: $n = 30$, $\mathcal{P}_{\boldsymbol{w}}[1] = \mathcal{P}_{\boldsymbol{w}}[2] = 15$, $g = 15$, $q = 2$, $D = (1, 3, 5, 15)$ and $T = 3$. Since $T \neq 1$, q is not an Abelian period: case $d = 1$ is done. When $d = 3$, $p = 7$ and 7 is not a starting position of a segment. When $d = 5$, $p = 11$ and 11 is a starting position of a segment then $p = 21$

```
ComputesPeriod($\boldsymbol{w}, n$)
 1  Compute $\mathcal{P}_{\boldsymbol{w}}, g, D$
 2  $q \leftarrow n/g$
 3  $(L, T) \leftarrow$ ComputesSegments($\boldsymbol{w}, n, q, \mathcal{P}_{\boldsymbol{w}}$)
 4  $R \leftarrow \emptyset$
    ▷ Deal quickly with easy cases
 5  if $T = 1$ then
 6      $R \leftarrow R \cup \{q\}$
 7      $d \leftarrow$ Pop($D$)
    ▷ Fast forward in $D$ past impossible cases
 8  repeat
 9      $d \leftarrow$ Pop($D$)
10  until $d \geqslant T$
11  while $d < g$ do
12      $p \leftarrow d \times q + 1$
        ▷ Test if all multiples of $p$ are starting positions of segments
13      while $p < n$ do
14          if $L[p] = 1$ then
15              $p \leftarrow p + d \times q$
16          else break
17      if $p \geqslant n$ then
18          $R \leftarrow R \cup \{d \times q\}$
19      $d \leftarrow$ Pop($D$)
20  if $q \neq n$ then
21      $R \leftarrow R \cup \{n\}$
22  return $R$
```

Figure 4. In ascending order of divisors d of g, use the array L to determine whether or not $\boldsymbol{w}$ is an Abelian repetition of period $d \times q$

and 21 is a starting position of a segment: 10 is an Abelian period. The case where $d = 15$ is trivial since it corresponds to Abelian period n. Thus the algorithm returns $\{10, 30\}$. In the worst case the algorithm could have scanned all the multiples of 3 (they are 5) and all the multiples of 5 (they are 3) less than or equal to 15.

Theorem 16. *The algorithm* ComputesPeriod($\boldsymbol{w}, n$) *computes all the Abelian periods of* $\boldsymbol{w}$ *in time* $O(n \log \log n)$.

5 Conclusions and perspectives

In this article we gave brute force algorithms for computing Abelian periods for a word $\boldsymbol{w}$ of length n in the two following cases: no head, no tail and no head with tail. These algorithms run in time $O(n^2)$ but is this complexity tight? We also present a quasi-linear time algorithm for computing all the Abelian periods of a word in the case no head, no tail. Does an algorithm of the same complexity exist for a word $\boldsymbol{w}$ of length at most $n + q - 1$ containing a substring of length n that is an Abelian repetition with neither head nor tail of some period $dq \leq n$?

References

1. S. Avgustinovich, A. Glen, B. Halldórsson, and S. Kitaev: *On shortest crucial words avoiding abelian powers.* Discrete Applied Mathematics, 158(6) 2010, pp. 605–607.
2. S. Avgustinovich, J. Karhumäki, and S. Puzynina: *On Abelian repetition threshold.* RAIRO Theoretical Informatics and Applications, 46(1) 2012, pp. 3–15.
3. F. Blanchet-Sadri, J. I. Kim, R. Mercas, W. Severa, and S. Simmons: *Abelian square-free partial words*, in Proceedings of the 4th International Conference Language and Automata Theory and Applications, A.-H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 94–105.
4. F. Blanchet-Sadri, A. Tebbe, and A. Veprauskas: *Fine and Wilf's theorem for Abelian periods in partial words*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.
5. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching*, in Proceedings of the 5th International Conference on Fun with Algorithms, FUN 2010, P. Boldi and L. Gargano, eds., vol. 6099 of Lecture Notes in Computer Science, Springer, 2010, pp. 89–101.
6. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *Algorithms for jumbled pattern matching in strings.* International Journal of Foundations of Computer Science, 23(2) 2012, pp. 357–374.
7. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On Approximate Jumbled Pattern Matching in Strings.* Theory of Computing Systems, 50(1) 2012, pp. 35–51.
8. J. Cassaigne, G. Richomme, K. Saari, and L. Zamboni: *Avoiding Abelian powers in binary words with bounded Abelian complexity.* International Journal of Foundations of Computer Science, 22(4) 2011.
9. M. Christou, M. Crochemore, and C. S. Iliopoulos: *Identifying all Abelian periods of a string in quadratic time and relevant problems.* International Journal of Foundations of Computer Science, 2012, (accepted, see also Report arXiv:1207.1307v1).
10. F. Cicalese, G. Fici, and Zs. Lipták: *Searching for jumbled patterns in strings*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 105–117.
11. S. Constantinescu and L. Ilie: *Fine and Wilf's theorem for abelian periods.* Bulletin of the European Association for Theoretical Computer Science, 89 2006, pp. 167–170.
12. L. J. Cummings and W. F. Smyth: *Weak repetitions in strings.* Journal of Combinatorial Mathematics and Combinatorial Computing, 24 1997, pp. 33–48.
13. J. D. Currie and A. Aberkane: *A cyclic binary morphism avoiding Abelian fourth powers.* Theoretical Computer Science, 410(1) 2009, pp. 44–52.
14. M. Domaratzki and N. Rampersad: *Abelian primitive words*, in Proceedings of the 15th Conference on Developments in Language Theory, G. Mauri and A. Leporati, eds., vol. 6795 of Lecture Notes in Computer Science, Springer, 2011.
15. G. Fici, T. Lecroq, A. Lefebvre, and É. Prieur-Gaston: *Computing Abelian periods in words*, in Proceedings of the Prague Stringology Conference 2011, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2011, pp. 184–196.
16. M. Lothaire: *Algebraic Combinatorics on Words*, Cambridge University Press, 2002.
17. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Information Processing Letters, 110(18-19) 2010, pp. 795–798.
18. T. M. Moosa and M. S. Rahman: *Sub-quadratic time and linear space data structures for permutation matching in binary strings.* Journal of Discrete Algorithms, 10 2012, pp. 5–9.
19. G. Robin: *Grandes valeurs de la fonction somme des diviseurs et hypothèse de Riemann.* J. Math. Pures Appl. (9), 63(2) 1984, pp. 187–213.
20. A. Samsonov and A. Shur: *On Abelian repetition threshold.* RAIRO Theoretical Informatics and Applications, 46(1) 2012, pp. 147–163.

