Proceedings of the Prague Stringology Conference 2015

Edited by Jan Holub and Jan Žďárek

August 2015

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

Conference Organisation

Program Committee

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

Organising Committee

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

External Referees

Loek Cleophas
Arnaud Lefebvre
Elise Prieur-Gaston

Preface

The proceedings in your hands contains a collection of papers presented in the Prague Stringology Conference 2015 (PSC 2015) held on August 24–26, 2015 at the Czech Technical University in Prague, which organizes the event. The conference focused on stringology, i.e., a discipline concerned with algorithmic processing of strings and sequences, and related topics.

The submitted papers were reviewed by the program committee subject to originality and quality. The twelve papers in this proceedings made the cut and were selected for regular presentation at the conference. In addition, this volume contains an abstract of the invited talk "A Faster Longest Common Extension Algorithm on Compressed Strings and its Various Applications" by Shunsuke Inenaga.

The Prague Stringology Conference has a long tradition. PSC 2015 is the nineteenth event of the Prague Stringology Club. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2014 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on web pages of the Prague Stringology Club. Selected contributions have been regularity published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, the International Journal of Foundations of Computer Science, and the Discrete Applied Mathematics.

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

We would like to thank all those who had submitted papers for PSC 2015 as well as the reviewers. Special thanks go to all the members of the program committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC 2015. Last, but not least, our thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2015

Jan Holub and Frantisek Franek

Table of Contents

Invited Talk

Contributed Talks

A Faster Longest Common Extension Algorithm on Compressed Strings and its Applications (*Invited talk*)

Shunsuke Inenaga

Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp

Abstract. In this talk, we introduce our recent data structure for *longest common extension (LCE)* queries on grammar-compressed strings. Our preprocessing input is a *straight-line program (SLP)* of size n describing a string w of length N, which is essentially a CFG in the Chomsky normal form generating only w. We can preprocess the input SLP in $O(n \log \log n \log N \log^* N)$ time so that later, given two variables and two positions in the strings derived by the variables, we can answer the corresponding LCE query in $O(\log N \log^* N)$ time. Our LCE data structure requires $O(z \log N \log^* N)$ words of space, where z is the size of the Lempel-Ziv 77 factorization of w. We also show several applications of our LCE data structure on SLPs.

1 Longest common extension (LCE) problem

The *longest common extension (LCE) problem* is to compute the length of the longest common prefix of two query suffixes of a string. More formally, the problem is defined as follows: Preprocess an input string w so that later, given a query pair (i, j) of positions on w, we can quickly answer the length ℓ of the longest common prefix of $w[i..|w|]$ and $w[j..|w|]$. The LCE problem often appears as important subproblems of various kinds of string processing problems, e.g., computing gapped palindromes [15] and gapped repeats [16], approximate pattern matching [4], computing runs [5], etc.

Let N denote the length of an input string w. It is well known that after preprocessing the string w in $O(N)$ time and with $O(N)$ words of space (or $O(N\omega)$ bits of space, if ω is the machine word size), the LCE of any two query suffixes can be computed in $O(1)$ time, by applying a lowest common ancestor data structure [11,23,6] to the suffix tree of w [24,10]. The $O(N)$-word space usage, however, can be problematic for massively long strings, and hence, a great deal of effort has been put towards developing more space-efficient LCE data structures.

2 LCE problem on grammar-compressed strings

In this research, we consider the LCE problem on *grammar compressed strings* which are represented by *straight-line programs (SLPs)*. An SLP for a string w is a context-free grammar in the Chomsky normal form which derives only w. Let $V = X_1, \ldots, X_n$ be the sequence of n variables of an SLP $\mathcal{S}$ which represents a string w of length N, where X_n is the last variable deriving w. The number n of variables is called the *size* of the SLP $\mathcal{S}$. We assume that V has no redundant variables, i.e., each X_u in V appears at least once in the derivation tree of X_n. On this assumption, $n \leq N$ always holds, and hence, any SLP is asymptotically never larger than the original string. Also, since every internal node of the derivation tree of any SLP has exactly

Shunsuke Inenaga: A Faster Longest Common Extension Algorithm on Compressed Strings and its Applications, pp. 1–4.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

two children, $\log_2 N \leq n$ holds. Indeed, SLPs are capable of *exponential compression* for some instances, i.e., the sizes of SLPs for highly repetitive strings can be as small as $\Theta(\log N)$.

We consider the LCE problem on SLPs in the context of *compressed string processing (CSP)* [3]: We assume that the string w is stored as an SLP $\mathcal{S}$, and $\mathcal{S}$ is given to us as an input for preprocessing. The task is to build a data structure which: (1) supports efficient LCE queries on any pair of variables of $\mathcal{S}$. Namely, given a query quartet (u, v, i, j), we are to compute the longest common prefix of $val(X_u)[i..|val(X_u)|]$ and $val(X_v)[j..|val(X_v)|]$, where $val(\cdot)$ denotes the string derived by the variable; (2) requires $n^{O(1)}$ space; and (3) can be constructed in $n^{O(1)}$ time. LCE data structures with properties (1)-(3) are of great significance, when the original string w is highly compressible. In particular, when N is as large as $\Theta(2^n)$, such LCE data structures on SLPs achieve exponential space-saving w.r.t. the uncompressed counterparts. Note that *no* algorithms which explicitly decompress the input SLP can achieve (3), since the length N of the original uncompressed string w can be as large as $\Theta(2^n)$.

A folklore LCE algorithm on SLPs is the following: Precompute the length of the decompressed string $val(X_u)$ for every variable X_u in V. This can be done in $O(n)$ total time in a simple bottom-up manner, and all the lengths can be stored with $O(n)$ words of total space (assuming the machine word size ω is at least $\log_2 N$). Then, we can simulate a traversal from the root to each leaf of the derivation tree of each variable X_u in $O(h)$ time, where h is the height of the last variable X_n. Thus, LCE query (u, v, i, j) on an input SLP can be answered in $O(h\ell)$ time, where ℓ is the answer (LCE length) to the query. Note that $\log_2 N \leq h \leq n$ always holds, and that the answer ℓ can be as large as $O(N)$.

Karpinski et al. [14] showed the first non-trivial LCE data structure on SLPs which requires $O(n^3)$ words of space and answers LCE queries of limited form $(u, v, i, 1)$ in $O(n \log n)$ time. Their data structure can be constructed in $O(n^4 \log n)$ time. Miyazaki et al. [19] proposed a data structure which requires $O(n^2)$ words of space and can answer LCE queries of limited form $(u, v, i, 1)$ in $O(n^2)$ time. Their algorithm can be extended to support LCE queries of general form (u, v, i, j) in $O(n^2 h)$ time with the same space bound [12]. Lifshits [17] showed how to construct Miyazaki et al.'s data structure in $O(n^2 h)$ time. I et al. [12] developed an LCE data structure on SLPs which requires $O(n^2)$ words of space, supports LCE queries of general form in $O(h \log N)$ time, and can be constructed in $O(n^2 h)$ time. The common basic idea to all these data structures is to virtually align the leaves of the derivation trees of two variables X_u and X_v with appropriate offsets, and compute maximal subtrees whose leaves correspond to the LCE. Bille et al. [7] proposed a randomized LCE data structure. We omit its details, since here we concentrate on deterministic LCE data structures.

2.1 A new faster LCE data structure on SLPs

In this talk, we introduce our new LCE data structure on SLPs which requires $O(z \log N \log^* N)$ words of space, and supports LCE queries of general form (u, v, i, j) on SLPs in $O(\log N \log^* N)$ time, where z is the size of the Lempel-Ziv 77 factorization [25] of the original string w. Rytter [21] showed that z is a lower bound of the size of *any* SLP representing the string w, i.e., $z \leq n$ always holds. Hence this new LCE data structure is rather small. Also, since $\log^* N$ is smaller than h, our LCE query time is always better than that of the state-of-the-art data structure by

I et al. [12]. We also show that our new LCE data structure can be constructed in $O(n \log \log n \log N \log^* N)$ time from a given SLP of size n.

The mechanism of our new LCE data structure is significantly different from the previous LCE data structures on SLPs. The new algorithm works on the trees induced by the *signature encodings* [2,1] of the strings derived by the variables, rather than on the derivation trees of the variables.

Using our faster LCE data structure, we improve the best known solutions to several important problems on SLPs, e.g. computing all palindromic substrings [18] and computing the Lyndon factorization of the original string [13].

These results are an outcome of a joint work with Takaaki Nishimoto, Tomohiro I, Hideo Bannai, and Masayuki Takeda. A full version of this work is available at [20].

3 Related work

Another line of research for space-efficient LCE data structures is to develop *succinct data structures* which use space close to the information theoretic lower bound. The longest common prefix (LCP) array for string w of length N is an array of length N which stores the lengths of the longest common prefixes of consecutive suffixes of w that are lexicographically sorted. Then, LCE queries on string w reduce to *range minimum queries (RMQs)*. Sadakane [22] proposed an RMQ data structure for an array of length N which occupies $4N + o(N)$ bits of space and answers each query in $O(1)$ time. His data structure can be constructed in $O(N)$ time with $O(N \log N)$ bis of working space. Later, Fischer and Heun showed a smaller RMQ data structure which uses only $2N + o(N)$ bits of space, answers each query in $O(1)$ time, and can be built in $O(N)$ time with $O(N)$ bits of working space. Each of these data structures is an *encoding* of the LCP array of w, namely, the LCP array is not needed for answering queries.

Yet another line of research is to find trade-offs between the space complexity and the LCE query time with a parameter τ with $1 \leq \tau \leq N$. Bille et al. [9] proposed an LCE data structure which requires $O(N/\sqrt{\tau})$ words of space, answers each LCE query in $O(\tau)$ time, and can be built in $O(N^2/\sqrt{\tau})$ time with $O(N/\sqrt{\tau})$ words of working space. Recently, Bille et al. [8] discovered a better trade-off with $O(N/\tau)$ words of space and $O(\tau)$ LCE query time. This data structure can be built in $O(N^{2+\varepsilon})$ time using $O(N/\tau)$ words of working space, where $\varepsilon > 0$ is any constant. Some randomized LCE algorithms were also proposed by these authors.

Note that all the above LCE data structures use space which is proportional to the length N of the uncompressed string w.

References

1. S. Alstrup, G. S. Brodal, and T. Rauhe: *Dynamic pattern matching*, tech. rep., Department of Computer Science, University of Copenhagen, 1998.
2. S. Alstrup, G. S. Brodal, and T. Rauhe: *Pattern matching in dynamic texts*, in SODA 2000, 2000, pp. 819–828.
3. A. Amir, G. Benson, and M. Farach: *Let sleeping files lie: Pattern matching in z-compressed files*. J. Comput. Syst. Sci., 52(2) 1996, pp. 299–307.
4. A. Amir, M. Lewenstein, and E. Porat: *Faster algorithms for string matching with k mismatches*. J. Algorithms, 50(2) 2004, pp. 257–275.
5. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *A new characterization of maximal repetitions by Lyndon trees*, in SODA 2015, 2015, pp. 562–571.

6. M. A. Bender and M. Farach-Colton: *The LCA problem revisited*, in LATIN 2000, 2000, pp. 88–94.
7. P. Bille, P. H. Cording, I. L. Gørtz, B. Sach, H. W. Vildhøj, and S. Vind: *Fingerprints in compressed strings*, in WADS 2013, 2013, pp. 146–157.
8. P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj: *Longest common extensions in sublinear space*, in CPM 2015, 2015, pp. 65–76.
9. P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj: *Time-space trade-offs for longest common extensions*. J. Discrete Algorithms, 25 2014, pp. 42–50.
10. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction*. J. ACM, 47(6) 2000, pp. 987–1011.
11. D. Harel and R. E. Tarjan: *Fast algorithms for finding nearest common ancestors*. SIAM J. Comput., 13(2) 1984, pp. 338–355.
12. T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara: *Detecting regularities on grammar-compressed strings*. Inf. Comput., 240 2015, pp. 74–89.
13. T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Faster Lyndon factorization algorithms for SLP and LZ78 compressed text*, in SPIRE 2013, 2013, pp. 174–185.
14. M. Karpinski, W. Rytter, and A. Shinohara: *An efficient pattern-matching algorithm for strings with short descriptions*. Nordic Journal of Computing, 4 1997, pp. 172–186.
15. R. Kolpakov and G. Kucherov: *Searching for gapped palindromes*. Theor. Comput. Sci., 410(51) 2009, pp. 5365–5373.
16. R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov: *Searching of gapped repeats and subrepetitions in a word*, in CPM 2014, 2014, pp. 212–221.
17. Y. Lifshits: *Processing compressed texts: A tractability border*, in CPM 2007, vol. 4580 of LNCS, 2007, pp. 228–240.
18. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto: *Efficient algorithms to compute compressed longest common substrings and compressed palindromes*. Theor. Comput. Sci., 410(8–10) 2009, pp. 900–913.
19. M. Miyazaki, A. Shinohara, and M. Takeda: *An improved pattern matching algorithm for strings in terms of straight-line programs*, in CPM 1997, 1997, pp. 1–11.
20. T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Dynamic index, LZ factorization, and LCE queries in compressed space*. CoRR, abs/1504.06954 2015.
21. W. Rytter: *Application of Lempel-Ziv factorization to the approximation of grammar-based compression*. Theor. Comput. Sci., 302(1-3) 2003, pp. 211–222.
22. K. Sadakane: *Succinct data structures for flexible text retrieval systems*. J. Discrete Algorithms, 5(1) 2007, pp. 12–22.
23. B. Schieber and U. Vishkin: *On finding lowest common ancestors: Simplification and parallelization*. SIAM J. Comput., 17(6) 1988, pp. 1253–1262.
24. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.
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.

Computing Left-Right Maximal Generic Words

Takaaki Nishimoto[1], Yuto Nakashima[1], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

Department of Informatics, Kyushu University, Japan
{takaaki.nishimoto, yuto.nakashima, inenaga, bannai, takeda}@inf.kyushu-u.ac.jp

Abstract. The *maximal generic words problem* was proposed by Kucherov et al. (SPIRE 2012). Let D be a set of documents. In this problem, given a pattern P and a threshold $d \leq |D|$, we want to compute all *right-maximal* extensions of P which occur in at least d distinct documents. They proposed an $O(n)$-space data structure which can solve this problem in $O(|P| + rocc)$ time where n is the total length of documents in D and $rocc$ is the number of right-maximal extensions of P. The data structure can be constructed in $O(n)$ time. In this paper, we propose a more generalized problem. Given a pattern P and a threshold $d \leq |D|$, we want to compute all *left-right-maximal* extensions of P which occur in at least d distinct documents. We propose an $O(n \log n)$-space data structure which can solve this problem in $O(|P| + occ \log^2 n + rocc \log n)$ time where occ is the number of left-right-maximal extensions of P.

1 Introduction

Let $D = \{T_1, \ldots, T_m\}$ be a set of strings of total n characters from an alphabet Σ, called *documents*. Kucherov et al. [8] proposed the *right-maximal generic words problem*: Given a pattern P and threshold $d \leq m$, return all maximal right extensions of P which occur in at least d distinct documents, where a right extension of P is a string which has P as a prefix. This problem is important to applications in computational biology, text mining, and text classification (see [8,3,7]). Kucherov et al. [8] solved the right-maximal generic words problem in $O(|P| + rocc)$ query time, $O(n)$ preprocessing time, and using $O(n)$ space, where $rocc$ is the number of the output right-maximal extensions. Later, Biswas et al. [3] proposed a succinct data structure of $n \log |\Sigma| + o(n \log |\Sigma|) + O(n)$ bits of space which solves the right-maximal generic words problem in $O(|P| + \log \log n + rocc)$ query time.

In this paper, we consider a more generalized problem: Given a pattern P and threshold $d \leq m$, return all maximal left-right extensions of P which occur in at least d documents, where a left-right extension of P is a superstring of P. For example, let $D = \{T_1, \ldots, T_4\}$, where $T_1 = bababaa\$_1$, $T_2 = ccabacc\$_2$, $T_3 = abaaccabab\$_3$, and $T_4 = cabacbaba\$_4$. Given a pattern $P = aba$ and threshold $d = 2$, then the answer is $\{cabac, abaa, ccaba, baba, abab\}$.

Since all right-maximal generic words of a given pattern P have P as a prefix, the right-maximal generic words problem can be solved by using the generalized suffix tree of D and segment intersection query data structure [4], in linear total space. In contrast, left-right-maximal generic words of P are superstrings of P, and hence P is not necessarily a prefix of the solutions. If we construct the generalized suffix trees of all substrings of documents in D and segment intersection query data structures, we would be able to quickly answer the left-right-maximal extensions of P. However, obviously this approach requires $\Omega(n^2)$ space. Hence, our left-right-maximal generic words problem seems more complicated than the right-maximal generic words problem.

Takaaki Nishimoto, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Computing Left-Right Maximal Generic Words, pp. 5–16.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

In this paper, we propose an $O(n \log n)$-space solution to the left-right-maximal words problem using the following data structures:

(i) A data structure which, given a string P and threshold d, finds all right extensions $x_1, \ldots, x_k$ of P satisfying the following properties: for $1 \leq i \leq k$,
   (1) x_i contains P only as a prefix.
   (2) There exists at least one left-right-maximal extension of x_i which occurs in at least d documents and contains x_i as a suffix. Note that every left-right-maximal extension of x_i is left-right-maximal extension of P.
   (3) x_i occurs in at least d distinct documents.
   The running time is $O(k \log^2 n + \mathit{rocc} \log n)$, where *rocc* is the number of answers of the right-maximal generic words problem with a given pattern P and threshold d.

(ii) A data structure which, given a string P, finds all left-right-maximal extensions of P which occur in at least d documents and contains P as a suffix in $O(\log \log n + c)$ time, where c is the number of the outputs.

Our algorithm is summarized as follows: (a) Firstly, we compute all right extensions $x_1, \ldots, x_k$ of P satisfying (1)(2)(3) using (i). (b) Secondly, for $1 \leq i \leq k$, we compute all maximal left-right extensions of P which occur in at least d distinct documents and contains x_i as a suffix using (ii). Hence we can solve our problem in $O(|P| + \mathit{occ} \log^2 n + \mathit{rocc} \log n)$ time.

2 Preliminaries

2.1 Strings

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. A prefix x and a suffix z of w are called a *proper prefix* and a *proper suffix* of w if $x \neq w$ and $z \neq w$, i.e., x and y is shorter than w, respectively. The i-th character of a string w is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$. For any string w, let w^R denote the reversed string of w, i.e., $w^R = w[|w|]w[|w| - 1] \cdots w[1]$.

Let $D = \{T_1, \ldots, T_m\}$ be a set of strings. The set of reversed strings of D is denoted by $D' = \{T_1^R, \ldots, T_m^R\}$. For any strings x and y, let $x \cdot y$ denote the concatenation of x and y.

A *right extension* of a string w is a string which has w as a prefix, a *left extension* of a string w is a string which has w as a suffix, and a *left-right extension* of a string w is a string which has w as a substring.

2.2 *d*-maximal

For any set D of documents (strings) and any string w, let $\mathit{Weight}_D(w)$ denote the number of distinct documents in D which have w as a substring. A string x is said to be *d-right-maximal* if $\mathit{Weight}_D(x) \geq d$ and $\mathit{Weight}_D(xa) < d$ for any $a \in \Sigma$. A string x is said to be *d-left-maximal* if $\mathit{Weight}_D(x) \geq d$ and $\mathit{Weight}_D(ax) < d$ for any $a \in \Sigma$. A string x is said to be *d-left-right-maximal* if x is *d-right-maximal* and *d-left-maximal*. Throughout the paper, the total length of documents in D will be denoted by n.

2.3 Computation Model

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

2.4 Tools

Generalized Suffix Trees. Let $\mathcal{T}$ be any edge labeled tree. For any node u of $\mathcal{T}$, let $str_{\mathcal{T}}(u)$ denote the string which is a concatenation of the edge labels from the root to u. We will abbreviate $str_{\mathcal{T}}(u)$ as $str(u)$ when clear from the context.

A *generalized suffix tree* of a set of strings is the suffix tree [11] that contains all suffixes of all the strings in the set. We denote a *generalized suffix tree* of D by GST_D. We define some notations and additional information of GST_D and $GST_{D'}$.

Let V_{GST_D} be the set of nodes of GST_D. The subtree rooted at node u is denoted by $GST_D(u)$. For any string x, let $L(x)$ be the node u which is the highest node in V_{GST_D} s.t. x is a prefix of $str(u)$. We denote the locus of x in $GST_{D'}$ by $L'(x)$. As in the previous work [8,7], we assume that $L(x)$ and $L'(x)$ for a given string x can be computed in $O(|x|)$ time using GST_D and $GST_{D'}$, respectively[1]. Let $weight(u)$ be the number of distinct documents in D which have $str(u)$ as a substring, and let $maxchild(u) = \max\{weight(v) \mid v \text{ is a child of } u\}$.

