Bořivoj Melichar

August 30, 2012

Festschrift for

Bořivoj Melichar

August 30, 2012

Edited by Jan Holub, Bruce W. Watson and Jan Žďárek

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

Preface

This diverse collection of papers — some deeply scientific, some less so — are in honour of Professor Bořivoj Melichar's seventieth birthday. The plan for a festschrift (a volume in tribute to a distinguish academic, often upon their retirement or other milestone) was hatched by the editors in 2012 has his birthday neared. By a happy coincidence, his birthday often falls during the Prague Stringology Conference, and this year's conference is dedicated to him.

This volume, with chapters ranging in topic from applied arbology (XML) to combinatorial properties of strings, is very appropriate for someone whose entire career has been deeply interwoven with the field of stringology. Bořivoj has personally made significant research advances in stringology and arbology. While that alone would qualify him as an outstanding scientist and engineer, Bořivoj has also served as a mentor to many of us, sometimes figuratively, but often as Master's, Ph.D or Habilitation supervisor and promotor. He has always had a talent for connecting seemingly unrelated results throughout our field, and that is reflected in the breadth of contributions here.

Happy birthday Bořivoj — we look forward to many more years of friendship and research,

The editors and contributors

In Eindhoven, Netherlands, and Prague, Czech Republic on August 2012

Jan Holub, Bruce W. Watson and Jan Žďárek

Table of Contents

Invited Contributions

Sergio De Agostino: *Bounded Memory LZW Compression and Distributed Computing: A Worst-Case Analysis* 1

Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda: *Finding Absent Words from Grammar Compressed Strings* 10

Manolis Christodoulakis and Michalis Christou: *Abelian Concepts in Strings: a Review* 19

Loek Cleophas and Bruce W. Watson: *On Factor Storacles: an Alternative to Factor Oracles?* 46

Maxime Crochemore, Tomasz Kociumaka, Wojciech Rytter, Chalita Toopsuwan, Wojciech Tyczyński, and Tomasz Waleń : *Algorithmics of Repetitions, Local Periods and Critical Factorization Revisited* 53

Domenico Cantone, Simone Faro, and Emanuele Giaquinta: *Fast Algorithms for Online Searching on Burrows-Wheeler Transformed Texts* 61

Simone Faro and Thierry Lecroq: *Twenty Years of Bit-Parallelism in String Matching* 72

Andrew Baker, Antoine Deza, and Frantisek Franek: *A Parameterized Formulation for the Maximum Number of Runs Problem* 102

Jan Holub: *Man of Four Research Topics (... so far)* 118

Jan Janousek: *On My Friendship with Bob Melichar* 123

Shmuel Tomi Klein: *A Rabin Karp Hash for Approximate Automata* 125

Derrick G. Kourie, Bruce W. Watson, Fritz Venter, and Loek Cleophas: *Formal Concept Analysis Applications in Stringology* 127

Boba Mannová: *Brtkovica – the Foundation of Arbology* 140

Tomáš Flouri, Kassian Kobert, Solon P. Pissis, and Alexandros Stamatakis : *A simple method for computing all subtree repeats in unordered trees in linear time* 145

Joshua Amavi, Béatrice Bouchou-Markhoff, and Agata Savary: *XMLCorrector: an Open Source Tool for XML Document Correction* 153

Emanuele Giaquinta and Esko Ukkonen: *Motif Matching Using Gapped q-gram Patterns* 161

Jan Žďárek: *Bořivoj Melichar and Multidimensional Pattern Matching* 168

Author Index 171

Bounded Memory LZW Compression and Distributed Computing: A Worst-Case Analysis

Sergio De Agostino

Computer Science Department, Sapienza University, 00198 Rome, Italy

Abstract. Sheinwald, Lempel and Ziv proved that the power of off-line coding is not useful if we want on-line decodable files, as far as asymptotic results are concerned [16]. In the finite case, we proved the NP-completeness of the optimal on-line decodable LZW compression problem and that a sublogarithmic factor approximation algorithm cannot be realized on-line [12]. Moreover, we showed that the on-line greedy LZW compression procedure is an $O(n^{\frac{1}{4}})$ approximation algorithm, where n is the size of the input file [10]. In this paper, we consider algorithms using bounded work space. In this context, the on-line greedy LZW compression procedure is an $O(\sqrt{d})$ approximation algorithm, where d is the dictionary size. Based on this result, we propose a more robust approach to LZW compression on a distributed system which is more suitable for highly disseminated data than the one in [9].

Keywords: compression, factorization, approximation, distributed system

1 Introduction

Ziv-Lempel compression [14] is based on string factorization. With the factorization process explained in [19], each factor is the extension by one character of the longest match with one of the previous factors. Ziv-Lempel compression is a dictionary-based technique. In fact, the factors of the string are substituted by *pointers* to copies stored in a dictionary. Given an input string S over an alphabet A, the on-line greedy factorization defined by Ziv and Lempel is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the shortest substring which is different from one of the previous factors. The encoding of each factor leaves one character uncompressed. To avoid this a different factorization was introduced (on-line greedy LZW factorization) where each factor f_i is the longest match with the concatenation of a previous factor and the next character [18]. f_i is encoded by a pointer q_i to such concatenation (LZW compression). A real time implementation is possible by storing the dictionary in a trie data structure.

The parallel complexity of LZW compression has been studied extensively [3]. The LZW factorization process is hard to parallelize [2], even if bounded memory versions are considered [11]. On the other hand, parallel decompression is possible [4]. This follows from the fact that the computational hardness is in the factorization rather than in the coding. We wish to point out that the decoding problem is interesting independently from the computational efficiency of the encoder. In fact, in the case of compressed files stored in a read-only memory only the computational efficiency of decompression is relevant.

Ziv and Lempel investigated in [19] the encoding power of a finite state one-way head machine with an unrestricted decoder and showed for each individual sequence an asymptotically attainable lower bound on the achievable compression ratio. Furthermore, they achieved this lower bound with an encoder/decoder pair where both

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

are one-way head machines (the algorithm described above), proving that imposing this condition only on the encoder does not lead to better compression. Sheinwald, Lempel and Ziv [16] showed that the same asymptotically attainable lower bound holds when the encoder is unrestricted and the decoder is a finite state one-way head machine, proving that imposing this condition only on the decoder does not lead to better compression either. In conclusion, the power of off-line coding is not useful if we want on-line decodable files, as far as asymptotic results are concerned.

De Agostino and Storer [12] compared on-line and off-line coding in the finite case, which models situations such as distribution of data on CD-ROM where much time can be spent encoding but decoding must be fast and simple. The pointer encoding the factor f_i has a size increasing with the index i. This means that the lower is the number of factors for a string of a given length the better is the compression. 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 positive. In [12] the notion of *on-line decodable* optimal LZW compression was introduced, proving that optimal on-line decodable LZW factorization is NP-complete and that a sublogarithmic factor approximation algorithm cannot be realized on-line. Moreover, the logarithmic lower bound to the approximation multiplicative factor was not proved to be tight. We remark that this does not contradict the asymptotic results mentioned above. In fact, only when the asymptotically attainable lower bound is zero the approximation factor might not converge to 1.

An LZW factorization $S = f_1 \cdots f_k$ is *feasible* if each factor f_i is equal to $f_j c$, where c is the first character of f_{j+1} and $j < i$. We define *optimal* LZW factorization the feasible LZW factorization with the snallest number of factors. The on-line decoder paired with the standard on-line greedy LZW encoder is the decoder for any encoder using any other feasible LZW factorization. The optimal LZW compressor is the one using an optimal LZW factorization. De Agostino and Silvestri showed that the standard on-line greedy LZW compressor is an $O(n^{\frac{1}{4}})$ approximation of the optimal one, where n is the size of the input file [10].

The factorization processes described 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. Bounded memory LZW compression employs several deletion heuristic to remove dictionary elements [17]. The most effective heuristic is the least recently used strategy and a relaxed version has been proved to be the most efficient in [5]. They were both proved to be hard to parallelize in [11]. In this paper, we introduce similar notions of on-line decodable optimal coding for the bounded memory versions of LZW compression. Bounded memory versions of LZW compression suitable for a distributed system have been realized in [6], [7], [8], [9]. We show in this paper that these versions are an $O(d)$ approximation of the optimal one, where d is the dictionary size. Based on this result, we propose a more robust approach to LZW compression on a distributed system which is an $O(\sqrt{d})$ approximation of the optimal one and it is more suitable for highly disseminated data than the one in [9].

In section 2, we describe standard bounded memory LZW compression and present the on-line decodable optimal versions. In section 3, we make the worst case analysis of the greedy approaches. In section 4, we discuss the implementations on a distributed system. Conclusions and future work are given in section 5.

2 Bounded Memory LZW Compression

Let $d + \alpha$ be the cardinality of the fixed size dictionary where α is the cardinality of the alphabet. The simplest deletion heuristic is FREEZE, which has 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 on-line greedy *d-frozen* LZW factorization of a string S 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. A *feasible* d-frozen LZW factorization $S = f_1 \cdots f_k$ is a feasible LZW factorization where the number of different concatenations of a factor with the next character is $\leq d$. We define *optimal* d-frozen LZW factorization the feasible d-frozen LZW factorization with the snallest number of factors. Computing the optimal solution in polynomial time is quite straightforward if the degree of the polynomial time function is the dictionary size but it is obviously unpractical and a better algorithm is not known.

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 starst deteriorating, restarts the factorization process. In other words, the on-line greedy *d-restarted* LZW factorization $S = f_1 f_2 \cdots f_j \cdots f_i \cdots f_k$ is such that if j is 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). A *feasible* d-restarted LZW factorization $S = f_1 \cdots f_j \cdots f_i \cdots f_k$ is a feasible LZW factorization such that if j and i are consecutive indices where the restart operation happens the number of different concatenations of a factor with the next character is $\leq d$ between f_j and f_i. We define *optimal* d-restarted LZW factorization the feasible d-restarted LZW factorization with the snallest number of factors. A practical algorithm to compute the optimal solution is obviously not known as for the optimal d-frozen LZW factorization.

3 Approximation and Greedy Algorithms

The compression models introduced in the previous section employ bounded size dictionaries. During the learning process before freezing and eventually restarting the dictionary, the on-line greedy factorization is the only feasible factorization producing factors which are all different from each other, that is, the number of factors equals the number of dictionary elements. This is the property we use to prove our result. In this section, we give upper bounds to the approximation multiplicative factor. A trivial upper bound to the approximation multiplicative factor of the on-line greedy factorization with respect to the optimal one is the maximum factor length of the optimal string factorization, that is, the height of the trie storing the dictionary. Such upper bound is $\Theta(d)$, where d is the dictionary size ($O(d)$ follows from the feasibility of the factorization and $\Omega(d)$ from the factorization of the unary string). There are strings for which the on-line greedy d-frozen LZW factorization is a $\Theta(d)$ approximation of the optimal one. In fact, if we bound the dictionary size to $d + 2$ and consider the input binary string $(\prod_{i=0}^{d/2-1} ab^iba^i)(\prod_{i=1}^{d} a^{d/2})$ then the on-line greedy *d-frozen* LZW factorization is:

$$a, b, ab, ba, abb, baa, \cdots ab^i, ba^i, \cdots ab^{d/2-1}, ba^{d/2-1}, a, a, \cdots, a$$

while the optimal *d-frozen* LZW factorization is:

$$a, b, ab, b, a, abb, b, aa, \cdots ab^i, b, a^i, \cdots ab^{d/2-1}, b, a^{d/2-1}, a^{d/2}, a^{d/2}, \cdots a^{d/2}$$

It follows that the cost of the greedy factorization is $d + d^2/2$ while the cost of the optimal one is $5d/2 - 1$. We prove now the following theorem:

Theorem. The on-line greedy d-restarted LZW factorization is an $O(\sqrt{d})$ approximation of the optimal one, where d is the dictionary size.

Proof. Without loss of generality, we can assume the restart operation happens as soon as the dictionary is filled up during the greedy factorization process since the static phase monitors the performance of the procedure. Let S be a string of length n and T be the trie storing the dictionary of factors of the optimal d-restarted LZW factorization Φ of S between two consecutive positions where the restart operation happens. Each dictionary element (but the alphabet characters) corresponds to the concatenation of a factor f of the optimal factorization with the first character of the next factor, that we call an *occurrence* of the dictionary element (node of the trie) in Φ. We call an element of the dictionary built by the greedy process *internal* if its occurrence is contained in the occurrence of a node of T and denote with M_T the number of internal occurences. The number of non-internal occurences is less than the number of factors of Φ. Therefore, we can consider only the internal ones. An occurrence f' of the greedy factorization internal to an factor f of Φ is represented by a subpath of the path representing f in T. Let u be the endpoint at the lower level in T of this subpath (which, obviously, represents a prefix of f). Let $d(u)$ be the number of subpaths representing internal phrases with endpoint u and let $c(u)$ be the total sum of their lengths. All the occurences of the greedy factorization are different from each other between two consecutive positions where the restart operation of the greedy procedure happens. Since two subpaths with the same endpoint and equal length represent the same factor, we have $c(u) \geq d(u)(d(u)+1)/2$. Therefore

$$1/2 \sum_{u \in T} d(u)(d(u)+1) \leq \sum_{u \in T} c(u) \leq 2n \leq 2|\Pi| H_T$$

where H_T is the height of T, $|\Phi|$ is the number of phrases of Φ and the multiplicative factor 2 is due to the fact that occurrences of dictionary elements may overlap. We denote with $|T|$ the number of nodes in T; since $M_T = \sum_{u \in T} d(u)$, we have

$$M_T^2 \leq |T| \sum_{u \in T} d(u)^2 \leq |T| \sum_{u \in T} d(u)(d(u)+1) \leq 4|T||\Phi| H_T$$

where the first inequality follows from the fact that the arithmetic mean is less than the quadratic mean. Then

$$M_T \leq \sqrt{4|T||\Phi| H_T} = |\Phi| \sqrt{\frac{4|T| H_T}{|\Pi|}} \leq 2|\Pi| \sqrt{H_T}$$

The statement of the theorem follows from the fact that the height of the trie is $\Theta(d)$ in the worst case. q. e. d.

4 Distributed Computing

We describe previous work on implementations of LZW compression on a distributed system and then provide a worst case analysis which suggests a more robust approach.

4.1 Previous Work

LZW compression is applied in parallel to input data blocks of even length. For each block, the dictionary is learned by applying on-line greedy LZW compression to the first half of it. Then, the second half is compressed in a static way using the dictionary learned by processing the first half (basicly, the restart operation is determined in advance). After we fill up the dictionary on the first half of the block, the greedy factorization we compute with such dictionary on the second half is not optimal since the dictionary is prefix but not suffix (a dictionary is prefix (suffix) if all the prefixes (suffixes) of a dictionary element are dictionary elements). However, there is an optimal semi-greedy factorization which is computed by the procedure of figure 1 [1], [13]. 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. The algorithm can even be implemented in real time with a modified suffix tree data structure [13], [15].

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.

To speed up the static phase, implementations on an extended star network (a rooted tree of height 2) were realized. An $O(km)$ time, $O(n/km)$ processors algorithm is guaranteed to produce a factorization with a cost approximating the cost of the optimal factorization of the second half of the block within the multiplicative factor $(k + 1)/k$ with k positive integer and m maximum factor length [8]. This algorithm provides an approximation scheme since the multiplicative approximation factor converges to 1 when km converges to L where $2L$ is the block length.

During the input phase, the central node broadcasts a block to each adjacent processor. Then, for each block the corresponding processor broadcasts to the adjacent leaves a sub-block of length $m(k + 2)$ of the second half, 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 first half 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.

We call a *boundary match* a factor covering positions of two adjacent sub-blocks. The approximation algorithm is the following:

– for each block, every processor but the one associated with the last sub-block computes the boundary match between its sub-block and the next one which ends furthest to the right;
– each processor computes the optimal factorization from the beginning of the boundary match on the left boundary of its sub-block to the beginning of the boundary match on the right boundary.

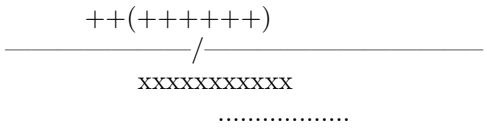

Figure 2. The making of a surplus factor.

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 multiplicative approximation factor $(k + 1)/k$ with respect to any factorization. In fact, as it is shown in figure 2, 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). Since it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorizarion is about the same as the sequential one, the algorithm is scalable. This is true even if the second phase 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 [13] 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 factor length is 10, one processor can compress down to 100 bytes independently. 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 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

Each compression procedure decribed in this subsection produces a feasible d-restarted LZW factorization. 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 approaches described, 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.

4.2 A Worst Case Analysis

If any of the procedures described in the previous subsection is applied to the input block of length d^2

$$b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)(\prod_{i=1}^{d} a^{d/2})$$

the dictionary is filled up by the greedy factorization process applied to the first half of the block, that is, $b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)$. Such factorization is

$$b, bb, \cdots, b^\ell, b^{\ell'}, a, b, ab, ba, abb, baa, \cdots ab^i, ba^i, \cdots ab^{d/2-1}, ba^{d/2-1}$$

where $\ell' \leq \ell + 1$ and the dictionary size is $d + \ell + 3$. The static factorization of the second half is $a, a, \cdots a, a$ and the total cost of the factorization of the block

is $\ell + 1 + d + d^2/2$ which is $\Theta(d^2)$. Similarly to the previous section, the cost of the optimal solution on the block is $\ell + 5d/2$ which is $\Theta(d)$. Observe that the $O(d)$ approximation multiplicative factor depends on the static phase and this happens when the dictionary learned on the first half of the block performs badly on the second half, that is in practice, when the data are highly disseminated.

4.3 A More Robust Approach

A different approach, which is more robust and in some cases (when the data are quite homegeneous) a little less effective in terms of comppression, restarts the dicionary as soon as it is filled up. Therefore, on a distributed system each processor stores a block of data and applies the on-line greedy LZW factorization adding a new element to the dictionary at each step. Obviously, blocks are short enough to observe the dictionary size bound d. From the the statement of the theorem in the previous section, such approach outputs an $O(\sqrt{d})$ approximation of the optimal solution since it computes the on-line greedy d-restarted factorization.

5 Conclusion

In this paper, we presented an approach to LZW compression on a distributed system with a better worst case analysis than the one previously proposed in literature. The feasible d-restarted LZW factorizations computable on a distributed system that have been previously designed work with data blocks 600 K long in practice. The on-line greedy LZW compression process is applied to the first half of it to learn the dictionary and a second static phase using the dictionary learned is implementable on an extended star network in a scalable way (this second phase is executed at the leaf level of the network and some interprocessor communication is required between the leaves and their parents during the first phase). The approach presented here instead applies the standard LZW compression algorithm to block 300 K long with the advantage of no interprocessor communication involved during the computational phase and a better running time. Scaling up the system is possible only on very large files but from this point of view it is still competitive since only the second phase of the previous approaches has no scalability issues. On the other hand, the previous approaches have a better compression effectiveness when data are quite homegeneous. The approach presented here is more suitable when data are highly disseminated.

References

1. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, 2003.
2. S. DeAgostino: *P-complete problems in data compression.* Theoretical Computer Science, 127 1994, pp. 181–186.
3. S. DeAgostino: *Parallelism and dictionary-based data compression.* Information Sciences, 135 2001, pp. 43–56.
4. S. DeAgostino: *Almost work-optimal pram erew decoders of lz-compressed text.* Parallel Processing Letters, 14 2004, pp. 351–359.
5. S. DeAgostino: *Bounded size dictionary compression: Relaxing the lru deletion heuristic*, in Proceedings Prague Stringology Conference, 2005, pp. 135–142.
6. S. DeAgostino: *Lempel-ziv data compression on parallel and distributed systems*, in Proceedings Data Compression, Communications and Processing Conference, 2011, pp. 193–202.
7. S. DeAgostino: *Lempel-ziv data compression on parallel and distributed systems.* Algorithms, 4 2011, pp. 183–199.

8. S. DeAgostino: *Lzw versus sliding window compression on a distributed system: Robustness and communication*, in Proceedings INFOCOMP, 2011, pp. 125–130.
9. S. DeAgostino: *Lzw data compression on large scale and extreme distributed systems*, in Proceedings Prague Stringology Conference, 2012.
10. S. DeAgostino and R. Silvestri: *A worst case analisys of the lz2 compression algorithm.* Information and Computation, 139 1997, pp. 258–268.
11. S. DeAgostino and R. Silvestri: *Bounded size dictionary compression:* SC^k*-completeness and* nc *algorithms.* Information and Computation, 180 2003, pp. 101–112.
12. S. DeAgostino and J. A. Storer: *On-line versus off-line computation for dynamic text compression.* Information Processing Letters, 59 1996, pp. 169–174.
13. A. Hartman and M. Rodeh: *Optimal parsing of strings*, 1985.
14. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 22 1976, pp. 75–81.
15. E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of ACM, 23 1976, pp. 262–272.
16. D. Sheinwald, A. Lempel, and J. Ziv: *On coding and decoding with two-way head machines.* Information and Computation, 116 1995, pp. 128–133.
17. J. A. Storer: *Data Compression: Methods and Theory*, Computer science Press, 1988.
18. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
19. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.

Finding Absent Words from Grammar Compressed Strings

Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, Japan
1SC09020E@s.kyushu-u.ac.jp, {inenaga,bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. A string Z is said to be an absent word of another string S if Z is not a substring of S. We present an efficient algorithm to find a shortest absent word of a string given in a compressed form, namely, as a straight line program (SLP). Our algorithm runs in $O(n \log_\sigma N)$ time and space, where N is the length of the uncompressed string, n is the size of the SLP, and σ is the alphabet size, and we show how to further improve the complexity of the algorithm. We also present efficient algorithms to find minimal absent words of bounded length from a string given as an SLP.

Keywords: absent words, forbidden words, missing patterns, straight line programs (SLPs), compressed string processing

1 Introduction

An *absent word* (a.k.a. *absent sequence*, *forbidden word*, *missing pattern*) of a given string S is a string which is not a substring of S. An absent word Z of S is said to be minimal if any proper substring of Z is a substring of S. Absent words of a string have applications in bioinformatics [13,14,18,24,9,1,5], and in text compression [7,20], and therefore, efficient algorithms to compute absent words from a given string are of great importance. It is known that a shortest absent word of a string of length N over an integer alphabet can be computed in $O(N)$ time and space [1]. Also, all minimal absent words of a string of length N can be computed in $O(N\sigma)$ time [6], where σ is the alphabet size.

In this paper, we investigate the problem of finding absent words of a string given in a compressed form. We present an efficient algorithm to find a shortest absent word of a string given as a straight line program (SLP). Our algorithm runs in $O(n \log_\sigma N)$ time and space, where N is the length of the uncompressed string and n is the size of the SLP. We then show how to further improve the complexity of the algorithm. We also present efficient algorithms to find minimal absent words of bounded length from a string given as an SLP.

Since the length N of the decompressed string can be as large as $O(2^n)$, decompressing the whole string is not permissive. All of our algorithms are based on *partial decompression* of a given SLP, and use string data structures such as suffix trees [23] and directed acyclic word graphs (DAWGs) [2,3] constructed on the partially decompressed strings. When the size n of a given SLP is small w.r.t. the uncompressed string length N, our algorithms are more efficient than algorithms working of the uncompressed string. Recent study [15] shows that a collection of DNA sequences of the same or similar species is highly compressive by a variant of the LZ77 algorithm [25]. Since the LZ77 factorization of a string can be efficiently converted to a corresponding SLP [19], our algorithms should be suitable for such data.

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

2 Preliminaries

2.1 Strings

Let Σ be a finite *alphabet* and $\sigma = |\Sigma|$. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $S = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of S, respectively. String P is said to be a proper substring (resp., prefix, suffix) of another string S if P is a substring (resp., prefix, suffix) of S and $|P| < |S|$.

The i-th character of a string S is denoted by $S[i]$ for $1 \leq i \leq |S|$, and the substring of a string S that begins at position i and ends at position j is denoted by $S[i : j]$ for $1 \leq i \leq j \leq |S|$. For convenience, let $S[i : j] = \varepsilon$ if $j < i$. For a string S and integer $q \geq 0$, let $pre(S, q)$ and $suf(S, q)$ represent respectively, the length-q prefix and suffix of T, that is, $pre(S, q) = S[1 : \min\{q, |S|\}]$ and $suf(S, q) = S[\max\{1, |S|-q+1\} : |S|]$. We also assume that the last character of the string is a special character '\$' that does not occur anywhere else in the string.

Our model of computation is the word RAM: We shall assume that the computer word size is at least $\lceil \log_2 |S| \rceil$, and hence, standard operations on values representing lengths and positions of string S can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

2.2 Absent Words

If a string P is *not* a substring of another string S, then P is said to be an *absent word* of S. An absent word P of S is said to be a *minimal absent word* of S if any proper substring of P is a substring of S.

2.3 Straight Line Programs

A *straight line program* (*SLP*) is a set of assignments $\mathcal{T} = \{X_1 \rightarrow expr_1, X_2 \rightarrow expr_2, \ldots, X_n \rightarrow expr_n\}$, where each X_i is a distinct non-terminal variable and each $expr_i$ is an expression that can be either $expr_i = a$ ($a \in \Sigma$), or $expr_i = X_{\ell(i)}X_{r(i)}$ ($i > \ell(i), r(i)$). An SLP is essentially a context free grammar in the Chomsky normal form, that derives a single string. Let $val(X_i)$ represent the string derived from variable X_i. To ease notation, we sometimes associate $val(X_i)$ with X_i and denote $|val(X_i)|$ as $|X_i|$. An SLP $\mathcal{T}$ *represents* the string $T = val(X_n)$. The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$.

The derivation tree of SLP $\mathcal{T}$ is a labeled ordered binary tree where each internal node is labeled with a non-terminal variable in $\{X_1, \ldots, X_n\}$, and each leaf is labeled with a terminal character in Σ. The root node has label X_n. Let $\mathcal{V}$ denote the set of internal nodes in the derivation tree. For any internal node $v \in \mathcal{V}$, let $\langle v \rangle$ denote the index of its label $X_{\langle v \rangle}$. Node v has a single child which is a leaf labeled with c when $(X_{\langle v \rangle} \rightarrow c) \in \mathcal{T}$ for some $c \in \Sigma$, or v has a left-child and right-child respectively denoted $\ell(v)$ and $r(v)$, when $(X_{\langle v \rangle} \rightarrow X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$. Each node v of the tree derives $val(X_{\langle v \rangle})$, a substring of T, whose corresponding interval $itv(v) = [b : e]$, with $T[b : e] = val(X_{\langle v \rangle})$, can be defined recursively as follows. If v is the root node, then $itv(v) = [1 : |T|]$. Otherwise, if $(X_{\langle v \rangle} \rightarrow X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$, then, $itv(\ell(v)) = [b_v : b_v + |X_{\langle \ell(v) \rangle}| - 1]$ and $itv(r(v)) = [b_v + |X_{\langle \ell(v) \rangle}| : e_v]$, where $[b_v : e_v] = itv(v)$. Let $vOcc(X_i)$ denote the number of times a variable X_i occurs in the derivation tree, i.e.,

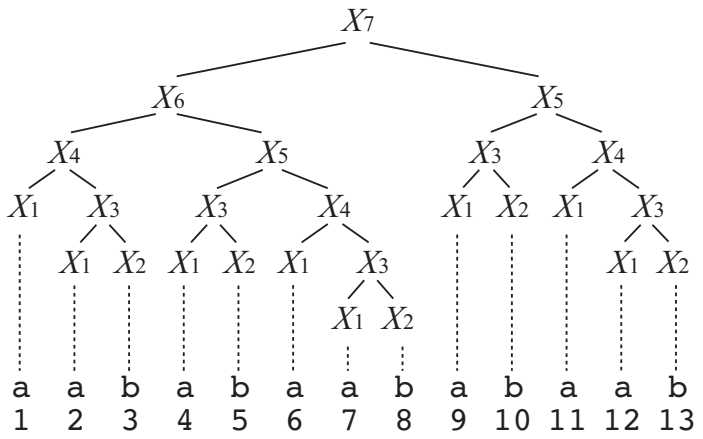

Figure 1. The derivation tree of SLP $\mathcal{T} = \{X_1 \to \mathtt{a},\ X_2 \to \mathtt{b},\ X_3 \to X_1X_2,\ X_4 \to X_1X_3,\ X_5 \to X_3X_4,\ X_6 \to X_4X_5,\ X_7 \to X_6X_5\}$. $T = val(X_7) = \mathtt{aababaababaab}$.

$vOcc(X_i) = |\{v \mid X_{\langle v \rangle} = X_i\}|$. We assume that any variable X_i is used at least once, that is $vOcc(X_i) > 0$.

For any interval $[b : e]$ of $T(1 \le b < e \le |T|)$, let $\xi_{\mathcal{T}}(b, e)$ denote the deepest node v in the derivation tree, which derives an interval containing $[b : e]$, that is, $itv(v) \supseteq [b : e]$, and no proper descendant of v satisfies this condition. We say that node v *stabs* interval $[b : e]$, and $X_{\langle v \rangle}$ is called the variable that stabs the interval. We have $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle} X_{\langle r(v) \rangle}) \in \mathcal{T}$, $b \in itv(\ell(v))$, and $e \in itv(r(v))$. When it is not confusing, we will sometimes use $\xi_{\mathcal{T}}(b, e)$ to denote the variable $X_{\langle \xi_{\mathcal{T}}(b,e) \rangle}$.

SLPs can be efficiently pre-processed to hold various information. $|X_i|$ and $vOcc(X_i)$ can be computed for all variables X_i $(1 \le i \le n)$ in a total of $O(n)$ time by a simple dynamic programming algorithm.

2.4 Suffix Trees

We give the definition of a very important and well known string index structure, the suffix tree. To assure property 3 for the sake of presentation, we assume that the string ends with a unique symbol that does not occur elsewhere in the string.

Definition 1 (Suffix Trees [23]). *The suffix tree of a string S, denoted $STree(S)$, is a labeled rooted tree which satisfies the following:*

1. *each edge is labeled with an element in Σ^+;*
2. *there exist exactly n leaves, where $n = |S|$;*
3. *for each string $s \in Suffix(S)$, there is a unique path from the root to a leaf which spells out s;*
4. *each internal node has at least two children;*
5. *the labels x and y of any two distinct out-going edges from the same node begin with different symbols in Σ*

Since any substring of S is a prefix of some suffix of S, positions in the suffix tree of S correspond to a substring of S that is represented by the string spelled out on the path from the root to the position. We can also define a *generalized* suffix tree of a set of strings, which is simply the suffix tree that contains all suffixes of all the strings in the set.

It is well known that suffix trees can be represented and constructed in linear time [23,17,22], even independently of the alphabet size for integer alphabets [8]. Generalized suffix trees for a set of strings $\mathbf{S} = \{S_1, \ldots, S_k\}$, can be constructed in linear time in the total length of the strings, by simply constructing the suffix tree of the string $S_1\$_1 \cdots S_k\$_k$, and pruning the tree below the first occurrence of any $\$_i$, where $\$_i$ $(1 \leq i \leq k)$ are unique characters that do not occur elsewhere in strings of $\mathbf{S}$.

2.5 Directed Acyclic Word Graphs

For a set $\mathbf{S} = \{S_1, \ldots, S_k\}$ of strings and string Y, let $EndPos_{\mathbf{S}}(Y)$ denote the set of ending positions of Y in strings of $\mathbf{S}$, i.e.

$$EndPos_{\mathbf{S}}(Y) = \{(i, j) \mid S_i[j - |Y| + 1 : j] = Y, S_i \in \mathbf{S}\}.$$

If Y is not a substring of any string of $\mathbf{S}$, then $EndPos_{\mathbf{S}}(Y) = \emptyset$. Define an equivalence relation $\equiv_{\mathbf{S}}$ of strings over alphabet Σ w.r.t. $\mathbf{S}$ by

$$Y \equiv_{\mathbf{S}} Z \iff EndPos_{\mathbf{S}}(Y) = EndPos_{\mathbf{S}}(Z).$$

The equivalence class of string Y w.r.t. $\equiv_{\mathbf{S}}$ is denoted by $[Y]_{\mathbf{S}}$.

Definition 2 (Directed Acyclic Word Graphs (DAWGs) [2,3]). *The DAWG of a set* $\mathbf{S}$ *of strings, denoted* $DAWG(\mathbf{S})$*, is an edge-labeled dag* (V, E) *such that*

$$V = \{[Y]_{\mathbf{S}} \mid EndPos_{\mathbf{S}}(Y) \neq \emptyset\},$$
$$E = \{([Y]_{\mathbf{S}}, a, [Ya]_{\mathbf{S}}) \mid EndPos_{\mathbf{S}}(Y) \neq \emptyset, EndPos_{\mathbf{S}}(Ya) \neq \emptyset, a \in \Sigma\}.$$

Theorem 3 ([3]). *For any set* $\mathbf{S}$ *of strings,* $DAWG(\mathbf{S})$ *can be constructed in* $O(M \log \sigma)$ *time and* $O(M)$ *space, where* M *is the total length of strings in* $\mathbf{S}$*.*

The suffix link of a node $[aY]_{\mathbf{S}}$ of $DAWG(\mathbf{S})$ is a reversed edge $([aY]_{\mathbf{S}}, [Y]_{\mathbf{S}})$, where $a \in \Sigma$, $Y \in \Sigma^*$, and $aY \not\equiv_{\mathbf{S}} Y$. The suffix links of all nodes of $DAWG(\mathbf{S})$ are computed and used during construction of $DAWG(\mathbf{S})$.

When the set $\mathbf{S}$ is a singleton, i.e., $\mathbf{S} = \{T\}$ for some $T \in \Sigma^*$, then we denote the DAWG of $\mathbf{S}$ by $DAWG(T)$.

3 Finding Shortest Absent Word from SLP

Consider any string $S \in \Sigma^*$ of length N. If $\sigma > N$, then clearly there is a single character a which is a shortest absent word of S, which can be easily found in $O(\sigma)$ time. If $\sigma = N$, then there exists $a \in \Sigma$ for which aa is a shortest absent word of S. If $\sigma < N$, then the following lemma is useful to find a shortest absent word of S:

Lemma 4 ([1]). *Let* S *be a string over* Σ*. If* $\sigma^k \geq N$ *for some integer* $k > 1$*, then there is an absent word of* S *of length* k*.*

We use techniques of [10,11] that partially decompress the given SLP, such that all substrings of specified length q occur in the partially decompressed strings. For any positive integer $q \geq 2$ and each variable $X_i \to X_{\ell(i)}X_{r(i)}$, any substring of length q that is stabbed by X_i is a substring of

$$t_i(q) = suf(val(X_{\ell(i)}), q - 1)pre(val(X_{r(i)}), q - 1).$$

On the other hand, all substrings of length q are stabbed by some variable. This means that if we consider the set of strings consisting of $t_i(q)$ for all variables such that $|X_i| \geq q$, any substring of length q of S is a substring of at least one of the strings. We can compute the set $\mathbf{T}_S(q) = \{t_i(q) \mid |X_i| \geq q\}$ of all such strings in time linear in the total length, i.e. in $O(nq)$ time by a straightforward dynamic programming [10].

It immediately follows from Lemma 4 that the length of a shortest absent word of S is at most $\lceil \log_\sigma N \rceil$. Let $c_N = \lceil \log_\sigma N \rceil$. Using the method described above, we compute the set $\mathbf{T}_S(c_N)$ of strings in $O(nc_N)$ time. We then extend the notion of a shortest absent word from a single string to a set of strings. For a set $\mathbf{S}$ of strings, a string P is said to be a shortest absent word of $\mathbf{S}$ if P is an absent word of every string in $\mathbf{S}$ and it is the shortest one of such strings.

Lemma 5. *A string P is a shortest absent word of string S if and only if P is a shortest absent word of $\mathbf{T}_S(c_N)$.*

Proof. ($\Rightarrow$) We prove it by contradiction. Assume Q is a shortest absent word of $\mathbf{T}_S(c_N)$ with $|Q| < |P|$. All substrings of S of length at most c_N occur as substrings of strings in $\mathbf{T}_S(c_N)$. This means that Q is an absent word of S, but this contradicts that P is a shortest absent word of S.

($\Leftarrow$) We prove it by contradiction. Assume Q is a shortest absent word of S with $|Q| < |P|$. Then clearly Q is an absent word of every string in $\mathbf{T}_S(c_N)$. This contradicts that P is a shortest absent word of $\mathbf{T}_S(c_N)$. □

By Lemma 5, the problem of finding a shortest absent word of S reduces to the problem of finding a shortest absent word of $\mathbf{T}_S(c_N)$. Our solution to this problem is an extension of the algorithm of [1], but uses the generalized suffix tree for $\mathbf{T}_S(c_N)$, rather than a suffix tree of a single string.

Theorem 6. *We can compute a shortest absent word of $\mathbf{T}_S(c_N)$ in $O(nc_N)$ time and space.*

Proof. We first build $STree(\mathbf{T}_S(c_N))$ in $O(nc_N)$ time and space, and then find a shortest, possibly implicit, node which has less than σ children. This node can be found in $O(nc_N)$ time by traversing $STree(\mathbf{T}_S(c_N))$. Let y be this node and Y be the substring represented by y, and let b be a character such that y has no out-going edge whose label begins with b. Then clearly Yb is a shortest absent word of $\mathbf{T}_S(c_N)$. The character b can be easily computed in $O(\sigma)$ time by using an array A of length σ, as follows: Initialize all the entries of A with 0, and for each character a going out from node y, we set $A[a] = 1$. By scanning A we find a character b with $A[b] = 0$, for which Yb is an absent word of $\mathbf{T}_S(c_N)$. □

Theorem 7. *Given an SLP of size n representing a string S of length N, we can compute a shortest absent word of S in $O(n \log_\sigma N)$ time, where σ is the alphabet size.*

Proof. By Lemmas 4, 5 and Theorem 6. □

When the given SLP is small (i.e., $n = o(N/\log_\sigma N)$), our algorithm is faster than optimal $O(N)$-time algorithms working on the uncompressed string S.

It is known that the LZ78 factorization [26] of size m can be converted to a corresponding SLP of size $O(m)$ in $O(m)$ time. Since $m = O(N/\log_\sigma N)$ [26], if the given SLP is based on the LZ78 factorization, then our algorithm is as efficient

as $O(N)$-time algorithms working on S even in the worst case, and is faster when $m = o(N/\log_\sigma N)$. Note that the size m of the LZ78 factorization can be as small as $O(\sqrt{N})$.

3.1 Faster Algorithms

Doubling Search. We can obtain an improved bound upon Theorem 7 by employing a simple doubling search on the length of partial decompression.

Corollary 8. *Given an SLP of size n representing a string S of length N, we can compute a shortest absent word of S in $O(n\ell)$ time and space, where ℓ is the length of a shortest absent word of S.*

Proof. Instead of using c_N for the length of partial decompression, we start from length 2. If all internal nodes of the generalized suffix tree has σ children for length 2^{i-1}, then we rebuild the generalized suffix tree for length 2^i. We stop increasing the value of i when we find the smallest i such that the generalized suffix tree has a (possibly implicit) node which has less than σ children. For each i construction of the generalized suffix tree takes $O(n2^i)$ time, and the total asymptotic complexity becomes $n(2 + \cdots + 2^{\lceil \log_2 \ell \rceil}) = O(n\ell)$. □

Reducing Partial Decompression. The algorithm of Theorem 7 becomes slower than an $O(N)$ algorithm working on uncompressed string when $n \log_\sigma N > N$. To overcome this, we can use the techniques of [11], which enable us to reduce the partial decompression conducted on the SLP. For any integer $1 \leq q \leq N$, let $I(q) = \{i \mid |X_i| \geq q\} \subseteq [1 : n]$. The technique exploits the overlapping portions of each of the strings in $\mathbf{T}_S(q)$. The algorithm of [11] shows how to construct, in time linear of its size, a trie of size $(q-1) + \sum_{i \in I(q)} (|t_i(q)| - (q-1)) = N - \alpha(q) = N_{\alpha(q)}$ such that there is a one-to-one correspondence between a path of length q in the trie and a substring of length q of a string in $\mathbf{T}_S(q)$. Here,

$$\alpha(q) = \sum_{i \in I(q)} ((vOcc(X_i) - 1) \cdot (|t_i(q)| - (q-1))) \geq 0 \tag{1}$$

can be seen as a quantity which depends on the amount of redundancy that the SLP captures with respect to substrings of length q.

Furthermore, a suffix tree of a trie can be constructed in linear time:

Lemma 9 ([21]). *Given a trie of size M over alphabet Σ with $\sigma = O(M)$, the suffix tree for the trie can be constructed in $O(M)$ time and space.*

The generalized suffix tree for $\mathbf{T}_S(c_N)$ used in our algorithm can be replaced with the suffix tree of the trie, and we can reduce the $O(n \log_\sigma N)$ term in the complexity to $O(N_{\alpha(c_N)})$, thus obtaining an $O(N_{\alpha(c_N)})$ time and $O(N_{\alpha(c_N)})$ space algorithm. Since $N_{\alpha(c_N)}$ is also bounded by $O(n \log_\sigma N)$, we obtain the following result:

Theorem 10. *Given an SLP of size n representing a string S of length N, we can compute a shortest absent word of S in $O(N_{\alpha(c_N)})$ time and space, where $N_{\alpha(c_N)} = O(\min\{N - \alpha(c_N), n \log_\sigma N\})$, and $\alpha(c_N) \geq 0$ is defined as in Equation (1).*

The above theorem is significant since the running time of the improved algorithm is at least as efficient as any algorithm working on the uncompressed string *independently of the size n of a given SLP*, and can be much faster when n is small w.r.t. the uncompressed size N.

4 Finding Minimal Absent Word of Bounded Length from SLP

In this section, we present two algorithms to find all minimal absent words of bounded length, from a given SLP.

4.1 DAWG Based Algorithm

The first algorithm uses the DAWG for partially decompressed strings from a given SLP:

Theorem 11. *Given an SLP of size n representing a string S, and a positive integer k, we can compute all minimal absent words of S of length at most k in $O(nk\sigma)$ time and $O(nk)$ space.*

Proof. Our algorithm is based on the algorithm of Crochemore et al. [6] that finds all minimal absent words from an uncompressed string in time linear in the string. The algorithm of Crochemore et al. constructs the DAWG for the input string T, namely $DAWG(T)$. Checking whether a single character $a \in \Sigma$ is a minimal absent word or not is trivial and can be done in $O(\sigma)$ time. Consider minimal absent words of length at least two. Let aY be the shortest member of an equivalence class $[aY]_T$ with $a \in \Sigma$ and $Y \in \Sigma^*$. Assume that aYb is an absent word of T. By definition, the following statement holds:

$$aYb \text{ is a minimal absent word of } T \iff aY \text{ and } Xb \text{ are substrings of } T.$$

For all characters $b \in \Sigma$, we can check whether or not aYb is an absent word in a total of $O(\sigma)$ time; if there is no out-going edge of $[aY]_T$ labeled with b, then aYb is an absent word. Now that aY is clearly a substring of T, what remains is to check whether or not Yb is a substring of T. This can be done efficiently by using the suffix link of node $[aY]_T$ that leads to node $[Y]_T$. Namely, if there is an out-going edge of $[Y]_T$ labeled with b, then Yb is a substring, and therefore aYb is a minimal absent word. Since there are $O(|T|)$ nodes in $DAWG(T)$, the total time complexity is $O(|T|\sigma)$.

To compute all minimal absent words of length at most k from a given SLP of size n describing string S, we partially decompress each variable X_i and obtain string $t_i(k)$ of length at most $2(k-1)$ that is stabbed by X_i. For the set $\mathbf{T}_S(k) = \{t_i(k) \mid 1 \leq i \leq n\}$ of those strings, we construct $DAWG(\mathbf{T}_S(k))$. Let $[aY]_{\mathbf{T}_S(k)}$ be any node of $DAWG(\mathbf{T}_S(k))$ with $a \in \Sigma$ and $Y \in \Sigma^*$, where aY is the shortest member of $[aY]_{\mathbf{T}_S(k)}$. We only need to consider the nodes with $|aY| < k$ since we are now only interested in minimal absent words of length at most k. $DAWG(\mathbf{T}_S(k))$ contains all substrings of S of length at most k, and the total length of strings in $\mathbf{T}_S(k)$ is $O(nk)$. It follows from Theorem 3 that the number of nodes of $DAWG(\mathbf{T}_S(k))$ is $O(nk)$ and it can be constructed in $O(nk \log \sigma)$ time. Hence, by using the above method by Crochemore at al. we can find all minimal absent words of S in $O(nk\sigma)$ time and $O(nk)$ space. □

If we regard k as a constant, then the above algorithm runs in $O(n\sigma)$ time and $O(n)$ space. This improves on the existing algorithms [6,5] which find all minimal absent words of bounded length from a given string S of length N in $O(N\sigma)$ time and $O(N)$ space.

4.2 Suffix Tree Based Algorithm

The second approach is based on suffix trees and maximal repeats of strings. For convenience, assume that the first and last characters of any string S are # and \$ which do not appear elsewhere in S. A string Z is said to be a maximal repeat of string S if for some $p_1 \neq p_2$, $Z = S[p_1 : p_1 + |Z| - 1] = S[p_2 : p_2 + |Z| - 1]$, $S[p_1 - 1] \neq S[p_2 - 1]$, and $S[p_1 + |Z|] \neq S[p_2 + |Z|]$.

The following lemma describes a relationship between minimal absent words and maximal repeats of a string.

Lemma 12 ([18]). *If aYb is a minimal absent word of string S, then Y is a maximal repeat of S.*

Given a set $\mathbf{S} = \{S_1, \ldots, S_k\}$ of strings, a string Z is said to be a maximal repeat of the set $\mathbf{S}$ if Z is a maximal repeat of the concatenated string $S_1 \cdots S_k$. Assuming that each string S_j terminates with unique character $\$_j$, any string that crosses the borders cannot be a maximal repeat of $\mathbf{S}$. For a string S, since $\mathbf{T}_S(k)$ contains all substrings of S of length at most k, the following lemma holds.

Lemma 13. *A string Y of length at most k is a maximal repeat of S, iff Y is a maximal repeat of $\mathbf{T}_S(k)$.*

Theorem 14. *Given an SLP of size n representing a string S, and a positive integer k, we can compute all minimal absent words of S of length at most k in $O(N_{\alpha(k)}\sigma^2)$ time and $O(N_{\alpha(k)}\sigma)$ space, where $N_{\alpha(k)} = O(\min\{N - \alpha(k), nk\})$, and $\alpha(k) \geq 0$ is defined as in Equation (1).*

Proof. We construct the suffix tree of a trie for set $\mathbf{T}_S(k)$ of strings using the techniques of [11]. We use the algorithm of [4] which constructs the suffix tree of the trie in $O(N_{\alpha(k)}\sigma)$ time and space. The algorithm of [4] is based on Weiner's suffix tree construction algorithm for a single string [23], where for each character $a \in \Sigma$ the function $f(a, Z)$ is stored in each node Z of the suffix tree, such that $f(a, Z)$ is true iff aZ is represented by the suffix tree. We will use this function in the sequel.

By applying the method of [12] to find all maximal repeats from a suffix tree of a string to our suffix tree for the trie, we can find all maximal repeats of S of length at most k in $O(N_{\alpha(k)})$ time and space. We then use Lemma 12: Given a maximal repeat Y of S, we can find minimal absent words of S by checking whether aY, Yb, and aYb occur in S for each pair $(a, b) \in \Sigma \times \Sigma$ of characters. Since Y is a maximal repeat, there is always an explicit node y of the suffix tree that represents Y. Hence, given a character $a \in \Sigma$, we can check if aY is represented by the suffix tree using function $f(a, y)$ in constant time. Using an array of size σ as was done in Theorem 6, we can compute all maximal absent words of length at most k in a total of $O(N_{\alpha(k)}\sigma^2)$ time and $O(N_{\alpha(k)}\sigma)$ space. □

5 Conclusions and Open Problems

In this paper we proposed efficient algorithms to compute shortest absent words and minimal absent words of bounded length from a string given as an SLP.

Interesting open problems are:

- Can we efficiently compute all minimal absent words from an SLP?
- Can we find a shortest pair of strings that do not occur within a specified distance, from an SLP? Several algorithms for finding such a pair from uncompressed strings are known [1,16].

References

1. S. Angelov, S. Inenaga, T. Kivioja, and V. Mäkinen: *Missing pattern discovery.* J. Discrete Algorithms, 9(2) 2011, pp. 153–165.
2. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theoret. Comput. Sci., 40 1985, pp. 31–55.
3. A. Blumer, J. Blumer, D. Haussler, R. Mcconnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis.* Journal of the ACM, 34(3) 1987, pp. 578–595.
4. D. Breslauer: *The suffix tree of a tree and minimizing sequential transducers.* Theoretical Computer Science, 191(1–2) 1998, pp. 131–144.
5. S. Chairungsee and M. Crochemore: *Using minimal absent words to build phylogeny.* Theoret. Comput. Sci., 450 2012, pp. 109–116.
6. M. Crochemore, F. Mignosi, and A. Restivo: *Minimal forbidden words and factor automata*, in Proc. MFCS 1998, vol. 1450 of Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 665–673.
7. M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi: *Text compression using antidictionaries*, Tech. Rep. IGM-98-10, Institut Gaspard-Monge, 1998.
8. M. Farach: *Optimal suffix tree construction with large alphabets*, in Proc. FOCS 1997, 1997, pp. 137–143.
9. S. P. Garcia, A. J. Pinho, J. M. Rodrigues, C. A. C. Bastos, and P. J. Ferreira: *Minimal absent words in prokaryotic and eukaryotic genomes.* PLoS ONE, 6(1) 2011, Article number e16065.
10. K. Goto, H. Bannai, S. Inenaga, and M. Takeda: *Fast q-gram mining on SLP compressed strings*, in Proc. SPIRE 2011, 2011, pp. 289–289.
11. K. Goto, H. Bannai, S. Inenaga, and M. Takeda: *Speeding up q-gram mining on grammar-based compressed texts*, in Proc. CPM 2012, 2012, pp. 220–231.
12. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
13. G. Hampikian and T. Andersen: *Absent sequences: Nullomers and primes*, in Pacific Symposium on Biocomputing, vol. 12, 2007, pp. 355–366.
14. J. Herold, S. Kurtz, and R. Giegerich: *Efficient computation of absent words in genomic sequences.* BMC Bioinformatics, 9 2008, Article number 167.
15. S. Kuruppu, S. J. Puglisi, and J. Zobel: *Reference sequence construction for relative compression of genomes*, in Proc. SPIRE, 2011, pp. 420–425.
16. S. C. Li: *Faster algorithms for finding missing patterns*, in Proc. CATS 2006, vol. 51 of CRPIT, 2006, pp. 107–111.
17. E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of ACM, 23(2) 1976, pp. 262–272.
18. A. J. Pinho, P. J. Ferreira, S. P. Garcia, and J. M. Rodrigues: *On finding minimal absent words.* BMC Bioinformatics, 10 2009, Article number 137.
19. W. Rytter: *Application of Lempel-Ziv factorization to the approximation of grammar-based compression.* Theoret. Comput. Sci., 302(1–3) 2003, pp. 211–222.
20. Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa: *Pattern matching in text compressed by using antidictionaries*, in Proc. 10th Ann. Symp. on Combinatorial Pattern Matching, vol. 1645 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 37–49.
21. T. Shibuya: *Constructing the suffix tree of a tree with a large alphabet.* IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E86-A(5) 2003, pp. 1061–1066.
22. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
23. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, Institute of Electrical Electronics Engineers, New York, 1973, pp. 1–11.
24. Z.-D. Wu, T. Jiang, and W.-J. Su: *Efficient computation of shortest absent words in a genomic sequence.* Information Processing Letters, 110(14-15) 2010, pp. 596–601.
25. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–349.
26. J. Ziv and A. Lempel: *Compression of individual sequences via variable-length coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.

Abelian Concepts in Strings: a Review

Manolis Christodoulakis[1] and Michalis Christou[2]

[1] Department of Electrical and Computer Engineering, University of Cyprus
P.O. Box 20537, 1687 Nicosia, Cyprus, manolisc@ucy.ac.cy
[2] Department of Informatics, King's College London
Strand, London WC2R 2LS, UK, michalis.christou@kcl.ac.uk

Abstract. In this paper, we survey the most important literature related to abelian concepts in strings, since its introduction in 1961. We investigate the problem of abelian pattern matching and its variants (with or without preprocessing, approximate, and others), the appearance of abelian regularities, such as periods and powers, the avoidance of abelian powers and regularities, and finally the abelian complexity in infinite strings. We go over various algorithms for their computation and reveal the combinatorics behind them. Connections with applications in several fields, such as bioinformatics, are also explored. Finally, we summarize some open problems and suggest future research directions.

Keywords: abelian, strings, Parikh vectors, pattern matching, periods, powers, pattern avoidance, complexity

1 Introduction

Research around abelian concepts in strings has been initiated by a problem given by Erdős in 1961 [52], that asks to construct an infinite string on an alphabet as small as possible that avoids abelian squares (adjacent strings that are permutations of each other). This problem is the abelian analogue of the classic problem considered by Thue in the beginning of the 19$^{\text{th}}$ century [87], that asks to construct an infinite string on an alphabet as small as possible that avoids squares (adjacent strings that are identical).

Parikh [79] presented a mapping, called the *Parikh mapping* after his name, that associates a string with the multiplicities of each of its letters. In this sense, two strings are equivalent (*abelian equivalent*) if they contain the same set of letters, regardless of the positions of the letters in each string. Parikh mappings have appeared in literature by various different names, such as Parikh vectors [4], compomers [17], permutation patterns [53], jumbled patterns [33] and others.

In recent years, a number of applications of the abelian concepts of strings on real-life scenarios has lead to an increased interest on the topic. For instance, in biology, Benson [10] suggested that abelian alignment provides better insight on the similarity of biologically related sequences, in comparison to classical alignment algorithms, and applied abelian alignment algorithms on human promoter sequences from the Eukaryotic Promoter Database. Eres et al. [53] and later Parida [78] performed abelian pattern discovery in order to find clusters of genes that frequently appear together on a set of *E. Coli* sequences. Böcker [17] used Parikh vectors for *Single Nucleotide Polymorphism* (SNP) discovery.

One of the first problems to appear in literature was that of avoiding abelian squares in infinite strings [52]. Since then, research on this field has expanded extensively, in aspects including: identification and enumeration of all abelian powers

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

in a string; avoidance of abelian powers in infinite strings; abelian periods and borders; abelian pattern matching and pattern discovery; combinatorial and complexity properties of abelian notions on strings.

In this paper we survey the most important research publications in the recent (and not so recent) literature on abelian concepts in strings. We begin with some definitions of strings, abelian notions on strings and some special sequences that are used frequently in the literature, in Section 2. In Section 3, we explore algorithms for several variants of abelian pattern matching, including pattern matching with and without preprocessing, approximate pattern matching, and others. In Section 4, we examine the notions of abelian periods and borders, their properties and algorithms for their computation in a string. Sections 5 and 6 go through the existence of abelian powers and the lack of them in finite and infinite strings. In Section 7, we describe combinatorial properties of the abelian theory on infinite strings. Finally, Section 8 describes some areas of research where the abelian notions have found applications. In each section, we list a number of related open problems for future research.

2 Definitions

We define an *alphabet* Σ as a finite, non-empty set of symbols, $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$. An ordering can be defined on the letters of Σ via a bijection $\phi : \Sigma \to \{1, 2, \ldots, \sigma\}$.

A *string* (or *word*) $x = x[1\,..\,n]$ is a finite sequence of symbols drawn from Σ. The size of x is denoted by $|x| = n$. The set of all finite strings over Σ is denoted by Σ^*. The empty string is denoted by ε. An *infinite string* $x = x[1]x[2]\ldots$ is an infinite sequence of symbols from Σ. The set of all infinite strings over Σ is denoted by $\Sigma^{\mathbb{N}}$. A *partial* word over Σ is a finite sequence over the augmented alphabet $\Sigma_\diamond = \Sigma \cup \{\diamond\}$, where $\diamond \notin \Sigma$ plays the role of a *don't care* symbol (also called a *hole* or a *wildcard*).

A string w is a *substring* (or *factor*) of x if $x = uwv$ for two strings u and v; w is a *prefix* of x if u is empty, and it is a *suffix* of x if v is empty. A string u is a *border* of x if u is both a prefix and a suffix of x. The concatenation of $k \geq 1$ copies of a string u is called the *k-power* of u and is denoted by u^k. A string u is a *period* of x, if x is a prefix of u^k for some positive integer k.

The Parikh vector of a string x, denoted by $\mathcal{P}(x)$ (or simply $\mathcal{P}$, when x is clear from the context), enumerates the cardinality (or, multiplicity) of each letter of Σ in x. That is $\mathcal{P}[c]$ is the cardinality of a_c in x, also denoted as $|x|_{a_c}$, where $1 \leq c \leq \sigma$. Two Parikh vectors are *equal* if their corresponding elements are equal. Two strings are *abelian equivalent* if their Parikh vectors are the same. A string y of length m is said to *abelian match* at position i of string x if $\mathcal{P}(y) = \mathcal{P}(x[i\,..\,i+m-1])$.

The *size* of a Parikh vector is the sum of its components and is denoted by $|\mathcal{P}|$. Given two Parikh vectors $\mathcal{P}_1, \mathcal{P}_2$ we write $\mathcal{P}_1 \subseteq \mathcal{P}_2$, if $\mathcal{P}_1[c] \leq \mathcal{P}_2[c]$ for every $c \in \{1, \ldots, \sigma\}$, and $\mathcal{P}_1 \subset \mathcal{P}_2$ if $\mathcal{P}_1 \subseteq \mathcal{P}_2$ and $\mathcal{P}_1[c] < \mathcal{P}_2[c]$ for at least one $c \in \{1, \ldots, \sigma\}$. The *sum*, $\mathcal{P}'$, of $\mathcal{P}_1$ and $\mathcal{P}_2$ is a Parikh vector with elements the sums of the corresponding elements of $\mathcal{P}_1$ and $\mathcal{P}_2$, $\mathcal{P}'[c] = \mathcal{P}_1[c] + \mathcal{P}_2[c]$, for all $c \in \{1, \ldots, \sigma\}$. The difference is defined accordingly.

The string x is said to have an *abelian period* (h, p) if $x = u_0 u_1 ... u_{k-1} u_k$ such that:

$$\mathcal{P}(u_0) \subset \mathcal{P}(u_1) = ... = \mathcal{P}(u_{k-1}) \supset \mathcal{P}(u_k) \quad \text{and} \quad |\mathcal{P}(u_0)| = h,\ |\mathcal{P}(u_1)| = p$$

Factors u_0 and u_k are called the *head* and the *tail* of the abelian period respectively. Moreover, x is said to have a *weak abelian period* p if $\mathcal{P}(u_0) = \mathcal{P}(u_1)$ and $|\mathcal{P}(u_0)| = p$.

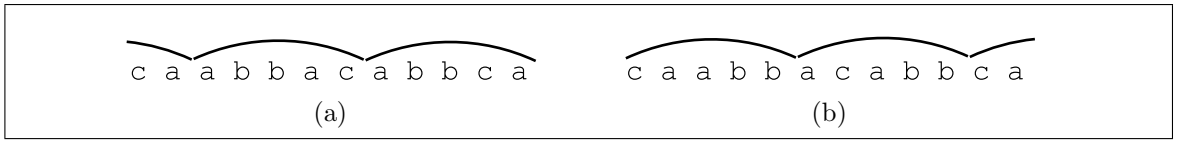

Figure 1: Abelian periods of the string $x = caabbacabbca$: (a) $(2, 5)$ is an abelian period of x, and (b) 5 is a weak abelian period of x

For example, the string $x = caabbacabbca$ has $(2, 5)$ as an abelian period and 5 as a weak abelian period, as shown in Figure 1. A natural order can be defined on abelian periods as follows: let (h, p) and (h', p') be abelian periods of a string x, then $(h, p) < (h', p')$ if $p < p'$ or $(p = p'$ and $h < h')$.

A string y of length $|y| = m < n$ is an *abelian border* of x if $\mathcal{P}(y) = \mathcal{P}(x[1\,..\,m]) = \mathcal{P}(x[n - m + 1\,..\,n])$. For example, the string $x = caabbacabbca$ has abelian borders of length $2, 5, 7$ and 10 as shown in Figure 2.

Figure 2: All abelian borders of the string $x = caabbacabbca$

A string $x = yz$, where $|y| = |z| > 0$ is an abelian square if $\mathcal{P}(y) = \mathcal{P}(z)$. Similarly a string x is an abelian k-power if it is the concatenation of k abelian equivalent strings, where $k \geq 2$ and $k \in \mathbb{N}$. Abelian primitive words are the words that are not abelian powers.

2.1 Special sequences

In this section, we define some special sequences that are used frequently in literature, as worst-case inputs in algorithms or for computing bounds for the number of regularities in strings.

The n^{th} *Fibonacci number*, denoted by f_n, is defined as:

$$f_0 = 1, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad \forall n \in \{2, 3, 4, \dots\}$$

The first few terms are: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$.

We define a (finite) *Fibonacci string*, F_n, as follows:

$$F_0 = b, \quad F_1 = a, \quad F_n = F_{n-1}F_{n-2} \quad \forall n \in \{2, 3, 4, \dots\}$$

Notice that $|F_n| = f_n$. The *infinite* Fibonacci string is the infinite string

$$F_\infty = abaababaabaababaababa\dots$$

which has every Fibonacci string except F_0 as a prefix. Alternatively, the Fibonacci string can be defined by the morphism

$$a \mapsto ab, \quad b \mapsto a$$

A generalization of the Fibonacci string, the k-bonacci string, for $k \geq 2$, is defined over the k-size alphabet $\{0, 1, \dots, k-1\}$ as $F_n^k = F_{n-1}^k F_{n-2}^k \dots F_{n-k+1}^k$, or by the substitution

$$0 \mapsto 01, \quad 1 \mapsto 02, \quad \dots \quad (k-2) \mapsto 0(k-1), \quad (k-1) \mapsto 0$$

For instance, the Tribonacci string ($k = 3$) is

$$F_0^3 = 0, \quad F_1^3 = 01, \quad F_2^3 = 0102, \quad F_3^3 = 0102010, \quad F_4^3 = 0102010010201, \ldots$$

Sturmian strings are infinite strings over a binary alphabet that have exactly $n+1$ factors of length n, for every $n \geq 0$. Note that the infinite Fibonacci string is a Sturmian string.

The *Thue-Morse* strings are defined by the following recursive equation:

$$T_0 = 0, \quad T_n = T_{n-1}\overline{T_{n-1}} \quad \forall n \in \mathbb{Z}^+$$

where $\overline{T_{n-1}}$ is the bitwise negation of T_{n-1}. The Thue-Morse strings can be also generated from the substitution map:

$$0 \mapsto 01 \quad \text{and} \quad 1 \mapsto 10$$

3 Pattern matching

In the following sections, consider $y[1\,..\,m]$ to be the pattern and $x[1\,..\,n]$ to be the text, where $n > m$. The abelian pattern matching problem is to identify all positions $i \in \{1, \ldots, n-m+1\}$ in x such that $\mathcal{P}(x[i\,..\,i+m-1]) = \mathcal{P}(y)$. In bibliography, it is a common assumption that $\mathcal{P}(y)$, rather than y, is given as input. If this is not the case, $O(m)$ time must be added to scan y in order to compute $\mathcal{P}(y)$.

3.1 Pattern matching without preprocessing

In this section, we consider the case where neither the text nor the pattern are given in advance and thus no preprocessing is performed on either.

The simplest way to solve this problem is to use a sliding window. A window of size m is aligned initially at position 1 of the text and the Parikh vector $\mathcal{P}(x[1\,..\,m])$ is computed. If this equals $\mathcal{P}(y)$, position 1 is reported as a match. Then, we slide the window to the right, one position at a time, and compare $\mathcal{P}(y)$ with the Parikh vector of the new window. As the window moves from position $i-1$ to i, the new Parikh vector is computed in constant time by removing the character $x[i-1]$ (the first character in the previous window) and inserting $x[i+m-1]$ (the last character in the current window), if the two are different. At each position i, $\mathcal{P}(x[i\,..\,i+m-1])$ has to be compared to $\mathcal{P}(y)$ in $\Theta(\sigma)$ time, thus overall this process takes time $\Theta(\sigma n)$. By using an additional $\Theta(\sigma)$ storage space the running time can be reduced to $\Theta(n)$ [50]:

F_0 *b*	T_0 0
F_1 *a*	T_1 01
F_2 *ab*	T_2 0110
F_3 *aba*	T_3 01101001
F_4 *abaab*	T_4 0110100110010110
F_5 *abaababa*	T_5 0110100110010110100101100110100 1
F_6 *abaababaabaab*	T_6 0110100110010110100101100110100 1
F_7 *abaababaabaababaababa*	1001011001101001011010011001011 0
(a)	(b)

Figure 3: The first few (a) Fibonacci strings, and (b) Thue-Morse strings

maintain an additional Parikh vector, $\mathcal{D}$, of size σ, to store the differences between the two Parikh vectors

$$\mathcal{D}[c] = \mathcal{P}(x[i\,..\,i+m-1)[c] - \mathcal{P}(y)[c] \qquad \text{for } c \in \{1,\ldots,\sigma\} \tag{1}$$

and an integer $k = |\mathcal{D}|$; after each sliding of the window, $\mathcal{D}$ and k are updated in constant time; a match is reported whenever $k = 0$.