A Computational Framework for Determining Square-maximal Strings[⋆]

Antoine Deza, Frantisek Franek, and Mei Jiang

Advanced Optimization Laboratory
Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada
{deza,franek,jiangm5}@mcmaster.ca
http://optlab.cas.mcmaster.ca/

Abstract. We investigate the function $\sigma_d(n) = \max\{\boldsymbol{s}(x) \mid x \text{ is a } (d,n)\text{-string}\}$, where $\boldsymbol{s}(x)$ denotes the number of distinct primitively rooted squares in a string x and (d,n)-string denotes a string of length n with exactly d distinct symbols. New properties of the $\sigma_d(n)$ function are presented. The notion of s-cover is presented and discussed with emphasis on the recursive computational determination of $\sigma_d(n)$. In particular, we were able to determine all values of $\sigma_2(n)$ for $n \leq 53$ and $\sigma_3(n)$ for $n \leq 42$ and to point out that $\sigma_2(33) < \sigma_3(33)$; that is, among all strings of length 33, no binary string achieves the maximum number of distinct primitively rooted squares. Noticeably, these computations reveal the unexpected existence of pairs (d,n) satisfying $\sigma_{d+1}(n+2) - \sigma_d(n) > 1$ such as (2,33) and (2,34), and of three consecutive equal values: $\sigma_2(31) = \sigma_2(32) = \sigma_2(33)$. In addition we show that $\sigma_2(n) \leq 2n - 66$ for $n \geq 53$.

Keywords: string, square, primitively rooted square, maximum number of distinct primitively rooted squares, parameterized approach, $(d, n-d)$ table

1 Introduction

In [2] the notion of an r-cover was introduced as a means to represent the distribution of the runs in a string and thus describe the structure of the run-maximal strings. Ignoring the number of distinct symbols d, a key assertion from [2] states that essentially any run-maximal string has an r-cover. A similar approach was adapted for run-maximal (d,n)-strings in [1] and we show in Section 2 how this approach can be adapted for square-maximal (d,n)-strings. This notion is used to speed up computations of the maximum number of distinct primitively rooted squares.

We encode a square as a triple (s,e,p) where s is the starting position of the square, e is the ending position of the square, and p is its period. Note that $e = s + 2p - 1$. The *join* $x[i_1 \mathinner{..} i_k] \cup x[j_1 \mathinner{..} j_m]$ of two substrings of a string $x = x[1 \mathinner{..} n]$ is defined if $i_1 \leq j_1 \leq i_k + 1$ and then $x[i_1 \mathinner{..} i_k] \cup x[j_1 \mathinner{..} j_m] = x[i_1 \mathinner{..} max\{i_k, j_m\}]$, or if $j_1 \leq i_1 \leq j_m + 1$ and then $x[i_1 \mathinner{..} i_k] \cup x[j_1 \mathinner{..} j_m] = x[j_1 \mathinner{..} max\{i_k, j_m\}]$. Simply put, the join is defined when the two substrings either are adjacent or overlapping. The join $S_1 \cup S_2$ of two squares of x encoded as $S_1 = (s_1, e_1, p_1)$ and $S_2 = (s_2, e_2, p_2)$ is defined as the join $x[s_1 \mathinner{..} e_1] \cup x[s_2 \mathinner{..} e_2]$. The alphabet of x is denoted by $\mathcal{A}(x)$, (d,n)-string refers to a string of length n with exactly d distinct symbols, $\boldsymbol{s}(x)$ denotes the number of distinct primitively rooted squares in a string x, and $\sigma_d(n)$ refers to the maximum number of distinct primitively rooted squares over all (d,n)-strings,

[⋆] This work was supported by the *Natural Sciences and Engineering Research Council of Canada* and *MITACS*, and by the *Canada Research Chair program*, and made possible by the facilities of the *Shared Hierarchical Academic Research Computing Network* (http://www.sharcnet.ca/).

i.e. $\sigma_d(n) = \max\{\boldsymbol{s}(x) \mid x \text{ is a } (d,n)\text{-string}\}$. A singleton is a symbol which occurs exactly once in the string under consideration. To simplify the notation, for an empty string ε we set $\boldsymbol{s}(\varepsilon) = 0$ and $\sigma_d(0) = 0$.

2 Computational approach to distinct primitively rooted squares

In the computational framework for determining $\sigma_d(n)$ we will be discussing later, we first compute a lower bound of $\sigma_d(n)$ denoted as $\sigma_d^-(n)$. It is enough to consider (d,n)-strings x that could achieve $\boldsymbol{s}(x) > \sigma_d^-(n)$ for determining $\sigma_d(n)$, thus significantly reducing the search space. The purpose of this section is to introduce the necessary conditions that guarantee that for such an x, $\boldsymbol{s}(x) > \sigma_d^-(n)$ for a given $\sigma_d^-(n)$. The necessary conditions are the existence of an *s-cover* and a sufficient *density* of the string, see Lemmas 5, 9, 10. The s-cover is guaranteed through generation, while the density is verified incrementally during the generation at the earliest possible stages. Note that the notion of s-cover, though similar to r-cover for runs, see [1,2], is slightly different.