Tools on Trees. For any nodes $u, v \in V_{GST_D}$, let $LCA(u, v)$ denote the lowest common ancestor (LCA) of u and v. We can preprocess the tree in linear time so that for any nodes u, v, $LCA(u, v)$ can be computed in constant time (e.g. [1]). For any node $u \in V_{GST_D}$ and integer $d \geq 0$, let $LA(u, d)$ denote the depth-d ancestor (*level ancestor*) of node u in GST_D. We can preprocess the tree in linear time so that for any node u and integer d, we can compute $LA(u, d)$, in constant time (e.g. [2]).

Segment Intersection Query. A horizontal segment $([x, x'], y)$ on a 2D plane (resp. vertical segment $(x, [y, y'])$) is a line connecting points (x, y) and (x', y) (resp. points (x, y) and (x, y')). We say that a vertical segment $p = (x_p, [y_p, y'_p])$ *stabs* a horizontal segment $q = ([x_q, x'_q], y_q)$ if $x_q \leq x_p \leq x'_q$ and $y_p \leq y_q \leq y'_p$. *Segment Intersection Queries* for a horizontal segment set $\mathcal{S}$ are: given a vertical segment p, return the subset of segments of $\mathcal{S}$ that p stabs. There exist many data structures for this segment intersection queries [5,6,4]. In this paper, we use the data structure that occupies $O(|\mathcal{S}|)$ space and supports segment intersection queries in $O(\log\log|\mathcal{S}| + k)$ time, where k is the output size [4]. Next, suppose that each segment in $q \in \mathcal{S}$ is associated with an integer weight $w(q) \geq 0$. *Segment Intersection Sum Queries* are: given a vertical segment p, return the total sum of weights of the segments in $\mathcal{S}$ that p stabs. Since segment intersection sum queries are a special case of *rectangle intersection sum queries*, there exists a data structure occupies $O(|\mathcal{S}| \max\{1, \log \frac{W}{|\mathcal{S}|}\})$ space and supports segment intersection sum queries in $O(\log|\mathcal{S}|)$ time, where $W = \sum_{q \in \mathcal{S}} w(q)$ [10]. Similarly, we define *Segment Intersection Count Queries* for a horizontal segment set $\mathcal{S}$. This is a special case of the segment intersection sum query where $w(q) = 1$ for every segment $q \in \mathcal{S}$. Hence there exists a data structure occupies that $O(|\mathcal{S}|)$ space and supports segment intersection count queries in $O(\log|\mathcal{S}|)$ time.

[1] If $|\Sigma|$ is constant, then clearly $L(x)$ and $L'(x)$ can be computed in $O(|x|)$ time. If $|\Sigma|$ is not constant, these can be computed in expected $O(|x|)$ time using hashing.

3 Problem and Properties

In this paper, we consider the following problem.

Problem 1. Let $D = \{T_1, \ldots, T_m\}$ be a set of documents. Given a pattern P and positive integer d ($\leq m$), compute all d-left-right-maximal extensions of P.

Let $Ans_D(P, d)$ be the set of answers to Problem 1. In Section 3.1, we show a relation between $Ans_D(P, d)$ and V_{GST_D}. In this paper, we output answers as a set of nodes in $GST_{D'}$.

3.1 Relation between Answers and GST

Lemma 2. *For any $z \in Ans_D(P, d)$, there exists a node $u \in V_{GST_D}$ s.t. $str(u) = z$.*

Proof. Since z is a substring of some document, we can traverse GST_D from the root with z. Because $z \in Ans_D(P, d)$, the number of occurrences of z and za in D are different for any $a \in \Sigma$. Thus there exists a node u s.t. $str(u) = z$. □

The following corollary can be easily obtained in a similar way.

Corollary 3. *For any $z \in Ans_D(P, d)$, there exists a node $u' \in V_{GST_{D'}}$ s.t. $str(u')^R = z$.*

Let Occ be the set of nodes $u' \in V_{GST_{D'}}$ s.t. $str(u')^R \in Ans_D(P, d)$. It follows from the above corollary that Occ is the set of nodes in $V_{GST_{D'}}$ that represent all occ answers to Problem 1. In this paper, we will compute Occ as an output.

Our main idea is the following. First, we choose a node $u \in V_{GST_D(L(P))}$. This node represents a (not necessarily maximal) *right extension* of P. Second, we compute the nodes $u' \in V_{GST_{D'}(L'(str(u)^R))}$ s.t. $u' \in Occ$. By the condition $u' \in Occ$, it is clear that $weight(u') \geq d$ and $maxchild(u') < d$ holds, and hence these nodes represent d-left-right-maximal extensions of P (see also Fig. 1). Thus, if we conduct the above procedure for all nodes in $u \in V_{GST_D(L(P))}$, we can obtain all solutions to Problem 1.

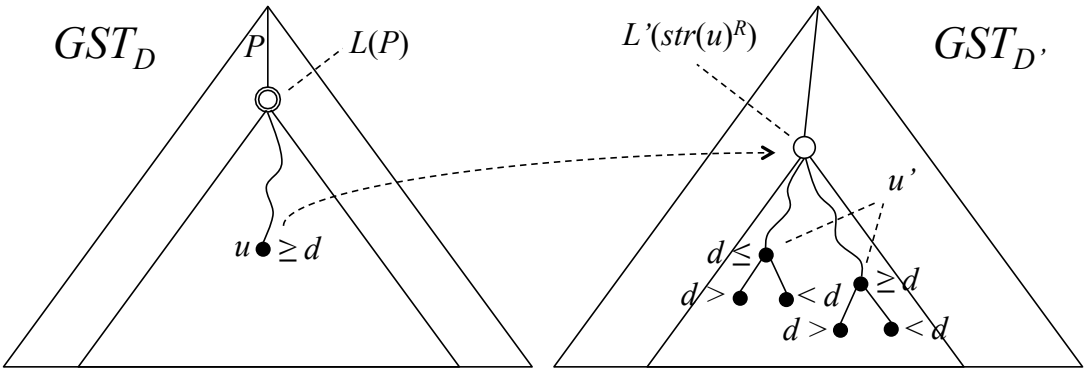


Figure 1. This is a conceptual diagram of our main idea.

In what follows, Section 3.2 characterizes the set of nodes in $GST_{D'}$ which represent (a subset of) the answers to the problem w.r.t. a given *right extension* of query pattern P. Note that the strings represented by these nodes have the *right extension* of P as a suffix. Then, Section 3.3 characterizes a subset of *right extensions* of P which are sufficient to compute all d-left-right-maximal extensions of P without duplicates.

3.2 *d*-Left-Right-Maximal Extensions of a Given Right Extension

First, we define a set $cand(u)$ for any node $u \in V_{GST_D}$ that represents *left extensions* of $str(u)$ which are d-left-maximal. Let $\ell'_u = L'(str(u)^R) \in V_{GST_{D'}}$, and

$$cand(u) = \{u' \mid u' \in V_{GST_{D'}(\ell'_u)}, weight(u') \geq d \text{ and } maxchild(u') < d\}.$$

We also define $Cand(V) = \cup_{u \in V} cand(u)$ for each $V \subseteq V_{GST_D}$. In our algorithm, given a *right extension* x of P that occurs in at least d distinct documents, we compute d-left-maximal extensions of x. Let $REx = \{u \mid u \in V_{GST_D(L(P))} \text{ and } weight(u) \geq d\}$. Figure 2 gives examples of some definitions.

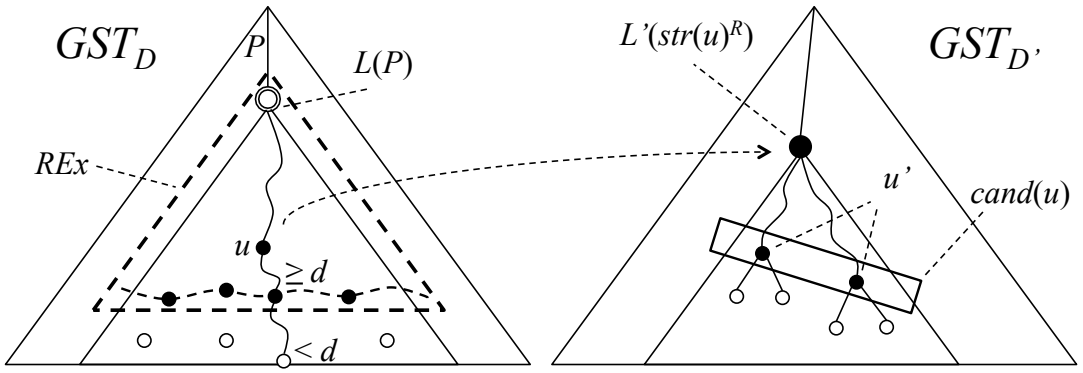


Figure 2. This figure shows examples of $cand(u)$ and REx. The black circles represent nodes s.t. its weight is larger than d. The white circles represent nodes s.t. its weight is strictly smallar than d.

Lemma 4. *$Cand(REx) \supseteq Occ$.*

Proof. Let u' be a node in Occ and let $z = str(u')^R \in Ans_D(P, d)$. Since $z \in Ans_D(P, d)$, $weight(u') \geq d$ and $maxchild(u') < d$ hold. On the other hand, there exists a node $w \in GST_D$ s.t. $z = str(w)$ by Lemma 2. Let z_1 be a suffix of z which has P as a prefix. Then there is a node $u \in GST_D$ s.t. $z_1 = str(u)$. It also holds that $weight(u) \geq d$, and z_1 has P as a prefix, so $u \in REx$. Since $str(u)^R$ is a prefix of z^R, $u' \in cand(u)$. Thus $u' \in Cand(REx)$. Therefore this lemma holds. □

Any string which is represented by a node in $Cand(REx)$ is guaranteed only to be d-left-maximal in D. Thus, there may exist a node in $Cand(REx) \cap \overline{Occ}$, where $\overline{Occ} = V_{GST_{D'}} - Occ$. To remove such nodes from $Cand(REx)$, the following lemma characterizes such nodes.

Lemma 5. *For any $u' \in Cand(REx) \cap \overline{Occ}$, there exists a node $v' \in Occ$ s.t. $str(v')^R$ has $str(u')^R$ as a proper prefix.*

Proof. By the definition of $Cand(REx)$, it is clear. □

By using the above lemma, we want to remove the nodes in $Cand(REx) \cap \overline{Occ}$ from $Cand(REx)$. For any node $u' \in V_{GST_{D'}}$ and any character c, let $FC(u', c)$ be the number of distinct documents in D which have $str(u')^R \cdot c$ as a substring. We define $MFC(u') = \max\{FC(u', c) \mid \forall c \in \Sigma\}$. $MFC(u')$ represents the maximum number of strings in D which have $str(u')^R \cdot c$ as a substring for any character c. We use $MFC(u')$ to remove nodes which are not in Occ from $Cand(REx)$.

Lemma 6. *For any node $u' \in cand(u)$ for some $u \in REx$, $MFC(u') \geq d$ iff $u' \notin Occ$.*

Proof. ($\Rightarrow$). By the definition of $MFC(u')$, $Weight_D(str(u')^R \cdot c) \geq d$ for some $c \in \Sigma$ holds. Thus $u' \notin Occ$. ($\Leftarrow$). It is clear from Lemma 5. □

Corollary 7. *For any node $u' \in cand(u_1)$ for some $u_1 \in REx$, If $MFC(u') \geq d$, then there exists a node u_2 which is a descendant of u_1 s.t. $cand(u_2) \cap Occ \neq \phi$.*

Proof. By Lemmas 5, 6, there exists $z = str(u')^R \cdot x \in Ans_D(P, d)$ for some $x \in \Sigma^+$. From Lemma 2, there is a node $w \in GST_D$ s.t. $str(w) = z$. Thus there is also a node u_2 s.t. $str(u_2) = str(u_1) \cdot x$. It is clear that u_2 is a descendant of u_1. □

Now we define a new set $Cand_1(REx)$ of nodes in $GST_{D'}$ s.t. $Cand'_1(REx) = Occ$. For any $u \in V_{GST_D}$, let

$$cand_1(u) = \{u' \mid u' \in V_{GST_{D'}(\ell'_u)}, weight(u') \geq d, maxchild(u') < d \text{ and } MFC(u') < d\}.$$

We define $Cand_1(V) = \cup_{u \in V} cand_1(u)$ for any $V \subseteq V_{GST_D}$.

Lemma 8. *$Cand_1(REx) = Occ$.*

Proof. It is clear from Lemmas 5, 6. □

3.3 Meaningful Right Extensions of P

Let $REx = \{u_1, \ldots, u_h\}$. By Lemma 8, $Cand_1(REx)$ and Occ are equivalent, but $|cand_1(u_1)| + \cdots + |cand_1(u_h)| \geq |Occ|$ holds. The following lemma characterizes this situation.

Lemma 9. *Let u_1 be a node in REx s.t. P occurs in $str(u_1)$ at least two times. For any node $u_2 \in REx$ s.t. $str(u_2)$ is a proper suffix of $str(u_1)$, $cand_1(u_1) \subseteq cand_2(u_2)$.*

Proof. For any $u' \in cand_1(u_1)$, $str(u_2)$ is a suffix of $str(u')^R$. Thus $str(u')^R \in cand_1(u_2)$. Therefore this lemma holds. □

We define a new set REx_1 s.t. $REx_1 \subseteq REx$. Let $REx_1 = \{u \mid u \in V_{GST(L(P))}$, $weight(u) \geq d$, and P occurs in $str(u)$ only as a prefix $\}$. By the above lemma, the following lemma holds.

Lemma 10. *$Cand_1(REx_1) = Occ$ and $\sum_{u \in REx_1} |cand_1(u)| = |Occ|$ hold.*

Proof. By Lemmas 8, 9, $Cand_1(REx_1) = Occ$ holds. Let u_1 and u_2 be elements of REx_1 s.t. $u_1 \neq u_2$. We assume $|str(u_1)| \leq |str(u_2)|$. By the definition of REx_1, $str(u_1)$ is not a suffix of $str(u_2)$. Thus $cand_1(u_1) \cap cand_1(u_2) = \phi$. Therefore $\sum_{u \in REx_1} |cand_1(u)| = |Occ|$ holds. □

Clearly, there may exist a node $u \in REx_1$ s.t. $cand_1(u) = \phi$. From Lemma 10, we do not want to compute $cand_1(u)$ for such $u \in REx_1$. Let $REx_2 = \{u \mid cand_1(u) \neq \phi\}$. For any $u \in REx_2$, $cand_1(u) \neq \phi$, $\cup_{u \in REx_2} cand_1(u) = Ans_D(P, d)$ and $cand_1(u_1) \cap cand_1(u_2)$ for any u_1 and u_2.

In the rest of this section, we show some lemmas which are useful to compute REx_2 efficiently. For any $u \in REx_1$, let $r'(u)$ be a node in $GST_{D'}$ s.t. $str(r'(u)) = str(u)^R$ ($r'(u)$ may be an *implicit node*). We define T_u as a tree which is a subgraph of $GST_{D'}(r'(u))$ s.t. the root is $r'(u)$ and leaves are all nodes in $cand(u)$. In fact, T_u

represents left extended strings of $str(u)$. Figure 3 shows an example of T_u. Let $Leaf(T_u)$ be a set of all leaves in T_u, and $size(T_u) = \sum_{v \in Leaf(T_u)} |str(v)|$. Then $size(T_{u_1}) - size(T_{u_2}) \geq 0$ for any u_1 and u_2 in REx_1 s.t. u_2 is a child of u_1 holds. To prove this, we show T_{u_2} can be superimposed on T_{u_1} for any u_1 and u_2 in REx_1 s.t. u_2 is a descendant of u_1. If there exists a node v in T_{u_1} s.t. $str(w)$ is a prefix of $str(v)$ for each leaf w in T_{u_2}, T_{u_2} can be superimposed on T_{u_1}.

Lemma 11. *Let u_1 and u_2 be nodes in REx_1 s.t. u_2 is a descendant of u_1. Then T_{u_2} can be superimposed on T_{u_1}.*

Proof. Let w be a leaf of T_{u_2}. There exists a node $u'_2 \in GST_{D'}$ s.t. $str(u'_2) = str(r'(u_2)) \cdot str(w)$. Since $str(r'(u_1))$ is a suffix of $str(r'(u_2))$, there exists a node $u'_1 \in GST_{D'}$ s.t. $str(u'_1) = str(r'(u_1)) \cdot str(w)$. Thus there also exists a node in T_{u_1} which corresponds to u'_1. □

Because of Lemma 11, $size(T_{u_1}) - size(T_{u_2}) \geq 0$ for any u_1 and u_2 in REx_1 s.t. u_2 is a child of u_1.

The following lemma shows a relation between $cand_1(u)$ and T_u. By using this lemma, we can determine whether $cand_1(u) = \phi$ or not. Now, let $RMax$ be the set of nodes in V_{GST_D} which are d-right-maximal extensions of P. In other words, $RMax$ is the set of answers to the *maximal generic words problem* in [8] for a given pattern P. Let G be the tree which is a subgraph of $GST_D(L(P))$ of which the root is $L(P)$ and the leaves are $RMax$.

Lemma 12. *Let u_1 and u_2 be nodes in REx_1 s.t. u_2 is a child of u_1 and u_2 have no siblings in G. Then $size(T_{u_1}) - size(T_{u_2}) > 0$ iff $cand_1(u_1) \neq \phi$.*

Proof. ($\Rightarrow$). Since $size(T_{u_1}) - size(T_{u_2}) > 0$, there exists some node $u' \in cand(u_1)$ s.t. $Weight_D(str(u')^R \cdot str(u_2)[|str(u_1)| + 1..|str(u_2)|]) < d$. By the definition of $cand$, $str(u')^R$ is *d-left-maximal*. Since u_2 is a child of u_1 and u_2 have no siblings in G, $str(u')^R$ is *d-right-maximal*. Thus $u' \in cand_1(u_1)$. ($\Leftarrow$). We assume $u' \in cand_1(u_1)$. Then there exists a node $w \in V_{T_{u_1}}$ s.t. $str(w) = str(u')[|str(u_1)| + 1..|str(u')|]$. Since $u' \in Occ$, $Weight_D(str(w)^R \cdot str(u_2)) < d$. Thus there doesn't exist a node $v' \in V_{T_{u_2}}$ s.t. $str(v) = str(w)$. Therefore $size(T_{u_1}) - size(T_{u_2}) > 0$. □

Figure 3 shows an example of Lemmas 11, 12.

To use the above lemma, we divide G into paths $\pi_1, \ldots, \pi_s$ s.t. any node which is a child of some node in the same path has no sibling for each path. The paths are defined as follows.

- Each node in G belongs to some path.
- The highest node of each path is a child of a *branching node*.
- The lowest node of each path is a *branching node* or a leaf.
- The other nodes are a *non-branching node*.

Figure 4 shows an example of G and paths.

We compute a node in REx_2 by binary search on each path.

In the rest of this section, we show how to check whether the number of occurrences of P in $str(u)$ is one or more than one. For any node $u \in V_{GST_D}$, let $preord(u)$ be the preorder traversal rank in GST_D, and $[beg(u), end(u)]$ be the interval of the preorder traversal rank of $GST_D(u)$. Obviously, $beg(u) = preord(u)$ holds. Let SA_x be the *suffix*

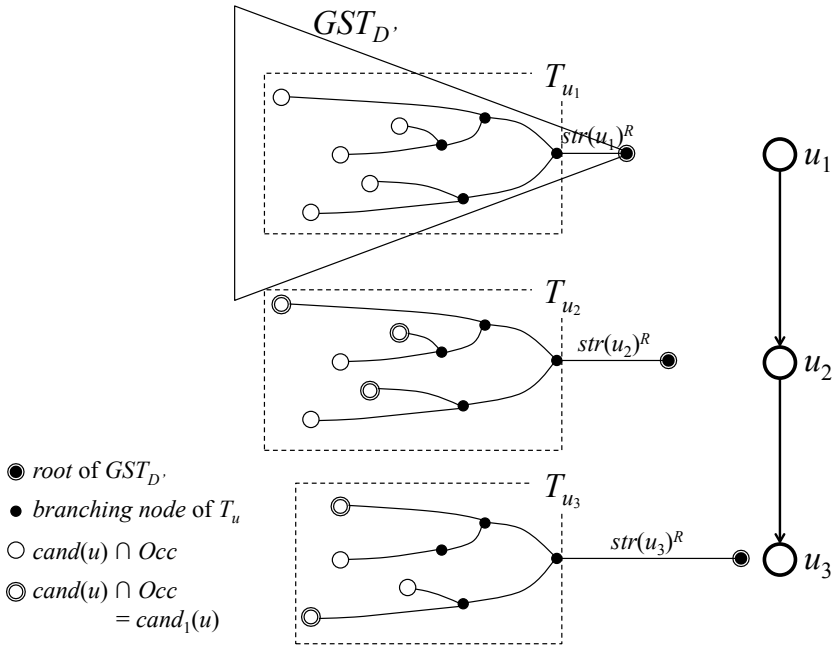


Figure 3. An example of T_{u_1}, T_{u_2}, T_{u_3} s.t. u_1, u_2 and u_3 are successive and non-branching nodes in G. In this example, $size(T_{u_1}) = size(T_{u_2}) > size(T_{u_3})$ since $cand_1(u_1) = \phi$ and $cand_1(u_2) \neq \phi$.