A Horspool-type [64] variant of the previous sliding window technique, is defined in [50]. Let the window be aligned at position i of the text. The characters inside the window are checked from right to left; if, while scanning the window, say at position j, the cardinality of $x[j]$ in the window exceeds that of the same symbol in y (called an *overflow*) —implying that $\mathcal{P}(x[j\,..\,i+m-1] \not\subseteq \mathcal{P}(y)$— we are sure that the current window will not match y, and we shift the window to the right so that the character that caused the overflow is now omitted; that is, the position of the new window is $j+1$. Unfortunately, although this algorithm makes larger shifts than the previous one, if the overflow happens towards the left end of the window, some characters will be scanned again when the window is shifted, leading to a worst-case time complexity of $O(nm)$. For this reason, this method is efficient only if it is known in advance that the number of matches of y in x is small.

In order to improve the worst-case running time, Ejaz et al. [50] proposed a mechanism to avoid rescanning the same characters after a shift. When the overflow occurs at position j of window $x[i\,..\,i+m-1]$, the Parikh vector $\mathcal{P}(x[j\,..\,i+m-1])$ is already computed; after the shift, the new window $x[j+1\,..\,j+m]$ overlaps with the previous at positions $x[j+1\,..\,i+m-1]$, whose Parikh vector is easily computed from $\mathcal{P}(x[j\,..\,i+m-1])$ by subtracting $x[j]$. In this way, the Parikh vector of the prefix of the new current window is already computed. Characters are scanned again from the right end of the window towards the left, but now only the symbols $x[i+m\,..\,j+m]$ have to be checked in order to identify a match (provided that no new overflow occurs). The complexity of this method is $O(n/(1-\epsilon))$, where ϵ is a user-defined constant.

3.2 Pattern matching with preprocessing

In this version of the abelian pattern matching problem, the text x is given in advance. This allows preprocessing x to construct a data structure (an *index*), which can subsequently be used to search for a number, $K \gg 1$, of different patterns in x. Without any preprocessing, this would require $O(Kn)$ time using one of the available pattern matching algorithms described in Section 3.1.

A simple method for solving this problem, is to use again a sliding window (as in Section 3.1) to scan the whole text, precomputing and storing all Parikh vectors that occur in x. However, this idea only works if the pattern length, m, is known in advance, which is rarely the case. Otherwise, this process must be repeated once for every $m \in \{1,\ldots,n\}$. In the worst case, as many as $O(n^2)$ dinstinct Parikh vectors may appear in a text x, thus $O(n^2)$ storage space is required. Finding all occurrences of a pattern is then a matter of searching the list of all Parikh vectors of x, which can be done in $O(\log n + M)$ time, where M is the number of matches, if the list is sorted.

Ejaz [49] proposed following the above process to find all the Parikh vectors of x of a given length m, but then storing them in a trie for fast searching. Each Parikh vector is an m-tuple of integers in the range $\{0,\ldots,m\}$ and thus can be considered as

a string of length m over the alphabet $\{0, \ldots, m\}$. In this way, the $n - m + 1$ Parikh vectors of length m that occur in x (not necessarily all distinct) can be inserted in a simple trie, called *the abelian tree*. Notice that the edges of the abelian tree are labelled with the multiplicities of the symbols of the Parikh vectors and that nodes at the same level of the tree represent the same character. The time to construct and the space required for this tree depend on the implementation of the internal nodes:

- an array can be used to store pointers to the children of a node; the time and space complexities for constructing the abelian tree will be $O(mn\sigma)$, while searching for a given pattern on this tree takes only $O(\sigma)$ time;
- a linked list, on the other hand, reduces the time to construct the abelian tree to $O(n(m + \sigma))$ and the space requirements to $O(n\sigma)$, but increases the time to search for a pattern to $O(m + \sigma)$.

There is clearly a trade-off between these two implementations. In [49], the author further suggests to use a truncated version of the array implementation of the abelian tree, where, instead of maintaining $(m+1)$-size arrays in each node, to store $(e_{max}+1)$-size arrays, where e_{max} is the maximum cardinality of any symbol in any Parikh vector of x. This compact abelian tree takes space similar to that of the linked-list implementation, $O(n(m + \sigma))$, while maintaining the search time of the array implementation, $O(\sigma)$, to find occurrences of a given pattern. The construction time is still $O(mn\sigma)$ though.

The abelian tree described above, stores all σ values contained in each Parikh vector. The drawback of this approach is that, especially when $\sigma \gg m$, a Parikh vector may contain several zero entries indicating non-existence of the particular symbols in the corresponding m-length substring of x. In such cases, the abelian tree, even in its compact form, occupies space larger than necessary. To tackle this problem, [49] proposed modifying the abelian tree so as to avoid storing edges labelled 0. In the new tree, nodes at the odd levels (*character nodes*) represent characters whose multiplicity is greater than zero in the Parikh vectors, while nodes at the even levels (*multiplicity nodes*) record the multiplicities of these characters. Assuming that there is a minimal perfect hash function for the alphabet Σ, the time and space complexities of this tree are $O(n(m+\sigma))$, while the time to find a pattern is now reduced to $O(\sigma_{\mathcal{P}})$, where $\sigma_{\mathcal{P}}$ is the number of non-zero entries in the Parikh vector $\mathcal{P}$, and consequently in the worst case $O(m)$. In case that this assumption does not hold, the time and space complexities of the tree are $O(mn)$ and the time to find the occurrences of a pattern $\mathcal{P}$ is $O(\sigma_{\mathcal{P}} \log \sigma)$.

A Boyer-Moore type of algorithm [19] is presented by Cicalese et al. [33]. A window is shifted along the text, first by moving its right boundary (expanding the window) and then by moving the left boundary (shrinking). A similar technique was used by Butman et al. [25] on run-length encoded texts (see Section 3.5). Let L be an index in $\{0, \ldots, m\}$ such that $L + 1$ marks the first character of the current window, and $R \in \{m, \ldots, n\}$ be the position of the last character of the window, and assume that the Parikh vectors of all the *prefixes* of x have been precomputed. At each step of the algorithm, R is moved to the earliest position that can "fit" (see below) the next occurrence of y, and then L is moved rightwards too until either the window length is m and a match has been found or it is confirmed that y cannot occur in $x[L+1\,..\,R]$ and the process is repeated. The rules for updating R and L are described next.

- The *first fit rule* is used at each step for moving R rightwards, to the earliest position that can possibly fit the next occurrence of y. Observe that in order for

y to match $x[L+1\,..\,R]$, for some L and R with $R-L=m$, R must necessarily be chosen such that

$$\mathcal{P}(y) \subseteq \mathcal{P}(x[1\,..\,R]) \tag{2}$$

thus, the first occurrence of y will end at the smallest R satisfying (2). Furthermore, notice that the first occurrence of y *after* position L of x will end in the smallest position R for which $\mathcal{P}(x[1\,..\,L]) + \mathcal{P}(y) \subseteq \mathcal{P}(x[1\,..\,R])$. In simple words, y may occur in $x[L+1\,..\,R]$, only if the prefix $x[1\,..\,R]$ can fit both the prefix $x[1\,..\,L]$ and the pattern.

– The *good suffix rule* is called after R has been updated, to update L. L should take the *largest* position for which $\mathcal{P}(y) \subseteq \mathcal{P}(x[L+1\,..\,R])$. If equality holds, then $x[L+1\,..\,R]$ matches y, therefore position L is reported and L is increased by 1. Otherwise, extraneous characters that don't appear in y are interspersed in $x[L+1\,..\,R]$ and thus L must be shifted right to the earliest position for which $\mathcal{P}(x[L+1\,..\,R]) \subseteq \mathcal{P}(y)$.

Both rules above are realized in [33] with the routine FirstFit, which is implemented using inverted tables. This has the advantage that it takes $O(n)$ space and that the original text x can subsequently be discarded. The running time of the algorithm is $O(\sigma J \log(\frac{n}{J}+m))$, where J is the number of "jumps" (iterations) required by the algorithm. Although J is $O(n)$ in the worst case, the authors proved that on average it is much smaller, yielding overall expected running time of $O(n \log m \sqrt{\frac{\sigma}{m \log \sigma}})$. Burcsi et al. [21,22] improved the running time by a $\log m$ factor, by replacing the inverted table with a wavelet tree [62], therefore making the new version sublinear whenever $m = \omega(\sigma/log\sigma)$.

Binary abelian pattern matching Here, both the text and the pattern are drawn from a binary alphabet. Cicalese et al. [33] provide an algorithm that only *decides* whether a given Parikh vector appears in a binary text, rather than locating all its occurrences. It creates an $O(n)$ size data structure in $\Theta(n^2)$ time and can subsequently decide each query in $O(1)$ time. The algorithm is based on the following very important property on binary strings: if two Parikh vectors $\mathcal{P}_1 = (a_1, b_1)$ and $\mathcal{P}_2 = (a_2, b_2)$, both of size m and with $a_1 \leq a_2$, occur in x, then so does any m-length Parikh vector $\mathcal{P} = (a, b)$ with $a_1 \leq a \leq a_2$. Thus, it suffices to identify for each m the maximum and minimum number of a's in any Parikh vector of length m in x. Moreover, instead of precomputing this index in $\Theta(n^2)$, one could construct it in a lazy manner, only computing those entries that are needed to decide the occurrence of the current pattern, and storing them for future queries. In this latter case though, deciding for a pattern y will take $O(n)$ time, if it is the first pattern of length m to be looked for in x, and $O(1)$ for subsequent queries of the same length.

Soon after the publication of [33], Burcsi et al. [21,22] and independently Moosa and Rahman [74] improved the preprocessing time from $O(n^2)$ to $O(n^2/\log n)$, by reducing the problem to a (min,+)-convolution. Then, [75] provided two additional ways of constructing an $O(n)$ size index in $O(n^2/\log n)$ time, one that uses the Four-Russians technique and a second one that uses a lookup table. By further combining these latter indexes, the authors achieved the current lowest worst-case complexity of $O(n^2/\log^2 n)$, assuming word-RAM operations.

Badkobeh et al. [8] presented more recently a new index, again for the decision problem, which is based on the run-length encoding of the text. The size of the index

is in the worst case $O(n)$, like the previous indices of this kind, but the authors proved experimentally that in practice it is linear to the run-length encoded size, n', of the text. The time to construct the index depends on the compressibility of the text; it is $O(r^2 \log r)$, where r is $\Theta(n')$, and therefore much faster when x is well-compressible using run-length encoding. Unfortunately, the price paid for the decrease in the size of the index is an increased time to decide occurrence of the pattern, which now becomes logarithmic to the size of the index.

An *approximate* index for binary strings was presented in [34]. The index is constructed in time $O(k_{\epsilon,\eta} \cdot n^{1+\eta})$, where n is the length of the text, ϵ and η are constants that determine the probability of error, and $k_{\epsilon,\eta}$ is a constant depending on the choices for ϵ and η. Once constructed, the index can *approximately* answer decision queries in constant time, where approximate here means that it may return some false positive matches, though no false negatives.

3.3 Approximate pattern matching

Recall that a pattern y matches a text x at position i if and only if the Parikh vectors of y and $x[i\,..\,i+m-1]$ are identical. *Approximate* pattern matching, in abelian terms, refers to finding substrings of x whose Parikh vector is *similar* —but, perhaps, not identical— to that of y. The similarity, or, equivalently, the *distance* (also called, the *error*) between two Parikh vectors can be defined in various ways. Formally, the approximate abelian pattern matching problem is: given a text x, a pattern y, a threshold t and a distance function d, find all positions i of x such that $d(\mathcal{P}(y), \mathcal{P}(x[i..j])) < t$, for some j in $i{+}1\,..\,n$. In this section, we present algorithms for the approximate abelian pattern matching problem under different distance models.

The substitution distance function The substitution distance between two strings y and y', both of length m, is the minimum number of substitutions that must be performed on symbols of y' so that $\mathcal{P}(y')$ becomes identical to $\mathcal{P}(y)$. Notice that by substituting a symbol a with a symbol $b \neq a$ in y', both the cardinalities of a and b change: the former is decreased by 1 and the latter is increased by 1. Hence, the distance between y and y' is

$$d(y, y') = \frac{1}{2} \sum_{c=1}^{\sigma} |\mathcal{P}(y)[c] - \mathcal{P}(y')[c]|$$

The algorithm for approximate pattern matching with this distance function [49] uses a sliding window of size m that is shifted along the text, similar to the pattern matching algorithms seen in Section 3.1. At step i, we observe that Parikh vector of the current window differs from the previous one only by the substitution of $x[i-1]$ with $x[i+m-1]$. We maintain a Parikh vector $\mathcal{D}$, similar to (1), to hold the absolute differences between the Parikh vectors of y and that of the current window, $x[i\,..\,i+m-1]$, and update $\mathcal{D}$ of the previous step, by decreasing $\mathcal{D}[x[i-1]]$ by 1 and increasing $\mathcal{D}[x[i+m-1]]$ by 1. Then, the new distance, $d = d(y, x[i\,..\,i+m-1])$ is computed in constant time from that of the previous step, $d' = d(y, x[i-1\,..\,i+m-2])$, as follows:

$$d = \begin{cases} d', & \text{if } \mathcal{D}[x[i-1]] < 0 \text{ and } \mathcal{D}[x[i+m-1]] \leq 0, \\ & \text{or if } \mathcal{D}[x[i-1]] \geq 0 \text{ and } \mathcal{D}[x[i+m-1]] > 0 \\ d'+1, & \text{if } \mathcal{D}[x[i-1]] < 0 \text{ and } \mathcal{D}[x[i+m-1]] > 0 \\ d'-1, & \text{if } \mathcal{D}[x[i-1]] \geq 0 \text{ and } \mathcal{D}[x[i+m-1]] \leq 0 \end{cases} \tag{3}$$

The algorithm takes constant time at each position of x, thus overall requires $\Theta(n)$ time. Space-wise, only $O(\sigma)$ additional space is required for the Parikh vector $\mathcal{D}$ and 1 integer for the current distance d.

The insertion/deletion (InDel) distance function The InDel distance between two strings $y = y[1\,..\,m]$ and $y' = y[1\,..\,m']$, where m is not necessarily equal to m', is the minimum number of insertions of new characters in y' and deletions of existing characters from y' that are required in order for $\mathcal{P}(y')$ to become equal to $\mathcal{P}(y)$. Formally,

$$d(y, y') = \sum_{c=1}^{\sigma} |\mathcal{P}(y)[c] - \mathcal{P}(y')[c]|$$

A sliding window method can be used again, only in this case the size of the window is not fixed, it can vary between $m-t$ and $m+t$. Therefore, at some positions in the text more than one approximate occurrences of y may exist. To overcome this problem, [49] describes an algorithm for locating only the *maximal* substrings of x that approximately match y. A substring $x[i\,..\,j]$ that approximately matches y ($d(y, x[i\,..\,j]) < t$) is maximal if there is no other substring $x[i'\,..\,j']$ with $i' \leq i$ and $j' \geq j$ that also approximately matches y.

The algorithm maintains a window of size $m-t$, the minimum allowed size. At each position i, we check whether the current window, say $x[i\,..\,j]$, is a *potential match*, that is, whether by extending the window to the right it is possible to approximately match y. If not, the window is shifted to the next position to the right. On the other hand, if the current window is a potential match, we extend the window to the right until we obtain the longest substring, $x[i\,..\,j']$, of x starting at position i that approximately matches y; if this match is maximal, then positions i, j' are reported as the starting and ending positions of a maximal approximate match. It is easy to identify whether a match in the current window is maximal, by comparing its ending position with that of the last maximal approximate match that was previously reported: if it is larger, the current match is maximal. A Parikh vector, $\mathcal{D}$, that holds the differences between the Parikh vector of y and that of the current window in x is maintained again. The values of $\mathcal{D}$ for the new shifted window, $x[i\,..\,i+m-t-1]$, are computed from those of the previous smallest-size window, $x[i-1\,..\,i+m-t-2]$, similar to (3), with the only difference that d is now increased/decreased by 2 in the last two cases —one insertion and one deletion operations are required.

The running time of this algorithm is $O(n + Mt)$, where M is the number of potential matches of length $m - t$ in x and t is the threshold, since at the positions where such a potential match exists we spend $O(t)$ time trying to extend the match to the right. The algorithm, other than the input, maintains the Parikh vector $\mathcal{D}$ and some integer variables, hence the space complexity is $O(\sigma)$.

The minimum operations (MinOp) distance function The MinOp distance function combines the previous two distance functions, in that it allows both substitutions and insertions/deletions. Note that the effect of one insertion and one deletion together can be achieved by a single substitution operation. Therefore, for two strings $y = y[1\,..\,m]$ and $y' = y'[1\,..\,m']$, we transform y' to y with the following set of operations:

- if $m' > m$, we perform $m' - m$ deletions and as many substitutions are required after that;

– if $m' < m$, we perform $m' - m$ insertions and as many substitutions are required after that;
– if $m' = m$, we only perform substitutions.

Hence the distance is now defined as

$$d(y, y') = \frac{1}{2}\left(\sum_{c=1}^{\sigma} |\mathcal{P}(y)[c] - \mathcal{P}(y')[c]| + |m - m'|\right)$$

Ejaz [49] describes an algorithm that slides a window of size m along the text. If, at position i of x, the window $x[i\,..\,i+m-1]$ approximately matches y, then position i is reported as a match. Let $d = d(y, x[i\,..\,i+m-1])$. Note that at position i there might also exist windows of length less than or larger than m that also approximately match y; such windows will have distance from y larger than or equal to d, for the reason that by increasing (resp. decreasing) the window, a number of deletions (resp. insertions) must be introduced. For this reason, the algorithm reports for each position i where a substring that approximately matches y occurs, a range of ending positions, $j'\,..\,j''$, such that $x[i\,..\,j]$, with $j' \leq j \leq j''$, approximately matches y.

This algorithm requires $O(n + Mt)$ time overall: $O(n)$ for finding approximate matches of length m, as in the substitution distance model, and $O(Mt)$ time for extending the windows, where M is the number of approximate matches of length m. The space complexity is $O(\sigma + t)$, additional to the input: $O(\sigma)$ space for the Parikh vector $\mathcal{D}$ and $O(t)$ space for extending each m-length match.

Searching for a range of patterns In [23], Burcsi et al. followed a different approach in how they define abelian approximate pattern matching. Instead of using a pattern y as input, they take two Parikh vectors, $\mathcal{P}_\ell$ and $\mathcal{P}_u$, such that $\mathcal{P}_\ell \subseteq \mathcal{P}_u$, which represent the lower and upper bound respectively of the Parikh vectors to be searched for in x. Thus, the approximate pattern matching problem in this case is defined as: given x, $\mathcal{P}_\ell$ and $\mathcal{P}_u$, locate all substrings $x[i\,..\,j]$ of x for which $\mathcal{P}_\ell \subseteq \mathcal{P}(x[i\,..\,j]) \subseteq \mathcal{P}_u$ and $\mathcal{P}(x[i\,..\,j])$ is maximal. The notion of maximality is necessary here too, to avoid reporting matches that occur within other matches. $\mathcal{P}(x[i\,..\,j])$ is maximal if neither $\mathcal{P}_\ell \subseteq \mathcal{P}(x[i-1\,..\,j]) \subseteq \mathcal{P}_u$ nor $\mathcal{P}_\ell \subseteq \mathcal{P}(x[i\,..\,j+1]) \subseteq \mathcal{P}_u$.

The algorithm moves a window on x using jumps, as in [33] (see Section 3.2), in three phases. In the *expanding phase*, the window's right end, R, is moved rightwards to the earliest position that can fit $\mathcal{P}_\ell$, that is, $\mathcal{P}_\ell \subseteq \mathcal{P}(x[L+1\,..\,R])$. In the *shrinkage phase*, its left end, $L+1$, is shifted to the right until $\mathcal{P}(x[L+1\,..\,R]) \subseteq \mathcal{P}_u$. Finally, in the *refining phase* the requirement of maximality is achieved by extending the window to the right as long as $\mathcal{P}(x[L+1\,..\,R]) \subseteq \mathcal{P}_u$.

This algorithm runs in $\Theta(n)$ time, using additional $\Theta(\sigma)$ space to hold the Parikh vector of the current window in x. The authors in [23], take the algorithm one step further by also providing a version with preprocessing, where a wavelet tree is used as an index of x, which runs in $O(\sigma n)$ worst-case time and $O(n\sqrt{\frac{\sigma}{|\mathcal{P}_\ell| \log \sigma}})$ expected time.

3.4 Sequence alignment

There are two versions of the sequence alignment problem, global alignment and local alignment. *Global sequence alignment* is defined as follows: given two strings x and y,

Figure 4: Abelian sequence alignment example [10]

find an alignment between them that requires the minimum number of substitutions, insertions and deletions. In the *local alignment* problem on the other hand, one of the two strings, say y, is smaller than the other, and one seeks the substring of x whose alignment cost with y is minimal. In classical string matching, both of these problems can be solved with dynamic programming in $O(nm)$ time.

In the abelian version of the problem [10], two substrings of x and y are considered to match (thus, zero cost) iff their Parikh vectors are equal. Figure 4 shows an example of abelian sequence alignment, where `|` denotes a character match and `<-->` denotes a substring abelian match. Alignment problems are usually described in terms of maximizing a scoring function, rather than minimizing a cost (or distance function). In the simplest case, which is equivalent to minimizing insertions, deletions and substitutions, the scoring function assigns score 1 to every character that matches, either directly or within a substring (abelian match). In the example of Figure 4, the score would be 11.

Benson [10] addressed the abelian alignment problem using again dynamic programming. A difficulty comes from the fact that it is much easier now for large substrings to (abelian) match each other, generating alignments that are not biologically very meaningful. For this reason, the author proposes the use of an additional input, an integer *limit*, for an upper bound on the length of the substring that may match in abelian terms. Initially, the algorithm preprocesses x and y to find, for each pair i, j with $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, the shortest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ that abelian match; this can be achieved in $O(nm)$ time. Then, the running time of the dynamic programming depends on the scoring function of choice, s. Two classes of functions were tested in [10]: functions that assign a score based on the match length and functions where the score depends both on the length and on the content. If the scoring function is either additive or subadditive ($s(i+j) \leq s(i) + s(j)$), then the algorithm runs in $O(nm)$ time, otherwise it takes $O(nm \cdot limit)$.

3.5 Other pattern matching variants

Sub-pattern matching The abelian sub-pattern matching problem is, for a given text x and pattern y, to locate in x all Parikh vectors $\mathcal{P}'$ for which $\mathcal{P}' \subseteq \mathcal{P}(y)$. This problem was addressed in [21] by modifying their algorithm for pattern matching with preprocessing [33] (see Section 3.2), without worsening its running time ($O(n\sigma)$ in the worst case). The idea is to assign a weight to each character and then look for occurrences y' in x of maximal weight with $\mathcal{P}(y') \subseteq \mathcal{P}(y)$.

Abelian pattern matching in run-length encoded text Butman et al. [24,25] presented an algorithm for finding all occurrences of a Parikh vector in a run-length encoded text. A window is shifted on x; at each step of the algorithm, first the right boundary of the window is moved to the right until it contains all the characters of y (*sufficient* window) and then the left boundary is moved to the rightmost position that does not violate sufficiency (*minimal* window). The algorithm runs in $O(n' + \sigma)$

time, where n' is the length of the run-length encoded version of x. This algorithm could prove useful even when x is not run-length encoded, if x is provided in advance and a large number of patterns will be searched on it; in this case, $O(n)$ preprocessing time would be required to first compress x.

Fingerprint matching Amir et. al [2] examined a variant of the abelian pattern matching problem, where one is interested only what symbols occur in a given substring (known as the *fingerprint* of the substring), as opposed to the cardinality of each symbol in the substring (Parikh vector). After preprocessing the text x in $O(n\sigma \log n \log \sigma)$ time, they show how to answer queries of two types: a) given an integer k, compute the number of distinct fingerprints of size k in x, in $O(1)$ time, and b) given a fingerprint ϕ compute the number of substrings of x with fingerprint ϕ, in $O(\sigma \log n)$ time. The preprocessing involves scanning the text with a variable size window, that is shifted rightwards such that it always contains k-sized fingerprints. Each distinct fingerprint encountered while scanning x is stored with a unique name, using the naming technique [3]. When a pattern y is given, its name is built using the Parikh mapping technique again, and that unique name is searched in the index built from x.

Pattern discovery Eres et al. [53] studied the problem of pattern discovery, which is defined as follows: given a set of strings, S, with total length n, and an integer K, find all the patterns (substrings of any of the strings in S) whose Parikh vector $\mathcal{P}$ occurs at least K times in S. Hence, in this problem the pattern is not given in advance, but rather is discovered by the algorithm. As the number of such patterns is quadratic in the worst case, the notion of pattern maximality is introduced, which ensures that no pattern discovered by the algorithm will be covered by some other pattern. The algorithm uses a sliding window together with the naming technique of [2] and runs in $O(Ln \log n \log \sigma)$ time, where L is the length of the longest pattern discovered. The motivation of this problem comes from biology, and the authors used the above algorithm to find clusters of genes that occur frequently close to each other in a large database of protein sequences from *E. Coli* (see Section 8).

Parida [78] later extended this work by allowing gaps (extraneous characters) between the characters of the discovered pattern. The algorithm executes in two stages: in the first stage, all maximal patterns are computed for which the gap is not bounded ($g = \infty$), and stored in a balanced binary tree; then, in the second stage, maximal patterns are chosen that satisfy the gap bound. The complexity of the algorithm is $O(n \log s + M\sigma \log \sigma)$, where n is the total length of the set of sequences, s is the number of sequences in the set, and M is the number of maximal patterns that match. The authors also provide an optional third stage for extracting the non-maximal patterns out of the maximal ones.

3.6 Open problems

Problem 1 (Complex pattern matching, [49]) *Develop algorithms for* complex *pattern matching in strings, where a complex pattern is a combination of abelian patterns, classical patterns and regular expressions.*

For example, $(2, 5)(a + b)^*abba$ is a complex pattern, which matches in the first 7 positions the Parikh vector that contains two a's and 5 b's, then it matches the

regular expression $(a+b)^*$ and in the last 4 positions it matches the classical pattern *abba*.

Problem 2 (Abelian pattern matching with fixed-length gaps, [49]) *Consider a pattern that is composed of a number of Parikh vectors separated by fixed-size gaps. Devise an algorithm to find all occurrences of the pattern in x.*

For example, $(2,0,1,0)3?(1,0,0,2)$ denotes the pattern where the first three positions match the Parikh vector $(2,0,1,0)$, the next three constitute a gap (that is, we ignore their content) and the last three match the Parikh vector $(1,0,0,2)$.

Problem 3 (Abelian pattern matching with bounded-length gaps) *Consider a pattern that is composed of a number of Parikh vectors separated by bounded-length gaps. Devise an algorithm to find all occurrences of the pattern in x.*

For example, $(2,0,1,0)3-5?(1,0,0,2)$ denotes the pattern where the first three positions match the Parikh vector $(2,0,1,0)$, the next three, four or five positions denote a gap, and the last three match the Parikh vector $(1,0,0,2)$.

Problem 4 (Character-bounded approximate abelian pattern matching,[23]) *For a given text x, pattern y and threshold $t \leq min_c\mathcal{P}(y)[c]$, find all positions in x where either $\mathcal{P}(y)$ occurs, or some y' with $|\mathcal{P}(y')[c]-\mathcal{P}(y)[c]| \leq t$ for all $c \in \{1,\ldots,\sigma\}$.*

Problem 5 ([75]) *Does there exist an $o(n^2/\log^2 n)$-time and $O(n)$-space data structure for abelian matching for binary alphabet?*

Problem 6 ([22]) *Create a compressed index for abelian pattern matching in alphabets of size $\sigma > 2$.*

Problem 7 ([34]) *Create an approximate index for abelian pattern matching in alphabets of size $\sigma > 2$.*

Problem 8 ([34]) *Devise Las Vegas algorithms for the abelian pattern matching problem.*

4 Abelian periods and borders

A string u is a *period* of a string x, if x is a prefix of u^k for some positive integer k (i.e. x is a prefix of ux). *The period* of x, denoted by $Period(x)$, is the length of the shortest period of x. A lot of research has been conducted on classical periods, e.g. algorithms for finding all periods of a string, algorithms for the computation of the period array of a string [71], etc. Abelian periods are more flexible than classical ones and are defined in terms of Parikh vectors as in [35] (see Section 2). Relevant research has concentrated on their combinatorial properties, on algorithms for their efficient computation and on abelian borders (the complementary notion of the period).

4.1 Properties

Fine and Wilf's theorem for abelian periods In 2006, Constantinescu and Ilie [35] proved a variant of Fine and Wilf's periodicity theorem; they showed that if a string x has two coprime abelian periods p, q and length at least $2pq - 1$, then x has also $gcd(p, q) = 1$ as a period. Blanchet-Sadri et al. [16] then proved that the length $2pq-1$ is optimal, by providing a way to construct a non unary string of length $2pq-2$ with abelian periods p and q for any coprime positive integers p, q (an open problem proposed by Constantinescu and Ilie [35]). They also showed that if $gcd(p, q) = d$, the string contains at most d distinct letters (another open problem proposed in [35]). By using the algorithm mentioned above (an implementation can be found in [15]), they extended these results for partial words and showed that:

- If w is a partial word with h holes and abelian periods p and q which are relatively prime and $|w| > (h + 2)pq - 1$, then w has period 1.
- The length $(h + 2)pq - 1$ is optimal for the above case.

Fine and Wilf's theorem has been also extended for k-abelian periods [68]. Before proceeding we need to give some more definitions:

Definition 1 ([68]). *Two strings u and v are k-abelian equivalent if they contain the same number of occurrences of each factor of length at most k.*

Definition 2 ([68]). *Consider a string $x = x[1\,..\,n]$, k-abelian equivalent strings $u_0, \ldots, u_{m+1}$ of length p, and a non negative integer $r \leq p - 1$ such that:*

$$x = u_0[p - r + 1\,..\,p]\ u_1\ \ldots\ u_m\ u_{m+1}[1\,..\,n - mp - r],$$

then x has k-abelian period p. If $r = 0$, then x has initial k-abelian period p.

Karhumäki et al. [68] gave only some bounds for the general case and an exact value for the case when p and q are initial periods and $k = 2$; the optimal value in that case is $max\{m_1p, n_{-1}q\}$ for some suitably chosen m_1 and n_{-1}.

Abelian versions of the Critical Factorization Theorem The Critical Factorization Theorem states that the global period of a string x is the maximum of its local periods (shortest squares centered in each position of the string). Avgustinovich et al. [7] attempted to give an abelian analogue of this theorem, in fact they proved that there exist infinite non-periodic strings with bounded abelian powers centered at every position of the string, contrary to the classical version of the theorem. They also showed some other cases where existence of abelian powers in the string implies global periodicity.

4.2 Computing all abelian periods

Recently Fici et al. [58] gave five algorithms for the computation of all abelian periods of a string. They proposed two offline algorithms, one brute force algorithm and one that uses a select array, that run in $O(n^2\sigma)$. While the brute force algorithm tries to compare all the Parikh vectors to check if an abelian period exists in x, the second algorithm keeps the occurrences of each letter in the select array, thus being able to exclude cases where the head of the period can not be matched in the subsequent positions of x.

The authors of [58] gave also three online algorithms, where the first two run in $O(n^3\sigma)$ and the other one runs in $O(n^3log(n)\sigma)$. The first algorithm keeps for each abelian period (h, p), where $0 \leq h < p$ and $1 \leq p \leq n$, the length of the longest prefix of x that has an abelian period (h, p) in a two dimensional array. Given the table for $x[1\,..\,i]$, the table for $x[1\,..\,i+1]$ is updated from the previous one by considering only the positions in the array with value i. The second algorithm runs in the same manner, however the values previously stored in the table are now stored in a list. Finally, the third algorithm is a bit different, instead of tables or lists uses heaps and partitions the abelian periods according to their tail lengths. Experimentally the offline algorithm that makes use of the select array is said to be the fastest in practice.

Later Christou et al. [32] presented two $O(n^2)$ algorithms for the computation of all abelian periods of a string $x = x[1\,..\,n]$. The first one maps each letter to a suitable number such that each factor of the string can be identified by the unique sum of the numbers corresponding to its letters. Formally, given a mapping $s : \Sigma \to B$, where B is the set that contains 0 and the first $\sigma - 1$ powers of $n + 1$, such that:

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

the *S-signature* of x is defined to be equal to $\sum_{i=1}^{|x|} s(x[i])$. The second one maps each letter to a prime number such that each factor of the string can be identified by the unique product of the numbers corresponding to its letters. Formally, given a mapping $p : \Sigma \to A$, where A is the set of the first σ prime numbers, such that $p(a_i)$ is the i^{th} prime number, the *P-signature* of x is defined to be equal to $\prod_{i=1}^{n} p(x[i])$.

Then the required checks of Parikh vectors, necessary to identify abelian periods, can be performed with a single operation in constant time. Additionally, they defined weak abelian periods on strings and gave an $O(n \log n)$ algorithm for their computation.

4.3 Abelian borders

In classical strings, the notion of the border is complementary to that of the period. Although this does not hold in the abelian case, abelian borders still provide useful information for the input string. Christodoulakis et al. [31] studied the appearance of abelian borders in strings. They have shown three linear time algorithms for the computation of all abelian borders of a string x. The first two algorithms that identify all borders of a string make extensive use of the *P-Signature* and *S-Signature* respectively, while the third one simply keeps track of the difference between Parikh vectors of prefixes and suffixes of the string of the same length. They also commented on the appearance of abelian borders in Fibonacci and Thue-Morse words. Finally, they proved that the average length of abelian borders of a string x —if x has any abelian borders— is $n/2$, and also that a binary string of length n has $\Theta(\sqrt{n})$ abelian borders on average.

4.4 Open problems

Problem 9 *Is there an $o(n^2)$ algorithm that identifies all abelian periods of a string?*

Problem 10 *Is there an $o(n \log n)$ algorithm that identifies all weak abelian periods of a string?*

Problem 11 *[7] Let k be an integer. Does there exist a non-periodic string x and integers l_1 and l_2, such that x contains an abelian $2k$-power of length l_1 or l_2 centered at every position?*

5 Abelian powers

The appearance of abelian powers in strings was one of the main problems that appeared in scientific literature [52]. Since then, research has been concentrated on algorithms for the identification of all abelian powers in a string, enumeration of abelian powers and abelian primitive strings as well as on their appearance in special strings such as Strurmian, Fibonacci and Thue-Morse words.

5.1 Computing all abelian powers

Cummings et al. [40] showed a rather straight-forward $\Theta(n^2)$ algorithm for the computation of all abelian squares of a string. They showed that their algorithm is optimal under the encoding of the output that they used, by showing the appearance of abelian squares in Fibonacci and other strings.

The above algorithm can form a basis for the identification of other powers in strings. However, no other relevant result has appeared in the literature yet.

5.2 Counting abelian powers

Richmond and Shallit [82] counted the number, $f_\sigma(n)$, of different abelian squares of length n over an alphabet of size σ, using basic combinatorial identities and gave an asymptotic estimate of this quantity:

$$f_\sigma(n) \sim \frac{\sigma^{2n+\frac{\sigma}{2}}(4\pi n)(1-\sigma)}{2}$$

Callan [26] presented a bijection between Barrucand deals and abelian squares, by showing a bijection between Barrucand deals and abelian matrices (a representation of an abelian square). We give the formal definition of a Barrucand deal as in [26]:

Definition 3 ([26]). *A* Barrucand n-deal *is formed as follows:*

- *Start with a deck of* $3n$ *cards,* n *colored red,* n *coloured green and* n *coloured blue, in denominations 1 through* n
- *Choose an arbitrary subset of the denominations and deal all cards of the chosen denominations into three equal-size hands to players designated red, green and blue in such a way that no player receives a card of her own color. Let* B_n *denote the set of* Barrucand n-deals.

In the same paper the Barrucand deal is then related to other special numbers and combinatorial identities, thus revealing their connections to abelian squares.

5.3 Abelian primitive strings

Abelian primitive strings are the strings that are not abelian powers. Domaratzki and Rampersad [48] showed that the set of abelian primitive strings is not context-free. Furthermore, contrary to the classical case, they proved that a string can have more than one abelian roots and they gave some bounds on that number. Regarding the problem of determining whether a string is abelian primitive, they showed a worst case linear time algorithm involving prime factorisation.

5.4 Appearance in special strings

Abelian powers have been identified in general Sturmian strings and also in Fibonacci and Thue-Morse words. Some researchers have also studied their appearance in circular strings. In this section, we give some of the properties and facts that appear in literature.

Karaman [67] showed a worst case linear time algorithm for the computation of all weak repetitions (abelian powers) in a Sturmian string by introducing a special encoding for the output. Later, Richomme et al. [83] showed that all Sturmian strings are everywhere abelian k-repetitive for all integers $k \geq 1$, i.e. every sufficiently long factor of the string has an abelian k^{th} power as a prefix. Furthermore, given a natural number k they proved that any position of a Sturmian string t starts with an abelian k^{th} power with abelian period ℓ_1 or ℓ_2. In the same context, Avgoustinovich et al. [7] showed that Sturmian strings have bounded right k-powers for every k, and they do not have bounded central powers. It is also interesting that the Fibonacci string, itself a Sturmian string, begins with arbitrarily high abelian powers [45].

Regarding the Thue-Morse word, Avgoustinovich et al. [7] proved that it has bounded abelian squares centered at every position in it and bounded abelian cubes to the right of every position in it. However there are no bounded abelian 4-powers centered or to the right of every position of the word.

Finally, S. Fraenkel et al. [59,60] studied the appearance of abelian squares in circular binary strings. The definition of the abelian square that they used is slightly different from the usual, as follows:

Definition 4 ([59,60]). *An abelian circular square is an abelian square which is possibly wrapped around the string: the tail protruding from the right end of the string reappears at the left end.*

They proved that the longest string with only k distinct abelian circular squares contains $4k + 2$ bits and has the form $(01)^{2k+1}$ or its complement.

5.5 Open problems

Problem 12 *Is there an $o(n^2)$ algorithm that identifies all abelian squares of a string?*

Problem 13 *How many strings of length $3n$, composed by σ letters, are abelian cubes?*

Problem 14 *How many strings of length rn, composed by σ letters, are abelian r-powers?*

Problem 15 *How many strings of length n, composed by σ letters, are abelian primitive?*

6 Pattern avoidance

Much of the research on abelian regularities in strings, and particularly abelian pattern avoidance, has been initiated by Erdős [52] who proposed the following problem:

Problem 5. Construct an infinite string on an alphabet as small as possible such that the string does not contain any abelian squares.

Since then, research has concentrated on the avoidance of abelian powers in infinite strings and also on the enumeration of abelian power-free strings, as well as on generalizations of them.

6.1 Infinite case

As mentioned earlier, Erdős [52] proposed the problem of constructing an infinite string avoiding abelian squares on an alphabet as small as possible, thus initiating the study of pattern avoidance in the abelian sense. It is easily seen by inspection that any ternary string of length 8 contains an abelian square [11]. While Erdős did not give a solution to the above problem, Evdokimov [54] gave a construction on 25 letters, which he later improved to 7 letters [55]. In 1970, Pleasants [80] gave a much better solution, a construction on 5 letters, until more recently Keränen [69] came with a construction on 4 letters. For his construction, Keränen used an endomorphism which he called g_{85}. The structure of g_{85} was further analysed in [70], where he gave a new endomorphism, g_{98}, the iteration of which also produces an infinite abelian square-free string. Compositions of g_{85} and g_{98} yield more infinite abelian square-free strings. Furthermore, Carpi [27] showed another abelian square-free substitution on four letters and proved that infinite abelian square-free quaternary strings are uncountable. Entringer et al. [51] proved that every infinite binary string has arbitrarily long abelian squares.

Regarding the problem of avoiding higher abelian powers, Dekking [47] showed the existence of an infinite binary string that avoids abelian 4-powers and the existence of an infinite ternary string that avoids abelian cubes. Much later, Currie and Aberkane [42] came with the same result by showing a cyclic binary morphism that avoids abelian 4-powers.

In the context of partial strings, Blanchet-Sadri et al. [12,13] investigated the problem of avoiding abelian squares in an infinite partial string. By employing constructions based on iterating morphisms, they proved that in the case of partial strings with one hole the minimal alphabet size is 4, while in the case of partial strings of more than one hole the minimal alphabet size is 5. In [14] they extended their results for higher powers showing:

- the existence of an infinite ternary partial string with infinitely many holes that avoids abelian 4-powers,
- the existence of an infinite quaternary partial string with infinitely many holes that avoids abelian cubes, and
- the existence of an infinite binary partial string with infinitely many holes that avoids abelian 6-powers.

Furthermore, they constructed an infinite abelian 4-free (i.e. avoiding 4-powers) binary partial string with one hole and an infinite abelian cube-free ternary partial string with one hole. Finally, they claim that by replacing the letters of an infinite arbitrary set of positions from an infinite abelian p-free string the string will cease to be abelian p-free.

6.2 Finite case

Power-free strings of finite length In the case of strings avoiding abelian squares the ternary case is quite important as 3 is the largest alphabet where every infinite

z_1 0
z_2 010
z_3 0102010
z_4 010201030102010
z_5 0102010301020104010201030102010
z_6 01020103010201040102010301020105010201030102010401020103010201

Figure 5: The first six Zimin strings

string does not avoid abelian squares. It is easy to observe that abelian square-free strings over an alphabet of size 3 have length ≤ 7 [11]. In the same context, Cummings [38] proved that there exist 127 different abelian square-free strings over an alphabet of size 3.

It has been shown that there exists an infinite string on a four letter alphabet that has no abelian squares [69]. Carpi [27] showed that the number of abelian square-free quaternary strings of each length grows exponentially. Regarding higher powers, Aberkane et al. [1] showed that the number of ternary strings of length n avoiding abelian cubes grows faster than $2^{\frac{n}{24}}$, while Currie [41] proved that the number of binary strings of length n avoiding abelian fourth powers grows faster than $2^{\frac{n}{16}}$.

Regarding the case that we are dealing with partial strings, Blanchet-Sadri et al. [12,13] investigated the number of partial strings of length n with a fixed number of holes over an alphabet of size 5 that avoid abelian squares and showed that this number also grows exponentially with n. As expected, the number of abelian p-free partial strings of length n with h holes also grows exponentially [14].

Maximal abelian power-free strings We begin by defining maximal abelian power-free strings:

Definition 6. *An abelian square-free string over an alphabet Σ which cannot be extended to the left or right with letters from Σ while remaining abelian square-free is called a maximal abelian square-free string (also denoted as σ-reflector).*

Zimin [90,91] showed that the length, $\ell(\sigma)$, of a maximal abelian square-free string over an alphabet of size σ, is bounded above by $2^\sigma - 1$, by giving the following recursive construction of the so called Zimin strings (see Figure 5):

$$z_1 = 0 \text{ and } z_k = z_{k-1}(k-1)z_{k-1} \text{ for } k > 1$$

Cummings [39] modified Zimin's strings to give strings for which $\ell(\sigma)$ is $O(2^{\frac{\sigma}{2}})$. He also proved that $\ell(\sigma) \leq 2^\sigma - 1$ using a construction based on gray codes [37]. More recently, Korn [72] managed to restrict $\ell(\sigma)$ to linear size. In fact, he gave constructions showing that $\ell(\sigma) \in [4\sigma - 7, 6\sigma - 10]$, for $\sigma \geq 3$. Finally, Bullock [20] refined Korn's methods to give some slightly better bounds on $\ell(\sigma)$, showing that $\ell(\sigma) \in [6\sigma - 29, 6\sigma - 12]$ for $\sigma \geq 8$.

A *crucial string* is a similar notion to a maximal abelian square-free string, defined as follows:

Definition 7 ([56,57]). *An abelian square-free string over an alphabet Σ which cannot be extended to the right with letters from Σ while remaining abelian square-free is called a crucial string.*

Evdokimov et al. [56,57] showed that the minimal length of a crucial string is $4\sigma - 7$ for $\sigma \geq 3$. Glen et al. [61] extended the above result for the case of abelian cubes and proved that the relevant length is $9\sigma - 13$ for $\sigma \geq 5$, and 2, 5, 11, and 20 for $\sigma = 1$, 2, 3, and 4 respectively. Moreover, for $\sigma \geq 4$ and $k \geq 2$, they gave a construction of length $k^2(\sigma - 1) - k - 1$ of a crucial string over Σ avoiding abelian k-th powers. For $k = 2$ and $k = 3$ the construction is optimal. For $k \geq 4$ and $\sigma \geq 5$, they showed that the length of crucial strings avoiding abelian k-powers is at least $k(3\sigma - 4) - 1$. Avgustinovich et al. [6] improved the above lower bound in the case that $\sigma \geq 2k - 1$, to $k^2\sigma - (2k^3 - 3k^2 + k + 1)$.

6.3 Generalizations

Avoiding small patterns An abelian pattern is said to be *abelian avoidable* if there exist infinitely many strings without it, more formally:

Definition 8 ([45]). *A pattern y is (abelian) k-avoidable if there are infinitely many strings over $\{1, 2, \ldots, k\}$ which avoid y (in the abelian sense). A pattern y is (abelian) avoidable if it is (abelian) k-avoidable for some k.*

Currie and Linek [43] classified all 3 letter patterns that are avoidable in the abelian sense and also gave a short list of four letter patterns for which abelian avoidance is undecided. Currie and Visentin [45] gave the first example of a binary pattern which is abelian 2-avoidable and contains no abelian fourth power. Later, they used some of the machinery built in that paper to prove that a sufficiently long binary pattern is abelian 2-avoidable [46]. In fact, they showed that binary patterns of length greater than 118 are abelian 2-avoidable.

Avoiding abelian inclusions *Abelian inclusions* provide a generalisation of abelian squares:

Definition 9 ([5]). *Consider a function $f(\ell) : N \to \mathbb{R}$. A string uv is said to be an $f(\ell)$-inclusion if $P(u) \subseteq P(v)$ and $|v| \leq |u| + f(|u|)$.*

Avoiding the zero function is the same as avoiding abelian squares. Avgustinovich and Frid [5] proved that $c\ell$-inclusions are unavoidable, but c-inclusions are avoidable for an arbitrary constant c.

***k*-abelian avoidability** The notion of k-abelian equivalence of strings, as introduced in Section 4, has been considered for avoidability problems. Huova et al. [65,66] showed that there exists an infinite quaternary string that avoids 2-abelian squares and an infinite octary string that avoids 2-abelian cubes. They also conjectured, and showed some evidence, that there exists an infinite binary string that avoids 2-abelian cubes.

Context-freeness For a formal definition of a context-free language see [79]. Main[73] showed that the set of strings over an alphabet of at least 16 letters containing an abelian square is not context-free by making use of the Interchange Lemma [77].

Morphisms preserving abelian square-free properties Carpi [28] studied morphisms that preserve the abelian square-free property of a string and the more general notion of substitutions with bounded abelian squares, i.e. substitutions assuring that the length of the abelian squares in a string remains below a certain value. It is also shown that there exist algorithms that can reveal if a substitution map has the above properties.

6.4 Open problems

Problem 16 *[14] How many letters do we need to construct a partial string with infinitely many holes that avoids abelian 3^{rd} powers (resp. 4^{th} powers, 5^{th} powers)?*

Problem 17 *[14] How many letters do we need to construct a 2-sided infinite partial string with one hole that avoids abelian 3^{rd} powers (resp. 4^{th} powers)?*

Problem 18 *It is known [20] that the shortest maximal abelian square-free string (also denoted as σ-reflector) over an alphabet of $\sigma \geq 8$ letters has length $\ell(\sigma) \in [6\sigma - 29, 6\sigma - 12]$. Is it possible to restrict $\ell(\sigma)$ in a smaller interval?*

Problem 19 *How the length of the shortest maximal abelian cube-free string over an alphabet of σ letters varies with σ?*

Problem 20 *How the length of the shortest maximal abelian r-free string over an alphabet of σ letters varies with σ?*

Problem 21 *[61] Prove or disprove the following conjecture:*
For $k \geq 4$ and sufficiently large σ, the length of a minimal crucial string over Σ avoiding abelian k-powers is given by $k^2(\sigma - 1) - k - 1$.

Problem 22 *[73] Is the set of permutation-containing strings over alphabets of fewer than 16 characters context-free?*

7 Abelian complexity of infinite strings

In this section, we examine combinatorial properties of the abelian theory on infinite strings. We begin with some definitions and properties of infinite strings.

An infinite string x is periodic (or *recurrent*) with period p, if $x[i + p] = x[i]$ for all i. x is *ultimately periodic* if $x[i+p] = x[i]$ for all sufficiently large i. x is *k-balanced* if for any two factors x_1 and x_2 of x, $|\mathcal{P}(x_1)[c] - \mathcal{P}(x_2)[c]| \leq k$ for all $c \in \{1, \ldots, \sigma\}$ and for some constant k. An 1-balanced string x is simply called *balanced*.

Let $\mathcal{F}_x(n)$ denote the set of all distinct n-length factors of the infinite string x. The *factor complexity*, $\rho_x(n)$, of x is defined as the number of distinct factors of x of length n, $\rho_x(n) = |\mathcal{F}_x(n)|$. In abelian terms, two factors are distinct iff their Parikh vectors are distinct; hence, the *abelian factor complexity*, $\rho_x^{ab}(n)$, is defined accordingly; $\rho_x^{ab}(n) = |\mathcal{F}_x^{ab}(n)|$, where $\mathcal{F}_x^{ab}(n)$ is the set of all distinct Parikh vectors of size n that occur in x.

There are similarities between the abelian and the non-abelian factor complexities. For instance, both can be used to characterise a periodic string; x is periodic if and only if either $\rho_x(n_0) \leq n_0$ for some $n_0 \geq 1$ [76], or $\rho_x^{ab}(n_0) = 1$ for some $n_0 \geq 1$ [36]. On the other hand, the two complexities may also differ significantly. For example, there exists an infinite string x where $\rho_x(n)$ grows exponentially, while $\rho_x^{ab}(n) \leq 3$ for all n [85]. In contrast, Cassaigne [29] found a string with linear $\rho(n)$ but unbounded $\rho^{ab}(n)$.

In 1983, Rauzy [81] set the question whether there exists any infinite string x with $\rho_x^{ab}(n) = 3$ for all $n \geq 1$ and suggested that most likely there is no such x. Some twenty five years later, Richomme et al. [85] proved that, on the contrary, such x does exist and provided two classes of strings that satisfy $\rho_x^{ab}(n) = 3$ for all $n \geq 1$:

– x is any aperiodic balanced string over $\{0, 1, 2\}$, or
– x is the image of an aperiodic binary string under the morphism

$$0 \mapsto 012 \quad \text{and} \quad 1 \mapsto 021$$

They further conjectured that there is no recurrent string x with $\rho_x^{ab}(n) = 4$ for all n.

This conjecture was proved by Currie and Rampersad [44], who actually obtained a stronger result: there is no infinite periodic string over an n-letter alphabet with constant abelian complexity $n \geq 4$. Saarela [86] proved that the situation changes if ones relaxes the requirement of the abelian complexity to be constant for all n. In particular, they proved that for every integer $c \geq 2$ there are infinite periodic strings with *ultimately* constant abelian complexity; that is, $\rho_x^{ab}(n) = c$ for all $n \geq c - 1$.

Balková et al. [9] developed a method for computing the abelian complexity of binary infinite strings associated with quadratic Parry numbers. Subsequently, Turek [89] developed an algorithm for computing the abelian complexity of infinite recurrent strings which are associated with Parry numbers. The latter method can also be used for proving that a certain value of ρ^{ab} is attained infinitely many times.

The relation of the abelian complexity of an infinite string x with the existence of k-powers in x was established in [85], where it was shown that if x has bounded abelian complexity, then it contains abelian k-powers for all $k \geq 1$. Cassaigne et al. [30] characterized the morphisms which map any infinite string to a string with bounded abelian complexity. On the other hand, there are morphisms under which the image of an infinite string x can have unbounded abelian complexity, but this is possible only when x itself has unbounded abelian complexity [30].

Naturally, the abelian complexity of some well-known infinite strings has been explored. For Sturmian strings, Coven and Hedlund [36] proved that $\rho^{ab}(n) = 2$ for all $n \geq 1$. The abelian complexity of the Thue-Morse string is $\rho^{ab}(n) = 2$ for n odd, and $\rho^{ab}(n) = 3$ for n even [85]. Richomme et al. [84] studied the abelian complexity of the Tribonacci string and showed that $\rho^{ab}(n)$ attains all values in $\{3, 4, 5, 6, 7\}$ for $n \geq 1$, and moreover the values 3 and 7 are attained infinitely often. Later, [89] proved that the values $4, 5, 6$ are also attained infinitely often in a Tribonacci string. Turek [88] computed the optimal bound on the abelian complexity of a special class of strings on a ternary alphabet.

7.1 Open problems

Problem 23 ([85]) *Does every uniformly recurrent infinite string with bounded abelian complexity begin with an abelian k-power for each positive integer k?*

Problem 24 ([84]) *In the Tribonacci string, for each value $m \in \{4, 5, 6, 7\}$, characterize those n for which $\rho^{ab}(n) = m$.*

Problem 25 ([84]) *Prove or disprove that the k-bonacci string is $(k-1)$-balanced.*

Problem 26 *Characterize the abelian complexity of the k-bonacci string.*

8 Applications

The abelian theory of strings and the associated notion of Parikh vectors have been used in a number of applications. Below, we present such applications from various fields.

8.1 Bioinformatics

DNA sequences contain biologically important sites, whose functionality in the organism depends mostly on the *content*, the characters, that appear in the site, and less so on the *order* of these characters. Such sites include, for instance, the *isochores* —where it is known that `GC`-rich isochores exhibit greater gene density—, `CpG` *islands* —sequences where the `C+G` content is larger than 50%—, or the *protein binding sites* [10]. Benson [10] devised abelian local and global alignment algorithms (see Section 3.4) and applied them on a set of 1796 human promoter sequences from the Eukaryotic Promoter Database (EPD). He showed that abelian alignment can give a better insight on the similarity of biologically related sequences, in comparison to classical (non-abelian) alignment algorithms.

At a higher level, the DNA can be considered as a sequence of genes which appear in varying orders in different genomes. It is believed that functionally related genes often appear next to each other, although perhaps in different order, in the genomes. Hence, an interesting biological problem is that of finding clusters of genes that frequently appear together. Eres et al. [53] translated the problem in that of *abelian pattern discovery* (see Section 3.5) and run their algorithm on a set of 8,394 *E. Coli* sequences, where they discovered a number of maximal patterns. Parida [78] extended the work of [53] to allow for gaps —a useful extension for handling noisy and incomplete data— and applied her algorithm for functional classification of genes and proteins, and for the construction of phylogeny of the genomes.

Böcker [17] used Parikh vectors (called *compomers* in their paper) for *Single Nucleotide Polymorphism* (SNP) discovery using base-specific cleavage and mass spectrometry. Mass spectrometry is a method for determining the molecular mass of the input molecules. The order of the bases or the amino acids does not influence the total mass of the molecule and thus one can concentrate on the multiplicity of each base or amino acid (Parikh vectors). Similarly, Böcker and Lipták [18] utilized Parikh vectors for protein identification using mass spectrometry data.

8.2 Pattern matching

Grossi and Luccio [63] devised an algorithm for the k-mismatches approximate pattern matching problem. In the preprocessing phase, the algorithm scans the text x to find locations that could potentially match the pattern $y = y[1\,..\,m]$, and then checks for actual matches. As potential matches are considered those m-length substrings of x whose Parikh vectors differ from the Parikh vector of y in at most k elements.

8.3 Games

In [21], the following Scrabble-like game was described. A random text is chosen (e.g. from a newspaper). Each player draws 8 letters from the sack of letters. When it is her turn, she aligns a word made with her letters to a position in the random text, trying to maximize the total score, which is the sum of the scores of the individual letters used. Then she draws new letters from the sack, until she has again 8 letters in front of her. Overlapping alignments are allowed, so the same position in the text can be matched more than once. The game ends when all letters finish, or when no player can move, and is won by the player with the highest total score. If we refer to the player's current Parikh vector (the contents of the tray) as $\mathcal{P}$, then the task is to find an abelian match of a sub-Parikh vector $\mathcal{P}' \subseteq \mathcal{P}$ to the text (see Section 3.5),

under the constraint that the substring matched be a word of English. At any point in the game, players want to maximize the score of $\mathcal{P}'$.

References

1. A. Aberkane, J. Currie, and N. Rampersad: *The number of ternary words avoiding abelian cubes grows exponentially.* Journal of Integer Sequences, 7(2) 2004, p. 3.
2. A. Amir, A. Apostolico, G. M. Landau, and G. Satta: *Efficient text fingerprinting via Parikh mapping.* Journal of Discrete Algorithms, 1(5–6) 2003, pp. 409–421.
3. A. Apostolico, C. S. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin: *Parallel construction of a suffix tree with applications.* Algorithmica, 3 1988, pp. 347–365.
4. J.-M. Autebert, J. Berstel, and L. Boasson: *Handbook of formal languages*, vol. 1, Springer-Verlag New York, Inc., New York, NY, USA, 1997, ch. Context-free languages and pushdown automata, pp. 111–174.
5. S. Avgustinovich and A. Frid: *Words avoiding abelian inclusions.* Journal of Automata, Languages and Combinatorics, 7(1) 2001, pp. 3–9.
6. 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.
7. S. Avgustinovich, J. Karhumaki, and S. Puzynina: *On abelian versions of critical factorization theorem.* Proceedings of the 13th Mons. Theoretical Computer Science Days, 2010.
8. G. Badkobeh, G. Fici, S. Kroon, and Z. Lipták: *Binary jumbled string matching: Faster indexing in less space.* CoRR, abs/1206.2523 2012.
9. L. Balková, K. Binda, and O. Turek: *Abelian complexity of infinite words associated with quadratic parry numbers.* Theoretical Computer Science, 412(45) Oct. 2011, pp. 6252–6260.
10. G. Benson: *Composition alignment*, in Proceedings of the Third International Workshop in Algorithms in Bioinformatics (WABI), G. Benson and R. D. M. Page, eds., vol. 2812 of Lecture Notes in Computer Science, Budapest, Hungary, September 15–20 2003, Springer, pp. 447–461.
11. J. Berstel: *Some recent results on squarefree words*, in Proceedings of the Symposium of Theoretical Aspects of Computer Science, Springer-Verlag, 1984, pp. 14–25.
12. F. Blanchet-Sadri, J. Kim, R. Mercaş, W. Severa, and S. Simmons: *Abelian square-free partial words.* Language and Automata Theory and Applications, 2010, pp. 94–105.
13. F. Blanchet-Sadri, J. Kim, R. Mercaş, W. Severa, S. Simmons, and D. Xu: *Avoiding abelian squares in partial words.* Journal of Combinatorial Theory, Series A, 119(1) 2012, pp. 257–270.
14. F. Blanchet-Sadri, S. Simmons, and D. Xu: *Abelian repetitions in partial words.* Advances in Applied Mathematics, 2011.
15. F. Blanchet-Sadri, A. Tebbe, and A. Veprauskas: *Abelian periods on partial words.* Available online at: `http://www.uncg.edu/cmp/research/finewilf6/`.
16. F. Blanchet-Sadri, A. Tebbe, and A. Veprauskas: *Fine and Wilfs theorem for abelian periods in partial words*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.
17. S. Böcker: *SNP and mutation discovery using base-specific cleavage and MALDI-TOF mass spectrometry.* Bioinformatics, 19(suppl 1) 2003, pp. i44–i53.
18. S. Böcker and Z. Lipták: *A fast and simple algorithm for the money changing problem.* Algorithmica, 48(4) 2007, pp. 413–432.
19. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communnications of the ACM, 20(10) 1977, pp. 762–772.
20. E. Bullock: *Improved bounds on the length of maximal abelian square-free words.* Journal of Combinatorics, 11(1) 2004, p. 17.
21. P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták: *On table arrangements, scrabble freaks, and jumbled pattern matching*, in Fun with Algorithms, Springer, 2010, pp. 89–101.
22. P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták: *Algorithms for jumbled pattern matching in strings.* International Journal of Foundations of Computer Science, 2011.
23. P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták: *On approximate jumbled pattern matching in strings.* Theory of Computing Systems, 2012, pp. 1–17.

24. A. Butman, R. Eres, and G. Landau: *Permuted and scaled string matching*, in String Processing and Information Retrieval, A. Apostolico and M. Melucci, eds., vol. 3246 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2004, pp. 305–332.
25. A. Butman, R. Eres, and G. Landau: *Scaled and permuted string matching.* Information processing letters, 92(6) 2004, pp. 293–297.
26. D. Callan: *Card deals, lattice paths, abelian words and combinatorial identities.* ArXiv e-prints, Dec. 2008, arXiv:0812.4784v1.
27. A. Carpi: *On the number of abelian square-free words on four letters.* Discrete Applied Mathematics, 81(1) 1998, pp. 155–167.
28. A. Carpi: *On abelian squares and substitutions.* Theoretical Computer Science, 218(1) 1999, pp. 61–81.
29. J. Cassaigne, S. Ferenczi, and L. Zamboni: *Imbalances in Arnoux-Rauzy sequences*, in Annales de l'institut Fourier, vol. 50, Chartres: L'Institut, 1950-, 2000, pp. 1265–1276.
30. J. Cassaigne, G. Richomme, K. Saari, and L. Q. Zamboni: *Avoiding abelian powers in binary words with bounded abelian complexity.* International Journal of Foundations of Computer Science, 22(4) 2011, pp. 905–920.
31. M. Christodoulakis, M. Christou, M. Crochemore, and C. S. Iliopoulos: *Abelian borders in words.* Submitted for publication in 2012.
32. M. Christou, M. Crochemore, and C. Iliopoulos: *Identifying all abelian periods of a string in quadratic time and relevant problems.* International Journal of Foundations of Computer Science, 2012, (accepted).
33. F. Cicalese, G. Fici, and Z. Lipták: *Searching for jumbled patterns in strings*, in Proceedings of the Prague Stringology Conference (PSC 2009), 2009.
34. F. Cicalese, E. S. Laber, O. Weimann, and R. Yuster: *Near linear time construction of an approximate index for all maximum consecutive sub-sums of a sequence*, in Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching, J. Kärkkäinen and J. Stoye, eds., vol. 7354 of Lecture Notes in Computer Science, Helsinki, Finland, jul 2012, Springer, pp. 149–158.
35. S. Constantinescu and L. Ilie: *Fine and Wilf's theorem for abelian periods.* Bulletin of the EATCS, 89 2006, pp. 167–170.
36. E. M. Coven and G. A. Hedlund: *Sequences with minimal block growth.* Mathematical Systems Theory, 7(2) 1973, pp. 138–153.
37. L. Cummings: *Gray codes and strongly square-free strings.* Sequences II: methods in communication, security and computer science, 1993, p. 439.
38. L. Cummings: *Strongly square-free strings on three letters.* Australasian Journal of Combinatorics, 14 1996, pp. 259–266.
39. L. Cummings and M. Mays: *A one–sided zimin construction.* The Electronic Journal of Combinatorics, 8(R27) 2001, p. 1.
40. L. J. Cummings and W. F. Smyth: *Weak repetitions in strings.* Journal of Combinatorial Mathematics and Combinatorial Computing, 24 1997, pp. 33–48.
41. J. Currie: *The number of binary words avoiding abelian fourth powers grows exponentially.* Theoretical Computer Science, 319(1) 2004, pp. 441–446.
42. J. Currie and A. Aberkane: *A cyclic binary morphism avoiding abelian fourth powers.* Theoretical Computer Science, 410(1) 2009, pp. 44–52.
43. J. Currie and V. Linek: *Avoiding patterns in the abelian sense.* Canadian Journal of Mathematics, 53(4) 2001, pp. 696–714.
44. J. Currie and N. Rampersad: *Recurrent words with constant abelian complexity.* Advances in Applied Mathematics, 47 2011, pp. 116–124.
45. J. Currie and T. Visentin: *On abelian 2-avoidable binary patterns.* Acta Informatica, 43(8) 2007, pp. 521–533.
46. J. Currie and T. Visentin: *Long binary patterns are abelian 2-avoidable.* Theoretical Computer Science, 409(3) 2008, pp. 432–437.
47. F. Dekking: *Strongly non-repetitive sequences and progression-free sets.* Journal of Combinatorial Theory, Series A, 27(2) 1979, pp. 181–185.
48. M. Domaratzki and N. Rampersad: *Abelian primitive words*, in Proceedings of the 15th International Conference on Developments in Language Theory, G. Mauri and A. Leporati, eds., vol. 6795 of Lecture Notes in Computer Science, Springer, 2011, pp. 204–215.