Definition 1. *An* s-cover *of a string $x = x[1 \mathrel{..} n]$ is a sequence of primitively rooted squares $\{S_i = (s_i, e_i, p_i) \mid 1 \le i \le m\}$ so that*

(1) *for any $1 \le i < m$, $s_i < s_{i+1} \le e_i + 1$ and $e_i < e_{i+1}$, i.e. two consecutive squares are either adjacent or overlapping;*

(2) $\displaystyle\bigcup_{1\le i\le m} S_i = x$*;*

(3) *for any occurrence of square S in x, there is $1 \le i \le m$ so that S is a substring of S_i, denoted by $S \subseteq S_i$.*

Lemma 2. *An s-cover of a string is unique.*

Proof. Let us assume that we have two different s-covers of x, $\{S_i \mid 1 \le i \le m\}$ and $\{S'_j \mid 1 \le j \le k\}$. We shall prove by induction that they are identical. By Definition 1 (3), $S_1 \subseteq S'_1$ and, by the same argument, $S'_1 \subseteq S_1$, and thus $S_1 = S'_1$. Let the induction hypothesis be $S_i = S'_i$ for $1 \le i \le t$. If $\bigcup_{1\le i\le t} S_i = x$, we have $t = m = k$ and we are done. Otherwise consider S_{t+1}. By Definition 1 (3), there is S'_v so that $S_{t+1} \subseteq S'_v$ and $v > t$. We need to show that $v = t+1$. If not, then S_{t+1} would neither be a substring of S'_t nor of S'_{t+1} contradicting Definition 1 (3). Therefore $S_{t+1} \subseteq S'_{t+1}$. By the same argument, $S'_{t+1} \subseteq S_{t+1}$ and so $S_{t+1} = S'_{t+1}$. □

Lemma 3. *If a string admits an s-cover, then it is singleton free.*

Proof. Let $\{S_j \mid 1 \le j \le m\}$ be the s-cover of $x = x[1 \mathrel{..} n]$. For any $1 \le i \le n$, $x[i] \in S_t$ for some t by Definition 1 (2). Since S_t is a square, the symbol $x[i]$ occurs in x at least twice. □

Before we can define what a *dense string* is, we must first define the notion of a *core* of a square, similarly to the core of a run, see [1,6]. For a square, its core is the set of indices formed by the intersection of the indices of all its occurrences in the string.

Definition 4. *The* core vector *$k(x)$ of a (d,n)-string x is defined by $k_i(x) =$ the number of cores of squares of x containing i for $i = 1,\ldots,n$. A singleton-free (d,n)-string x is* dense*, if its core vector $k(x)$ satisfies $k_i(x) > \sigma_d^-(n) - \boldsymbol{s}(x[1 \mathrel{..} i-1]) - m_i$ for $i = 1,\ldots,n$, where $m_i = \max\,\{\sigma_{d'}(n-i) \;:\; d - |\mathcal{A}(x[1 \mathrel{..} i-1])| \le d' \le \min(n-i, d)\}$.*

Lemma 5. *If a (d,n)-string x is not dense, then $\boldsymbol{s}(x) \leq \sigma_d^-(n)$.*

Proof. The proof follows from the basic observation that for any string x, $\boldsymbol{s}(x) \leq \boldsymbol{s}(x[1 \mathinner{..} i-1]) + \boldsymbol{s}(x[i+1 \mathinner{..} n]) + k_i(x)$ for any i. Note that the inequality occurs when there are the same type of squares in both $x[1 \mathinner{..} i-1]$ and $x[i+1 \mathinner{..} n]$. If x is not dense, then for some i_0, $k_{i_0}(x) \leq \sigma_d^-(n) - \boldsymbol{s}(x[1 \mathinner{..} i_0-1]) - m_{i_0}$. Then $\boldsymbol{s}(x) \leq \boldsymbol{s}(x[1 \mathinner{..} i_0-1]) + \boldsymbol{s}(x[i_0+1 \mathinner{..} n]) + k_{i_0}(x) \leq \boldsymbol{s}(x[1 \mathinner{..} i_0-1]) + m_{i_0} + k_{i_0}(x) \leq \boldsymbol{s}(x[1 \mathinner{..} i_0-1]) + m_{i_0} + \sigma_d^-(n) - \boldsymbol{s}(x[1 \mathinner{..} i_0-1]) - m_{i_0} = \sigma_d^-(n)$. □

Lemma 6. *If the core vector $k(x)$ of a (d,n)-string x satisfies $k_i(x) > 0$ for $i = 1,\ldots,n$, then x has an s-cover.*

Proof. We build an s-cover by induction: Since the $k_1(x) \geq 1$, 1 is in at least one core, hence there must be at least one square starting at position 1. Among all squares starting at position 1, set the one with the largest period to be S_1. Suppose that we have built the s-cover $\{S_i = (s_i, e_i, p_i) \ : \ i \leq t\}$. If $\bigcup_{1 \leq i \leq t} S_i = x$, we are done. Otherwise $\bigcup_{1 \leq i \leq t} S_i = x[1 \mathinner{..} e_t]$ where $e_t < n$. Since $k_{e_t+1}(x) \geq 1$, there is at least one square (s,e,p) in x so that $s \leq e_t + 1 \leq s + 2p - 1$. From all such squares choose the leftmost ones, and among them choose the one with the largest period and set it as S_{t+1}. It is straightforward to verify that all the conditions of Definition 1 are satisfied and that we have built the s-cover of x. □

Note that for a (d,n)-string, having an s-cover implies being singleton free. However it does not imply that every $k_i(x) \geq 1$, even though it is quite close to it. Consider the s-cover $\{S_j = (s_j, e_j, p_j) : 1 \leq j \leq m\}$ of x. If S_1 has another occurrence in x and there is no other square in x starting at position 1, then 1 is not in any core and $k_1(x) = 0$. Similarly, if the s-cover has two consecutive adjacent squares S_j and S_{j+1}, if there is no other square occurring at position s_{j+1}, and if the square S_{j+1} has some other occurrence, then $k_{s_{j+1}}(x) = 0$. In this sense, the s-cover is a computationally efficient structural generalization of the property that every $k_i(x) \geq 1$.