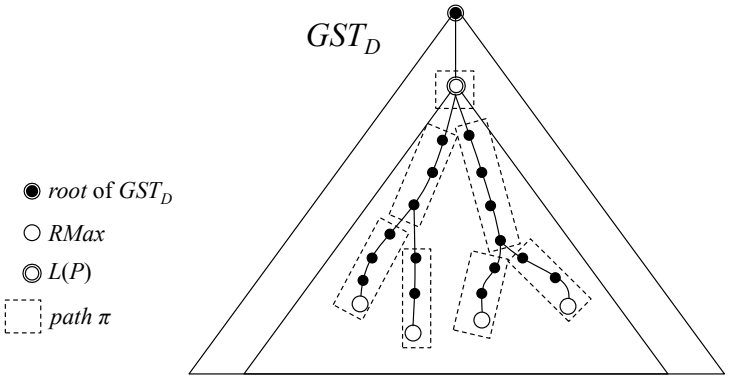


Figure 4. An example of G and its divided paths.

array [9] of a string x, and k be the integer s.t. $SA_{str(u)}[k] = 1$. Let u_s be the node in GST_D s.t. $str(u_s) = str(u)[SA_{str(u)}[k-1]..|str(u)|]$. Let u_ℓ be the node in GST_D s.t. $str(u_\ell) = str(u)[SA_{str(u)}[k+1]..|str(u)|]$. At each node u, we store $[beg(u), end(u)]$ and pointers to u_s and u_ℓ.

Lemma 13. *Let u be a node in $GST_D(L(P))$. $preord(u_s) < beg(L(P)) \leq end(L(P)) < preord(u_\ell)$ iff $str(u)$ has only one occurrence of P (P is a prefix of $str(u)$.).*

Proof. ($\Rightarrow$) Since $preord(u_s) < beg(L(P))$, $str(u_s)$ does not have P as a prefix. Since $end(L(P)) < preord(u_\ell)$, $str(u_\ell)$ does not have P as a prefix. By the definition of u_s and u_ℓ, $str(u)$ have P only as a prefix.

($\Leftarrow$) Since $str(u)$ have P only as a prefix, $str(u_s)$ and $str(u_\ell)$ do not have P as a prefix. By the definition of u_s and u_ℓ, $preord(u_s) < beg(L(P)) \leq end(L(P)) < preord(u_\ell)$ holds. □

4 Algorithm

In this section, we show how to compute *Occ*. We use the lemmas in the previous section. In Section 4.1, we show how to compute $cand_1(u)$ for a given node $u \in REx_2$. In Section 4.2, we show how to compute REx_2. Finally in Section 4.3, we summarize our algorithm.

4.1 Computing $cand_1(u)$

First, we show how to compute $cand_1(u)$ for a given node $u \in REx_1$.

Lemma 14. *There exists a data structure which can compute* $cand_1(u)$ *for any node* $u \in REx_1$ *and any integer* d *in* $O(\log\log n + |cand_1(u)|)$ *time. The size of the data structure is* $O(n)$.

Proof. For each node $u' \in V_{GST_{D'}}$, we represent u' as a horizontal segment $([\max\{maxchild(u'), MFC(u')\} + 1, weight(u')], preord(u'))$. Let $(d, [beg(\ell'_u), end(\ell'_u)])$ be a vertical segment. We use *Segment Intersection Query* for a set of the above horizontal segments [4]. Then a returned horizontal segment corresponds to a node $u' \in GST_{D'}$ s.t. $\max\{maxchild(u'), MFC(u')\} \leq d \leq weight(u')$ and $beg(\ell'_u) \leq preord(u') \leq end(\ell'_u)$. By the definition of $cand_1(u)$, $u' \in cand_1(u)$. Clearly, the number of horizontal segments is $O(n)$. Therefore this lemma holds. □

Using the data structure of the above lemma for a set of horizontal segments, we can compute $cand_1(u)$ for any node $u \in REx_1$. Figure 5 shows an example.

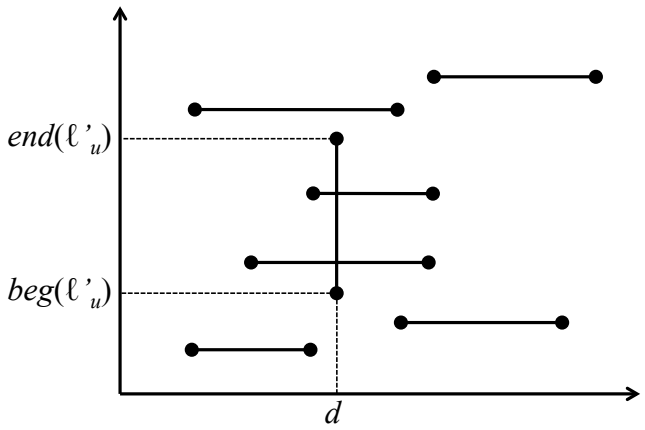


Figure 5. Two horizontal segments which are stabbed by the vertical segment correspond to nodes in $cand_1(u)$.

4.2 Computing REx_2

Second, we show how to compute REx_2.

Lemma 15. *There exists a data structure which can compute* REx_2 *for any node* $L(P) \in V_{GST_D}$ *and any integer* d *in* $O(|REx_2|\log^2 n + |RMax|\log n)$ *time, where* P *is a given string. The size of the data structure is* $O(n\log n)$ *space.*

To prove this lemma, we show some other lemmas. In our algorithm, we use *binary search* based on Lemma 12. The following lemma shows how to compute $size(T_u)$ for any node $u \in REx_1$.

Lemma 16. *For any node $u \in REx_1$, there exists a data structure which can compute $size(T_u)$ in $O(\log n)$ time. The size of the data structure is $O(n \log n)$.*

Proof. By the definition, $size(T_u) = \sum_{u' \in cand(u)} |str(u')| - |cand(u)| \times |str(u)|$. To compute the first term, we use *Segment Intersection Sum Query* as follows. For each node $u' \in V_{GST_{D'}}$, we represent u' as a horizontal segment $([maxchild(u') + 1, weight(u')], preord(u'))$. We define its weight as $|str(u')|$. Let $(d, [beg(\ell'_u), end(\ell'_u)])$ be a vertical segment. Then we can compute the first term in $O(\log n)$ time. Since the sum of the weights is clearly $O(n^2)$, The size of a data structure is $O(n \log n)$ space. On the other hand, we can compute $|cand(u)|$ in $O(\log n)$ time by using *Segment Intersection Count Query* for the above set of horizontal segments and a vertical segment. So we can compute $size(T_u)$ in $O(\log n)$ time with $O(n \log n)$-space data structure. □

To use Lemma 12, we need to compute paths of G. Hence we show that we can compute all branching nodes of G from $RMax$ by the following two lemmas.

Second, we compute branching nodes in G. They are also the lowest node of each path.

Lemma 17. *Given $RMax$, we can compute all branching nodes of G in $O(|RMax| \log |RMax|)$ time.*

Proof. We sort $RMax$ by preorder rank in $O(|RMax| \log |RMax|)$ time. Note that any branching node of G is the lowest common ancestor of two leaf nodes or branching nodes of G. Hence we can compute all branching nodes of G in $O(|RMax|)$ time from sorted $RMax$ by LCA query. □

The following corollary is true by Lemma 13.

Corollary 18. *Given $L(P)$ and a node $u \in V_{GST_D(L(P))}$, we can check in constant time whether $str(u)$ contains P only as a prefix.*

Lemma 19. *Let $\pi = u_1, \ldots, u_k$ denote a path on G s.t. u_1 is a non branching node and the parent node is a branching node of G, u_k is a leaf node or branching node of G and $u_2, \ldots, u_{k-1}$ are non branching nodes of G. Given u_1 and u_k, we can compute all nodes in REx_2 on π in $O((\alpha \log k + 1) \log n)$ time using the data structure of size $O(n \log n)$, where α is the number of the output nodes.*

Proof. Note that we can access any node on π in constant time by level ancestor query. First, we compute the maximum integer $c \leq k$ such that P occurs in $str(u_c)$ only as a prefix by Corollary 18 in $O(\log k)$ time. Second, we check whether there exists at least one node in REx_2 on $u_1, \ldots, u_c$ by comparing $size(T_{u_1})$ and $size(T_{u_c})$ in $O(\log n)$ time. Hence we use the data structures of size $O(n \log n)$ space of Lemma 16. Note that we can compute $r'(u)$ for $u \in V_{GST_D}$ in constant time by preprocessing D in linear space. From Lemma 12, if $size(T_{u_1}) - size(T_{u_c}) > 0$, we compute all nodes in REx_2 on $u_1, \ldots, u_c$ in $O((\alpha \log c + 1) \log n)$ time by binary search. □

Proof of Lemma 15 is the following.

Proof. By Lemma 17. we can compute all branching nodes and leaf nodes of G in $O(|RMax| \log |RMax|)$ time from $RMax$. Note that we can compute $RMax$ in $O(|RMax|)$ time from $L(P)$ using the data structures of size $O(n)$ [8]. By Lemma 19, we can compute all nodes in REx_2 on each path of G in $O((\alpha \log k + 1) \log n)$ time. Hence we can compute REx_2 in $O(|REx_2| \log^2 n + |RMax| \log n)$ time. □

4.3 Overall Complexity

Theorem 20. *There exists a data structure which can solve Problem 1 in* $O(|P| + |Occ| \log^2 n + |RMax| \log n)$ *time. The size of the data structure is* $O(n \log n)$.

Proof. We compute $L(P)$ in $O(|P|)$ time by traversing GST_D. Then we can compute REx_2 by Lemma 15. Finally we compute $cand_1(u)$ for any $u \in REx_2$ by Lemma 14. So the total time complexity is $O(|P| + |Occ| \log^2 n + |RMax| \log n)$. The space requirement of the data structure is $O(n \log n)$. □

5 Conclusion and Future Work

We proposed an $O(n \log n)$-space data structure which can solve the *left-right maximal generic words problem* in $O(|P| + |Occ| \log^2 n + |RMax| \log n)$ time.

Our future work includes the following.

- Can we solve the left-right maximal generic words problem more efficiently? A difficulty of this problem is that a given pattern can be extended to *both* directions.
- When we are given a single text string (a single document), a pattern P, and a threshold d on the number of occurrences of P in the text, is there a simpler algorithm to find the left-right maximal words for this special case?
- In this paper we only considered the maximal generic word problem. In [8], they also considered the minimal discriminating words problem. So the minimal discriminating words problem for the left-right extensions of a given pattern P is also interesting.

References

1. M. A. Bender and M. Farach-Colton: *The LCA problem revisited*, in LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10–14, 2000, Proceedings, G. H. Gonnet, D. Panario, and A. Viola, eds., vol. 1776 of Lecture Notes in Computer Science, Springer, 2000, pp. 88–94.
2. M. A. Bender and M. Farach-Colton: *The level ancestor problem simplified.* Theor. Comput. Sci., 321(1) 2004, pp. 5–12.
3. S. Biswas, M. Patil, R. Shah, and S. V. Thankachan: *Succinct indexes for reporting discriminating and generic words*, in String Processing and Information Retrieval – 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20–22, 2014. Proceedings, E. S. de Moura and M. Crochemore, eds., vol. 8799 of Lecture Notes in Computer Science, Springer, 2014, pp. 89–100.
4. T. M. Chan: *Persistent predecessor search and orthogonal point location on the word RAM.* ACM Transactions on Algorithms, 9(3) 2013, p. 22.
5. B. Chazelle: *Filtering search: A new approach to query-answering.* SIAM J. Comput., 15(3) 1986, pp. 703–724.
6. M. Edahiro, K. Tanaka, T. Hoshino, and T. Asano: *A bucketing algorithm for the orthogonal segment intersection search problem and its practical efficiency.* Algorithmica, 4(1) 1989, pp. 61–76.

7. P. Gawrychowski, G. Kucherov, Y. Nekrich, and T. A. Starikovskaya: *Minimal discriminating words problem revisited*, in String Processing and Information Retrieval – 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7–9, 2013, Proceedings, O. Kurland, M. Lewenstein, and E. Porat, eds., vol. 8214 of Lecture Notes in Computer Science, Springer, 2013, pp. 129–140.
8. G. Kucherov, Y. Nekrich, and T. A. Starikovskaya: *Computing discriminating and generic words*, in String Processing and Information Retrieval – 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21–25, 2012. Proceedings, L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, eds., vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 307–317.
9. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
10. C. Sheng and Y. Tao: *New results on two-dimensional orthogonal range aggregation in external memory*, in Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12–16, 2011, Athens, Greece, M. Lenzerini and T. Schwentick, eds., ACM, 2011, pp. 129–139.
11. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symposium on Switching and Automata Theory, Institute of Electrical Electronics Engineers, New York, 1973, pp. 1–11.

Combinatorics of the Interrupted Period

Adrien Thierry

Advanced Optimization Laboratory
McMaster University, Hamilton, Ontario, Canada
adrien.thierry@gmail.com

Abstract. This article is about discrete periodicities and their combinatorial structures. It presents and describes the unique structure caused by the alteration of a pattern in a repetition. Those alterations of a pattern arise in the context of double squares and were discovered while working on bounding the number of distinct squares in a string. Nevertheless, they can arise in other phenomena and are worth being presented on their own.

Keywords: string, period, primitive string, factorization

If x is a primitive word, and x_1 a prefix of x, the sequence $x^n x_1 x^m$ has a singularity: it has a periodic part of period x, an interruption, and a resumption of the pattern x. That interruption creates a different pattern, one that does not appear in x^n. The goal of this article is to unveil that pattern.

1 Preliminaries

In this section, we introduce the notations and present a simple property and two of its corollaries. These observations are not complicated, but their proofs introduce the technique used in the proof of the main theorem, Theorem 7, and allow for a clear understanding of the phenomenon described there.

We first fix some notations. An *alphabet* A is a finite set. We call *letters* the elements of A. If $|A| = 2$, the words are referred to as binary and are used in computers. Another well known example for $|A| = 4$ is DNA.
A vector of A^n is a *word* w of length $|w| = n$, which can also be presented under the form of an array $w[1 \dots n]$. Two words are *homographic* if they are equal to each other. If $x = x_1x_2x_3$ for non-empty words x_1, x_2 and x_3, then x_1 is a *prefix* of x, x_2 is a *factor* of x, and x_3 is a *suffix* of x (if both the prefix and the suffix are non empty, we refer to them as proper). We define *multiplication* as concatenation. In english, *breakfast* = *break* $\cdot$ *fast*. In a traditional fashion, we define the n^{th} *power* of a word w as n time the multiplication of w with itself. A word x is *primitive* if x cannot be expressed as a non-trivial power of another word x'.
A word $\tilde{x}$ is a *conjugate* of x if $x = x_1x_2$ and $\tilde{x} = x_2x_1$ for non-empty words x_1 and x_2. The set of conjugates of x together with x form the conjugacy class of x which is denoted $Cl(x)$.
A factor $x, |x| = n$ of w has *period* p if $x[i] = x[i + |p|], \forall i \in [1, \dots, n - |p|]$.
The *number of occurrences* of a letter c in a word w is denoted $n_c(w)$, the *longest common prefix* of x and y as $lcp(x, y)$, while $lcs(x, y)$ denotes the *longest common suffix* of x and y (note that $\mathrm{lcs}(x, y)$ and $\mathrm{lcp}(x, y)$ are words).

The properties presented next rely on a simple counting argument. If the proofs are not interesting in themselves, they still allow for meaningful results.

Adrien Thierry: Combinatorics of the Interrupted Period, pp. 17–21.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

Proposition 1 *A word w and all of its conjugates have the same number of occurrences for all of their letters, i.e. $\forall \tilde{w} \in Cl(w), \forall a \in A,\ n_a(w) = n_a(\tilde{w})$.*

Proof. Note that $\forall \tilde{w} \in Cl(w), \exists w_1, w_2$, such that $w = w_1 w_2, \tilde{w} = w_2 w_1$. Then, $\forall a \in A, n_a(w) = n_a(w_1) + n_a(w_2) = n_a(\tilde{w})$. □

The negation of Property 1 gives the following corollary:

Corollary 1. *If two words do not have the same number of occurrence for the same letter, they are not conjugates.*

Another important corollary of Property 1 is the following:

Corollary 2. *Let x be a word, $|x| \geq n+1$. If $u = x[1 \ldots n]$ and $v = x[2 \ldots n+1]$ are conjugates of each other, then $x[1] = x[n+1]$, i.e. v is a cyclic shift of u.*

Proof. Note that u and v have the factor $x[2 \ldots n]$ in common. Since u and v are conjugates, they have the same number of occurrences for all of their letters (Proposition 1). It follows that $n_{x[1]}(u) = n_{x[1]}(x[1 \ldots n]) = n_{x[1]}(x[2 \ldots n]) + 1 = n_{x[1]}(v) = n_{x[1]}(x[2 \ldots n]) + n_{x[1]}(x[n+1])$, hence $n_{x[1]}(x[n+1]) = 1$, i.e. $x[1] = x[n+1]$. □

2 Theorem

Discrete periods were described by N.J. Fine and H.S. Wilf in 1965 in the article "Uniqueness theorem for periodic functions" [1]. A corollary of that theorem, the synchronization principle, was proved by W. Smyth in [2] and L. Ilie in [3]:

Theorem 3. *If w is primitive, then, for all conjugates $\tilde{w}$ of $w, w \neq \tilde{w}$.*

Which is about the synchronization of patterns. The next theorem is about the impossible synchronization when a pattern is interrupted.

First, we need to formalize what we call an interruption of the pattern. Let x be a primitive word and x_1 be a proper prefix of x, i.e. $x_1 \neq x$. Write $x = x_1 x_2$ for some suffix x_2 of x.

Let $W = x^{e_1} x_1 x^{e_2}$ with $e_1 \geq 1, e_2 \geq 1, e_1 + e_2 \geq 3$.

We see that W has a repetition of a pattern x as a prefix: $x^{e_1} x_1$, and then the repetition is interrupted at position $|x^{e_1} x_1|$, before starting again in the suffix x^{e_2}. We need one more definition (albeit that definition is not necessary, it is presented here for better understanding) before introducing the two factors that we claim have very restricted occurrences in W.

Definition 4. *Let $\tilde{p}$ be the prefix of length $|lcp(x_1 x_2, x_2 x_1)| + 1$ of $x_1 x_2$ and $\tilde{s}$ the suffix of length $|lcs(x_1 x_2, x_2 x_1)| + 1$ of $x_2 x_1$. The factor $\tilde{s}\tilde{p}$ starting at position $|x^{e_1}| + |x_1| - |lcs(x_1 x_2, x_2 x_1)| - 1$ is the* core of the interrupt *of W.*

If W and its interrupt are clear from the context, we will just speak of the core (of the interrupt).