49. T. Ejaz: *Abelian pattern matching in strings*, PhD thesis, Technische Universität Dortmund, 2010.
50. T. Ejaz, S. Rahmann, and J. Stoye: *Online abelian pattern matching*, tech. rep., Technische Universität Dortmund, 2008.
51. R. Entringer, D. Jackson, and J. Schatz: *On nonrepetitive sequences.* Journal of Combinatorial Theory, Series A, 16(2) 1974, pp. 159–164.
52. P. Erdős: *Some unsolved problems*, Magyar Tudományos Akadémia Matematikai Kutató Intézete, 1961.
53. R. Eres, G. M. Landau, and L. Parida: *Permutation pattern discovery in biosequences.* Journal of Computational Biology, 11(6) 2004, pp. 1050–1060.
54. A. Evdokimov: *Strongly asymmetric sequences generated by a finite number of symbols*, in Dokl. Akad. Nauk SSSR, vol. 179, 1968, pp. 1268–1271.
55. A. Evdokimov: *The existence of a basis that generates 7-valued iteration-free sequences.* Diskret. Analiz, 18 1971, pp. 25–30.
56. A. Evdokimov and S. Kitaev: *Crucial words and the complexity of some extremal problems for sets of prohibited words*, tech. rep., Chalmers University of Technology and Göteborg University, 2001.
57. A. Evdokimov and S. Kitaev: *Crucial words and the complexity of some extremal problems for sets of prohibited words.* Journal of Combinatorial Theory, Series A, 105(2) 2004, pp. 273–289.
58. G. Fici, T. Lecroq, A. Lefebvre, and É. Prieur-Gaston: *Computing abelian periods in words*, in Proceedings of the Prague Stringology Conference (PSC 2011), J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2011, pp. 184–196.
59. A. Fraenkel, J. Simpson, and M. Paterson: *On weak circular squares in binary words*, in Combinatorial Pattern Matching, Springer, 1997, pp. 76–82.
60. A. Fraenkel, J. Simpson, and M. Paterson: *On abelian circular squares in binary words*, in Paul Erdos and his Mathematics II, vol. 11 of Bolyai Soc., Mathematical Studies, Budapest, 2002, pp. 329–338, Available online at: `http://maths.curtin.edu.au/local/docs/simpson/Abeliancircular.ps`.
61. A. Glen, B. Halldórsson, and S. Kitaev: *Crucial words for abelian powers*, in Proceedings of the 13th International Conference on Developments in Language Theory, Springer-Verlag, 2009, pp. 264–275.
62. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM/SIAM, 2003, pp. 841–850.
63. R. Grossi and F. Luccio: *Simple and efficient string matching with k mismatches.* Information Processing Letters, 33(3) 1989, pp. 113–120.
64. R. N. Horspool: *Practical fast searching in strings.* Software - Practice and Experience (SPE), 10(6) 1980, pp. 501–506.
65. M. Huova and J. Karhumäki: *Observations and problems on k-abelian avoidability*, in Combinatorial and Algorithmic Aspects of Sequence Processing (Dagstuhl Seminar 11081), M. Crochemore, L. Kari, M. Mohri, and D. Nowotka, eds., vol. 1, Dagstuhl, Germany, 2011, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, arXiv:1104.4273v1 [math.CO].
66. M. Huova, J. Karhumäki, and A. Saarela: *Problems in between words and abelian words: k-abelian avoidability.* Theoretical Computer Science, 2012.
67. A. Karaman: *Weak repetitions in Sturmian strings.* Theoretical Computer Science, 290(3) 2003, pp. 2137–2146.
68. J. Karhumäki, S. Puzynina, and A. Saarela: *Fine and Wilfs theorem for k-abelian periods*, in Developments in Language Theory, Springer, 2012, pp. 296–307.
69. V. Keränen: *Abelian squares are avoidable on 4 letters.* Automata, Languages and Programming, 1992, pp. 41–52.
70. V. Keränen: *New abelian square-free DT0L-languages over 4 letters*, in Proceedings of the International Arctic Seminar, Murmansk, Russia, May 2002.
71. 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.
72. M. Korn: *Maximal abelian square-free words of short length.* Journal of Combinatorial Theory, Series A, 102(1) 2003, pp. 207–211.
73. M. Main: *Permutations are not context-free: An application of the interchange lemma.* Information Processing Letters, 15(2) 1982, pp. 68–71.

74. T. Moosa and M. Rahman: *Indexing permutations for binary strings.* Information Processing Letters, 110(18-19) 2010, pp. 795–798.
75. T. Moosa and M. Rahman: *Sub-quadratic time and linear space data structures for permutation matching in binary strings.* Journal of Discrete Algorithms, 10 2011, pp. 5–9.
76. M. Morse and G. Hedlund: *Symbolic dynamics II. Sturmian trajectories.* American Journal of Mathematics, 62(1) 1940, pp. 1–42.
77. W. Ogden, R. Ross, and K. Winklmann: *An "interchange lemma" for context-free languages.* SIAM Journal on Computing, 14(2) 1985, pp. 410–415.
78. L. Parida: *Gapped permutation patterns for comparative genomics.* Algorithms in Bioinformatics, 4175 2006, pp. 376–387.
79. R. Parikh: *On context-free languages.* Journal of the ACM, 13(4) 1966, pp. 570–581.
80. P. Pleasants: *Non-repetitive sequences*, in Mathematical Proceedings of the Cambridge Philosophical Society, vol. 68, Cambridge Univ Press, 1970, pp. 267–274.
81. G. Rauzy: *Suites àtermes dans un alphabet fini.* Séminaire de Théorie des nombres de Bordeaux, 25 1983, pp. 1–16.
82. L. Richmond and J. Shallit: *Counting abelian squares.* The Electronic Journal of Combinatorics, 16(1) 2009, p. R72.
83. G. Richomme, K. Saari, and L. Zamboni: *Standard words and abelian powers in Sturmian words*, in Proceedings of the 12th Mons Theoretical Computer Science Days (Mons Days of Theoretical Computer Science), 2008.
84. G. Richomme, K. Saari, and L. Zamboni: *Balance and abelian complexity of the tribonacci word.* Advances in Applied Mathematics, 45(2) 2010, pp. 212–231.
85. G. Richomme, K. Saari, and L. Q. Zamboni: *Abelian complexity in minimal sub-shifts.* ArXiv e-prints, 2009, arXiv:0911.2914v1 [math.CO].
86. A. Saarela: *Ultimately constant abelian complexity of infinite words.* Journal of Automata, Languages and Combinatorics, 14(255–258) 2009, p. 40.
87. A. Thue: *Über unendliche zeichenreihen.* Norske Vid. Skrifter I Mat.-Nat. Kl., 1 1906, pp. 1–22.
88. O. Turek: *Balances and abelian complexity of a certain class of infinite ternary words.* RAIRO-Theoretical Informatics and Applications, 44(03) 2010, pp. 313–337.
89. O. Turek: *Abelian complexity and abelian co-decomposition.* ArXiv e-prints, 2012, arXiv:1201.2109v1 [math.CO].
90. A. Zimin: *Blocking sets of terms.* Matematicheskii Sbornik, 161(3) 1982, pp. 363–375.
91. A. Zimin: *Blocking sets of terms.* Mathematics of the USSR-Sbornik, 47 1984, p. 353.

On Factor Storacles: an Alternative to Factor Oracles?

Loek Cleophas[1] and Bruce W. Watson[2]

[1] Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands loek@fastar.org
[2] FASTAR Research Group, Department of Information Science, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa bruce@fastar.org

Abstract. In [1], the *factor oracle* is presented as a data structure for weak factor recognition. It is an automaton built on a string p of length m that is acyclic, recognizes at least all factors of p, has $m+1$ states which are all final, is homogeneous, and has m to $2m-1$ transitions. At the end of [4,6], we gave an example of an alternative automaton having the five properties mentioned, and having less transitions than the factor oracle for the same string. In the current paper, we present a construction algorithm for the alternative automaton. It turns out that this alternative automaton, which we call *factor storacle* (for *shortest forward transition factor oracle*), has a smaller number of transitions in some, but a larger number of transitions in other cases—and somewhat surprisingly, that it does not strictly satisfy one of the properties of the factor oracle. This brings up a number of interesting future research questions related to factor storacles and their relation to factor oracles.

Keywords: factor oracle, finite automaton, weak factor recognition, algorithm derivation, pattern matching

1 Introduction

In [1], the *factor oracle* is presented as a data structure for weak factor recognition. It is an automaton built on a string p of length m that (a) is acyclic, (b) recognizes at least all factors of p, (c) has $m+1$ states (which are all final), and (d) has m to $2m-1$ transitions (cf. [1]). In addition, (e) the resulting automaton is homogeneous, i.e. for every state, all of its incoming transitions are on the same symbol. Some example factor oracles are given in Figures 1 and 2. Factor oracles are introduced in [1] as

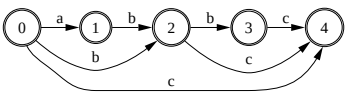

Figure 1. Factor oracle (with initial state 0) for *abbc* (recognizing $abc \notin \mathbf{fact}(p)$)

an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. In such algorithms, a window on a text is read backward while attempting to match a keyword factor. When this fails, the window is shifted using the information on the longest factor matched and the mismatching character.

Instead of an automaton recognizing exactly the set of factors of the keyword, it is possible to use a factor oracle: although it recognizes more strings than just the factors and thus might read backwards longer than necessary, it cannot miss any matches. The advantage of using factor oracles is that they are easier to construct and

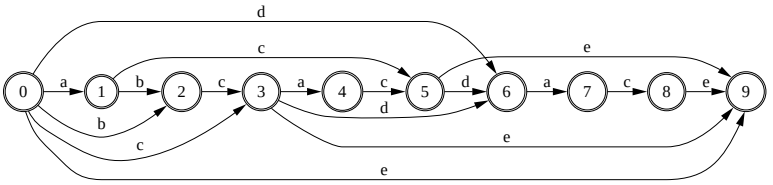

Figure 2. Factor oracle (with initial state 0) recognizing a superset of **fact**(p) (including for example $cace \notin$ **fact**(p)), for $p = abcacdace$

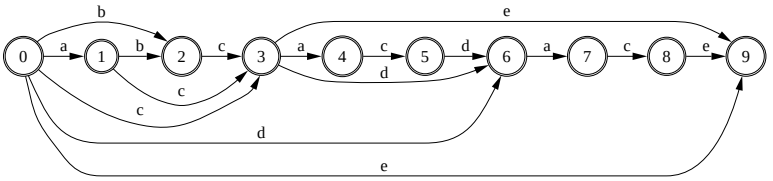

Figure 3. Alternative automaton (with initial state 0) with $m + 1$ states satisyfing Glushkov's property yet recognizing a *different* superset of **fact**(p) than the factor oracle for p (including for example $acacdace \notin$ **factstoracle**(p), but not $cace$) and having less transitions, for $p = abcacdace$

take less space to represent compared to the automata that were previously used in these factor-based algorithms, such as suffix, factor and subsequence automata. This is the result of the latter automata lacking one or more of the essential properties of the factor oracle.

In [4,6], we presented an alternative construction algorithm for factor oracles. This algorithm was based on considering the suffixes of the string p in order of decreasing length. Being an $\mathcal{O}(m^2)$ algorithm, it was not efficient compared to the linear algorithm given in [1], but it made some of the factor oracle's properties immediately obvious (while at the same time making some other properties harder to prove).

At the end of that paper, we gave an example of an alternative automaton for string $p = abcacdace$ having the five properties (a)-(e), yet having less transitions than the factor oracle for the same string. This factor oracle is depicted in Figure 2, while the alternative automaton is depicted in Figure 3. The alternative automaton was in fact obtained when manually trying to create the factor oracle for the string, due to accidentally misapplying the new factor oracle construction algorithm. In Section 2 of the current paper, we present the construction algorithm for the alternative automaton, which we call the *factor storacle* (for *shortest forward transition factor oracle*).

As indicated by the example of Figure 3 and Figure 2, the factor storacle construction algorithm may result in an automaton having less transitions than the factor oracle, and thus requiring even less storage space. Although the algorithm, like the factor oracle one we presented in [4,6], is not linear and will therefore be less efficient than the linear factor oracle construction algorithm, it satisfies most of the aforementioned properties of the factor oracle, as we show in Section 3. The algorithm and its properties had previously been discussed in an e-mail conversation between the first author of this paper and Bob Melichar, but it turned out afterwards that the factor storacle does not always have a smaller number of transitions than the corresponding factor oracle. While our initial assumption was that the number of transitions would satisfy the same upper bound as for the case of factor oracles, namely $2m - 1$, prac-

tical experiments show even this is not the case in general. We demonstrate this and discuss the results.

Finally, Section 4 concludes this short exposition and discusses a number of interesting questions brought up by these results. These questions may give rise to future research on the subject.

1.1 Related Work

As mentioned before, factor oracles were introduced in [1] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. A pattern matching algorithm using the factor oracle is described in that paper as well.

In [4], we presented alternative $\mathcal{O}(m^2)$ construction algorithms for factor oracles. An extended version of that paper appears as [5] and in the Master's thesis [7, Chapter 4]. In those versions, some properties of the language of a factor oracle are discussed as well. The thesis also discusses pattern matching algorithms—among them those using factor oracles—and the implementation of the factor oracle in the SPARE Time toolkit, a revised and extended version of SPARE Parts [14]. A further extended and revised version of the work appears in [6].

The language of a factor oracle was finally characterized completely in a paper by Mancheron and Moan [12].

Apart from their use in pattern matching algorithms, factor oracles have been used in a heuristic to compute repeated factors of a string [9] as well as to compress text [10]. An improvement for those uses of factor oracles is introduced in [11] in the form of the *repeat oracle*.

Related to the factor oracle, the *suffix oracle*—in which only those states corresponding to a suffix of p are marked final—is introduced in [1]. In [2] the factor oracle is extended to apply to a set of strings. The conversion of suffix trees into suffix or factor oracles is discussed by Rusu in [13]. Crochemore et al. show how factor oracles can be obtained directly from the trie of the factors of a keyword [8]. In [3], the authors present a statistical average-case analysis on the size of factor and suffix oracles.

1.2 Preliminaries

A *string* $p = p_1...p_m$ of length m is a sequence of characters from an alphabet V. A string u is a *factor* (resp. *prefix*, *suffix*) of a string v if $v = sut$ (resp. $v = ut$, $v = su$), for $s, t \in V^*$. We will use $\mathbf{pref}(p)$, $\mathbf{suff}(p)$ and $\mathbf{fact}(p)$ for the set of prefixes, suffixes and factors of p respectively. A prefix (resp. suffix or factor) is a *proper* prefix (resp. suffix or factor) of a string p if it does not equal p. We write $u \leq_s v$ to denote that u is a suffix of v, and $u <_s v$ to denote that u is a proper suffix of v.

2 Construction of the Factor Storacle

Our factor storacle construction algorithm constructs a 'skeleton' automaton for p—recognizing $\mathbf{pref}(p)$—and then constructs a path for each of the suffixes of p in order of decreasing length, such that eventually at least $\mathbf{pref}(\mathbf{suff}(p)) = \mathbf{fact}(p)$ is recognized. If such a suffix of p is already recognized, no transition needs to be constructed. If on the other hand the complete suffix is not yet recognized there is a longest prefix of such a suffix that is recognized.

A transition on the next, non-recognized symbol is then created, from the state in which this longest prefix of the suffix is recognized, to the next state from that state onward that has an incoming transition on the non-recognized symbol, i.e. the shortest forward transition that keeps the automaton homogeneous.

This procedure of creating transitions is repeated while the complete suffix is not yet recognized.

Build_Storacle($p = p_1p_2...p_m$)

1: **for** i from 0 to m **do**
2: Create a new final state i
3: **for** i from 0 to $m-1$ **do**
4: Create a new transition from i to $i+1$ by p_{i+1}
5: **for** i from 2 to m **do**
6: Let the longest path from state 0 that spells a prefix of $p_i...p_m$ end in state j and spell out $p_i...p_k$ $(i-1 \leq k \leq m)$
7: **while** $k \neq m$ **do**
8: Let the first state from state j onward that has an incoming transition on p_{k+1} be state l $(j < l <= k+1)$
9: Build a new transition from j to l by $p_l (= p_{k+1})$
10: Let the longest path from state 0 that spells a prefix of $p_i...p_m$ end in state j and spell out $p_i...p_k$ $(i-1 \leq k \leq m)$

The factor storacle on p built using this algorithm is referred to as Storacle(p) and the language recognized by it as **factstoracle**(p).

The difference between this algorithm and the $\mathcal{O}(m^2)$ factor oracle construction algorithm of [4,6] originates from the choice of the target of the first (if any) newly created transition for each suffix:

- Here, that transition leads to the next state (from a particular state onward) that has an incoming transition on the non-recognized symbol. This procedure may then need to be repeated for further symbols of the suffix to be recognized.
- In the case of the factor oracle construction, the newly created transition leads to the unique state from which the remainder of the suffix leads to the last state of the automaton—thus immediately guaranteeing that the entire suffix is recognized.

3 Properties of Factor Storacles

In this section, we first prove that properties (a)–(c) and (e) as mentioned in the introduction hold for factor storacles just as they do for factor oracles.

Property 1. Storacle(p) is an acyclic automaton.
Proof: Oracle(p) is an acylic automaton, as proven in [1] and [4,6]. The above algorithm is a modification of algorithm **Build_Oracle_2** given in Section 2 of [4]. It creates transitions to the first state from state j onward that has an incoming transition on p_{k+1} (which may happen to be state $k+1$), instead of to state $k+1$ as in algorithm **Build_Oracle_2**. The factor storacle therefore recognizes a factor in the same or an earlier state than the corresponding factor oracle. This means that there is always a state from such a state j onward that has an incoming transition on p_{k+1}, and therefore only forward transitions are created. □

Property 2. **fact**$(p) \subseteq$ **factstoracle**(p).
Proof: The algorithm constructs a path for all suffixes of p and all states are final. □

Property 3. For p of length m, Storacle(p) has exactly $m+1$ states.
Proof: States can be constructed in steps 1-2 only, and exactly $m+1$ states are constructed there. □

Property 4 (Homogeneousness). All transitions reaching a state i of Storacle(p) are labeled by p_i.
Proof: The only steps of the algorithm that create transitions are steps 4 and 9. In step 4, transitions to a state $i+1$ are created labeled by p_{i+1}. In step 9, transitions to a state l which has incoming transitions on $p_l (= p_{k+1})$ are created, labeled by p_l. □

Furthermore, like for factor oracles, we have the following obvious property for factor storacles:

Property 5 (Weak determinism). For each state of Storacle(p), no two outgoing transitions of the state are labeled by the same symbol.
Proof: The algorithm never creates an outgoing transition by some symbol if such a transition already exists. □

This leaves one of the original factor oracle's properties, property (d), stating that it has between m and $2m-1$ transitions, open for the factor storacle. The factor oracle construction algorithm of [4,6], as should be clear from the discussion in the preceding section, creates at most $m-1$ extra transitions on top of the m transitions created initially to recognize the keyword itself. This yields the factor oracle's $2m-1$ upper bound on the number of transitions.

As became clear in the preceding section though, the factor storacle construction algorithm we presented might create multiple transitions per keyword suffix to be recognized. We can easily give a very coarse upper bound on the total number of transitions of the factor storacle:

Property 6. For p of length m, Storacle(p) has between m and $m(m+1)/2$ transitions.
Proof: The lower bound follows from the second **for**-loop of the algorithm. Disregarding any properties of the keyword and alphabet used (except for the keyword's length m), an upper bound of $m(m+1)/2$ can be proven in at least two ways. Firstly, the sum of the lengths of all the suffixes of a keyword of length m, including the keyword itself, equals $m(m+1)/2$. Secondly, since all transitions are forward transitions, and the factor storacle is kept homogeneous, there can be at most one transition between each pair of states, hence at most $(|Q|-1)|Q|/2$ in total, and this equals $m(m+1)/2$ since $m = |Q| - 1$. □

When first discovering the factor storacle however, we formed the conjecture that the upper bound on the number of transitions would be the same $2m-1$ linear bound as for the factor oracle. It turns out that this is not the case—as perhaps the fact that multiple transitions may be created per suffix is an indication of. Experiments generating all keywords of length m out of an alphabet of size $|m|$ (modulo renaming of alphabet symbols) show that

- from $m = 5..7$ the upper bound is exactly $2m$, reached for e.g. keyword *abaac*;
- for $m = 8$ it becomes $2m+1$, e.g. for *abaacbbd*;
- for $m = 9$ it becomes $2m+2$, e.g. for *abacbbaad*;

- for $m = 10$ it becomes $2m + 3$, e.g. for *abacbaabbd*;
- for $m = 11$ it becomes $2m + 4$, e.g. for *abacbabbaad*;
- for $m = 12$ it becomes $2m + 5$, e.g. for *abcabaaccbbd*, as depicted in Figure 4.

These experiments seem to suggest the upper bound is linear, but this remains to be proven. Note that the example given for keywords of length $m = 12$, *abcabaaccbbd*, also shows a situation where the number of transitions of the corresponding factor oracle does not equal the upper bound for the factor oracle: for this keyword, the factor oracle has just 22 i.e. $2m - 2$ transitions.

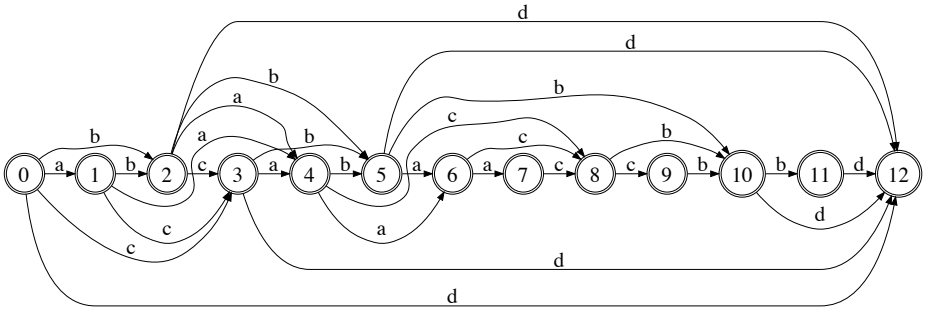

Figure 4. Factor storacle (with initial state 0) for $p = abcabaaccbbd$ with $|p| = m = 12$, having $2m + 5 = 29$ transitions

Conjecture 7. For p of length m, Storacle(p) has a linear number of transitions.

Note that a coarse bound on the time complexity for the algorithm is $\mathcal{O}(m^3)$ as the operations on line 6 and 10 can be implemented using a **while** loop, but we expect that as for factor oracles, more efficient algorithms for constructing a factor storacle exist.

4 Conclusions and Future Work

We have presented a construction algorithm for factor storacles and shown that such automata satisfy most of the properties that factor oracles satisfy. While we originally discovered the factor storacle by accident for a keyword for which it ends up having less transitions than the corresponding factor oracle, we have shown that the latter is not true in general. Furthermore, the upper bound of $2m - 1$ transitions for the factor oracle does not hold for the factor storacle: experiments constructing factor storacles for all keywords of sizes up to $m = 12$ seem to indicate an upper bound of at most $3m$, although this is clearly a conjecture.

The results in this paper thus give rise to a number of interesting research questions:

- Is the factor storacle still linear in the number of transitions, with an upper bound of $3m$ or otherwise? And if this is not the case, is there a straightforward characterization of the set of keywords for which it is linear?
- Does a more efficient construction algorithm for factor storacles exist, in parallel to factor oracle construction algorithms that are more efficient than the one from which our factor storacle construction algorithm was derived?
- Does an (efficient) algorithm exist which creates an automaton satisying properties (a)-(e) that is minimal among all such automata?

We challenge the reader to tackle one or more of these questions, and hope to see future publications answering such questions—perhaps authored by members of the Prague Stringology Club including Bob Melichar.

References

1. Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Efficient Experimental String Matching by Weak Factor Recognition. In *Proceedings of the 12th conference on Combinatorial Pattern Matching*, volume 2089 of *LNCS*, pages 51–72, 2001.
2. Cyril Allauzen and Mathieu Raffinot. Oracle des facteurs d'un ensemble de mots. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.
3. Jérémie Bourdon and Irena Rusu. Statistical properties of factor oracles. *Journal of Discrete Algorithms*, 9(1):57–66, March 2011.
4. Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing Factor Oracles. In *Proceedings of the Prague Stringology Conference 2003*. Department of Computer Science and Engineering, Czech Technical University, Prague, September 2003.
5. Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing Factor Oracles. Technical Report 04/01, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, January 2004.
6. Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. Constructing Factor Oracles. *Journal of Automata, Languages and Combinatorics*, 10(5/6):627–640, 2005.
7. Loek G. W. A. Cleophas. Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms. Master's thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, August 2003.
8. Maxime Crochemore, Lucian Ilie, and Emine Seid-Hilmi. The structure of factor oracles. *International Journal of Foundations of Computer Science*, 18(4):781–797, August 2007.
9. Arnaud Lefebvre and Thierry Lecroq. Computing repeated factors with a factor oracle. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop on Combinatorial Algorithms*, pages 145–158, 2000.
10. Arnaud Lefebvre and Thierry Lecroq. Compror: on-line losless data compression with a factor oracle. *Inf. Process. Lett.*, 83(1):1–6, 2002.
11. Arnaud Lefebvre, Thierry Lecroq, and J. Alexandre. Drastic improvements over repeats found with a factor oracle. In E. Billington, D. Donovan, and A. Khodkar, editors, *Proceedings of the 13th Australasian Workshop on Combinatorial Algorithms*, pages 253–265, 2002.
12. Alban Mancheron and Christophe Moan. Combinatorial Characterization of the Language Recognized by Factor and Suffix Oracles. In *Proceedings of the Prague Stringology Conference 2004*. Department of Computer Science and Engineering, Czech Technical University, Prague, August 2004.
13. Irena Rusu. Converting suffix trees into factor/suffix oracles. *Journal of Discrete Algorithms*, 6(2):324–340, June 2008.
14. Bruce W. Watson and Loek Cleophas. SPARE Parts: A C++ toolkit for String PAttern REcognition. *Software—Practice & Experience*, 34(7):697–710, June 2004.

Algorithmics of Repetitions, Local Periods and Critical Factorization Revisited

Maxime Crochemore[1,3⋆], Tomasz Kociumaka[2], Wojciech Rytter[2,4], Chalita Toopsuwan[1], Wojciech Tyczyński[2], and Tomasz Waleń[2]

[1] King's College London, UK
[2] University of Warsaw, Poland
[3] Université Paris-Est, France
[4] University Copernicus, Poland

Abstract. Maximal periodicities, called runs, capture consecutive repetitions in strings. We design a direct algorithm to compute them all for a string drawn from an infinite alphabet. In the associated computation model letter comparisons are done with an equality operator only. On a string of length n, the algorithm runs optimally in time $O(n \log n)$ although there is a linear number of runs.
Under the same hypothesis, we also design a time-optimal algorithm to compute all the local periods of a string, which additionally produces all its critical factorisations.
None of the above algorithms depend on an ordering of the alphabet. They show the power of the concept of a Prefix table associated with a string for the design of string algorithms.
We also design a simple algorithm based on the Dictionary of Basic Factor of the input string.

1 Introduction

The notion of repetitions in strings is related to the fundamental areas of Computer Science, such as word combinatorics, pattern matching, data compression and in general text algorithms. It also extend to application of these subject to the analysis of biological sequences and music sequences for example.

The study of repetitions and other structures of periodicity have been considered since the beginning of last century. A sample of methods and algorithm for detecting repetitions can be found in [8, Chapter 7] and [6, Chapter 8]. A run is a non-extensible occurrence of a repetition, that is, a maximal periodicity in the string. Main [12] consider leftmost periodicities, the concept of run is by Iliopoulos et al. [9], and the most significant algorithmic contribution is by Kolpakov and Kucherov [11]. It is known the number of runs in a string is linear with respect to the length of the string (see [11,14,3]). The exact bounds on the number of runs in a string is a fascinating open problem, especially interesting since the the number of repetitions can be $\Omega(n \log n)$.

The local periods of a string, like its runs, capture the repetitions in the string. They have to do with its critical factorisations or critical positions, where the local period equals the global smallest period of the string. The computation of local periods is known to be done in linear time using a data structure that depends on an alphabet ordering [7]. There are other algorithms for computing such repetitions or local periods of words in linear time, however all these algorithm require again and ordered alphabet. One example is presented in [4]), where instead of suffix trees,

⋆ Corresponding author: Maxime.Crochemore@kcl.ac.uk

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

authors utilise suffix arrays. In term of local periods, they create an algorithm as powerful as in [7] but in a simpler way applying the solution of the Manhattan Skyline Problem.

In this paper we present an algorithm for computing all runs in a length-n string y in time $O(n \log n)$ in the model of computation where the comparison of letters is done with the equality operator only. This gives the same running time to compute local periods and all critical factorisations of the string y on an unordered alphabet. This $O(n \log n)$-time local periods computation is optimal since it implies square detection. The main tool we used in algorithms is the Prefix table of the string.

2 Preliminaries

In this section we will introduce the terminology that will be used in this paper.

Let $y = y[0\,..\,n-1]$ be the string of length $|y| = n$. We say that y has period p, $1 \leq p \leq n$) if $y[i] = y[i+p]$ for all i, $0 \leq i \leq n-p-1$. The exponent of y is the ratio $e = n/p$ where is the smallest period of y. Then, y can be written u^e where u is its prefix of length p, and is called an e-power. It is called a square if its exponent is an even number. A string y is called primitive if it is not an e-power for any integer $e \geq 2$. In other words y is primitive if none of its periods is a divisor of its length.

A string v is a factor of y if $y = uvw$ for some strings u and w. If $i = |u|$ and $j = |uv| - 1$ then $v = y[i]y[i+1]\cdots y[j]$, which is written $y[i\,..\,j]$. The occurrence of $v = y[i\,..\,j]$ at position i on y is called a run (or a maximal periodicity) if its exponent is at least 2 and it is not extensible with the same period. If p is the period of v, the second condition means that $y[i-1] \neq y[i+p-1]$ when $i > 0$ and that $y[j-p+1] \neq y[j+1]$ when $j+1 < n$. Note that the prefix $v[0\,..\,p-1]$ of v is primitive, and then all its conjugates are.

3 Runs in the concatenation of two words

In the next sections we use divide and conquer to compute runs and local periods of a string. Therefore the main element of the algorithm has to do the runs occurring when we concatenate two strings. We follow the approach initiated by Main and Lorentz [13] for detecting squares in strings (see also [1]).

Let $y = uv$ be the concatenation of two strings u and v. In this section we show how to compute all runs of y that start in u and end in v. Since our goal is to design a complete algorithm running on a potentially infinite alphabet, we are allowed to compare letters with an equality operator only.

Let $y[i\,..\,j]$ be a run of period p we are looking for, that is, a run that satisfies $i < |u|$ and $j \geq |u|$. Then the run has a full period in u or in v (or in both), which means that at least one of the following conditions holds: $i \leq |u| - p$ and $j \geq |u| + p$

In the following we deal with runs satisfying the second condition; they have a full period in v. (The other case is symmetric.) We consider all possible periods of the runs, $p = 1, 2, \ldots, |v|$, and extend the prefix $v[0\,..\,p-1]$ of v into a factor of period p as much as possible to the right and the the left. This is done with the help of two precomputed tables, Pref_v and $\mathrm{Pref}_{\tilde{u}\#\tilde{v}\tilde{u}}$.

Recall that, for any nonempty string x, the table Pref_x is defined by

$$\mathrm{Pref}_x[i] = \text{ longest common prefix between } x \text{ and } x[i\,..\,|x|-1]$$

for $i = 0, \ldots |x| - 1$. In the Section 2 of [13] authors show that Pref_x can be computed in time $O(|x|)$ using only the equality operator to compare letters (see also [2, Section 1.6]).

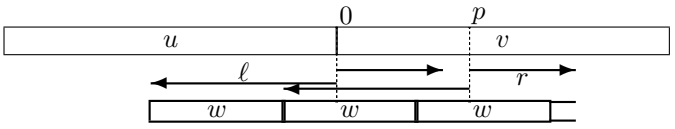

Figure 1. Illustration of period extensions: $r = \mathrm{Pref}_v[p]$ and $\ell = \mathrm{Pref}_{\tilde{u}\#\tilde{v}\tilde{u}}[|uv| - p + 1]$. A run of period p is detected when $\ell + r \geq p$, which guarantees an exponent at least 2

The end position of a potential run of period p in uv is $j = p + r - 1$ where $r = \mathrm{Pref}_v[p]$. The starting position of that potential run is $i = |u| - \ell$ where ℓ is the maximal length of common suffixes between u and $uv[0\,..\,p-1]$, that is $\ell = \mathrm{Pref}_{\tilde{u}\#\tilde{v}\tilde{u}}[|uv| - p + 1]$. The situation is illustrated in Figure 1.

To report the correct period of each run the algorithm has to avoid non-primitive period strings (i.e. non-primitive $v[0\,..\,p-1]$). To do so, it contains an additional feature: periods are marked when they are multiples of previously found periods of runs and they are not treated when examined later.

```
RRIP(u, v)
 1  P ← Pref_v
 2  S ← Pref_{ũ#ṽũ}
 3  unmark all periods 1, 2, ..., |v|
 4  for p ← 1 to |v| do
 5      if p is not marked then
 6          r ← P[p]
 7          ℓ ← S[|uv| − p + 1]
 8          if ℓ + r ≥ p then
 9              process run u[|u| − ℓ .. |u| − 1]v[0 .. p + r − 1] of period p
10              q ← 2p
11              while q ≤ p + r − 1 do
12                  mark period q
13                  q ← q + p
14  return R
```

Proposition 1. *Algorithm* RRIP *applied to string u and v drawn from an infinite alphabet runs in time $O(|uv|)$.*

Proof. The computation of P at line 1 takes time $O(|u|)$ and can be implemented to run on strings drawn from an infinite alphabet. Similarly, the computation of S at line 2 satisfies the same property and takes time $O(|uv|)$.

Note that a period is marked only once: if it were marked twice, the periodicity lemma applied to the corresponding prefix of v would imply that at least one root of run is not primitive, a contradiction. Therefore, the loop at lines 4–13 executes in time $O(|v|)$. Which gives the result.

Algorithm LRIP, dealing with runs having a full period in u, is designed symmetrically to the above algorithm. It has the same property and runs in the same time.

To detect runs in a string y a standard balanced divide-and-conquer method using Algorithms RRIP and LRIP leads to a solution running in time $O(|y| \log |y|)$. However, this solution is not correct because left and right halves of y cannot be processed independently. For example, a run discovered in the first half of y may be the prefix of a run that ends in the second half. Additionally, the solution is not satisfactory if each run has to be processed because some runs may be detected several times.

4 Computing all runs of A string

In this section we how to compute all runs occurring in y and report them once each in time $O(n \log n)$. The solution is built on the above algorithms and cope with the problems raised at the end of the section.

Let y be divided into two halves $u = y[0\,..\,k-1]$ and $v = y[k\,..\,n-1]$ where $k = \lceil n/2 \rceil + 1$. We use the following observation to create an algorithm based on the divide-and-conquer technique.

Observation 1 *Runs of y divides into three categories:*

1. *runs that end at or before position $k-1$,*
2. *runs that start at or after position k,*
3. *runs that start at or before position $k-1$ and end after it.*

To check that runs do expand beyond the ends a given factor we consider the letters preceding its occurrence and following it. To do so, Algorithm Runs has parameters a, z and b. It produces runs in azb which do not involve letter a nor letter b. Assuming letters # and \$ do not belong to the alphabet of y we get runs in y by calling Runs(#, y, \$). Letters a and b are also forwarded to the procedures producing runs in products, RightRunsInProduct and LeftRunsInProduct.

```
Runs(a, z, b)
 1  if |z| ≥ 2 then
 2      k ← ⌊|z|/2⌋ + 1
 3      R ← Runs(a, z[0..k−1], z[k])
 4      R ← R ∪ Runs(z[k−1], z[k..|z|−1], b)
 5      R ← R ∪ RightRunsInProduct(a, z, k, b)
 6      R ← R ∪ LeftRunsInProduct(a, z, k, b)
 7      return R
 8  elsereturn ∅
```

RightRunsInProduct(a, z, k, b)

```
 1  R ← ∅
 2  P ← Pref_{z[k..|z|−1]}
 3  S ← Pref_{z#z̃[0..k−1]}
 4  unmark all periods 1, 2, ..., |z| − k
 5  for p ← 1 to |z| − k do
 6      if p is not marked then
 7          r ← P[k + p]
 8          ℓ ← S[|z| − p + 1]
 9          if ℓ + r ≥ p then
10              if ℓ > 0 and p + r ≥ ℓ
                   and (k − ℓ > 0)
                    or ((k − ℓ = 0) and (a ≠ z[k − ℓ + p − 1]))
                   and (k + p + r < |z| − 1)
                    or ((k + p + r = |z| − 1) and (z[k + r] ≠ b)) then
11                  R ← R ∪ {(k − ℓ, k + p + r − 1, p)}
12              q ← 2p
13              while q ≤ p + r − 1 do
14                  mark period q
15                  q ← q + p
16  return R
```

To avoid reporting false runs or run duplicates, the last two algorithms are updated according to the above observation. This is done at line 10:

- Test $\ell > 0$ ensures the detected run, which has a full period in the right half of z, effectively starts in the left half.
- The role of test $p+r \geq \ell$ is to distinguish runs found with RightRunsInProduct from runs found with LeftRunsInProduct. Note the equivalent comparison in LeftRunsInProduct has to be strict ($>$ instead of $\geq$).
- Test $(k - \ell > 0)$ or $((k - \ell = 0)$ and $(a \neq z[k - \ell + p - 1]))$ ensures that the run does not expand to the left of z.
- Test $(k + p + r < |z| - 1)$ or $((k + p + r = |z| - 1)$ and $(z[k + r] \neq b))$ ensures that the run does not expand to the right of z.

Am I missing something here?

As a consequence of the changes, no run is added twice to the output set and we get the following theorem. The running time deduces from the analysis of the balanced divide-and-conquer technique it uses and from Proposition 1.

Theorem 2. *Algorithm* Runs *reports all the runs occurring its input string y in time $O(|y| \log |y|)$.*

5 Computing local periods

The algorithm of the previous section leads to a simple algorithm for computing local periods of strings over infinite alphabets.

Recall that an overlap of a pair of strings s and t is a nonempty string w for which

$$A^*s \cap A^*w \neq \emptyset \text{ and } tA^* \cap wA^* \neq \emptyset.$$

The length $|w|$ is a local period of st at position $|s|$. The smallest of these quantities is called the local period of st at position $|s|$ and denoted by $\mathrm{LocPer}_s t(|s|)$.

In other words, following [5], an integer p is the local period of string y at position i if p is the length of the shortest string w that satisfies one of the four conditions:

1. $y[0\,..\,i-1]$ is a suffix of w and $y[i\,..\,|y|-1]$ is a prefix of w,
2. $y[0\,..\,i-1]$ is a suffix of w and w is a prefix of $y[i\,..\,|y|-1]$,
3. w is a suffix of $y[0\,..\,i-1]$ and $y[i\,..\,|y|-1]$ is a prefix of w,
4. w is a suffix of $y[0\,..\,i-1]$ and w is a prefix of $y[i\,..\,|y|-1]$.

In the first case $p = |w|$ is the (shortest) period of y. Its computation can done with the Prefix table of y because

$$p = \min\{q \mid 0 \le q < |y| \text{ and } q + \mathrm{Pref}_y(q) = |y|\} \cup \{|y|\}.$$

The second and third cases can be solved using just the Prefix table of y and the Prefix table of $\tilde{y}$ respectively.

The fourth case occurs when there is a square centred at position i. The occurrence of the square is part of a run and then its detection can be done with the algorithm of the previous section.

The whole algorithm is updated to compute the table LocPer_z instead of reporting runs. The main change, apart from the technical details on the code, is done to the instruction at line 11 of RightRunsInProduct (and at similar line of LeftRunsInProduct). More precisely the instruction is changed to:

```
1  for i ← k − ℓ + p to k + r do
2      LocPer[i] ← min{LocPer[i], p}
```

Theorem 3. *The local periods of a string of length n drawn from an infinite alphabet can be computed optimally in time $O(n \log n)$.*

Proof. The treatments of cases 1, 2 and 3 clearly take linear time each. Discarding the change in the algorithm, case 4 is solved in time $O(n \log n)$.

To evaluate the whole running time we need to count the number of executions of the instruction at line 2. Since each run is processed only once at that line **Is it true?**, all the updates of $\mathrm{LocPer}[i]$ are done with different values of the period p. It is simple to prove there are a logarithmic number of them (see for example [2] for the number of occurrences of primitively-rooted squares in a string) therefore there are no more than $O(n \log n)$ updates.

Adding the running time of all the steps gives the stated running time.

The optimality comes from the fact that the algorithm, indeed only the part dealing with case 4, can be used to test whether the string contains a square or not. It is known that such a test has a $\Omega(n \log n)$ lower bound (see [13]). Therefore, the algorithm is time optimal.

Recall that a position i on the string y is called a critical position if the local period at i is the (global) period of y. The factorisation of y into $y[0\,..\,k-1]y[k\,..\,|y|-1]$ is said to be critical. The next statement is another consequence of the above algorithms.

Corollary 4. *The critical factorisations of a string of length n drawn from an infinite alphabet can be computed in time $O(n \log n)$.*

6 Computing runs with DBF

In this section we describe a simple $O(n \log n)$ algorithm for computing the Local Periods of the word y from its Dictionary of Basic Factors (DBF, see [6, Chapter 7]).

We start with two technical definitions: for a set of integers X and an integer k denote

$$k \ominus X \;=\; \{k - x \;:\; x \in X\}, \qquad k \oplus X \;=\; \{k + x \;:\; x \in X\}.$$

To simplify the computations we use the following two lemmas concerning closely overlapping occurrences of factors. Their proofs can be found in [10].

Lemma 5. *If $M \geq |u|/2$ then BordersLarger(u, M) is a single arithmetic progression.*

Lemma 6. *Assume $u, v \in \mathcal{BF}(y)$, $|u| = |v| > 1$ and $u = u_1u_2$ and $v = v_2v_1$, $|u_1| = |u_2| = |v_1| = |v_2|$. If $|Occ(u_1, v)| \geq 3$ and $|Occ(v_1, u)| \geq 3$ then $\mathsf{per}(u_1) = \mathsf{per}(v_1)$, that is, the arithmetic progressions $Occ(u_1, v)$ and $Occ(v_1, u)$ have the same difference.*

The algorithm investigates all inter positions of the length-n word y. For each interposition p it performs at most $O(\log n)$ steps. In each step it decides if there is local period of length $[2^k \,..\, 2^{k+1})$. Since we are only interested in computing the minimal local periods the computation terminates at the first success.

LocalPeriodsUsingDBF(y)

```
 1  LocPer[p] ← undefined for p = 0, ..., |y|
 2  compute dictionary of basic factors BF(y)
 3  preprocess elements of BF(y) for Occ queries
 4  for p ← 1 to |y| − 1 do
 5      for k ← 0 to ⌊log2 min(p, |y| − p)⌋ do
 6          u ← y[p .. p + 2^k − 1]
 7          v ← y[p − 2^k .. p − 1]
 8          ▷ note that u, v ∈ BF(y)
 9          O_u ← 2^{k+1} ⊖ Occ(u, y[p − 2^{k+1} .. p − 1])
10          O_v ← 2^k ⊕ Occ(v, y[p .. p + 2^{k+1} − 1])
11          C ← O_u ∩ O_v
12          if C ≠ ∅ then
13              LocPer[p] ← min C
14              break
15  return LocPer
```

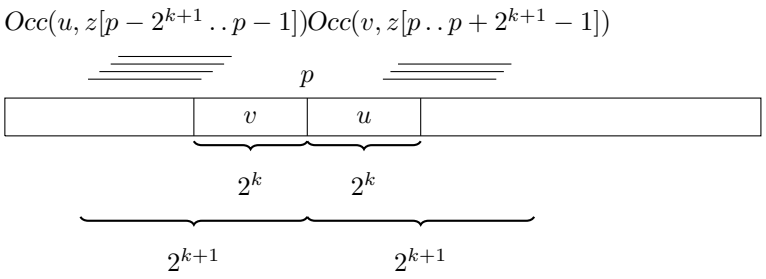

Figure 2. Single loop iteration of the algorithm LocalPeriodsUsingDBF

Theorem 7. *The local periods of a string of length n drawn from an ordered alphabet can be computed in time $O(n \log n)$ using algorithm LocalPeriodsUsingDBF.*

Proof. The first important step of the algorithm is the preprocessing of Occ queries at line 3. This can be done in $O(n \log n)$ time using hashing arrays. For each $w \in \mathcal{BF}(z)$ we define $A_{w,i}$ as the arithmetic sequence of occurrences of w starting in $z[i|w| \,..\, (i+1)|w| - 1]$. We can store non-empty entries $A_{w,i}$ in the perfect hash table, such that further queries of the form $Occ(w, z[i \,..\, j])$ (for $j - i = O(|w|)$) can be answered in $O(1)$ time.

The other crucial part of the algorithm is the computation of the intersection of arithmetic sequences $O_u \cap O_v$ at line 11. Luckily this step can be done in constant time. We can observe that we have two cases:

- one of the sequences is short ($|O_u| < 3$ or $|O_v| < 3$) — so the intersection requires some simple arithmetic operations,
- both sequences are long ($|O_u| \geq 3$ and $|O_v| \geq 3$) — from Lemma 6 we know that the sequences share the same arithmetic progression. Then once again the computation of the intersection requires only some simple arithmetic operations.

We conclude that each iteration of the loop from line 5 requires only $O(1)$ time. A single iteration of this loop is illustrated on the figure 2. The total time and space complexity of the algorithm is $O(n \log n)$.

References

1. M. Crochemore: *Transducers and repetitions.* Theoret. Comput. Sci., 45(1) 1986, pp. 63–86.
2. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, 2007, 392 pages.
3. M. Crochemore, L. Ilie, and L. Tinta: *The "runs" conjecture.* Theoretical Computer Science, 412(27) 2011, pp. 2931–2941.
4. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń: *Extracting powers and periods in a string from its runs structure*, in SPIRE, E. Chávez and S. Lonardi, eds., vol. 6393 of Lecture Notes in Computer Science, Springer, 2010, pp. 258–269.
5. M. Crochemore and D. Perrin: *Two-way string matching.* J. ACM, 38(3) 1991, pp. 651–675.
6. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific Publishing, Hong-Kong, 2002, 310 pages.
7. J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre: *Linear-time computation of local periods.* Theor. Comput. Sci., 326(1-3) 2004, pp. 229–240.
8. D. Gusfield: *Algorithms on strings, trees and sequences: computer science and computational biology*, Cambridge University Press, Cambridge, 1997.
9. C. S. Iliopoulos, D. Moore, and W. F. Smyth: *A characterization of the squares in a fibonacci string.* Theor. Comput. Sci., 172(1-2) 1997, pp. 281–291.
10. T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń: *Efficient data structures for the factor periodicity problem*, in SPIRE, Lecture Notes in Computer Science, Springer, 2012, In press.
11. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in FOCS, IEEE Computer Society, 1999, pp. 596–604.
12. M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Mathematics, 25(1-2) Sept. 1989, pp. 145–153.
13. M. G. Main and R. J. Lorentz: *An $O(n \log n)$ algorithm for finding all repetitions in a string.* J. Algorithms, 5(3) 1984, pp. 422–432.
14. W. Rytter: *The number of runs in a string.* Inf. Comput., 205(9) 2007, pp. 1459–1469.

Fast Algorithms for Online Searching on Burrows-Wheeler Transformed Texts

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | faro | giaquinta}@dmi.unict.it

Abstract. In this paper we propose an alternative approach to the problem of counting the occurrences of a pattern in a given text encoded by the Burrows-Wheeler transform. Our approach, based on a partial computation of the *select* data structure, leads to algorithms which in most cases are faster and use less space than existing online algorithms, as shown by extensive experimentation.

1 Introduction

The *encoded string matching problem* is a variant of the classical string matching problem. It consists in searching for all the occurrences of a given pattern P in a text T stored in encoded form. Typical text encodings are, for instance, text compression, text encryption or text transformation.

A straightforward solution, the so-called *decode-and-search* strategy, consists in decoding the text and then using any classical string matching algorithm for searching. However, recent results show that in many cases searching directly in encoded texts can be more efficient.

In this paper we consider the problem of searching for a given pattern in a text encoded by the Burrows-Wheeler transform [4].

The Burrows-Wheeler transform (BWT) is a powerful reversible transformation which yields a permutation of the text that can be better compressed using 0-th order coding algorithms [10,15]. It is used in compression programs [14,13] which are among the best compression wise.

There are two main approaches to searching in an encoded text, *offline* and *online*. The offline approach consists in building at encoding time an index of some sort of the encoded data which can be used to efficiently search arbitrary substrings of the indexed text. An index should support at least two types of queries: counting the occurrences of a pattern and locating their positions. Most offline algorithms (cf. [8,5,12]) are based on the relationship between the Burrows-Wheeler transform and the suffix array of the text [11]. They consist in creating an index, based on the compression of the suffix array, which contains the indexed text.

Traditionally, the online approach should preprocess only the pattern. However, the existing online algorithms for the Burrows-Wheeler transform perform a preprocessing also on the encoded text; in this respect, they are not strictly online. The difference is that in this case the index is built at search time and resides in main memory. The major drawback is that they require more than one iteration over the encoded text to perform the preprocessing and that the size of the preprocessed data is linear in the size of the text.

In this paper we propose a new type of preprocessing for online algorithms to answer count queries. While still requiring linear space in the worst case, it uses

i	M ($id = 5$)	F	L	V	W	I	Hr
1	i m i s s i s s i p p	i	p	6	5	11	5
2	i p p i m i s s i s s	i	s	8	7	8	4
3	i s s i p p i m i s s	i	s	9	10	5	11
4	i s s i s s i p p i m	i	m	5	11	2	9
5	m i s s i s s i p p i	m	i	1	4	1	3
6	p i m i s s i s s i p	p	p	7	1	10	10
7	p p i m i s s i s s i	p	i	2	6	9	8
8	s i p p i m i s s i s	s	s	10	2	7	2
9	s i s s i p p i m i s	s	s	11	3	4	7
10	s s i p p i m i s s i	s	i	3	8	6	6
11	s s i s s i p p i m i	s	i	4	9	3	1

	C
i	1
m	5
p	6
s	8

Figure 1. The matrix M and related arrays for the string "`mississippi`"

less space on average when the alphabet is moderately large, and requires just one iteration over the encoded text. We also illustrate variants based on this new preprocessing of existing online algorithms and present experimental results under various conditions. It turns out that such variants show better time and space behaviours, as compared with solutions currently available in literature.

The rest of the paper is organized as follows. In Sections 2 we describe the Burrows-Wheeler transform and present, in Section 3, the most efficient online algorithms used for searching on Burrows-Wheeler transformed texts. Then, in Section 4, we illustrate a new efficient preprocessing algorithm for online searching and present some novel variants of existing algorithms, based on the new preprocessing technique. Such variants are compared under different conditions with the most efficient algorithms available in the literature and the resulting experimental data are reported in Section 5. Finally, we briefly draw our conclusions in Section 6.

2 The Burrows-Wheeler transform

Before reviewing the Burrows-Wheeler transform, it is convenient to run through some notations and terminology. A string P of length $|P| = m \geq 1$ is represented as a finite array $P[1\,..\,m]$ of characters from a finite alphabet Σ. By $P[i]$ we denote the i-th character of P, for $1 \leq i \leq m$. Likewise, by $P[i\,..\,j]$ we denote the substring of P contained between the i-th and the j-th characters of P, for $1 \leq i \leq j \leq m$. A substring of the form $P[1\,..\,i]$ is called a *prefix* of P and a substring of the form $P[i\,..\,m]$ is called a *suffix* of P, for $1 \leq i \leq m$. For any two strings P and Q, we write $Q \sqsubseteq P$ to indicate that Q is a prefix of P. Similarly, we write $Q \sqsupseteq P$ to indicate that Q is a suffix of P.

To define the Burrows-Wheeler transform of a given string $T[1\,..\,n]$, we introduce a conceptual matrix M whose rows are the cyclic shifts of T sorted in lexicographic order. We indicate with M_i the i-th row of M, for $1 \leq i \leq n$. Fig. 2 shows the matrix M corresponding to the string "`mississippi`". Note that each column of M is a permutation of the characters of T. Let F and L be the first and the last columns of M, respectively. Hence F, by definition of M, can be obtained by sorting lexicographically the characters of T, and can thus be computed from any column of M, in particular from L. Then the BWT-encoding of T is defined as the pair (L, id), where id is the index of the row of M corresponding to T, i.e., such that

$M_{id} = T[1\,..\,n]$. It turns out that in general the string L is highly compressible, as it contains with high probability long runs of identical characters.

The BWT-encoding of a string of length n can be performed in $\mathcal{O}(n^2 \log n)$-time by the naive method outlined above. However, it can be computed more efficiently by using appropriate data structures, such as suffix trees, suffix arrays, and some of their variants.

The reverse BWT can be computed by a simple method, based on n subsequent sortings. However, a more efficient approach, proposed by Burrows and Wheeler, requires only two iterations over the data and has a linear cost. This consists in building a *transform array* V with the property that, for any character $L[i]$, the preceding character in the text is given by $L[V[i]]$. Thus, the array V can be used to decode the text backwards as follows

$$T[n-i] = L[V^i[id]]\,, \quad \text{for } 0 \le i \le n-1\,,$$

where $V^0[j] = j$ and, recursively, $V^{i+1}[j] = V[V^i[j]]$.

Plainly, such transformation is pratical only for decoding the entire text backwards. For left-to-right scanning, one can use a forward transformation, which is defined by iterating the inverse function $W = V^{-1}$ as follows

$$T[i] = L[W^i[id]]\,, \quad \text{for } 1 \le i \le n\,.$$

Both V and W can be computed in linear time, as shown in Fig. 2. To this end, it is helpful to define an array C, of dimension σ, where $C[c] - 1$ is the number of all occurrences in T of the characters which precede alphabetically c, for $c \in \Sigma$. (Notice that here and in the following we are implicitly identifying the ordered alphabet Σ with the integer range $[0\,..\,\sigma - 1]$, where $\sigma = |\Sigma|$.) By definition of M and C, $C[c]$ turns out to be the index of the first row in M starting with c. Observe that C can be used to implicitly define the array F since $C[c]$ and $C[c+1] - 1$ are, respectively, the indexes in F of the first and last occurrence of the character c, for each $c \in \Sigma$.

3 Searching on BWT-encoded texts

Let P be a pattern of length m, and let L be the BWT-encoding of a text T of length n, both over a finite alphabet Σ of dimension σ. In this section we describe the existing solutions for the (online) problem of searching the pattern P in T via L. Note that a direct online solution consists in decoding L and then using any classical string matching algorithm for the searching phase.

The first nontrivial online solution [2] is based on a function Hr which maps all characters from T to F and allows to access text positions in random order. Formally the mapping Hr and his inverse I are defined so as to satisfy

$$T[i] = F[Hr[i]] \quad \text{and} \quad T[I[i]] = F[i], \quad \text{for} \quad 1 \le i \le n.$$

The construction of Hr and I (cf. Fig. 2) requires three iterations over the BTW-encoded text and the computation of the arrays W and C, yielding an overall extra space of size $(3n + \sigma)$.

The resulting method turns out to be simple and flexible since one can use any existing string matching algorithm *as it is*, including non-standard pattern matching algorithms. In particular, [2] has chosen to adapt the Boyer-Moore [3] algorithm,

which has a $\mathcal{O}(nm)$ worst-case time complexity, but a sublinear behavior on average. However, the use of a linear algorithm may lead to an overall $\mathcal{O}(n)$ worst-case time algorithm.

A more remarkable result is the Binary-Search algorithm [2], which is based on the following observation. Since all rows M_i such that $P \sqsubseteq M_i$ are contiguous, it is possible to count the occurrences of the pattern P by locating the first and last matching rows. The idea is to use binary search to locate the pattern in the range of rows $[C[P[1]] \,..\, C[P[1]+1]-1]$. Rows are decoded as needed, using the array W, and are lexicographically compared with the pattern to update the current interval as in standard binary search. Once a matching row M_i is found, the first and last rows are searched using a slightly modified binary search in the ranges $[C[P[1]] \,..\, i-1]$ and $[i \,..\, C[P[1]+1]-1]$, respectively.

When the range is found, it is possible to query the corresponding positions in constant time, using the mapping I, since this, by the property above, represents a mapping between F and T. The computation of the mapping I requires, as for Hr, three iterations over the encoded text. However, if one is only interested in counting the matching occurrences, the preprocessing just requires two iterations, to build W.

The search time of the Binary-Search algorithm is $\mathcal{O}(m \log n)$ in the worst case and decreases to $\mathcal{O}(m \log n/\sigma)$ on average. Thus, the overall worst-case time complexity of the algorithm is $\mathcal{O}(n + m \log n)$.

The indexing data structures for the BWT are based on the compression of the suffix array of the text, which is strictly related to the BWT. In fact, they are more than a traditional index in that they also encode the indexed text. Such indexes allow one to efficiently compute two useful generic operations on symbol sequences:

- *rank*(L, c, i), which returns the number of occurrences of the character c in the prefix $L[1 \,..\, i]$;
- *select*(L, c, i), which returns the index in L of the i-th occurrence of the character c.

Note that the *rank* function allows one to compute the array V as needed, using the relation $V[i] = C[L[i]] + \mathit{rank}(L, L[i], i)$, for $1 \leq i \leq n$.

Recently, much attention has been devoted to the efficient implementation, time and space wise, of this data structure. In [5], Ferragina and Manzini present an efficient algorithm based on *rank* queries, which finds, as in binary search, the range of rows having P as a prefix. In particular they show that $\mathcal{O}(m)$ *rank* queries on the BWT are needed to count the occurrences of a pattern of length m. The latest version of their compressed index [6] uses wavelet trees to index a text T of length n with $nH_0(T) + o(n)$ bits of space, where $H_0(T)$ is the 0-th order entropy of T. The complexity of *rank* and *select* queries on a wavelet tree is $\mathcal{O}(\log \sigma)$. In a subsequent paper [7], an alternative data structure is used to resolve queries in constant time, when $\sigma = \mathcal{O}(\mathrm{polylog}\, n)$.

4 A new efficient approach for online searching BWT-encoded texts

In this section we present a new approach for online searching BWT-encoded texts, which yields algorithms with sublinear extra space in most practical cases, especially in the case of large alphabets and short patterns. The main idea consists in building a

```
BUILD-C (L)
1. for i ← 0 to σ − 1
2.     K[i] ← 0
3. for i ← 1 to n
4.     K[L[i]] ← K[L[i]] + 1
5. sum ← 1
6. for i ← 0 to σ − 1
7.     C[i] ← sum
8.     sum ← sum + K[i]
```

```
BUILD-V-W (L, C)
1. for i ← 1 to n
2.     V[i] ← C[L[i]]
3.     W[C[L[i]]] ← i
4.     C[L[i]] ← C[L[i]] + 1
```

```
BWT-DECODE (L, W, id)
1. i ← id
2. for j ← 1 to n
3.     i ← W[i]
4.     T[j] ← L[i]
```

```
BUILD-I-HR (W, id)
1. i ← id
2. for j ← 1 to n
3.     Hr[j] ← i
4.     I[i] ← j
5.     i ← W[i]
```

Figure 2. Algorithms to compute the C, V, W, I, Hr and T arrays

data structure which allows to efficiently implement *select* queries only for characters occurring within the pattern.

To this end, let P be, as above, a pattern of length m over a finite ordered alphabet Σ of size σ and let L be the BWT-encoding of a text T of length n, over the same alphabet. Let $\Sigma_P \subseteq \Sigma$ be the collection of the characters occurring within P. Trivially, $|\Sigma_P| \leq m$.

We construct a *Partial-select* data structure, called Ps and implemented as an array of size σ. For each $c \in \Sigma$, the entry $Ps[c]$ points to a list containing all positions in L of the character c, in increasing order. Thus, it turns out that $select(L, c, i) = Ps[c, i]$, where $Ps[c, i]$ is the i-th entry in the list pointed to by $P[c]$. If occurrences lists are implemented as arrays, each *select* query can be answered in constant time. For a given character c, let $K[c]$ be the number of the occurrences of c in L. The extra space needed for computing Ps is given by

$$\sigma + \sum_{c \in \Sigma_P} K[c] \leq \sigma + n\,,$$

where the equality holds if $\Sigma_P = \Sigma$. Of course, if the alphabet is small, the gain, if any, is negligible, but for moderately large alphabets, or when m is smaller than σ, it favorably compares with the total size of the text.

The *Partial-select* data structure, in combination with the array C introduced in Section 2, allows one to compare in an efficient way a pattern P of length m with any row M_i. In particular, one can check whether P is lexicographically smaller, equal, or greater than the prefix of length m of row M_i, for $1 \leq i \leq n$.

To this end, let us suppose that we have succesfully compared a prefix $P[1\,..\,k]$ of the pattern with row M_i, with $k < m$, and also assume that j is the index of $P[k]$ in L, i.e., $L[j] = P[k] = M_i[k]$. In order to compare $P[k+1]$ with $M_i[k+1]$, observe that the index of $M_i[k+1]$ in F is actually j; we thus need to know whether $F[j] = P[k+1]$. This can be done by exploiting the properties of the array C. In particular, given an index j and a character c, it can be verified in constant time if $F[j] = c$, by checking whether j is contained in the interval $[C[c]\,..\,C[c+1]-1]$. If the answer is negative, we can also verify if $F[j]$ is smaller or greater than c by checking whether j is smaller than $C[c]$ or greater than $C[c+1]-1$, respectively.

```
BUILD-C-PS (P, L)
 1. m ← len(P)
 2. for i ← 1 to σ
 3.       K[i] ← 0
 4.       H[i] ← 0
 5. for i ← 1 to m
 6.       H[P[i]] ← 1
 7. for i ← 1 to n
 8.       K[L[i]] ← K[L[i]] + 1
 9.       if H[L[i]] > 0
10.             Ps[L[i], H[L[i]] − 1] ← i
11.             H[L[i]] ← H[L[i]] + 1
12. Z ← 0
13. for i ← 1 to σ
14.       C[i] ← Z
15.       Z ← Z + K[i]
16. return (C, Ps)
```

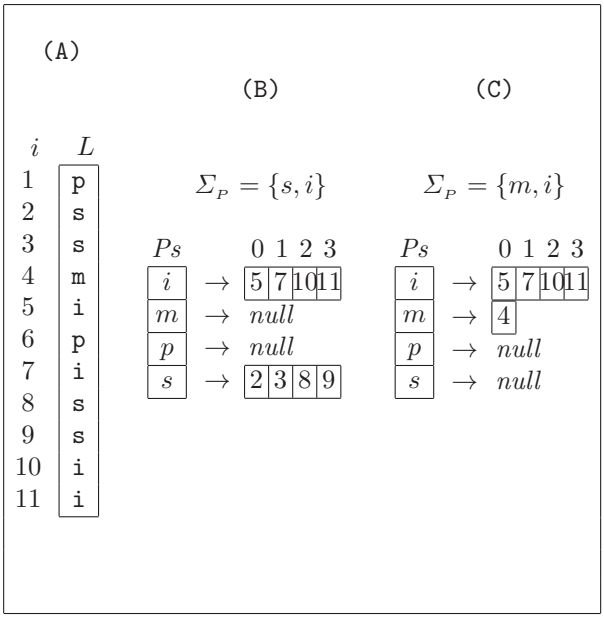

Figure 3. On the left: procedure BUILD-C-PS for computing the new data structure Ps. On the right: **(A)** an example: the BWT-encoding of the string "`mississippi`"; **(B)** the data structure Ps relative to L, for the alphabet $\Sigma_P = \{i, s\}$; **(C)** the data structure Ps relative to L, for the alphabet $\Sigma_P = \{i, m\}$

We use this tecnique to check whether $M_i[k+1] = P[k+1]$. If the answer is positive, we reiterate the same procedure on $L[j']$, where j' is the index of $M_i[k+1]$ in L.

To compute j', we observe that, by definition of M, the i-th occurrence of a character c in F and in L maps onto the same character in T. Moreover, we note that the index j in F corresponds to the $(j - C[F[j]])$-th occurrence of the character $F[j]$ in F. It thus turns out that the index j' can be computed in constant time by quering the Ps data structure as follows:

$$j' = Ps\Big[P[k+1], j - C[P[k+1]]\Big].$$

A comparison function, named PS-STRCMP and based on the Ps data structure, is shown in Fig. 4. It requires $\mathcal{O}(m)$-time for comparing P with $M_i[1\,..\,m]$.

Fig. 3 also shows the code of the procedure for computing the *Partial-select* data structure Ps. It requires a single iteration on the BWT-encoded text and has a $\mathcal{O}(n)$-time and -space complexity.