Lemma 7. *Let $\{S_i = (s_i, e_i, p_i) \mid 1 \leq i \leq m\}$ be an s-cover of x. Let $k = k(x)$ be the core vector of x. Then for any $1 \leq i < m$ and the core vector $k' = k(x[1 \mathinner{..} e_i])$, $(\forall\ 1 \leq j < s_{i+1})(k'_j \geq k_j)$.*

Proof. Let us assume that for some $i = 1, 2, \ldots m-1$ there is a j so that $k_j > k'_j$. Then there must exists a square (s,e,p) in $x = x[1 \mathinner{..} e_m]$ that is not a square of $x[1 \mathinner{..} e_i]$, i.e. $e > e_i$ and $s < s_{i+1}$, so it is an intermediate square violating the definition of s-cover, see Definition 1 (3). □

Lemma 8. *If a square-maximal (d,n)-string x has an s-cover with two consecutive adjacent squares, then $\sigma_d(n) \leq \sigma_{d_1}(n_1) + \sigma_{d_2}(n_2)$ for some $2 \leq d_1, d_2 \leq d \leq d_1 + d_2$ and some n_1, n_2, possibly equal to zero, such that $n_1 + n_2 = n$.*

Proof. Let $\{S_i \ : \ 1 \leq i \leq m\}$ be the s-cover of x and let $S_j \cap S_{j+1} = \emptyset$. Then $\boldsymbol{s}(x) \leq \boldsymbol{s}(x_1) + \boldsymbol{s}(x_2)$, where $x_1 = \bigcup\limits_{1 \leq i \leq j} S_i$ and $x_2 = \bigcup\limits_{j < i \leq m} S_i$. Therefore $\sigma_d(n) = \boldsymbol{s}(x) \leq \boldsymbol{s}(x_1) + \boldsymbol{s}(x_2) \leq \sigma_{d_1}(n_1) + \sigma_{d_2}(n_2)$ where x_1 and x_2 are, respectively, a (d_1, n_1)- and a (d_2, n_2)-string. □

Lemma 9. *If a singleton-free square-maximal (d,n)-string x does not have an s-cover, then $\sigma_d(n) = \sigma_d(n-1)$.*

Proof. Since x does not have an s-cover, there exist some i_0 such that $k_{i_0} = 0$ by Lemma 6. Remove $x[i_0]$ to form a $(d, n-1)$-string y. This will not decrease the number of distinct squares in x since there is no core of any square containing i_0. Then $\sigma_d(n) = \boldsymbol{s}(x) \le \boldsymbol{s}(y) \le \sigma_d(n-1)$. Since $\sigma_d(n) \ge \sigma_d(n-1)$ (see [4]), therefore $\sigma_d(n) = \sigma_d(n-1)$. □

Lemma 10. *If a square-maximal (d, n)-string has a singleton, then* $\sigma_d(n) = \sigma_{d-1}(n-1)$.

Proof. Remove the singleton to form a $(d-1, n-1)$-string y with $\sigma_d(n) = \boldsymbol{s}(x) \le \boldsymbol{s}(y) \le \sigma_{d-1}(n-1)$. Since $\sigma_d(n) \ge \sigma_{d-1}(n-1)$ (see [4]), therefore $\sigma_d(n) = \sigma_{d-1}(n-1)$. □

3 Heuristics for a lower bound $\sigma_d^-(n)$

Recall that $\sigma_d^-(n)$ denotes the best available lower bound for $\sigma_d(n)$. The higher the value of $\sigma_d^-(n)$, the less computational effort must be spent on determining $\sigma_d(n)$. For $d = 2$, generate $\mathcal{L}_2(n)$, the set of $(2, n)$-strings which admit an s-cover and are balanced over every prefix (the frequencies of a's and b's differ by at most a predefined constant), have a maximum period bounded by at most a predefined constant, and contain no triples (aaa or bbb). Then we set

$$\sigma_2^-(n) = \max\,\{\sigma_2(n-1),\ \max_{x \in \mathcal{L}_2(n)} \boldsymbol{s}(x)\}.$$

For $d \ge 3$, we simply set $\sigma_d^-(n) = \max\,\{\sigma_{d-1}(n-1),\ \ \sigma_{d-1}(n-2)+1,\ \ \sigma_d(n-1)\}$. The heuristic was found to be quite efficient in the fact that in almost all cases it gave the appropriate maximum value.