Example 5. Consider $x = aaabaaaaaabaaaa$ and $x_1 = aaabaaaaaabaaa$, then xx_1x^2 has $xx_1x = aaabaaaaaabaaaa\mathbf{aaabaaaaaabaaa}aaabaaaaaabaaaa$ as a prefix and $x_2 = a$. It follows that $\mathrm{lcp}(x_1x_2, x_2x_1) = aaa$, and $\tilde{p} = aaab, \mathrm{lcs}(x_1x_2, x_2x_1) = aaa$, and $\tilde{s} = baaa$. The core of the interrupt, $\tilde{s}\tilde{p}$, is the underlined in:

$$xx_1x = aaabaaaaaabaaaaaaabaaaaaa\underbrace{baaaaaab}_{\tilde{s}\tilde{p}}aaaaaabaaaa.$$

The factors that were previously known to have very restricted occurrences in W, to the best of the author's knowledge, were the inversion factors defined by A. Deza, F. Franek and A. Thierry in [4]:

Definition 6. *Let $W = x^{e_1}x_1x^{e_2}$ with $x = x_1x_2$ a primitive word and$e_1 \geq 1, e_2 \geq 1, e_1 + e_2 \geq 3$. An* inversion factor *of W is a factor that starts at position i and for which:*

- *$W[i + j] = W[i + j + |x| + |x_1|]$ for $0 \leq j < |x_1|$, and*
- *$W[i + j] = W[i + j + |x_1|]$ for $|x_1| \leq j \leq |x| + |x_1|$.*

Those inversion factors, which have the structure of $x_2x_1x_1x_2 = \tilde{x}x$, and which length are twice the length of x, were used as two notches that forces a certain synchronization of certain squares in the problem of the maximal number of squares in a word, and allowed to offer a new bound to that problem. The main anticipated application of the next result is an improvement of that bound, though the technique has already proved useful in the improvement of M. Crochemore and W. Rytter's three squares lemma, [5], by H. Bay, A. Deza and F. Franek, [6], and in the proof of the New Periodicity Lemma by H. Bay, F. Franek and W. Smyth [7].

Now, let w_1 be the factor of length $|x|$ of W that has the core of the interrupt of W as a suffix, and let w_2 be the factor of length $|x|$ that has the core of the interrupt of W as a prefix. We will show that both w_1 and w_2 have very restricted occurrences in W.

Theorem 7. *Let x be a primitive word, x_1 a proper prefix of x and $W = x^{e_1}x_1x^{e_2}$ with $e_1 \geq 1, e_2 \geq 1, e_1 + e_2 \geq 3$. Let w_1 be the factor of length $|x|$ of W ending with the core of the interrupt of W, and let w_2 be the factor of length $|x|$ starting with the core of the interrupt of W. The words w_1 and w_2 are not in the conjugacy class of x.*

Proof. Define $p = \mathrm{lcp}(x_1x_2, x_2x_1)$ and $s = \mathrm{lcs}(x_1x_2, x_2x_1)$ (note that p and s can be empty).
Deza, Franek, and Thierry showed that $|\mathrm{lcs}(x_1x_2, x_2x_1)|+|\mathrm{lcp}(x_1x_2, x_2x_1)| \leq |x_1x_2|-2$ when x_1x_2 is primitive (see [4]). Note that in the case $|\mathrm{lcs}(x_1x_2, x_2x_1)|+|\mathrm{lcp}(x_1x_2, x_2x_1)| = |x| - 2$, w_1 w_2 are the same factor.
Write $x = pr_prr_ss$ and $\tilde{x} = pr'_pr'r'_ss$ for the letters r_p, r'_p, r_s, r'_s, $r_p \neq r'_p, r_s \neq r'_s$ (by maximality of the longest common prefix and suffix) and the possibly empty and possibly homographic words r and r'.
We have, by construction, $w_1 = r'r'_sspr_p$ and $w_2 = r'_sspr_pr$.
Note that $n_{r_p}(w_1) = n_{r_p}(\tilde{x}) + 1$ and that $n_{r'_p}(\tilde{x}) = n_{r'_p}(w_1) + 1$ and, by Corollary 1,

w_1 is not a conjugate of $\tilde{x}$, nor of x. And because $|w_1| = |x|, w_1$ is neither a factor of $x^{e_1}x_1$ nor of x^{e_2}.
Similarly for w_2, $n_{r'_s}(w_2) = n_{r'_s}(x) + 1$ and $n_{r_s}(x) = n_{r_s}(w_2) + 1$ and, by corollary 1, w_2 is not a conjugate of x, and because $|w_2| = |x|, w_2$ is neither a factor of $x^{e_1}x_1$ nor of x^{e_2}. □

Example 8. Consider again $x = aaabaaaaaabaaaa$, $x_1 = aaabaaaaaabaaa$ and $x_2 = a$. We have $|x| = 15$, and:

$$xx_1x = aaabaaaaaabaaaaaaa\overbrace{baaaaaa\underbrace{\mathbf{baaaaaab}}_{w_2}aaaaaab}^{w_1}aaaa$$

The core of the interrupt is presented in bold.
The two factors w_1 and $w_2 = w_1 = baaaaaabaaaaaab$ (note that w_2 needs not be equal to w_1), starting at different positions, are not factors of x^2. Yet, the factor $aaaaaabaaaaaabaaaaaa$ of length $|x| + |\mathrm{lcs}(x, \tilde{x})| + |\mathrm{lcp}(x, \tilde{x})|$ and which contains the core of the interrupt is a factor of x^2. The same goes for the factors of length $|x|-1$ that starts and ends with the core of the interrupt, $aaaaaabaaaaaab$ and $baaaaaabaaaaaa$: they both are factors of x^2. For those reasons, the theorem can be regarded as tight

3 Conclusion

The core of the interrupt was discovered while studying double squares. An important result in the study of that problem is M. Crochemore and W. Rytter's three squares lemma, [5], of which L. Ilie offers a shorter proof in [3]. We offer here a very short proof of that result which relies on the core of the interrupt.

Lemma 9. *In a word, no more that two squares can have their last occurrence starting at the same position.*

Proof. Suppose that three squares $u_1^2, u_2^2, u_3^2, |u_1| < |u_2| < |u_3|$ start at the same position. Because u_2^2 and u_3^2 start at the same position, we can write $u_2 = x_0^{e_1}x_1$, $u_3 = x_0^{e_1}x_1x_0^{e_2}$ for $x_0 = x_1x_2$ a primitive word, x_1 a proper prefix of x_0 and $e_1 \geq e_2 \geq 1$, hence u_3 contains a core of the interrupt. Now, by synchronization principle, Theorem 3, $u_1, |u_1| < |u_2|$, cannot end in the suffix $\mathrm{lcs}(x_1x_2, x_2x_1)$ of u_2 (since u_1 has x_0 as a prefix) and ends before the core of the interrupt of u_3, but if $|u_1^2| \geq |u_3|$, the second occurrence of u_1 contains the core of the interrupt and a word of length $|x_0|$ that starts with it, while the first occurrence doesn't: which, by Theorem 7, is a contradiction.

Thanks to my supervisors Antoine Deza and Franya Franek for helpful discussions and advices and to Alice Heliou for proof reading of a preliminary version of this article.

References

1. N. J. Fine and H. S. Wilf: *Uniqueness theorems for periodic functions*, in Proceedings of the American Mathematical Society, vol. 16, no. 1, 1965, pp. 109–114.
2. B. Smyth: *Computing Patterns in Strings*. ACM Press Bks, Pearson/Addison-Wesley, 2003.

3. L. Ilie: *A simple proof that a word of length n has at most* $2n$ *distinct squares*. Journal of Combinatorial Theory, Series A, vol. 112, no. 1, 2005, pp. 163–164.
4. A. Deza, F. Franek, and A. Thierry: *How many double squares can a string contain?* Discrete Applied Mathematics, vol. 180, 2015, pp. 52–69.
5. M. Crochemore and W. Rytter: *Squares, cubes, and time-space efficient string searching*. Algorithmica, vol. 13, no. 5, 1995, pp. 405–425.
6. H. Bay, A. Deza, and F. Franek: *On a Lemma of Crochemore and Rytter*, to appear in Journal of Discrete Algorithms.
7. H. Bay, F. Franek, and W. Smyth: *The New Periodicity Lemma Revisited*, to appear in Journal of Discrete Applied Mathematics.

An Efficient Skip-Search Approach to the Order-Preserving Pattern Matching Problem*

Domenico Cantone[1], Simone Faro[1], and M. Oğuzhan Külekci[2]

[1] Università di Catania, Department of Mathematics and Computer Science, Italy
[2] ERLAB Software Co. ITU ARI Teknokent, İstanbul, Turkey
{cantone, faro}@dmi.unict.it, oguzhankulekci@gmail.com

Abstract. Given a pattern and text, both over a common ordered alphabet, the *order-preserving pattern matching* problem consists in finding all substrings of the text with the same relative order as the pattern. This problem, an approximate variant of the well-known *exact pattern matching* problem, finds applications in such fields as time series analysis (e.g., share prices on stock markets), weather data analysis, musical melody matching, etc., and has gained increasing attention in recent years. In this paper we present a new efficient approach to this problem inspired to the well-known Skip Search algorithm for the exact string matching problem. It makes use of efficient SIMD SSE instructions in order to speed up the searching phase. Experimental results show that our proposed algorithm is up to twice as faster than previous solutions.

1 Introduction

Given a pattern x of length m and a text y of length n, both over a common alphabet Σ, the *exact string matching problem* consists in finding all occurrences of the string x in y. String matching is a very important source of challenging problems in the wider domain of text processing. String matching algorithms are often basic components in practical softwares existing under most operating systems They also emphasize programming methods that serve as paradigms in other fields of computer science.

The worst-case lower bound $\mathcal{O}(n)$ for the string matching problem has been achieved for the first time by the well-known Knuth-Morris-Pratt algorithm [13] (KMP, for short). However, several string matching algorithms with a sublinear $\mathcal{O}(n \log m/m)$ performance on average have also been developed over the years. Among them, the Boyer-Moore algorithm [2] deserves a special mention, since it has been particularly successful and has inspired much work.

The *order-preserving pattern matching problem* [2,3,8,9] (OPPM, in short) is an approximate variant of the exact pattern matching problem in which the pattern x and text y are drawn from a totally ordered alphabet Σ and one is searching for all the substrings of y with the same relative order as x. For instance, when the alphabet is the set $\mathbb{N}$ of natural numbers with the standard order relation, the relative order of the sequence $x = \langle 6, 5, 8, 4, 7\rangle$ is the sequence $\langle 2, 1, 4, 0, 3\rangle$ since 6 has rank 2, 5 has rank 1, and so on. Thus x has an order-preserving occurrence in the string

$$y = \langle 8, 11, 10, 16, 15, 20, 13, 17, 14, 18, 20, 18, 25, 17, 20, 25, 26\rangle$$

at position 3, since x and the subsequence $\langle 16, 15, 20, 13, 17\rangle$ share the same relative order. Another order-preserving occurrence of x in y is at position 10 (see Fig. 1).

* This work has been supported by the Scientific & Technological Research Council Of Turkey (TUBITAK), the Department Of Science Fellowships & Grant Programs (BIDEB), 2221 Fellowship Program, and by G.N.C.S., Istituto Nazionale di Alta Matematica "Francesco Severi".

Domenico Cantone, Simone Faro, M. Oğuzhan Külekci: An Efficient Skip-Search Approach to the Order-Preserving Pattern Matching Problem, pp. 22–35.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

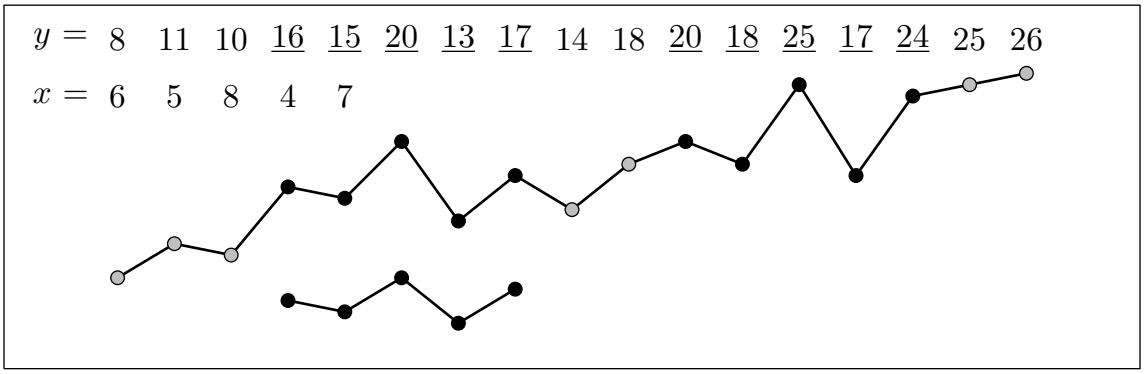


Figure 1. Example of a pattern x of length 5 over an integer alphabet with two order preserving occurrences in a text y of length 17, at positions 3 and 10.

The OPPM problem finds applications in all situations in which one is interested only in the "shape" of the pattern (intended as the relative order of its characters) rather than in the pattern itself. For instance, it can be applied successfully to time series analysis like share prices on stock markets and weather data, or to melody matching of musical scores.

In the last few years some solutions have been proposed for the OPPM problem. The first solution was presented by Kubica *et al.* [12] in 2013. They proposed a $\mathcal{O}(n+m\log m)$ solution over generic ordered alphabets based on the KMP algorithm [13] and a $\mathcal{O}(n+m)$ solution in the case of integer alphabets. A few months later, Kim *et al.* presented in [11] a similar solution running in $\mathcal{O}(n+m\log m)$ time, still based on the KMP approach. Although Kim *et al.* stressed some doubts about the applicability of the Boyer-Moore approach [2] to the OPPM problem, in 2013 Cho *et al.* [5] presented a method for deciding the order-isomorphism between two sequences showing that the Boyer-Moore approach can be applied also to the order-preserving variant of the pattern matching problem. More recently, Chhabra and Tarhio [4] presented a more practical solution based on approximate string matching techniques. Specifically, their solution consists in converting the input sequences into binary sequences and then applying any standard algorithm for exact string matching as a filtration method.

In this paper we present a new algorithm for the OPPM problem which turns out to be more effective in practice than currently available solutions. Our proposed algorithm is based on the well-known Skip Search algorithm [3] for the exact string matching problem, which consists in processing separately chunks of the text for any occurrence of the pattern. For all substrings of a given length of the pattern, a fingerprint is computed and indexed. Then, by using this information, candidate occurrences of the pattern are located in the text. We propose to use efficient SIMD SSE instructions [10] for computing the fingerprints of the pattern substrings. Experimental results show that our proposed approach leads to algorithmic variants that are up to twice as faster than previous solutions present in the literature.

The paper is organized as follows. In Section 2 we review some preliminary notions and properties relative to the OPPM problem and give an overview of the Skip Search algorithm and its searching approach. We present our proposed algorithm for the OPPM problem in Section 3 and then compare its performance with previous known algorithms in Section 4. Finally, we draw our conclusions in Section 5.

2 Preliminaries

A string x over an ordered alphabet Σ, of size σ, is defined as a sequence of elements in Σ. We shall assume that a total order relation "$\preceq$" is defined on it.

By $|x|$ we denote the length of a string x. We refer to the i-th element in x as $x[i]$ and use the notation $x[i\,..\,j]$ to denote the subsequence of x from the element at position i to the element at position j (including the extremes), where $0 \leq i \leq j < |x|$.

2.1 Order-Isomorphism and Related Properties

We say that two (nonnull) sequences x, y over Σ are order-isomorphic if the relative order of their elements is the same. More formally:

Definition 1 (Order-isomorphism). *Two nonnull sequences x, y of the same length, over a totally ordered alphabet $(\Sigma, \preceq)$, are said to be* order-isomorphic*, and we write $x \approx y$, if the following condition holds*

$$\text{for } 0 \leq i, j < |x|\,, \qquad x[i] \preceq x[j] \iff y[i] \preceq y[j].$$

The following lemma states some elementary properties of order-isomorphism which follow directly from the definition.

Lemma 2. *Let x and y be two nonnull sequences of the same length, over a totally ordered alphabet $(\Sigma, \preceq)$, such that $x \approx y$. Then*

(a) $x[j] \prec x[i]$ *iff* $y[j] \prec y[i]$, *for* $0 \leq i, j < |x|$;
(b) $x[j] = x[i]$ *iff* $y[j] = y[i]$, *for* $0 \leq i, j < |x|$. □

From a computational point of view, it is convenient to characterize the order of a sequence by means of two functions: the *rank* and the *equality* functions. These are defined below, together with some of their elementary properties.

Definition 3 (Rank function). *Let x be a nonnull sequence over a totally ordered alphabet $(\Sigma, \preceq)$. The* rank function *of x is the bijection from $\{0, 1, \ldots, |x| - 1\}$ onto itself defined, for $0 \leq i < |x|$, by*

$$rk_x(i) \;=_{Def} \left|\{k : x[k] \prec x[i] \text{ or } (x[k] = x[i] \text{ and } k < i)\}\right|.$$

The following properties are easy consequences of Definition 3.

Lemma 4. *Given a nonnull sequence x over a totally ordered alphabet $(\Sigma, \preceq)$, we have:*

(a) if $x[j] \prec x[i]$, then $rk_x(j) < rk_x(i)$, for $0 \leq i, j < |x|$;
(b) if $x[j] = x[i]$ and $0 \leq j < i < |x|$, then $rk_x(j) < rk_x(i)$. □

Corollary 5. *Let x be a nonnull sequence over a totally ordered alphabet $(\Sigma, \preceq)$. Then we have $x[rk_x^{-1}(i)] \preceq x[rk_x^{-1}(i+1)]$, for $0 \leq i < |x| - 1$.* □

For any nonnull sequence x, we shall refer to the sequence

$$\langle rk_x^{-1}(0), rk_x^{-1}(1), \ldots, rk_x^{-1}(|x| - 1)\rangle$$

as the *relative order* of x (see Example 8).

From Corollary 5, it follows that the relative order of x can be computed in time proportional to the time required to (stably) sort x.

The rank function alone allows one to characterize order-isomorphic sequences only when characters are pairwise distinct. To handle the more general case in which multiple occurrences of the same character are permitted, we also need the *equality function*.

Definition 6 (Equality function). *Let x be a sequence of length $m \geq 2$ over a totally ordered alphabet $(\Sigma, \preceq)$. The* equality function *of x is the binary map $eq_x \colon \{0, 1, \ldots, m-2\} \to \{0, 1\}$ where, for $0 \leq i \leq m-2$,*

$$eq_x(i) =_{Def} \begin{cases} 1 & \text{if } x[rk_x^{-1}(i)] = x[rk_x^{-1}(i+1)] \\ 0 & \text{otherwise.} \end{cases}$$

The rank and equality functions allow to fully characterize order-isomorphism, as stated in the following lemma, whose proof can be found in the Appendix.

Lemma 7. *For any two sequences x and y of the same length $m \geq 2$, over a totally ordered alphabet, we have*

$$x \approx y \quad \text{iff} \quad rk_x = rk_y \text{ and } eq_x = eq_y\,. \qquad \square$$

Example 8. Consider the following three sequences of length 7:

$$x = \langle 6, 3, 8, 3, 10, 7, 10\rangle, \quad y = \langle 2, 1, 4, 1, 5, 3, 5\rangle, \quad z = \langle 6, 3, 8, 4, 9, 7, 10\rangle.$$

They have the same rank function $\langle 2, 0, 4, 1, 5, 3, 6\rangle$ and, therefore, the same relative order $\langle 1, 3, 0, 5, 2, 4, 6\rangle$. However, x and y are order-isomorphic, whereas x and z (as well as y and z) are not. Notice that, in agreement with Lemma 7, we have $eq_x = eq_y = \langle 1, 0, 0, 0, 0, 1\rangle$ and $eq_z = \langle 0, 0, 0, 0, 0, 0\rangle$.

Based on the preceding lemma, in order to establish whether two given sequences of the same length m are order-isomorphic, it is enough to compute their rank and equality functions, and then compare them. The cost of such a test is dominated by the cost $\mathcal{O}(m \log m)$ of sorting the two sequences. However, if one needs to find all the sequences from a set $\mathcal{S}$ that are order-isomorphic to a fixed sequence (all the sequences having the same size m), the simple iteration of the previous test would lead to an overall complexity of $\mathcal{O}(|\mathcal{S}| \cdot m \log m)$. In this case a better approach is possible, based on the following characterization of order-isomorphism which requires the computation of the rank and equality functions of the fixed sequence only, yielding an overall complexity of $\mathcal{O}\big((|\mathcal{S}| + \log m) \cdot m\big)$.

Lemma 9. *Let x and y be two sequences of the same length $m \geq 2$, over a totally ordered alphabet. Then $x \approx y$ iff the following conditions hold:*

(i) $y[rk_x^{-1}(i)] \preceq y[rk_x^{-1}(i+1)]$, *for* $0 \leq i < m-1$
(ii) $y[rk_x^{-1}(i)] = y[rk_x^{-1}(i+1)]$ *if and only if* $eq_x(i) = 1$, *for* $0 \leq i < m-1$. $\square$

Based on Lemma 9, the procedure Order-Isomorphic in Fig. 2 correctly verifies whether a sequence y is order-isomorphic to a sequence x of the same length as y. It receives as input the functions rk_x and eq_x and the sequence y, and returns true if $x \approx y$, false otherwise. A mismatch occurs when one of the three conditions of lines 2, 3, or 4 holds. Notice that the time complexity of the procedure Order-Isomorphic is linear in the size of its input sequence y.