4.1 A Standard-Search algorithm

Our first algorithm works as a standard pattern matching algorithm; it compares the pattern P with the windows of the text $T[i\,..\,i+m-1]$, for $1 \le i \le n-m$. It exploits that fact that to locate all occurrences of P in T it is enough to compare the pattern with all the windows starting with character $T[i] = P[1]$. This corresponds to comparing the pattern with all the rows M_i starting with $P[1]$. Trivially, M_i starts with symbol $P[1]$ if and only if i is in the range $[C[P[1]]\,..\,C[P[1]+1]-1]$. Thus, our proposed algorithm exploits the property that the i-th occurrence in L of character $P[1]$ is found at position $Ps[P[1], i]$.

```
STANDARD-SEARCH (P, L)
1. count ← 0
2. (C, Ps) ← BUILD-C-PS(P, L)
3. for i = C[P[1]] to C[P[1] + 1] − 1
4.       if PS-STRCMP(P, C, Ps, i) = 0
5.             then count ← count + 1
6. return count
```

```
PS-STRCMP (P, C, Ps, i)
1. m ← len(P), c ← P[1]
2. for j ← 2 to m
3.       i ← Ps[c, i − C[c]], c ← P[j]
4.       if i < C[c] then return 1
5.       if i >= C[c + 1] then return −1
6. return 0
```

```
BINARY-SEARCH (P, L)
 1. count ← 0
 2. (C, Ps) ← BUILD-C-PS(P, L)
 3. c ← P[1]
 4. low ← C[c]
 5. high ← C[c + 1] − 1
 6. while low < high
 7.       mid ← (low + high)/2
 8.       cmp ← PS-STRCMP(P, C, Ps, mid)
 9.       if cmp = 0 then break
10.       if cmp > 0 then low ← mid + 1
11.       else high ← mid
12. if cmp = 0 then
13.       h ← mid − 1
14.       while low < h
15.             m ← (low + h)/2
16.             if PS-STRCMP(P, C, Ps, m) > 0
17.             then low ← m + 1
18.             else h ← m
19.       if PS-STRCMP(P, C, Ps, low) ≠ 0
20.       then low ← mid
21.       l ← mid + 1
22.       while l < high
23.             m ← (l + high + 1)/2
24.             if PS-STRCMP(P, C, Ps, l) ≥ 0
25.             then l ← m
26.             else high ← m − 1
27.       if PS-STRCMP(P, C, Ps, high) ≠ 0
28.       then high ← mid
29.       count ← high − low + 1
30. return count
```

```
RANK-SEARCH (P, L)
 1. count ← 0
 2. (C, Ps) ← BUILD-C-PS(P, L)
 3. i ← len(P)
 4. c ← P[i]
 5. sp ← C[c]
 6. ep ← C[c + 1] − 1
 7. while sp ≤ ep and i ≥ 2
 8.    c ← P[i − 1]
 9.    sp ← C[c] + PS-RANK(Ps, c, sp − 1) + 1
10.    ep ← C[c] + PS-RANK(Ps, c, ep)
11.    i ← i − 1
12. if ep ≥ sp then
13.    count ← ep − sp + 1
14. return count

PS-RANK (Ps, c, i)
 1. low ← 1
 2. high ← len(Ps[c])
 3. while low < high
 4.       mid ← (low + high)/2
 5.       if Ps[c][mid] > i then
 6.          high ← mid
 7.       else if Ps[c][mid] < i then
 8.             low ← mid + 1
 9.          else return mid
10. return low
```

Figure 4. Algorithms for online searching BWT-encoded texts. **On the top:** the Standard-Search algorithm. **On the left:** the Binary-Search algorithm. **On the right:** the Rank-Search algorithm

The resulting method is simple and flexible and can be used in combination with any existing string matching algorithm which processes the text from left to right, including non-standard pattern matching algorithms.

Fig. 4 (on the top) shows the code of the Standard-Search algorithm, where procedure PS-STRCMP is used as a subroutine for comparing the pattern with a row M_i. Despite its worst-case $\mathcal{O}(nm)$-time complexity, the algorithm turns out to be efficient in practice, expecially when the number of occurrences of the character $P[1]$ is small.

In particular, for a given pattern P, the searching phase has a $\mathcal{O}(mK[P[1]])$-time complexity.

4.2 A Binary-Search algorithm

Our second proposed solution is a variant of the Binary-Search algorithm, described in Section 3 (cf. [2]). It makes use of the data structure Ps to facilitate the comparison of the pattern P with the rows of the matrix M. The resulting algorithm, whose code is shown in Fig. 4 (on the left), has the same structure of the Binary-Search algorithm. As in the Standard-Search algorithm, we search the pattern in the range of rows $[C[P[1]] \,..\, C[P[1]+1]-1]$. A first binary search is applied (lines 4-11) to locate a matching row M_i such that $P \sqsubseteq M_i$. When a matching row M_i is found (line 12), a slightly modified binary search is used to locate the first row in the range $[C[P[1]] \,..\, i]$ (lines 13-20) and to locate the last row in the range $[i \,..\, C[P[1]+1]-1]$ (lines 21-28).

The new Binary-Search method, as the original algorithm, achieves a $\mathcal{O}(n + m \log n)$ overall time complexity. However, since $|Ps| + |C| \leq |W| + |C|$, in practical cases it uses less space than the Binary-Search algorithm, as shown in Section 5.

4.3 A Ranking algorithm

Our last solution for online searching BWT-encoded data is based on the use of the *rank* function, as done in standard indexing algorithms.

The Rank-Search algorithm counts the number of occurrences of P in T by locating two indexes, sp and ep, such that $P \sqsubseteq M_i$, for all i in the range $[sp \,..\, ep]$. This can be done with $O(m)$ *rank* queries. The code of the Rank-Search algorithm is presented in Fig. 4 (on the right).

The new approach uses the subroutine Ps-Rank to exploits the Ps data structure in order to efficiently compute any *rank* query on L. In particular, for $c \in \Sigma_P$, the query $rank(L, c, i)$ can be answered via the following relation

$$rank(L, c, i) = \max\{j \mid Ps[c, j] \leq i\} \cup \{0\} \,.$$

Since the occurrences lists in Ps are in increasing order, it is possible to use a binary search for locating the value of $rank(L, i, c)$. The procedure Ps-Rank achieves a $\mathcal{O}(\log K[c])$-time complexity for answering any *rank* query on characters occurring in the pattern, where we recall that $K[c]$ is the number of occurrences of c in T. The Rank-Search algorithm achieves a $\mathcal{O}(n + m \log n)$ overall time complexity.

5 Experimental results

In this section we provide and comment experimental results, in terms of space usage, preprocessing running times, and searching running times, of the following online algorithms for searching BWT-encoded texts:

- HS: the Horspool algorithm which searches the text using the Hr mapping;
- D&S: the Decode-and-Search method with the Horspool algorithm;
- BS: the Binary-Search algorithm;
- SS: our Standard-Search algorithm;
- BS2: our modified variant of the Binary-Search algorithm;
- RS: our *rank* based algorithm.

In the case of the HS and D&S algorithms, we used the Horspool algorithm [9] which is a simple and efficient variant of the Boyer-Moore algorithm.

All algorithms have been implemented in the C programming language and have been compiled using the optimization options `-O2 -fno-guess-branch-probability`. All tests have been performed on a 1.5 GHz PowerPC G4 with 1Gb memory and running times have been measured with a hardware cycle counter, available on modern CPUs.

Our tests have been carried out on four real world problems using the following text buffers:

(1) the English King James version of the "Bible", with an alphabet of 63 characters and a length of $4,047,391$ characters;
(2) the English CIA World Fact Book, with an alphabet of 94 different characters and a length of $2,473,400$ characters;
(3) a protein sequence from the human genome, with an alphabet of 22 characters and a length of $2,900,352$ characters;
(4) a DNA sequence from the *Escherichia coli* genome, with an alphabet of 4 characters and a length of $4,638,690$ base pairs.

Data (1), (2), and (4) have been taken from the Large Canterbury Corpus [1].

For each input file, we have generated sets of 100 patterns, randomly extracted from the text, of fixed length m, with m ranging in the set {4, 8, 12, 16, 20, 24, 28, 32}.

5.1 Average space usage

For each set of patterns, we computed the space used during preprocessing, expressed as number of bytes for text character. In particular, integers have been represented in our tests by 4 bytes and characters by 1 byte. Hence, the arrays L and W require n bytes and $4n$ bytes, respectively.

Note that the space required by the BS, HS, and D&S algorithms is independent of the alphabet size and of the pattern size.

		RS, BS2 and SS								HS	D&S	BS
text	σ	4	8	12	16	20	24	28	32			
King James Bible	(63)	1.16	1.76	2.16	2.52	2.68	2.88	2.92	3.04	8.00	5.00	4.00
CIA World Fact Book	(93)	0.72	1.32	1.64	1.84	2.04	2.24	2.40	2.56	8.00	5.00	4.00
Protein sequence	(22)	0.84	1.56	1.96	2.40	2.68	2.96	3.16	3.28	8.00	5.00	4.00
DNA sequence	(4)	2.80	3.44	3.68	3.84	3.92	3.92	3.96	3.96	8.00	5.00	4.00

Average extra space required by the algorithms in bytes for text character

From the above experimental results, it turns out that the extra space required by our newly presented variants is up to four times smaller than that required by the BS algorithm, whose space-performance is better than those of the algorithms HS and D&S. As expected, the best results are obtained for large alphabets and short patterns. The gap relative to the BS algorithm decreases with the size of the alphabet. In particular, for an alphabet of dimension 4 and long patterns the space required by the *Ps* data structure is almost the same as in the BS algorithm.

5.2 Preprocessing and searching times

Next we present experimental results relative to the preprocessing times, needed for computing all auxiliary arrays, and searching times, needed for counting all occurrences of the patterns in the BWT-encoded texts. For each set of patterns, we report the mean over the running times of 100 runs. Running times are expressed in milliseconds.

	text (1) : preprocessing times						text (1) : searching times					
m	HS	D&S	BS	BS2	RS	SS	HS	D&S	BS	BS2	RS	SS
4	289.05	263.95	118.61	78.72	79.03	78.57	65.08	17.03	0.31	0.10	0.09	1.09
8	289.29	266.69	119.59	90.45	90.69	91.63	57.35	12.29	0.31	0.12	0.19	1.43
12	303.30	283.59	124.56	101.09	100.85	100.89	54.61	10.66	0.34	0.15	0.25	1.42
16	316.52	288.57	124.96	108.77	108.33	108.88	55.08	10.49	0.33	0.16	0.27	1.38
20	300.67	268.06	119.71	106.16	105.79	105.58	46.62	9.47	0.30	0.15	0.26	1.63
24	301.69	267.65	120.20	110.33	109.72	109.62	43.13	9.27	0.32	0.17	0.26	1.73
28	299.88	265.20	119.17	109.98	110.19	110.11	39.17	8.90	0.31	0.16	0.25	1.57
32	316.54	287.83	125.37	117.63	116.95	117.56	40.29	9.33	0.34	0.18	0.31	1.81

Experimental results on the BWT-encoded version of the King James version of the Bible

	text (2) : preprocessing times)						text (2) : searching times					
m	HS	D&S	BS	BS2	RS	SS	HS	D&S	BS	BS2	RS	SS
4	193.10	167.16	74.64	43.23	42.98	43.32	43.05	10.15	0.15	0.04	0.04	0.40
8	194.15	168.07	75.92	49.46	50.16	49.51	37.47	7.23	0.16	0.06	0.11	0.58
12	194.18	168.62	75.76	53.71	54.08	53.44	34.18	6.40	0.17	0.07	0.09	0.45
16	203.26	179.27	77.96	57.36	57.62	57.25	35.49	5.93	0.18	0.07	0.12	0.55
20	206.34	179.75	78.11	59.20	59.14	59.14	28.70	5.84	0.20	0.08	0.15	0.71
24	206.85	180.48	78.54	61.70	62.05	61.86	26.29	5.53	0.17	0.09	0.16	0.60
28	205.89	179.87	78.24	63.52	63.37	63.42	23.72	5.55	0.19	0.10	0.17	0.71
32	210.96	183.26	78.45	65.29	65.08	69.16	22.90	5.35	0.24	0.11	0.18	0.85

Experimental results on the BWT-encoded version of the CIA World Fact Book

	text (3) : preprocessing times						text (3) : searching times					
m	HS	D&S	BS	BS2	RS	SS	HS	D&S	BS	BS2	RS	SS
4	240.95	217.05	90.12	57.01	57.69	56.97	53.07	12.50	0.20	0.06	0.06	0.46
8	247.35	225.49	91.23	70.58	70.88	71.63	46.66	9.37	0.20	0.07	0.08	0.67
12	245.42	219.10	90.50	76.55	76.35	76.18	41.23	7.84	0.20	0.08	0.08	0.60
16	248.07	225.07	91.60	82.91	83.23	83.28	40.48	7.49	0.22	0.09	0.10	0.74
20	250.87	226.58	91.25	87.58	87.70	85.78	35.85	7.23	0.19	0.09	0.12	0.91
24	239.23	217.87	91.63	87.35	86.31	87.21	33.28	7.32	0.21	0.09	0.12	0.99
28	243.14	227.61	91.86	91.12	91.05	90.66	30.74	6.97	0.21	0.10	0.12	0.95
32	246.40	223.29	92.09	90.57	90.43	90.82	30.06	6.95	0.22	0.10	0.11	0.93

Experimental results on the BWT-encoded version of a protein sequence

	text (4) : preprocessing times						text (4) : searching times					
m	HS	D&S	BS	BS2	RS	SS	HS	D&S	BS	BS2	RS	SS
4	380.69	360.17	138.32	134.01	133.69	135.98	99.25	31.99	0.62	0.27	0.17	9.27
8	389.59	367.17	139.57	143.48	143.58	142.75	89.35	26.15	0.74	0.34	0.35	9.45
12	375.05	351.71	137.04	143.98	142.74	142.98	83.65	23.92	0.67	0.35	0.38	9.22
16	381.71	354.33	141.81	148.65	149.29	147.97	89.78	25.85	0.63	0.34	0.33	10.34
20	380.08	358.18	137.77	149.12	147.29	147.24	83.33	24.27	0.64	0.36	0.36	9.95
24	369.93	351.81	139.86	148.72	151.86	148.10	83.91	24.33	0.64	0.37	0.39	9.59
28	380.30	372.17	140.94	156.90	151.64	151.02	84.50	24.90	0.71	0.40	0.38	9.69
32	375.33	350.23	137.29	148.76	147.63	147.57	88.01	26.66	0.68	0.34	0.41	10.08

Experimental results on the BWT-encoded version of a DNA sequence

The last three algorithms, BS2, RS, and SS, use the same data structure; thus their preprocessing times are almost identical. The HS algorithm, based on the Hr mapping, turns out to be the worst, even worse than the D&S method. Accessing the text in a non-sequential way is not cache friendly and since the space needed to decode the text is the same as the one required by Hr, there is no reason to prefer the HS algorithm to the D&S method. The algorithms BS2 and the RS always achieve better running times as compared with the Binary-Search algorithm. The SS algorithm may

be faster than the other algorithms when the frequency of the first character of the pattern is very low, but on average is always slower.

6 Conclusions

We have presented an alternative approach to the problem of counting the occurrences of a pattern in the Burrows-Wheeler transform of a given text. Specifically, we have proposed a data structure which allows to efficiently answer *select* queries for characters occurring within the pattern. Such data structure has then been used for comparing a string with the rows of the BWT-matrix and to answer *rank* queries. We have adapted existing online algorithms to this method and carried out extensive tests. Experimental results show that, when the alphabet is moderately large, our modified algorithms are faster and use less space on average than other algorithms currently present in literature.

References

1. R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *DCC '97: Proceedings of the Conference on Data Compression*, pages 201–210, Washington, DC, USA, 1997. IEEE Computer Society.
2. T. Bell, M. Powell, A. Mukherjee, and D. Adjeroh. Searching BWT compressed text with the Boyer-Moore algorithm and binary search. In *DCC '02: Proceedings of the Data Compression Conference*, pages 112–121, Washington, DC, USA, 2002. IEEE Computer Society.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.
4. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm, 1994. Digital SRC Research Report 124.
5. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, Washington, DC, USA, 2000. IEEE Computer Society.
6. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly fm-index. In *String Processing and Information Retrieval*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160. Springer-Verlag Berlin, 2004.
7. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):Article 20, 2007.
8. R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.
9. R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.
10. D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1951.
11. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.
12. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *ISAAC '00: Proceedings of the 11th International Conference on Algorithms and Computation*, pages 410–421, London, UK, 2000. Springer-Verlag Berlin.
13. M. Schinlder. The szip home page, 1997. `http://www.compressconsult.com/szip/`.
14. J. Seward. The bzip2 home page, 1997. `http://www.bzip.org/`.
15. I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.

Twenty Years of Bit-Parallelism in String Matching

Simone Faro[1] and Thierry Lecroq[2]

[1] Università di Catania, Viale A. Doria n. 6, 95125 Catania, Italy
[2] Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

Abstract. It has been twenty years since the publication of the two seminal papers of Baeza-Yates and Gonnet and of Wu and Manber in the September 1992 issue of the Communications of the ACM. The use of intrinsic parallelism of the bit operations inside a computer word, the so-called bit-parallelism, allows to cut down the number of operations that an algorithm performs by a factor up to ω, where ω is the number of bits in the computer word. This was then achieved by the Shift-Or and the Shift-And string matching algorithms. These two papers has inspired a lot of works and since 1992 a large number of papers were published describing string matching algorithms using this technique. In this survey we will review these solutions for exact single string matching, for exact multiple string matching and for approximate single string matching.

1 Introduction

String matching consists in finding one or more generally all the occurrences (exact or approximate) of a single string (or a finite set of strings) in a text. It is an extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry. String matching is a very important subject in the wider domain of text processing and algorithms for the problem are also basic components used in implementations of practical softwares existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science. Finally they also play an important role in theoretical computer science by providing challenging problems.

Although data are memorized in various ways, text remains the main form to exchange information. This is particularly evident in literature or linguistics where data are composed of huge corpus and dictionaries. This apply as well to computer science where a large amount of data are stored in linear files. And this is also the case, for instance, in molecular biology because biological molecules can often be approximated as sequences of nucleotides or amino acids.

Furthermore, the quantity of available data in these fields tend to double every eighteen months. This is the reason why algorithms should be efficient even if the speed and capacity of storage of computers increase regularly.

Solutions can be based on direct comparisons between symbols of the string and of the text, or on the use of various kinds of automata or by simulating these automata by using bit-parallelism.

Bit-parallelism is a technique firstly introduced in [29], and later revisited, twenty years ago, in [10,64], which takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to ω, where ω is the number of bits in the computer word. Bit-parallelism is indeed particularly suitable for the efficient simulation

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

of non-deterministic automata. In the following we will review the solutions, based on bit-parallelism, for exact single string matching, exact multiple string matching and for approximate single string matching.

The remaining of this paper is organized as follows. Section 2 introduces the basic definitions and the notations used throughout the remaining of the article. In Section 3 we present solutions for exact single string matching. Solutions for exact multiple string matching are presented in Section 4 while solutions for approximate single string matching are presented in Section 5.

2 Notions and Basic Definitions

A string P of length $|P| = m$ over a given finite alphabet Σ is any sequence of m characters of Σ. For $m = 0$, we obtain the empty string ε. Σ^* is the collection of all finite strings over Σ. We denote by $P[i]$ the $(i+1)$-st character of P, for $0 \leq i < m$. Likewise, the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P is denoted by $P[i\,..\,j]$, for $0 \leq i \leq j < m$. We also put $P_i =_{\text{Def}} P[0\,..\,i]$, for $0 \leq i < m$, and make the convention that P_{-1} denotes the empty string ε. It is common to identify a string of length 1 with the character occurring in it. For any two strings P and P', we write $P.P'$ to denote the concatenation of P' to P, and $P' \sqsupset P$ to express that P' is a *proper* suffix of P, i.e., $P = P''.P'$ for some nonempty string P''. The notation $P' \sqsupseteq P$ will be used with the obvious meaning. Analogously, $P' \sqsubseteq P$ ($P' \sqsubset P$) expresses that P' is a (proper) prefix of P, i.e., $P = P'.P''$ for some (nonempty) string P''. We say that P' is a factor of P if $P = P''.P'.P'''$, for some strings $P'', P''' \in \Sigma^*$, and we denote by $\mathit{Fact}(P)$ the set of the factors of P. Likewise, we denote by $\mathit{Suff}(P)$ the set of the suffixes of P. We write P^{r} to denote the reverse of the string P, i.e., $P^{\mathsf{r}} = P[m-1]P[m-2]\ldots P[0]$. Given a finite set of patterns $\mathcal{P}$, we put $\mathcal{P}^{\mathsf{r}} =_{\text{Def}} \{P^{\mathsf{r}} \mid P \in \mathcal{P}\}$ and $\mathcal{P}_l =_{\text{Def}} \{P[0\,..\,l-1] \mid P \in \mathcal{P}\}$. Also we put $\mathit{size}(\mathcal{P}) =_{\text{Def}} \sum_{P\in\mathcal{P}} |P|$ and extend the maps $\mathit{Fact}(\cdot)$ and $\mathit{Suff}(\cdot)$ to $\mathcal{P}$ by putting $\mathit{Fact}(\mathcal{P}) =_{\text{Def}} \bigcup_{P\in\mathcal{P}} \mathit{Fact}(P)$ and $\mathit{Suff}(\mathcal{P}) =_{\text{Def}} \bigcup_{P\in\mathcal{P}} \mathit{Suff}(P)$.

The algorithms reviewed in this paper make use of bitwise operations, i.e. operations which operate on one or more bit vectors at the level of their individual bits. On modern architectures bitwise operations have the same speed as addition but are significantly faster than multiplication and division.

We use the C-like notations in order to represent bitwise operations. In particular:

"|" represents the bitwise operation Or; ((01101101) | (10101100) = (11101101));
"&" represents the bitwise operation And; ((01101101) & (10101100) = (00101100));
"$\sim$" represents the one's complement; ($\sim$ (01101101) = (10010010));
"$\ll$" represents the bitwise left shift; and ((01101101) $\ll$ 2 = (10110100));
"$\gg$" represents the bitwise right shift. ((01101101) $\gg$ 2 = (00011011)).

All operations listed above use a single CPU cycle to be computed. Moreover some of the algorithms described below make use of the following, non trivial, bitwise operations:

"reverse" represents the reverse operation; (reverse(01101101) = (10110110));
"bsf" represents the bit scan forward operation; (bsf(00010110) = 3);
"popcount" represents population count operation. (popcount(01101101) = 5);

Specifically, for a given bit-vector B, the reverse operation inverts the order of the bits in a bit-vector B and can be implemented efficiently with $\mathcal{O}(\log_2(\text{length}(B)))$-time, the bsf operation counts the number of zeros preceeding the leftmost bit set to one in B, while the popcount operation counts the number of bits set to one in B and can be performed in $\mathcal{O}(\log_2(\text{length}(B)))$-time. (see [9] for the detailed implementation of the operations listed above).

The functions that compute the first and the last bit set to 1 of a word x are $\lfloor \log_2(x \;\&\; (\sim x + 1)) \rfloor$ and $\lfloor \log_2(x) \rfloor$, respectively. [1]

A nondeterministic finite automaton (NFA) with ε-transitions is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the collection of final states, Σ is an alphabet, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathscr{P}(Q)$ is the transition function ($\mathscr{P}(\cdot)$ is the powerset operator).[2] For each state $q \in Q$, the ε-closure of q, denoted as ECLOSE(q), is the set of states that are reachable from q by following zero or more ε-transitions. ECLOSE can be generalized to a set of states by putting $\text{ECLOSE}(D) = \bigcup_{q \in D} \text{ECLOSE}(q)$. In the case of an NFA without ε-transitions, we have $\text{ECLOSE}(q) = \{q\}$, for any $q \in Q$.

The *extended* transition function $\delta^* : Q \times \Sigma^* \to \mathscr{P}(Q)$ induced by δ is defined recursively by

$$\delta^*(q, u) =_{Def} \begin{cases} \bigcup_{p \in \delta^*(q,v)} \text{ECLOSE}(\delta(p, c)) & \text{if } u = v.c, \text{ for some } v \in \Sigma^* \\ & \quad \text{and } c \in \Sigma, \\ \text{ECLOSE}(q) & \text{otherwise (i.e., if } u = \varepsilon). \end{cases}$$

In particular, when no ε-transition is present, then

$$\delta^*(q, \varepsilon) = \{q\} \quad \text{and} \quad \delta^*(q, v.c) = \bigcup_{p \in \delta^*(q,v)} \delta(p, c)\,.$$

Both the transition function δ and the extended transition function δ^* can be naturally generalized to handle set of states, by putting $\delta(D, c) =_{Def} \bigcup_{q \in D} \delta(q, c)$ and $\delta^*(D, u) =_{Def} \bigcup_{q \in D} \delta^*(q, u)$, respectively, for $D \subseteq Q$, $c \in \Sigma$, and $u \in \Sigma^*$. The *extended* transition function satisfies the following property:

$$\delta^*(q, u.v) = \delta^*(\delta^*(q, u), v), \text{ for all } u, v \in \Sigma^*\,. \tag{1}$$

Given a set $\mathcal{P}$ of patterns over a finite alphabet Σ, the *trie* $\mathcal{T}_\mathcal{P}$ associated with $\mathcal{P}$ is a rooted directed tree, whose edges are labeled by single characters of Σ, such that

1. distinct edges out of the same node are labeled by distinct characters,
2. all paths in $\mathcal{T}_\mathcal{P}$ from the root are labeled by prefixes of the strings in $\mathcal{P}$,
3. for each string P in $\mathcal{P}$ there exists a path in $\mathcal{T}_\mathcal{P}$ from the root labeled by P.

For any node p in the trie $\mathcal{T}_\mathcal{P}$, we denote by $lbl(p)$ the string which labels the path from the root of $\mathcal{T}_\mathcal{P}$ to p and put $len(p) =_{\text{Def}} |lbl(p)|$. Plainly, the map lbl is injective. Additionally, for any edge (p, q) in $\mathcal{T}_\mathcal{P}$, the label of (p, q) is denoted by $lbl(p, q)$.

[1] Modern architectures include assembly instructions for this purpose; for example, the *x86* family provides the `bsf` and `bsr` instructions, whereas the *powerpc* architecture provides the `cntlzw` instruction. For a comprehensive list of machine-independent methods for computing the index of the first and last bit set to 1, see [9].

[2] In the case of NFAs with no ε-transitions, the transition function has the form $\delta : Q \times \Sigma \to \mathscr{P}(Q)$. For the basics on NFAs, the reader is referred to [40].

For a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ over an alphabet Σ, the *maximal trie* of $\mathcal{P}$ is the trie $\mathcal{T}_{\mathcal{P}}^{max}$ obtained by merging into a single node the roots of the linear tries $\mathcal{T}_{\mathcal{P}_1}, \mathcal{T}_{\mathcal{P}_2}, \ldots, \mathcal{T}_{\mathcal{P}_r}$ relative to the patterns $P_1, P_2, \ldots, P_r$, respectively. Strictly speaking, the maximal trie is a nondeterministic trie, as property (i) above may not hold at the root.

The directed acyclic word graph (DAWG) for a finite set of patterns $\mathcal{P}$ is a data structure representing the set $Fact(\mathcal{P})$. To describe it precisely, we need the following definitions. Let us denote by $end\text{-}pos(u)$ the set of all positions in $\mathcal{P}$ where an occurrence of u ends, for $u \in \Sigma^*$; more formally, we put

$$end\text{-}pos(u) =_{\text{Def}} \{(P, j) \mid u \sqsupseteq P_j, \text{ with } P \in \mathcal{P} \text{ and } |u| - 1 \leq j < |P|\}.$$

For instance, we have $end\text{-}pos(\varepsilon) = \{(P, j) \mid P \in \mathcal{P} \text{ and } -1 \leq j < |P|\}$, since $\varepsilon \sqsupseteq P_j$, for each $P \in \mathcal{P}$ and $-1 \leq j < |P|$ (we recall that $P_{-1} = \varepsilon$, by convention).

We also define an equivalence relation $R_{\mathcal{P}}$ over Σ^* by putting

$$u \; R_{\mathcal{P}} \; v = \; end\text{-}pos(u) = end\text{-}pos(v)\,, \tag{2}$$

for $u, v \in \Sigma^*$, and denote by $R_{\mathcal{P}}(u)$ the equivalence class of $R_{\mathcal{P}}$ containing the string u. Also, we put

$$val(R_{\mathcal{P}}(u)) =_{\text{Def}} \text{the longest string in the equivalence class } R_{\mathcal{P}}(u)\,. \tag{3}$$

Then the DAWG for a finite set $\mathcal{P}$ of patterns is a directed acyclic graph (V, E) with an edge labeling function $lbl()$, where

$$V = \{R_{\mathcal{P}}(u) \,|\, u \in Fact(\mathcal{P})\}$$

$$E = \{(R_{\mathcal{P}}(u), R_{\mathcal{P}}(uc)) \mid u \in \Sigma^*, c \in \Sigma, uc \in Fact(\mathcal{P})\},$$

and $lbl(R_{\mathcal{P}}(u), R_{\mathcal{P}}(uc)) = c$, for $u \in \Sigma^*$, $c \in \Sigma$ such that $uc \in Fact(\mathcal{P})$ (cf. [13]).

3 Exact String Matching

Given a text t of length n and a pattern P of length m over some alphabet Σ of size σ, the *exact single string matching problem* consists in finding *all* occurrences of the pattern P in the text t.

Applications require two kinds of solutions depending on which string, the pattern or the text, is given first. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem. This kind of problem is generally referred as *online* string matching. The notion of indexes realized by trees or automata is used instead in the second kind of problem, generally referred as *offline* string matching. In this paper we are only interested in algorithms of the first kind.

Online string matching algorithms (hereafter simply string matching algorithms) can be divided into three classes: algorithms which solve the problem by making use only of comparisons between characters, algorithms which make use of deterministic automata and algorihtms which simulate nondeterministic automata using bit-parallelism.

Most string matching algorithms generally work as follows. They scan the text with the help on a window of the text whose size is generally equal to m. For each

window of the text they check the occurrence of the pattern (this specific work is called an *attempt*) by comparing the characters of the window with the characters of the pattern, or by performing transitions on some kind of automaton, or by using some kind of filtering method. After a whole match of the pattern or after a mismatch they shift the window to the right by a certain number of positions. This mechanism is usually called the *sliding window mechanism*. At the beginning of the search they align the left ends of the window and the text, then they repeat the sliding window mechanism until the right end of the window goes beyond the right end of the text. We associate each attempt with the position s in the text where the window is positioned, i.e., $T[s\,..\,s+m-1]$.

The worst case lower bound of the string matching problem is $\mathcal{O}(n)$. The first algorithm to reach the bound was given by Morris and Pratt [51] later improved by Knuth, Morris and Pratt [48]. The reader can refer to Section 7 of [48] for historical remarks. Linear algorithms have been developed also based on bit-parallelism [10]. An average lower bound in $\mathcal{O}(n \log m/m)$ (with equiprobability and independence of letters) has been proved by Yao in [66].

Many string matching algorithms have been also developed to obtain sublinear performance in practice (see [24]). Among them the Boyer-Moore algorithm [14] deserves a special mention, since it has been particularly successful and has inspired much work. Among the most efficient comparison based algorithms we mention the well known Horspool [41] and Quick-Search [63] algorithms which, despite their quadratic worst case time complexity, show a sublinear behavior.

Automata based solutions have been also developed to design algorithms which have optimal sublinear performance on average. This is done by using factor automata [12,2], data structures which identify all factors of a word. Among them the Backward Oracle Matching algorithm [2] is one of the most efficient algorithms especially for long patterns.

For short patterns, algorithms based on bit-parallelism are among the most efficient in practice. We will review them, especially the most recent ones, in the following. The reader is referred to [34] for a recent survey and to [33] for an experimental comparison study.

3.1 Standard Algorithms

The Shift-And algorithm simulates the behavior of the non-deterministic string matching automaton (*NSMA*, for short) that recognizes the language $\Sigma^* P$ for a given string P of length m.

The bit-parallel representation of the automaton $NSMA(P)$ uses an array B of $|\Sigma|$ bit-vectors, each of size m, where the i-th bit of $B[c]$ is set iff $\delta(q_i, c) = q_{i+1}$ or equivalently iff $P[i] = c$, for $c \in \Sigma$, $0 \leq i < m$. Automaton configurations $\delta^*(q_0, S)$ on input $S \in \Sigma^*$ are then encoded as a bit-vector D of m bits (the initial state does not need to be represented, as it is always active), where the i-th bit of D is set iff state q_{i+1} is active, i.e. $q_{i+1} \in \delta^*(q_0, S)$, for $i = 0, \ldots, m-1$. For a configuration D of the NFA, a transition on character c can then be implemented by the following bitwise operations

$$D \leftarrow ((D \ll 1) \mid 1) \;\&\; B[c]\,.$$

The bitwise Or with 1 (represented as $0^{m-1}1$) is performed to take into account the self-loop labeled with all the characters in Σ on the initial state. When a search starts, the initial configuration D is initialized to 0^m. Then, while the text is read

from left to right, the automaton configuration is updated for each text character, as described before. If the final state is active after reading character at position j in the text we report a match at position $j - m + 1$.

The Shift-Or algorithm [10] uses the complementary technique of the Shift-And algorithm. In particular an active state of the automaton is represented with a zero bit while ones represent non active states. The algorithm updates the state vector D in a similar way as in the Shift-And algorithm, but is based on the following basic shift-or operation:

$$D \leftarrow (D \ll 1) \mid B[c].$$

Then an occurrence of the pattern is reported if the bit which identifies the final state is set to 0. Both Shift-And and Shift-Or algorithms achieve $O(n\lceil m/\omega \rceil)$ worst-case time and require $O(\sigma\lceil m/\omega \rceil)$ extra-space.

The Backward-Non-deterministic-DAWG-Matching algorithm (BNDM) simulates the non-deterministic factor automaton for $\bar{P}$ with the bit-parallelism technique, using an encoding similar to the one described before for the Shift-And algorithm.

The BNDM algorithm encodes configurations of the automaton in a bit-vector D of m bits (the initial state and state q_0 are not represented). The i-th bit of D is set iff state q_{i+1} is active, for $i = 0, 1, \ldots, m-1$, and D is initialized to 1^m, since after the ε-closure of the initial state I all states q_i represented in D are active. The first transition on character c is implemented as $D \leftarrow (D \;\&\; B[c])$, while any subsequent transition on character c can be implemented as

$$D \leftarrow ((D \ll 1) \;\&\; B[c])\,.$$

The BNDM algorithm works by shifting a window of length m over the text. Specifically, for each window alignment, it searches the pattern by scanning the current window backwards and updating the automaton configuration accordingly. Each time a suffix of $\bar{P}$ (i.e., a prefix of P) is found, namely when prior to the left shift the m-th bit of $D\&B[c]$ is set, the window position is recorded. An attempt ends when either D becomes zero (i.e., when no further prefixes of P can be found) or the algorithm has performed m iterations (i.e., when a match has been found). The window is then shifted to the start position of the longest recognized proper prefix. The time and space complexities of the BNDM algorithm are $\mathcal{O}(\lceil m^2/\omega \rceil)$ and $\mathcal{O}(\sigma\lceil m/\omega \rceil)$, respectively.

The bit-parallel encoding used in the Shift-And and BNDM algorithms requires one bit per pattern symbol, for a total of $\lceil m/\omega \rceil$ computer words. Bit-parallel algorithms are extremely fast as long as a pattern fits in a computer word. Specifically, when $m \leq \omega$, the Shift-And and BNDM algorithms achieve $\mathcal{O}(n)$ and $\mathcal{O}(nm)$ time complexity, respectively, and require $\mathcal{O}(\sigma)$ extra space.

When the pattern size m is larger than ω, the configuration bit-vector and all auxiliary bit-vectors need to be splitted over $m\omega$ multiple words. For this reason the performance of the Shift-And and BNDM algorithms, and of bit-parallel algorithms more in general, degrades considerably as $m\omega$ grows.

Much work has been done in recent years in order to improve the performance of bit parallel algorithms. In the following sections we review in details such solutions.

3.2 Fast Variants of the BNDM Algorithm

A simplified version of the BNDM algorithm (SBNDM for short) has been presented in [61]. Independently, Navarro has adopted a similar approach earlier in the code of

his NR-grep [55]. The SBNDM algorithm differs from the original algorithm in the main loop where it skips the examining of longest prefixes. If s is the current alignment position in the text and j is the number of updates done in the window, then the algorithm simply sets $s + m - j + 1$ to be the start position of the next alignment. Moreover, if a complete match is found, the algorithm advances the window of δ positions to the right, where the value δ is the length of the longest prefix of the pattern which is also a suffix of P. The value of δ is computed in a preprocessing phase in $\mathcal{O}(m)$-time.

Despite the fact that the average length of the shift is reduced, the innermost loop of the algorithm becomes simpler leading to better performance in practice.

Another fast variant of the BNDM algorithm was presented during a talk [39]. When the pattern is aligned with the text window $T[s \,..\, s + m - 1]$ the state vector D is not initialized to 1^m, but is initialized according with the rightmost 2 characters of the current window. More formally the algorithm initializes the bit mask D as

$$D \leftarrow (B[T[s + m - 1]] \ll 1) \;\&\; B[T[s + m - 2]].$$

Then the main loop starts directly with a test on D. If $D = 0$ the algorithm performs directly a shift of $m - 1$ positions to the right. Otherwise a standard BNDM loop is executed starting at position $s + m - 3$ of the text.

The resulting algorithm, called BNDM2, turns out to be faster than the original BNDM algorithm in practical cases. Moreover the same improvements presented above can be applied also to BNDM2, obtaining the variant SBNDM2.

In [39] the authors presented also two improvements of the BNDM algorithm by combining it with the Horspool algorithm [41] according to the dominance of either methods. If the BNDM algorithm dominates then they suggest to use for shifting a simple modification of the Horspool bad character rule [41]. In particular, if the test in the main loop finds D equal to 0, the algorithm shifts the current window of $d[T[s + 2m - 1]]$ positions to the right, where the function $d : \Sigma \to \{m + 1, \ldots, 2m\}$ is defined as $d(c) = m + hbc_{_P}(c)$, for $c \in \Sigma$. If D if found to be greater than 0, then a standard loop of the BNDM algorithm is performed followed by a standard shift. The resulting algorithm is called BNDM-BMH.

Otherwise, if the Horspool algorithm dominates, the authors suggest to replace the standard naive check of Horspool algorithm with a loop of the BNDM algorithm, which generally is faster and simpler. At the end of each verification the pattern is advanced according to the shift proposed by the BNDM algorithm, which increases the shift defined by table d for the last symbol of the pattern. The resulting variant is called BMH-BNDM.

All variants of the BNDM algorithm listed above maintain the same $\mathcal{O}(n\lceil m/w \rceil)$-time and $\mathcal{O}(\sigma\lceil m/w \rceil)$-space complexity of the original algorithm.

3.3 Forward SBNDM Algorithm

The Forward-SBNDM algorithm [32] (FSBNDM for short) is the bit-parallel version of the Forward-BOM algorithm [32].

The algorithm makes use of the non-deterministic automaton $NDawg(\bar{P})$ augmented of a new initial state in order to take into account the *forward character* (character next to the right of the window) of the current window of the text. The resulting automaton has $m+1$ different states and needs $m+1$ bits to be represented.

Thus the FSBNDM algorithm is able to search only for patterns with $1 \leq m < \omega$, where ω is the dimension of a word in the target machine.

For each character $c \in \Sigma$, a bit vector $B[c]$ of length $m+1$ is initialized in the preprocessing phase. The i-th bit is 1 in this vector if c appears in the reversed pattern in position $i-1$, otherwise it is 0. The bit of position 0 is always set to 1.

According to the SBNDM and FBOM algorithms the main loop of each iteration starts by initializing the state vector D with two consecutive text characters (including the forward character) as follows

$$D \leftarrow (B[T[j+1]] \ll 1) \ \& \ B[T[j]]$$

where j is the right end position of the current window of the text.

Then, if $D \neq 0$, the same kind of right to left scan in a window of size m is performed as in the SBNDM, starting from position $j-1$. Otherwise, if $D = 0$, the window is advanced m positions to the right, instead of $m-1$ positions as in the SBNDM algorithm. The resulting algorithm obtains the same $\mathcal{O}(n\lceil m/w \rceil)$-time complexity of the BNDM algorithm but turns out to be faster in practical cases, expecially for small alphabets and short patterns.

3.4 Two-Way-Non-deterministic-DAWG-Matching Algorithm

The Two-Way-Non-deterministic-DAWG-Matching algorithm (TNDM for short) was introduced in [61]. It is a two way variant of the BNDM algorithm which uses a backward search and a forward search alternately.

Specifically, when the pattern P is aligned with the text window $T[j-m+1\,..\,j]$, different cases can be distinguished. If $P[m-1]$ is equal to $T[j]$ or if $T[j]$ does not occur in P the algorithm works as in BNDM by scanning the text from right to left with the automaton $\mathit{NDawg}(\bar{P})$. In contrast, when $T[j] \neq P[m-1]$, but $T[j]$ occurs elsewhere in P, the TNDM algorithm scans forward starting from character $T[j]$.

The main idea is related with the Quick-Search algorithm which uses the text position immediately to the right of the current window of the text for determining the shift advancement. Because $T[j] \neq P[m-1]$ holds, we know that there will be a shift forward anyway before the next occurrence is found. Thus the algorithm examines text characters forward one by one until it finds the first position k such that $T[j\,..\,k]$ does not occur in P or $T[j\,..\,k]$ is a suffix of P. This is done by scanning the text, from left to right, with the automaton $\mathit{NDawg}(P)$.

If a suffix is found the algorithm continues to examine backwards starting from the text position $j-1$. This is done by resuming the standard BNDM operation. To be able to resume efficiently examining backwards, the algorithm preprocesses the length of the longest prefixes of the pattern in the case a suffix of the pattern has been recognized by the BNDM algorithm. This preprocessing can be done in $\mathcal{O}(m)$-time and -space complexity.

Experimental results presented in [61] indicate that on the average the TNDM algorithm examines less characters than BNDM. However average running time is worse than BNDM.

In order to improve the performance the authors proposed further enhancements of the algorithm. In particular if the algorithm finds a character which does not occur in P, while scanning forward, it shifts the pattern entirely over it. This test is computationally light because after a forward scan only the last character can be

missing from the pattern. The test reduces the number of fetched characters but is beneficial only for large alphabets. The resulting algorithm has been called TNBMa.

Finally in [39] the authors proposed a further improvement of the TNBM algorithm. They observed that generally the forward scan for finding suffixes dominates over the BNDM backward scan. Thus they suggested to substitute the backward BNDM check with a naive check of the occurrence, when a suffix is found. The algorithm was called Forward-Non-deterministic-DAWG-Matching (FNDM for short). It achieves better results on average than the TNDM algorithm.

All the algorihtms listed above have an $\mathcal{O}(n\lceil m/w\rceil)$-time complexity and require $\mathcal{O}(\sigma\lceil m/w\rceil)$ extra space.

3.5 Bit Parallel Wide Window Algorithm

The Bit Parallel Wide Window algorithm [38] (BPWW for short) is the bit parallel version of the Wide-Window algorithm [38].

The BPWW algorithm divides the text in $\lceil n/m\rceil$ consecutive windows of length $2m-1$. Each search attempt, on the text window $T[j-m+1\,..\,j+m-1]$ centered at position j, is divided into two steps. The first step consists in scanning the m rightmost characters of the window, i.e. the subwindow $T[j\,..\,j+m-1]$, from left to right, using the automaton $NDawg(P)$, until a full match occurs or the vector state which encodes the automaton becomes zero. At the end of the first step the BPWW algorithm has computed the length ℓ of the longest suffix of P in the right part of the window. If $\ell > 0$, the second step is performed. It consists in scanning the $m-1$ leftmost characters of the window, i.e. the subwindow $T[j-m+1\,..\,j-1]$, from right to left using the $NSMA(\bar{P})$ and starting with state ℓ of the automaton. This is done until the length of the remembered suffix of p, given by ℓ, is too small for finding an occurrence of p. Occurrences of p in T are only reported during the second phase.

The BPWW algorithm requires $\mathcal{O}(\sigma\lceil m/w\rceil)$ extra space and inspects $\mathcal{O}(n)$ text characters in the worst case. Moreover it inspects $\mathcal{O}(n\log m/m)$ characters in the average case.

3.6 Shift Vector Matching Algorithm

Many bit parallel algorithms do not remember text positions which have been checked during the previous alignments. Thus, in certain cases, if the shift is shorter than the pattern length some alignment of the pattern may be tried in vain. In [61] an algorithm, called Shift-Vector-Matching (SVM for short), was introduced which maintains partial memory. Specifically the algorithm maintains a bit vector S, called *shift-vector*, which tells those positions where an occurrence of the pattern can or cannot occur. A text position is *rejected* if we have an evidence that an occurrence of the pattern cannot end at that position. For convention a bit set to zero denotes a text position not yet rejected.

The shift-vector has length m and maintains only information corresponding to text positions in the current window. While moving the pattern forward the algorithm shifts also the shift-vector so that the bits corresponding to the old knowledge go off from the shift-vector. Thus the bit corresponding to the end of the pattern is the lowest bit and the shift direction is to the right.

During the preprocessing phase the SVM algorithm creates a bit-vector C for each character of the alphabet. In particular for each $c \in \Sigma$, the bit-vector $C[c]$ has a zero

bit on every position where the character c occurs in the pattern, and one elsewhere. Moreover the algorithm initializes the shift-vector S, in order to keep track of possible end positions of the pattern, by setting all bits of S to zero.

During the searching phase the algorithm updates the shift-vector by taking OR with the bit-vector corresponding to text character aligned with the rigthmost character of the pattern. Then, if the lowest bit of S is one, a match cannot end here and the algorithm shifts the pattern of ℓ positions to the right, where ℓ is the number of ones which preceed the rightmost zero in the shift-vector S. In addition the SVM algorithm also shifts the shift-vector of ℓ positions to the right.

Otherwise, if the lowest bit in S is zero the algorithm continues by naively checking text characters for the match. In addition the shift-vector S is updated with all characters that were fetched during verifying of alignments.

The value of ℓ is efficiently computed by using the bitwise operations $\ell = \mathsf{bsf}(\sim(S \gg 1)) + 1$, where we recall that the bsf function returns the number of zero bits before the leftmost bit set to 1.

The resulting algorithm has an $\mathcal{O}(n\lceil m/w \rceil)$ worst case time complexity and requires $\mathcal{O}(\sigma\lceil m/w \rceil)$ extra space. However the SVM algorithm is sublinear in practice, because at the same alignment it fetches the same text characters as the Horspool algorithm and can never make shorter shifts.

3.7 Average Optimal Algorithms

Fredriksson and Grabowski presented in [35] a new bit-parallel algorithm, based on Shift-Or, with an optimal average running time, as well as optimal $\mathcal{O}(n)$ worst case running time, if we assume that the pattern representation fits into a single computer word. The algorithm is called Average-Optimal-Shift-Or algorithm (AOSO for short). Experimental results presented by the authors show that the algorithm is the fastest in most of the cases in which it can be applied displacing even the BNDM family of algorithms.

Specifically the algorithm takes a parameter q, which depends on the length of the pattern. Then from the original pattern P a set $\mathcal{P}$ of q new patterns is generated, $\mathcal{P} = \{P^0, P^1, \ldots, P^{q-1}\}$, where each P^j has length $m' = \lfloor m/q \rfloor$ and is defined as

$$P^j[i] = P[j + iq], \quad j = 0, \ldots, q-1, \quad i = 0, \ldots, \lfloor m/q \rfloor - 1$$

The total length of the pattern P^j is $q\lfloor m/q \rfloor \leq m$.

The set of patterns is then searched simultaneously using the Shift-Or algorithm. All the patterns are preprocessed together, in a similar way as in the Shift-Or algorithm, as if they were concatenated in a new pattern $P' = P^0 P^1 \cdots P^{q-1}$. Moreover the algorithm initializes a mask M which has the bits of position $(j+1)m'$ set to 1, for all $j = 0, \ldots, q-1$.

During the search the set $\mathcal{P}$ is used as a filter for the pattern P, so that the algorithm needs only to scan every q-th character of the text. If the pattern P^j matches, then the $(j+1)m'$-th bit in the bit vector D is zero. This is detected with the test $(D\&M) \neq M$. The bits in M have also to be cleared in D before the shift operation, to correctly initialize the first bit corresponding to each of the successive patterns. This is done by the bitwise operation $(D\& \sim M)$.

If P^j is found in the text, the algorithm naively verifies if P also occurs, with the corresponding alignment. To efficiently identify which patterns in $\mathcal{P}$ match, the algorithm sets $D \leftarrow (D\&M)^\wedge M$, so that the $(j+1)m'$-th bit in D is set to 1 if P^j

matches and all other bits are set to 0. Then the algorithm extracts the index j of the highest bit in D set to 1 with the operation

$$b \leftarrow \lfloor \log_2(D) \rfloor, \quad j \leftarrow \lfloor b/m' \rfloor$$

The corresponding text alignment is then verified. Finally, the algorithm clears the bit b in D and repeats the verification until D becomes 0.

In order to keep the total time at most $\mathcal{O}(n/q)$ on average, it is possible to select q so that $n/q = mn/\sigma^{m/q}$, i.e. $q = \mathcal{O}(m/log_\sigma m)$. the total average time is therefore $\mathcal{O}(n \log_\sigma m/m)$, which is optimal.

3.8 Bit-Parallel Algorithms with *q*-grams

The idea of using q-grams for shifting was applied successfully to bit parallel algorithms in [30].

First, the authors presented a variation of the BNDM algorithm called BNDMq. Specifically, at each alignment of the pattern with the current window of the text ending at position j, the BNDMq algorithm first reads a q-gram before testing the state vector D. This is done by initializing the state vector D at the beginning of the iteration in the following way

$$D \leftarrow B[T[j-q+1]] \ \& \ (B[T[j-q]] \ll 1) \ \& \ \cdots \ \& \ (B[T[j]] \ll (q-1)).$$

If the q-gram is not present in the pattern the algorithm quickly advances the window of $m - q + 1$ positions to the right. Otherwise the inner while loop of the BNDM algorithm checks the alignment of the pattern in the right-to-left order. In the same time the loop recognizes prefixes of the pattern. The leftmost found prefix determines the next alignment of the algorithm.

The authors presented also a simplified variant of the BNDMq algorithm (called SBNDMq) along the same line of the SBNDM algorithm [61,55] (see Section 3.2).

Finally the authors presented also an efficient q-grams variant of the Forward-Non-deterministic-DAWG-Matching algorithm [39]. The resulting algorithm is called UFNDMq, where the letter U stands for *upper bits* because the algorithm utilizes those in the state vector D.

The idea of the original algorithm is to read every m-th character c of the text while c does not occur in the pattern. If c is present in the pattern, the corresponding alignments are checked by the naive algorithm. However, while BNDM and its descendants apply the Shift-And approach, FNDM uses Shift-Or.

Along the same line of BNDMq, the UFNDMq algorithm reads q characters before testing the state vector D. Formally the state vector D is initialized as follow

$$D \leftarrow B[T[j]] \mid (B[T[j-1]] \ll 1) \mid \cdots \mid (B[T[j-q+1]] \ll (q-1)).$$

A candidate is naively checked only if at least q characters are matched.

Finally we notice that a similar approach adopted in [47] can be used for BNDMq and SBNDMq. In particular in [30] the authors developed three versions for both BNDMq and SBNDMq.

3.9 Bit-(Parallelism)^2 Algorithms for Short Patterns

Bit-parallelism^2 is a technique recently introduced in [22] which increases the *instruction level parallelism* in string matching algorithms, a measure of how many operations in an algorithm can be performed simultaneously.

The idea is quite simple: when the pattern size is small enough, in favorable situations it becomes possible to carry on in parallel the simulation of multiple copies of a same NFA or distinct NFAs, thus getting to a second level of parallelism.

Two different approaches have been presented. According to the first approach, if the algorithm searches for the pattern in fixed-size text windows then, at each attempt, it processes simultaneously two (adjacent or partially overlapping) text windows by using in parallel two copies of a same automaton.

Differently, according to the second approach, if each search attempt of the algorithm can be divided into two steps (which possibly make use of two different automata) then it executes simultaneously the two steps.

By way of demonstration the authors applied the two approaches to the bit-parallel version of the Wide-Window algorithm [38], but their approaches can be applied as well to other (more efficient) solutions based on bit-parallelism.

In both variants of the BPWW algorithm(see Section 3.5), a word of ω bits is divided into *two* blocks, each being used to encode a *NDawg*. Thus, the maximum length of the pattern gets restricted to $\lfloor \omega/2 \rfloor$. Moreover both of them searches for all occurrences of the pattern by processing text windows of fixed size $2m-1$, where m is the length of the pattern. For each window, centered at position j, the two algorithms computes the sets $\mathcal{S}_j$ and $\mathcal{P}_j$, defined as the sets all starting positions (in P) of the suffixes (and prefixes, respectively) of P aligned with position j in T.

More formally

$$\mathcal{S}_j = \{0 \le i < m \mid P[i\,..\,m-1] = T[j\,..\,j+m-1-i]\}\,;$$
$$\mathcal{P}_j = \{0 \le i < m \mid P[0\,..\,i] = T[j-i\,..\,j]\}\,.$$

Taking advantage of the fact that an occurrence of P is located at position $(j-k)$ of T if and only if $k \in \mathcal{S}_j \cap \mathcal{P}_j$, for $k = 0, \ldots, m-1$, the number of all the occurrences of p in the attempt window centered at j is readily given by the cardinality $|\mathcal{S}_j \cap \mathcal{P}_j|$.

In the bit-parallel implementation of the two variants of the BPWW algorithm the sets $\mathcal{P}$ and $\mathcal{S}$ are encoded by two bit masks PV and SV, respectively. The nondeterministic automata *NDawg*(P) and *NDawg*($\bar{P}$) are then used for searching the suffixes and prefixes of P on the right and on the left parts of the window, respectively. Both automata state configurations and final state configuration can be encoded by the bit masks D and $M = (1 \ll (m-1))$, so that $(D \,\&\, M) \neq 0$ will mean that a suffix or a prefix of the search pattern P has been found, depending on whether D is encoding a state configuration of the automaton *NDawg*(P) or of the automaton *NDawg*($\bar{P}$). Whenever a suffix (resp., a prefix) of length $(\ell+1)$ is found (with $\ell = 0, 1, \ldots, m-1$), the bit $SV[m-1-\ell]$ (resp., the bit $PV[\ell]$) is set by one of the following bitwise operations:

$$\begin{array}{lll} SV \leftarrow SV \;\mid\; ((D \,\&\, M) \gg \ell) & & \text{(in the suffix case)} \\ PV \leftarrow PV \;\mid\; ((D \,\&\, M) \gg (m-1-\ell)) & & \text{(in the prefix case)}\,. \end{array}$$

If we are only interested in counting the number of occurrences of P in T, we can just count the number of bits set in $(SV \,\&\, PV)$. This can be done in $\log_2(\omega)$ operations by using a popcount function, where ω is the size of the computer word in

bits (see [9]). Otherwise, if we want also to retrieve the matching positions of P in T, we can iterate over the bits set in $(SV \,\&\, PV)$ by repeatedly computing the index of the highest bit set and then masking it. The function that computes the highest bit set of a register x is $\lfloor \log_2(x) \rfloor$, and can be implemented efficiently in either a machine dependent or machine independent way (see again [9]).

In the first variant (based on the first approach), named Bit-Parallel Wide-Window[2] (BPWW2, for short), two partially overlapping windows size $2m-1$, centered at consecutive attempt positions $j-m$ and j, are processed simultaneously. Two automata are represented in a single word and updated in parallel.

Specifically, each search phase is again divided into two steps. During the first step, two copies of $\mathit{NDawg}(P)$ are operated in parallel to compute simultaneously the sets $\mathcal{S}_{j-m}$ and $\mathcal{S}_j$. Likewise, in the second step, two copies of $\mathit{NDawg}(\bar{P})$ are operated in parallel to compute the sets $\mathcal{P}_{j-m}$ and $\mathcal{P}_j$.

To properly detect suffixes in both windows, the bit mask M is initialized as

$$M \leftarrow (1 \ll (m+k-1)) \mid (1 \ll (m-1))$$

and transitions are performed in parallel with the following bitwise operations

$$\begin{array}{ll} D \leftarrow (D \ll 1) \,\&\, ((B[T[j-m+\ell]] \ll k) \mid B[T[j+\ell]]) & \text{(in the first phase)} \\ D \leftarrow (D \ll 1) \,\&\, ((C[T[j-m-\ell]] \ll k) \mid C[T[j-\ell]]) & \text{(in the second phase)}\,, \end{array}$$

for $\ell = 1, \ldots, m-1$ (when $\ell = 0$, the left shift of D does not take place).

Since two windows are simultaneously scanned at each search iteration, the shift becomes $2m$, doubling the length of the shift with respect to the BPWW algorithm.

The second variant of the BPWW algorithm (based on the second approach) was named Bit-Parallel[2] Wide-Window algorithm (BP2WW, for short). The idea behind it consists in processing a single window at each attempt (as in the original BPWW algorithm) but this time by scanning its left and right sides simultaneously.

Automata state configurations are again encoded simultaneously in a same bit mask D. Specifically, the most significant k bits of D encode the state of the suffix automaton $\mathit{NDawg}(P)$, while the least significant k bits of D encode the state of the suffix automaton $\mathit{NDawg}(\bar{P})$. The BP2WW algorithm uses the following bitwise operations to perform transitions[3] of both automata in parallel:

$$D \leftarrow (D \ll 1) \,\&\, ((B[T[j+\ell]] \ll k) \mid C[T[j-\ell]])\,,$$

for $\ell = 1, \ldots, m-1$. Note that in this case the left shift of k positions can be precomputed in B by setting $B[c] \leftarrow B[c] \ll k$, for each $c \in \Sigma$.

Using the same representation, the final-states bit mask M is initialized as

$$M \leftarrow (1 \ll (m+k-1)) \mid (1 \ll (m-1))\,.$$

At each iteration around an attempt position j of T, the sets $\mathcal{S}_j$ and $\mathcal{P}^*_j$ are computed, where $\mathcal{S}_j$ is defined as in the case of the BPWW algorithm, and $\mathcal{P}^*_j$ is defined as $\mathcal{P}^*_j = \{0 \le i < m \mid P[0\,..\,m-1-i] = T[j-(m-1-i)\,..\,j]\}$, so that $\mathcal{P}_j = \{0 \le i < m \mid (m-1-i) \in \mathcal{P}^*_j\}$.

The sets $\mathcal{S}_j$ and $\mathcal{P}^*_j$ can be encoded with a single bit mask PS, in the rightmost and the leftmost k bits, respectively. Positions in $\mathcal{S}_j$ and $\mathcal{P}^*_j$ are then updated simultaneously in PS by executing the following operation:

$$PS \leftarrow PS \mid ((D \,\&\, M) \gg \ell)\,.$$

[3] For $\ell = 0$, D is simply updated by $D \leftarrow D \,\&\, ((B[T[j+l]] \ll k) \mid C[T[j-l]])$.

At the end of each iteration, the bit masks SV and PV are retrieved from PS with the following bitwise operations:

$$PV \leftarrow reverse(PS) \gg (\omega - m)\,, \qquad SV \leftarrow PS \gg k\,,$$

In fact, to obtain the correct value of PV we used bit-reversal modulo m, which has been easily achieved by right shifting $reverse(PS)$ by $(\omega - m)$ positions. We recall that the *reverse* function can be implemented efficiently with $\mathcal{O}(\log_2(\omega))$ operations.

Both BPWW2 and BP2WW algorithms need $\lceil m/\omega \rceil$ words to represent all bit masks and have an $\mathcal{O}(n\lceil m/\omega \rceil + \lfloor n/m \rfloor \log_2(\omega))$ worst case time complexity.

3.10 The Bit-Parallel Length Invariant Matcher

The general problem in all bit parallel algorithms is the limitation defined on the length of the input pattern, which does not permit efficient searching of strings longer than the computer word size.

In [49] the author proposed a method, based on bit parallelism, with the aim of searching patterns independently of their lengths. The algorithm was called Bit-Parallel Length Invariant Matcher (BLIM for short). In contrast with the previous bit parallel algorithms, which require the pattern length not to exceed the computer word size, BLIM defines a unique way of handling strings of any length.

Given a pattern P, of length m, the algorithm ideally computes an alignment matrix A which consists of ω rows, where ω is the size of a word in the target machine. Each row row_i , for $0 \leq i < \omega$, contains the pattern right shifted by i characters. Thus, A contains $wsize = \omega + m - 1$ columns. During the preprocessing phase the algorithm computes a mask matrix M of size $|\Sigma| \times wsize$, where $M[c, h] = b_{\omega-1} \dots b_1 b_0$ is a bit vector of ω bits, for $c \in \Sigma$ and $0 \leq h < wsize$. Specifically the i-th bit, b_i of $M[c, h]$ is defined as

$$b_i = \begin{cases} 0 & \text{if } (0 \leq h - i < m) \text{ and } (c = P[h - i]) \\ 1 & \text{otherwise .} \end{cases}$$

The main idea of BLIM is to slide the alignment matrix over the text, on windows of size *wsize*, and at each attempt check for any possible placements of the pattern.

The characters of the window are visited in order to perform the minimum number of character accesses. Specifically the algorithm checks the characters at positions $m - i, 2m - i, \dots, km - i$, where $km - i < wsize$, for $i = 1, 2, \dots, m$ in order. The main idea behind this ordering is to investigate the window in such a way that at each character access a maximum number of alignments is checked. The scan order is precomputed and stored in a vector S of size *wsize*.

When the window is located at $T[i \,..\, i + wsize - 1]$, a flag variable F is initialized to the mask value $M[T[i+S[0]], S[0]]$. The traversal of other characters of the window continues by performing the following basic bitwise operation

$$F \leftarrow F \& M[T[i + S[j]], S[j]]$$

for $j = 1, \dots, wsize$, till the flag F becomes 0 or all the characters are visited. If F becomes 0, this implies that the pattern does not exist on the window. Otherwise, one or more occurrences of the pattern are detected at the investigated window. In that case, the 1 bits of the flag F tells the exact positions of occurrences.

At the end of each attempt the BLIM algorithm uses the shift mechanism proposed in the Quick-Search algorithm [42]. The immediate text character following the

window determines the shift amount. If that character is included in the pattern then the shift amount is $wsize - k$, where $k = \max\{i \mid P[i] = T[s + wsize]\}$, otherwise the shift value is equal to $wsize + 1$.

The BLIM algorithm has a $\mathcal{O}(\lceil n/\omega \rceil(\omega+m-1))$ overall worst case time complexity and requires $\mathcal{O}(\Sigma \times (\omega + m - 1))$-extra space.

3.11 Bit-Parallel Algorithms for Long Patterns

In [57] the authors introduced an efficient method, based on bit-parallelism, to search for patterns longer than w. Their approach consists in constructing an automaton for a substring of the pattern fitting in a single computer word, to filter possible candidate occurrences of the pattern. When an occurrence of the selected substring is found, a subsequent naive verification phase allows to establish whether this belongs to an occurrence of the whole pattern. However, besides the costs of the additional verification phase, a drawback of this approach is that, in the case of the BNDM algorithm, the maximum possible shift length cannot exceed w, which could be much smaller than m.

Later in [61] another approach for long patterns was introduced, called LBNDM. In this case the pattern is partitioned in $\lfloor m/k \rfloor$ consecutive substrings, each consisting in $k = \lfloor (m-1)/\omega \rfloor + 1$ characters. The $m - k\lfloor m/k \rfloor$ remaining characters are left to either end of the pattern. Then the algorithm constructs a superimposed pattern P' of length $\lfloor m/k \rfloor$, where $P'[i]$ is a class of characters including all characters in the i-th substring, for $0 \leq i < \lfloor m/k \rfloor$.

The idea is to search first the superimposed pattern in the text, so that only every k-th character of the text is examined. This filtration phase is done with the standard BNDM algorithm, where only the k-th characters of the text are inspected. When an occurrence of the superimposed pattern is found the occurrence of the original pattern must be verified.

The shifts of the LBNDM algorithm are multiples of k. To get a real advantage of shifts longer than that proposed by the approach of Navarro and Raffinot, the pattern length should be at least about two times ω.

More recently Durian *et al.* presented in [31] another efficient algorithm for simulating the suffix automaton in the case of long patterns. The algorithm is called BNDM with eXtended Shift (BXS). The idea is to cut the pattern into $\lceil m/w \rceil$ consecutive substrings of length w except for the rightmost piece which may be shorter. Then the substrings are superimposed getting a superimposed pattern of length w. In each position of the superimposed pattern a character from any piece (in corresponding position) is accepted. Then a modified version of BNDM is used for searching consecutive occurrences of the superimposed pattern using bit vectors of length w but still shifting the pattern by up to m positions. The main modfication in the automaton simulation consists in moving the rightmost bit, when set, to the

first position of the bit array, thus simulating a circular automaton. Like in the case of the LBNDM, algorithm the BXS algorithm works as a filter algorithm, thus an additional verification phase is needed when a candidate occurrence has been located.

3.12 A bit-parallel algorithm for small alphabets

The bit-parallel algorithm for small alphabets (SABP for short) [67] consists in scanning the text with a window of size $\ell = \max\{m + 1, \omega\}$. At each attempt, where the

window is positioned on $T[j\,..\,j+\ell-1]$ it maintains a vector of ω bits whose bit at position $\omega-1-i$ is set to 0 if P cannot be equal to $T[j+i\,..\,j+m-1+i]$.

The preprocessing phase consists in computing:

- an array T of $\max\{m,\omega\}\times\sigma$ vectors of ω bits as follows:

$$T[j,c]_{\omega-1-i} = \begin{cases} 0 & \text{if } 0 \le j-i < \omega \text{ and } P[j-i] \neq c \\ 1 & \text{if } j-i \notin [0,\omega) \text{ or } P[j-i] = c \end{cases}$$

  for $0 \le j < \ell$, $0 \le i < \omega$ and $c \in \Sigma$.
- an array T' of σ vectors of ω bits as follows:

$$T'[c] = (T[m-1,c] \gg 1) \quad | \quad 10^{\omega-1}$$

  for $c \in \Sigma$.
- the Quick Search bad character rule qbc_P [42].