4 Generating special (d, n)-strings admitting an s-cover

Rather than generating strings, we generate their s-covers. By *special* we mean only s-covers that have no consecutive adjacent squares. The generation proceeds by extending the partially built s-cover in all possible ways. Every time a potential square of the s-cover is to be extended by one position, all previously used symbols and the first unused symbol are tried. For each symbol, the frequency counter is checked that the symbol does not exceed $n+2-2d$. Once a symbol is used, the frequency counter is updated. When the whole s-cover is generated, the counter is checked whether all d symbols occurred in the resulting string; if not, the string is rejected. A typical implementation of the generation of the s-cover would be through recursion as backtracking is needed. For computational efficiency reasons we opted instead for a user-stack controlled backtracking implemented as a co-routine `Next()` allowing us to call the co-routine repeatedly to produce the next string. Note that the strings are generated in a lexicographic order. The generation of the s-cover follows these principles: The generator for the first square is created by iterative calls to `Next()` producing all the possible generators. Each generator is checked for the additional properties (must be primitive, did not create an intermediate square in the partial string, etc.) before it is accepted. For each subsequent square, its generator may be partially or fully determined. If it is partially determined, iterative calls to `Next()` are used to generate all possible completions of the generator. The complete generator is checked and accepted or rejected. In addition, if the density of the string being generated is to be checked, we use Lemma 7 and the core vector of the partially generated string to reject the string or allow it to be extended further.

5 Recursive computation of $\sigma_d(n)$

First, $\sigma_d^-(n)$ is computed by the heuristic of Section 3. Then it is verified that $\sigma_{d_1}(n_1)+\sigma_{d_2}(n_2) \le \sigma_d^-(n)$ for any $2 \le d_1, d_2 \le d \le d_1 + d_2$ and any $n_1 + n_2 = n$. Then $\mathcal{U}_d(n)$, the set of all dense special (d, n)-strings admitting an s-cover is generated as described in Section 4. It follows that

$$\sigma_d(n) = \max\,\{\sigma_d^-(n),\ \max_{x\in\mathcal{U}_d(n)} \boldsymbol{s}(x)\}.$$

To see that, first consider the existence of a square-maximal (d, n)-string with singletons: by Lemma 10, $\sigma_d(n) = \sigma_{d-1}(n-1)$. Then consider the existence of a singleton-free square-maximal string x not in $\mathcal{U}_d(n)$:
(i) either x does not have an s-cover, in which case by Lemma 9, $\sigma_d(n) = \sigma_d(n-1)$;
(ii) or x has an s-cover with two consecutive adjacent squares and by Lemma 8, $\sigma_d(n) \le \sigma_{d_1}(n_1) + \sigma_{d_2}(n_2)$ for some $2 \le d_1, d_2 \le d$ and some $n_1 + n_2 = n$, and so $\sigma_d(n) \le \sigma_d^-(n)$;
(iii) or x has a special s-cover, but is not dense and by Lemma 5, $\sigma_d(n) \le \sigma_d^-(n)$.

6 Recursive computation of $\sigma_d(2d)$

To compute the values on the main diagonal we can use s-covers satisfying additional necessary parity condition. The s-cover $\{S_i = (s_i, e_i, p_i) : 1 \le i \le m\}$ of $x = x[1 \mathinner{..} n]$ satisfies the *parity condition* if for any $1 \le i < m$, $\mathcal{A}(x[1 \mathinner{..} e_i]) \cap \mathcal{A}(x[s_{i+1} \mathinner{..} n]) \subseteq \mathcal{A}(x[s_{i+1} \mathinner{..} e_i])$.

Lemma 11. *The singleton-free part of a square-maximal $(d, 2d)$-string x with all its singletons at the end has an s-cover satisfying the parity condition.*

Proof. We can assume that x has $0 \le v \le d-2$ singletons, all at the end. Let $k(x)$ be the core vector of x. Suppose the singleton-free part $x[1 \mathinner{..} 2d-v]$ does not have an s-cover, then there exist some $1 \le i_0 \le 2d-v$ such that $k_{i_0}(x) = 0$. Remove $x[i_0]$ to form a $(d, 2d-1)$-string y. Therefore, $\sigma_d(2d) = \boldsymbol{s}(x) \le \boldsymbol{s}(y) \le \sigma_d(2d-1) = \sigma_{d-1}(2d-2)$, a contradiction. So $x[1 \mathinner{..} 2d-v]$ has an s-cover $\{S_i : 1 \le i \le m\}$. Let us assume that the s-cover does not satisfy the parity condition. Then either
(i) $\bigcup_{1\le i\le t} S_i$ and $\bigcup_{t<i\le m} S_i$ for some $1 \le t \le m$ are adjacent and their respective alphabets have at least one symbol in common, say c. If we replace c in $\bigcup_{1\le i\le t} S_i$ by a new symbol $\hat{\boldsymbol{c}} \notin \mathcal{A}(x)$, we get a new $(d+1, 2d)$-string y so that $\boldsymbol{s}(y) \ge \boldsymbol{s}(x)$. Thus $\sigma_d(2d) = \boldsymbol{s}(x) \le \boldsymbol{s}(y) \le \sigma_{d+1}(2d) = \sigma_d(2d-1) = \sigma_{d-1}(2d-2)$, a contradiction, or
(ii) $\bigcup_{1\le i\le t} S_i$ and $\bigcup_{t<i\le m} S_i$ for some $1 \le t \le m$ are overlapping, and there is a symbol c occurring in $\bigcup_{1\le i\le t} S_i$ and in $\bigcup_{t<i\le m} S_i$, but not in the overlap $S_t \cap S_{t+1}$. If we replace c in $\bigcup_{1\le i\le t} S_i$ by a new symbol $\hat{\boldsymbol{c}} \notin \mathcal{A}(x)$, we get a new $(d+1, 2d)$-string y so that $\boldsymbol{s}(y) \ge \boldsymbol{s}(x)$. Thus $\sigma_d(2d) = \boldsymbol{s}(x) \le \boldsymbol{s}(y) \le \sigma_{d+1}(2d) = \sigma_d(2d-1) = \sigma_{d-1}(2d-2)$, a contradiction. □

With additional assumptions, Lemma 11 can be strengthen to exclude consecutive adjacent squares from the s-cover of a square-maximal $(d, 2d)$-string.