The OPPM problem consists in finding all the substrings of the text with the same relative order as the pattern. More precisely,

Definition 10 (Order-preserving pattern matching). *Let x and y be two sequences of length m and n, respectively, with $n > m$, both over an ordered alphabet $(\Sigma, \preceq)$. The* order-preserving pattern matching problem *consists in finding all positions i, with $0 \leq i \leq n-m$, such that $y[i\,..\,i+m-1] \approx x$.*

```
ORDER-ISOMORPHIC(inv-rk, eq, y)
1.  for i ← 0 to |y| − 2 do
2.    if (y[inv-rk(i)] ≻ y[inv-rk(i + 1)]) then return false
3.    if (y[inv-rk(i)] ≺ y[inv-rk(i + 1)] and eq(i) = 1) then return false
4.    if (y[inv-rk(i)] = y[inv-rk(i + 1)] and eq(i) = 0) then return false
5.  return true
```

Figure 2. The procedure to verify whether a sequence y is order-isomorphic to a sequence of length $|y|$, whose inverse rank and equality functions are the parameters *inv-rk* and *eq*, respectively.

If $y[i\,..\,i+m-1] \approx x$, we say that x has an *order-preserving occurrence in y at position i*.

In Section 3, we shall present an algorithm for the OPPM problem, based on the Alpha Skip Search algorithm. For convenience, we briefly review it next.

2.2 The Skip Search Algorithm and its Alpha Variant

The Skip Search algorithm is an elegant and efficient solution to the exact pattern matching problem, firstly presented in [3] and subsequently adapted to many other problems and variants of the exact pattern matching problem.

Let x and y be a pattern and a text of length m and n, respectively, over a common alphabet Σ of size σ. For each character c of the alphabet, the Skip Search algorithm collects in a bucket $B[c]$ all the positions of that character in the pattern x, so that for each $c \in \Sigma$ we have:

$$B[c] =_{\text{Def}} \{i \ : \ 0 \leq i \leq m-1 \text{ and } x[i] = c\}.$$

Plainly, the space and time complexity needed for the construction of the array B of buckets is $\mathcal{O}(m+\sigma)$. Notice that when the pattern is shorter than the alphabet size, buckets are empty.

The search phase of the Skip Search algorithm examines all the characters $y[j]$ in the text at positions $j = km-1$, for $k = 1, 2, \ldots, \lfloor n/m \rfloor$. For each such character $y[j]$, the bucket $B[y[j]]$ allows one to compute the possible positions h of the text in the neighborhood of j at which the pattern could occur.

By performing a character-by-character comparison between x and the subsequence $y[h\,..\,h+m-1]$ until either a mismatch is found, or all the characters in the pattern x have been considered, it can be tested whether x actually occurs at position h of the text.

The Skip Search algorithm has a quadratic worst-case time complexity, however, as shown in [3], the expected number of text character inspections is $\mathcal{O}(n)$.

Among the variants of the Skip Search algorithm, the most relevant one for our purposes is the Alpha Skip Search algorithm [3], which collects buckets for substrings of the pattern rather than for its single characters.

During the preprocessing phase of the Alpha Skip Search algorithm, all the factors of length $\ell = \lfloor \log_\sigma m \rfloor$ occurring in the pattern x are arranged in a trie T_x, for fast retrieval. In addition, for each leaf ν of T_x a bucket is maintained which stores the positions in x of the factor corresponding to ν. Provided that the alphabet size is

considered as a constant, the worst-case running time of the preprocessing phase is linear.

The searching phase consists in looking into the buckets of the text factors $y[j \,..\, j+\ell-1]$, for all $j = k(m-\ell+1)-1$ such that $1 \leq k \leq \lfloor (n-\ell)/m \rfloor$, and then test, as in the previous case, whether there is an occurrence of the pattern at the indicated positions of the text.

The worst-case time complexity of the searching phase is quadratic, though the expected number of text character comparisons is $\mathcal{O}(n \log_\sigma m/(m - \log_\sigma m))$.

3 A New Order-Preserving Pattern Matching Algorithm

In this section we present a new algorithm, called Order-Preserving-Skip-Search, for the Order-Preserving Pattern Matching problem. However, for brevity, in the following we shall often refer to it as SkSop algorithm.

Our algorithm combines the same approach of the Skip Search algorithm with the power of the SIMD (Single Instruction Multiple Data) instruction set, and specifically the Intel SSE (Streaming SIMD Extensions) instruction set, as discussed below.

In the last two decades a lot of effort has been spent exploiting the power of the word-RAM model of computation in order to speed-up string matching algorithms for a single pattern.

In this model, the computer operates on words of length w, so that usual arithmetic and logic operations on words all take one unit of time. Most of the solutions which exploit the word-RAM model are based on the *bit-parallelism* technique [1] or on the *packed string matching* technique [8,9]. In the packed string matching technique, multiple characters can be packed into a single word, so that the characters can be compared in bulk rather than individually.

Next we discuss our model in details.

3.1 The Model

In the design of our algorithm, we use specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology. SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data via a set of special instructions working on a limited number of special registers.

In our model of computation we assume that w is the number of bits in a word and σ is the size of the alphabet. The *packing factor* $\alpha = w/\log \sigma$ (or, rather, its floor) is the number of characters which fit in a single computer word, whereas the number of bits used to encode an alphabet character is $\gamma = \log \sigma$.

In most practical applications we have $\sigma = 256$ (ASCII code). Moreover SSE specialized instructions allow one to work on 128-bit registers, so that blocks of sixteen 8-bit characters can be read and processed in a single time unit ($\alpha = 16$). In particular, our algorithm makes use of specialized word-size packed instructions which we call wsrv (*word-size rank vector*) and wsrp (*word-size relative position*). These are reviewed next.

The instruction wsrv

The instruction $\mathsf{wsrv}(B, i)$ computes an α-bit fingerprint from a w-bit register B handled as a block of α small integers values. Assuming that $B[0 \,..\, \alpha-1]$ is a w-bit

integer parameter, wsrv(B, i) returns an α-bit value $r[0\,..\,\alpha-1]$, where $r[j] = 1$ iff $B[i] \geq B[j]$, and $r[j] = 0$ otherwise.

The wsrv(B, i) specialized instruction can be emulated in constant time by the following sequence of specialized SIMD instructions:

```
wsrv(B, i)
    D ← _mm_set1_epi8(B[i])
    C ← _mm_cmpgt_epi8(B, D)
    r ← _mm_movemask_epi8(C)
    return r
```

Specifically the _mm_set1_epi8$(B[i])$ instruction creates a w-bit register D handled as a block of α small integers values, where $D[j] = B[i]$ for $0 \leq j < \alpha$. The _mm_cmpgt_epi8(B, D) instruction compares the α integers in B and the α integers in D for "greater than". It creates a w-bit register C handled as a block of α small integers where $C[j] = 1^\gamma$ if $B[j] \geq D[j]$, and $C[j] = 0^\gamma$ otherwise, and where we remember that $\gamma = \log \sigma$ is the number of bits to encode an alphabet character. Finally, the _mm_movemask_epi8(D) instruction gets a 128 bit parameter D, handled as sixteen 8-bit integers, and creates a 16-bit mask from the most significant bits of the 16 integers in D, and zero extends the upper bits.

The instruction wsrp

The instruction wsrp(B) computes an α-bit fingerprint from a w-bit register B handled as a block of α small integers values. Assuming that $B[0\,..\,\alpha-1]$ is a w-bit integer parameter, wsrp(B) returns an α-bit value $r[0\,..\,\alpha-1]$, where $r[j] = 1$ iff $B[j] \geq B[j+1]$, and $r[j] = 0$ otherwise (we put $r[\alpha-1] = 0$).

The wsrp(B) specialized instruction can be emulated in constant time by the following sequence of specialized SIMD instructions

```
wsrp(B)
    D ← _mm_slli_si128(B, 1)
    C ← _mm_cmpgt_epi8(B, D)
    r ← _mm_movemask_epi8(C)
    return r
```

where the _mm_slli_si128$(B, 1)$ instruction shifts the w-bit register in B to the left by one position (α bits) while shifting in zeros and the _mm_cmpgt_epi8 and the _mm_movemask_epi8 instructions are as described above.

3.2 The Fingerprint Functions

The preprocessing phase of the algorithm indexes the subsequences of the pattern (of length q) in order to locate them during the searching phase. For efficiency reasons, each numeric sequence of length q is converted into a numeric value, called *fingerprint*, which is used to index the substring. A fingerprint value ranges in the interval $\{0\,..\,\tau-1\}$, for a given bound τ. The value τ is set to 2^{16}, so that a fingerprint can fit into a single 16-bit register.

The procedure FNG for computing the fingerprints is shown in Fig. 3 (on the left). Given a sequence x of length m, an index i such that $0 \leq i < m-q$, and two integers k and q such that $k \leq q \leq m$, the procedure FNG combines k different values computed

FNG(x, i, q, k)
1. $B \leftarrow 0^{\alpha-q}.x[i\,..\,i+q-1]$
2. $v \leftarrow \mathsf{wsrp}(B)$
3. for $j \leftarrow 0$ to $k-2$ do
4. $\quad v \leftarrow (v \ll 1) + \mathsf{wsrv}(B, \alpha-q+j)$
5. return v

EXAMPLE ($q = 5$, $k = 3$, and $\alpha = 8$)
$x[i\,..\,i+q-1] = \langle 3, 6, 2, 4, 7\rangle$
$B = [0, 0, 0, 3, 6, 2, 4, 7]$
$\mathsf{wsrp}(B) = [0, 0, 0, 1, 0, 1, 1, 0] = 22_{10}$
$\mathsf{wsrv}(B, 3) = [0, 0, 0, 1, 1, 0, 1, 1] = 27_{10}$
$\mathsf{wsrv}(B, 4) = [0, 0, 0, 0, 1, 0, 0, 1] = 9_{10}$
$v = 22 \times 2^2 + 27 \times 2^1 + 9 = 151_{10}$

Figure 3. On the left: the pseudo-code of the procedure FNG for the computation of the fingerprint of a substring a length q combining k distinct fingerprints. On the right: an example of a computation of a fingerprint by the procedure FNG.

on the substring $x[i\,..\,i+q-1]$ in order to compute the fingerprint v. Preliminarily, the substring $x[i\,..\,i+q-1]$ is inserted in the rightmost portion of a w-bit register B. Then the fingerprint v is computed as

$$v = \mathsf{wsrp}(B) \times 2^{k-1} + \sum_{j=0}^{k-2} \left(\mathsf{wsrv}(B, \alpha-q+j) \times 2^{k-2-j}\right) .$$

Plainly, the time complexity of the procedure FNG is $\mathcal{O}(k)$.

Fig. 3 (on the right) shows an example of how the procedure FNG works on a subsequence $x[i\,..\,i+q-1] = \langle 3, 6, 2, 4, 7\rangle$ of length $q = 5$, combining $k = 3$ different values.

3.3 The Order-Preserving Skip Search algorithm

We are now ready to briefly describe our algorithm for the OPPM problem, based on the Alpha variant of the Skip Search algorithm. We distinguish in it a preprocessing and a searching phase.

The preprocessing phase of the SKSOP algorithm, which is reported in Fig. 4 (on the left), consists in compiling the fingerprints of all possible substrings of length q contained in the pattern x. Thus a fingerprint value v, with $0 \leq v < 2^\alpha$, is computed for each subsequence $x[i\,..\,i+q-1]$, for $0 \leq i < m-q$.

To this purpose a table F of size 2^α is maintained for storing, for any possible fingerprint value v, the set of positions i such that FNG$(x, i, q, k) = v$. More precisely, for $0 \leq v < 2^\alpha$, we have

$$F[v] = \Big\{ i \mid 0 \leq i < m-q \text{ and } \text{FNG}(x, i, q, k) = v \Big\}.$$

The preprocessing phase of the SKSOP algorithm requires some additional space to store the $(m-q)$ possible alignments in the 2^α locations of the table F. Thus, the space requirement of the algorithm is $\mathcal{O}(m-q+2^\alpha)$ that approximates to $\mathcal{O}(m)$, since α is constant. The first loop of the preprocessing phase just initializes the table F, while the second loop is run $(m-q)$ times, which makes the overall time complexity of the preprocessing phase $\mathcal{O}(m+2^\alpha)$ that, again, approximates to $\mathcal{O}(m)$.

The basic idea of the searching phase is to compute a fingerprint value every $(m-q)$ positions of the text y and to check whether the pattern appears in y, involving the

```
PREPROCESSING(x, q, m, k)
1. for v ← 0 to 2^α − 1 do
2.     F[v] ← ∅
3. for i ← 0 to m − q do
4.     v ← FNG(x, i, q, k)
5.     F[v] ← F[v] ∪ {(i + q − 1)}
6. return F
```

```
SKSOP(x, r, y, n, q, k)
1. F ←Preprocessing(x, q, m, k)
2. for j ← m − 1 to n step m − q + 1 do
3.     v ← FNG(y, j, q, k)
4.     for each i ∈ F[v] do
5.         z ← y[j − i .. j − i + m − 1]
6.         if ORDER-ISOMORPHIC(rk_x^{-1}, eq_x, z)
7.             then output (j)
```

Figure 4. The pseudo-code of the SkSop algorithm for the OPPM problem.

block $y[j \,..\, j+q-1]$. If the fingerprint value indicates that some of the alignments are possible, then the candidate positions are checked naively for matching.

The pseudo-code provided in Fig. 4 (on the right) reports the skeleton of the SkSop algorithm. The main loop investigates the blocks of the text y in steps of $(m-q+1)$ blocks. If the fingerprint v computed on $y[j \,..\, j+q-1]$ points to a nonempty bucket of the table F, then the positions listed in $F[v]$ are verified accordingly.

In particular $F[v]$ contains a linked list of the values i marking the pattern x and the beginning position of the pattern in the text. While looking for occurrences on $y[j \,..\, j+q-1]$, if $F[v]$ contains the value i, this indicates the pattern x may potentially begin at position $(j-i)$ of the text. In that case, a matching test is to be performed between x and $y[j-i \,..\, j-i+m-1]$ via a character-by-character inspection.

The total number of filtering operations is exactly $n/(m-q)$. At each attempt, the maximum number of verification requests is $(m-q)$, since the filter provides information about that number of appropriate alignments of the patterns. On the other hand, if the computed fingerprint points to an empty location in F, then there is obviously no need for verification. The verification cost for a pattern x of length m is assumed to be $\mathcal{O}(m)$, with the brute-force checking approach. Hence, in the worst case the time complexity of the verification is $\mathcal{O}(m(m-q))$, which happens when all alignments in x must be verified at any possible beginning position. Hence, the best case complexity is $\mathcal{O}(n/(m-q))$, while the worst case complexity is $\mathcal{O}(nm)$.

4 Experimental Evaluations

In this section we present experimental results in order to evaluate the performance of our Skip-Search based algorithm SkSop. In particular we tested our algorithm against the filter approach of Chhabra and Tarhio [4], which is, to the best of our knowledge, the most effective solution to the OPPM problem in practical cases. In the experimental evaluations reported in [4], the SBNDM2 and SBNDM4 algorithms [7] turned out to be the most effective exact string matching algorithms which can be used in combination with the filter technique. In our experimental evaluations, we used the SBNDM2 algorithm to test the filter approach by Chhabra and Tarhio. In our dataset we use the following short names to identify the algorithms that we have tested:

– Fct: the SBNDM2 algorithm based on the filter approach by Chhabra and Tarhio presented in [4];

– SkSop(k, q): our Skip Search-based algorithm presented in Section 3, which combines k different fingerprint values on subsequences of length q.

More specifically, in our tests, we considered the SkSop(k, q) algorithm, for $k \in \{1, 2, 3, 4, 5\}$ and $q \in \{3, 4, 5, 6, 7, 8\}$.

The Boyer-Moore approach by Cho *et al.* [5] has not been included in our evaluations, as it was shown to be less efficient than the algorithm by Chhabra and Tarhio in all cases.

All algorithms have been evaluated in terms of efficiency, i.e. running times, and accuracy, i.e., number of verifications performed during the searching phase. In particular, they have been tested on sequences of small integer values (i.e., in the range $[0\,..\,256]$), big integer values (i.e., in the range $[0\,..\,10.000]$) and real numbers (i.e., in the range $[0, 0\,..\,10.000, 99]$). However, we did not observe any significant difference in the results; thus, for brevity, in the following table we report only the results relative to small integer sequences. Each text consists in a sequence of 1 million elements. In particular we tested our algorithm on the following set of small integer sequences:

– Rand-δ: a sequence of random integer values ranging around a fixed mean μ with a variability of δ and a uniform distribution, i.e. each value is uniformly distributed in the range $\{\mu - \delta\,..\,\mu + \delta\}$;
– Periodic-ρ: a sequence of random integer values uniformly ranging around a cyclic function with a period of ρ elements.

For each text in the set, we randomly selected 100 patterns extracted from the text and computed the average running time over 100 runs. We also computed the average number of false positives detected by the algorithms during the search. Algorithms have been implemented using the `C` programming language and have been compiled using the `gcc` compiler Apple LLVM version 5.1 (based on LLVM 3.4svn) with 8Gb Ram. Compilation has been performed with the `-O3` optimization option.

For each value of k, we have also reported in round parentheses the value of q which led to the best performance.

Average Number of Verifications

We evaluated the accuracy of our solutions in terms of the number of verifications performed during the search. Specifically, we counted the average number of verifications that each algorithm performs every 2^{10} text characters and computed the mean of such value over 100 runs. Table 1 and Table 2 show, respectively, the results obtained on Rand-δ sequences, with $\delta = 5, 20, 40$, and on Periodic-ρ sequences, with $\rho = 8, 16, 32$. Best results have been underlined.

Results on Rand-δ sequences (Table 1) show that the difference in the number of verifications performed during the search is sensible when different fingerprints are used. Using a single fingerprint (algorithm SkSop$(1, q)$) leads to a quite high number of verifications, up to 50 every 2^{20} characters. The best results are obtained by combining 4 different fingerprint values (algorithm SkSop$(4, q)$), in which case the number of verifications decreases to 0.25 every 2^{20} characters.

When we combine more than 4 fingerprint values (algorithm SkSop$(5, q)$), the number of verifications sensibly increases to 0.5. This behavior is due to the combination process which uses a final hash value of 16 bits and which causes loss of information.

δ	m	SkSop$(1,q)$	SkSop$(2,q)$	SkSop$(3,q)$	SkSop$(4,q)$	SkSop$(5,q)$
5	8	52.64 (8)	3.87 (8)	1.20 (8)	0.25 (8)	0.43 (8)
	12	46.22 (8)	4.27 (8)	1.02 (8)	0.25 (8)	0.41 (8)
	16	45.65 (8)	4.01 (8)	1.06 (8)	0.24 (8)	0.41 (8)
	20	48.13 (8)	4.18 (8)	1.09 (8)	0.24 (8)	0.42 (8)
	24	45.02 (8)	4.13 (8)	1.05 (8)	0.24 (8)	0.42 (8)
	28	44.92 (8)	4.05 (8)	1.03 (8)	0.24 (8)	0.41 (8)
	32	46.99 (8)	4.23 (8)	1.04 (8)	0.23 (8)	0.44 (8)
20	8	37.78 (8)	3.96 (8)	1.02 (8)	0.23 (8)	0.51 (8)
	12	40.65 (8)	4.17 (8)	1.04 (8)	0.25 (8)	0.52 (8)
	16	39.78 (8)	4.65 (8)	1.00 (8)	0.25 (8)	0.52 (8)
	20	39.05 (8)	4.12 (8)	1.02 (8)	0.24 (8)	0.49 (8)
	24	39.24 (8)	4.35 (8)	1.02 (8)	0.25 (8)	0.50 (8)
	28	40.15 (8)	4.34 (8)	1.00 (8)	0.24 (8)	0.49 (8)
	32	40.00 (8)	4.39 (8)	1.01 (8)	0.25 (8)	0.51 (8)
40	8	42.34 (8)	4.37 (8)	1.03 (8)	0.27 (8)	0.54 (8)
	12	35.64 (8)	4.50 (8)	0.99 (8)	0.25 (8)	0.49 (8)
	16	41.08 (8)	4.40 (8)	1.01 (8)	0.26 (8)	0.54 (8)
	20	40.71 (8)	4.29 (8)	1.05 (8)	0.26 (8)	0.54 (8)
	24	37.77 (8)	4.33 (8)	0.96 (8)	0.25 (8)	0.52 (8)
	28	39.98 (8)	4.51 (8)	1.02 (8)	0.25 (8)	0.53 (8)
	32	38.26 (8)	4.46 (8)	0.99 (8)	0.26 (8)	0.54 (8)

Table 1. Average number of verifications performed every 2^{10} characters, computed on a Rand-δ small integer sequence, with $\delta = 5, 20$, and 40.