Then during the searching phase the algorithm maintains a vector F of ω bits such that when the window of size ℓ is positioned on $T[j\,..\,j+\ell-1]$ the bit at position $\omega-1-i$ of F is equal to 0 if P cannot be equal to $T[j+i\,..\,j+m-1+i]$ for $0 \le j \le n-\ell$ and $0 \le i < \omega$. Initially all the bits of F are set to 1. At each attempt the algorithm first scan $T[j+m-1]$ and $T[j+\omega-1]$ as follows:

$$F = F \quad \& \quad T[m-1,T[j+m-1]] \quad \& \quad T[\omega-1,T[j+\omega-1]]$$

then it scans $T[j+m-2]$ and $T[j+m]$ as follows:

$$F = F \quad \& \quad T[m-2,T[j+m-2]] \quad \& \quad T'[j+m]$$

and finally it scans $T[j+k]$ for $k = m-3,\ldots,0$ as follows:

$$F = F \quad \& \quad T[k,T[j+k]]$$

while $F_{\omega-1} = 1$. If all the characters have been scanned and $F_{\omega-1} = 1$ then an occurrence of the pattern is reported and $F_{\omega-1}$ is set to 0. In all cases a shift of the window is performed by taking the maximum between the Quick Search bad character rule and the difference between ω and the position of the righmost bit of value 1 in F which can be computed by $\omega - \lfloor \log_2 F \rfloor$. The bit-vector F is shifted accordingly.

The preprocessing phase of the SABP algorithm has an $\mathcal{O}(m\sigma)$ time and space complexity and the searching phase has an $\mathcal{O}(mn)$ time complexity.

3.13 Tighter packing for bit-parallelism

In order to overcome the problem due to handling long patterns with bit-parallelism, in [22] a new encoding of the configurations of non-deterministic automata for a given pattern P of length m was presented, which on the average requires less than m bits and is still suitable to be used within the bit-parallel framework. The effect is that bit-parallel string matching algorithms based on such encoding scale much better as m grows, at the price of a larger space complexity (at most of a factor σ). The authors illustrated the application of the encoding to the Shift-And and the BNDM algorithms, obtaining two variants named Factorized-Shift-And and Factorized-BNDM (F-Shift-And and F-BNDM for short). However the encoding can also be applied to other variants of the BNDM algorithm as well.

The encoding has the form (D, a), where D is a k-bit vector, with $k \leqslant m$ (on the average k is much smaller than m), and a is an alphabet symbol (the last text character read) which will be used as a parameter in the bit-parallel simulation with the vector D.

The encoding (D, a) is obtained by suitably factorizing the simple bit-vector encoding for NFA configurations and is based on the 1-factorization of the pattern.

More specifically, given a pattern $P \in \Sigma^m$, a 1-factorization of size k of P is a sequence $\langle u_1, u_2, \ldots, u_k \rangle$ of nonempty substrings of P such that $P = u_1 u_2 \cdots u_k$ and each factor u_j contains at most *one* occurrence for any of the characters in the alphabet Σ, for $j = 1, \ldots, k$. A 1-factorization of P is *minimal* if such is its size.

For $x \in \Sigma^*$, let $\mathit{first}(x) = x[0]$ and $\mathit{last}(x) = x[|x| - 1]$. It can easily be checked that a 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P is minimal if $\mathit{first}(u_{i+1})$ occurs in u_i, for $i = 1, \ldots, k - 1$.

Observe, also, that $\lceil \frac{m}{\sigma} \rceil \leq k \leq m$ holds, for any 1-factorization of size k of a string $P \in \Sigma^m$, where $\sigma = |\Sigma|$. The worst case occurs when $P = a^m$, in which case P has only the 1-factorization of size m whose factors are all equal to the single character string a.

A 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of a given pattern $P \in \Sigma^*$ induces naturally a partition $\{Q_1, \ldots, Q_k\}$ of the set $Q \setminus \{q_0\}$ of nonstarting states of the canonical automaton $\mathit{SMA}(P) = (Q, \Sigma, \delta, q_0, F)$ for the language $\Sigma^* P$.

Hence, for any alphabet symbol a the set of states Q_i contains at most one state with an incoming arrow labeled a. Indicate with symbol $q_{i,a}$ the unique state q of $\mathit{SMA}(P)$ with $q \in Q_i$, and q has an incoming edge labeled a.

In the F-Shift-And algorithm the configuration $\delta^*(q_0, Sa)$ is encoded by the pair (D, a), where D is the bit-vector of size k such that $D[i]$ is set iff Q_i contains an active state, i.e., $Q_i \cap \delta^*(q_0, Sa) \neq \emptyset$, iff $q_{i,a} \in \delta^*(q_0, Sa)$.

For $i = 1, \ldots, k - 1$, we put $\overline{u}_i = u_i.\mathit{first}(u_{i+1})$. We also put $\overline{u}_k = u_k$ and call each set $\overline{u}_i$ the *closure* of u_i. Plainly, any 2-gram can occur at most once in the closure $\overline{u}_i$ of any factor of our 1-factorization $\langle u_1, u_2, \ldots, u_k \rangle$ of P.

In order to encode the 2-grams present in the closure of the factors u_i the algorithm makes use of a $|\Sigma| \times |\Sigma|$ matrix B of k-bit vectors, where the i-th bit of $B[c_1, c_2]$ is set iff the 2-gram $c_1 c_2$ is present in $\overline{u}_i$ or, equivalently, iff

$$\begin{aligned} &(\mathit{last}(u_i) \neq c_1 \wedge q_{i,c_2} \in \delta(q_{i,c_1}, c_2)) \vee \\ &\qquad (i < k \wedge \mathit{last}(u_i) = c_1 \wedge q_{i+1,c_2} \in \delta(q_{i,c_1}, c_2)) \,, \end{aligned} \tag{4}$$

for every 2-gram $c_1 c_2 \in \Sigma^2$ and $i = 1, \ldots, k$.

To properly take care of transitions from the last state in Q_i to the first state in Q_{i+1}, the algorithm makes also use of an array L, of size $|\Sigma|$, of k-bit vectors encoding for each character $c \in \Sigma$ the collection of factors ending with c. More precisely, the i-th bit of $L[c]$ is set iff $\mathit{last}(u_i) = c$, for $i = 1, \ldots, k$.

The matrix B and the array L, which in total require $(|\Sigma|^2 + |\Sigma|)k$ bits, are used to compute the transition $(D, a) \xrightarrow{\mathit{SMA}} (D', c)$ on character c. In particular D' can be computed from D by the following bitwise operations:

$$\text{(i) } D \leftarrow D \;\&\; B[a, c]; \quad \text{(ii) } H \leftarrow D \;\&\; L[a]; \quad \text{(iii) } D \leftarrow (D \;\&\; \sim H) | (H \ll 1) \,.$$

To check whether the final state q_m belongs to a configuration encoded as (D, a), we have only to verify that $q_{k,a} = q_m$. This test can be broken into two steps: first, one checks if any of the states in Q_k is active, i.e. $D[k] = 1$; then, one verifies that

the last character read is the last character of u_k, i.e. $L[a][k] = 1$. The test can then be implemented with the test $D \;\&\; M \;\&\; L[a] \neq 0^k$, where $M = (1 \ll (k-1))$.

The same considerations also hold for the encoding of the factor automaton $Dawg(P)$ in the F-BNDM algorithm. The only difference is in the handling of the initial state. In the case of the automaton $SMA(P)$, state q_0 is always active, so we have to activate state q_1 when the current text symbol is equal to $P[0]$. To do so it is enough to perform a bitwise or of D with $0^{k-1}1$ when $a = P[0]$, as $q_1 \in Q_1$. Instead, in the case of the suffix automaton $Dawg(P)$, as the initial state has an ε-transition to each state, all the bits in D must be set, as in the BNDM algorithm.

The preprocessing procedure which builds the arrays B and L has a time complexity of $\mathcal{O}(|\Sigma|^2 + m)$. The variants of the Shift-And and BNDM algorithms based on the encoding of the configurations of the automata $SMA(P)$ and $Dawg(P)$ (algorithms F-Shift-And and F-BNDM, respectively) have a worst-case time complexities of $\mathcal{O}(n\lceil k/\omega \rceil)$ and $\mathcal{O}(nm\lceil k/\omega \rceil)$, respectively, while their space complexity is $\mathcal{O}(|\Sigma|^2\lceil k/\omega \rceil)$, where k is the size of a minimal 1-factorization of the pattern.

4 Multiple String Matching

Given a set $\mathcal{P}$ of r patterns and a text T of length n, all strings over a common finite alphabet Σ of size σ, the *multiple pattern matching problem* is to determine all the occurrences in T of the patterns in $\mathcal{P}$.

Multiple string matching is an important problem in many application areas of computer science. For example, in computational biology, with the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application and there is an increasing demand for fast computer methods for analysis and data retrieval. Although there are various kinds of comparison tools which provide aligning and approximate matching, most of them are based on exact matching in order to speed up the process. Another important usage of multiple pattern matching algorithms appears in network intrusion detection systems as well as in anti-virus software. The major performance bottleneck of the regarding solutions to these problems is to achieve high-speed matching required to detect malicious patterns of ever growing sets.

The first linear solution for the multiple pattern matching problem based on finite automata is due to Aho and Corasick in [1]. The Aho-Chorasick algorithm uses a deterministic incomplete finite automaton based on the *trie* for the input patterns and on the *failure function*, a generalization of the border function of the Knuth-Morris-Pratt algorithm [48]. The optimal average complexity of the problem is $\mathcal{O}(n \log_\sigma(rl_{min})/l_{min})$ [56], where l_{min} is the length of the shortest pattern in the set $\mathcal{P}$; this bound has been achieved by algorithms based on the suffix automaton induced by the DAWG data structure, namely the Backward-DAWG-Matching (BDM) and Set-Backward-DAWG-Matching (SBDM) algorithms [27,59].

To simulate efficiently an NFA with the bit-parallelism technique, the states of the automaton must be mapped into the positions of a bit-vector by a suitable topological ordering of the NFA.[4]

In the case of a single pattern, the construction of the topological ordering is quite simple, since it is unique [10]. Appropriate topological orderings can be ob-

[4] We recall that a *topological ordering* of an NFA is any total ordering $<$ of the set of its states such that $p < q$, for each edge (p, q) of the NFA.

tained also for the maximal trie of a set of patterns, by interleaving the tries of the single patterns in either a parallel fashion, under the restriction that all the patterns have the same length [65], or in a sequential fashion [57]. The Shift-And and BNDM algorithms can be easily extended to the multiple patterns case by deriving the corresponding automaton from the maximal trie of the set of patterns. The resulting algorithms have a $\mathcal{O}(\sigma\lceil size(\mathcal{P})/w\rceil)$-space complexity and work in $\mathcal{O}(n\lceil size(\mathcal{P})/w\rceil)$ and $\mathcal{O}(n\lceil size(\mathcal{P})/w\rceil l_{min})$ worst-case searching time complexity, respectively, where $size(\mathcal{P}) = \sum_{P\in\mathcal{P}} |P|$ is the sum of the lengths of the strings in $\mathcal{P}$.

In both cases, the bit-parallel simulation is based on the following property of the topological ordering π associated to the trie which allows to encode the transitions using a shift of k bits and a bitwise `and`: for each edge (p, q), the distance $\pi(q) - \pi(p)$ is equal to a constant k. In particular when a parallel simulation is carried out the shift is always of $k = r$ bits.

The above techniques used to extend string matching algorithms based on bit-parallelism to the multiple-string matching problem consists, on a conceptual basis, in sequentially concatenating the automata for each pattern. The drawback of this method is that it is not possible to exploit the prefix redundancy in the patterns, a property which can be significant in the case of small alphabets. The trie and the DAWG data structures make it possible to factor common prefixes in the patterns. However, because of the lack of regularity in such structures, in general, there might be no topological ordering π such that, for each edge (p, q), the distance $\pi(q) - \pi(p)$ is fixed.

Cantone and Faro presented in [20] a bit-parallel simulation of the Aho-Chorasick NFA which is able to encode variable length shifts. Their solution is based on a suitable topological ordering of the NFA, called *weakly safe* topological ordering, and distinguish between 1-bit edges, i.e. edges (p, q) such that $\pi q - \pi(p) = 1$, and long-bit edges, i.e. edges (p, q) such that $\pi q - \pi(p) > 1$. The simulation of long-bit edges is then simulated using the carry property of addition. In particular a topological ordering π is said to be *weakly safe* if, for each $c \in \Sigma$, the π-intervals of any two distinct long-bit edges labeled by a same character and not originating from the root of T are disjoint.

They proposed an algorithm, called Multiple-Trie-Shift-And, with a $\mathcal{O}(\sigma\lceil m/w\rceil)$-space and $\mathcal{O}(n\lceil m/w\rceil)$-searching time complexity, where m is the number of nodes in the trie. The construction of the safe topological ordering is based on a DFS approach and runs in $O(L)$-time and -space, under suitable hypotheses. However, such topological orderings do not always exist and the problem of finding one is probably even intractable.

More recently Cantone, Faro and Giaquinta presented in [23] a new more general approach to the efficient bit-parallel simulation of the Aho-Chorasick NFAs and suffix NFAs. When the prefix redundancy is nonnegligible, this method yields a representation that requires smaller bit-vectors and, correspondingly, less words. Therefore, if we restrict to single-word bit-vectors, it results that more patterns can be packed into a word. The construction is based on a result for the Glushkov automaton [60], which however requires exponential space in the number of states in the NFA to encode the transition function. They show that, by exploiting the relation between active states of the NFA and its associated failure function, it is possible to represent the transition function in polynomial space using a similar encoding.

Indicate with symbol $B(c)$, for $c \in \Sigma$, the set of states with an incoming transition labeled by c, and with symbol $Follow(q)$, for $q \in Q$, the set of states reachable from

state q with one transition over a character in Σ. Their solution is based on the property [60] that, for every $q \in Q$, $D \subseteq Q$, and $c \in \Sigma$, we have $\delta(q, c) = Follow(q) \cap B(c)$ and $\delta(D, c) = \phi(D) \cap B(c)$, which is particularly suitable for bit-parallelism, as set intersection can be readily implemented by the bitwise and operation.

Moreover in order to find an efficient way of storing and accessing the maps $\phi()$ and $B()$ the authors show that each nonempty reachable configuration D can be represented in terms of a unique state, which will be referred to as $lead(D)$. This will allow us to represent $\Phi(D)$ as $\dot{\Phi}(lead(D))$, where $\dot{\Phi} : Q \to \mathscr{P}(Q)$ is the map such that the q-th bit of $\dot{\Phi}(p)$ is set if and only if there is a transition to state q originating from p or any other state belonging to the reachable configuration uniquely identified by p. Plainly, the map $\dot{\Phi}$ can be stored in $\mathcal{O}(m^2)$-space and allows to state that $\delta(D, c) = \dot{\Phi}(lead(D)) \cap B(c)$, which in turn translates readily into the bit-parallel assignment $D \leftarrow \dot{\Phi}[lead(D)]$ & $B[c]$.

They also presented two simple algorithms, based on such a technique, for searching a set $\mathcal{P}$ of patterns in a text T of length n over an alphabet Σ of size σ. The algorithms, named Log-And and Backward-Log-And, require $\mathcal{O}((m+\sigma)\lceil m/w \rceil)$-space, and work in $\mathcal{O}(n\lceil m/w \rceil)$ and $\mathcal{O}(n\lceil m/w \rceil l_{min})$ worst-case searching time, respectively, where w is the number of bits in a computer word, m is the number of states of the automaton, and l_{min} is the length of the shortest pattern in $\mathcal{P}$.

5 Approximate String Matching

Approximate pattern matching is a classic problem in computer science, with applications in various areas, such as spelling correction, bioinformatics, signal processing, musical information retrieval. It has been actively studied since the sixties [54].

Approximate pattern matching consists in general in searching for substrings of a text T that are within a predefined edit distance threshold from a given pattern P.

Let $d(x, y)$ denote the approximate distance between the strings x and y over a common alphabet Σ, and let k be the maximum allowed distance. Using this notation, the task of approximate string matching is to find all positions j in the text such that $d(P, T[i..j]) \leq k$, for some $i \leq j$.

Perhaps the most common form of edit distance is the Levenshtein edit distance [50], which is defined as the minimum number of single-character insertions, deletions and substitutions needed in order to make x and y equal. Other common forms of edit distances have been studied over the years and solved by using bit-parallelism. In what follows we survey solutions on approximate string matching under the Damerau distance, the Swap distance and allowing for gaps.

5.1 String Matching with Levenshtein Distance

Perhaps the most common form of edit distance between two strings P and T is the Levenshtein edit distance [50], which is defined as the minimum number of single-character insertions, deletions and substitutions needed in order to make P and T equal.

In this case the dynamic programming algorithm fills a $(|P|+1)\times(|T|+1)$ dynamic programming table D, where at the end each cell $D[i, j]$ will hold the edit distance between $P[0..i]$ and $T[0..j]$.

Although the algorithm is not very efficient it is among the most flexible ones to adapt to different distance functions.

$$D[i,j] = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ D[i-1,j-1] & \text{if } i,j > 0 \text{ and } P[i] = T[j] \\ 1 + \min(D[i-1,j], D[i,j-1], D[i-1,j-1]) & \text{otherwise} \end{cases} \tag{5}$$

Instead of computing the edit distance between strings P and T, the dynamic programming algorithm can be changed to find approximate occurrences of P somewhere inside T by changing the boundary condition $D[i,j] = j$ with $D[i,j] = 0$, when $i = 0$, that is the empty pattern matches with zero errors at any text position. It was converted into a search algorithm only in by Seller [62].

During the last decade, algorithms based on bit-parallelism have emerged as the fastest approximate string matching algorithms in practice for the Levenshtein edit distance [54].

The algorithm of Meyers [53] is based on representing the dynamic programming table D with vertical, horizontal and diagonal differences. This is done by using the length-m bit-vectors VP, VN (vertical positive and vertical negative vector), HP and HN (horizontal positive and horizontal negative vector). Specifically $VP[i] = 1$ at text position j iff $D[i,j] - D[i-1,j] = 1$, while $VN[i] = 1$ at text position j iff $D[i,j] - D[i-1,j] = -1$. A similar definition holds for HP and HN. In addition a diagonal delta vector R is maintained, where $R[i] = 1$ at text position j iff $D[i,j] = D[i-1,j-1]$. Initially $VP = 1^m$ and $VN = 0^m$. The complete formula for computing the updated vectors at text position j is

$$\begin{aligned} R' &= (((B[T[j]] \;\&\; VP) + VP) \;\wedge VP) \mid B[T[j]] \mid VN \\ HP' &= VN \mid \sim (R' \mid VP) \\ HN' &= VP \;\&\; R' \\ VP' &= (HN' \ll 1) \mid \sim (R' \mid (HP' \ll 1)) \\ VN' &= (HP' \ll 1) \;\&\; R' \end{aligned}$$

The current value of the dynamic programming cell $D[m,j]$ can be updated at each text position j by using the horizontal delta vectors (the initial value is $D[m,0] = m$). A match of the pattern with at most k errors is found whenever $D[m,j] \leq k$. If $m \leq w$, the run time of this algorithm is $O(n)$ as there is again only a constant number of operations per text character. The general run time is $O(\lceil m/w \rceil n)$ as a vector of length m may be simulated in $O(\lceil m/w \rceil)$ time using $O(\lceil m/w \rceil)$ bit-vectors of length w.

The bit-parallel approximate string matching algorithm of Wu and Manber [64] is based on representing a non-deterministic finite automaton (NFA) by using bit-vectors. The automaton has $(k+1)$ rows, numbered from 0 to k, and each row contains m states. Let us denote the automaton as D, its row d as D_r and the state i on its row r as $D_r[i]$. The state $D_r[i]$ is active after reading the text up to the j-th character if and only if $ed(P[0..i], T[h..j]) \leq d$ for some $h \leq j$. An occurrence of the pattern with at most k errors is found when the state $D_k[m]$ is active. Assume for now that $m \leq w$. Wu and Manber represent each row D_r as a length-m bit-vector, where the i-th bit tells whether the state $D_r[i]$ is active or not. In addition they build a length-m match vector for each character in the alphabet. We denote the match vector for the character c as $B[c]$. The i-th bit of $B[c]$ is set if and only if $P[i] = c$. Initially each vector D_r has the value $0^{m-r}1^r$ (this corresponds to the

boundary conditions in Recurrence 1). The formula to compute the updated values D'_r from the row-vectors D_r at text position j is the following

$$
\begin{aligned}
D'_0 &= ((D_0 \ll 1) \mid 1) \;\&\; B[T[j]] \\
D'_r &= ((D_r \ll 1) \;\&\; B[T[j]]) \mid D_{r-1} \mid (D_{r-1} \ll 1) \mid (D'_{r-1} \ll 1) \mid 1, \text{ for } 1 \leq r \leq k.
\end{aligned}
$$

When $m \leq w$, the running time of this algorithm is $O(kn)$ as there are $O(k)$ operations per text character. The general run time is $O(kn\lceil m/w \rceil)$.

In [11] Baeza-Yates and Navarro found a bit parallel formula for a diagonal parallelization of the NFA. They packed the states of the automaton along diagonals instead of rows or columns which run in the same direction of the diagonal arrows. There are $m - k + 1$ complete diagonals, the others are not really necessary, which are numbered from 0 to $m - k$. To describe the status of the i-th diagonal it sufices to record the position of the first active state in it. Thus the number D_i is the row of the first active state in diagonal i all the subsequent states in the diagonal are active because of the transitions. If the first active state on the i-th diagonal is f_i, then Baeza-Yates and Navarro represent the diagonal as the bit-sequence $D_i = 0^{k+1-f_i}1^{f_i}$. A match with at most k errors is found whenever $f_{m-k} < k + 1$. The bit-sequences are stored consecutively with a single separator zero-bit between two consecutive states. Let $\bar{D}$ denote the complete diagonal representation. Then $\bar{D}$ is the length-$(k+2)(m-k)$ bit-sequence $0D_10D_20...0D_{m-k}$. We assume for now that $(k + 2)(m - k) \leq w$ so that $\bar{D}$ fits into a single bit-vector. Baeza-Yates and Navarro encode also the pattern match vectors differently. Let $\bar{B[c]}$ be their pattern match vector for the character. first of all the role of the bits is reversed: a 0-bit denotes a match and a 1-bit a mismatch. To align the matches with the diagonals in $\bar{D}$, $\bar{B[c]}$ has the form $0B_10B_20...0B_{m-k}$, where $B_i = \sim B[c][i..i + k]$. Initially no diagonal has active states and so $\bar{D} = (01^{k+1})^{m-k}$. The formula for updating $\bar{D}$ at text position j is:

$$
\begin{aligned}
x &= (\bar{D} \gg (k + 2)) \mid \bar{B[T[j]]} \\
\bar{D'} &= ((D \ll 1) \mid (0^{k+1}1)^{m-k} \;\&\; (\bar{D} \ll (k + 3)) \mid (0^{k+1}1)^{m-k-1}01^{k+1} \;\& \\
&\quad (((x + (0^{k+1}1)^{m-k}) \;\&\; x) \gg 1) \;\&\; (01^{k+1})^{m-k}
\end{aligned}
$$

If $(k+2)(m-k) \leq w$, the run time of this algorithm is $O(n)$ as there is only a constant number of operations per text character. The general run time is $O(\lceil km/w \rceil n)$ as a vector of length $(k + 2)(m - k)$ may be simulated in $O(\lceil km/w \rceil)$ time using $O(\lceil km/w \rceil)$ bit-vectors of length w.

Later Hyyro proposed a new variant [43] of the bit-parallel NFA of Baeza-Yates and Navarro for approximate string matching. The algorithm decreases the original NFA complexity to $(m - k)(k + 1)$, and also give a slightly more efficient simulation algorithm for the NFA. In experiments the method by Hyyro turns out to be often noticeably more efficient than the original algorithm under moderate values of k and m.

In [44] Hyyro *et al.* showed how multiple (short) patterns can be packed in a single computer word so as to search for multiple patterns simultaneously obtaining $\mathcal{O}(\lceil r/\lfloor w/m \rfloor \rceil n)$ time to search for r patterns of length $m < w$.

5.2 Damerau Distance

Another common form of edit distance is the Damerau edit distance [28], which is in principle an extension of the Levenshtein distance by permitting also the swap of two adjacent characters.

Navarro [55] has modified the Wu-Manber algorithm [64] described in Section 5.1 to use the Damerau distance by appending the automaton to have a temporary state vector S_r row to keep track of the positions where transposition may occur. In particular T_r is initialized to 0^m, for $0 \leq r \leq k$. Then we have

$$\begin{aligned} D'_0 &= ((D_0 \ll 1) \mid 1) \;\&\; B[T[j]] \\ D'_r &= ((D_r \ll 1) \;\&\; B[T[j]]) \mid D_{r-1} \mid (D_{r-1} \ll 1) \mid (D'_{r-1} \ll 1) \mid \\ &\quad (T_r \;\&\; (B[T[j]] \ll 1)) \mid 1, \text{ for } 1 \leq r \leq k. \\ T'_r &= (D_{r-1} \ll 2) \;\&\; B[T[j]], \text{ for } 1 \leq r \leq k. \end{aligned}$$

The formula adds $6k$ operations into the basic version for the Levenshtein edit distance

Later Hyyro proposed in [45] a different modification of the Wu-Manber algorithm which adds a total of 6 operations into the basic version for the Levenshtein edit distance. Therefore it makes the same number of operations as Navarro's version when $k = 1$, and wins when $k > 1$.

Moreover in [45] the author gives formulas for extending also the algorithm of Baeza-Yates and Navarro [11] and the algorithm of Myers [53] to use the Damerau distance. The resulting algorithms add 7 extra operations and 6 extra operations, respectively. Thus they do not affect the computational complexity of the original algorithms.

5.3 String Matching Allowing for Swaps

The *Pattern Matching problem with Swaps* (Swap Matching problem, for short) is a well-studied variant of the classic Pattern Matching problem. It consists in finding all occurrences, up to character swaps, of a pattern P of length m in a text T of length n, with P and T sequences of characters drawn from a same finite alphabet Σ of size σ. More precisely, the pattern is said to *swap-match the text at a given location* j if adjacent pattern characters can be swapped, if necessary, so as to make it identical to the substring of the text ending (or, equivalently, starting) at location j. All swaps are constrained to be disjoint, i.e., each character can be involved in at most one swap. Moreover adjacent equal characters are not allowed to be swapped.

The swap matching problem was introduced in 1995 as one of the open problems in nonstandard string matching [52]. This problem is of relevance in practical applications such as text and music retrieval, data mining, network security, and many others. Following [8], we also mention a particularly important application of the swap matching problem in biological computing, specifically in the process of translation in molecular biology, with the genetic triplets (otherwise called *codons*).

The first nontrivial result was reported by Amir *et al.* [3], who provided an algorithm which achieves $\mathcal{O}(nm^{\frac{1}{3}} \log m)$-time in the case of alphabet sets of size 2, showing also that the case of alphabets of size exceeding 2 can be reduced to that of size 2 with a $\mathcal{O}(\log^2 \sigma)$-time overhead, subsequently reduced to $\mathcal{O}(\log \sigma)$ in the journal version [4]. Amir *et al.* [6] studied some rather restrictive cases in which a $\mathcal{O}(m \log^2 m)$-time algorithm can be obtained. More recently, Amir *et al.* [5] solved the

swap matching problem in $\mathcal{O}(n \log m \log \sigma)$-time. We observe that the above solutions are all based on the fast Fourier transform (FFT) technique.

In 2008 the first attempt to provide an efficient solution to the swap matching problem without using the FFT technique has been presented by Iliopoulos and Rahman in [46]. They introduced a new graph-theoretic approach to model the problem and devised an efficient algorithm, based on the bit-parallelism technique [10], which runs in $\mathcal{O}((n+m)\log m)$-time, provided that the pattern size is comparable to the word size in the target machine.

More recently, in 2009, Cantone and Faro [21] presented a first approach for solving the swap matching problem with short patterns in linear time. Their algorithm, named Cross-Sampling, though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient bit-parallel implementation, named BP-Cross-Sampling, which achieves $\mathcal{O}(n)$ worst-case time and $\mathcal{O}(\sigma)$ space complexity in the case of short patterns fitting in few machine words.

Th BPCS algorithm is a natural generalization of the SA algorithm to the swap matching problem. It uses vectors of m bits, D_j and D'_j respectively. The i-th bit of D_j is set to 1 if P_i matches T_j, whereas the i-th bit of D'_j is set to 1 if P_{i-1} matches T_{j-1} and $P[i] = T[j+1]$. All remaining bits in the bit vectors are set to 0. As in the SA algorithm, for each character c of the alphabet Σ, a bit mask $M[c]$ is maintained, where the i-th bit is set to 1 if $P[i] = c$.

The bit vectors D_0 and D'_0 are initialized to 0^m. Then the algorithm scans the text from the first character to the last one and, for each position $j \geq 0$, it computes the bit vector D_j in terms of D_{j-1} and D'_{j-1}, by performing the following bitwise operations:

$$D_j \leftarrow (((D_{j-1} \ll 1) \mid 1) \,\&\, M[T[j]]) \mid ((D'_{j-1} \ll 1) \,\&\, M[T[j-1]])$$
$$D'_j \leftarrow ((D_{j-1} \ll 1) \mid 1) \,\&\, M[T[j+1]]$$

During the j-th iteration, we report a swap match at position j, provided that the leftmost bit of D_j is set to 1, i.e., if $(D_j \,\&\, 10^{m-1}) \neq 0^m$.

In practice, we can use only two vectors to maintain D_j and D'_j, for $j = 0, \ldots, n-1$. Thus during iteration j of the algorithm, vector D_{j-1} is transformed into vector D_j, whereas vector D'_{j-1} is transformed into vector D'_j.

In a subsequent paper[15] a more efficient algorithm, named Backward-Cross-Sampling (BCS) and based on a similar structure as the one of the Cross-Sampling algorithm, has been proposed. The BCS algorithm scans the text from right to left and has a $\mathcal{O}(nm^2)$-time complexity, whereas its bit-parallel implementation, named Bit-Parallel Backward-Cross-Sampling (BPBCS), works in $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity.

The BPCS and BPBCS algorithms could be also modified in order to solve the more general *Approximate Pattern Matching problem with Swaps*. Such a problem seeks to compute, for each text location j, the number of swaps necessary to convert the pattern to the substring of length m ending at j, provided there is a swapped matching at j.

A straightforward solution to the approximate swap matching problem consists in searching for all occurrences (with swap) of the input pattern P, using any algorithm for the standard swap matching problem. Once a swap match is found, to get the number of swaps, it is sufficient to count the number of mismatches between the pattern and its swap occurrence in the text and then divide it by 2.

In [7], Amir *et al.* presented an algorithm that counts in time $\mathcal{O}(\log m \log \sigma)$ the number of swaps at every location containing a swapped matching, thus solving the approximate pattern matching problem with swaps in $\mathcal{O}(n \log m \log \sigma)$-time.

In [16], the authors extended the BPCS and BPBCS algorithms designing two algorithms for the Approximate Swap Matching problem, which achieve $\mathcal{O}(n)$ and $\mathcal{O}(nm)$ worst-case time, respectively, and $\mathcal{O}(\sigma)$-space complexity for patterns having length similar to the word-size of the target machine.

In this case the two vectors $\bar{D}_j$ and $\bar{D}'_j$ are maintained as a list of q bits, where $q = \log(\lfloor m/2 \rfloor + 1) + 1$ and m is the length of the pattern. If P_i has a swap occurrence ending at position j of the text, with k swaps, then then the rightmost bit of the i-th block of $\bar{D}_j$ is set to 1 and the leftmost $q-1$ bits of the i-th block are set so as to contain the value k (notice that we need exactly $\log(\lfloor m/2 \rfloor + 1)$ bits to represent a value between 0 and $\lfloor m/2 \rfloor$). Otherwise the rightmost bit of the i-th block of $\bar{D}_j$ is set to 0. If $m \log(\lfloor m/2 \rfloor + 1) + m \leq w$, then the entire list fits in a single computer word, otherwise we need $\lceil m(\log(\lfloor m/2 \rfloor + 1)/w \rceil$ computer words to represent the sets $\bar{D}_j$ and $\bar{D}'_j$.

For each character c of the alphabet Σ the algorithm maintains a bit mask $M[c]$, where the rightmost bit of the i-th block is set to 1 if $P[i] = c$. Moreover, for each character $c \in \Sigma$, the algorithm maintains, a bit mask $B[c]$ whose i-th block have all bits set to 1 if $P[i] = c$, whereas all remaining bits are set to 0.

The generalization of the BPBCS algorithm to the approximate swap matching problem requires only $\log(\lfloor m/2 \rfloor + 1)$ bits to implement the counter for keeping track of the number of swaps. This compares favorably with the BPACS algorithm which uses instead m counters of $\log(\lfloor m/2 \rfloor + 1)$ bits, one for each prefix of the pattern.

The resulting BPABCS algorithm achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma \lceil m/w \rceil + \log(\lfloor m/2 \rfloor + 1))$ extra-space. If the pattern fits in few machine words, then the algorithm finds all swapped matches and their corresponding counts in $\mathcal{O}(nm)$-time and $\mathcal{O}(\sigma)$ extra-space.

5.4 String Matching with class of characters and general gaps

The δ-approximate string matching problem with α-bounded gaps (or (δ, α)-matching) [26,25,17] arises in many questions in music information retrieval and music analysis. This is particularly true, for instance, in the context of monophonic music, when one wants to retrieve occurrences of a given melody from a complex musical score. It finds also large applications in computational biology.

More formally, let Σ be a finite alphabet of *integer numbers* and let δ and α be nonnegative integers. Two symbols a and b of Σ are said to be δ-approximate, in which case we write $a =_\delta b$, if $|a - b| \leq \delta$. Given a pattern P of length m and a text T of length n over the alphabet Σ, by a δ-approximate occurrence with α bounded gaps of P in T, or simply a (δ, α)-occurrence of P in T, we mean a sequence $(i_0, i_1, \ldots, i_{m-1})$ of indices such that $0 \leq i_0 < i_1 < \cdots < i_{m-1} < n$, $T[i_j] =_\delta P[j]$, for $0 \leq j < m$, and $i_h - i_{h-1} \leq \alpha + 1$, for $0 < h < m$, provided that $m > 1$.

Given an index i, with $0 \leq i < n$, a (δ, α)-occurrence of P at position i in T is a (δ, α)-occurrence $(i_0, i_1, \ldots, i_{m-1})$ of P in T such that $i_{m-1} = i$. We write $P \trianglelefteq^i_{\delta,\alpha} T$ to mean that there is a (δ, α)-occurrence of P at position i in T (in fact, when the bounds δ and α are well understood from the context, one can simply write $P \trianglelefteq^i T$).

The δ-approximate string matching problem with α-bounded gaps has been first formally defined in [26], where the δ-Bounded-Gaps algorithm has been proposed

(see also [25,17]). The δ-Bounded-Gaps algorithm, whose time and space complexity is $\mathcal{O}(nm)$, with n and m the lengths of the text T and of the pattern P respectively, is presented as an incremental procedure, based on the dynamic programming approach. Scanning the pattern P from left to right, the δ-Bounded-Gaps algorithm looks for the (δ, α)-occurrences of each prefix P_j of the pattern P in the whole text T, for $0 \leq j < m$. Specifically, the δ-Bounded-Gaps algorithm proceeds by filling in a table D of dimensions $m \times n$ such that

$$D[j, i] = \max(\{k \geq 0 : i - \alpha \leq k \leq i \text{ and } P_j \trianglelefteq^k T\} \cup \{-1\})$$

for $0 \leq j < m$ and $0 \leq i < n$. Notice that $P_j \trianglelefteq^i T$ if and only if $D[j, i] = i$.

An algorithm, slightly more efficient than the δ-Bounded-Gaps, has been presented by the authors in [17], under the name (δ, α)-Sequential-Sampling. As in the case of the δ-Bounded-Gaps algorithm, the (δ, α)-Sequential-Sampling is also based on dynamic programming, but it follows a different computation ordering than the δ-Bounded-Gaps algorithm does; more precisely, it scans the text T from left to right and for each position i of T it looks for the (δ, α)-occurrences at position i of all prefixes of the pattern P. The (δ, α)-Sequential-Sampling algorithm has an $\mathcal{O}(nm)$ running time and requires $\mathcal{O}(m\alpha)$-space. A much more efficient variant of it is the (δ, α)-Tuned-Sequential-Sampling algorithm, which has an average case running time of $\mathcal{O}(n)$, in the case in which α is assumed constant (cf. [18]).

Another algorithm, named (δ, α)-Shift-And, has also been described in [18]. The (δ, α)-Shift-And algorithm is a very simple variant of a forward search algorithm presented in [58] for a pattern matching problem with gaps and character classes, particularly suited for applications to protein searching. It uses bit-parallelism to simulate the behavior of a nondeterministic finite automaton with ε-transitions. The automaton has $\ell = (\alpha + 1)(m - 1) + 2$ states, and the simulation is carried out by representing it as a bit mask B of length $\ell - 1$ (the initial state of the automaton need not be represented in the bit mask since it is always active during the computation). When $\ell < w$ (the computer word length), the entire bit mask B fits in a single computer word. In this case the (δ, α)-Shift-And algorithm becomes extremely fast in practice.

Other efficient algorithms for the (δ, α)-matching problem have been presented more recently in [36] and [37]. In particular, [36] presents two algorithms, called DA-bpdb and DA-mloga-bits. The first one inherits the basic idea from the dynamic programming algorithm δ-Bounded-Gaps presented in [25]. It uses bit-parallelism to compute an $m \times n$ bit-matrix $\mathcal{D}$ such that $(\mathcal{D})_{j,i} = 1$ if and only if $P_j \trianglelefteq^i T$, for $0 \leq j < m$ and $0 \leq i < n$. Basically, the algorithm DA-bpdb partitions each row of the matrix $\mathcal{D}$ as a sequence of $\lceil n/w \rceil$ consecutive bit masks, each of which represents a group of w bits on that row. Then, the computation of the j-th bit mask in row i is performed bit-parallely by using the $(j-1)$-st and the j-th bit masks of the $(i-1)$-st row. It turns out that DA-bpdb has an $\mathcal{O}(n\delta + \lceil n/w \rceil m)$ worst-case execution time, which becomes $\mathcal{O}(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$ on the average. The second algorithm presented in [36], namely DA-mloga-bits, is based on a compact representation, in the form of a systolic array, of the nondeterministic automaton used in the algorithm (δ, α)-Shift-And. The systolic array is composed of m building blocks, called *counters* in [36], one for each symbol of the pattern, and is represented as a bit mask of length $(m-1)(\lceil \log_2(\alpha + 1) \rceil + 1) + 1$. Notice that this improves the representations used in [58,18] in which $(\alpha + 1)(m - 1) + 1$ bits are needed to represent the automaton. It

turns out that the DA-mloga-bits algorithm has an $\mathcal{O}(n\lceil(m\log_2\alpha)/w\rceil)$ worst-case searching time.

The algorithms presented in [37], called SDP-rows, SDP-columns, SDP-simple, and SDP-simple-compute-L_0, use different computation orderings, in combination with sparse dynamic programming techniques, to implement the calculation of the table D above. Specifically, in the case of the SDP-rows algorithm, the computation is performed row-wise, whereas a column-wise computation is used by SDP-columns. The algorithm SDP-simple, which can be considered as a brute force variant of SDP-rows, performs very well in practice, especially for small values of δ and α; SDP-simple-compute-L_0 improves the average case running time of SDP-simple by using a Boyer-Moore-Horspool-like shifting strategy [41], suitably adapted to handle gaps. In particular, the latter two algorithms turn out to be among the most efficient ones, in terms of running time, in many practical cases, especially for small values of α, as shown in [37]. However, although these algorithms are very fast in practice, they require additional $\mathcal{O}(n)$-space, plus $\mathcal{O}(\sigma)$-space in the case of SDP-simple-compute-L_0.

More recently, in [19], the authors presented four new efficient variants of the algorithm (δ,α)-Sequential-Sampling, all based on bit-parallelism. In particular, one of these variants, the (δ,α)-Tuned-Sequential-Sampling-HBP algorithm, is extremely efficient in most practical cases and outperforms both algorithms SDP-simple and SDP-simple-compute-L_0. The variant (δ,α)-Sequential-Sampling-BP$^+$ turns out to be faster than existing algorithms (e.g., (δ,α)-Shift-And) in the case of short patterns and very small values of α.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
2. C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99, Theory and Practice of Informatics*, number 1725 in Lecture Notes in Computer Science, pages 291–306, Milovy, Czech Republic, 1999. Springer-Verlag, Berlin.
3. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. In *IEEE Symposium on Foundations of Computer Science*, pages 144–153, 1997.
4. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37(2):247–266, 2000.
5. A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003.
6. A. Amir, G. M. Landau, M. Lewenstein, and N. Lewenstein. Efficient special cases of pattern matching with swaps. *Information Processing Letters*, 68(3):125–132, 1998.
7. A. Amir, M. Lewenstein, and Ely Porat. Approximate swapped matching. *Inf. Process. Lett.*, 83(1):33–39, 2002.
8. P. Antoniou, C.S. Iliopoulos, I. Jayasekera, and M.S. Rahman. Implementation of a swap matching algorithm using a graph theoretic model. In *Bioinformatics Research and Development, Second International Conference, BIRD 2008*, volume 13 of *Communications in Computer and Information Science*, pages 446–455. Springer, 2008.
9. Jörg Arndt. *Matters Computational*. Springer, 2011. `http://www.jjj.de/fxt/`.
10. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
11. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

12. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel. Linear size finite automata for the set of all subwords of a word: an outline of results. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 21:12–20, 1983.
13. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.
14. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
15. M. Campanelli, D. Cantone, and S. Faro. A new algorithm for efficient pattern matching with swaps. In *IWOCA 2009: 20th International Workshop on Combinatorial Algorithms*, Lecture Notes in Computer Science. Springer, 2009.
16. M. Campanelli, D. Cantone, S. Faro, and E. Giaquinta. Pattern matching with swaps in practice. *International Journal of Foundation of Computer Science*, 23(2):323–342, 2012.
17. D. Cantone, S. Cristofaro, and S. Faro. An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In S. E. Nikoletseas, editor, *Proceedings of 4-th International Workshop on Experimental and Efficient Algorithms (WEA 2005)*, volume 3503 of *Lecture Notes in Computer Science*, pages 428–439. Springer-Verlag, 2005.
18. D. Cantone, S. Cristofaro, and S. Faro. On tuning the (δ, α)-sequential-sampling algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In S. D. Reiss and G. A. Wiggins, editors, *Proceedings of 6-th International Conference on Music Information Retrieval (ISMIR 2005)*, pages 454–459, 2005.
19. D. Cantone, S. Cristofaro, and S. Faro. New efficient bit-parallel algorithms for the (δ, α)-matching problem with applications in music information retrieval. *International Journal of Foundation of Computer Science*, 20(6):1087–1108, 2009.
20. D. Cantone and S. Faro. A space efficient bit-parallel algorithm for the multiple string matching problem. *Int. J. Found. Comput. Sci.*, 17(6):1235–1252, 2006.
21. D. Cantone and S. Faro. Pattern matching with swaps for short patterns in linear time. In *SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2009.
22. D. Cantone, S. Faro, and E. Giaquinta. Bit-(parallelism)^2: Getting to the next level of parallelism. In Paolo Boldi and Luisa Gargano, editors, *Fun with Algorithms*, volume 6099 of *Lecture Notes in Computer Science*, pages 166–177. Springer-Verlag, Berlin, 2010.
23. D. Cantone, S. Faro, and E. Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Information and Computation*, 213:3–12, 2012.
24. C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. King's College Publications, 2004.
25. M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic J. of Computing*, 9(1):54–65, 2002.
26. M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and W. Rytter. Finding motifs with gaps. In Don Byrd and J. Stephen Downie, editors, *Proceedings of International Symposium on Music Information Retrieval: Music IR 2000*, Amherst, MA, 2000. University of Massachusetts at Amherst.
27. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
28. F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964.
29. B. Dömölki. A universal compiler system based on production rules. *BIT Numerical Mathematics*, 8:262–275, 1968.
30. B. Durian, J. Holub, H. Peltola, and J. Tarhio. Tuning BNDM with q-grams. In I. Finocchi and J. Hershberger, editors, *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2009*, pages 29–37, New York, New York, USA, 2009. SIAM.
31. B. Durian, H. Peltola, L. Salmela, and J. Tarhio. Bit-parallel search algorithms for long patterns. In *SEA*, LNCS 6049, pages 129–140, 2010.
32. S. Faro and T. Lecroq. Efficient variants of the Backward-Oracle-Matching algorithm. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 146–160, Czech Technical University in Prague, Czech Republic, 2008.
33. S. Faro and T. Lecroq. The exact string matching problem: a comprehensive experimental evaluation. Report arXiv:1012.2547, 2010.

34. S. Faro and T. Lecroq. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, 45(2), 2013. to appear.
35. K. Fredriksson and S. Grabowski. Practical and optimal string matching. In M. P. Consens and G. Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 376–387. Springer-Verlag, Berlin, 2005.
36. K. Fredriksson and Sz. Grabowski. Efficient bit-parallel algorithms for (δ, α)-matching. In *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA'06)*, LNCS 4007, pages 170–181. Springer–Verlag, 2006.
37. K. Fredriksson and Sz. Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Information Retrieval*, March 2008. to appear (currently available only online).
38. L. He, B. Fang, and J. Sui. The wide window string matching algorithm. *Theor. Comput. Sci.*, 332(1-3):391–404, 2005.
39. J. Holub and B. Durian. Talk: Fast variants of bit parallel approach to suffix automata. In *The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf*, 2005.
40. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2001.
41. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
42. A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.
43. H. Hyyrö. Tighter packed bit-parallel nfa for approximate string matching. In *CIAA*, pages 287–289, 2006.
44. H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate string matching. In *WEA*, pages 285–298, 2004.
45. Heikki Hyyrö. A bit-vector algorithm for computing levenshtein and damerau edit distances. *Nord. J. Comput.*, 10(1):29–39, 2003.
46. C. S. Iliopoulos and M. S. Rahman. A new model to solve the swap matching problem and efficient algorithms for short patterns. In *SOFSEM 2008*, volume 4910 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 2008.
47. P. Kalsi, H. Peltola, and J. Tarhio. Comparison of exact string matching algorithms for biological sequences. In M. Elloumi, J. Küng, M. Linial, R. F. Murphy, K. Schneider, and C. Toma, editors, *Proceedings of the Second International Conference on Bioinformatics Research and Development, BIRD'08*, volume 13 of *Communications in Computer and Information Science*, pages 417–426, Vienna, Austria, 2008. Springer-Verlag, Berlin.
48. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
49. M. Oğuzhan Külekci. A method to overcome computer word size limitation in bit-parallel pattern matching. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation, ISAAC 2008*, volume 5369 of *Lecture Notes in Computer Science*, pages 496–506, Gold Coast, Australia, 2008. Springer-Verlag, Berlin.
50. VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.
51. J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.
52. S. Muthukrishnan. New results and open problems related to non-standard stringology. In *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95*, volume 937 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 1995.
53. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.
54. G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
55. G. Navarro. NR-grep: a fast and flexible pattern-matching tool. *Softw. Pract. Exp.*, 31(13):1265–1312, 2001.
56. G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theor. Comput. Sci.*, 321(2-3):283–290, 2004.

57. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *J. Exp. Algorithmics*, 5:4, 2000.
58. G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *RECOMB '01: Proceedings of the fifth annual international conference onComputational biology*, pages 231–240, New York, NY, USA, 2001. ACM.
59. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
60. G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2005.
61. H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In M. A. Nascimento, E. Silva de Moura, and A. L. Oliveira, editors, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval SPIRE'03*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94, Manaus, Brazil, 2003. Springer-Verlag, Berlin.
62. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1(10):359–373, 1980.
63. D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.
64. S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.
65. S. Wu and U. Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, 1992.
66. A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.
67. G. Zhang, E. Zhu, L. Mao, and M. Yin. A bit-parallel exact string matching algorithm for small alphabet. In X. Deng, J. E. Hopcroft, and J. Xue, editors, *Proceedings of the Third International Workshop on Frontiers in Algorithmics, FAW 2009, Hefei, China*, volume 5598 of *Lecture Notes in Computer Science*, pages 336–345. Springer-Verlag, Berlin, 2009.

A Parameterized Formulation for the Maximum Number of Runs Problem*

Andrew Baker, Antoine Deza, and Frantisek Franek

Advanced Optimization Laboratory
Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada
{bakerar2,deza,franek}@mcmaster.ca
http://optlab.cas.mcmaster.ca/

Dedicated to Professor Bořivoj Melichar on the occasion of his 70th birthday

Abstract. A parameterized approach to the problem of the maximum number of runs in a string was introduced by Deza and Franek. In the approach referred to as the *d-step approach*, in addition to the usual parameter the length of the string, the size of the string's alphabet is considered. The behaviour of the function $\rho_d(n)$, the maximum number of runs over all strings of length n with exactly d distinct symbols, can be handily expressed in the terms of properties of a table referred to as the $(d, n-d)$ *table* in which $\rho_d(n)$ is the entry at the dth row and $(n-d)$th column. The approach leads to a conjectured upper bound $\rho_d(n) \leq n-d$ for $2 \leq d \leq n$. The parameterized formulation shows that the maximum within any column of the $(d, n-d)$ table is achieved on the main diagonal, i.e. for $n = 2d$, and motivates the investigation of the structural properties of the run-maximal strings of length n bounded by a constant times the size of the alphabet d. We show that $\rho_d(n) = \rho_{n-d}(2n-2d)$ for $2 \leq d \leq n \leq 2d$, $\rho_d(2d) \leq \rho_{d-1}(2d-1)+1$ for $d \geq 3$, $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2) = \rho_{d-3}(2d-3)$ for $d \geq 5$, and $\{\rho_d(n) \leq n-d \text{ for } 2 \leq d \leq n\} \Leftrightarrow \{\rho_d(9d) \leq 8d \text{ for } d \geq 2\}$. The results allow for an efficient computational verification of entries in the $(d, n-d)$ table for higher values of n and point to a plausible way of either proving the maximum number of runs conjecture by showing that possible counter-examples on the main diagonal would exhibit an impossible structure, or to discover an unexpected counter-example on the main diagonal of the $(d, n-d)$ table. This approach provides a purely analytical proof of $\rho_d(2d) = d$ for $d \leq 15$ and, using the computational results of $\rho_2(d+2)$ for $d = 16, \ldots, 23$, a proof of $\rho_d(2d) = d$ for $d \leq 23$.

Keywords: string, runs, maximum number of runs, parameterized approach, $(d, n-d)$ table

1 Foreword

The two first authors of this contribution have known of Bořislav Melichar's work since they ventured into the field of stringology a few years ago, while the third author has known him and his work in compilers for many years. Bořek's – as known to his friends – accomplishments include establishing a highly reputed research group, nurturing an impressive list of graduate students, and his pioneering and high-impact research work as an internationally recognized leader in the field. It is an equal honour and pleasure to dedicate to Bořek our work originally presented at the 2011 edition of the vibrant Prague Stringology Conference series.

* 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/).

2 Introduction

The problem of determining the maximum number of runs in a string has a rich history and many researchers have contributed to the effort. The notion of a run is due to Main [17], the term itself was introduced in [13]. Kolpakov and Kucherov [14,15] showed that the function $\rho(n)$, the maximum number of runs over all strings of length n, is linear. Several papers dealt with lower and upper bounds or expected values for $\rho(n)$, see [3,4,5,9,10,11,12,18,19,20,21,22] and references therein.

The counting estimates leading to the best upper bounds [4,5] rely heavily on a computational approach and seem to reach a point where it gets highly challenging, bordering intractability, to verify the results or make further progress. A few researchers tried a structural approach, for instance [8,16].

A parameterized approach to the investigation of the structural aspects of run-maximal strings was introduced by Deza and Franek [6]. In addition to considering the length of the string they introduced the parameter d giving the function $\rho_d(n)$, the maximum number of runs over all strings of length n with exactly d distinct symbols. These values are presented in the so-called $(d, n-d)$ table, where the value of $\rho_d(n)$ is the entry at the row d and the column $n-d$. In Table 1, the entries for the first 10 rows and the first 10 columns are presented. Several properties of the table were presented in [6], the most important being the fact that $\rho_d(n) \leq n-d$ for $2 \leq d \leq n$ is equivalent with $\rho_d(2d) \leq d$ for $d \geq 2$. In other words, if the diagonal obeys the upper bound $n-d$, so do all the entries in the table everywhere. Though in the related literature, the *maximum number of runs conjecture* – or simply *runs conjecture* – refers to the hypothesis that $\rho(n) \leq n$, in this paper we will take it to be $\rho_d(n) \leq n-d$. Note that while the upper bound of n is not achieved for any known string, the $n-d$ bound is achieved for all pairs (d, n) satisfying $n - d \leq \min(23, d)$.

We discuss several additional properties of the $(d, n-d)$ table, the behaviour of the function $\rho_d(n)$ on or nearby the main diagonal, and discuss some structural properties of run-maximal strings on the main diagonal. The results allow for the extension of computational verification of the maximum number of runs conjecture to higher values of n and also indicate a viable approach to an analytical investigation of the conjecture by either showing a possible counter-example to the conjecture would have to exhibit an impossible structure, or exhibiting a counter-example on the main diagonal of the $(d, n-d)$ table and direct calculation of entries for smaller columns.

Let us remark, that although we believe with the majority of the researchers in the field that the conjecture is true and hence view the d-step approach as a possible tool to prove it, if a counter-example exists, there must be one on the main diagonal and we believe it will be easier to find there as the run-maximal strings of length being twice the size of the alphabet seem to exhibit a richer structure than general run-maximal strings. For example, all tractable run-maximal strings satisfying $n = 2d$ are, up to relabeling, unique. A counter-example would be in essence a quite striking result. The parameterized approach is inspired by a similar $(d, n-d)$ table used for investigating the Hirsch bound for the diameter of bounded polytopes. The associated Hirsch $(d, n-d)$ table exhibits similar property as the $(d, n-d)$ table considered in this paper. The Conjecture of Hirsch was recently disproved by Santos [23] by exhibiting a violation on the main diagonal with $d = 43$.

d \ $n-d$	1	2	3	4	5	6	7	8	9	10	11
1	**1**	*1*	1	1	1	1	1	1	1	1	.
2	1	**2**	*2*	3	4	5	5	6	7	8	.
3	1	2	**3**	*3*	4	5	6	6	7	8	.
4	1	2	3	**4**	*4*	5	6	7	7	8	.
5	1	2	3	4	**5**	*5*	6	7	8	8	.
6	1	2	3	4	5	**6**	*6*	7	8	9	.
7	1	2	3	4	5	6	**7**	*7*	8	9	.
8	1	2	3	4	5	6	7	**8**	*8*	9	.
9	1	2	3	4	5	6	7	8	**9**	*9*	.
10	1	2	3	4	5	6	7	8	9	**10**	.
11	.	.	.	.	.	.	.	.	.	.	.

Table 1. Values for $\rho_d(n)$ with $1 \leq d \leq 10$ and $1 \leq n - d \leq 10$. For more values, see [2]

3 Notation and Preliminaries

Throughout this paper, we refer to k-tuples: a symbol which occurs exactly k times in the string under consideration. Specially named k-tuples are the *singleton* (1-tuple), *pair* (2-tuple), *triple* (3-tuple), *quadruple* (4-tuple), and *quintuple* (5-tuple).

Definition 1. *A safe position in a string $\boldsymbol{x}$ is one which, when removed from $\boldsymbol{x}$, does not result in two runs being merged into one in the resulting new string.*

A safe position does not ensure that the number of runs will not change when that position is removed, only that no runs will be lost through being merged; runs may still be destroyed by having an essential symbol removed. Safe positions are important in that they may be removed from a string while only affecting the runs which contain them. When the position of a symbol is unambiguous, we may thus refer to a *safe symbol* rather than to its position – for instance we can talk about a safe singleton, or about the first member of a pair being safe, etc.

At various points we will need to relabel all occurrences of a symbol in a string or substring. Let $\boldsymbol{x}^a_b$ denote the string $\boldsymbol{x}$, in which all occurrences of a are replaced by b, and vice versa. $S_d(n)$ refers to the set of strings of length n with exactly d distinct symbols. For a string $\boldsymbol{x}$, $\mathcal{A}(\boldsymbol{x})$ denotes the alphabet of $\boldsymbol{x}$, while $r(\boldsymbol{x})$ denotes the number of runs of $\boldsymbol{x}$.

Lemma 2. *There exists a run-maximal string in $S_d(n)$ with no unsafe singletons for $2 \leq d \leq n$.*

Proof. Let $\boldsymbol{x}$ be a run-maximal string in $S_d(n)$. We will show that one of the following conditions must hold:

- (i) $\boldsymbol{x}$ has no singletons, or
- (ii) $\boldsymbol{x}$ has exactly one singleton which is safe, or
- (iii) $\boldsymbol{x}$ has exactly one singleton which is unsafe, and there exists another run-maximal string $\boldsymbol{x'} \in S_d(n)$ where $\boldsymbol{x'}$ has no unsafe singletons, or
- (iv) $\boldsymbol{x}$ has more than one singleton, all of which are safe.

Let $\boldsymbol{x}$ have some unsafe singletons.

First, consider the case that $\boldsymbol{x}$ has exactly one singleton, C, which is unsafe: $\boldsymbol{x} = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}C a\boldsymbol{v}a\boldsymbol{v}\boldsymbol{w}$, where $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ are (possibly empty) strings, and $a \in \mathcal{A}(\boldsymbol{x}) - \{C\}$.

Let $\boldsymbol{x'} = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}(Ca\boldsymbol{v}a\boldsymbol{v}\boldsymbol{w})^a_C = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}(aC\boldsymbol{v}^a_C C\boldsymbol{v}^a_C\boldsymbol{w}^a_C) = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}aC\tilde{\boldsymbol{v}}C\tilde{\boldsymbol{v}}\tilde{\boldsymbol{w}}$. Clearly, $\boldsymbol{x'} \in S_d(n)$, $r(\boldsymbol{x'}) \geq r(\boldsymbol{x})$, so $\boldsymbol{x'}$ is run-maximal and has no singletons.

Next, consider the case that $\boldsymbol{x}$ has at least 2 singletons C, D, of which one is unsafe, C. Without loss of generality, we can assume C occurs before D : $\boldsymbol{x} = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}Ca\boldsymbol{v}a\boldsymbol{v}\boldsymbol{w}D\boldsymbol{z}$, where $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$, and $\boldsymbol{z}$ are (possibly empty) strings and $a \in \mathcal{A}(\boldsymbol{x}) - \{C, D\}$. Let $\boldsymbol{x_1} = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}(Ca\boldsymbol{v}a\boldsymbol{v}\boldsymbol{w}D\boldsymbol{z})^a_C = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}aC\tilde{\boldsymbol{v}}C\tilde{\boldsymbol{v}}\tilde{\boldsymbol{w}}D\tilde{\boldsymbol{z}}$. Clearly, $\boldsymbol{x_1} \in S_d(n)$ and $r(\boldsymbol{x_1}) \geq r(\boldsymbol{x})$. We then modify $\boldsymbol{x_1}$ by removing the safe symbol a immediately to the left of the first occurrence of C, yielding $\boldsymbol{x_2}$. Finally, we add a second copy of D adjacent to the original D, restoring the original length: $\boldsymbol{x_3} = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}C\tilde{\boldsymbol{v}}C\tilde{\boldsymbol{v}}\tilde{\boldsymbol{w}}DD\tilde{\boldsymbol{z}}$. $\boldsymbol{x_3} \in S_d(n)$ and $r(\boldsymbol{x_3}) > r(\boldsymbol{x_2}) \geq r(\boldsymbol{x_1}) \geq r(\boldsymbol{x})$, which contradicts the run-maximality of $\boldsymbol{x}$. □

Lemma 3 is a simple observation that for a position to be unsafe, a symbol must occur twice to the left and twice to the right of that position.

Lemma 3. *If a string $\boldsymbol{x}$ consists only of singletons, pairs, and triples, then every position is safe.*

A corollary of Lemma 3 is that the maximum number of runs in a string with only singletons, pairs, and triples is limited by the number of pairs and triples. Specifically, $r(\boldsymbol{x}) = \#pairs + \lfloor \frac{3}{2} \#triples \rfloor$. This is because a pair can only be involved in a single run, and a triple can be involved in at most 2 runs. The densest structure achievable is through overlapping triples in the pattern *aababb*, which has three runs for every two triples. The pairs, meanwhile, are maximized through adjacent copies.

4 Run-maximal strings below the main diagonal and in the immediate neighbourhood above

We first remark that every value below the main diagonal in the $(d, n-d)$ table is equal to the value on the main diagonal directly above it. In other words, the values on and below the main diagonal in a column are constant.

Proposition 4. *We have $\rho_d(n) = \rho_{n-d}(2n-2d)$ for $2 \leq d \leq n < 2d$.*

Proof. Consider a run-maximal string $\boldsymbol{x} \in S_d(n)$, where $2 \leq d \leq n < 2d$. By Lemma 2, we can assume $\boldsymbol{x}$ has no unsafe singletons. Since $n < 2d$, $\boldsymbol{x}$ must have a singleton, and hence it must be safe. We can remove this safe singleton, yielding a new string $\boldsymbol{y} \in S_{d-1}(n-1)$ and so $\rho_d(n) = r(\boldsymbol{x}) = r(\boldsymbol{y}) \leq \rho_{d-1}(n-1)$. Recall the following inequality noted in [6]:

$$\rho_d(n) \leq \rho_{d+1}(n+1) \text{ for } 2 \leq d \leq n \tag{1}$$

Thus, $\rho_{d-1}(n-1) = \rho_d(n)$. □

Proposition 4 together with inequality (1) gives the following equivalence noted in [6]: $\{\rho_d(n) \leq n-d \text{ for } 2 \leq d \leq n\} \Leftrightarrow \{\rho_d(2d) \leq d \text{ for } 2 \leq d\}$.

If there is a counter-example to the conjectured upper bound, then the main diagonal must contain a counter-example. If it falls under the main diagonal, then by Proposition 4 there must be a counter-example on the main diagonal – i.e. it can be *pushed up*, and if it falls above the main diagonal, by the inequality (1), there must

be a counter-example on the main diagonal – i.e. the counter-example can be *pushed down*.

We extend Proposition 4 to bound the behaviour of the entries in the immediate neighbourhood above the main diagonal in the $(d, n-d)$ table. Proposition 5 establishes that the difference between the entry on the main diagonal and the entry immediately above it is at most 1. In addition, the difference is 1 if and only if every run-maximal string in $S_d(2d)$ consists entirely of pairs; otherwise, the difference is 0.

Proposition 5. *We have* $\rho_d(2d) \leq \rho_{d-1}(2d-1) + 1$ *for* $d \geq 3$.

Proof. Let $\boldsymbol{x} \in S_d(2d)$ be a run-maximal string with no unsafe singletons (by Lemma 2). If $\boldsymbol{x}$ does not have a singleton, then it consists entirely of pairs. It is clear that the pairs must be adjacent and that $r(\boldsymbol{x}) = d$ and so $\boldsymbol{x} = aabbcc\ldots$. Removing the first a and renaming the second to b, $\boldsymbol{y} = bbbcc\ldots \in S_{d-1}(2d-1)$ and $\rho_{d-1}(2d-1) \geq r(\boldsymbol{y}) = r(\boldsymbol{x}) - 1 = \rho_d(2d) - 1$. If $\boldsymbol{x}$ has a singleton, since it is safe we can remove it forming a string $\boldsymbol{y} \in S_{d-1}(2d-1)$ so that $\rho_{d-1}(2d-1) \geq r(\boldsymbol{y}) = r(\boldsymbol{x}) = \rho_d(2d)$, and so $\rho_{d-1}(2d-1) = \rho_d(2d)$. □

We have seen that the gap between the first entry above the diagonal and the diagonal entry is at most 1. Proposition 6 establishes that the three entries just above the diagonal are identical.

Proposition 6. *We have* $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2) = \rho_{d-3}(2d-3)$ *for* $d \geq 5$.

Proof. Let $\boldsymbol{x}$ be a run-maximal string in $S_{d-1}(2d-1)$. By Lemma 2 we can assume that either it has a safe singleton or no singletons at all. In the former case, we can remove the safe singleton obtaining $\boldsymbol{y} \in S_{d-2}(2d-2)$ so that $\rho_{d-2}(2d-2) \geq r(\boldsymbol{y}) \geq r(\boldsymbol{x}) = \rho_{d-1}(2d-1)$, and so $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2)$. In the latter case, $\boldsymbol{x}$ consists of pairs and one triple, and thus, by Lemma 3, all positions are safe. Therefore, we can move all the pairs to the end of the string, yielding $\boldsymbol{y} = aaabbcc\ldots \in S_{d-1}(2d-1)$ and by removing the first a and renaming the remaining a's to c's, $\boldsymbol{z} = ccbbcc\ldots \in S_{d-2}(2d-2)$. It follows that $\rho_{d-2}(2d-2) \geq r(\boldsymbol{z}) = r(\boldsymbol{y}) = r(\boldsymbol{x}) = \rho_{d-1}(2d-1)$, and so $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2)$.