Lemma 12. *Let $\sigma_{d'}(2d') = d'$ for any $d' < d$. Either $\sigma_d(2d) = d$ or for every square-maximal $(d, 2d)$-string x with v singletons all at the end, $0 \leq v \leq d-2$, its singleton-free part $x[1 \mathinner{.\,.} 2d - v]$ has an s-cover satisfying the parity condition and which has no consecutive adjacent squares.*

Proof. The existence of the s-cover $\{S_i \mid 1 \leq i \leq m\}$ of $x[1 \mathinner{.\,.} 2d - v]$ satisfying the parity condition follow from Lemma 11. We need to prove that either $\sigma_d(2d) = d$ or there are no adjacent squares in the s-cover. Since $\sigma_{d'}(2d') = d'$ for any $d' < d$, $\sigma_{d'}(n') \leq n' - d'$ for any $n' - d' < d$. Let us assume that the s-cover of x has two adjacent squares S_t and S_{t+1}. Let $x_1 = \bigcup_{1 \leq i \leq t} S_i$ and let $x_2 = \bigcup_{t<i \leq m} S_i$. Then $\boldsymbol{s}(x) \leq \boldsymbol{s}(x_1) + \boldsymbol{s}(x_2)$ where x_1 and x_2 are, respectively, a (d_1, n_1)- and a (d_2, n_2)-string with $n_1 + n_2 = 2d - v$ and $d_1 + d_2 \geq d - v$. Since the s-cover satisfies the parity condition, $\mathcal{A}(x_1)$ and $\mathcal{A}(x_2)$ are disjoint and hence $d_1 + d_2 = d - v$. Therefore $(n_1 - d_1) + (n_2 - d_2) = d$. Since both x_1 and x_2 are singleton-free, $n_1 - d_1 > 0$ and $n_2 - d_2 > 0$. Hence $n_1 - d_1 < d$ and $n_2 - d_2 < d$, and therefore $\sigma_d(2d) = \boldsymbol{s}(x) \leq \boldsymbol{s}(x_1) + \boldsymbol{s}(x_2) \leq \sigma_{d_1}(n_1) + \sigma_{d_2}(n_2) \leq (n_1 - d_1) + (n_2 - d_2) = d$. □

Since the number of distinct squares in a singleton-free $(d, 2d)$-string is at most d, we do not need to consider singleton-free strings. Moving a singleton to the end of a string does not decrease the number of distinct squares, therefore we shall only consider $(d, 2d)$-strings that have singletons at the end. We can set $\sigma_d^-(2d) = \sigma_{d-1}(2d-2) + 1$ and thus consider only the strings that have the non-singleton part dense. By Lemma 12 we need only to consider strings whose s-covers of the non-singleton part satisfy the parity condition with no consecutive adjacent squares. Moreover, the number of singletons must be at least $\lceil \frac{2d}{3} \rceil$, see [4]. Let $\mathcal{T}_v$ denote the set of all singleton-free $\sigma_d^-(2d)$-dense $(d - \lceil \frac{2d}{3} \rceil, 2d - \lceil \frac{2d}{3} \rceil)$-strings admitting an s-cover satisfying the parity condition with no consecutive adjacent squares. Then we set

$$\sigma_d(2d) = \max \{d, \max_{x \in \mathcal{T}_v} \boldsymbol{s}(x)\}.$$

7 Additional properties of $\sigma_d(n)$

Though fundamental properties of $\sigma_d(n)$ were presented previously in [4], here we present some additional properties concerning the gaps between consecutive values in the $(d, n-d)$ table where the value of $\sigma_d(n)$ is the entry on the d-th row and the $(n-d)$-th column. Lemma 13, respectively Lemma 14, shows that the difference between any two consecutive entries along a row, respectively between any two consecutive entries on the main diagonal, in the $(d, n-d)$ table is bounded by 2.

Lemma 13. *For any $2 \leq d \leq n$, $\sigma_d(n+1) - \sigma_d(n) \leq 2$.*

Proof. Let $(d, n+1)$-string $x = x[1 \mathinner{.\,.} n+1]$ be square-maximal, then $\boldsymbol{s}(x) = \sigma_d(n+1)$. Without a loss of generality we can assume that the first symbol of x is not a singleton – otherwise we can move all singletons from the beginning of x to the end of x without destroying any square type. Let $y = x[2 \mathinner{.\,.} n+1]$. Then y is a (d, n)-string and $\boldsymbol{s}(y) \leq \sigma_d(n)$. By Fraenkel-Simpson [5], there are at most two rightmost occurrences of squares starting at the same position in a string. In other words, the removal of $x[1]$ destroyed at most two square types. That is, $\boldsymbol{s}(x) - 2 \leq \boldsymbol{s}(y)$. Therefore, $\sigma_d(n+1) - 2 \leq \boldsymbol{s}(y) \leq \sigma_d(n)$, implying $\sigma_d(n+1) - \sigma_d(n) \leq 2$. □

Lemma 14. *For any* $2 \leq d$, $\sigma_{d+1}(2d+2) - \sigma_d(2d) \leq 2$.

Proof. By Lemma 13, $\sigma_{d+1}(2d+2) - \sigma_{d+1}(2d+1) \leq 2$. By the results from [4], the entries under and on the main diagonal along a column are constant; that is, $\sigma_{d+1}(2d+1) = \sigma_d(2d)$. Therefore, $\sigma_{d+1}(2d+2) - \sigma_d(2d) \leq 2$. □

Lemma 15. *For any* $d \geq 2$*, if there is a square-maximal singleton-free* $(d, 2d+1)$*-string* x*, then there exists a square-maximal* $(d, 2d+1)$*-string* y *of the form* $y = aaabbccdd\ldots$

Proof. Since x contains no singletons, then x contains exactly $d-1$ pairs and 1 triple. To prove there exists a square-maximal string in the form that all pairs consist of adjacent symbols and the triple also consists of adjacent symbols, we need to show the non-adjacent symbols can be moved together without reducing the number of distinct squares. Let us suppose that there is a non-adjacent pair of $\boldsymbol{c}$'s in x.