ρ	m	SkSop$(1,q)$	SkSop$(2,q)$	SkSop$(3,q)$	SkSop$(4,q)$	SkSop$(5,q)$
8	8	92.22 (8)	37.40 (8)	14.22 (8)	8.01 (8)	8.83 (8)
	12	98.48 (8)	35.46 (8)	14.72 (8)	8.27 (8)	10.38 (8)
	16	98.27 (8)	36.46 (8)	15.71 (8)	8.77 (8)	10.46 (8)
	20	96.95 (8)	35.91 (8)	15.14 (8)	8.47 (8)	10.12 (8)
	24	96.88 (8)	36.06 (8)	14.87 (8)	8.34 (8)	10.18 (8)
	28	97.63 (8)	35.79 (8)	14.60 (8)	7.94 (8)	9.67 (8)
	32	97.65 (8)	35.93 (8)	15.09 (8)	8.31 (8)	10.23 (8)
16	8	173.85 (8)	40.23 (8)	5.19 (8)	3.74 (8)	7.03 (8)
	12	179.84 (8)	46.64 (8)	5.35 (8)	4.25 (8)	7.62 (8)
	16	179.20 (8)	46.94 (8)	5.59 (8)	4.35 (8)	7.57 (8)
	20	176.24 (8)	45.61 (8)	5.40 (8)	4.20 (8)	7.07 (8)
	24	181.67 (8)	46.50 (8)	5.53 (8)	4.24 (8)	7.31 (8)
	28	176.67 (8)	46.27 (8)	5.47 (8)	4.18 (8)	7.09 (8)
	32	179.96 (8)	46.12 (8)	5.55 (8)	4.34 (8)	7.52 (8)
32	8	125.55 (8)	35.52 (8)	3.23 (8)	2.26 (8)	3.96 (8)
	12	134.48 (8)	35.41 (8)	3.19 (8)	2.13 (8)	3.84 (8)
	16	136.69 (8)	39.27 (8)	3.34 (8)	2.31 (8)	4.07 (8)
	20	140.14 (8)	40.58 (8)	3.51 (8)	2.33 (8)	3.99 (8)
	24	138.36 (8)	39.70 (8)	3.52 (8)	2.39 (8)	4.15 (8)
	28	139.04 (8)	37.90 (8)	3.44 (8)	2.35 (8)	4.05 (8)
	32	136.39 (8)	39.09 (8)	3.44 (8)	2.33 (8)	4.06 (8)

Table 2. Average number of verifications performed every 2^{10} characters, computed on a Periodic-ρ small integer sequence, with $\rho = 8, 16$, and 32.

Results on Periodic-ρ sequences show how the number of verifications performed by the algorithms is affected by the value of ρ. Specifically the number of verifications

δ	m	FCT	SKSOP$(1,q)$	SKSOP$(2,q)$	SKSOP$(3,q)$	SKSOP$(4,q)$	SKSOP$(5,q)$
5	8	42.32	0.81 (5)	1.14 (4)	1.22 (4)	1.27 (4)	1.27 (4)
	12	27.09	0.80 (7)	1.21 (5)	1.35 (5)	1.37 (5)	1.34 (5)
	16	20.38	0.83 (8)	1.33 (6)	1.44 (5)	1.52 (5)	1.50 (5)
	20	16.59	0.88 (8)	1.39 (7)	1.54 (6)	1.58 (5)	1.56 (6)
	24	13.56	0.89 (8)	1.44 (7)	1.60 (6)	1.61 (6)	1.63 (6)
	28	11.50	0.85 (8)	1.47 (7)	1.56 (7)	1.59 (5)	1.62 (6)
	32	9.97	0.81 (8)	1.47 (7)	1.57 (7)	1.59 (6)	1.60 (6)
20	8	42.13	0.81 (4)	1.12 (4)	1.22 (4)	1.19 (4)	1.14 (4)
	12	27.41	0.84 (6)	1.24 (5)	1.40 (5)	1.40 (5)	1.35 (5)
	16	19.78	0.85 (7)	1.28 (6)	1.43 (6)	1.46 (5)	1.40 (5)
	20	15.73	0.90 (8)	1.33 (7)	1.49 (6)	1.51 (5)	1.50 (6)
	24	13.24	0.89 (8)	1.40 (7)	1.51 (6)	1.55 (6)	1.55 (6)
	28	11.37	0.86 (8)	1.45 (7)	1.57 (6)	1.57 (6)	1.58 (6)
	32	9.89	0.85 (8)	1.42 (7)	1.58 (7)	1.56 (6)	1.54 (7)
40	8	41.32	0.81 (4)	1.11 (4)	1.19 (4)	1.16 (4)	1.11 (4)
	12	27.36	0.83 (6)	1.22 (5)	1.38 (5)	1.39 (5)	1.34 (5)
	16	19.78	0.84 (7)	1.27 (6)	1.42 (6)	1.43 (5)	1.40 (6)
	20	16.21	0.90 (8)	1.34 (7)	1.51 (6)	1.52 (6)	1.52 (6)
	24	13.26	0.90 (8)	1.40 (7)	1.51 (7)	1.54 (6)	1.57 (6)
	28	11.38	0.86 (8)	1.43 (7)	1.56 (7)	1.56 (6)	1.57 (6)
	32	9.93	0.84 (8)	1.43 (8)	1.56 (7)	1.58 (6)	1.58 (7)

Table 3. Running times on a RAND-δ small integer sequence, with $\delta = 5, 20$ and 40. Running times (in milliseconds) are reported for the FCT algorithm, while speed-up values are reported for the SKSOP(k,q) algorithms.

increases when the period of the function decreases. This is due to the high presence of similar patterns in the text. In addition, in this case, the best results are obtained by the SKSOP(k,q) algorithm. Good results are obtained also by combining 3 or 5 fingerprint values.

We observe also that in all cases the number of verifications for each text character is less than 0.1 and is not affected by the length of the pattern. Thus we can observe that the SKSOP(k,q) algorithm has a linear behavior on average, as will become apparent in the following evaluation of the running times.

Running Times

The performance of the algorithms presented has been evaluated in terms of their running times. We compared the time required by each algorithm while searching the text for the set of 100 patterns. Table 3 and Table 4 show, respectively, the experimental results obtained on RAND-δ sequences, with $\delta = 5, 20, 40$, and on PERIODIC-ρ sequences, with $\rho = 8, 16, 32$. Running times are expressed in milliseconds. Best results have been underlined.

In particular, for the FCT algorithm we reported the average running times obtained as the mean of 100 runs. Instead, in the case of the execution of SKSOP(k,q) algorithms, we reported the speed up of the running times obtained when compared with the time taken by the FCT algorithm. Specifically, if *time*(FCT) is the running time of the FCT algorithm and t is the running time of our algorithm, then the speed up is computed as *time*(FCT)$/t$.

Experimental results on RAND-δ sequences (Table 3) show that the performances of all algorithms are not affected by the value of δ. The FCT algorithm is dominated

ρ	m	Fct	SkSop$(1,q)$	SkSop$(2,q)$	SkSop$(3,q)$	SkSop$(4,q)$	SkSop$(5,q)$
8	8	39.90	0.79 (3)	0.87 (4)	0.89 (4)	0.87 (4)	0.87 (4)
	12	32.94	0.87 (5)	1.09 (6)	1.19 (6)	1.24 (6)	1.17 (6)
	16	27.21	0.90 (7)	1.24 (7)	1.44 (7)	1.55 (7)	1.46 (7)
	20	21.47	0.90 (8)	1.21 (7)	1.44 (7)	1.60 (7)	1.50 (7)
	24	19.29	0.94 (8)	1.27 (7)	1.59 (8)	1.77 (7)	1.68 (8)
	28	16.90	0.91 (8)	1.28 (7)	1.65 (8)	1.81 (7)	1.77 (8)
	32	15.58	0.89 (8)	1.28 (7)	1.68 (8)	1.87 (8)	1.83 (8)
16	8	37.40	0.68 (4)	0.83 (4)	0.93 (4)	0.93 (4)	0.90 (4)
	12	25.03	0.60 (5)	0.79 (4)	1.03 (5)	1.04 (5)	0.99 (5)
	16	18.63	0.60 (6)	0.80 (7)	1.15 (6)	1.15 (6)	1.10 (6)
	20	15.22	0.53 (8)	0.81 (8)	1.22 (7)	1.19 (7)	1.15 (7)
	24	12.75	0.50 (7)	0.78 (8)	1.26 (7)	1.24 (7)	1.19 (7)
	28	10.52	0.45 (7)	0.73 (8)	1.21 (7)	1.20 (7)	1.14 (7)
	32	10.01	0.43 (8)	0.78 (8)	1.31 (8)	1.29 (7)	1.23 (8)
32	8	38.82	0.76 (4)	1.00 (4)	1.11 (4)	1.09 (4)	1.05 (4)
	12	24.86	0.65 (6)	0.91 (4)	1.16 (5)	1.18 (5)	1.15 (5)
	16	18.85	0.61 (6)	0.89 (5)	1.24 (6)	1.27 (5)	1.24 (6)
	20	15.02	0.58 (6)	0.86 (6)	1.31 (6)	1.34 (6)	1.30 (6)
	24	12.32	0.52 (7)	0.83 (7)	1.31 (7)	1.32 (6)	1.28 (6)
	28	10.89	0.50 (8)	0.85 (8)	1.38 (7)	1.38 (6)	1.34 (6)
	32	9.50	0.48 (8)	0.81 (8)	1.37 (7)	1.38 (6)	1.34 (7)

Table 4. Running times on a Periodic-δ integer sequence, with $\delta = 5, 20$, and 40. Running times (in milliseconds) are reported for the Fct algorithm, while speed-up values are reported for the SkSop(k,q) algorithms.

by our Skip Search-based algorithms in all cases, especially for long patterns. When the length of the pattern is between 8 and 20, the SkSop$(4,q)$ algorithm obtains the best results, whereas the SkSop$(5,q)$ algorithm is the best solution when the length of the pattern is greater or equal to 20. In this latter case, the SkSop$(5,q)$ algorithm obtains a significant speed up of 1.60 if compared with the Fct algorithm.

Experimental results on Periodic-ρ sequences (Table 4) show that the algorithm SkSop$(4,q)$ obtains the best results in most of the cases, with the best speed up (up to 1.9) most of the times, especially for long patterns.

The Fct algorithm is still the fastest solution only when the pattern is very short ($m = 8$), while the SkSop$(3,q)$ algorithm obtains very good results when the length of the pattern is between 8 and 20.

5 Conclusions

In this paper we discussed the Order-Preserving Pattern Matching Problem and presented a new algorithm to solve such problem, based on the well-known Skip Search approach. It turns out that our solution is much more effective in practice than existing algorithms. Our algorithm uses SIMD SSE instructions to speed up the searching process. Experimental results show that our solution is up to twice as faster than previous solutions, while exhibiting a linear behavior on average.

References

1. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Comm. of the ACM, 35(10), 1992, pp. 74–82.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM 20(10), 1977, pp. 762–772.
3. C. Charras, T. Lecroq, and J. D. Pehoushek: *A very fast string matching algorithm for small alphabets and long patterns*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., Piscataway, New Jersey, vol. 1448 of Lecture Notes in Computer Science, Springer-Verlag, Berlin 1998, pp. 55–64.
4. T. Chhabra and J. Tarhio: *Order-preserving matching with filtration*, in Proc. SEA '14, 13th International Symposium on Experimental Algorithms, vol. 8504 of Lecture Notes in Computer Science, Springer 2014, pp. 307–314.
5. S. Cho, J. C. Na, K. Park, and J. S. Sim: *Fast order-preserving pattern matching*, in Widmayer, P., Xu, Y., Zhu, B., eds., COCOA 2013, vol. 8287 of Lecture Notes in Computer Science, Springer, Chengdu 2013, pp. 295–305.
6. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen: *Order-preserving incomplete suffix trees and order-preserving indexes*, in Proc. SPIRE 2013, 20th International Symposium, vol. 8214 of Lecture Notes in Computer Science, Springer, Jerusalem 2013, pp. 84–95.
7. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Information Processing Letters 110(4), 2010, pp. 148–152.
8. S. Faro and M. O. Külekci: *Fast Packed String Matching for Short Patterns*, in Proc. of the 15th Meeting on Algorithm Engineering and Experiments, 2013, pp. 113–121.
9. S. Faro and M. O. Külekci: *Fast and flexible packed string matching.* J. Discrete Algorithms vol. 28, 2014, pp. 61–72.
10. Intel Corporation: *Intel (R) 64 and IA-32 Architectures Optimization Reference Manual*, 2011.
11. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order preserving matching.* Theoretical Computer Science 525, 2014, pp. 68–79.
12. M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen: *A linear time algorithm for consecutive permutation pattern matching.* Information Processing Letters 113(12), 2013, pp. 430–433.
13. D. E. Knuth, J. M. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing 6(2), 1977, pp. 323–350.
14. G. Navarro and M. Raffinot: *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, New York, NY 2002.

Alternative Algorithms for Order-Preserving Matching

Tamanna Chhabra[1], M. Oğuzhan Külekci[2], and Jorma Tarhio[1]

[1] Department of Computer Science, Aalto University
P.O. Box 15400, FI-00076 Aalto, Finland
firstname.lastname@aalto.fi

[2] ERLAB Software Co.
ITU Ari2 Teknokent, Istanbul, Turkey
okulekci@medipol.edu.tr

Abstract. The problem of order-preserving matching is to find all substrings in the text which have the same relative order and length as the pattern. Several online and one offline solution were earlier proposed for the problem. In this paper, we introduce three new solutions based on filtration. The two online solutions rest on the SIMD (Single Instruction Multiple Data) architecture and the offline solution is based on the FM-index scheme. The online solutions are implemented using two different SIMD instruction sets, SSE (streaming SIMD extensions) and AVX (Advanced Vector Extensions). Our main emphasis is on the practical efficiency of algorithms. Therefore, we show with practical experiments that our new solutions are faster than the previous solutions.

Keywords: order-preserving matching, string searching, FM-index, SIMD, SSE, AVX

1 Introduction

The string matching problem [26] of finding all occurrences of a pattern string P of length m in a text string T of length n is one of the classical problems in computer science. Over the last few decades, there has been active development in the field of string matching. One string problem is to locate all the substrings in the text T which have the same relative order and length as the pattern P. This problem is known as *order-preserving matching* [1,4,7,8,21,23]. It has applications in time series studies [20] such as the analysis of development of share prices in a stock market.

In classical string matching, the text T and the pattern P are strings of characters. In order-preserving matching, T and P are strings of numbers. The term relative order means the numerical order of the numbers in the string. In $P = (35, 42, 29, 24, 32, 40)$, number 24 is the smallest number of the pattern, 29 is the second smallest and so on. Therefore, the relative order of P is $4, 6, 2, 1, 3, 5$. The aim of order-preserving matching is to find all the substrings in T which have the same length and relative order as P. It can be observed that the relative order of the substring at location 4 of text $T = (10, 18, 22, 30, 39, 15, 12, 20, 35, 24, 32)$, matches that of the pattern P as shown in Fig. 1, where indexing of T starts from zero.

Several online [1,4,8,21,23] and one offline solution [6] have been proposed for order-preserving matching. Kubica et al. [23] proposed the first online solution based on the Knuth–Morris–Pratt algorithm (KMP) [22]. The second solution was put forward by Kim et al. [21] which also rested on the KMP algorithm. Both the solutions were linear. Later, Cho et al. [4] introduced a solution based on the Boyer–Moore–Horspool (BMH) algorithm [17] and it was the first practical sublinear average-case

Tamanna Chhabra, M. Oğuzhan Külekci, Jorma Tarhio: Alternative Algorithms for Order-Preserving Matching, pp. 36–46.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

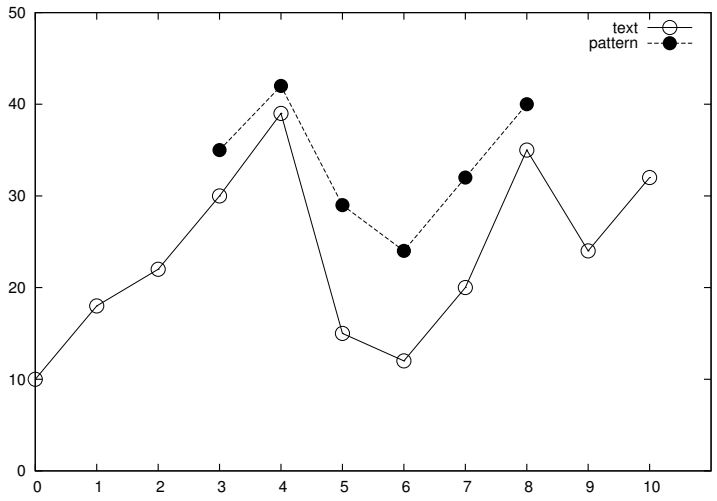


Figure 1. Example of order preserving matching.

solution of the problem. At the same time, Belazzougui et al. [1] derived an optimal algorithm which is sublinear on average. Independently, Chhabra and Tarhio [8] presented another sublinear average-case solution based on filtration and it was shown to be faster in practice than the previous solutions. In the upcoming sections of the paper, we will refer to this solution as OPMF, short for order-preserving matching with filtration. In addition, Crochemore et al. [6] proposed an offline solution based on indexing.

In this paper, we will introduce two new online solutions utilizing the *SIMD (single instruction, multiple data)* architecture [19] and one offline solution based on the FM-index [14]. The online solutions use specialized packed string instructions with a low latency and throughput and turned out to be clearly faster than the previous online solutions. The OPMF algorithm is based on computing a transformed pattern and text by creating their respective bitmaps where a 1 bit means the successive element is greater than the current one and a 0 bit means the opposite. In the online solutions, we aim to perform this transformation quickly with SSE4.2 (streaming SIMD extensions) and AVX (Advanced Vector Extensions) instructions. In the offline solution, the computed bitmap of the text is stored in the compressed form via the FM-index scheme. The transformed pattern is then searched in the FM-index to get potential matches which are then verified. Our main emphasis is on the practical efficiency of algorithms. Therefore, we compared our new solutions with OPMF which was proven to be the fastest practical solution so far. Our experiments show that at least one of our new online solutions is in most cases faster than the original OPMF. And the indexing solution was the most efficient as one may expect.

The paper is organized as follows. Section 2 presents the background, Section 3 describes the problem definition, Section 4 outlines the previous solutions for the order-preserving matching, Section 5 introduces our solution based on filtration, Section 6 interprets the analysis of the solution, Section 7 presents the results of practical experiments, and Section 8 concludes the article.

2 Background

Our new online solutions apply filtration and the SIMD instruction set architecture [19]. These instructions were originally developed for multimedia but are recently employed for pattern matching. The general trend in the last decades for speeding up string matching algorithms has been based on the word-RAM model, where in practice several operations on items occupying a single *word* are assumed to be achieved in constant time. In that context, the advance of the SIMD technology gave rise to packed string matching [2], where one can assume several consecutive symbols of the underlying text are packed into a single register, and there exist special instructions on those special registers to operate on those items individually. The SIMD instructions were used to create a filter while searching for single long patterns in [24]. The filtration code was listed among the best performing 11 pattern matching algorithms in a recent survey [13]. The same idea was deployed for multiple string matching [11], and then extended to also cover short patterns [12,10]. Ladra et al. [16] investigated the benefits of using SIMD instructions on compressed data structures, mainly on rank/select operations, and analyzed the BMH algorithm [17] as a case study. Our results in this paper show that SIMD instructions can also be very efficient in order-preserving pattern matching as well.