Let $\boldsymbol{x}$ be now a run-maximal string in $S_{d-2}(2d-2)$. Again, if $\boldsymbol{x}$ has a singleton, we can assume by Lemma 2 it is safe and form $\boldsymbol{y}$ by removing the singleton. $\boldsymbol{y} \in S_{d-3}(2d-3)$ and $\rho_{d-3}(2d-3) \geq r(\boldsymbol{y}) \geq r(\boldsymbol{x}) = \rho_{d-2}(2d-2)$. If $\boldsymbol{x}$ does not have a singleton, then $r(\boldsymbol{x}) = d-1$. To see this, consider the two cases:

(*i*) $\boldsymbol{x}$ consists of two triples and several pairs. The most runs which may be obtained in such a string, after grouping the pairs at the end of the string, is through the arrangement $aababbccddee\ldots$. In this case, there are $d-4$ runs from the pairs, and 3 runs from the triples, giving a total of $d-1$ runs.

(*ii*) $\boldsymbol{x}$ consists of a quadruple and several pairs. The most runs which may be obtained in this case is from a string with either the structure $aabbaaccddee\ldots$, or $aabaabccddee\ldots$, where all the pairs have been grouped at the end, except for the pair of bs which is used to break up the quadruple. In both cases, there are $d-4$ runs involving characters c onward, and three runs involving the characters a and b, again giving a total of $d-1$ runs.

Now consider a string $\boldsymbol{y} = aabbaabbcdee\ldots \in S_{d-2}(2d-2)$, which has two quadruples (of a's and b's), two singletons (c and d), and several pairs ($e\ldots$). This string has

$d-6$ runs from the pairs ee onward, and 5 runs from the characters a and b, giving a total of $d-1$ runs, i.e. $r(\boldsymbol{x}) = r(\boldsymbol{y})$. The singleton c in $\boldsymbol{y}$ being clearly safe, we can remove it and continue as in the previous case. □

Remark 7 below providing a lower bound for the first 4 entries above the main diagonal of the $(d, n-d)$ table, is a corollary of the inequality $\rho_{d+s}(n+2s) \geq \rho_d(n)+s$, noted in [6], applied to $\rho_2(k) = k-3$ for $k = 5, 6, 7$ and 8.

Remark 7. We have $\rho_{d-k}(2d-k) \geq d-1$ for $k = 1, 2, 3$ and 4 and $d \geq 6$.

5 Structural properties of run-maximal strings on the main diagonal

We explore structural properties of the run-maximal strings on the main diagonal. These results yield properties for run-maximal strings that have their length bounded by nine times the number of distinct symbols they contain. We can thus shift the critical region of the $(d, n-d)$ table as summarized in Theorem 8, the proof for which can be found at the end of this section.

Theorem 8. *We have* $\{\rho_d(n) \leq n-d \text{ for } 2 \leq d \leq n\} \Leftrightarrow \{\rho_d(9d) \leq 8d \text{ for } d \geq 2\}$.

Proposition 9 describes useful structural properties of run-maximal strings on the main diagonal. The proof of the proposition relies on a series of lemmas all of which are dealing with the same basic scenario: assuming we know that the table obeys the conjecture for all columns to the left of column d, which is the first *unknown* column, we investigate the run-maximal strings of $S_d(2d)$.

Proposition 9. *[Proposition] Let* $\rho_{d'}(2d') \leq d'$ *for* $2 \leq d' < d$. *Let* $\boldsymbol{x}$ *be a run-maximal string in* $S_d(2d)$. *Either* $r(\boldsymbol{x}) = \rho_d(2d) = d$ *or* $\boldsymbol{x}$ *has at least* $\lceil \frac{7d}{8} \rceil$ *singletons, and no symbol occurs exactly 2, 3, . . . , 8 times in* $\boldsymbol{x}$.

Proof. The proof that each symbol must be a singleton or occur at least 9 times is a direct result of the lemmas which make up the remainder of this section. Then, let $\boldsymbol{x} \in S_d(2d)$ be run-maximal, m_1 denote the number of singletons, and m_2 the number of non-singleton symbols of $\boldsymbol{x}$. We have $m_1 + 9m_2 \leq 2d$ and $m_1 + m_2 = d$, which implies that $m_2 \leq d/8$ and hence $m_1 \geq \lceil 7d/8 \rceil$. □

Proposition 9 provides a purely structural proof that $\rho_d(2d) = d$ for $d \leq 15$, and using the computer generated results of $\rho_2(d+2)$ for $d = 16, \ldots, 23$, that $\rho_d(2d) = d$ for $d \leq 23$.

Corollary 10. *We have* $\rho_d(2d) = d$ *for* $d \leq 23$ *and* $\rho_d(n) \leq n-d$ *for* $n-d \leq 23$.

Proof. Assume that run-maximal $\boldsymbol{x} \in S_d(2d)$ satisfies $r(\boldsymbol{x}) = \rho_d(2d) > d$. By Proposition 9, $\boldsymbol{x}$ consists only of singletons for $2 \leq d \leq 6$, $r(\boldsymbol{x}) = \rho_1(d+1) = 1$ for $8 \leq d \leq 15$, and $d < r(\boldsymbol{x}) = \rho_2(d+2)$ for $16 \leq d \leq 23$, which are impossible. □

Before we begin the lemmas to support 9Structural properties of run-maximal strings on the main diagonalproposition.9, we first introduce a few concepts.

Definition 11 (Map). *A run (s, p, d) of period p, starting at position s and ending at position d of a string $\boldsymbol{x}$* maps *position i to position j if $s \leq i < j \leq d$ and $j - i = p$. We denote a mapping from i to j by $i \rightarrow j$ and call it a single-mapping. We extend the mapping notation to $(i_1, i_2) \rightarrow (j_1, j_2)$, denoting $s \leq i_1 < i_2 < j_1 < j_2 < d$ and $j_1 - i_1 = j_2 - i_2 = p$ and call it a double-mapping. The triple- and higher order mappings are defined analogously.*

A *multi-mapping* will be any mapping which is not a single-mapping. The presence of a multi-mapping imposes equality on the substrings bounded on each side. For example, in the double-mapping $(i_j, i_{j+1}) \rightarrow (i_{j+2}, i_{j+3})$, $\boldsymbol{x}[i_j..i_{j+1}]$ the substring between i_j and i_{j+1} is the same as $\boldsymbol{x}[i_{j+2}..i_{j+3}]$ the substring between i_{j+2} and i_{j+3}.

In the following lemmas, we assume that for $2 \leq d' < d$, the conjecture holds, i.e. $\rho_{d'}(2d') \leq d'$. Note that it is equivalent to $\rho_{d'}(n') \leq n' - d'$ for $2 \leq d' \leq n'$ when $n' - d' < d$. We consider a run-maximal string $\boldsymbol{x} \in S_d(2d)$ containing a k-tuple of c's such that $\boldsymbol{x} = \boldsymbol{u_0} c \boldsymbol{u_1} c \ldots \boldsymbol{u_{k-1}} c \boldsymbol{u_k}$. We show that either the string $\boldsymbol{x}$ obeys the conjectured upper bound, or can be manipulated to obtain a new string $\boldsymbol{y}$ with a larger alphabet of the same or shorter length. We ensure that the manipulation process does not destroy more runs than the the amount the alphabet is increased or the length decreased. This allows us to estimate the number of runs in $\boldsymbol{y}$ based on the values in the table for some $d' < d$. In essence, we manipulate a string from column d to a string from some column $d' < d$ while monitoring the number of runs. In the manipulation process, we put an upper limit on the number of runs which are destroyed (π), and a lower limit on how many additional symbols are introduced (δ).

In order to have more distinct symbols in $\boldsymbol{y}$ than in $\boldsymbol{x}$ we employ several strategies. We can change all but one of the c's to new characters c_2, c_3, $\ldots c_k$, thus introducing $k - 1$ new characters. When multiple disjoint copies of a substring occur in $\boldsymbol{x}$, we can replace all copies of a symbol within one copy of the substring with a new symbol which does not occur elsewhere in $\boldsymbol{x}$. Given $\boldsymbol{x} = \boldsymbol{uvu}$, we can increase the number of distinct symbols with $\boldsymbol{y} = \boldsymbol{uv\widehat{u}}$. $\boldsymbol{uv\widehat{u}w\widehat{u}}$ has two distinct symbols more than $\boldsymbol{uvuwu}$ does, etc.

Since the length of the string remains constant while the number of distinct characters increases, $\boldsymbol{y} \in S_{d+\delta}(n)$. Since $n - (d + \delta) < n - d$, by the induction hypothesis we know that $r(\boldsymbol{y}) \leq n - d - \delta$. Therefore, $r(\boldsymbol{x}) - \pi \leq r(\boldsymbol{y}) \leq n - d - \delta$, so $\rho_d(n) = r(\boldsymbol{x}) \leq n - d - \delta + \pi$. Thus, whenever $\pi \leq \delta$, $\rho_d(2d) \leq d$.

Lemma 12. *[Lemma]*
Let $\rho_{d'}(2d') \leq d'$ for $d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) \leq d$ or $\boldsymbol{x}$ does not contain a pair.

Proof. As shown in [6], a pair of c's can be involved in at most one run. We change the second c to a new symbol c_2 creating $\boldsymbol{y} = \boldsymbol{u_0} c \boldsymbol{u_1} c_2 \boldsymbol{u_2}$. We destroy at most the single run which contains the pair ($\pi \leq 1$), and gain 1 symbol ($\delta = 1$). As $\pi \leq \delta$, $\boldsymbol{x}$ satisfies the conjecture or $\boldsymbol{x}$ does not contain a pair. □

Lemma 13. *[Lemma] Let $\rho_{d'}(2d') \leq d'$ for $d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) \leq d$ or $\boldsymbol{x}$ does not contain a triple.*

Proof. If a triple of c's is involved in less than two runs, we can proceed as in the proof of the previous lemma. Let us thus assume that the c's are involved in two runs.

The string has the form $\boldsymbol{x} = \boldsymbol{u_0} c \boldsymbol{u_1} c \boldsymbol{u_2} c \boldsymbol{u_3}$. In this case, we replace two of the c's with new symbols c_2 and c_3 creating $\boldsymbol{y} = \boldsymbol{u_0} c \boldsymbol{u_1} c_2 \boldsymbol{u_2} c_3 \boldsymbol{u_3}$. This destroys at most only

the two possible runs, while we gain two symbols ($\delta = 2$). δ is again sufficiently large, so either $\boldsymbol{x}$ satisfies the conjecture or $\boldsymbol{x}$ does not have a triple. □

For the above two lemmas, we did not need to use the notion of mappings. But it can be seen that the runs involved only corresponded to single-mappings. If only single-mappings are involved, then it is straight-forward to obtain a new string with more distinct symbols while limiting the number of runs destroyed. In the following cases, we must always deal with a multi-mapping.

Lemma 14. *[Lemma] Let $\rho_{d'}(2d') \leq d'$ for $d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) \leq d$ or $\boldsymbol{x}$ does not contain a quadruple.*

Proof. A quadruple of c's at positions $i_1 < i_2 < i_3 < i_4$ can be involved in at most four runs, corresponding to a double-mapping $(i_1, i_2) \to (i_3, i_4)$ and single-mappings $i_1 \to i_2$, $i_2 \to i_3$, and $i_3 \to i_4$. If there are only three or fewer runs the c's are involved in, replacing three occurrences of c's by three new symbols will give $\delta = 3$ and $\pi \leq 3$, hence $\pi \leq \delta$, giving the result of this lemma.

Hence we assume that the c's are involved in exactly four runs. In this case replacing three of the c's by new symbols is no longer enough, as π would be greater than δ. However, from the double-mapping $(i_1, i_2) \to (i_3, i_4)$, we know that $\boldsymbol{x}[i_1..i_2] = \boldsymbol{x}[i_3..i_4]$. Thus if $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_4}$, then $\boldsymbol{u_1} = \boldsymbol{u_3}$, hence $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_4}$.

If $\boldsymbol{u_1}$ is non-empty, let $a \in \boldsymbol{u_1}$. We replace the last three copies of c by new symbols c_2, c_3, and c_4, and all instances of a in the second occurrence of $\boldsymbol{u_1}$ by a new symbol a_1 producing $\widehat{\boldsymbol{u_1}}$: $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2\boldsymbol{u_2}c_3\widehat{\boldsymbol{u_1}}c_4\boldsymbol{u_4}$. This gives $\pi \leq 4$, but now $\delta = 4$, satisfying the lemma.

If $\boldsymbol{u_1}$ were empty, either $\boldsymbol{u_2}$ is non-empty, giving $\boldsymbol{x}$ the form: $\boldsymbol{x} = \boldsymbol{u_0}$ c c $\boldsymbol{u_2}$ c c $\boldsymbol{u_4}$. Since $\boldsymbol{u_2}$ is non-empty, in order for the single mapping $i_2 \to i_3$ to exist, there is a symbol in $\boldsymbol{u_2}$ which must occur between the first and second c's, or the third and fourth c's. However, this requires $\boldsymbol{u_1}$ to be non-empty, a contradiction. Therefore, the mapping $i_2 \to i_3$ cannot refer to a run in the string, a contradiction with our assumption of the c's being involved in four different runs.

Therefore, $\boldsymbol{u_2}$ must be empty as well and so we have $\boldsymbol{x} = \boldsymbol{u_0}cccc\boldsymbol{u_4}$, merging the 4 possible runs containing the quadruple into a single run, a contradiction. □

Lemma 15. *[Lemma] Let $\rho_{d'}(2d') \leq d'$ for $d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) \leq d$ or $\boldsymbol{x}$ does not contain a quintuple.*

Proof. A quintuple of c's at positions $i_1 < i_2 < i_3 < i_4 < i_5$ can be involved in at most 5 runs despite there being 6 possible mappings: double-mappings $(i_1, i_2) \to (i_3, i_4)$ and $(i_2 \to i_3) \to (i_4, i_5)$, and single-mappings $i_1 \to i_2$, $i_2 \to i_3$, $i_3 \to i_4$, and $i_4 \to i_5$. If both double-mappings exist, they correspond to the same run, as they have the same period p and overlap by at least p.

Again, if the quintuple is involved in fewer than 5 runs, we can just replace 4 of the c's with new symbols as in the previous lemmas. Thus we are assuming that the quintuple is involved in exactly 5 runs. However, in this case we do not need to introduce 5 new symbols, as we can always introduce 1 new symbol while only destroying a single run. There are 3 cases to discuss:

1. All mappings exist. Then $\boldsymbol{x}[i_5]$ is involved in two runs, one corresponding to $(i_1, i_2) \to (i_3, i_4)$ and $(i_2, i_3) \to (i_4, i_5)$, and one corresponding to $i_4 \to i_5$. If we replace $\boldsymbol{x}[i_5]$ by a new symbol c_5, we destroy the run corresponding to $i_4 \to i_5$, but only a part of the run corresponding to $(i_1, i_2) \to (i_3, i_4)$ and $(i_2 \to i_3) \to (i_4, i_5)$. We thus obtain $\pi \leq 1 = delta$.

2. The mapping $(i_1, i_2) \to (i_3, i_4)$ exists, but $(i_2, i_3) \to (i_4, i_5)$ does not, while all single-mappings exist. We can proceed as in the previous case.
3. The mapping $(i_1, i_2) \to (i_3, i_4)$ does not exist, but $(i_2, i_3) \to (i_4, i_5)$ does, while all possible single-mappings exist. We proceed as in the first case, but with $\boldsymbol{x}[\boldsymbol{i_1}]$ instead of $\boldsymbol{x}[\boldsymbol{i_5}]$. □

Lemma 16. *[Lemma] Let $\rho_{d'}(2d') \leq d'$ for $d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) \leq d$ or $\boldsymbol{x}$ does not contain a 6-tuple.*

Proof. A 6-tuple at positions $i_1 < \ldots < i_6$ can be involved in at most 8 runs, despite there being 9 available mappings:

– triple-mapping: $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$
– double-mappings: $(i_1, i_2) \to (i_3, i_4)$, $(i_2 \to i_3) \to (i_4, i_5)$, and $(i_3, i_4) \to (i_5, i_6)$
– single-mappings: $i_1 \to i_2$, $i_2 \to i_3$, $i_3 \to i_4$, $i_4 \to i_5$, and $i_5 \to i_6$

As in Lemma 15, if either both $(i_1, i_2) \to (i_3, i_4)$ and $(i_2 \to i_3) \to (i_4, i_5)$, or $(i_2 \to i_3) \to (i_4, i_5)$ and $(i_3, i_4) \to (i_5, i_6)$ exist, the two runs they correspond to are actually the same run.

Let $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_4}c\boldsymbol{u_5}c\boldsymbol{u_6}$. We consider each configuration of of multi-mappings separately:

1. $(i_1, i_2) \to (i_3, i_4)$, $(i_3, i_4) \to (i_5, i_6)$, and all single-mappings: By the double-mappings, $\boldsymbol{u_1} = \boldsymbol{u_3} = \boldsymbol{u_6}$, and therefore the string $\boldsymbol{x}$ has the form: $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_4}c\boldsymbol{u_1}c\boldsymbol{u_6}$. We consider the different cases of empty and non-empty substrings separately:
   (a) If $\boldsymbol{u_1}$ is non-empty, we replace 5 of the c's with new symbols, and all instances of some symbol in 2 of the 3 copies of $\boldsymbol{u_1}$, giving $\widehat{\boldsymbol{u_1}}$. So, $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2\boldsymbol{u_2}c_3\widehat{\boldsymbol{u_1}}c_4\boldsymbol{u_4}c_5\widehat{\boldsymbol{u_1}}c_6\boldsymbol{u_6}$. This gives $\pi \leq 7 = \delta$.
   (b) Otherwise, $\boldsymbol{u_1}$ is empty. Assume that both $\boldsymbol{u_2}$ and $\boldsymbol{u_4}$ are non-empty. The string then has the form: $\boldsymbol{x} = \boldsymbol{u_0}cc\boldsymbol{u_2}cc\boldsymbol{u_4}cc\boldsymbol{u_6}$. This eliminates the possibility of runs from the single-mappings $i_2 \to i_3$ and $i_4 \to i_5$. By replacing 5 of the c's with new symbols, we have $\pi \leq 5 = \delta$.
2. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$ and all single-mappings: By the triple-mapping, $\boldsymbol{u_1} = \boldsymbol{u_4}$ and $\boldsymbol{u_2} = \boldsymbol{u_5}$. If $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$ are both be empty, then the possible run from the single mapping $i_1 \to i_2$ is merged with the one from $i_2 \to i_3$, and $i_4 \to i_5$ is merged with $i_5 \to i_6$. By replacing 5 of the c's with new symbols, we have $\pi \leq 4 < \delta = 5$. Therefore, assume at least one of $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$ are non-empty. In this case, we can also replace all instances of some symbol in one of them (whichever is non-empty), giving $\boldsymbol{y} = \boldsymbol{u_0}c_1\boldsymbol{u_1}c_2\boldsymbol{u_2}c_3\boldsymbol{u_3}c_4\widehat{\boldsymbol{u_1}}c_5\widehat{\boldsymbol{u_2}}c_6\boldsymbol{u_6}$. This transformation destroys at most 6 runs and introduces 6 or 7 new symbols ($\pi \leq 6 \leq \delta \leq 7$).
3. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, one of $(i_1, i_2) \to (i_3, i_4)$ or $(i_3, i_4) \to (i_5, i_6)$ (but not both), and all the single-mappings: Having one or the other of the double-mappings are clearly mirror cases of each other, so we will assume without loss of generality that $(i_1, i_2) \to (i_3, i_4)$ exists. By the double- and triple-mappings, $\boldsymbol{u_1} = \boldsymbol{u_3} = \boldsymbol{u_4}$ and $\boldsymbol{u_2} = \boldsymbol{u_5}$.
   (a) If $\boldsymbol{u_1}$ is non-empty, replace each instance of a symbol in 2 copies of it, along with 5 of the c's: with new symbols, $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2\boldsymbol{u_2}c_3\widehat{\boldsymbol{u_1}}c_4\widehat{\boldsymbol{u_1}}c_5\boldsymbol{u_2}c_6\boldsymbol{u_6}$. This increases the number of distinct symbols by 7 while destroying at most 7 runs from the mappings ($\pi \leq 7 = \delta$).

(b) If $\boldsymbol{u_1}$ is empty, we have $\boldsymbol{x} = \boldsymbol{u_0}cc\boldsymbol{u_2}ccc\boldsymbol{u_2}c\boldsymbol{u_6}$. This arrangement loses 1 possible run due to merging $i_3 \to i_4$ and $i_4 \to i_5$, and eliminates the possible run from the mapping $i_2 \to i_3$ when $\boldsymbol{u_2}$ is non-empty, since if $\boldsymbol{u_2}$ is empty, all the runs merge down to a single run. This reduces π from 7 down to 5, so by replacing 5 of the c's with new symbols, we achieve $\pi \leq 5 = \delta$.

4. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_2, i_3) \to (i_4, i_5)$ exist, and so do all the single-mappings. By the double- and triple-mappings, $\boldsymbol{u_1} = \boldsymbol{u_2} = \boldsymbol{u_4} = \boldsymbol{u_5}$.
   (a) If $\boldsymbol{u_1}$ is non-empty, we relabel each instance of a symbol in 3 copies of $\boldsymbol{u_1}$: $\boldsymbol{y} = \boldsymbol{u_0}c_1\boldsymbol{u_1}c_2\widehat{\boldsymbol{u_1}}c_3\boldsymbol{u_3}c_4\widehat{\boldsymbol{u_1}}c_5\widehat{\boldsymbol{u_1}}c_6\boldsymbol{u_6}$. This increases the number of distinct symbols by 8 while destroying at most 7 runs ($\pi \leq 7 < \delta = 8$).
   (b) If $\boldsymbol{u_1}$ is empty, $\boldsymbol{x} = \boldsymbol{u_0}ccc\boldsymbol{u_3}ccc\boldsymbol{u_6}$, and 2 single-mappings are lost through merging $i_1 \to i_2$ with $i_2 \to i_3$ and $i_4 \to i_5$ with $i_5 \to i_6$. Replacing 5 of the c's with new symbols is sufficient to give $\pi \leq 5 = \delta$.
5. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_1, i_2) \to (i_3, i_4)$, $(i_3, i_4) \to (i_5, i_6)$ exist, and so do all the single-mappings. From the double- and triple-mappings, $\boldsymbol{u_1} = \boldsymbol{u_2} = \boldsymbol{u_3} = \boldsymbol{u_4} = \boldsymbol{u_5}$. All the possible runs are actually one long run, so the last c may be replaced with a new symbol without destroying any runs. This gives $\pi = 0 < \delta = 1$. □

Lemma 17. *[Lemma] Let $\rho_{d'}(2d') \leq d'$ for $d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) \leq d$ or $\boldsymbol{x}$ does not contain a 7-tuple.*

Proof. A 7-tuple of c's at positions $i_1 < \ldots < i_7$ can be involved in 9 runs, despite there being 12 possible mappings:

- triple-mappings: $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, and$(i_2, i_3, i_4) \to (i_5, i_6, i_7)$
- double-mappings: $(i_1, i_2) \to (i_3, i_4)$, $(i_2, i_3) \to (i_4, i_5)$, $(i_3, i_4) \to (i_5, i_6)$, and $(i_4, i_5) \to (i_6, i_7)$
- single-mappings: $i_1 \to i_2$, $i_2 \to i_3$, $i_3 \to i_4$, $i_4 \to i_5$, $i_5 \to i_6$, and $i_6 to i_7$

As with the overlapping double-mappings, if both triple-mappings are present, they correspond to the same run. As having both triple-mappings cannot increase the possible number of runs, we assume without loss of generality that if a triple-mapping is present, it is $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$.

As with 15Structural properties of run-maximal strings on the main diagonallemma.15, we need every element of the 7-tuple to be covered by a multi-mapping.

There are 5 cases to consider:

1. $(i_1, i_2) \to (i_3, i_4)$, $(i_4, i_5) \to (i_6, i_7)$, and all single-mappings (a total of 8 mappings). Due to the double-mappings, $\boldsymbol{u_1} = \boldsymbol{u_3}$ and $\boldsymbol{u_4} = \boldsymbol{u_6}$, the string $\boldsymbol{x}$ must have the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_4}c\boldsymbol{u_5}c\boldsymbol{u_4}c\boldsymbol{u_7}$.
   (a) $\boldsymbol{u_1}$ non-empty, $\boldsymbol{u_4}$ non-empty: replace all instances of a symbol in 1 copy of each of $\boldsymbol{u_1}$ and $\boldsymbol{u_4}$, along with 6 of the c's with new symbols: $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2\boldsymbol{u_2}c_3\widehat{\boldsymbol{u_1}}c_4\boldsymbol{u_4}c_5\boldsymbol{u_5}c_6\boldsymbol{u_4}c_7\boldsymbol{u_7}$. This destroys at most 8 runs and introduces 8 new symbols ($\pi \leq 8 = \delta$).
   (b) $\boldsymbol{u_1}$ non-empty, $\boldsymbol{u_4}$ empty. The string $\boldsymbol{x}$ then has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}cc\boldsymbol{u_5}cc\boldsymbol{u_7}$. This eliminates the possibility of a run corresponding to the mapping $i_5 \to i_6$, unless $\boldsymbol{u_5}$ is empty, in which case 2 possible runs are lost to merging into one. Replacing all instances of a symbol in 1 copy of $\boldsymbol{u_1}$ along with 6 of the c's by new symbols gives $\pi \leq 7 = \delta$.
   (c) $\boldsymbol{u_1}$ empty, $\boldsymbol{u_4}$ non-empty. This is a reversal of the previous case, and is satisfied accordingly.

(d) $\boldsymbol{u_1}$ and $\boldsymbol{u_4}$ empty. The possibility of runs corresponding to the mappings $i_2 \to i_3$ and $i_5 \to i_6$ are eliminated, so relabeling 6 of the c's gives $\pi \leq 6 = \delta$.

2. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_4, i_5) \to (i_6, i_7)$, and all single-mappings (a total of 8 mappings). From the multi-mappings, $\boldsymbol{u_1} = \boldsymbol{u_4} = \boldsymbol{u_6}$ and $\boldsymbol{u_2} = \boldsymbol{u_5}$, so the string $\boldsymbol{x}$ must have the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_7}$.
   (a) If $\boldsymbol{u_1}$ is non-empty, we replace all instances of a symbol in 2 copies of $\boldsymbol{u_1}$, along with 6 of the c's with new symbols
   $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2\boldsymbol{u_2}c_3\boldsymbol{u_3}c_4\widehat{\boldsymbol{u_1}}c_5\boldsymbol{u_2}c_6\widehat{\boldsymbol{u_1}}c_7\boldsymbol{u_7}$. This gives $\pi \leq 8 = \delta$.
   (b) Otherwise, $\boldsymbol{u_1}$ is empty, so the string $\boldsymbol{x} = \boldsymbol{u_0}cc\boldsymbol{u_2}c\boldsymbol{u_3}cc\boldsymbol{u_2}cc\boldsymbol{u_7}$. When $\boldsymbol{u_2}$ is non-empty, this eliminates the possibility of a run corresponding to the mapping $i_5 \to i_6$, so by replacing all instances of a symbol in a $\boldsymbol{u_2}$ along with 6 of the c's with new symbols, we achieve $\pi \leq 7 = \delta$.
   (c) If $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$ are both empty, 3 possible runs are lost through merging, so relabeling 6 of the c's gives $\pi \leq 5 < \delta = 6$.

3. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_1, i_2) \to (i_3, i_4)$, $(i_4, i_5) \to (i_6, i_7)$, and all the single-mappings (a total of 9 mappings). From the multi-mappings, $\boldsymbol{u_1} = \boldsymbol{u_4} = \boldsymbol{u_6}$ and $\boldsymbol{u_2} = \boldsymbol{u_3} = \boldsymbol{u_5}$.
   (a) If $\boldsymbol{u_1}$ are $\boldsymbol{u_2}$ are both non-empty, replacing all instances of a symbol in 2 copies of each of $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$ along with 6 of the c's with new symbols, gives us $\pi \leq 9 < \delta = 10$.
   (b) If $\boldsymbol{u_1}$ is empty, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}cc\boldsymbol{u_2}c\boldsymbol{u_2}cc\boldsymbol{u_2}cc\boldsymbol{u_7}$. The possible run corresponding to the mapping $i_5 \to i_6$ is eliminated, so replacing all instances of a symbol in 2 copies of $\boldsymbol{u_2}$ along with 6 of the c's with new symbols is sufficient to give $\pi \leq 8 = \delta$.
   (c) If $\boldsymbol{u_2}$ is empty, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}ccc\boldsymbol{u_1}cc\boldsymbol{u_1}c\boldsymbol{u_7}$. The runs corresponding to the mappings $i_2 \to i_3$ and $i_3 \to i_4$ are merged, and the possible run corresponding to the mapping $i_4 \to i_5$ is eliminated, so replacing all instances of a symbol in 2 copies of $\boldsymbol{u_1}$ along with 6 of the c's with new symbols is sufficient to give $\pi \leq 7 < \delta = 8$. □

Lemma 18. *[Lemma] Let $\rho_{d'}(2d') \leq d'$ for $d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) \leq d$ or $\boldsymbol{x}$ does not contain an 8-tuple.*

Proof. 1. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_5, i_6) \to (i_7, i_8)$, and all single-mappings (a total of 9 mappings). By the multi-mappings, the string has the form
$\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_6}c\boldsymbol{u_2}c\boldsymbol{u_8}$.
   (a) If $\boldsymbol{u_2}$ is non-empty, we can replace all instances of a symbol in 2 copies of $\boldsymbol{u_2}$:
   $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2\boldsymbol{u_2}c_3\boldsymbol{u_3}c_4\boldsymbol{u_1}c_5\widehat{\boldsymbol{u_2}}c_6\boldsymbol{u_6}c_7\widehat{\boldsymbol{u_2}}c_8\boldsymbol{u_8}$. This gives $\pi \leq 9 = \delta$.
   (b) Otherwise, $\boldsymbol{u_2}$ is empty, giving $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}cc\boldsymbol{u_3}c\boldsymbol{u_1}cc\boldsymbol{u_6}cc\boldsymbol{u_8}$. This eliminates the possibility of a run from the mapping $i_6 \to i_7$. This means $\pi \leq 8$.
      i. If $\boldsymbol{u_1}$ is non-empty, we can replace all instances of a symbol in 1 of the copies of $\boldsymbol{u_1}$ along with 7 of the c's, giving
      $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2c_3\boldsymbol{u_3}c_4\widehat{\boldsymbol{u_1}}c_5c_6\boldsymbol{u_6}c_7c_8\boldsymbol{u_8}$. This results in $\pi \leq 8 = \delta$.
      ii. If $\boldsymbol{u_1}$ is also empty, the string is structured as $\boldsymbol{x} = \boldsymbol{u_0}ccc\boldsymbol{u_3}ccc\boldsymbol{u_6}cc\boldsymbol{u_8}$. In addition to the elimination of the mapping $i_6 \to i_7$, the runs corresponding to the single mappings $i_1 \to i_2$ and $i_2 \to i_3$ are merged, along with the runs corresponding to the mappings $i_4 \to i_5$ and $i_5 \to i_6$. This reduces the maximum number of runs to $\pi \leq 6$. By relabeling 7 of the c's, we obtain $\pi \leq 6 < \delta = 7$.

2. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_1, i_2) \to (i_3, i_4)$, $(i_5, i_6) \to (i_7, i_8)$, and all single-mappings (a total of 10 mappings). By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_6}c\boldsymbol{u_2}c\boldsymbol{u_8}$.
   (a) If $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$ are both non-empty, we can replace all instances of a symbol in 2 copies of each, along with 7 of the c's:
   $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\widehat{\boldsymbol{u_1}}c\widehat{\boldsymbol{u_1}}c\widehat{\boldsymbol{u_2}}c\boldsymbol{u_6}c\widehat{\boldsymbol{u_2}}c\boldsymbol{u_8}$. This results in $\pi \leq 10 < \delta = 11$.
   (b) If $\boldsymbol{u_1}$ is empty and $\boldsymbol{u_2}$ is non-empty, we have $\boldsymbol{x} = \boldsymbol{u_0}cc\boldsymbol{u_2}ccc\boldsymbol{u_2}c\boldsymbol{u_6}c\boldsymbol{u_2}c\boldsymbol{u_8}$. This eliminates the possibility of a run corresponding to the mapping $i_2 \to i_3$, and merges the runs corresponding to $i_3 \to i_4$ and $i_5 \to i_6$, so $\pi \leq 8$. We replace all instances of a symbol in 2 of the copies of $\boldsymbol{u_2}$, giving $\boldsymbol{y} = \boldsymbol{u_0}cc_2\boldsymbol{u_2}c_3c_4c_5\widehat{\boldsymbol{u_2}}c_6\boldsymbol{u_6}c_7\widehat{\boldsymbol{u_2}}c_8\boldsymbol{u_8}$. This results in $\pi \leq 8 < \delta = 9$.
   (c) If $\boldsymbol{u_1}$ is non-empty, and $\boldsymbol{u_2}$ is empty, we have $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}cc\boldsymbol{u_1}c\boldsymbol{u_1}cc\boldsymbol{u_6}cc\boldsymbol{u_8}$. This eliminates the possibility of a run corresponding to the mapping $i_6 \to i_7$ (unless $\boldsymbol{u_6}$ is empty, which results in 3 possible runs being merged). We replace all instances of a symbol in 2 copies of $\boldsymbol{u_1}$, along with 7 of the c's with new symbols, giving $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2c_3\widehat{\boldsymbol{u_1}}c_4\widehat{\boldsymbol{u_1}}c_5c_6\boldsymbol{u_6}c_7c_8\boldsymbol{u_8}$. This results in $\pi \leq 9 = \delta$.
   (d) If $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$ are both empty, we have $\boldsymbol{x} = \boldsymbol{u_0}ccccccc\boldsymbol{u_6}cc\boldsymbol{u_8}$, merging 5 runs corresponding to the single mappings, and preventing the possible run corresponding to $(i_1, i_2) \to (i_3, i_4)$ because because its generator would be non-primitive. Therefore, by replacing 7 of the c's with new symbols, we obtain$\pi \leq 5 < \delta = 7$.
3. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_2, i_3) \to (i_4, i_5)$, $(i_5, i_6) \to (i_7, i_8)$, and all single-mappings (a total of 10 mappings). By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_3}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_6}c\boldsymbol{u_1}c\boldsymbol{u_8}$.
   (a) If $\boldsymbol{u_1}$ is non-empty, we can replace all instances of a symbol in 4 copies of $\boldsymbol{u_1}$, along with 7 of the c's ith new symbols, yielding
   $\boldsymbol{y} = \boldsymbol{u_0}c\boldsymbol{u_1}c_2\widehat{\boldsymbol{u_1}}c_3\boldsymbol{u_3}c_4\widehat{\boldsymbol{u_1}}c_5\widehat{\boldsymbol{u_1}}c_6\boldsymbol{u_6}c_7\widehat{\boldsymbol{u_1}}c_8\boldsymbol{u_8}$. This results in $\pi \leq 10 = \delta$.
   (b) If $\boldsymbol{u_1}$ is empty, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}ccc\boldsymbol{u_3}ccc\boldsymbol{u_6}cc\boldsymbol{u_8}$. This merges the runs corresponding to the mappings $i_1 \to i_2$ with $i_2 \to i_3$, and $i_4 \to i_5$ with $i_5 \to i_6$, and eliminates the possible run corresponding to the mapping $(i_2, i_3) \to (i_4, i_5)$ (unless $\boldsymbol{u_3}$ is empty, in which case the 2 more runs are lost through merging). This gives $\pi \leq 7 = \delta$ by just replacing 7 of the c's with new symbols.
4. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_3, i_4) \to (i_5, i_6)$, $(i_5, i_6) \to (i_7, i_8)$, and all single-mappings (a total of 10 mappings). By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_6}c\boldsymbol{u_2}c\boldsymbol{u_8}$.
   (a) If $\boldsymbol{u_2}$ is non-empty, replace all instances of a symbol in 3 copies of $\boldsymbol{u_2}$ with new symbols, giving $\pi \leq 10 = \delta$.
   (b) If $\boldsymbol{u_2}$ is empty, the runs corresponding to the single mappings $i_2 \to i_3$ and $i_3 \to i_4$ are merged, giving 9 possible runs. If $\boldsymbol{u_1}$ is non-empty, the mapping corresponding to the $i_4 \to i_5$ is also prevented, giving 8 possible runs. (If $\boldsymbol{u_1}$ is empty, 5 possible runs are lost through merging, making the process trivial.) By replacing all instances of some symbol in 1 copy of $\boldsymbol{u_1}$ along with 7 of the c's gives $\pi \leq 8 = \delta$.
5. $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, $(i_1, i_2) \to (i_3 \to i_4)$, $(i_3, i_4) \to (i_5 \to i_5)$, $(i_5, i_6) \to (i_7, i_8)$, and all single-mappings (a total of 11 mappings). By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_6}c\boldsymbol{u_1}c\boldsymbol{u_8}$. If $\boldsymbol{u_1}$ is non-empty, replace all instances of a symbol in 5 copies of $\boldsymbol{u_1}$, along with 7 of the c's with new symbols, giving $\pi \leq 11 < \delta = 12$. Otherwise, $\boldsymbol{u_1}$ is empty, and 4 single runs are lost through being merged, giving $\pi \leq 7 = \delta$.

6. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$ and all single-mappings (a total of 8 mappings), By the quadruple-mapping, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_4}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_8}$. $\boldsymbol{u_1}$, $\boldsymbol{u_2}$ and $\boldsymbol{u_3}$ cannot all be empty (or several runs are merged), so we replace all instances of a symbol in at least 1 of them, along with 7 of the c's with new symbols. This gives $\pi \leq 8 \leq \delta \leq 10$.
7. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_1, i_2) \to (i_3, i_4)$, and all single-mappings (initially 9 runs). Having the double-mapping completely enclosed within one side of the quadruple-mapping, means it exists on the other side of the quadruple-mapping too, so $(i_5, i_6) \to (i_7, i_8)$ also exists. By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_4}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_8}$. This gives a total of 10 runs.
   (a) If $\boldsymbol{u_1}$ is non-empty, replaces all instances of a symbol in 3 of the copies of it, along with 7 of the c's, giving $\pi \leq 10 = \delta$.
   (b) Otherwise, $\boldsymbol{u_1}$ is empty, giving the structure $\boldsymbol{x} = \boldsymbol{u_0}cc\boldsymbol{u_2}cc\boldsymbol{u_4}cc\boldsymbol{u_2}cc\boldsymbol{u_8}$. However, this eliminates the possibility of the single-mappings $i_2 \to i_3$, $i_4 \to i_5$, and $i_6 \to i_7$ (unless $\boldsymbol{u_2}$ or $\boldsymbol{u_4}$ are empty, in which case 4 or 2 possible runs are lost through merging, respectively). This reduces the number of possible runs to at most 7, and we can achieve $\pi \leq 7 = \delta$ by simply replacing 7 of the c's with new symbols.
8. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_2, i_3) \to (i_4, i_5)$, and all single-mappings (a total of 9 mappings): By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_8}$.
   (a) If $\boldsymbol{u_2}$ is non-empty, replace all instances of a symbol in 2 copies of it, along with 7 of the c's with new symbols, giving $\pi \leq 9 = \delta$.
   (b) Otherwise, $\boldsymbol{u_2}$ is empty, giving $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}cc\boldsymbol{u_3}cc\boldsymbol{u_1}cc\boldsymbol{u_3}c\boldsymbol{u_8}$. This eliminates the possibility of a run corresponding to the single mappings $i_3 \to i_4$ and $i_5 \to i_6$ (unless $\boldsymbol{u_1}$ or $\boldsymbol{u_3}$ are empty; in either case, 2 possible runs are lost through merging), giving $\pi \leq 7$, which is achievable by replacing 7 of the c's.
9. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_3, i_4) \to (i_5, i_6)$, and all single-mappings (a total of 9 mappings). By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_4}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_8}$. This same configuration was previously discussed when we assumed it had 10 mappings, so it can be satisfied again in this case.
10. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_1, i_2) \to (i_3, i_4)$, $(i_3, i_4) \to (i_5, i_6)$, and all single-mappings (a total of 10 mappings). By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_4}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_8}$. This same configuration was previously discussed when we assumed it had 10 mappings, so it can be satisfied again in this case.
11. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_2, i_3) \to (i_4, i_5)$, $(i_4, i_5) \to (i_6, i_7)$, and all single-mappings (a total of 10 mappings). By the multi-mappings, $\boldsymbol{u_1} = \boldsymbol{u_5}$, $\boldsymbol{u_2} = \boldsymbol{u_4} = \boldsymbol{u_6}$, and $\boldsymbol{u_3} = \boldsymbol{u_7}$, so the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_3}c\boldsymbol{u_8}$.
   (a) If $\boldsymbol{u_2}$ and one of $\boldsymbol{u_1}$ or $\boldsymbol{u_3}$ is non-empty, replace all instances of a symbol in 1 copy of $\boldsymbol{u_1}$ or $\boldsymbol{u_3}$ and 2 copies of $\boldsymbol{u_2}$, along with 7 of the c's with new symbols, giving $\pi \leq 10 \leq \delta = 10$ or 11.
   (b) If $\boldsymbol{u_2}$ is non-empty, but both $\boldsymbol{u_1}$ and $\boldsymbol{u_3}$ are empty, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}cc\boldsymbol{u_2}cc\boldsymbol{u_2}cc\boldsymbol{u_2}cc\boldsymbol{u_8}$. The possibility of runs corresponding to the mappings $i_2 \to i_3$, $i_4 \to i_5$, and $i_6 \to i_7$ is eliminated, so by replacing 7 of the c's we achieve $\pi \leq 7 = \delta$.

(c) If $\boldsymbol{u_2}$ is empty, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}cc\boldsymbol{u_3}cc\boldsymbol{u_1}cc\boldsymbol{u_3}c\boldsymbol{u_8}$. The possibility of runs corresponding to the mappings $i_3 \to i_4$ and $i_5 \to i_6$ is eliminated. Since neither $\boldsymbol{u_1}$ nor $\boldsymbol{u_3}$ are empty (or many more possible runs are lost through merging), raising 1 copy of each of these gives $\pi \leq 8 < \delta = 9$.

12. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_1, i_2) \to (i_3, i_4)$, $(i_4, i_5) \to (i_6, i_7)$, and all single-mappings (a total of 10 mappings). By the multi-mappings, the string has the form $\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_2}c\boldsymbol{u_1}c\boldsymbol{u_8}$. Therefore,
$\boldsymbol{x} = \boldsymbol{u_0}(c\boldsymbol{u_1}c\boldsymbol{u_2})^3c\boldsymbol{u_1}c\boldsymbol{u_8}$, so we can replace the first c only destroying at most a single run ($\pi \leq \delta = 1$).
13. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_1, i_2, i_3) \to (i_4, i_5, i_6)$, and all single-mappings (a total of 9 mappings). By the multi-mappings, the string has the form
$\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_3}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_3}c\boldsymbol{u_8}$. This merges the possible runs from $i_1 \to i_2$ and $i_2 \to i_3$, as well as $i_4 \to i_5$, $i_5 \to i_6$, and $i_6 \to i_7$, leaving 6 possible runs. Replacing 7 of the c's with new symbols is sufficient to give $\pi \leq 6 < \delta = 7$.
In addition, we can layer up to 2 double-mappings on top of the triple and quadruple mappings, giving a total of 11 mappings. Again, there are at least 3 possible runs lost through merging, giving at most 8 runs. Since $\boldsymbol{u_1}$ and $\boldsymbol{u_3}$ cannot both be empty, we can replace all instances of a symbol in 1 of the copies of $\boldsymbol{u_1}$ or $\boldsymbol{u_3}$. Therefore, $\pi \leq 8 \leq \delta$.
14. $(i_1, i_2, i_3, i_4) \to (i_5, i_6, i_7, i_8)$, $(i_2, i_3, i_4) \to (i_5, i_6, i_7)$, and all single-mappings (a total of 9 mappings). By the multi-mappings, the string has the form
$\boldsymbol{x} = \boldsymbol{u_0}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_4}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_1}c\boldsymbol{u_8}$. This merges the possible runs corresponding to $i_1 \to i_2$, $i_2 \to i_3$, and $i_3 \to i_4$, along with $i_5 \to i_6$, $i_6 \to i_7$, and $i_7 \to i_8$, decreasing the maximum number of runs by 4. By replacing 7 of the c's with new symbols, we get $\pi \leq 5 < 7 = \delta$.
Once again, we can also layer on up to 2 additional double-mappings on top of the triple- and quadruple-mappings. However, we are still limited to 11 possible runs. Less the 4 possible runs lost to merging gives us $\pi \leq 7 = \delta$ from replacing 7 of the c's with new symbols. □

Remark 19. While the previous lemmas were provided for entries on the main diagonal, the result can be generalized to any entry in column $n-d$ where $\rho_{d'}(n') \leq n' - d'$ for $n' - d' < n - d$. Either $\rho_d(n) \leq n - d$, or no run-maximal $\boldsymbol{x} \in S_d(n)$ has a pair, triple, ..., 8-tuple. The induction hypothesis only requires that all entries to the left of the *unknown* column satisfy the conjecture; there is no restriction within the *unknown* column.

Having proven Proposition 9, we can present the proof of Theorem 8:

Proof. The proof follows directly from Proposition 9. If the conjecture does not hold, let d be the first column for which $\rho_d(2d) > d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. By Proposition 9, $\boldsymbol{x}$ has at least $k = \lceil \frac{7d}{8} \rceil$ singletons, and by Lemma 2 they must all be safe. Let us form $\boldsymbol{y}$ by removing all these safe singletons. This gives a string $\boldsymbol{y} \in S_{d-k}(2d-k)$ violating the conjecture, i.e. $r(\boldsymbol{y}) > d$. $d' = d - k = \frac{d}{8}$ and $d = 8d'$ and $2d - k = 9d'$. Thus we found a $\boldsymbol{y} \in S_{d'}(9d')$ such that $r(\boldsymbol{y}) > 8d'$. □

When investigating a single column, the first counter-example in the column cannot have a singleton, as otherwise the counter-example could be *pushed up*. Nor, by Proposition 9, can it contain a k-tuple for $2 \leq k \leq 8$. Theorem 8 together with these facts give a simplified way to computationally *verify* that the whole column d satisfies the conjecture: *show that there are no counter-examples for* $2 \leq d' \leq \frac{d}{8}$, *and*

only strings with no k-tuples, $1 \leq k \leq 8$*, need to be considered when looking for the counter-examples.*

6 Conclusion

The properties presented in this paper constrain the behaviour of the entries in the $(d, n-d)$ table below the main diagonal and in an immediate neighbourhood above the main diagonal. One of the the main contributions lies in the characterization of structural properties of the run-maximal strings on the main diagonal, giving yet another property equivalent with the maximum number of runs conjecture. Not only do these results provide a faster way to computationally check the validity of the conjecture for greater lengths, they indicate a possible way to prove the conjecture along the ideas presented in Proposition 9 and its proof: a first counter-example on the main diagonal could not possibly have a k-tuple for any conceivable k. We were able to carry the reasoning up to $k = 8$, but these proofs are not easy to scale up as the combinatorial complexity increases. The hope and motivation for further research along these lines is that there is a common thread among all these various proofs that may lead to a uniform method ruling out all the k-tuples and thus proving the conjecture, or to exhibit an unexpected counter-example on the main diagonal of the $(d, n-d)$ table. Recent extensions of the parameterized approach shows the unexpected existence of a binary run-maximal string of length 66 containing a substring of four identical symbols $aaaa$, [1]. Similarly, considering squares instead of runs, the approach shows that, among all strings of length 33, no binary string achieves the maximum number of distinct primitively rooted squares [7] .

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: *Run-maximal strings.* website, 2011, `http://optlab.mcmaster.ca/~bakerar2/research/runmax/`.
3. M. Crochemore and L. Ilie: *Maximal repetitions in strings.* Journal of Computer and System Sciences, 74(5) 2008, pp. 796–807.
4. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture.* Lecture Notes in Computer Science, 5029 2008, pp. 290–302.
5. M. Crochemore, L. Ilie, and L. Tinta: *The "runs" conjecture.* website, 2011, `http://www.csd.uwo.ca/faculty/ilie/runs.html`.
6. A. Deza and F. Franek: *A d-step analogue for runs on strings*, AdvOL-Report 2010/02, McMaster University, 2010.
7. A. Deza, F. Franek, and M. Jiang: *Computational framework for determining square-maximal strings*, in Proceedings of Prague Stringology Conference 2012, Prague, Czech Republic, 2012.
8. F. Franek, C. Fuller, J. Simpson, and W. Smyth: *More results on overlapping squares.* to appear.
9. 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.
10. F. Franek, R. Simpson, and W. Smyth: *The maximum number of runs in a string*, in Proceedings of 14th Australasian Workshop on Combinatorial Algorithms AWOCA 2003, Seoul National University, Seoul, Korea, 2008.
11. F. Franek and Q. Yang: *An asymptotic lower bound for the maximal number of runs in a string.* International Journal of Foundations of Computer Science, 19(1) 2008, pp. 195–203.

12. M. Giraud: *Not so many runs in strings*, in LATA 2008, Tarragona, Spain, 2008.
13. C. Iliopoulos, D. Moore, and W. Smyth: *A characterization of the squares in a Fibonacci string.* Theoretical Computer Science, 172 1997, pp. 281–291.
14. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, 1999, pp. 596–604.
15. R. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in Proc. 12th Intl. Symp. on Fund. of Comp. Sci. 1999, vol. 1684, 1999, pp. 374–385.
16. E. Kopylov and W. Smyth: *The three squares lemma revisited.* Journal of Discrete Algorithms, 11 2012, pp. 3–14.
17. M. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Mathematics, 25 1989, pp. 145–153.
18. W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara: *A series of run-rich strings.* Lecture Notes in Computer Science, 5457 2009, pp. 578–587.
19. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *New lower bounds for the maximum number of runs in a string*, in Proceedings of Prague Stringology Conference 2008, Prague, Czech Republic, 2008, pp. 140–145.
20. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *Lower bounds for the maximum number of runs in a string.* website, 2011, `http://www.shino.ecei.tohoku.ac.jp/runs/`.
21. S. Puglisi, R. Simpson, and W. Smyth: *How many runs can a string contain?* Theoretical Computer Science, 401 2008, pp. 165–171.
22. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound.* Lecture Notes in Computer Science, 3884 2006, pp. 184–195.
23. F. Santos: *A counterexample to the Hirsch conjecture.* Annals of Mathematics, 176 2012, pp. 383–412.

Man of Four Research Topics (... so far)

Jan Holub

Department of Theoretical Computer Science
Faculty of Information Technology, Czech Technical University in Prague
Thákurova 9, 160 00 Prague 6, Czech Republic
Jan.Holub@fit.cvut.cz

Abstract. The contribution presents a light overview of the influence of prof. Melichar to compiler construction, stringology, arbology, and 2D pattern matching.

1 Introduction

Bořivoj Melichar called Bob is a man of science. I remember when I was his first year Ph.D. student he told me "On Saturday during my trip I got an idea ...". I was surprised that instead of enjoying the trip and relaxing he was working=doing research. After that I realized that doing real research requires big involvement. He was strongly supporting research in all his positions like head of department and head of research group. He is able to discuss research at any time day or night. (Verified.)

He started in compiler construction and using the same instruments (The automata theory, see Fig. 1) jumped into Stringology in 1995 (using finite automata) then to arbology (using pushdown automata). Each intersection of Bob with research topic resulted in numerous recognized research papers.

2 Compiler Construction

Bob Melichar's research in compiler construction field concerns attributed grammars [1,2], translation directed by LR parsing [3–5], improving LR parsing [6], and parallel LL parsing [7,8].

3 Stringology

Bob started in Stringology by studying simulation of nondeterministic finite automata [9–21]. He also created a classification of pattern matching problems [22–25] and played with cyclic strings [26]. For sequence matching he designed sequence automata [27,28] and for complete index worked with factor automata [29–32]. He searched for repetitions in indexed texts [33–38] and he also contributed to data compression [39–43].

4 Arbology

In Arbology Bob came up with the idea to use pushdown automata to tree processing. First the tree pattern and texts are transformed into one-dimensional text using for example the prefix notation, suffix notation, or bar notation. Then the pushdown automata usage in arbology is inspired by usage of finite automata in Stringology [44–53].

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

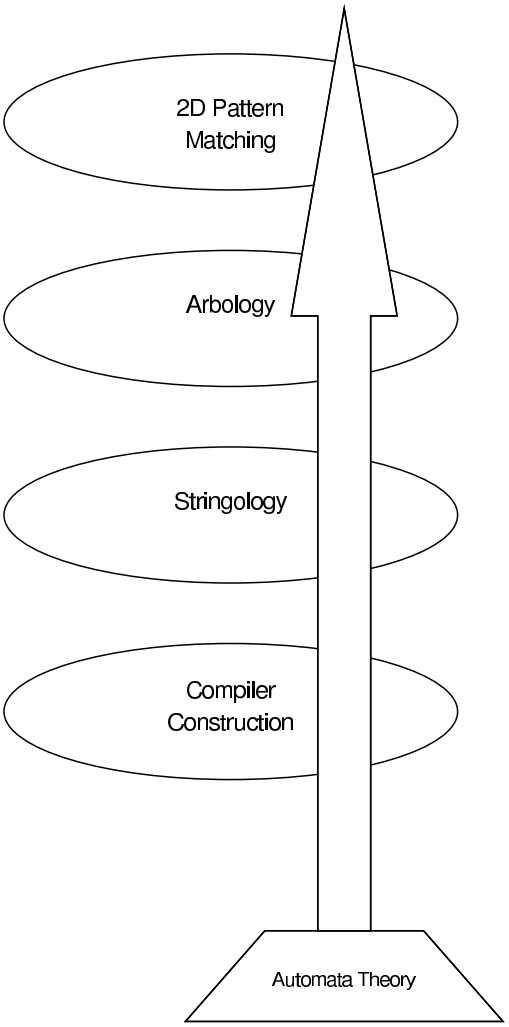

Figure 1. Automata theory in research topics

5 2D Pattern Matching

Bob's work in 2D pattern matching is focused on usage of finite automata [54,55] and tree-based indexing [56,57].

6 Conclusions

On all the research described Bob collaborated with his Ph.D. students and his colleagues from Czech Republic as well as from abroad. Bob's involvement in research is unflagging and new research publications appear.

References

1. Rieks op den Akker, Bořivoj Melichar, and Jorma Tarhio. The hierarchy of LR-attributed grammars. In Pierre Deransart and Martin Jourdan, editors, *WAGA*, volume 461 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1990.
2. Rieks op den Akker, Bořivoj Melichar, and Jorma Tarhio. Attribute evaluation and parsing. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 187–214. Springer, 1991.

3. Bořivoj Melichar. Syntax directed translation with LR parsing. In Uwe Kastens and Peter Pfahler, editors, *CC*, volume 641 of *Lecture Notes in Computer Science*, pages 30–36. Springer, 1992.
4. Jan Janoušek and Bořivoj Melichar. The output-store formal translator directed by LR parsing. In Frantisek Plasil and Keith G. Jeffery, editors, *SOFSEM*, volume 1338 of *Lecture Notes in Computer Science*, pages 432–439. Springer, 1997.
5. Jan Janoušek and Bořivoj Melichar. Formal translations described by translation grammars with LR(k) input grammars. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *PLILP*, volume 1292 of *Lecture Notes in Computer Science*, pages 421–422. Springer, 1997.
6. John Aycock, R. Nigel Horspool, Jan Janoušek, and Bořivoj Melichar. Even faster generalized LR parsing. *Acta Inf.*, 37(9):633–651, 2001.
7. Ladislav Vagner and Bořivoj Melichar. Parallel LL parsing. *Acta Inf.*, 44(1):1–21, 2007.
8. Ladislav Vagner and Bořivoj Melichar. Formal translation directed by parallel LLP parsing. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, editors, *SOFSEM (1)*, volume 4362 of *Lecture Notes in Computer Science*, pages 532–543. Springer, 2007.
9. B. Melichar. Approximate string matching by finite automata. In V. Hlaváč and R. Šára, editors, *Computer Analysis of Images and Patterns*, number 970 in Lecture Notes in Computer Science, pages 342–349. Springer-Verlag, Berlin, 1995.
10. B. Melichar. Space complexity of linear time approximate string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 28–36, Czech Technical University in Prague, Czech Republic, 1996. Collaborative Report DC–96–10.
11. B. Melichar. String matching with k differences by finite automata. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume II., pages 256–260, Vienna, Austria, 1996. IEEE Computer Society Press.
12. J. Holub and B. Melichar. Implementation of nondeterministic finite automata for approximate pattern matching. In *Proceedings of the 3rd International Workshop on Implementing Automata*, pages 74–81, Rouen, France, 1998.
13. J. Holub and B. Melichar. Pattern matching and finite automata. In *Proceedings of the Summer School of Information Systems and Their Applications*, pages 154–81, Ruprechtov, Czech Republic, 1998.
14. J. Holub and B. Melichar. Algorithms for pattern matching. In *Proceedings of the Summer School of Information Systems and Their Applications*, pages 69–78, Ruprechtov, Czech Republic, 1999.
15. J. Holub and B. Melichar. Implementation of nondeterministic finite automata for approximate pattern matching. In J.-M. Champarnaud, D. Maurel, and D. Ziadi, editors, *Proceedings of the 3rd International Workshop on Implementing Automata'98*, number 1660 in Lecture Notes in Computer Science, pages 92–99, Rouen, France, 1999. Springer-Verlag, Berlin.
16. J. Holub, C. S. Iliopoulos, B. Melichar, and L. Mouchard. Distributed string matching using finite automata. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 114–128, Perth, WA, Australia, 1999.
17. B. Melichar and J. Skryja. On the size of deterministic finite automata. In *Implementation and Application of Automata*, volume 2494 of *Lecture Notes in Computer Science*, pages 202–213. Springer-Verlag, Berlin, 2002.
18. C. S. Iliopoulos, I. Jayasekera, B. Melichar, and J. Šupol. Weighted degenerated approximate pattern matching. In *Proceedings of the 1st International Conference on Language and Automata Theory and Applications*, Tarragona, Spain, March 2007.
19. Jan Šupol and Bořivoj Melichar. Two-dimensional bitwise memory matrix: A tool for optimal parallel approximate pattern matching. In Jan Holub and Jan Žďárek, editors, *Stringology*, pages 18–28. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2006.
20. Jan Šupol and Bořivoj Melichar. A new approach to determinisation using bit-parallelism. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, pages 228–241. Springer, 2006.
21. Costas S. Iliopoulos, Inuka Jayasekera, Bořivoj Melichar, and Jan Šupol. Weighted degenerated approximate pattern matching. In Remco Loos, Szilárd Zsolt Fazekas, and Carlos Martín-Vide, editors, *LATA*, volume Report 35/07, pages 285–296. Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, Tarragona, 2007.

22. B. Melichar, J. Holub, and T. Polcar. Text searching algorithms. Vol. I: Forward string matching. Textbook for course Text Searching Algorithms, 2005.
23. B. Melichar and J. Holub. 6D classification of pattern matching problems. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 24–32, Czech Technical University in Prague, Czech Republic, 1997. Collaborative Report DC–97–03.
24. Jan Antoš and Bořivoj Melichar. Backward pattern matching automaton. In Holub and Šimánek [58], pages 81–94.
25. Jan Antoš and Bořivoj Melichar. Finite automata for generalized approach to backward pattern matching. In Domaratzki and Salomaa [60], pages 49–58.
26. Bořivoj Melichar. Deterministic parsing of cyclic strings. In Jean-Marc Champarnaud and Denis Maurel, editors, *CIAA*, volume 2608 of *Lecture Notes in Computer Science*, pages 301–306. Springer, 2002.
27. Maxime Crochemore, Bořivoj Melichar, and Zdeněk Troníček. Directed acyclic subsequence graph – overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003.
28. Bořivoj Melichar and Tomáš Polcar. The longest common subsequence problem a finite automata approach. In Oscar H. Ibarra and Zhe Dang, editors, *CIAA*, volume 2759 of *Lecture Notes in Computer Science*, pages 294–296. Springer, 2003.
29. B. Melichar. A simple method of complete indexing. In *Proceedings DATASEM*, pages 183–188, Brno, Czech Republic, 1997.
30. J. Holub and B. Melichar. Approximate string matching using factor automata. In C. S. Iliopoulos, editor, *Proceedings of the 9th Australasian Workshop On Combinatorial Algorithms*, pages 28–39, Perth, WA, Australia, 1998.
31. J. Holub and B. Melichar. Approximate string matching using factor automata. *Theor. Comput. Sci.*, 249(2):305–311, 2000.
32. J. Holub, C. S. Iliopoulos, B. Melichar, and L. Mouchard. Distributed pattern matching using finite automata. *J. Autom. Lang. Comb.*, 6(2):191–204, 2001.
33. B. Melichar. Repetitions in text and finite automata. In L. Cleophas and B. W. Watson, editors, *Proceedings of the Eindhoven FASTAR Days 2004*, pages 1–46. TU Eindhoven, The Netherlands, 2004.
34. P. Antoniou, J. Holub, C. S. Iliopoulos, B. Melichar, and P. Peterlongo. Finding common motifs with gaps using finite automata. In O. H. Ibarra and H.-C. Yen, editors, *Implementation and Application of Automata*, number 4094 in Lecture Notes in Computer Science, pages 69–77. Springer-Verlag, Heidelberg, 2006.
35. M. Šimůnek and B. Melichar. Borders and finite automata. In O. H. Ibarra and H.-C. Yen, editors, *Implementation and Application of Automata*, number 4094 in Lecture Notes in Computer Science, pages 58–68. Springer-Verlag, Heidelberg, 2006.
36. Martin Šimůnek and Bořivoj Melichar. Borders and finite automata. *Int. J. Found. Comput. Sci.*, 18(4):859–871, 2007.
37. M. Šimůnek and B. Melichar. Approximate periods with levenshtein distance. In Ibarra and Ravikumar [59], pages 286–287.
38. O. Guth and B. Melichar. Finite automata approach to computing all seeds of strings with the smallest hamming distance. *IAENG International Journal of Computer Science*, 36(2), 2009.
39. J. Lahoda and B. Melichar. Pattern matching in text coded by finite translation automaton. In M. Heričko et al., editor, *Proceedings of the 7th International Multiconference Information Society*, pages 212–214, Institut Jožef Stefan, Ljubljana, Slovenia, 2004.
40. Jan Šupol and Bořivoj Melichar. Arithmetic coding in parallel. In Milan Šimánek and Jan Holub, editors, *Stringology*, pages 168–187. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2004.
41. Jan Šupol and Bořivoj Melichar. Arithmetic coding in parallel. *Int. J. Found. Comput. Sci.*, 16(6):1207–1217, 2005.
42. Jan Lahoda and Bořivoj Melichar. General pattern matching on regular collage system. In Holub and Šimánek [58], pages 153–162.
43. J. Lahoda, B. Melichar, and J. Žd'árek. Pattern matching in DCA coded text. In Ibarra and Ravikumar [59], pages 151–160.
44. Jan Janoušek and Bořivoj Melichar. On regular tree languages and deterministic pushdown automata. *Acta Inf.*, 46(7):533–547, 2009.
45. Tomáš Flouri, Bořivoj Melichar, and Jan Janoušek. Subtree matching by deterministic pushdown automata. In *IMCSIT*, pages 659–666. IEEE, 2009.

46. Tomáš Flouri, Jan Janoušek, and Bořivoj Melichar. Subtree matching by pushdown automata. *Comput. Sci. Inf. Syst.*, 7(2):331–357, 2010.
47. Tomáš Flouri, Bořivoj Melichar, and Jan Janoušek. Aho-corasick like multiple subtree matching by pushdown automata. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 2157–2158. ACM, 2010.
48. Bořivoj Melichar. Arbology: Trees and pushdown automata. In Adrian Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2010.
49. Michalis Christou, Maxime Crochemore, Tomáš Flouri, Costas S. Iliopoulos, Jan Janoušek, Bořivoj Melichar, and Solon P. Pissis. Computing all subtree repeats in ordered ranked trees. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *SPIRE*, volume 7024 of *Lecture Notes in Computer Science*, pages 338–343. Springer, 2011.
50. Martin Plicka, Jan Janoušek, and Bořivoj Melichar. Subtree oracle pushdown automata for ranked and unranked ordered trees. In Ganzha et al. [61], pages 903–906.
51. Tomáš Flouri, Jan Janoušek, Bořivoj Melichar, Costas S. Iliopoulos, and Solon P. Pissis. Tree indexing by pushdown automata and repeats of subtrees. In Ganzha et al. [61], pages 899–902.
52. Jan Travnicek, Jan Janoušek, and Bořivoj Melichar. Nonlinear tree pattern pushdown automata. In Ganzha et al. [61], pages 871–878.
53. Tomáš Flouri, Jan Janoušek, Bořivoj Melichar, Costas S. Iliopoulos, and Solon P. Pissis. Tree template matching in ranked ordered trees by pushdown automata. In Béatrice Bouchou-Markhoff, Pascal Caron, Jean-Marc Champarnaud, and Denis Maurel, editors, *CIAA*, volume 6807 of *Lecture Notes in Computer Science*, pages 273–281. Springer, 2011.
54. Tomáš Polcar and Bořivoj Melichar. Two-dimensional pattern matching by two-dimensional online tessellation automata. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *CIAA*, volume 3317 of *Lecture Notes in Computer Science*, pages 327–328. Springer, 2004.
55. Jan Ždárek and Bořivoj Melichar. On two-dimensional pattern matching by finite automata. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *CIAA*, volume 3845 of *Lecture Notes in Computer Science*, pages 329–340. Springer, 2005.
56. Jan Ždárek and Bořivoj Melichar. A note on a tree-based 2D indexing. In Domaratzki and Salomaa [60], pages 300–309.
57. Jan Ždárek and Bořivoj Melichar. Tree-based 2D indexing. *Int. J. Found. Comput. Sci.*, 22(8):1893–1907, 2011.
58. Jan Holub and Milan Šimánek, editors. *Proceedings of the Prague Stringology Conference, Prague, Czech Republic, August 29-31, 2005*. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2005.
59. Oscar H. Ibarra and Bala Ravikumar, editors. *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings*, volume 5148 of *Lecture Notes in Computer Science*. Springer, 2008.
60. Michael Domaratzki and Kai Salomaa, editors. *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers*, volume 6482 of *Lecture Notes in Computer Science*. Springer, 2011.
61. Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors. *Federated Conference on Computer Science and Information Systems - FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*, 2011.