(i) If the $\boldsymbol{c}$'s did not occur in any square, then we could move both $\boldsymbol{c}$'s to the end of the string without destroying any square type. Moreover, we would gain a new square $\boldsymbol{cc}$, contradicting the square-maximality of x.
(ii) If the $\boldsymbol{c}$'s occur in exactly one square $u\boldsymbol{c}vu\boldsymbol{c}v$, where u and v are some strings, we can move both $\boldsymbol{c}$'s to the end of x to form a new string y. The new squares created by this move are $uvuv$ and $\boldsymbol{cc}$ while the old square $u\boldsymbol{c}vu\boldsymbol{c}v$ was destroyed. If $uvuv$ did not exist in any other part of x, then $\boldsymbol{s}(y) > \boldsymbol{s}(x)$ which contradicts the square-maximality of x; thus $uvuv$ already existed in some other part of x, so we lost the square $u\boldsymbol{c}vu\boldsymbol{c}v$, but gained $\boldsymbol{cc}$, so $\boldsymbol{s}(y) = \boldsymbol{s}(x)$.
(iii) If the $\boldsymbol{c}$'s occur in more than one square, these squares must form a non-trivial run, i.e. a run with a non-empty tail. Since there is only one symbol $\boldsymbol{t}$ occurring in x 3 times, the only form of such a non-trivial run can be $\boldsymbol{t}u\boldsymbol{c}v\boldsymbol{t}u\boldsymbol{c}v\boldsymbol{t}$. If $u = v = \varepsilon$, then the run is $\boldsymbol{tctct}$ containing two distinct squares $\boldsymbol{tctc}$ and $\boldsymbol{ctct}$. We can change it to $\boldsymbol{tttcc}$, destroying the two squares $\boldsymbol{tctc}$ and $\boldsymbol{ctct}$, but gaining two new squares $\boldsymbol{tt}$ and $\boldsymbol{cc}$. If either $u \neq \varepsilon$ or $v \neq \varepsilon$, then by moving both $\boldsymbol{c}$'s to the end of x, we destroy the two distinct squares $\boldsymbol{t}u\boldsymbol{c}v\boldsymbol{t}u\boldsymbol{c}v$ and $u\boldsymbol{c}v\boldsymbol{t}u\boldsymbol{c}v\boldsymbol{t}$, but gain three new squares $\boldsymbol{t}uv\boldsymbol{t}uv$, $uv\boldsymbol{t}uv\boldsymbol{t}$, and $\boldsymbol{cc}$. Note that neither $\boldsymbol{t}uv\boldsymbol{t}uv$ nor $uv\boldsymbol{t}uv\boldsymbol{t}$ can exist anywhere else in x for the lack of $\boldsymbol{t}$'s. Thus we have more distinct squares than x, which contradicts the maximality of x.
Since we can move safely all pairs together to the end of x, the symbols of the triple will end up also adjacent at the beginning of the string. □

Lemma 16 shows that the two entries of the $(d, n-d)$ table in the same column just above the main diagonal must be identical.

Lemma 16. *For any* $3 \leq d$, $\sigma_d(2d+1) = \sigma_{d-1}(2d)$.

Proof. We prove it by induction. Let (H_d) be the statement that $\sigma_d(2d+1) = \sigma_{d-1}(2d)$. (H_d) for $2 \leq d \leq 10$ is true from the values in the $(d, n-d)$ table computed so far, see [4]. This takes care of the base case of the induction. Thus let us assume that H_1 through H_{d-1} are true, and let us prove that (H_d) is true. Let $(d, 2d+1)$-string x be square-maximal. If x contains a singleton, remove it to form a new $(d-1, 2d)$-string y. Then $\sigma_d(2d+1) = \boldsymbol{s}(x) \leq \boldsymbol{s}(y) \leq \sigma_{d-1}(2d)$ and since $\sigma_d(2d+1) \geq \sigma_{d-1}(2d)$, see [4], thus $\sigma_d(2d+1) = \sigma_{d-1}(2d)$. If x contains no singletons, by Lemma 15 we can assume that it has the form $aaabbccdd\ldots$ Remove a pair from z forming a new $(d-1, 2d-1)$-string y. Then $\sigma_d(2d+1) - 1 = \boldsymbol{s}(x) - 1 = \boldsymbol{s}(y) \le \sigma_{d-1}(2d-1)$ and since $\sigma_d(2d+1) - 1 \ge \sigma_{d-1}(2d-1)$ by [4], therefore $\sigma_d(2d+1) = \sigma_{d-1}(2d-1) + 1$. Since H_{d-1}, $\sigma_{d-1}(2d) \ge \sigma_{d-2}(2d-2) + 1$ and $\sigma_d(2d+1) \ge \sigma_{d-1}(2d)$ according to [4], hence $\sigma_d(2d+1) = \sigma_{d-1}(2d)$. □

Corollary 17 demonstrates the fact that the difference between any two consecutive entries on the two diagonals immediately above the main diagonal of the $(d, n-d)$ table is also bounded by 2.

Corollary 17. *For any* $3 \le d$, $\sigma_d(2d+1) - \sigma_{d-1}(2d-1) \le 2$ *and* $\sigma_d(2d+2) - \sigma_{d-1}(2d) \le 2$.