The SIMD architecture [19] allows the execution of multiple data on single instruction. SSE (streaming SIMD extensions) [19,29] is a family of SIMD instruction sets supported by modern processors. Intel added sixteen new 128-bit registers known as XMM0 through XMM15. However, the registers XMM7–XMM15 are only accessible in the 64-bit operating mode. As the registers are 128 bits long, four floating point numbers could be handled at the same time (a single precision floating point number is considered 32 bits long), thereby providing important speedups in algorithms. Furthermore, the functionality provided by SSE instructions was extended by Intel AVX (Advanced Vector Extensions) [29]. It provides support for 256-bit registers known as YMM0 through YMM15. As with SSE, in AVX also the registers YMM7–YMM15 are only accessible in the 64-bit operating mode. As the registers have been extended from 128 bits to 256 bits, hence eight floating point numbers could be managed simultaneously, thereby rendering substantial performance gain. Dedicated data types are utilized in SIMD programming. In SSE4.2, we have the following data types:

- _m128: four 32-bit floating point values
- _m128d: two 64-bit floating point values
- _m128i: 16/8/4/2 integer values, depending on the size of the integers

The similar data types are available in AVX but the length of registers is 256. These vector data types are defined in separate header files depending on the type of instruction set architecture. To perform a task, we have different intrinsic functions. The name of the function starts with _mm. After that follows the name describing the operation. The next character specifies whether the operation is on a packed vector or on a scalar value: p stands for packed and s for scalar operation. The last character relates to single precision or double precision floating point values. For example, _mm_cmpgt_ps is a function for comparing two values.

The offline solution rests on the FM-index scheme [14]. Ferragina and Manzini [14] proposed that if the Burrows-Wheeler transform [3] is coupled with another data structure, namely Suffix Arrays (SA) [25], we get a space efficient index which is a sort of compressed suffix array called the FM-index. It can be used to count efficiently

the occurrences of a pattern in the compressed text and to determine the locations of each pattern in the text.

3 Problem definition

Problem definition. Two strings u and v of the same length over Σ are called *order-isomorphic* [23], written $u \approx v$, if

$$u_i \leq u_j \Leftrightarrow v_i \leq v_j \text{ for } 1 \leq i, j \leq |u|.$$

In the *order-preserving pattern matching problem*, we want to locate all the substrings in the text T which are order-isomorphic with the pattern P.

In the example of Section 1, the substring $w = (30, 39, 15, 12, 20, 35)$ of T is order-isomorphic with pattern $P = (35, 42, 29, 24, 32, 40)$ as can also be seen in Fig. 1.

4 Previous solutions

This section describes the previous solutions formulated for the order-preserving matching problem. First of all we explain the online solutions given so far. The first solution was presented by Kubica et al. [23] based on Knuth–Morris–Pratt algorithm (KMP) [22]. In this approach the fail function in the KMP algorithm was modified to compute the order-borders table. With this table one can find out in linear time if the text contains substring with the same relative order as that of the pattern.

The second solution to the problem was given by Kim et al. [21] and was grounded on the prefix representation. It is based on finding the rank of each number in the prefix. They used the dynamic order statistic tree as the data structure and the text is searched using the KMP-order-matcher function. The total time complexity is $O(n \log m)$. This approach was then enhanced using the nearest neighbor representation. Thereafter, the total time complexity is $O(n + m \log m)$.

However, Cho et al. [4] provided a solution based on the BMH approach [17]. They applied the variant of BMH algorithm built on q-gram. This was the first practical sublinear solution of the problem. The time complexity in the worst case is $O(mn)$. Later on, they also developed a version which is linear in the worst case [5], but that is in practice a bit slower than the original one.

The algorithm by Belazzougui et al. [1] is optimal sublinear. They viewed the problem in a slightly different way: T is a permutation of $1, \ldots, n$ and P consists of m distinct integers of $[1, n]$. They constructed a forward search automaton working in $O(m^2 \log\log m + n)$ time which is too large for long patterns. With a Morris-Pratt representation of the forward automaton, they accieved $O(m \log\log m + n)$ search time. Furthermore, the automaton was extended to accept a set of patterns. Besides these linear solutions, they presented a sublinear average case algorithm. Firstly, a tree is constructed of all isomorphic order factors of P by inserting factors one at a time. Thereafter search is performed along the text through a window of size m. The construction time of the tree is $O(\frac{m \log m}{\log\log m})$ and average-case time complexity is $O(\frac{n \log m}{m \log\log m})$. However, there exists no implementation of this algorithm so far.

Another sublinear average-case solution is OPMF [8]. The solution consisted of two phases: filtration and verification. In filtration, the pattern P and the text T are transformed to P' and T' by creating their respective bitmaps such that a 1 bit

means the successive element is greater than the current one, and a 0 bit means otherwise. The text is transformed incrementally online in order to be able to skip characters. Any (sublinear) exact string matching is then applied to filter out the text. As a result, we get match candidates, which are then verified using a checking function. In addition to exact order-preserving matching, the same filtration method can also be applied to approximate order-preserving matching [7] and to multiple order-preserving matching [28].

Lastly, let us consider the offline solution by Crochemore et al. [6]. This approach is grounded on the construction of an index that handles the queries in linear time with respect to the length of the pattern. The index is based on the incomplete suffix tree and its construction takes $O(n \log \log n)$ time. They extended their work to complete order-preserving suffix trees and showed how these can be constructed in $O(n \log n / \log \log n)$ time. There exists no practical implementation of this algorithm.

5 Our solutions

We propose two online and one offline solutions for order preserving matching. The online solutions utilize the SIMD architecture [19]. The first online solution employs the SSE4.2 instruction set architecture and the second solution utilizes the AVX instruction set architecture.

Online solution using SSE4.2

In the OPMF algorithm, the pattern P is transformed into P' and the text T is transformed incrementally to T'. We aim to perform the online transformation faster than in OPMF. This solution for order-preserving matching consists of two parts: filtration and verification. First the text is filtered and then the match candidates are verified using a checking routine.

Filtration using SSE4.2. Assume that we have 32 bits long floating point numbers and the processor has SSE4.2 support. The preprocessing of the pattern consists of two parts. First a bit mask, which is the reverse of P', is formed and after that a shift table is constructed based on the mask. For the bit mask, the consecutive numbers in the pattern $P = p_0 p_1 \cdots p_{m-1}$ are compared pairwise, $(p_0 > p_1)(p_1 > p_2)(p_2 > p_3) \cdots (p_{m-2} > p_{m-1})$. This can be achieved by creating $\_mm128$ type pointers $ptr1$ and $ptr2$ pointing to p_0 and p_1 respectively. Furthermore, we use the PCMPGT instruction (_mm_cmpgt_ps()) to compare $ptr1$ with $ptr2$ to compute $(p_0 > p_1)(p_1 > p_2)(p_2 > p_3)(p_3 > p_4)$ in parallel. It compares the packed single-precision floating-point values in the source operand (the second operand) and the destination operand (the first operand) and returns the results of the comparison to the destination operand. The result of this instruction is 128 bits long. Additionally, we use the MOVMSK instruction (_mm128_movemask_ps()) which extracts the most significant bits from the packed single-precision floating-point value in the source operand. The reverse of the result is stored in the four low-order bits of the destination operand. The upper bits of the destination operand are filled with zeros. The result will be the bit mask *mask*. Alg. 1 shows how the transformation of the pattern P into *mask* can be carried out rapidly.

Since SSE4.2 allows four numbers to be compared in parallel, we apply binary 4-grams and set the size of the shift table *delta* to 16 ($=2^4$). The construction algorithm

```
PREPROCESSING(mask)
for (i = 0; i < 16; i++)
   delta[i]=m-1
   k = (mask<<3) & 0xf;
for (i = 0; i < 8 ; i++)
   delta[k+i]=m-2
   k = (mask<<2) & 0xf;
for (i = 0; i < 4 ; i++)
   delta[k+i]=m-3
   k = (mask<<1) & 0xf;
for (i = 0; i < 2 ; i++)
   delta[k+i]=m-4
for (i = 0; i < m-4 ; i++)
   delta[(mask>>i) & 0xf] = m-i-5
```

```
SEARCH(Text, delta)
i=m-5;
while i<n do
  k=1
  while k>0 do
    k = delta[simd-comp(t_i,t_{i+1},4)]
    i = i+k
  for (j=i-m+5; j<i; j+=4)
    z = simd-comp(t_j,t_{j+1},4)
    if (z != ((mask>>(j-i+m-5)) & 0xf))
     then goto out
  verify occurrence
  out: i = i+1
```

Figure 2. The PREPROCESSING and SEARCH phases.

for *delta* is shown in the left-hand side of Fig. 2. The computation of the parameter *mask* is explained above. The entry $delta[x]$ is zero if x is the reverse of the last 4-gram of P'. The entries of the table are initialized to $m-1$. Thereafter, the entries are updated according to the preprocessing algorithm of Fig. 2. Fig. 3 shows how the shift table is formed for the pattern P of Fig. 1. At the end, entry 12 is zero. This means that $12 = 1100$ is the reverse of the last 4-gram of P'.

Algorithm 1 (Transformation of pattern into bitmap)

```
mask = 0
for (i = 0; i < (m-1); i=i+4)
   x_ptr = _mm_loadu_ps(pattern+i+1)
   y_ptr = _mm_loadu_ps(pattern+i)
   mask = mask | _mm_movemask_ps(_mm_cmpgt_ps(x_ptr, y_ptr)) << i
```

The search algorithm shown in the right-hand side of Fig. 2 is a variation of the BMH algorithm [17,27] utilizing 4-grams. Inside the main loop there are two loops. The first loop searches for occurrences of the last 4-gram of P' by using the shift table *delta*. The tested 4-gram is formed online with SIMD instructions in the same way as used for the pattern. The numbers are compared in parallel using PCMPGT instruction explained above (simd-comp in Fig. 2). The second loop checks whether a complete occurrence of P' is found. If an occurrence of P' is found, the corresponding part of T is verified. The search algorithm uses a copy of the pattern as a sentinel (not shown in Fig. 2) to be able to recognize the end of input.

As each occurrence of P' in T' is only a match candidate, it should be verified. In simple words, for instance, if $P = (15, 18, 20, 16)$ and $T = (2, 4, 6, 1, 5, 3)$ then the transformed pattern P' and T' are 110 and 110101 respectively where 1 indicates increase and 0 indicates the opposite. The match candidate of P' at location 0 of T' needs to be verified because though P' appears in T', the relative order of the numbers is 0,2,3,1 in the pattern and 1,2,3,0 in the text. Therefore P' is only a match candidate.

Verification. The verification process is the same as in OPMF. In the preprocessing phase, the numbers of the pattern $P = p_0 p_1 \cdots p_{m-1}$ are sorted. As a result, we obtain an auxiliary table r: $p_{r[i]} \leq p_{r[j]}$ holds for each pair $i < j$ and $p_{r[0]}$ is the smallest number in P. The potential candidates obtained from the filtration phase are traversed in accordance with the table r. If the candidate starts from t_j in T, the first comparison is done between $t_{j+r[0]}$ and $t_{j+r[1]}$.

$delta[0000]$	$\leftarrow 5$				
$delta[0001]$	$\leftarrow 5$				
$delta[0010]$	$\leftarrow 5$			$\leftarrow 2$	
$delta[0011]$	$\leftarrow 5$			$\leftarrow 2$	
$delta[0100]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[0101]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[0110]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[0111]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[1000]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1001]$	$\leftarrow 5$	$\leftarrow 4$			$\leftarrow 1$
$delta[1010]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1011]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1100]$	$\leftarrow 5$	$\leftarrow 4$			$\leftarrow 0$
$delta[1101]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1110]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1111]$	$\leftarrow 5$	$\leftarrow 4$			

Figure 3. Computation of the shift table for $mask = 11001$ for $P' = 10011$.

Online solution using AVX

This is similar to the above solution with a few exceptions. The difference is in the comparison of numbers and in computation of the shift function. Instead of four numbers, eight floating point numbers are compared at a stretch. The comparison instruction is _mm256_cmp_ps(), which requires three operands. The predicate operand (the third operand) specifies the type of comparison to be performed on each of the pairs of packed values.

Offline Solution

This solution also enumerates the bitmaps but they are stored in the compressed form via the FM-index. In this case, when a pattern is queried, we just extract the possible candidate positions from the index, and then apply naive check. It also consists of two parts: filtration and verification.

Filtration. In the preprocessing phase, the consecutive numbers in the pattern $P = p_0p_1 \cdots p_{m-1}$ are compared pairwise and the pattern P is transformed into a bitmap P' in the same way as in OPMF. The text is also encoded and an FM-index is created of the encoded text. Alg. 2 below shows how the encoded text is stored in the form of FM-index. Thereafter, the occurrences of transformed pattern P' are found within the compressed text. As an occurrence of P' is only a potential match candidate, it should be verified with a checking routine.

Note. It was thought that there might be an inefficiency in the FM-index for a bit string. It is because the FM-index uses a wavelet tree, and it would be useless in the case of a binary text. So a modified FM-index without a wavelet tree might be more efficient. Therefore we implemented another FM-index without a wavelet tree. To keep the FM-index compressed, the Burrows-Wheeler transform of the bit-string was computed and was compressed via rank and select dictionaries, and then the backward search on the compressed bit string was implemented via rank/select queries. But we observed that this approach was slower than the standard one.

Verification. The verification process is the same as in the online solution because once we get the potential matches they are verified using the same checking function.

Algorithm 2 (FM-index)

```
std::string str((char *) & text[0], n);
construct_im(fm_index, str.c_str(), 1);
matches=count(fm_index, (const char*)P');
auto locations=locate(fm_index, (const char*)P');
```

6 Analysis

Let us assume that the numbers in $P = p_0 p_1 \cdots p_{m-1}$ and $T = t_0 t_1 \cdots t_{n-1}$ are integers and they are statistically independent of each other and the distribution of numbers is discrete uniform. Let P' and T' be the corresponding bitmaps for filtration. In case of online solutions using SIMD, the analysis is similar to the analysis of the original OPMF algorithm. It is obvious that our SIMD search algorithms are sublinear on average, because the search algorithm based on BMH is sublinear on average. The verification time approaches zero when m grows, and the filtration time dominates. When filtering the text T, it is encoded incrementally online in order to skip characters, and the solution as a whole becomes sublinear on average. In the worst case, the total algorithm requires $O(nm)$ time, if for example P' is 1^{m-1} and T' is 1^{n-1}. The preprocessing phase requires $O(m \log m)$ due to sorting of the pattern positions. See the analysis of OPMF [8] for more details.

In the case of the offline solution using the FM-index, the verification time also approaches zero when m grows and the filtration time dominates. During the preprocessing phase, the text T' is compressed and stored via the FM-index. The operation *count* takes a pattern P' and returns the number of occurrences of that pattern in the text T'. It can count all matching positions in $O(m)$ time. The operation *locate* finds the locations of all the occurrences (occ) of the pattern P' in T' in time $O(m + occ \log^{\epsilon} n)$. However, in the worst case, this solution also requires $O(nm)$ time because checking a match candidate takes $O(m)$ time.

7 Experiments

The tests were run on Intel 2.70 GHz i7 processor with 16 GB of memory. All the algorithms were implemented in C and run in the testing framework of Hume and Sunday [18]. In case of offline solution, the FM-index was implemented using the sdsl library [15].

We tested our algorithms on ten different data sets. Of all the results, we present the results on three texts: a random text and two real texts. The random data [20] is 320 MB long. The real data comprised of time series of the Dow Jones index and feature data [20]. The Dow Jones data consisted of 15,128 integers and feature data is 198 MB long. The patterns were randomly picked from the text. We had eight sets of 300 patterns in case of random and feature data set with lengths $5, 9, 10, 12, 15, 18, 20$ and 25 and five sets of 300 patterns with lengths $5, 10, 15, 20$ and 25 in case of Dow Jones data. Each test was repeated nine times.

We compared our new solutions with our earlier OPMF solutions [9] based on the SBNDM2 and SBNDM4 algorithms. Because the latter solutions were faster than the other old solutions in the tests of [9], we do not present results for other methods. Tables 1 and 2 show the average execution times of the algorithms for a set of 300 patterns for random and feature data in seconds, respectively, whereas table 3 depicts the average execution times of the algorithms for a set of 300 patterns for Dow Jones

data in 10 of milliseconds. In addition, graphs on times for random data are shown in Fig. 4 respectively. In the tables given below, SBNDM2 represents the OPM algorithm based on SBNDM2 filtration, SBNDM4 represents the OPM algorithm based on SBNDM4 filtration, SSE represents the online solution based on SSE4.2 instruction set, AVX represents the online solution based on AVX instruction set and FM-INDEX represents the offline solution based on the FM index.

m	SBNDM2	SBNDM4	SSE	AVX	FM-INDEX
5	18.44	22.56	12.26	—	405.53
9	15.96	13.34	8.71	9.06	31.53
10	13.99	12.21	7.98	7.37	14.01
12	11.56	10.66	7.07	5.69	3.14
15	9.07	8.80	6.35	4.59	0.39
18	7.52	7.34	5.92	4.10	0.05
20	6.85	6.69	5.82	3.87	0.01
25	5.59	5.40	5.44	3.59	0.00

Table 1. Execution times of algorithms in seconds for random data

m	SBNDM2	SBNDM4	SSE	AVX	FM-INDEX
5	16.92	20.28	12.33	—	306.26
9	9.41	8.02	5.37	5.63	21.29
10	8.26	7.32	4.86	4.58	8.52
12	6.93	6.53	4.29	3.48	3.16
15	5.42	5.27	3.80	2.81	0.32
18	4.52	4.43	3.62	2.52	0.11
20	4.06	4.30	3.45	2.36	0.04
25	3.28	3.29	3.29	2.17	0.02

Table 2. Execution time of algorithms in seconds for feature data

m	SBNDM2	SBNDM4	SSE	AVX	FM-INDEX
5	1.14	1.49	0.83	—	14.17
10	0.49	0.31	0.36	0.45	0.41
15	0.29	0.16	0.34	0.22	0.04
20	0.18	0.14	0.28	0.21	0.03
25	0.12	0.12	0.23	0.10	0.04

Table 3. Execution times of algorithms in 10 of milliseconds for Dow Jones data

From Tables 1, 2, and 3, it can be clearly seen that our solutions based on the FM-index, SSE4.2 and AVX are the fastest depending on the value of m. Irrespective of the data, the solution based on SSE4.2 is the fastest for $m = 5$. In case of random and feature data, as the value of m reaches 10, the AVX solution becomes the fastest. However, when m is greater than or equal to 12, the FM-index based solution is the fastest. And as the value of m reaches 25, the execution time of FM-index based solution approaches zero. However, in the case of Dow Jones data, the FM-index based solution is the fastest as the value of m reaches 10.

The construction times of the FM-index for our Dow Jones and random texts were 0.07 and 3.2 seconds, respectively.

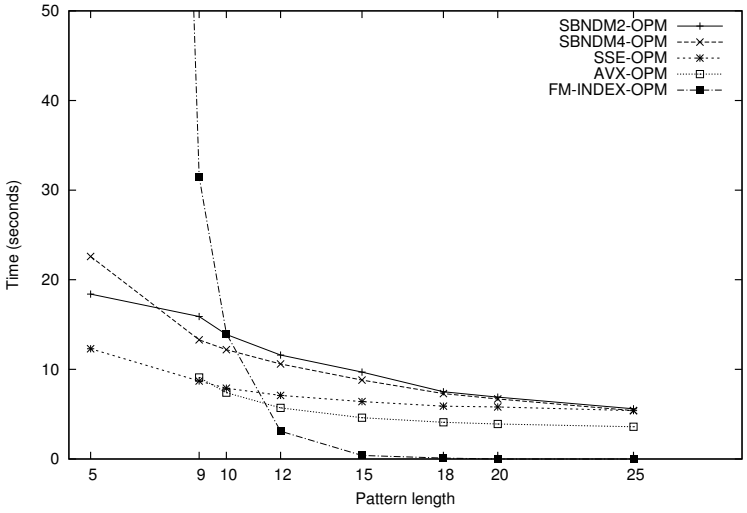


Figure 4. Execution times of algorithms for random data

8 Concluding remarks

We pioneered two online and one offline solution for the order-preserving matching problem. The online solutions are based on SIMD which improves the execution time substantially in most cases. The SIMD architecture requires careful redesigning of an algorithm, and the outcome is not necessarily efficient for an arbitrary string matching problem. The offline solution, which is based on the FM-index, is superior for long patterns. However, the search algorithm of the offline solution was slower than we expected for short patterns. We have proved with practical experiments that our solutions are competitive with the previous solutions.