On My Friendship with Bob Melichar

Jan Janousek

Department of Theoretical Computer Science
Faculty of Information Technology, Czech Technical University
Thakurova 9
160 00 Prague 6
Czech Republic
Jan.Janousek@fit.cvut.cz

This article describes certain substantial parts of my life and its main purpose is to show some characteristics of Borek Melichar (Bob) and also the fact that Bob is one of the key persons in my life. Perhaps the description from a personal point of view might be a good way of transferring my own experiences to the reader.

First of all, I should mention in brief how I met Bob for the first time. In my childhood my father gave me sometimes technically oriented books that could be interesting for me. At the age of thirteen I got a preliminary version of a famous Czech book for programmers, Programming in Language Pascal by Karel Muller, who was later my colleague from our faculty and department (unfortunately, Karel died two years ago). This book captured my interest indeed and I started to be keen on computers, which becomes my main hobby. I can still remember my first home computer Sinclair ZX Spectrum, which was a fantastic piece of electronics. Because of my hobby I was attending secondary school Arabska, which was focused on programming and where my other current colleagues Bozena Mannova and Ivan Halaska worked as my teachers. We had some of our lessons at Karlovo namesti at the Department of Computer Science and Engineering, Faculty of Electrical Engeneering of the Czech Technical University, where we worked on computer Tesla 200 running compilers of Pascal and Cobol programming languages. Therefore, when I considered studies at a university studying at the Department of Computer Science and Engineering was a natural choice. When I studied there Bob was our teacher, an associate professor and a legendary head of the department. For me (and for many my colleagues–students), Bob certainly was a computer science guru to whom we looked up. After my graduation I asked Bob for a position of his PhD student, but he replied me in a sense that I had not been his diploma thesis student and he gave me some papers and offered me to go to the Computational Centre of the Czech Army where I could spend my compulsory military service. The military service time was to show whether my interest in PhD study would have been real or not. During free time in army I read many papers on the formal translation directed by LR parsing and I think out some new ideas in this topic. After my military service, based on my research considerations, Bob took me as his PhD student.

Working with Bob in his research group gave me a lot, it gave me most of my basics how to work. Bob was always able to read and correct various versions of our papers, which we were writing together. We always had some topics on which we were doing our research. Bob can create the feeling of a close community, in which working is a real joy for its members. Generally, he has always been involved personally in building a strong computer science community. Many times we considered and thought of a problem together and in mutual discussions. When I was on my first computer science conference (SOFSEM) as a PhD student I knew and felt that I would want to become a part of this computer science community. My PhD studies was, above all,

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

the time of hard work – when I traveled by tram, cooked dinner, did almost anything I was considering our problems. I received my PhD in 2001, became a teacher at our university. I must say that I love my work and Bob certainly is the person who contributed the most to preparing me for this profession. He has been a perfect leader and educated many PhD students. In these days, I am an associate professor and the head of our department, which are the same positions as Bob's ones in times of my own MSc. studies. Since 2008 we have been working with Bob on our new research topic on algorithms on trees, which we call Arbology. I think we have been enjoying our research very much. I have been trying to continue in the same way of building strong computer science community among my own students as I have learned from Bob.

Meanwhile, many personal connections have appeared and occurred between me and Bob. Generally, when I am considering the past, I can say that I have never seen Bob lying (despite every person may have lied sometimes, I am not aware of any Bob's lying). Furthermore, he always tries to do what he said and promised. I can say that Bob is extremely faithful from its human fundamental. He has never been a member of Communist party. I think we became close friends with Bob and I like his moral principles very much. I am very proud of being his friend.

I always thought that I would have my wedding at the age of 35. It was 3 days before my 36th birthday when I get married with my wife Katka (we had known each other for 10 years and had been friends before Katka became my girl–friend and later my wife). And with my wife, my connections with Bob got even stronger, although neither I, Katka nor Bob had known this. These new connections are:

- Katka's parents Kveta and Karel were Bob's colleagues–students during his studies at the Czech Technical University. They studied at the same year–class and are friends.
- When Bob was eighteen and came to his own admission exams to the university, one girl had already been in the room waiting for the beginning of the exams. This girl was Kveta, Katka's mother.
- My wife had a best friend Jitka during her studies at the Charles University, Faculty of Mathematics. And Jitka is Bob's daughter! It is to be mentioned here that Katka had never seen Bob personally before our wedding day.

Many other connections between us exist: Our cottage is 10 km far from Bob's home in Mnisek pod Brdy. My father knows Bob from SOFSEM conferences in the 80s. And I must have forgotten something...

I must thank Bob for many things: for his friendship, for transferring me many of his own experiences and for learning me a lot; I use all the knowledge which I have got from him in my own life.

Bob celebrates his 70th birthday on 30th August 2012. Bob, thank you indeed for all, and all the best to you to the upcoming years.

A Rabin Karp Hash for Approximate Automata

Shmuel Tomi Klein

Department of Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel
tomi@cs.biu.ac.il

Abstract. In [2,3], Coetser, Watson & Kourie give an adaptation of Brzozowski's well-known derivatives construction [1] which maps a regular expression to an equivalent automaton. In Brzozowski's algorithm, the *derivative* regular expressions of the input regular expression are computed along with the transition function; subsequently, these regular expressions are mapped to integers, yielding the automaton. In the adapted algorithm, for construction performance reasons the regular expressions are *hashed* to integers using some has function h, possibly leading to collisions. Rather than disambiguating such collisions, they are allowed, thereby *merging* some states and producing an approximate automaton — on which possibly accepts more strings than just those denoted by the input regular expression. In this short paper, we consider how to quantify the rate of collisions and make the probability of collision arbitrarily small, thereby minimizing the 'approximateness' of the resulting automaton while keeping the efficiency gains of the adapted algorithm.

1 Background

Let x_i be the regular expressions and $y_i = h(x_i)$. Consider using an idea similar to the one used in the Rabin and Karp probabilistic pattern matching algorithm. Instead of working with the elements y_i using a sophisticated hash function, encode the regular expressions x_i in the simplest way that can assure injectivity. One could for example encode atomic symbols in ASCII or some other fixed code (e.g. Unicode), set aside special codewords for operators, and concatenate all of that for each expression. The encoding of each x_i can thus be quite long (proportional to the length of the expression itself), but needs some a priori upper limit m on its length, which could be quite large. Denote by z_i the integer represented by this encoding of x_i. As a second layer, apply then a hash using the remainder of z_i modulo a large random prime number p that is fixed in advance for the whole algorithm. The steps are thus:

1. Choose a suitable constant k as the size of the prime number
2. Choose a random prime number p with k bits
3. Define $\overline{z}_i \longleftarrow z_i \bmod p$ for all $1 \leq i \leq n$
4. Work with $\overline{z}_i$ rather than with z_i

Even though the z_i could be thousands of bits long, the actual work is done with the $\overline{z}_i$, each of which is limited to a length of k bits. Using the modulo can obviously lead to collisions, but we now show that with a suitable choice of k, the probability for such collisions can be made arbitrarily small, so that they can safely be ignored.

2 Probability of collision

When coming to evaluate the probability of a collision, we cannot rely on any knowledge about the distribution of the occurrence of the various possible forms of the

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

regular expressions. The only input parameter of our problem on which such a probability might depend is the randomly chosen prime number p. For a good choice of p, all the different z_i will also yield different $\overline{z}_i$, that is, there will be no collision, and thus no loss at all in the construction of the automaton. But if p is poorly chosen, there are going to be indices i and j such that $z_i \neq z_j$ and nevertheless $\overline{z}_i = \overline{z}_j$. Let us call such a prime a "bad" prime.

To study the properties of bad primes, define the following number:

$$H = \prod_{1 \leq i < n} \prod_{\substack{i < j \leq n \\ z_i \neq z_j}} (z_i - z_j).$$

H is the product of the differences of all the possible pairs of z_i, where it has been taken care of to consider only those pairs for which $z_i \neq z_j$. Thus it follows that $H \neq 0$. On the other hand, consider

$$\overline{H} = \quad H \bmod p = \prod_{1 \leq i < n} \prod_{\substack{i < j \leq n \\ z_i \neq z_j}} (\overline{z}_i - \overline{z}_j).$$

If p is a bad prime, at least one of the factors of the product is zero, so we conclude that if p is bad then $\overline{H} = 0$, but that means that p is a divisor of H.

Denote by ℓ the number of bad primes, and denote these by $p_1, \ldots, p_\ell$. Each of the p_i divides H, and therefore also their product $P = \prod_{i=1}^{\ell} p_i$ divides H (here we use the fact that they are prime). If P divides H then necessarily $P \leq H$. From this inequality we derive one on the number of bits necessary for each of these quantities: P is a product of ℓ primes of k bits each, H is a product of about $\frac{1}{2}n^2$ factors of m bits each, so we get

$$k\ell \leq \frac{1}{2} n^2 m.$$

We are now ready to evaluate the probability of a collision, which is the probability of choosing a bad prime, which is the number of bad primes divided by the total number of possible primes. As is well known, the number of primes of k bits is $\Theta\left(\frac{2^k}{k}\right)$. So we get

$$\text{Prob(collision)} = \text{Prob(choosing bad prime)} = \frac{\text{\# bad primes}}{\text{total \# primes}} = \frac{\ell}{\frac{2^k}{k}} = \frac{k\ell}{2^k} \leq \frac{n^2 m}{2^{k+1}}.$$

To take a concrete real life example, suppose there are about a million regular expressions and that the length m of the encoding of each z_i is limited by 65000 bits. Choosing then the length of p to be 10 bytes (80 bits) will give a probability of collision smaller than

$$\frac{\left(2^{20}\right)^2 \quad 2^{16}}{2^{81}} = 2^{-35} = \frac{1}{32,000,000,000}.$$

References

1. J. A. Brzozowski: *Derivatives of regular expressions.* Journal of the ACM, 11(4) 1964, pp. 481–494.
2. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size*, in Proceedings of the Prague Stringology Conference 2008, J. Holub and J. Žd'árek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 227–241.
3. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size.* International Journal of Foundations of Computer Science, 20(6) 2009, pp. 1069–1086.

Formal Concept Analysis Applications in Stringology

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

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

Abstract. The application of formal concept analysis technology to several stringology-based problems is described. The problems include the construction of taxonomies of stringological algorithms, 2D pattern matching and multiple keyword pattern matching, the generation of failure deterministic finite automata, and the conversion of nondeterministic finite automata to deterministic finite automata.

1 Introduction

Questions regarding the idea of a *concept* have intrigued philosophers through the ages, reaching back as far as Aristotle.

> What is a concept? What does the term mean? How, if at all, does the *concept of a table* differ from the *set of all tables*? Can one define a concept when a definition is itself a concept? etc.

From the beginning, it was clear that the notion of a concept is related to the notion of an abstraction, and it was also clear that the ability to abstract is strongly associated with human learning and intelligence. Hence, it is not surprising that with the rise of research interest in machine learning as a subarea of Artificial Intelligence (AI), there arose also a concurrent interest in developing a mathematically formalised and computationally amenable notion of a concept. The instinct was that if one could translate what it means to conceptualise into a computational task, then one would have a platform for machine learning, and hence for AI.

One response to this instinct was a mathematical formalisation of the notion of a concept as developed by a research group in Darmstadt, led by Wille, Ganter and Murmeister in the early 1980s. This formalisation has subsequently formed the heart of a field of study known as formal concept analysis (FCA), which will be introduced briefly in Section 2. It will be seen that the theory relies on a context consisting of a finite number of discrete objects and attributes. The finiteness and discreteness reflect the way in which AI was approached in the 1980s and 1990s—a way which has come to be characterised as symbolic AI. Since symbolic AI systems (FCA systems included) are prone to state space explosions, the twenty-first century has seen a shift away from symbolic AI and machine learning towards so-called computational intelligence where technologies such as artificial neural networks are used to simulate learning.

Nevertheless, because FCA offers a powerful and comprehensive approach to clustering and ordering of information, it remains an active and vibrant field of re-

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

search, albeit with less focus on its traditional roots of machine learning and AI. In fact, there are several annual conferences and workshops devoted to the field, as well as an FCA mailing list, FCA related software, etc. (See, for example, `http://www.upriss.org.uk/fca/fca.html`.) Ganter and Wille's foundational text on the topic can be found in [8], and a subsequent text book by Carpineto and Romano is available in [2].

For several years now, the Fastar research group has regarded FCA as a useful and interesting complement to its research in stringology. Section 3 will show how we have connected the themes of FCA and stringology in recent years. Our closing reflections on the matter are given in Section 4.

2 Formal Concept Analysis

FCA roots the notion of a concept in a given domain of discourse in a two-dimensional matrix that is called the context. The context consists of a finite number of rows representing objects in the domain of discourse, and a finite number of columns representing the discrete attributes of those objects. A fictitious example of a context representing people on the beach at Ipanema is shown in Figure 1. The cells of the matrix indicate whether or not an object—one of the named people on the beach—have a given attribute. Thus, either Alice or Amy could be the tall and tanned and young and lovely girl mentioned in the well-known song, Alice being additionally characterised as bright and Amy as big. The boys on the beach, Bob and Bill, also have their attributes noted in the context.

	Tall	Tanned	Young	Lovely	Big	Fat	Bright
Alice	×	×	×	×			×
Amy	×	×	×	×	×		
Bob	×		×				×
Bill		×			×	×	

Figure 1: The Ipanema Context

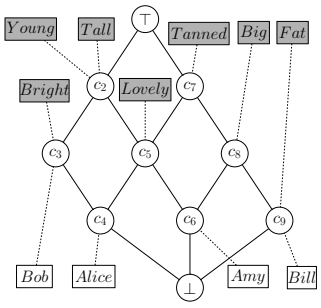

Figure 2: Line diagram of Ipanema Concept Lattice

The semantics given to the objects and attributes should be noted. To infer that the first two objects in the context are girls while the latter two are boys goes beyond the data given. If gender was an important attribute, an additional column could be inserted called, say, Female, in which the first two rows receive a cross, and the last two are empty. Alternatively, two additional columns, called Male and Female respectively, could be inserted but this would be somewhat redundant because of the binary nature both of gender and of the attributes—i.e. an object is considered to either have or not have an attribute. However, attributes sometimes fall into non-binary classes such that an object can have only one attribute in the class. For example, there might be a class of attributes—say Black, Red, Brunette, Grey—indicating the hair colour of objects. Non-discrete attributes can also be partitioned into mutually exclusive

discrete attributes. For example there could be Short, Medium and Tall attributes characterising objects less than 1.5 meters in height, those between 1.5 and 1.8 meters, and those over 1.8 meters.

The context embeds information that is not immediately evident. It has already been pointed out that the objects Alice and Amy share all and only the attributes Tall and Tanned and Young and Lovely. It is also the case that these attributes do not collectively characterise any objects other than Amy and Alice. FCA therefore views the pair consisting of the set of objects and set of attributes involved in such an exclusive relation as a *concept*. Thus, the context embeds a concept that is designated c_5 (to conform with the line diagram entry to be discussed below) that may be describe as follows:

$$c_5 = \langle \{\text{Alice, Amy}\}, \{\text{Tall, Tanned, Young, Lovely}\} \rangle$$

The object set {Alice, Amy} is called the *extent* of c_5 and is denoted by $\mathbf{extent}(c_5)$. Similarly, the attribute set {Tall,Tanned,Young,Lovely} is the concept's *intent*, denoted by $\mathbf{intent}(c_5)$.

The context embeds many more such concepts, two further examples being c_2 and c_8, respectively defined as:

$$c_2 = \langle \{\text{Alice, Amy, Bob}\}, \{\text{Tall, Young}\} \rangle$$
$$c_8 = \langle \{\text{Amy, Bill}\}, \{\text{Tanned, Big}\} \rangle$$

Furthermore, FCA regards the concepts as being partially ordered by set inclusion on the extents, i.e. for arbitrary concepts c_i and c_j,

$$c_i \leq c_j \equiv \mathbf{extent}(c_i) \subseteq \mathbf{extent}(c_j)$$

Thus, in the Ipanema beach example, $c_5 \leq c_2$ while c_5 and c_8 are incommensurate.

The full set of concepts that can be induced from a given context are not only partially ordered. They also constitute a complete lattice—i.e. every subset of concepts has a least upper bound and a greatest lower bound. This means that there will always be a top concept (denoted by $\top$) and bottom concept (denoted by $\bot$) in such a concept lattice.

Many algorithms have been developed to infer the concept lattice derivable from a given context, to record the ordering of concepts and/or to display the concepts and orderings in a line diagram. Figure 2 shows the line diagram of concepts derived from the Ipanema context of Table 1. The Concept Explorer package[1] was used to produce this diagram. Objects and attributes are shown in rectangles connected by dashed lines to various concept nodes, attribute rectangles being shaded. Each of the thirteen nodes in this graph represents a concept, and these are conventionally ordered with the smallest concepts placed at the bottom of the diagram, and the largest concepts at the top.

The extent of any concept may be found by collecting together all objects reachable on downward paths from the concept. A concept's intent is found by collecting together

[1] Note: Concept Explorer's author requests that users cite his Russian text, [16], as a reference to the package.

all attributes reachable on upward paths from the concept. In Figure 2 the top and bottom concepts are respectively:

$\top = \langle \{\text{Alice, Amy, Bob, Bill}\}, \varnothing \rangle$
$\bot = \langle \varnothing, \{\text{Tall, Tanned, Young, Lovely, Big, Fat, Bright}\} \rangle$

An object is called the *own object* of a concept C when that concept is the smallest that includes the object in its extent, and the set of C's own objects is denoted by **ownobj**(C). The own object sets of all concepts discussed thusfar in the Ipanema example are empty; for example, **ownobj**$(c_1) = \varnothing$, **ownobj**$(\bot) = \varnothing$, etc. However, the concept marked c_4 in Figure 2 is an example of one which has a non-empty own object set; i.e. **ownobj**$(c_4) = \{\text{Alice}\}$. In general, Concept Explorer provides links from concepts to their own objects.

There is a clear duality between the role of attributes and objects which will not be spelled out in detail here. Thus, for example, Concept Explorer also links concepts to their so-called own attributes—defined similarly but dually to own objects.

It is a property of a concept lattice that it is set-intersection closed with respect to both the intents and to the extents of concepts—i.e. the intersection of the intents of any two concepts will be an intent of some lattice concept, and dually for extents. Furthermore, the larger a concept, the larger its extent and the smaller its intent. This is clearly seen in the two extremes where **extent**$(\top)$ in the example lattice is the full set of objects, while **extent**$(\bot)$ is $\varnothing$, and dually in respect of their extents. This parallels our instinctive notion of abstraction (generalisation): the more abstract an object, the less specific its properties (thus less attributes in its intent), and the more entities (objects in its extent) it represents.

FCA concept lattices are therefore information-rich—arguably the most information rich representation possible of the relationship between objects and their attributes. However, this richness comes at a cost: the number of concepts in a concept lattice is, in the worst case, exponentially dependent on the number of objects and attributes. Specifically, if N is the minimum over the number of objects and the number of attributes, then the maximum number of concepts is 2^N. Suppose that N is the number of attributes. Then this worst-case scenario arises when there is a set of N objects, each of which is characterised by exactly $N-1$ attributes, and each of which differs from all the other N objects in that set by exactly one attribute. As the number of entries in the context matrix declines, the number of concepts in the lattice falls to more tractable levels.

3 Applications in Stringology

For some years now, members of the Fastar Research Group have been conducting research into the use of FCA in several areas, including stringology. This section will review some of the work carried out to date. Note that the details will inevitably be sketchy; original sources should be referenced for fuller information.

3.1 Taxonomies

One of the longstanding interests of the research group has been the taxonomisation of algorithms in general. When known algorithms are developed in a uniform correctness-by-construction style, their common and differing features tend to be naturally highlighted, thus supporting the placement of such algorithms in a taxonomic format. Such a taxonomy has several advantages: it serves as a didactic instrument, it supports the development of a coherent toolkit based on a class inheritance hierarchy corresponding to the taxonomy, and it tends to expose knowledge gaps where alternative algorithms could be developed in the domain of discourse.

The so-called Taxonomy Based Software Construction (TABASCO) approach to domain engineering was pioneered by Watson, who first applied it to regular language algorithms [14]. It has since been applied in a number of other domains such as algorithms in respect of tree automata [3] and algorithms for constructing minimal acyclic deterministic finite automata [15]. As an example, Figure 3 shows the taxonomy of exact keyword pattern matching algorithms derived in [14] and then extended in [6,5]. The top node represents the high-level algorithm for solving this problem (which only specifies that the set of all matches is assigned to the output variable), while algorithms further down in the taxonomy present refinements, with the algorithms close to or at the taxonomy graph leaves representing concrete algorithms. The various branch labels indicate specific attributes of the algorithms that subsequently occur on the paths from those branches. For example the branch labeled P means that the matches are found by considering prefixes of the text in some order; + indicates that the prefixes are considered in increasing length order; E that matches are registered by their endpoint; etc.

In [4] it was shown that if the attributes are known *a priori*, then FCA could have been used to derive this taxonomy, albeit with some slight modifications. In this case the set of FCA objects is taken to be algorithms (either abstract or concrete) that possess a subset of the identified attributes. The resulting line diagram is given in Figure 4. Algorithm names (i.e. the FCA objects) have been omitted to avoid clutter—indeed, many of the abstract algorithms do not even possess formal names. The essential differences between the two diagrams in the figure will not be enumerated here since this is done in the original article. Suffice it to note that the modifications brought about by the concept lattice naturally highlight certain algorithm commonalities that were obscured in the original taxonomy. However, the orginal article goes on to recommend that, because the identification of appropriate algorithm attributes is generally quite challenging, the investigation of an FCA technique called attribute exploration for this purpose should be considered. This is an interactive technique whereby a given FCA context is analysed and the user is systematically prompted for exemplars of objects that might have various combinations of attributes. [4] suggests that this could assist in building taxonomies such as those in Figure 4.

In a somewhat different exercise, Liang [10] provides a unified treatment of various spell checkers and correctors. Part of the study entailed the use of FCA to classify ten different spell checkers in terms of twelve attributes. The study mentions examples of

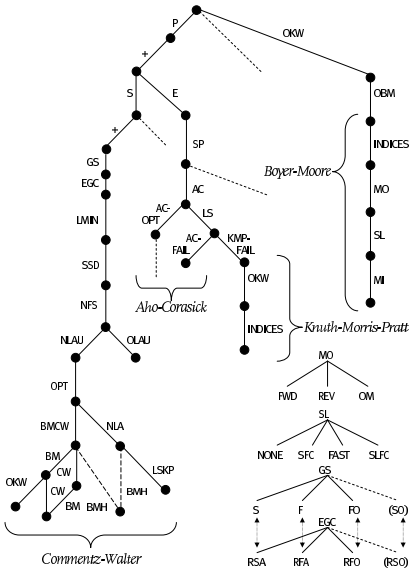

Figure 3: Pattern Matching Taxonomy

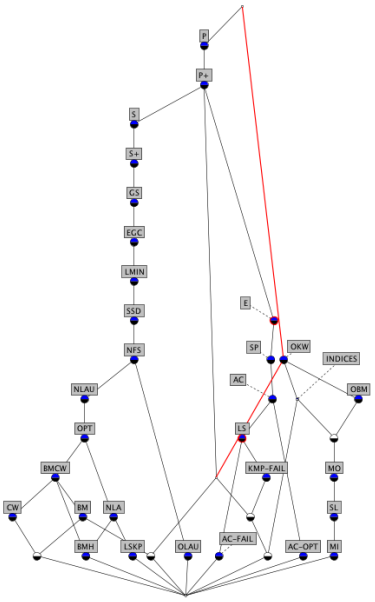

Figure 4: Concept Lattice Taxonomy

implication rules that can be inferred from the intents and extents of concepts[2] and notes that the rules

> . . . provide an interesting starting point to investigate whether they are indeed universally true, or whether algorithms can be found that negate them. Indeed, if no existing algorithm can be found, this might signal a challenge to explore the possibilities of actually developing one.

As shown above, FCA can be used in the classification of stringological algorithms or algorithms in a more general sense. In the next subsections, we show how it can also be used in stringological algorithms themselves.

3.2 2D Pattern Matching

Pattern matching lies at the heart of stringology. We have investigated several ways in which FCA can be used in a pattern matching context.

In [12,13] a new approach to two-dimensional pattern matching has been proposed. The target domain is assumed to consist of 2D rectilinear shapes, such as is typically the case with components on a chip. Satellite images could also be approximated in this fashion. An encoding scheme was worked out whereby any rectilinear such shape—i.e. pattern—can be described in terms of a finite set of attributes. Figure 5 shows a purpose built editor to describe and store such patterns in terms of these attributes. The set of patterns to be sought are then regarded as FCA objects described in terms of these attributes and stored in a context. Such a context then in turn yields a concept lattice in which the intent of each concept is a set of attributes held in common by a subset of the pattern objects that are sought. Figure 6 gives

[2] How such rules are derived from a concept lattice is part of FCA theory. Concept Explorer provides an option for generating the rules.

a partial view of an example line diagram used in a 2D search. An algorithm was developed which systematically traverse the search space and as attributes are encountered, a marker is appropriately moved from one concept to the next in the line diagram. It is shown that a hit (i.e. a 2D pattern match) can be inferred when certain concepts in the lattice are encountered.

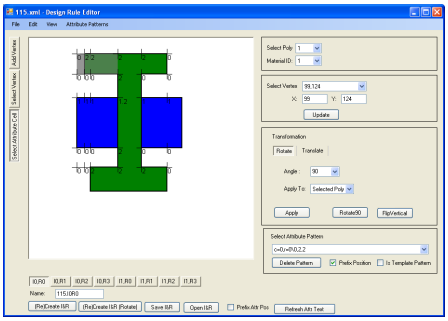

Figure 5: 2D Pattern Matching Editor

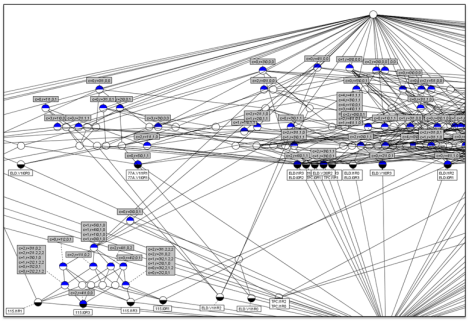

Figure 6: 2D Pattern Matching Lattice

3.3 Multiple Keyword Pattern Matching

We have also been experimenting with a concept lattice based approach to conventional multiple keyword pattern matching. The approach views a character string as an FCA object whose attributes are pairs of the form $\langle n, s\rangle$, where n is the position of the character s in the string, i.e. the attributes are position-encodings of the characters in the string. Using such a scheme, a set of patterns can be described as objects in a context such as shown in Figure 7. Thus, the string "abc" has attributes $\langle$1,a$\rangle$, $\langle$2,b$\rangle$ and $\langle$3,c$\rangle$, etc. Once again, such a context yields a concept lattice, which we call a Position Encoded Pattern Lattice (PEPL), whose line diagram appears in Figure 8.

	$\langle 1,a\rangle$	$\langle 2,a\rangle$	$\langle 2,b\rangle$	$\langle 3,b\rangle$	$\langle 3,c\rangle$	$\langle 4,c\rangle$
abc	×		×		×	
aabc	×	×		×		
abcc	×		×		×	×

Figure 7: Position encoded context for $P = \{abc, aabc, abcc\}$

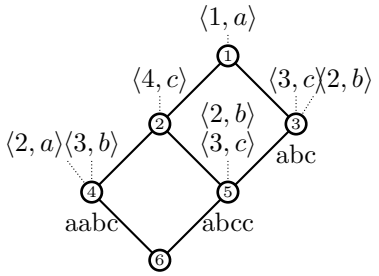

Figure 8: Line diagram of PEPL for $P = \{abc, aabc, abcc\}$

Algorithm 1 described below is a somewhat naive algorithm that uses a PEPL to do multiple keyword pattern matching. Its top level procedure is called *PMatch*, which takes as input a PEPL $\mathfrak{P}$ and a text, s. It then finds in s all match occurrences of words in P, recording them in *MO*. The definition of *PMatch* assumes constant *minlength(P)* as the length of the shortest keyword in P. To avoid notational clutter, $\mathfrak{P}$, s and *MO* are assumed to be globally accessible to all procedures. A special symbol, *nil*, is used to designate a non-existent concept, specifically the parent of the top concept.

PMatch calls *matchIntent* for each character in s where a match could possibly start (i.e. the tail is ignored). The condition in the associated for-loop, $t \in [0, |s| - j + 1)$, is intended to signify that these probes are from left to right. In each call the intent of the top of the lattice and its non-existent parent, *nil*, are used as parameters.

matchIntent takes a string position t, and two concepts, c and p, as parameters. It is assumed that c is a child of p and the set difference, Δ, between their intents is computed. The special case of $\top$, which has no parent, is catered for. A loop checks whether all the attributes in Δ indicate positional matches in the text s as offset by the current search position, t—i.e. the loop removes from Δ all attributes of the form $\langle i, \alpha \rangle$ such that $s[t + i - 1] = \alpha$. If this reduces Δ to the empty set, then a match occurrence is considered to have been found for each own object at c. Moreover, $match(c, t)$ can be called to investigate whether further match occurrences at t can be inferred by considering c's children. $match(p, t)$, in turn, simply sweeps through the children of p, recursively invoking *matchIntent* in each case.

Algorithm 1 *PEPL Based Matching*

proc $PMatch(\mathfrak{P}, s)$
  $MO, j := \varnothing, minlength(P)$;
  { *Traverse target string s from left to right* }
  for $(t \in [0, |s| - j + 1)) \rightarrow$
    $matchIntent(t, \top, nil)$
  rof
corp{ ***post*** : *MO is the set of match occurrences of P in s* }

proc $matchIntent(t, c, p)$
  if $(p = nil) \rightarrow \Delta := \mathbf{intent}(c)$
  [] $(p \neq nil) \rightarrow \Delta := \mathbf{intent}(c) \setminus \mathbf{intent}(p)$
  fi;
  do $(\exists \langle i, \alpha \rangle : \langle i, \alpha \rangle \in \Delta : (s[t + i - 1] = \alpha)) \rightarrow$
    $\Delta := \Delta \setminus \{\langle i, \alpha \rangle\}$
  od;
  if $(\Delta = \varnothing) \rightarrow MO := MO \cup \mathbf{ownobj}(c) \times \{t\}$;
      $match(c, t)$
    $(\Delta \neq \varnothing) \rightarrow$ **skip**
  fi
corp

proc $match(p, t)$
  $T := children(p)$; { T *is the (possibly empty) set of children remaining* }
  do $(T \neq \varnothing) \rightarrow$
    let $c \in T$;
    $T := T \setminus \{c\}$;
    $matchIntent(t, c, p)$
  od
corp

To illustrate how Algorithm 1 works, consider the keywords to match $P = \{abc, aabc, abcc\}$ and the target $s = aaabcdabccd$. The formal context was given in Figure 7 and the

line diagram of the associated PEPL is in Figure 8. For convenience, the intents and own object sets of each concept are made explicit in Table 1. Table 2 provides a trace

Id of c	**intent**(c)	**ownobj**(c)
$1 = \top$	$\{\langle 1, a\rangle\}$	
2	$\{\langle 1, a\rangle, \langle 4, c\rangle\}$	
3	$\{\langle 1, a\rangle, \langle 3, c\rangle, \langle 2, b\rangle\}$	abc
4	$\{\langle 1, a\rangle, \langle 4, c\rangle, \langle 3, b\rangle, \langle 2, a\rangle\}$	aabc
5	$\{\langle 1, a\rangle, \langle 4, c\rangle, \langle 3, c\rangle, \langle 2, b\rangle\}$	abcc
$6 = \bot$	$\{\langle 1, a\rangle, \langle 4, c\rangle, \langle 3, b\rangle, \langle 3, c\rangle, \langle 2, b\rangle, \langle 2, a\rangle\}$	

Table 1: Details of concepts of the PEPL in Fig. 7

summary of calls to *matchIntent*. The first column shows t, the offset into s from which matching positions are calculated. The second and third columns show the lattice concept visited and its concept participating in the call to *matchIntent*. The fourth column marked Δ gives the set difference between the intent of the child and parent concept. A column per symbol in the string *aaabcdabccd* then follows. The last column gives the own object set to be to be used to update *MO* when a match has been found. Note that since $minlength(P) = 3$ and $|s| = 11$, the trace ranges over $t \in [0, 9)$. Each row is a matching step of the algorithm—i.e. every row represents

t	c	p	Δ	a	a	a	b	c	d	a	b	c	c	d	**ownobj**(c)
0	1	nil	$\{\langle 1,a\rangle\}$	T											$\varnothing$
0	2	1	$\{\langle 4,c\rangle\}$				F								
0	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$		F										
1	1	nil	$\{\langle 1,a\rangle\}$		T										$\varnothing$
1	2	1	$\{\langle 4,c\rangle\}$					T							$\varnothing$
1	4	2	$\{\langle 2,a\rangle, \langle 3,b\rangle\}$			T	T								$\{aabc\}$
1	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$			F									
2	1	nil	$\{\langle 1,a\rangle\}$			T									
2	2	1	$\{\langle 4,c\rangle\}$						F						
2	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$				T	T							$\{abc\}$
2	5	3	$\{\langle 4,c\rangle\}$						F						
3	1	nil	$\{\langle 1,a\rangle\}$				F								
4	1	nil	$\{\langle 1,a\rangle\}$					F							
5	1	nil	$\{\langle 1,a\rangle\}$						F						
6	1	nil	$\{\langle 1,a\rangle\}$							T					$\varnothing$
6	2	1	$\{\langle 4,c\rangle\}$										T		$\varnothing$
6	4	2	$\{\langle 2,a\rangle, \langle 3,b\rangle\}$								F				
6	5	2	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$								T	T			$\{abcc\}$
6	3	1	$\{\langle 2,b\rangle, \langle 3,c\rangle\}$								T	T			$\{abc\}$
7	1	nil	$\{\langle 1,a\rangle\}$								F				
8	1	nil	$\{\langle 1,a\rangle\}$									F			

Table 2: Algorithm 1 trace: matching $\{abc, aabc, abcc\}$ in *aaabcdabccd*

a call of the function *matchIntent* . As an example, the first row indicates that the matching position $t = 0$ and the attribute set to match is $\Delta = \{\langle 1,a\rangle\}$. The first (and only) element of the set is $\langle i,\alpha\rangle = \langle 1,a\rangle$. This means that position $t + i - 1 = 1$ is checked for the symbol $\alpha = a$, which is indeed the case as indicated by the "T" (for the boolean value *true*) shown in the first column for the target string. All "T" entries in the table indicate that attributes in Δ have been successfully matched in the do-loop of *matchIntent*. Once Δ has been reduced to $\varnothing$, *MO* has to be updated. Of course, if the concept has no own object—as is the case for the top concept marked 1—then nothing is added to *MO* (i.e. $\mathbf{ownobj}(c) \times \{t\} = \varnothing$). Subsequent calls to *match* without updating t, recursively deal with children concepts of the one currently under test. The second row of the table therefore logs the results the call to *matchIntent*

made via *match* in respect of concept 2, the leftmost child of concept 1. In this case, the intent difference set is $\Delta = \{\langle 4,c\rangle\}$, and since $\nexists\langle i, \alpha\rangle : \{\langle 4,c\rangle\} : (s[t+i-1] = \alpha)$, (or, more explicitly, $s[0+4-1] = b$ and not c) *matchIntent* cannot reduce Δ to $\varnothing$. This is indicated by "F" (for false) as an entry in the relevant column of the table. Control now returns to *match*, where the next child of concept 1, namely concept 3, is considered. Further rows of the table illustrate the execution steps of Algorithm 1 for the rest of the target string.

PMatch eliminates sets of words from P that do not match in s without ever backing up in s, i.e. t is monotonically increasing. In this sense *PMatch* is an *online* algorithm, similar to the Aho-Corasick algorithm [1]. However, *PMatch* sometimes *revisits* symbols in s. Such revisits are reflected by the multiple entries in various columns representing symbols in *aaabcdabccd* in Table 2. Investigations are currently underway to find a PEPL-based algorithm that avoids such revisits and thus becomes competitive with the Aho-Corasick algorithm in terms of space and time efficiency. To date, an algorithm is available which can be proven to achieve this goal if P complies with the following condition:

$$\forall(v, a\cdot w) : (v, a\cdot w) \in P \times P : (\forall x, y : (v = x\cdot a\cdot y) : (x \neq \varepsilon) \Rightarrow (y = w\cdot u))$$

3.4 Failure DFAs

Taking the cue from the way in which failure arcs are defined and used in one of the variants of the Aho-Corasick algorithm, we have generalised the notion of a failure deterministic finite automaton (FDFA). We have shown how FCA can be used to derive from a given deterministic finite automaton (DFA) a language-equivalent FDFA [9].

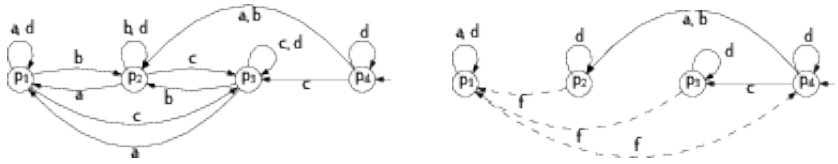

Figure 9: Initial DFA and an equivalent FDFA

As a brief survey of the approach, consider the DFA on the left hand side of Figure 9. The structure on the right hand side of Figure 9 is an example of an FDFA. The FDFA has three failure transitions (represented by dashed arcs) and labelled $\mathfrak{f}$. The semantics of these failure transitions differ from those of normal DFA transitions. In the latter case, if an arc is labelled by the symbol at the head of the string under test, then that symbol is consumed and a transition is made to the arc's destination. In the former case, if the head of the string under test does *not* match the labels on any of the current state's outgoing DFA arcs, then the failure transition is made to the next state, *without consuming* the head of the string under test. Thus, to recognise the string *abca*, the following DFA transitions are made in Figure 9

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

However, in the case of the FDFA in Figure 9 the transitions made are as follows

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

It can easily be verified that the FDFA recognises the same language as the DFA, and is in this sense equivalent to it. Note that the FDFA has only eight arcs, and three failure transitions (represented by dashed arcs) and labelled $\mathfrak{f}$, while the DFA has sixteen arcs. It is a potential savings in arc space such as this which motivates the introduction of failure transitions. The trade-off to be made is that an additional time cost to consume the entire string is incurred.

However, it is not immediately evident how to build an FDFA that is language-equivalent to a given DFA. The problem is to determine where to position failure transitions and which DFA transitions to remove. It turns out that one can rely on FCA to assist with these decisions. The approach starts with a so-called state / out-transition context in which DFA states are treated as objects and arc-label / arc-destination pairs are treated as the attributes of the objects as shown in Figure 10. This then leads to a state / out-transition lattice as shown in Figure 11.

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

Figure 10: The state/out-transition context of DFA in Figure 9

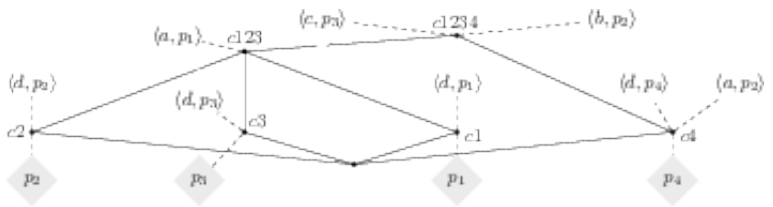

Figure 11: State/out-transition formal concept lattice of DFA in Figure 9

The algorithm described in [9] relies on a so-called arc redundancy metric computed for each concept in the lattice. By "greedily" selecting concepts with the highest arc redundancy, sets of DFA transitions can be selected which are replaced by a single failure transition, thereby building a sequence of FDFAs each of which is language-equivalent to its predecessor. The algorithm terminates when it is no longer possible to reduce any more DFA transitions from the structure obtained to date. Although this algorithm does not guarantee optimality (in the sense of maximally reducing DFA transitions to FDFA failure transitions), empirical data obtained to date suggests that the number of DFA arcs on randomly generated DFAs can be reduced by between 5% and 15%. Because the FDFA produced by this algorithm has the same set of states (and therefore topological layout of the state diagram) as the starting DFA, we have called it the DFA-homomorphic algorithm. Refinements to this algorithm are currently under investigation. An alternative algorithm is also being investigated that adds new FDFA states in accordance with concepts in the state/out-transition lattice. Because of this state correspondence with the lattice rather than the original DFA, we have designated this a lattice-homomorphic algorithm.

3.5 NFA to DFA conversion

As a final FCA-based FCA application in the stringology space, an algorithm has been developed for converting a nondeterministic finite automata (NFA) to an equivalent DFA. The standard *subsect construction* algorithm (sometimes known as the 'powerset construction') constructs a DFA whose states are *sets* of states from the input NFA. Those sets of states are not necessarily disjoint, and constructing the DFA's transitions can entail recomputing collective transitions for common (sub)sets of NFA transitions. The new algorithm involves maintaining a lattice of the DFA states (= sets of NFA states) and avoiding such recomputation.

4 Closing Reflections

Our work to date has demonstrated the theoretical viability of leveraging FCA technology in a variety ways to look anew at many stringology problems. Indeed, wherever information needs to be clustered together in some ordered fashion, FCA could have a potential role to play. Concerns about the practical viability of these strategies could well be raised, particularly in the light of the state space explosion problem associated with FCA lattices. Three observations deserve mention in this regard.

- Constrained lattice: In many of the applications explored to date, the problem space explicitly prevents worst case lattice sizes from being generated. For example, the number of attributes in the context used for the FDFA application is maximally $|\Sigma| \times |Q|$ where Q is the set of DFA states and Σ is the alphabet. Each attribute represents a symbol / state destination pair. The theoretical worst case lattice size would be when every object (state) has exactly $(|\Sigma| \times |Q|) - 1$ attributes. However, this is not feasible in the problem domain being considered—there can only be $|Q|$ out-transitions for a DFA! This ameliorates the state space explosion problem.

- Incremental Application: It should also be noted that in some cases it is not necessary to build the entire lattice, because one is not seeking absolute optimality, but merely an improvement. Again, considering the FDFA construction example, it is not necessary that the entire DFA state space be considered for transformation to an FDFA—one may judiciously select areas of the DFA state space that are most likely to be profitably changed. Precisely how this should be done is a matter of future research.

- Inherently Small Problems: Finally, we observe that many of the problems inherently involve only a limited number of objects. For example, the building of a taxonomy of domain-specific algorithms is unlikely to involve more than 10 or 20 algorithms.

One avenue of investigation that suggests itself is to explore the use of FCA in the context of arbology based on the use of string pushdown automata and string encodings of trees—a domain of research pioneered at the Czech Technical University in Prague by Melichar [11].

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. C. Carpineto and G. Romano: *Concept Data Analysis: Theory and Applications*, John Wiley & Sons, Ltd, 2004.
3. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Eindhoven University of Technology, the Netherlands, Apr. 2008.
4. L. Cleophas, B. W. Watson, D. G. Kourie, A. Boake, and S. Obiedkov: *TABASCO: Using concept-based taxonomies in domain engineering.* South African Computer Journal, (37) Dec. 2006, pp. 30–40.
5. L. Cleophas, B. W. Watson, and G. Zwaan: *A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms.* Science of Computer Programming, 75 2010, pp. 1095–1112.
6. L. G. Cleophas: *Towards SPARE Time: A new taxonomy of keyword pattern matching algorithms*, Master's thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 2003.
7. S. Ferré and S. Rudolphs, eds., *Proceedings of the Seventh International Conference on Formal Concept Analysis*, Darmstadt, Germany, May 2009, Springer.
8. B. Ganter and R. Wille: *Formal Concept Analysis: Mathematical Foundations*, Springer, Berlin, 1999.
9. D. G. Kourie, B. W. Watson, L. Cleophas, and F. Venter: *Failure deterministic finite automata*, in Proceedings of the Sixteenth Prague Stringologic Conference, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, Aug. 2012.
10. H. L. Liang: *Spell checkers and correctors: A unified treatment*, Master's thesis, Department of Computer Science, University of Pretoria, South Africa, 2008.
11. B. Melichar: *Arbology: Trees and pushdown automata*, in LATA, A. H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 32–49.
12. F. Venter, D. G. Kourie, and B. W. Watson: *FCA-based two dimensional pattern matching*, in Ferré and Rudolphs [7].
13. F. Venter, B. W. Watson, and D. G. Kourie: *Pattern matching in structured multi-sensor/layered image big-data*, in Proceedings of the 33rd Canadian Symposium on Remote Sensing (Abstracts), Ontario, Canada, June 2012.
14. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, Sept. 1995.
15. B. W. Watson: *Constructing minimal acyclic deterministic finite automata*, PhD thesis, Dep. Comput. Sci., University of Pretoria, 2010.
16. S. A. Yevtushenko: *System of data analysis "Concept Explorer"*, in Proceedings of the 7th National Conference on Artificial Intelligence KII-2000, Russia, 2000.

Brtkovica – the Foundation of Arbology

Boba Mannová

Department of Computer Science and Engineering, Faculty of Electrical Engineering
Czech Technical University in Prague, Karlovo naměstí 13, 121 35 Prague 2, Czech Republic
mannova@fel.cvut.cz

This paper is dedicated to 70th Jubilee of Bořivoj Melichar and 50th Anniversary of the project Brtkovica.

Abstract. The paper tries to identify the roots of Arbology discipline, which was founded by three members of Prague Stringology Group: Bořivoj Melichar (Bořek), Jan Janoušek and Tomáš Flouri and which was officially first introduced as a new algorithmic discipline at London Stringology Days conference in February 2009. The foundation of Arbology was put down in fact much earlier during the research in frame of the project Brtkovica, which was funded by CTU FEE in 1962. Because Bořivoj Melichar was head of the research group and I was member of the team of 16 project researchers, I would like to present here some results of the ancient research and some basic algorithms used at that time. Even the logo of Arbology group will be analyzed here from perspectives of that research.

Keywords: arbology, Brtkovica, tree, tree destruction algorithms, resources of arbology

1 Introduction

The project Brtkovica was started in 1962 by team of students from Faculty of Electrical Engineering. Head of the student team was Bořek Melichar and the most important members were Bedřich Bín, Michal Matějíček and Jan Crha. The name of the project came from a location of centre situated at Slovakia mountains Low Tatras in building called Brtkovica where the research in the Arbology field initiated. This was in time when there was not common to start research with construction of research centre in intelligent buildings at universities. If you look into Table 1, *History of Arbology in the context of history of computers*, it was 25 years before the Internet, nine years before programming language Pascal and 19 years before the first PC computer.

The world without Internet, without e-mails and without PCs, it was at Brtkovica almost the same environment as in Prague at CTU at that time. Researchers traveled to research building in February 1962. In this year there was very cold winter and much snow. Brtkovica is situated 1 175 m above sea-level. The researchers had to go there from the level of 700 m, where a train station is situated. For illustration, there is altitude profile of the trip in Figure 1. If we imagine almost two meters of snow and 30 kilograms of research reserve per person (rice, cookies, pasta, sugar, java coffee, rum etc.) it was not easy to get there. But despite of these conditions, a research started right at the beginning of the trip.

2 First stages of research

We realized that there are many trees around Brtkovica and sometimes we had to climb them during the trip for water. Bořek decided that we have to use some good already known algorithms for this, but also we have to look for some new one.

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

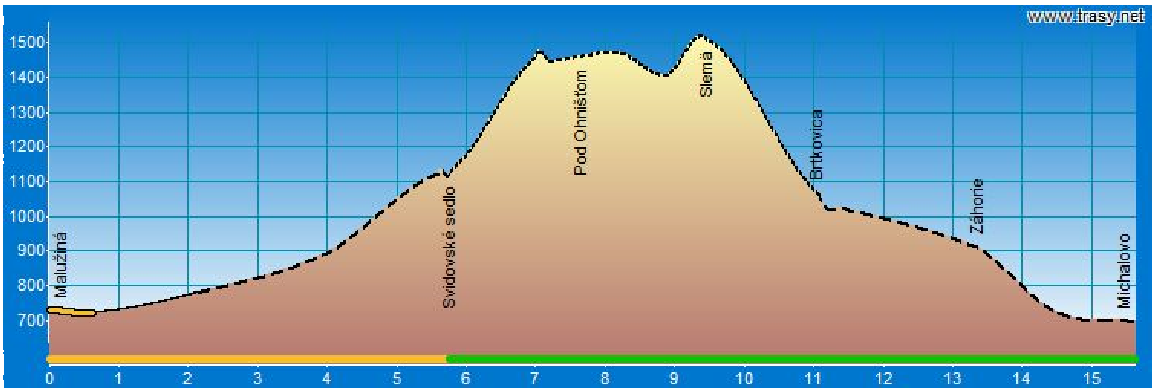

Figure 1. Altitude profile of Brtkovica [1]

Brtkovica was a log cabin build for Slovak partisans during Slovak uprising in 1944. Because it was too small for all researchers to stay during night, we build special tent for two and they stayed there overnight on investigation of the most complicated problems. We found that brain in minus 30 degree of Celsius with some

Year	History of Computers	World event
5000 BC	Abacus	
1854	Boole defined Boolean Logic	
1890	Hollerith introduced the first electromagnetic punched cards machine	The first public power plant started being built in the UK
1936	Alan Turing published paper with definition of algorithm and description of Turing Machine	
1940		First colour TV broadcast
1941	Konrad Zuse finished his Z3 computer	
1942	***Bořivoj Melichar was born***	
1944	Computer Mark I	
1945	John von Neumann described his model of computer	
1947		First transistor produced in Bell laboratories
1952	Grace Hooper described first compiler	
1954	First assembler	
1957	FORTRAN	
1958	Algol	
1960	Cobol	
1962	***Project Brtkovica started***	Founded the first computer science department at Purdue
1964	IBM 360 PL/1 and Basic	Department of Computer Science has been established at CTU, Faculty of Electrical Engineering
1968	Computer mouse	Russians came to Prague
1969	Unix	
1971	Pascal and Prolog	
1981	First PC computer	
1987	Internet	
2009	Arbology presented	

Table 1. History of Arbology in the context of history of computers

external support is very productive, so in the morning those from the tent went to the forest to look for the best trees. Very soon they found that trees around are very well designed and they may be used for systematic approach to the construction of algorithms on trees. They started to use a linear notation of trees instead of finite automaton, which was till that time the most used model by Bořek Melichar. Because they do not have to build any trees as there were many of them around, they had to destruct trees to set a goal to bring part of trees into the centre Brtkovica to support research going on there.

For destruction algorithms with good complexity they used tools as an axe, a saw and sometime a rum. All these were easy to find in the forest.

3 Brtkovica tree theory

The theory of formal tree languages [3] has been extensively studied and developed since the 1950s and 1960s, respectively. This theory describes fundamentals for some computer applications. The elements of tree languages are trees.

Every connected graph having n nodes and $n-1$ edges is a tree. We used for our research model of Brtkovica tree as is shown in Figure 2. Each branch of the tree is represented by one edge. End of a branch is node and attachment of the branch is another node. For tree destruction algorithm we had to cut off all branches. It means that we had to disconnect all edges of the tree.

Indeed, this is what we did with all trees to be destructed in Brtkovica.

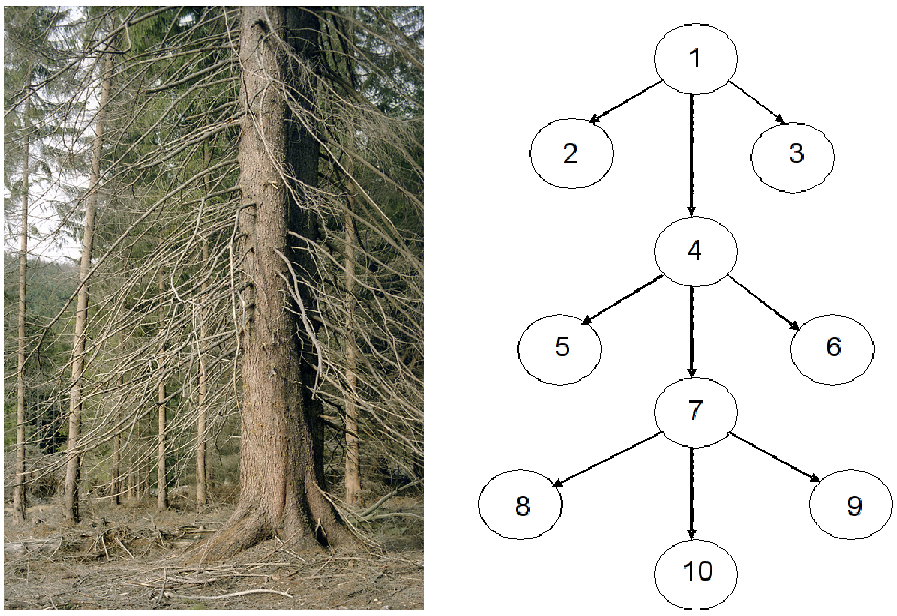

Figure 2. Tree from Brtkovica [5] and the model of this tree having 10 nodes and 9 edges

For a description of destructive algorithm developed within Brtkovica project, we will use linear notation of a tree. A directed rooted labeled ranked tree is used as a source for this notation [4], and an example is shown in Figure 3.

If we put disconnection of all edges as target of destruction algorithm, we can see that we have to do it step by step. If you look into the linear notation it means that first are disconnected nodes a_0, then a_1 and finally a_2. We can use also a recursive

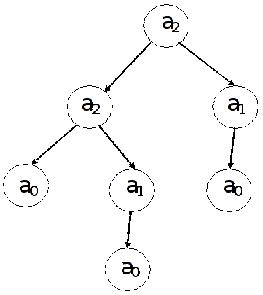

Figure 3. A directed rooted labeled ranked tree, in linear (prefix) notation $a_2a_2a_0a_1a_0a_1a_0$

algorithm, where we define for each node its proper sub-trees. Recursion is finished when a sub-tree consists of only one node.

4 Results

This basic research continued for two weeks. After that it was prolonged by Ministry of Education by another week due to more snow, low temperature in February 1962 and a lack of coal for schools heating. All research aimed to find a systematic approach to the destruction algorithms for processing trees.

After 16 years Bořivoj Melichar makes a conclusion of this research by introducing Arbology.

In [2], the description of Arbology as a new algorithmic discipline was given. Arbology aims to represent a unified and systematic approach to the construction of algorithms on trees, we will see that this was already researched in Brtkovica project. Many algorithms for operations on trees were created in applications at that time, however they were ad-hoc algorithms and not constructed from a systematic theory point of view.

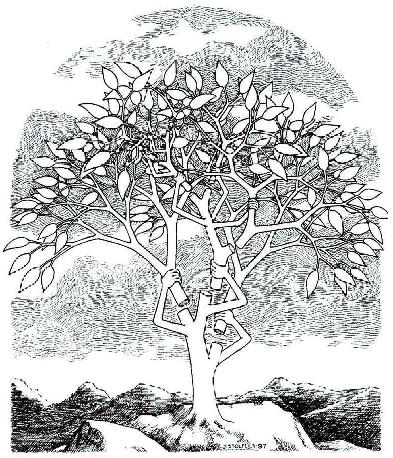

Figure 4. Arbology logo

5 Conclusion

What have outlasted from that 1962 obsolete research in Arbology? It was in fact the foundation of Arbology. You may see proof for this statement from the Arbology logo in Figure 4. Looking at the picture, one will find there a construction of already destructed tree. In the background, there are the mountains of Tatras as had been sawn by Bořivoj Melichar from the tent in Brtkovica in 1962.

References

1. *Altitude profile of Brtkovica*: Aug. 2012, `http://www.testar.cz/Tatry%202011/Ohni%C5%A1t%C4%9B%20profil.htm`.
2. J. Janoušek: *Arbology*, 2011, lecture at Department of Theoretical Computer Science, Faculty of Information technology, CTU in Prague.
3. J. Janoušek and B. Melichar: *On regular tree languages and deterministic pushdown automata*. Acta Inf., 46(7) Oct. 2009, pp. 533–547.
4. B. Melichar: *Arbology: Trees and pushdown automata*, in 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 Berlin / Heidelberg, 2010, pp. 32–49.
5. J. Moravec: *Smrk*, Aug. 2012, `http://www.etf.cuni.cz/~moravec/fotky/jpeg/n22833-v.jpg`.

A simple method for computing all subtree repeats in unordered trees in linear time

Tomáš Flouri[1], Kassian Kobert[1], Solon P. Pissis[1,2⋆], and Alexandros Stamatakis[1,3]

[1] Heidelberg Institute for Theoretical Studies, 35 Schloss-Wolfsbrunnenweg, Heidelberg D-69118, Germany
{tomas.flouri,kassian.kobert,solon.pissis,alexandros.stamatakis}@h-its.org
[2] University of Florida, Florida Museum of Natural History, 1659 Museum Road, Gainesville, FL 32611, USA
[3] Karlsruhe Institute of Technology, Institute for Theoretical Informatics, 6980 Postfach, Karlsruhe 76128, Germany

Abstract. Tree pattern matching has been intensively studied over the past decades and it is a central problem in a wide range of applications. In many of these applications, it is essential to extract the repeated patterns in a tree within a mathematical structure. Recently, a linear-time algorithm for computing all subtree repeats in unlabeled ordered trees was presented (Christou et al., 2011)—a problem directly analogous to the well-known problem of computing all repetitions in strings (Crochemore, 1981). This algorithm was then extended to compute all subtree repeats in labeled ordered trees in linear time and space (Christou et al., 2012). In this article, we complete this series of results by presenting an algorithm to compute all subtree repeats in labeled unordered trees in linear time and space.

Keywords: tree pattern matching, unordered trees, labeled trees, subtree repeats

1 Introduction

Tree pattern matching has been intensively studied over the past decades and it is a central problem in a wide range of applications, among others, in the implementation of functional programming languages [10], term-rewriting systems [9], programming environments [2], code optimisation in compiler design [1], code selection [7], theorem proving [12], and computational biology [13].

In many applications, it is essential to extract the repeated patterns in a tree within a mathematical structure [6]. In particular, the *common subtrees* problem consists of finding all of the subtrees having the same structure and the same labels on the corresponding nodes of two labeled ordered trees [8]. This problem of equivalence, which is strictly related to the *common subexpression* problem [6], arises, for instance, in the code optimization phase of compiler design or in saving storage for symbolic computations [1,6].

Recently, Christou et al. [3] presented a linear-time algorithm for computing all subtree repeats in unlabeled ordered trees. This problem is directly analogous to the well-known problem of computing all repetitions in strings [5]. Computing all subtree repeats in labeled ordered trees can be solved by the algorithm presented in [8]. However this solution requires the construction of a suffix tree, which is not efficient in practice. Moreover, by analogy with string suffix automata, all subtree repeats can be directly computed by analysing states of the deterministic subtree pushdown automaton, which represents a full index of a tree for all subtrees [11]. However this

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

way of computing all subtree repeats requires time $\mathcal{O}(n \log n)$ [14]. In [4], Christou et al. extended the algorithm introduced in [3] to compute all subtree repeats in labeled ordered trees in linear time and space. In this article, we complete this series of results by presenting an algorithm to compute all subtree repeats in labeled unordered trees in linear time and space.

2 Preliminaries

An *alphabet* Σ is a finite, non-empty set whose elements are called *symbols*. A *string* over an alphabet Σ is a finite, possibly empty, string of symbols of Σ. The length of a string x is denoted by $|x|$, and the concatenation of two strings x and y by xy. A string w is a *factor* of a string x if there exist two strings y and z, such that $x = ywz$, and is represented as $w = x[i\,..\,j]$, $1 \leq i \leq j \leq |x|$.

An ordered (resp. unordered) and rooted tree is an ordered (resp. unordered) directed acyclic graph $T = (V, E)$ where V is the set of nodes and E the set of edges such that $E \subset V \times V$ with a special node v called the *root*, such that v has in-degree 0, and all other nodes of T have in-degree 1, where in-degree of a node v is the number of edges leading to v. A tree T is *labeled* if every node of T is labeled by a symbol from some alphabet. Different nodes may have the same label.

The number of nodes of a tree T is denoted by $|T|$. The subtree with node v as its root node is denoted by $T(v)$. We will consider only full subtree, i.e. it consists of all nodes and edges that can be reached from v. The height of a tree T is denoted by $h(T)$ and is defined as the number of edges of the longest path from the root of T to some *leaf* (a node with out-degree 0) of T. Analogously, the height of a node v in some tree T is denoted by $h(v)$ and is defined as $h(v) := h(T(v))$. For simplicity, in the rest of the text, we will refer to rooted unordered and labeled trees as trees.

Two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are *equal* ($T_1 = T_2$) if there exists a bijective mapping $f : V_1 \to V_2$ such that the following two properties hold

$$(v_1, v_2) \in E_1 \Leftrightarrow (f(v_1), f(v_2)) \in E_2$$
$$\mathit{label}(v) = \mathit{label}(f(v)), \forall v \in V_1$$

A *subtree repeat* R in a tree T is a set of nodes $v_1, v_2, \ldots, v_{|R|}$, such that $T(v_1) = T(v_2) = \ldots = T(v_{|R|})$.

In this article, we consider the problem of computing all subtree repeats of a tree T.

3 Algorithm

In this section, we present the algorithm for computing all subtree repeats in unordered trees. We first present a brief description of the steps used in the algorithm. We then give a formal description of each step in Algorithm 1.

In the following text we will identify each node by a unique number in the range of 1 to $|T|$; for example the index of the node's order in the preorder or postorder notation.

The idea of the algorithm can be explained by the following steps:

1. Partition nodes by height.
2. Assign a unique identifier to each label in Σ.
3. For each height level starting from 0.
   (i) For each node v of the current height level construct a string containing the identifier of label of v and the identifiers of subtrees attached to it.
   (ii) Sort the identifiers such that the order of identifiers is consistent in all strings.
   (iii) Find identical strings as repeats.
   (iv) Assign unique identifiers to each new subtree class.

We will explain each step by referring to the corresponding lines in Algorithm 1.

Partitioning the nodes according to their height is done in time linear to the size of the subject tree and is described in lines 1-2 of Algorithm 1. We then assign a unique identifier to each label in Σ in lines 3-7. The main loop of the algorithm starts at line 8 and processes the nodes of each height level starting from the leaves. The main loop consists of four steps. First, a string is constructed for each node v which consists of the identifier of the label of v followed by the identifiers given to $u_1, u_2, \ldots, u_{outdegree(v)}$, which represent the subtrees $T(u_1), T(u_2), \ldots, T(u_{outdegree(v)})$, where $u_1, u_2, \ldots, u_{outdegree(v)}$ are the children of v (lines 11-16). Assume that this particular step constructs k strings $s_1, s_2, \ldots, s_k$.

The next step is to sort the identifiers such that their order is consistent in all strings. To obtain this result we first need to remap individual identifiers contained in those strings to the range $[1, m]$ where m is the number of unique identifiers in the strings constructed for this particular height, and it holds that $m \leq \sum_{i=1}^{k} |s_k|$. We then use radix sort on the remapped identifiers and reconstruct the ordered strings (lines 17-20), $r_1, r_2, \ldots, r_k$.

The next step is to find identical strings as repeats. For that we lexicographically sort the ordered strings $r_1, r_2, \ldots, r_k$ and check neighbouring strings for equivalence (lines 23-33). Each subtree class R_i gets a new, unique identifier, assigned to the root nodes of all the subtrees in that class (lines 26 and 33).

At the end, each set R_i contains exactly those nodes that are the roots of repeats of a particular subtree of T.

Remapping from $\mathcal{D}_1 = [1, |T| + |\Sigma|)$ to $\mathcal{D}_2 = [1, |H[i]| + \sum_{v \in H[i]} outdegree(v)]$ can be done using an array A of size $|T| + |\Sigma|$, a counter m, and a queue Q. We read the numbers of the strings one by one. If a number x from domain $\mathcal{D}_1$ is read for the first time, we increase the counter m by one, set $A[x]$ to m and place m in Q. Subsequently, we replace x by m in the string. In case a number x has already been read before, i.e. $A[x] \neq 0$, we replace x in the string with $A[x]$. When the remapping is finished, only the altered positions in the array A are cleared, by traversing the elements of Q.