Proof. By Lemma 13, $\sigma_{d-1}(2d) - \sigma_{d-1}(2d-1) \le 2$, and by Lemma 16, $\sigma_{d-1}(2d) = \sigma_d(2d+1)$. Therefore, $\sigma_d(2d+1) - \sigma_{d-1}(2d-1) \le 2$. Similarly, $\sigma_d(2d+2) - \sigma_d(2d+1) \le 2$ by Lemma 13, and $\sigma_d(2d+1) = \sigma_{d-1}(2d)$ by Lemma 16. Therefore, $\sigma_d(2d+2) - \sigma_{d-1}(2d) \le 2$. □

Remark 18. Fraenkel-Simpson [5] gave the upper bound of $2n-8$ for $n \ge 5$ and any d, and $\sigma_2(n) \le 2n - 29$ for $n \ge 22$. Ilie [8] provided an asymptomatic bound of $2n - \Theta(\log n)$. We slightly improve Fraenkel-Simpson's bounds with: for any $2 \le d \le n$ and $n \ge d_0 + 2$, $\sigma_d(n) \le 2n - d_0 - 2d$, where d_0 is the maximum d such that $\sigma_d(2d) = d$ is known. Currently, $d_0 = 23$. In addition, since $\sigma_2(53) = 40$ we get $\sigma_2(n) \le 2n - 66$ for $n \ge 53$. Similarly, since $\sigma_3(42) = 31$, $\sigma_4(32) = 22$, $\sigma_5(33) = 23$, $\sigma_6(28) = 17$, $\sigma_7(30) = 18$, $\sigma_8(25) = 14$, $\sigma_9(23) = 12$ and $\sigma_{10}(23) = 11$, we get $\sigma_3(n) \le 2n - 53$ for $n \ge 42$, $\sigma_4(n) \le 2n - 42$ for $n \ge 32$, $\sigma_5(n) \le 2n - 43$ for $n \ge 33$, $\sigma_6(n) \le 2n - 39$ for $n \ge 28$, $\sigma_7(n) \le 2n - 42$ for $n \ge 30$, $\sigma_8(n) \le 2n - 36$ for $n \ge 25$, $\sigma_9(n) \le 2n - 34$ for $n \ge 23$ and $\sigma_{10}(n) \le 2n - 35$ for $n \ge 23$.

Proof. By Lemma 14, $\sigma_d(n) \le d_0 + 2k$, where $n - d = d_0 + k$ and $k \ge 1$. Thus $\sigma_d(d_0 + k + d) \le d_0 + 2k = 2(d_0 + k + d) - d_0 - 2d$. Therefore, $\sigma_d(n) \le 2n - d_0 - 2d$ for $n \ge d_0 + 2$.

8 Computational Results

We implemented the described algorithms in C++, and ran the programs in parallel on the SHARCNET computer cluster. We were able to compute all $\sigma_2(n)$ values for $n \le 53$ in a matter of hours. The 10 largest new values are: $\sigma_2(44) = 33$, $\sigma_2(45) = 34$, $\sigma_2(46) = 35$, $\sigma_2(47) = 36$, $\sigma_2(48) = 36$, $\sigma_2(49) = 37$, $\sigma_2(50) = 37$, $\sigma_2(51) = 38$, $\sigma_2(52) = 39$ and $\sigma_2(53) = 40$. The results and sample square-maximal strings may be found at [3]. Whenever the computation required determining the number of distinct primitively rooted squares in a concrete string, a C++ implementation of the Franek, Jiang, and Weng's algorithm [7] was used. The values of interest include: three consecutive equal values: $\sigma_2(31) = \sigma_2(32) = \sigma_2(33)$, the unexpected existence of pairs (d, n) satisfying $\sigma_{d+1}(n+2) - \sigma_d(n) > 1$ such as (2,33) and (2,34), and $\sigma_2(33) < \sigma_3(33)$; that is, among all strings of length 33, no binary string achieves the maximum number of distinct primitively rooted squares.

9 Conclusion

We presented the notion of s-cover as a structural generalization of a uniform distribution of squares in a string. We showed that it is sufficient to consider special strings admitting an s-cover in order to recursively determine the maximum number of distinct primitively rooted squares $\sigma_d(n)$. Based on these observations, we presented an efficient computational framework with significantly reduced search space for computations of $\sigma_d(n)$ based on the notion of density and exploiting the tightness of the available lower bound. We used an implementation of this algorithm to obtain the previously unknown values of $\sigma_d(n)$, and in particular $\sigma_2(n)$ up to $n = 53$.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada and by the Canada Research Chair program, and made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (`http://www.sharcnet.ca/`).

References

1. A. Baker, A. Deza, and F. Franek: *Computational framework for determining run-maximal strings*, AdvOL-Report 2011/06, McMaster University, 2011.
2. A. Baker, A. Deza, and F. Franek: *On the structure of run-maximal strings.* Journal of Discrete Algorithms, 14 2012, pp. 10–14.
3. A. Deza, F. Franek, and M. Jiang: *Square-maximal strings*, `http://optlab.mcmaster.ca/~jiangm5/research/square.html`.
4. A. Deza, F. Franek, and M. Jiang: *A d-step approach for distinct squares in strings.* Lecture Notes in Computer Science, 5029 2011, pp. 77–89.
5. A. S. Fraenkel and J. Simpson: *How many squares can a string contain?* Journal of Combinatorial Theory Series A, 82 1998, pp. 112–120.
6. F. Franek and J. Holub: *A different proof of Crochemore-Ilie lemma concerning microruns*, in London Algorithmics 2008: Theory and Practice, College Publications, London, UK, 2009, pp. 1–9.
7. F. Franek, M. Jiang, and C. Weng: *An improved version of the runs algorithm based on Crochemore's partitioning algorithm*, in Proceedings of Prague Stringology Conference 2011, Prague, Czech Republic, 2011, pp. 98–105.
8. L. Ilie: *A note on the number of squares in a word.* Theoretical Computer Science, 380 2007, pp. 373–376.

Author Index