Algorithm 1: COMPUTE-SUBTREE-REPEATS

Input : Unordered tree $T = (V, E)$ labeled from Σ
Output: Sets $\mathcal{R}_{reps}$ of subtree repeats of T

▷ Partition tree nodes by height
for *all* $v \in V$ **do** ENQUEUE($H[h(T(v))], v$)
$cnt \leftarrow 0$
▷ Give each label a number from 1 to $|\Sigma|$
for *all labels* $l \in \Sigma$ **do**
    $cnt \leftarrow cnt + 1$
    $L[l] \leftarrow cnt$
▷ Compute subtree repeats
$reps \leftarrow 0$
for $i \leftarrow 0$ **to** $h(T)$ **do**
    $S \leftarrow \emptyset$
    ▷ Construct a string of numbers for each node v and its children
    foreach $v \in H[i]$ **do**
        Let $children(v) = \{u_1, u_2, \ldots, u_{outdegree(v)}\}$
        $s_v = L[label(v)]K[u_1]K[u_2] \ldots K[u_{outdegree(v)}]$
        $S \leftarrow S \cup \{s_v\}$
    ▷ Remap numbers $[1, |T| + |\Sigma|)$ to $[1, |H[i]| + \sum_{v \in H[i]} outdegree(v)]$
    $R \leftarrow$ REMAP(S)
    ▷ Bucket sort strings
    Bucket sort the numbers of all strings in R.
    Let R' be the set of individually sorted strings extracted from the sorted list of numbers from the previous step.
    Lexicographically sort the strings in R' using radix sort and obtain a sorted list R'' of strings $r_1, r_2, \ldots, r_{|R''|}$.
    Let each r_i be of form $k_1^i k_i^2 \ldots k_{|r_i|}^i$ and the corresponding, original unsorted string s_i of the form $L[v_1^i]K[v_2^i] \ldots K[v_{|r_i|}^i]$.
    $reps \leftarrow reps + 1$
    $\mathcal{R}_{reps} = \{v_1^1\}$
    $K[v_1^1] \leftarrow reps + cnt$
    for $j \leftarrow 2$ **to** k **do**
        if $r_j = r_{j-1}$ **then**
            $\mathcal{R}_{reps} \leftarrow \mathcal{R}_{reps} \cup \{v_1^j\}$
        else
            $reps \leftarrow reps + 1$
            $\mathcal{R}_{reps} \leftarrow \{v_1^j\}$
        $K[v_1^j] \leftarrow reps + cnt$

Theorem 1 (Correctness). *Algorithm 1 correctly computes all subtree repeats in a given unordered tree T.*

Proof. First note that if any two subtrees T_1 and T_2 are repeats of each other, they must, by definition, be of the same height. So the algorithm is correct in only comparing trees of the same height.

We show that the algorithm correctly computes all repeats for a tree of any height by induction. For the base case we look at an arbitrary tree of height 1 (trees with height 0 are trivial). Any tree of height 1 only has the root node and any number of leafs attached to it. At the root we can never find a repeat of more than once, so we only look at the next lower height level, the leaf nodes. Every two leafs with identical labels are, by construction of the algorithm, assigned the same identifiers and thus correctly found to be repeats of one another.

For the induction hypothesis assume that all (sub)trees of height $m-1$ are correctly assigned labels/identifiers, that are identical for two (sub)trees if and only if they are unordered repeats of one another, by the algorithm.

Consider an arbitrary tree of height $m+1$. As always, the number of repeats for the tree spanned from the root (node r) is one (the whole tree). Now consider the subtrees of height m. The root of any subtree of height m must be a child of r. For any child of r that induces a tree of height smaller than m, all repeats have been correctly calculated by assumption already.

Two (sub)trees are repeats of one another if and only if the two roots have the same label and there is a one to one mapping from subtrees induced by children of the root of one tree to topologically equivalent subtrees induced by children of the root of the second tree. By the induction hypothesis, all such topologically equivalent subtrees of height $m-1$ or smaller have already been assigned identifiers/labels that are unique for each equivalence class. So deciding whether two subtrees are repeats of one another can be done by comparing the root labels and the corresponding identifiers of their children, which is exactly the process described in the algorithm. The approach used in the algorithm correctly identifies identically labeled strings since the order of identifiers has been sorted for a given height class. Thus the algorithm finds all repeats of height m (and $m+1$ at the root). □

Theorem 2 (Complexity). *Given a tree T, Algorithm 1 runs in time $\mathcal{O}(|T|)$.*

Proof. We can prove the linearity of the algorithm by analyzing each of the steps in the brief description of the algorithm. Steps 1 and 2 are trivial and can be computed in $|T|$ and $|\Sigma|$ steps, respectively. Note that $|\Sigma| \leq |T|$.

The main for loop visits each node of T once. For each node v a string s_v is constructed which contains the identifier of the label of v and the identifiers assigned to the children nodes of v. Thus, each node is considered at most twice: once as the parent and once as a child. This leads to $2n-1$ node traversals, since the root node is the only node that is considered exactly once. The constructed strings for a height level i are composed of the nodes in $H[i]$ and their respective children. In total we have $c(i) := \sum_{v \in H[i]} \mathit{outdegree}(v)$ children nodes at some height level i. Therefore, the total size of all constructed strings for a particular height level i is $|H[i]| + c(i)$.

Step 3ii runs in time linear to the number of nodes in each height level i and their children. This is because the remapping can be done in linear time to $|H[i]|+c(i)$ and ensures that the identifiers in each string are within the range of 1 to $|H[i]| + c(i)$. Using bucket sort we can sort the remapped identifiers in time $|H[i]| + c(i)$ for each

height level i. Consequently, sorting the identifiers in each string can be done in time $|H[i]| + c(i)$ by traversing the sorted list of identifiers and positioning the respective identifier in the corresponding string on a first-read-first-place basis. This requires an additional storage space of size $|H[i]| + c(i)$ for keeping track which remapped identifier corresponds to which strings.

After remapping and sorting the strings, finding identical strings as repeats requires lexicographical sorting of the strings. Strings that are identical form classes of repeats. Lexicographical sorting using radix sort in this case requires time $\mathcal{O}(|H[i]| + c(i))$ and a memory space of at most $|T| + |\Sigma|$ elements since the identifiers are in the range of 1 to $|T| + |\Sigma|$. The considered memory space is allocated only once and only the used elements are cleared at each step with the aid of a queue as explained in the remapping function.

Summing over all height levels we obtain $\sum_{i=0}^{h(T)}(|H[i]| + c(i)) = 2n - 1$. Thus the total runtime over all height levels for each step described in the loop is $\mathcal{O}(|T|)$. In total, the asymptotic time and space complexity of the algorithm is $\mathcal{O}(|T|)$. □

With a slight modification, our algorithm can also be used for detecting subtree repeats in ordered trees. By omitting step 3ii (lines 19-21 in Algorithm 1) repeats are found if and only if the order of the children is also consistent.

We conclude with an example demonstrating Algorithm 1. Consider the tree T in Figure 1. The superscript indexes denote the number associated with each node, which in this case is the order in which each node is visited during a preorder traversal of T. Lines 1-2 partition the nodes of T in $h(T) + 1$ sets according to their height, where $h(T)$ is the height of the root node of T, or the height of the tree. The sets $H[0] = \{3, 5, 6, 7, 8, 10, 11, 13, 14, 15, 17, 19, 20, 23, 25, 26, 28\}$, $H[1] = \{4, 12, 18, 22, 24, 27\}$, $H[2] = \{2, 9, 21\}$, $H[3] = \{16\}$, and $H[4] = \{1\}$ are thus created. Lines 5-7 create a mapping between labels and numbers. $L[a] = 1$, $L[b] = 2$, $L[c] = 3$ and $L[d] = 4$.

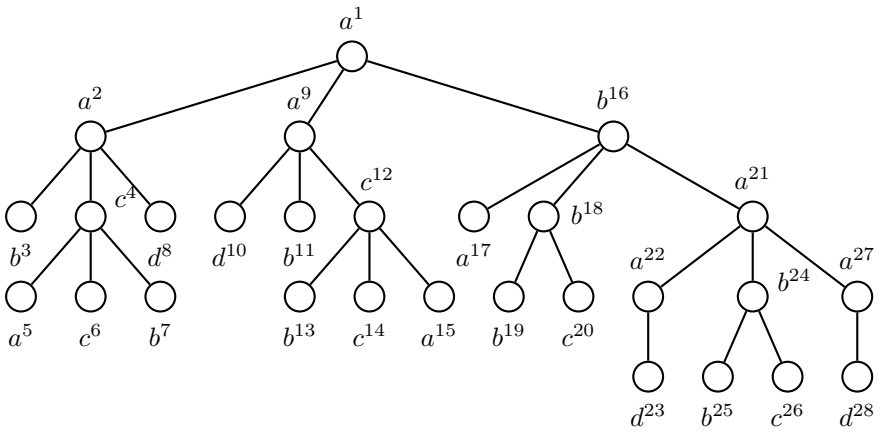

Figure 1. Graphical representation of tree T

Table 1 shows the state of lists S, R, R', R'' during the computation of the main loop of Algorithm 1 for each height level. Figure 2 depicts tree T with the associated identifier for each node as assigned by Algorithm 1.

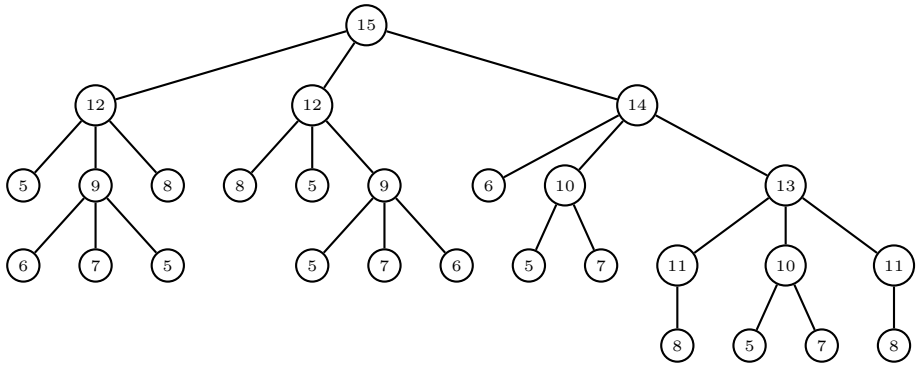

Figure 2. Graphical representation of tree T with the associated identifier for each node as assigned by Algorithm 1

Height	Step	Process	Repeats
0	Strings: S	$2,1,3,2,4,4,2,2,3,1,1,2,3,4,2,3,4$	$\mathcal{R}_1 = \{3,7,11,13,19,25\}$
	Remapping: R	$1,2,3,1,4,4,1,1,3,2,2,1,3,4,1,3,4$	$\mathcal{R}_2 = \{5,15,17\}$
	Sorting: R'	$1,2,3,1,4,4,1,1,3,2,2,1,3,4,1,3,4$	$\mathcal{R}_3 = \{6,14,20,26\}$
	Repeats: R''	$\underbrace{1,1,1,1,1,1}_{5},\underbrace{2,2,2}_{6},\underbrace{3,3,3,3}_{7},\underbrace{4,4,4,4}_{8}$	$\mathcal{R}_4 = \{8,10,23,28\}$
1	Strings: S	3 6 7 5, 3 5 7 6, 2 5 7, 1 8, 2 5 7, 1 8	
	Remapping: R	1 2 3 4, 1 4 3 2, 5 4 3, 6 7, 5 4 3, 6 7	$\mathcal{R}_7 = \{22,27\}$
	Sorting: R'	1 2 3 4, 1 2 3 4, 3 4 5, 6 7, 3 4 5, 6 7	$\mathcal{R}_5 = \{4,12\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4,1\ 2\ 3\ 4}_{9},\underbrace{3\ 4\ 5,3\ 4\ 5}_{10},\underbrace{6\ 7,6\ 7}_{11}$	$\mathcal{R}_6 = \{18,24\}$
2	Strings: S	1 5 9 8, 1 8 5 9, 1 11 10 11	
	Remapping: R	1 2 3 4, 1 4 2 3, 1 5 6 5	$\mathcal{R}_8 = \{2,9\}$
	Sorting: R'	1 2 3 4, 1 2 3 4, 1 5 5 6	$\mathcal{R}_9 = \{21\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4,1\ 2\ 3\ 4}_{12},\underbrace{1\ 5\ 5\ 6}_{13}$	
3	Strings: S	2 6 10 13	
	Remapping: R	1 2 3 4	
	Sorting: R'	1 2 3 4	$\mathcal{R}_{10} = \{16\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4}_{14}$	
4	Strings: S	1 12 12 14	
	Remapping: R	1 2 3 4	
	Sorting: R'	1 2 3 4	$\mathcal{R}_{11} = \{1\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4}_{15}$	

Table 1. State of lists S, R, R', R'' for each height level and resulting sets $\mathcal{R}_{reps}$ of subtree repeats

4 Conclusion

In this article, we have presented a remarkably simple linear-time algorithm for computing all subtree repeats in a given unordered tree with respect to the size of the tree. In addition to the linear runtime, the algorithm also requires linear memory space. This result completes the series of results presented in [3,4].

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman: *Compilers: principles, techniques, and tools*, Addison Wesley, 2 ed., 2006.
2. D. R. Barstow, H. E. Shrobe, and E. Sandewall: *Interactive Programming Environments*, McGraw-Hill, Inc., 1984.
3. M. Christou, M. Crochemore, T. Flouri, C. Iliopoulos, J. Janoušek, B. Melichar, and S. Pissis: *Computing all subtree repeats in ordered ranked trees*, in String Processing and Information Retrieval, R. Grossi, F. Sebastiani, and F. Silvestri, eds., vol. 7024 of Lecture Notes in Computer Science, Springer, 2011, pp. 338–343.
4. M. Christou, M. Crochemore, T. Flouri, C. Iliopoulos, J. Janoušek, B. Melichar, and S. Pissis: *Computing all subtree repeats in ordered trees.* Information Processing Letters, 2012, (submitted).
5. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Information Processing Letters, 12(5) 1981, pp. 244–250.
6. P. J. Downey, R. Sethi, and R. E. Tarjan: *Variations on the common subexpression problem.* Journal of ACM, 27(4) 1980, pp. 758–771.
7. C. Ferdinand, H. Seidl, and R. Wilhelm: *Tree automata for code selection.* Acta Inf., 31 1994, pp. 741–760.
8. R. Grossi: *On finding commong subtrees.* Theoretical Computer Science, 108(2) 1993, pp. 345–356.
9. C. M. Hoffmann and M. J. O'Donnell: *Programming with equations.* ACM Trans. Program. Lang. Syst., 4 1982, pp. 83–112.
10. P. Hudak: *Conception, evolution, and application of functional programming languages.* ACM Computing Surveys, 21 1989, pp. 359–411.
11. J. Janoušek: *String suffix automata and subtree pushdown automata*, in Proceedings of the Prague Stringology Conference 2009 (PSC 2009), J. Holub and J. Žďárek, eds., 2009, pp. 160–172.
12. D. E. Knuth and P. B. Bendix: *Simple word problems in universal algebra*, in Computational problems in abstract algebra, J. Leech, ed., Pergamon Press, 1970, pp. 263–297.
13. G. Mauri and G. Pavesi: *Algorithms for pattern matching and discovery in RNA secondary structure.* Theoretical Computer Science, 335(1) 2005, pp. 29–51.
14. B. Melichar: *Arbology: trees and pushdown automata*, in Proceedings of the fourth International Conference on Language and Automata Theory and Applications (LATA 2010), A.-H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 32–49.

XMLCorrector: an Open Source Tool for XML Document Correction

Joshua Amavi[1], Béatrice Bouchou-Markhoff[2], and Agata Savary[2]

[1] LIFO - Université d'Orléans, Orléans, France
joshua.amavi@univ-orleans.fr
[2] Université François Rabelais Tours, LI, Blois Campus, France
beatrice.bouchou+agata.savary@univ-tours.fr

Abstract. Given a well-formed XML document t seen as a tree, a schema S expressed with a DTD and a non negative threshold th, XMLCorrector[1] allows to find every tree t' valid with respect to S such that the edit distance between t and t' is no higher than th. Its underlying algorithm is based on a recursive exploration of the finite-state automata representing structural constraints in S, as well as on the construction of an edit distance matrix storing edit sequences leading to valid trees. It has been made public under the GNU LGPL v3 license. We present its structure and main components, and we describe its use through the user interface. We conclude by situating it among related works.

Keywords: XML document-to-schema correction, open source tool

1 Introduction

For a given well-formed XML document, a DTD and a threshold, the aim of XMLCorrector is to find all correct documents within the threshold, *i.e.*, all valid documents whose distance from the original document does not exceed the threshold. It outputs the list of all operation sequences leading to such correct documents, as well as the resulting XML documents themselves.

Indeed, for correcting an XML document (*i.e.*, an XML tree) t, we allow three kinds of elementary operations on nodes: (i) insertion of a node at a position in t, (ii) deletion of a leaf in t, (iii) relabeling of a node in t. A sequence of node operations transforms a tree t into another tree t'. Figure 1 shows an example of an initial tree t and of a tree t' resulting from t by the application of an operation sequence. Each node operation has a cost. The cost of the node operation sequence is equal to the sum of the costs of each operation in the sequence. The distance between t and t' is defined as the minimal cost of all operation sequences allowing to transform t into t'. For instance, if we admit cost 1 for each node operation, the operation sequence in Figure 1 has cost 3. As there exists no operation sequence transforming t into t' with a lower cost, the distance between t and t' is 3. The distance between a tree t and a tree language defined by a schema S is the minimal distance between t and any tree valid with respect to S.

The algorithm used by XMLCorrector to correct an XML document t w.r.t. a schema S allows (i) to compute the distance between t and S, more precisely to compute one minimal-cost correction, (ii) to compute all minimal-cost corrections and (iii) to compute all threshold-bounded corrections. Indeed, given a non negative threshold th, it can find every tree t' valid with respect to S such that the edit

[1] http://www.info.univ-tours.fr/~savary/English/xmlcorrector.html

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

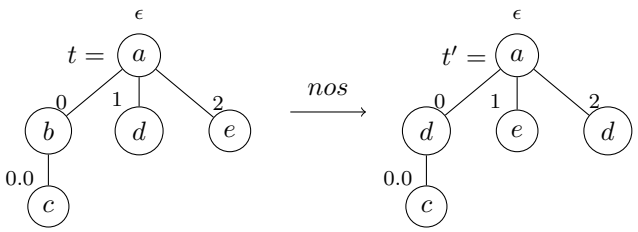

Figure 1. Application of the node edit sequence $nos = \langle (add, 1, e), (delete, 2, /), (relabel, 0, d) \rangle$ on the tree t, $cost(nos) = 3$

distance between t and t' is no higher than th. In this way it offers the warranty of getting all solutions verifying the threshold criterion. This allows to be sure that the best solution is actually in the resulting set. As the features defining what is *the best solution* highly depend on application contexts and are often difficult to select automatically, XMLCorrector allows to determine them interactively, before designing scripts to do it. A common scenario is to compute one minimal-cost solution, which gives the distance, then to assign this distance to the threshold th, and then to increase th following the application needs.

Besides its usefulness as an interactive tool, XMLCorrector is made public under the GNU LGPL v3. This license allows not only to use but also to modify the source codes. Thus, an application can integrate the core of XMLCorrector source code, while performing pre-processing and post-processing tasks in order to take advantage of the computed solutions in a way adapted to its particular context. But it can also adapt these codes to its context, for instance for considering other edit operation costs, or other kinds of edit operations, or other types of schemas, etc.

XMLCorrector's algorithms are formally presented, proven and deeply analyzed (with many experimental results) in [1]. Due to the fact that all edit sequences and the corresponding corrections are generated, the theoretical complexity is exponential, but the experiments show polynomial behaviors for all the tested cases. In this paper we focus on the tool, not on these underlying algorithms. The rest of this paper is organized as follows: in Section 2 we present the general structure of XMLCorrector and its main components. In Section 3, a scenario of usage is given, that shows how a user can specify her input parameters, and how the system outputs the results and allows to browse it. We conclude by discussing the strengths of XMLCorrector *w.r.t.* related works in Section 4.

2 Structure and components

Our algorithms consist in generating all possible candidate corrections while dynamically computing tree-edit distances for subtree corrections. Due to their high theoretical complexity (as for most of enumeration problems), XMLCorrector relies on a very careful implementation. It is structured in many packages, as illustrated in Figure 2, which can be divided in the following groups:

- Trees, nodes and addresses (or node identifiers): classes *Address*, *Node* and *Tree*.
- Operations and operation sequences: classes *Operation*, *OperationSequence*, *EditOperation*, *AddNodeOperation*, *RemoveNodeOperation* and *RelabelNodeOperation*.

– Edit-distance matrix and cells: classes $Matrix$, $SimpleMatrix$ and $RightExtended\text{-}Matrix$.
– Schema rules and finite state automata: class $TreeSchema$ and the automata library (cf. Acknowledgments).
– Tree-to-grammar correction: classes $XmlCorrectorMatrix$ and $XmlCorrector$.
– Tree-edit-distance computation
– Pre-processing of DTDs and XML documents
– Post-processing of the resulting structure
– User interface

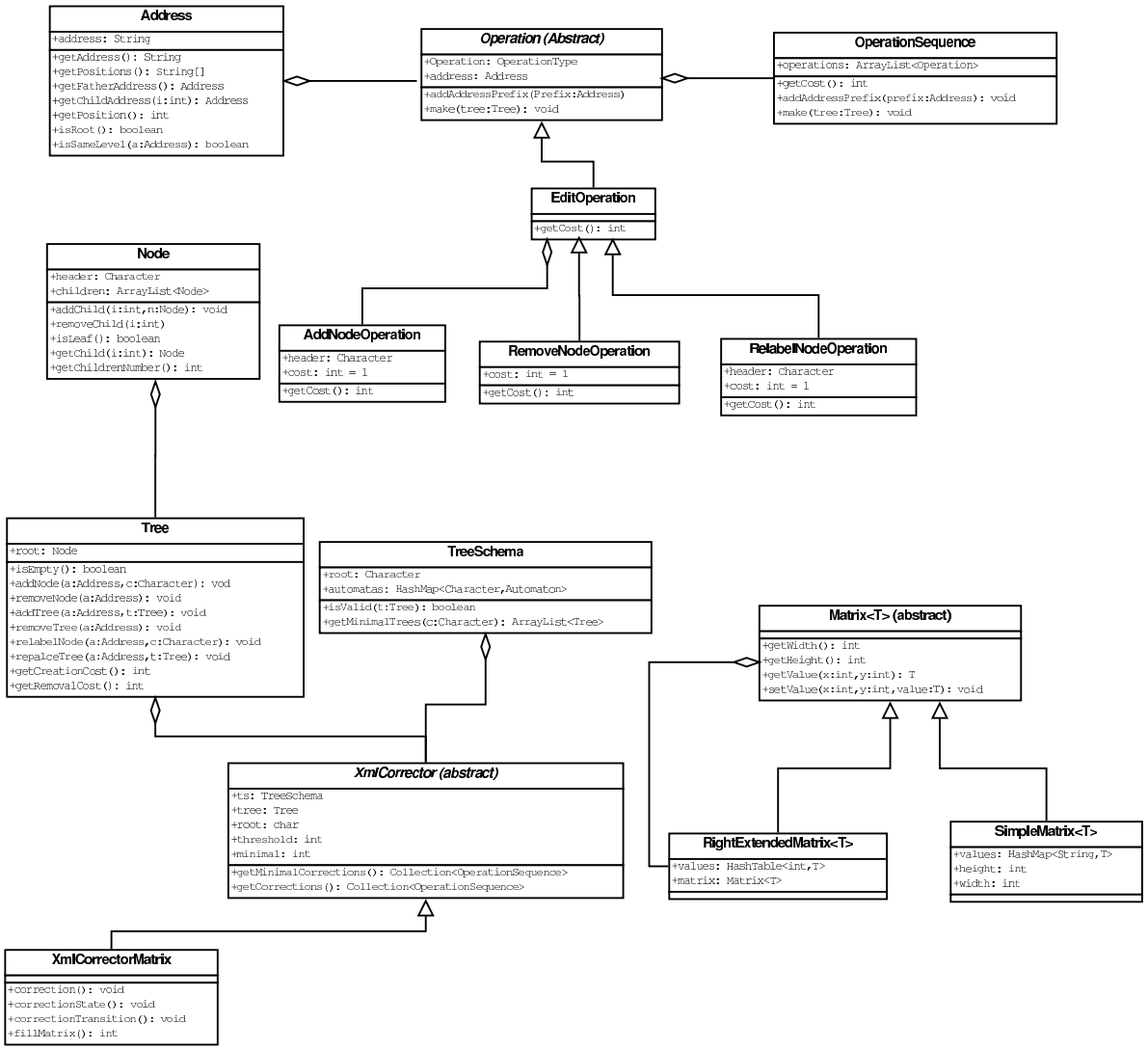

Figure 2. XMLCorrector's Structure

An instance of the class XmlCorrectorMatrix is created in order to correct the given tree t (corresponding to the given XML document), with respect to the given schema S, within a given threshold th. It returns an instance of the class Results, *i.e.*, a set of pairs (cost, operationSequence). Each operation sequence can lead to a different tree. This can be easily verified with the provided user interface, that we illustrate in the next section.

3 Demonstration of an interactive scenario

In order to launch the application, one has just to click on the jar file `XmlCorrector.jar` or open a terminal and enter the following command: `java -jar XmlCorrector.jar`. The main window is then opened as shown in Figure 3.

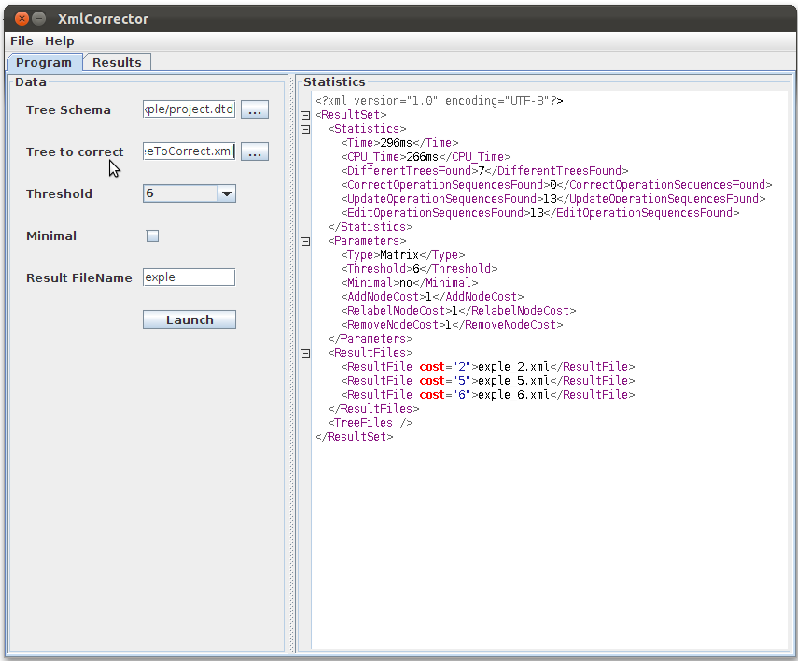

Figure 3. Main Window

It is necessary to fill out the left hand tab form and click on the button "Launch". The following data are needed to fill out the form fields[2]:

– Tree Schema – the DTD file which contains the description of the schema. The tree schema selected in Figure 3 is the file `project.dtd`, shown in Fig. 4, provided in the directory `example/projects`. According to this DTD a project description requires: the name of the project, the manager (name and salary), and the list of employees (names and salaries). A project can have a list of sub-projects.

```
<!ELEMENT projs (proj*)>
<!ELEMENT proj (name,emp,proj*,emp*)>
<!ELEMENT emp (name,salary)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT salary (#PCDATA)>
```

Figure 4. The DTD of the XML file used for the example

– Tree to correct – the XML document to be corrected with respect to the schema mentioned above. The document selected in Figure 3 is the file `projectTreeToCorrect.xml`, shown in Fig. 5, provided in the directory `example/projects`. Note that this document is incorrect: the `<salary>` is missing for the manager of the project "Cooking Pierogies", and there is an additional `<address>` tag for the

[2] The running example, provided in `example/projects`, has been inspired by [4].

manager of the project "Preparing Stuffing".
If one want to correct an empty tree, the selected XML file should contain the tag `<EmptyTree />` only.

```
<?xml version="1.0" encoding="UTF-8"?>
<projs>
   <proj>
      <name>Cooking Pierogies</name>
      <emp>
         <name>Smith</name>
      </emp>
      <proj>
         <name> Preparing Stuffing</name>
         <emp>
            <name>John</name>
            <salary>80K</salary>
            <address>London</address>
         </emp>
         <emp>
            <name>Mary</name>
            <salary>40K</salary>
         </emp>
      </proj>
      <emp>
         <name>Peter</name>
         <salary>30K</salary>
      </emp>
      <emp>
         <name>Steve</name>
         <salary>50K</salary>
      </emp>
   </proj>
</projs>
```

Figure 5. XML document which is correct

- Threshold – the maximum value for the edit distance between the tree to correct and the possible candidate trees. Only trees within this threshold are proposed.
- Minimal – the type of search for the correction algorithm. If the box is checked then only the trees which have the minimal edit distance are proposed. Otherwise all trees within the threshold are proposed.
- Result Filename – prefix for the name of the result files which are stored in the directory `results/`.

When the correction terminates, statistics are shown in the right hand panel of the window in Figure 3:

- execution time,
- number of correction candidates found,
- parameters used (the threshold value, the Boolean value for the minimal search, the cost of adding, removing or relabeling a node)
- names of result files `prefix_i.xml` (where `prefix` is the word entered for the field `Result Filename`).

The result file `prefix_i.xml` contains all operation sequences with cost equal to i, as well as the resulting XML trees obtained by applying these sequences on the initial XML tree. We can visualize these sequences and trees in the second tab `Results`. The upper left hand panel in Figure 6 shows the list of result files, and the XML tree to correct is displayed in upper right hand panel. After selecting one result file (here `exple_2.xml`) the list of operation sequences contained in this file is displayed below. After selecting one operation sequence in this list, its detailed content is shown in the lower left hand panel and the resulting correct XML tree is displayed in the lower right hand panel.

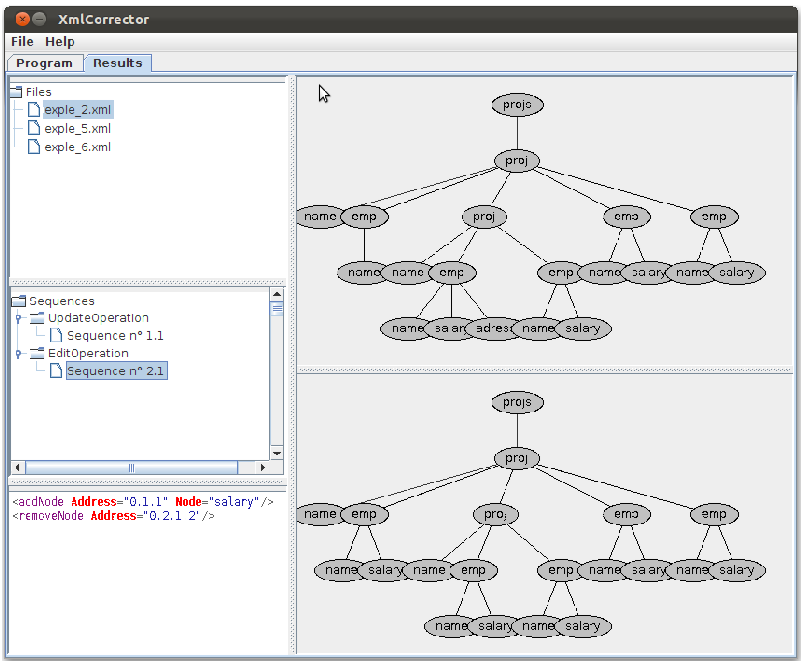

Figure 6. Content of the `Results` tab

It is possible to modify the cost of each of the three node operations (add, relabel and remove) via the menu option *File* → *Options*.

The package also contains a full data benchmark, that has been used to perform the experiments reported in [1]. This benchmark is also described in [1]. As already mentioned in Introduction, the license of the whole package allows not only to use it but also to modify the source codes and to integrate it in other applications.

4 Conclusion

We have presented the main features of XMLCorrector, a tool for correcting XML documents w.r.t. a DTD. In the web domain there are many situations where it is necessary to get documents that are close to an original one but that fit a given schema. This is due in particular to its *constant need of evolution*, both for XML documents and schemas. For instance in [3] we consider documents modified frequently, which implies the correction of the resulting documents where (incremental) validation fails. Conversely, in [7] updates on the schema are considered. The update on a DTD leads to computing corresponding updates for documents that were valid *w.r.t.* the old DTD. Many other applications are considered in [9], such as XML data integration,

web service searching and composition, consistent queries on XML databases, and XML document classification.

There exist several proposals of algorithms to compute the distance between a document t and a schema S (as in [10] or [4]), or to compute the minimal-cost correction of t *w.r.t.* S, or to compute all minimal-cost corrections (see for instance [2] or [8]), or to compute all K-minimal corrections (as done in [6]). But to the best of our knowledge, there is no tool that can be compared to XMLCorrector. Indeed, as we show in our extensive related work Section in [1], even if experimental results are provided by several of the approaches ([2], [4], [8] and ours), few of them (including ours) operate on real-life rather than synthetic data. Moreover, only four approaches offer downloadable implementations (executables and/or source codes). Two of them ([4] and [5]) lack any user's documentation. We have tried to use the demo cited by [8], but it seems to admit a non standard schema format (it does not run on our data). Our approach seems to be the only one that offers, in addition to the executable and the source code, also the user's guide and the set of testing data used to obtain the experimental results. Consequently, it seems to be the only reproducible one.

Acknowledgements

XMLCorrector includes the following open source libraries:

- dk.brics.automaton v1.6 under the BSD 2-Clause license, developed by Anders Møller at Aarhus University et al. (`http://www.brics.dk/automaton/`)
- jdom v1.1 under an open source license derived from the BSD 3-Clause License, developed by the JDOM Project. (`http://www.jdom.org/`)
- bounce v0.18 under the BSD 3-Clause License, developed by Edwin Dankert. (`http://www.edankert.com/bounce/`)
- DTDParser v1.21 under the GNU Lesser General Public License v3, developed by Mark Wutka. (`http://mvnrepository.com/artifact/com.wutka/dtdparser/1.21/`)

References

1. J. Amavi, B. Bouchou, and A. Savary: *On Correcting XML Documents With Respect to a Schema*. Submitted to The Computer Journal.
2. U. Boobna and M. de Rougemont: *Correctors for XML Data*, in Proceedings of XSym 04, Toronto, Canada, vol. 3186 of Lecture Notes in Computer Science, Springer, 29–30 August 2004, pp. 97–111.
3. B. Bouchou, A. Cheriat, M. Halfeld Ferrari Alves, and A. Savary: *XML Document Correction: Incremental Approach Activated by Schema Validation*, in Proceedings of IDEAS 06, Delhi, India, IEEE Computer Society, 11–14 December 2006, pp. 228–238.
4. S. Staworko and J. Chomicki: *Validity-Sensitive Querying of XML Databases*, in Proceedings of EDBT 06, Munich, Germany, Revised Selected Papers, vol. 4254 of Lecture Notes in Computer Science, Springer, 26–31 March 2006, pp. 164–177.
5. S. Staworko, E. Filiot, and J. Chomicki: *Querying Regular Sets of XML Documents*, in Proceedings of LiD 08, Rome, Italy, 2008.
6. N. Suzuki: *Finding K Optimum Edit Scripts between an XML Document and a RegularTree Grammar*, in Proceedings of EROW 07, Barcelona, Spain, CEUR-WS.org, 13 January 2007.
7. N. Suzuki: *An algorithm for inferring k optimum transformations of xml document from update script to dtd*. IEICE Transactions, 93-D(8) 2010, pp. 2198–2212.
8. M. Svoboda and I. Mlýnková: *Correction of Invalid XML Documents with Respect to Single Type Tree Grammars*, in Proceedings of NDT 11, Macau, China, vol. 136 of Communications in Computer and Information Science, Springer, 11–13 July 2011, pp. 179–194.

9. J. Tekli, R. Chbeir, A. Traina, and C. Traina: *XML document-grammar comparison: related problems and applications.* Central European Journal of Computer Science, 1 2011, pp. 117–136.
10. G. Xing, C. R. Malla, Z. Xia, and S. D. Venkata: *Computing Edit Distances Between an XML Document and a Schema and its Application in Document Classification*, in Proceedings of SAC 06, Dijon, France, ACM, 23–27 April 2006, pp. 831–835.

Motif Matching Using Gapped q-gram Patterns

Emanuele Giaquinta and Esko Ukkonen

Department of Computer Science, University of Helsinki, Finland
{emanuele.giaquinta | ukkonen}@cs.helsinki.fi

Abstract. We present a new algorithm for the problem of *multiple string matching* of gapped q-gram patterns, where a gapped q-gram pattern is a sequence of q symbols such that there is a gap of fixed length between each two consecutive symbols. The problem has applications in the discovery of transcription factor binding sites in DNA sequences when using the Feature Motif Model to describe transcription factor specificities. We also provide experimental results which show that the presented algorithm is fast in practice.

1 Introduction

We consider the problem of matching a set $\mathcal{P}$ of gapped q-gram patterns against a given text of length n, where a q-gram pattern is a sequence of q symbols, over a finite alphabet Σ of size σ, such that there is a gap of fixed length between each two consecutive symbols. In particular, we are interested in computing the list of matching patterns for each position in the text. This problem is a specific instance of the *Variable Length Gaps string matching problem* (VLG problem) for multiple patterns and has applications in the discovery of transcription factor (TF) binding sites in DNA sequences when using the Feature Motif Model [11] to represent TF binding specificities.

In the VLG problem a pattern is a concatenation of strings and of variable-length gaps. An efficient approach to solve the problem for a single pattern is based on the simulation of nondeterministic finite automata [7,4]. A method to solve this problem for one or more patterns is to translate the patterns into a regular expression [8,3]. The best bound for a regular expression is $O(n(k\frac{\log w}{w} + \log k))$ [3], where k is the number of the strings and gaps in the pattern and w is the word size in bits. This approach provides a huge gain when k is small compared to the length of the pattern. Observe that in our case $k = \Theta(len(\mathcal{P}))$, where $len(\mathcal{P})$ is the sum of the number of symbols in each pattern. Although the resulting bound is good, the method used to implement fixed-length gaps, based on maintaining multiple bit queues using word level parallelism, is not practical to our knowledge. There also exist algorithms efficient in terms of the total number α of occurrences of the strings in the patterns (keywords) within the text [6,10,2]. Note that the number of occurrences of a keyword that occurs in r patterns and in l positions in the text is equal to $r \times l$. The best bound obtained for a single pattern is $O(n \log k + \alpha)$ [2]. This method can also be extended to multiple patterns: however, if all the keywords have unitary length this result is not ideal, because α is $\Omega(n\frac{len(\mathcal{P})}{\sigma})$ on average if we assume a uniform distribution for the patterns. A similar approach to deal with multiple patterns [5] leads to $O(n(\log k + K) + \alpha')$ time, where K is the maximum number of suffixes of a keyword that are also keywords and α' is the number of occurrences in the text of pattern prefixes ending with a keyword. This result can be preferable in general when $\alpha' < \alpha$, but in our problem a bound similar to the one on α holds also for α', as

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

the number of prefixes of unitary length is $\Omega(n\frac{|\mathcal{P}|}{\sigma})$ on average. Note that the above bounds do not include preprocessing time and the $\log k$ term in them is due to the simulation of the Aho-Corasick automaton for the strings in the patterns (and can be removed by computing the full transition table of the automaton).

In this paper we present a new simple algorithm for the problem of matching a set of gapped q-gram patterns. Our algorithm is based on dynamic programming and bit-parallelism [1] and has $O(ng_{\text{w-span}}\lceil len(\mathcal{P})/w\rceil + occ)$-time complexity, where $1 \leq g_{\text{w-span}} \leq w$ is, roughly speaking, the maximum number of distinct gap lengths that span a single word in our encoding. It is preferable only when $g_{\text{w-span}} \ll w$ and $len(\mathcal{P})$ is small, but can also be used as a filter to speed up the algorithm by Haapasalo et al [5]. When the gap lengths are constrained, the proposed algorithm is fast in practice, as shown by experimental evaluation.

The rest of the paper is organized as follows. In Section 2 we recall some preliminary notions and elementary facts. In Section 3 we discuss the motivation for our work. In Section 4 we present a new algorithm for the problem of matching gapped q-gram patterns. Finally, In Section 5 we present experimental results to evaluate the performance of our algorithm.

2 Basic notions and definitions

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ denote an alphabet of size σ. Let Σ^* denote the Kleene star of Σ, and Σ^m the set of all possible sequences of length m over Σ. $|S|$ is the length of string S, $S[i], i \geq 0$, denotes its $(i+1)$-th character, and $S[i \ldots j]$ its substring between the $(i+1)$-st and the $(j+1)$-st characters (inclusive).

A gapped q-gram pattern P is of the form

$$(p_1 \cdot j_1 \cdot p_2 \cdot \ldots \cdot j_{q-1} \cdot p_q)$$

where $\langle p_1, p_2, \ldots, p_q\rangle$ is a sequence of q symbols and $\langle j_1, j_2, \ldots, j_{q-1}\rangle$ is the associated sequence of gaps, i.e., $j_k \in N$ is the length of the gap between symbols p_k and p_{k+1}. We say that P occurs in a string S at ending position i if

$$S[i - \sum_{i=k}^{q-1}(j_i + 1)] = p_k\,,$$

for $k = 1, \ldots, q$. In this case we write $P \sqsupseteq_g S_i$. We denote with $len(P)$ the number of symbols in P. Given two gapped q-gram patterns $P = (p_1 \cdot j_1 \cdot p_2 \cdot \ldots \cdot j_{q-1} \cdot p_q)$ and $P' = (p'_1 \cdot j'_1 \cdot p'_2 \cdot \ldots \cdot j'_{l-1} \cdot p'_l)$, we say that P' is a prefix of P of length l iff $l \leq q$, $p_i = p'_i$, for $i = 1, \ldots, l$, and $j_i = j'_i$, for $i = 1, \ldots, l-1$. Given a set of gapped q-gram patterns $\mathcal{P}$, we denote with $g_{\max}(\mathcal{P})$ the maximum gap length in $\mathcal{P}$.

W.l.o.g. we assume that all patterns in $\mathcal{P}$ are distinct. In general, we can radix-sort the patterns and replace duplicates with lists of their IDs from the original problem setting, which is performed in $O(|\mathcal{P}|)$ time.

The RAM model is assumed, with words of size w in bits. We use some bitwise operations following the standard notation as in C language: &, |, $\sim$, $\ll$ for `and`, `or`, `not` and `left shift`, respectively. The function to compute the most significant set bit of a word x is $\lfloor \log_2(x) \rfloor$.

3 Motivation

Given a DNA sequence and a motif that describes the binding specificities of a given transcription factor, we study the problem of finding all the binding sites in the sequence that match the motif. The traditional model used to represent transcription factor motifs is the Position Weight Matrix (PWM). This model assumes that there is no correlation between positions in the sites, that is, the contribution of a nucleotide at a given position to the total affinity does not depend on the other nucleotides which appear in other positions. The problem of matching the locations in DNA sequences at which a given transcription factor binds to is well studied under the PWM model [9]. Recently, a new model, named Feature Motif Model (FMM), has been proposed [11]. In this model the TF binding specificities are described with so-called *features*, i.e., rules that assign a weight to a set of associations between symbols and positions. Given a DNA sequence, a set of features and a motif length m, the matching problem consists in computing the score of each site (substring) of length m in the sequence, where the score of a site is the sum of the weights of all the features that occur in the site. Formally, a *feature* can be denoted as

$$\{(a_1, i_1), \ldots, (a_q, i_q)\} \to w$$

where w is the affinity contribution of the feature and $a_j \in \{A, C, G, T\}$ is the nucleotide which must occur at position i_j, for $j = 1, \ldots, q$ and $1 \leq i_j \leq m$. It is easy to transform these rules into new rules where the left side is a gapped q-gram pattern: if $i_1 < i_2 < \ldots < i_q$, we can induce the following gapped q-gram pattern rule

$$(a_1 \cdot (i_2 - i_1 - 1) \cdot \ldots \cdot (i_q - i_{q-1} - 1) \cdot a_q) \to (i_q, w).$$

This transformation has the advantage that the resulting pattern is position independent. Moreover, after this transformation, different features may share the same gapped q-gram pattern. Hence, the matching problem can be decomposed into two components: the first component identifies the occurrences of the groups of features by searching for the corresponding gapped q-gram patterns, while the second component computes the score for each candidate site using the information provided by the the first component. For a motif of length m, the second component can be easily implemented by maintaining the score for m site alignments simultaneously with a circular queue of length m. Each time a group of features with an associated set of position/weight pairs $\{(i_1, w_1), \ldots, (i_r, w_r)\}$ is found at position j in the sequence, the algorithm adds the weight w_k to the score of the alignment that ends at position $j + m - i_k$ in the sequence, if $j \geq i_k$.

4 Matching a set of gapped q-gram patterns

In this section we present a practical algorithm to search for a set $\mathcal{P}$ of gapped q-gram patterns. We first devise a simple algorithm based on dynamic programming whose time complexity matches the best known bound and then show how to parallelize it using word-level parallelism.

Let P^k be the k-th pattern in $\mathcal{P}$ and let p_i^k and j_i^k be its i-th symbol and gap length, respectively. Let also P_l^k be the prefix of P^k of length l. Let

$$D_i = \{(k, l) \mid P_l^k \sqsupseteq_g T_i\},$$

```
GQ-MATCHER (P, T)

    PREPROCESSING
 1. G ← ∅
 2. m ← len(P)
 3. I ← 0^m, M ← 0^m
 4. for c ∈ Σ do B(c) ← 0^m
 5. for g = 0, ..., g_max(P) do C(g) ← 0^m
 6. l ← 0
 7. for (p_1 · j_1 · p_2 · ... · j_{q-1} · p_q) ∈ P do
 8.       I ← I | 1 ≪ l
 9.       for k = 1, ..., q do
10.            B(p_k) ← B(p_k) | 1 ≪ l
11.            if k = q then
12.                 M ← M | 1 ≪ l
13.            else C(j_k) ← C(j_k) | 1 ≪ l
14.                 G ← G ∪ {j_k}
15.            l ← l + 1

    SEARCHING
16. for i = 0, ..., |T| − 1 do
17.       H ← 0^m
18.       for g ∈ G do
19.          H ← H | (D_{i−1−g} & C(g))
20.       D_i ← ((H ≪ 1) | I) & B(T[i])
21.       H ← D_i & M
22.       while H ≠ 0^m do
23.            k ← ⌊log_2(H)⌋
24.            report(k)
25.            H ← H & ∼(1 ≪ k)
```

Figure 1. The GQ-MATCHER algorithm for the string matching problem with gapped q-gram patterns

for $i = 0, \ldots, n-1$, $1 \leq k \leq |\mathcal{P}|$ and $1 \leq l \leq len(P^k)$. The algorithm computes, for each position i in T, the set D_i of the prefixes of the patterns that occur at i. From the definition of D_i it follows that the pattern P^k occurs in T at position i if and only if $(k, len(P^k)) \in D_i$. The sets D_i can be computed using the following lemma:

Lemma 1. *Let $\mathcal{P}$ and T be a set of q-gram patterns and a text of length n, respectively. Then $(k,l) \in D_i$, for $1 \leq k \leq |\mathcal{P}|$, $1 \leq l \leq len(P^k)$ and $i = 0, \ldots, n-1$, if and only if*

$$(l = 1 \text{ or } (k, l-1) \in D_{i-1-j^k_{l-1}}) \text{ and } T[i] = p^k_l.$$

To compute the sets D_i the algorithm preprocesses the set of patterns so as to obtain, for each symbol $c \in \Sigma$, the set

$$B(c) = \{(k,l) \mid p^k_l = c\},$$

of all the occurrences of c in the patterns in $\mathcal{P}$. The searching phase of the algorithm consists in iterating, for each position i, over all the elements in $B(T[i])$ and extending D_i based on Lemma 1. With $O(1)$-time set membership queries, the time complexity of the algorithm is $O(n + \alpha)$, where $\alpha = \sum_{i=0}^{n-1} |B(T[i])|$. Let $len(\mathcal{P}) = \sum_{i=1}^{|\mathcal{P}|} len(P^i)$ be the sum of the number of symbols in each pattern. The sets $B(c)$ require $\Theta(len(\mathcal{P}))$ space. Moreover, for the recursion of Lemma 1, the algorithm needs to keep the sets

D computed in the last $g_{\max}(\mathcal{P})$ iterations, and the maximum cardinality of each such set is bounded by $len(\mathcal{P})$. Hence, the space complexity is $O(g_{\max}(\mathcal{P})len(\mathcal{P}))$.

We now describe the version of the algorithm based on word-level parallelism. Let G be the set of all the distinct gap lengths in the patterns. In addition to the sets $B(c)$, we preprocess also a set $C(g)$, for each $g \in G$, defined as follows:

$$C(g) = \{(k, l) \mid j_l^k = g\},$$

for $1 \leq k \leq |\mathcal{P}$ and $1 \leq l < len(P^k)$. We encode all the sets as bit-vectors of $len(\mathcal{P})$ bits. The generic element (k, l) is mapped onto bit $\sum_{i=1}^{k-1} len(P^i) + len(P_{l-1}^k)$, where $len(P_0^k) = 0$ for any k. We denote with D_i, $\mathsf{B}(c)$ and $\mathsf{C}(g)$ the bit-vectors corresponding to the sets with the same name. We also compute two additional bit-vectors I and M, such that the bit corresponding to the element $(k, 1)$ in I and $(k, len(P^k))$ in M is set to 1, for $1 \leq k \leq |\mathcal{P}|$. We basically mark the first and the last bit of each pattern, respectively. Let H_i be the bit-vector equal to the bitwise or of the bit-vectors

$$\mathsf{D}_{i-1-g} \;\&\; \mathsf{C}(g), \tag{1}$$

for each $g \in G$. The corresponding set H_i is equal to

$$\bigcup_{g \in G} \{(k, l) \mid (k, l) \in D_{i-1-g} \wedge j_l^k = g\}.$$

The value of the bit-vector D_i can then be computed using the following bitwise operations:

$$\mathsf{D}_i \leftarrow ((\mathsf{H}_i \ll 1) \mid \mathsf{I}) \;\&\; \mathsf{B}(T[i])$$

which correspond to the relation

$$\{(k, l) \mid (l = 1 \vee (k, l-1) \in H_i) \wedge (k, l) \in B(T[i])\}.$$

To report all the patterns that match at position i it is enough to iterate over all the bits set in $\mathsf{D}_i \;\&\; \mathsf{M}$. The algorithm is depicted in Figure 3.

The bit-vector H_i can be processed in time $O(g_{\text{w-span}}\lceil len(\mathcal{P})/w \rceil)$, $1 \leq g_{\text{w-span}} \leq w$, as follows: we compute equation 1 for each word of the bit-vector separately, starting from the least significant one. For a given word with index j, we have to compute equation 1 only for each $g \in G$ such that the j-th word of $C(g)$ has at least one bit set. Each position in the bit-vector is spanned by exactly one gap, so the number of such g is at most w. Hence, if we maintain, for each word index, the list of all the distinct gap lengths that span the corresponding positions, we can compute H_i with the advertised bound, where $g_{\text{w-span}}$ is the maximum size of any such list. Observe that $g_{\text{w-span}}$ may depend also on the ordering of the patterns when more than one pattern is packed into the same word. Hence, it can be possibly reduced by finding an ordering that maps onto the same word groups of patterns that share many gap lengths.

The time complexity of the searching phase of the algorithm is then $O(ng_{\text{w-span}}\lceil len(\mathcal{P})/w \rceil + occ)$, while the space complexity is $O((g_{\max}(\mathcal{P}) + \sigma)\lceil len(\mathcal{P})/w \rceil)$.

Theorem 2. *Given a set $\mathcal{P}$ of gapped q-gram patterns and a text T of length n over Σ, all the occurrences in T of the patterns in $\mathcal{P}$ can be reported in time $O(ng_{\text{w-span}}\lceil len(\mathcal{P})/w \rceil + occ)$, where $1 \leq g_{\text{w-span}} \leq w$.*

Note that, for $g_{\text{w-span}} = o(\log w)$, this bound is a slight improvement over the more general bound for regular expressions by Bille et al [3]. This algorithm is preferable only when $g_{\text{w-span}} \ll w$ and $len(\mathcal{P})$ is small. However, it can also be used as a filter to speed up the PMA algorithm [5]. The idea is to search for the set of the prefixes of a fixed small length k of the patterns in $\mathcal{P}$ with the GQ-MATCHER algorithm and feed all the occurrences to PMA in such a way that PMA starts from the prefixes of length k. In this way we reduce the α' term in the time complexity of PMA to the number of occurrences in the text of pattern prefixes of length $\geq k$, which can be significantly better in this context.

5 Experimental results

The proposed algorithm has been experimentally validated. In particular, we compared the GQ-MATCHER algorithm and the PMA algorithm [5] with $q = 6$. The GQ-MATCHER has been implemented in the C++ programming language and compiled with the `GNU C++ Compiler 4.4`, using the options `-O3`. The source code of the PMA algorithm was kindly provided by the authors. The test machine was a 2.53 GHz Intel Xeon E5630 running Ubuntu 10.04 and running times were measured with the getrusage function. The benchmarks consisted in searching for a set of randomly generated gapped q-gram patterns on the genome sequence of 4,638,690 base pairs of *Escherichia coli* ($\sigma = 4$)[1]. Figure 2 shows the running times for searching a set of randomly generated 6-gram patterns with a fixed number of patterns equal to 50 and 100, respectively, and such that the maximum gap varies between 5 and 60. Figure 3 shows the running times for searching a set of randomly generated 6-gram patterns with a fixed maximum gap of 20 and 40, respectively, and such that the number of patterns varies between 25 and 200. We used a logarithmic scale on the y axis. The experimental results show that the new algorithm is significantly faster (up to 50 times) than the PMA algorithm in this particular scenario. The approach based on locating the occurrences of the keywords does not scale well when all the keywords, and in particular the first keyword, are of length 1. Note that in the general case of arbitrary length keywords the PMA algorithm is very fast [5].

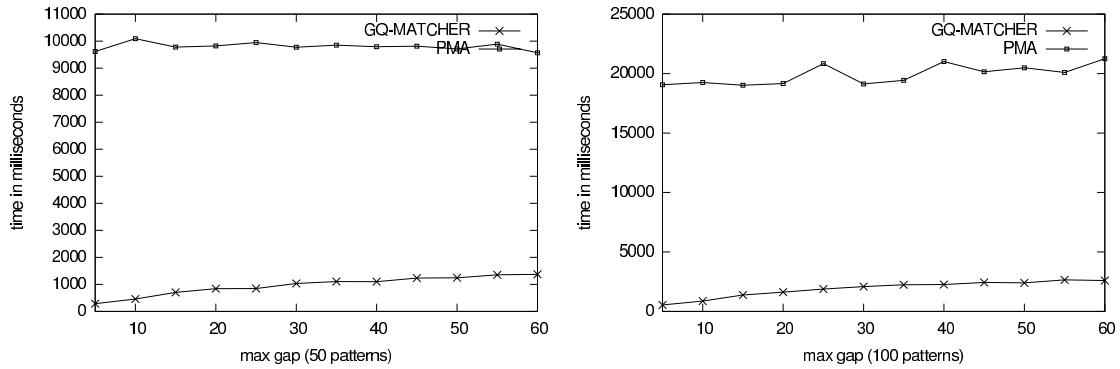

Figure 2. Experimental results on a genome sequence of *Escherichia coli* with randomly generated gapped 6-gram patterns obtained for varying gap interval with a set of 50 and 100 patterns

[1] `http://corpus.canterbury.ac.nz/`

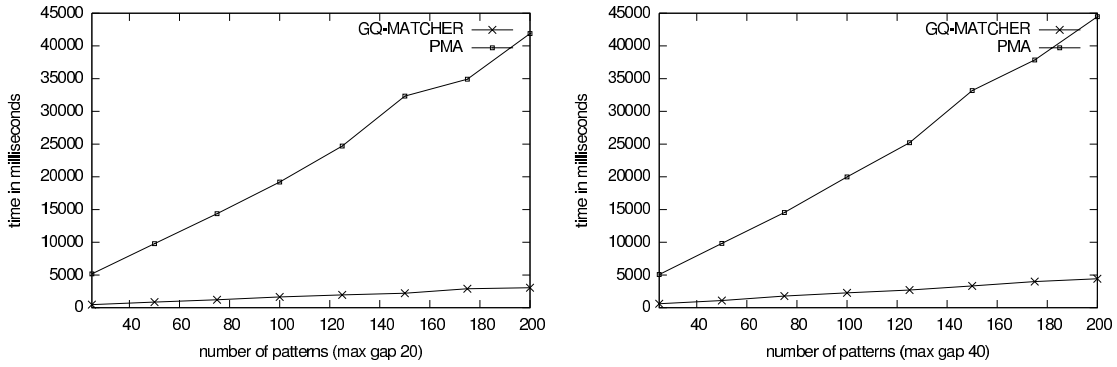

Figure 3. Experimental results on a genome sequence of *Escherichia coli* with randomly generated gapped 6-gram patterns obtained for varying number of patterns with maximum gap 20 and 40

References

1. Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
2. Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. In Edgar Chávez and Stefano Lonardi, editors, *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 385–394. Springer, 2010.
3. Philip Bille and Mikkel Thorup. Regular expression matching with multi-strings and intervals. In Moses Charikar, editor, *SODA*, pages 1297–1308. SIAM, 2010.
4. Kimmo Fredriksson and Szymon Grabowski. Nested counters in bit-parallel string matching. In Adrian Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *LATA*, volume 5457 of *Lecture Notes in Computer Science*, pages 338–349. Springer, 2009.
5. Tuukka Haapasalo, Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen. Online dictionary matching with variable-length gaps. In Panos M. Pardalos and Steffen Rebennack, editors, *SEA*, volume 6630 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2011.
6. Michele Morgante, Alberto Policriti, Nicola Vitacolonna, and Andrea Zuccolo. Structured motifs search. *Journal of Computational Biology*, 12(8):1065–1082, 2005.
7. Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003.
8. Gonzalo Navarro and Mathieu Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2004.
9. Cinzia Pizzi and Esko Ukkonen. Fast profile matching algorithms - a survey. *Theor. Comput. Sci.*, 395(2-3):137–157, 2008.
10. M. Sohel Rahman, Costas S. Iliopoulos, Inbok Lee, Manal Mohamed, and William F. Smyth. Finding patterns with variable length gaps or don't cares. In Danny Z. Chen and D. T. Lee, editors, *COCOON*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2006.
11. Eilon Sharon, Shai Lubliner, and Eran Segal. A feature-based approach to modeling protein-dna interactions. *PLoS Computational Biology*, 4(8), 2008.

Bořivoj Melichar and Multidimensional Pattern Matching

Jan Žďárek

Department of Theoretical Computer Science, Faculty of Information Technology,
Czech Technical University in Prague, Thákurova 9, 160 00 Prague 6, Czech Republic
zdarekj@fit.cvut.cz

Abstract. A short story of the results obtained by Bořivoj Melichar in multidimensional pattern matching.

1 Introduction

As a regular student at the Department of Computer Science and Engineering, I met professor Bořivoj Melichar (*Bořek*) many times. Bořek was the key person in teaching theory of languages and theory of compilers. I mastered the tools he presented to us since they are very important in practice and I needed them. But I never thought I will be really interested in any of the domains. I felt the domains are well studied and there is no potential to make any but subtle advances. My opinion has greatly changed when I attended the course *Text information systems*. I learned there the theory of languages, and specifically the theory of automata, can be used in another domain: text searching. Yet before the semester ended, I asked Bořek if it is possible to do some research oriented master thesis under his supervision and this was the moment I have become involved in this story.

2 Two-dimensional pattern matching

2.1 Matching using finite automata

Our first work was to adapt then known finite automata methods and algorithms for text matching into multiple dimensions. Meanwhile, we explored some dead-ends, I remember e.g. fractal covering of a picture which linearises it perfectly, the matching in such text is harder, however. Despite no publishable output, these efforts did not ultimately go in vain.

The results were finally published in 2005/6 [2]. The main point of this work is that we reused a pattern matching automata in a new area of application and we offered a systematic approach for describing two-dimensional pattern matching. We presented a general finite automata based approach to modelling of two-dimensional pattern matching problems. Based on the generic algorithm, two particular methods have been presented, one for 2D exact pattern matching and one for 2D approximate pattern matching using the 2D Hamming distance. For the sake of simplicity the general algorithm produces nondeterministic finite automata. This presents a problem in practice, since there is an additional step of automata determinisation, or the automaton should be simulated. Fortunately, there exists direct construction method of equivalent deterministic pattern matching automata for the exact matching case, for the approximate case we do not have such construction so we provided a simple simulation algorithm.

Festschrift for Bořivoj Melichar. J. Holub, B. W. Watson and J. Žďárek (Eds.) Published by Czech Technical University in Prague, Czech Republic

2.2 Matching using pushdown automata

With the tree pattern matching algorithm development in recent years, we realized we have seen tree-like structures many times during the previous efforts. These tree-like structures may represent many things related to processing, searching in and indexing multidimensional objects (texts).

In 2010–12 we presented a new method of indexing multidimensional text and linear searching in such index using pushdown automata [3]. Although we concentrate on two-dimensional matching, this method is easily extensible into more than two dimensions, since it is orthogonal in this aspect. The key points of our method are: a picture should be transformed into a tree, where the context of each element of the picture is preserved. The tree can be linearised into its prefix notation and a pattern matching can be performed over this representation. To keep an analogy to the one-dimensional case we started from years ago, pushdown automata constructed for this type of matching can search for a two-dimensional prefix, 2D suffix, or 2D factor of the 2D text.

For the searching, we use efficient algorithms from the area of arbology [1]. For the indexing we do not rely on any existing tree pattern matching or indexing method and we developed a new algorithm.

The pushdown automata index the 2D text and allow to search for the 2D pattern in time proportional to the size of the pattern itself, independently of the size of the text. 2D factor and suffix automata are nondeterministic. However, 2D prefix automaton is deterministic and optimal in size, since its size is proportional to the size of a 2D text. Measurements of the algorithms' performance are in manuscript [5].

2.3 Matching using linear bounded automata

In [3] and in some more detail in [6] we have shown a new notation for picture description. Its main features are: it is linear with the size of the original text and what is even more important, it includes so called back-links or back-jumps, providing the context of each element in the way similar to the tree-index representation. This seemingly simple structure in fact allows to have a neat index of the picture and its full representation (reconstructible in linear time) in one structure. It seems the pushdown automata are a bit shorthanded in this case and the natural model for this structure processing are linear bounded automata [4].

3 Conclusions

From the very beginning of my career I learned much from Bořek and I am still learning. I have spent several years doing research with him and I have found his fellowship very stimulating and pleasant. It is not only matter of research but also friendly attitude in personal contacts, the way of team building, work efficiency, and many subtle details I experienced but I am not able to name exactly.

I feel that the story has just begun. There is quite a lot of work ahead of us and I wish our cooperation with Bořek and other brilliant members of the Prague Stringology Club will continue. I conclude it is a privilege to meet, to learn from, and to work with a *real professor*.

References

1. J. Janoušek: *String suffix automata and subtree pushdown automata*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žd'árek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 160–172.
2. B. Melichar and J. Žďárek: *On two-dimensional pattern matching by finite automata*, in Proceedings of the 10th Conference on Implementation and Application of Automata, J. Farré, I. Litovsky, and S. Schmitz, eds., no. 3845 in Lecture Notes in Computer Science, Springer-Verlag, Berlin/Heidelberg, 2006, pp. 329–340, revised selected papers.
3. B. Melichar and J. Žďárek: *Tree-based 2D indexing.* J. Foudations Comp. Sc.., 22(8) Dec. 2011, pp. 1893–1907.
4. B. Melichar and J. Žďárek: *New representation of a picture index*, 2012, in preparation.
5. J. Šembera and J. Žďárek: *Tree-based 2D indexing implementation*, Jan. 2011, manuscript.
6. J. Žďárek: *Two-dimensional Pattern Matching Using Automata Approach*, PhD thesis, Czech Technical University in Prague, 2010.

Author Index

Amavi, Joshua, 153

Baker, Andrew, 102
Bannai, Hideo, 10
Bouchou-Markhoff, Béatrice, 153

Cantone, Domenico, 61
Christodoulakis, Manolis, 19
Christou, Michalis, 19
Cleophas, Loek, 46, 127
Crochemore, Maxime, 53

De Agostino, Sergio, 1
Deza, Antoine, 102

Faro, Simone, 61, 72
Flouri, Tomáš, 145
Franek, Frantisek, 102

Giaquinta, Emanuele, 61, 161

Holub, Jan, 118

Inenaga, Shunsuke, 10

Janoušek, Jan, 123

Klein, Shmuel Tomi, 125
Kobert, Kassian, 145
Kociumaka, Tomasz, 53
Kourie, Derrick G., 127

Lecroq, Thierry, 72

Mannová, Boba, 140

Pissis, Solon P., 145

Rytter, Wojciech, 53

Savary, Agata, 153
Stamatakis, Alexandros, 145
Sugimoto, Shiho, 10

Takeda, Masayuki, 10
Toopsuwan, Chalita, 53
Tyczyński, Wojciech, 53

Ukkonen, Esko, 161

Venter, Fritz, 127

Waleń, Tomasz, 53
Watson, Bruce W., 46, 127

Žďárek, Jan, 168

Festschrift for Bořivoj Melichar
Edited by Jan Holub, Bruce W. Watson 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 Kresta & Honsnejman, Bělohorská 21, Praha 6 – Břevnov, 169 00, Czech Republic