References

1. D. Belazzougui, A. Pierrot, M. Raffinot, and S. Vialette: *Single and multiple consecutive permutation motif search*, in Proceedings of 24th International Symposium on Algorithms and Computation, ISAAC 2013, Hong Kong, China, December 16–18, 2013, pp. 66–77.
2. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann: *Optimal packed string matching*, in Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS vol. 13, 2011, pp. 423–432.
3. M. Burrows and D.J. Wheeler: *A block sorting lossless data compression algorithm*. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
4. S. Cho, J. C. Na, K. Park, and J. S. Sim: *Fast order-preserving pattern matching*, in Widmayer, P., Xu, Y., Zhu, B., eds., 7th International Conference on Combinatorial Optimization and Applications 2013, vol. 8287 of Lecture Notes in Computer Science, 2013, pp. 295–305.
5. S. Cho, J. C. Na, K. Park, and J. S. Sim: *A fast algorithm for order-preserving pattern matching*. Inf. Process. Lett., 115(2) 2015, pp. 397–402.
6. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen: *Order-preserving incomplete suffix trees and order-preserving indexes*, in Kurland, O., Lewenstein, M., Porat, E., eds., 20th String Processing and Information Retrieval Symposium 2013, vol. 8214 of Lecture Notes in Computer Science, 2013, pp. 84–95.
7. T. Chhabra, E. Giaquinta, and J. Tarhio: *Filtration algorithms for approximate order-preserving matching*. Submitted, 2015.

8. T. Chhabra and J.Tarhio: *Order-preserving matching with filtration*, in Gudmundsso, J., Katajainen, J., eds., 13th International Symposium on Experimental Algorithms 2014, vol. 8504 of Lecture Notes in Computer Science, 2014, pp. 307–314.
9. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
10. S. Faro and M. O. Külekci: *Fast and flexible packed string matching.* Journal of Discrete Algorithms, 28 (2014), pp. 61–72.
11. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in L. Calderón-Benavides et al., eds., 19th International Symposium on String Processing and Information Retrieval 2012, vol. 7608 of Lecture Notes in Computer Science, pp. 217–228.
12. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments 2013, pp. 113–121.
13. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Computing Surveys (CSUR), 45(2) 2013, p. 13.
14. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in 41st Annual Symposium on Foundations of Computer Science, IEEE 2000, pp. 390–398.
15. S. Gog: *Succinct Data Structure Library 2.0*, `https://github.com/simongog/sdsl-lite`.
16. S. Ladra, O. Pedreira, J. Duato, and N.R. Brisaboa: *Exploiting SIMD instructions in current processors to improve classical string algorithms*, in 16th East European Conference on Advances in Databases and Information Systems 2012, T. Morzy, T. Haerder, and R. Wrembel, eds., vol. 7503 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg 2012, pp. 254–267.
17. R. N. Horspool: *Practical fast searching in strings.* Software–Practice and Experience, 10(6) 1980, pp. 501–506.
18. A. Hume and D. Sunday: *Fast string searching.* Software–Practice and Experience, 21(11) 1991, pp. 1221–1248.
19. H. Jeong, S. Kim, W. Lee, and S.-H. Myung: *Performance of SSE and AVX instruction sets.* CoRR abs/1211.0820, 2012.
20. E. Keogh, Q. Zhu, B. Hu, Y. Hao., X. Xi, L. Wei, and C. A. Ratanamahatana: *The UCR Time Series Classification/Clustering Homepage*, `http://www.cs.ucr.edu/~eamonn/UCRsuite.html`
21. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C.S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order preserving matching.* Theor. Comp. Sci., 525 (2014), pp. 68–79.
22. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
23. M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen: *A linear time algorithm for consecutive permutation pattern matching.* Inf. Process. Lett., 113(12) 2013, pp. 430–433.
24. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in J. Holub and J. Žďárek, eds., Prague Stringology Conference 2009, pp. 118–128.
25. U. Manber and G. Myers: *Suffix arrays. A new method for on-line string searches*, in 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM 1990, pp. 319–327.
26. G. Navarro and M. Raffinot: *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, New York, NY, 2002.
27. J. Tarhio and H. Peltola: *String matching in the DNA alphabet.* Software–Practice and Experience, 27(7) 1997, pp. 851–861.
28. B. Watson: *Personal Communication*, 2015.
29. Intel Corporation: *Intel Architecture Instruction Set Extensions Programming Reference*, `https://software.intel.com/sites/default/files/m/9/2/3/41604`.

Efficient Algorithm for δ - Approximate Jumbled Pattern Matching

Iván Castellanos and Yoan Pinzón

Faculty of Engineering,
National University of Colombia,
Bogotá, Colombia
{iycastellanosm, ypinzon}@unal.edu.co

Abstract. The Jumbled Pattern Matching problem consists on finding substrings which can be permuted to be equal to a given pattern. Similarly the δ - Approximate Jumbled Pattern Matching problem asks for substrings equivalent to a permutation of the given pattern, but allowing a vector of possible errors δ. Here we provide a new efficient solution for the δ - Approximate Jumbled Pattern Matching problem using indexing tables and bit vectors which, according to the experimental results, gives a speed up about $1.5 - 3.5$ times faster than the solution based on Wavelet trees. This speed up depends mainly of the size of the alphabet. Further there are presented some solutions to another problems related to δ - Approximate Jumbled Pattern Matching, as the All Matching problem, where it is necessary to calculate all the occurrences of a given pattern allowing an error in the text, or the Min-Error problem, where the objective is to find the occurrences which are closer to the pattern.

Keywords: Parikh vectors, jumbled pattern matching, bit vectors, bit parallelism

1 Introduction

Stringology or String Matching is one of the most widely studied problems in computer science, due to the extensive many applications where this problem is used. The main idea of this problem is to find patterns in a text. However, there are many different versions of the original String Matching Problem. In this paper we study one of these versions.

The problem of Jumbled Pattern Matching, also known as Parikh [11], Abelian [8] or Permutation Matching [13], was firstly introduced at [14]. This is a variant of the Pattern Matching problem, where instead of looking in a text at a substring identical to the given pattern, the main interest is to find a substring which has a permutation identical to the given pattern [7].

The Jumbled Pattern Matching can be used to solve different problems in bioinformatics, such as alignment [2], interpretation of mass spectometry data [3], SPN discovery [4], gene clusters, repeated pattern discovery, scrabble and table arragement problems [5], to name some. This variant of Pattern Matching has also been used as a filtering step in the approximate Pattern Matching algorithms.

The Jumbled Pattern Matching problem has been already studied through different versions, which can be applied to the same applications of the original Jumbled Pattern Matching. In [6,12] an approximate version of the problem is considered, here what it is important is to calculate the maximal occurrences in a text between two bound queries, each of them in the form of a vector known as Parikh.

More formally, the Parikh vector of a string s with characters from a finite ordered alphabet Σ, denoted $p(s) = (p_1, \ldots, p_\sigma)$, is defined as the vector of frequencies from

Iván Castellanos, Yoan Pinzón: Efficient Algorithm for δ - Approximate Jumbled Pattern Matching, pp. 47–56.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

each character of Σ in s. For the problem of Exact Jumbled Pattern Matching the target is to find either every substring s' of s such that $p(s')$ is equal to a given Parikh vector q (occurrence problem) or if there is at least one substring s' of s such that $p(s')$ is equal to a given Parikh vector q (decision problem).

For the approximate version of the Jumbled String Matching we need in addition to the Parikh vector q a vector of possible errors δ. The occurrence problem from the δ - Approximate Jumbled Pattern Matching consists in finding all the matchings of a pattern q in the text s such that the absolute difference between the occurrence and the pattern is not bigger than the error δ, i.e. (i, j) is a match if for the substring $s' = s_i \cdots s_j$, $|p(s') - q| \leq \delta$. Similarly the decision version of this problem consists in deciding whether q occurs in s allowing the error δ.

Example 1. Consider the alphabet $\Sigma = \{a, b, c\}$, the string $s = ccabbcbaaccbbaaccbabab$ and the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$. It has 5 maximal occurrences, namely $(5, 11)$, $(6, 12)$, $(8, 17)$, $(13, 19)$, and $(14, 21)$ with errors 2, 2, 3, 2 and 3 respectively. in Figure 1 it is showed this example, and the substrings whose corresponding Parikh vector is a maximal match.

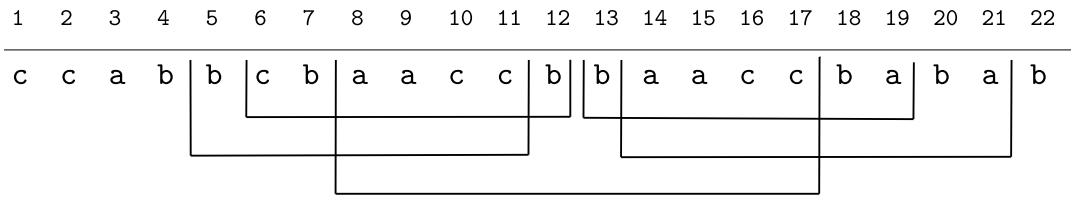


Figure 1. Maximal ocurrences of the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ in the text *ccabbcbaaccbbaaccbabab*.

Our main contribution is to provide an efficient solution for the δ - Approximate Jumbled Pattern Matching problem. We use different data structures and a new approach based on bit vectors, producing a speed up on previous solutions. Also there are presented some adaptions of the algorithm for another problems that fit in the category of approximate jumbled pattern matching and that to the best of our knowledge, they were not defined before.

The rest of the paper is organized as follows, in section 2 we give some basic definitions and an overview of the used algorithm. In section 3 we present a new implementation of this algorithm, follow by some experimental results in section 4. Finally, the paper is closed with some concluding marks and future work in section 5.

2 Preliminaries

Given a finite ordered alphabet Σ with σ elements, i.e. $\Sigma = \{a_1, \ldots, a_\sigma\}$, $a_1 < \cdots < a_\sigma$ and a string $s \in \Sigma^*$ of length $|s| = n$, i.e., $s = s_1 \cdots s_n$. The Parikh vector of s denoted $p(s) = (p_1, \ldots, p_\sigma)$ counts the multiplicity from each character of Σ in s, i.e. $p_i = |\{j \mid s_j = a_i\}|$ for $i = 1, \ldots, \sigma$, additionally, by $pr(s, i)$ we denote the prefix of s until position i inclusive, i.e. $pr(s, i) = s_1, \ldots, s_i$. If s is clear, it can be used just $pr(i)$, also we represent $p(pr(s, i))$ or $p(pr(i))$ by $prv(s, i)$ or $prv(i)$ respectively. By $s[i, j] = s_i \cdots s_j$ we denote the substring of s from i to j inclusive for $1 \leq i \leq j \leq n$, note that $p(s[i, j]) = prv(j) - prv(i - 1)$.

For a Parikh vector $q \in \mathbb{N}^\sigma$, where $\mathbb{N}$ denotes the set of positive integers including zero, and let $|q| := \sum_{i=1,\ldots,\sigma} q_i$ denote the length of q. It is said that (i, j) is an occurrence of q in s if and only if $p(s[i, j]) = q$. By convention, it is said that the empty string ε occurs in each string once. The problem of deciding whether q occurs in s, known as decision problem, or finding all the occurrences of q in s, known as occurrence problem, is called Jumbled Pattern Matching (JPM) [9].

For two Parikh vectors $p, q \in \mathbb{N}^\sigma$, the binary operations $p \leq q$ and $p + q$ are defined component-wise, i.e. $p \leq q$ if and only if $p_i \leq q_i$ for all $i = 1, \ldots, \sigma$, and $p + q = u$ where $u_i = p_i + q_i$ for $i = 1, \ldots, \sigma$ respectively. Similarly, if $p \geq q$, we set $q - p = v$ where $v_i = q_i - p_i$ for $i = 1, \ldots, \sigma$. Note that for two Parikh vectors p and q it is possible that neither $p \leq q$ nor $q \leq p$. Finally, if the Parikh vector p is greater or equal than the Parikh vector q, it is said that p is a super-Parikh vector of q or also that q is sub-Parikh vector of p.

Let $s \in \Sigma^*$ a text and $u, v \in \mathbb{N}^\sigma$ a pair of given Parikh vectors with $|s| = n$ and $u \leq v$. u, v are called the query bounds. The problem of finding all maximal occurrences in s of some Parikh vector q such that $u \leq q \leq v$ is one version of what is refered to Approximmate Jumbled Pattern Matching (AJPM). An occurrence (i, j) of q is maximal (w.r.t u and v) if neither $(i - 1, j)$ nor $(i, j + 1)$ is an occurrence of some Parikh vector q' such that $u \leq q' \leq v$. The decision version of the problem is where we only want to know whether some q occurs in s satisfying the bounds $u \leq q \leq v$. In the rest of this paper the lower bound u will be denoted as $q - \delta$ and the upper bound v as $q + \delta$.

In both, the exact and the approximate problem, it is possible to determine if there are occurrences of one query in $O(n)$ time using a window approach. However, usually it is of more importance to find occurrences in a text for many queries. Because of that it is necessary to make a preprocessing of the text, in the following sections we assume that K many queries arrive over time. Obviously, all sub-Parikh vectors of s can be precomputed, then stored them (sorted, e.g, lexicograpically) and when a query arrives, binary search can be done to find the occurrences in the text. In this case, preprocessing time is $\Theta(n^2 \log n)$, because the number of Parikh vectors of s is at most $\binom{n}{2} = O(n^2)$ and there are nontrivial strings with quadratic number of Parikh vectors over arbitrary alphabets. With this preprocessing the time for each query is $O(\log n)$ for the decision problem and $O(\log n + M)$ for the occurrence problem where M is the number of occurrences of the query. On the other hand, the storage space is $\Theta(n^2)$ and this is inacceptable in many applications.

2.1 ESR Algorithm

In [6] were introduced the concepts of Expansion, Shrinkage and Refining to move two pointers L and R, our solution is based on this approach. The pointers L and R represent a window pointing the potential positions $i - 1$ and j where it can be found an occurence, clearly these pointers can be moved linearly for each position in the text, however, this is not optimal. The algorithm instead, updates these pointers in jumps, alternating between updates of R and L, in a manner such that many positions are skipped. In addition, because of the way we update the pointers, we can ensure that, every time we have a maximal occurrence (i, j) the pointers will have the values $i - 1$ and j.

For the Expansion phase R is moved, extending the window to the right until its corresponding Parikh vector is a super-Parikh vector of $q - \delta$. At the end of this

phase, it can be possible that the Parikh vector of the substring contained in the window does not satisfy the bounds because is not a sub-Parikh vector of the upper bound $q + \delta$. Consequently, we switch to the Shrinkage phase. For this phase L is moved, shrinking the window from the left until its corresponding Parikh vector is a sub-Parikh vector of $q + \delta$.

After this it might be that the Parikh vector of the string contained in the window is not a super-Parikh vector of $q - \delta$ anymore. In this case, we need to start a new cycle with an Expansion phase.

On the other hand, if after the Shrinkage phase the Parikh vector corresponding to the window is still a super-Parikh vector of $q - \delta$, then the window with positions (L, R) satisfies the condition $|p(s[L + 1, R]) - q| \leq \delta$, i.e., it is a match, although it is not necessarily maximal. In order to make this occurrence a maximal one, it is necessary to enter the Refining phase moving again R, extending the window to the right as long as the Parikh vector of the window is a sub-Parikh vector of $q - \delta$. After this, a match of the query is kept and it is maximal, because if we extend the window to the left or to the right it is not a match anymore. Finally, after reporting the found occurrence, the process is restarted by expanding the window to the right by one character and entering a new cycle starting with a Shrinkage phase.

More formally, we can express these functions as

$$Expand(k, v) := \min\{j \mid prv(j) \geq prv(k) + v\} \tag{1}$$

$$Shrink(k, v) := \min\{j \mid prv(k) - prv(j) \leq v\} \tag{2}$$

$$Refine(k, v) := \max\{j \mid prv(j) - prv(k) \leq v\} \tag{3}$$

In Figure 2 is showed an example of the ESR Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Expand(0,{2,0,2})	c	c	a	b	b	c	b	a	a	c	c	b	b	a
Shrink(8,{4,2,4})	c	c	a	b	b	c	b	a	a	c	c	b	b	a
Expand(5,{2,0,2})	c	c	a	b	b	c	b	a	a	c	c	b	b	a
Refine(5,{4,2,4})	c	c	a	b	b	c	b	a	a	c	c	b	b	a

Figure 2. Results of the first 4 operations using the ESR Algorithm for query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ in the text *ccabbcbaaccbba*.

During the algorithm are used $Expand(L, q-\delta)$, $Shrink(R, q+\delta)$ and $Refine(L, q+\delta)$. In [6] the ESR algorithm is defined and it is mentioned that the complexity of this algorithm depends on how the functions *prv*, *Expand*, *Shrink*, *Refine* and some comparations are implemented and on the number of times these functions are executed or the number of times we update the pointers R and L.

3 Our Work

In the following section, we show a new implementation of the ESR algorithm, using different data structures to those used previously in the literature [6] to solve the

problem of δ - Approximate Jumbled Pattern Matching, namely Wavelet trees. Firstly, $prv(s,\, i)$ is stored in vectors, making this takes linear space on memory and also can be calculated linearly in time, initially $prv(0) = (0_1, \ldots, 0_\sigma)$ and then for $i = 1, \ldots, n$ and $j = 1, \ldots, \sigma$ we have:

$$prv_j(i) = \begin{cases} prv_j(i-1) & s[i] \neq a_j \\ prv_j(i-1) + 1 & s[i] = a_j \end{cases} \tag{4}$$

Clearly, having this table $prv(i)$ can be calculated during the algorithm in constant time.

Theorem 1. *Given a Parikh vector v and a position k we have that for the text s:*

$$Expand(k,\, v) := \max\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k))'th \, occurrence \, of \, j\} \tag{5}$$

$$Shrink(k,\, v) := \max\{i_j \,|\, i_j \, pos \, of \, the \, (prv_j(k) - v_j)'th \, occurrence \, of \, j\} \tag{6}$$

$$Refine(k,\, v) := \min\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k) + 1)'th \, occurrence \, of \, j\} - 1 \tag{7}$$

Proof. To show the equivalence between (1) and (5), first set $R' = \max\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k))'th \, occurrence \, of \, j\}$, then $prv(R') \geq prv(k) + v$, because for every $j = 1, \ldots, \sigma$ we have that $prv_j(R') \leq v_j + prv_j(k)$ due to the definition of R', now without loss of generality assume that $R' = i_1$, then taking any $R'' < R'$ we have that $prv(R'') \not\leq v + prv(k)$ because $prv_1(R'') \not\leq v_1 + prv_1(k)$. So it can be concluded that $R' = \min\{j \,|\, prv(j) \geq prv(k) + v\}$. Analogously the equivalence between (2) and (6) can be proved.

To show the last equivalence, set $R' = \min\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k) + 1)'th \, occurrence \, of \, j\}$, without loss of generality taking $R' = i_1$ then $prv_1(R') - prv_1(k) = v_1 + 1$ and $prv_j(R') - prv_j(k) < v_j + 1$ for $j = 2, \ldots, \sigma$, because of the definition of R'. Then it can be seen that $prv(R') - prv(k) \not\leq v$. However, taking $R' - 1$ we have that $prv_1(R' - 1) - prv_1(k) < v_1 + 1$, so $prv(R' - 1) - prv(k) \leq v$. It proves that $R' - 1 = \max\{j \,|\, prv(j) - prv(k) \leq v\}$.

As a consequence of this theorem, calculating the functions of *Expand*, *Shrink* and *Refine* can be done in $O(\sigma)$ using an inverted index table, Basically, the position of each character in the string is stored, then it can be known which is the position in the text of the $n'th$ occurrence of the character $a \in \Sigma$ in constant time. Note that using this approach with all the elements of the alphabet it can be calculated each position in the text where a Parikh vector can fit, if the $n'th$ occurrence of character $a \in \Sigma$ does not exist, then it means that in the text there are less than $n \, a's$.

For the functions *Expand* and *Shrink* we calculate first the Parikh vectors corresponding to $prv(L) + q - \delta$ and $prv(R) - q + \delta$ then we use the inverted index table to find the minimum position in which every character of these Parikh vectors can fit in the text, finally we calculte the maximum of this positions, because it is necessary that all the characters of the Parikh vectors fit in the text. For the function *Refine* we calculate the Parikh vector $prv(L) + q + \delta$ and then we use the inverted index table to find the positions where the occurrences + 1 are, then we take the minimum and we make -1 to get the last position where the Parikh vector can fit completely.

Obviously, implementing these functions as described above has a time complexity of $\Theta(\sigma)$, which makes the complete algorithm to have a worst case time complexity of $\Theta(\sigma n)$ for a query, however, this happens in a few singular cases. in [6] it is proved that the average time complexity is sublinear when implementing this functions in $\Theta(\sigma)$.

3.1 Parikh vectors as bit vectors

Previously, all the implementations of Parikh vectors had been made using integer vectors. Nevertheless, due to the intrinsic paralellism of bit words, we represent here the Parikh vectors as bit vectors, storing many integers in the same bit word to make faster some operations of the Parikh vectors. For each element in the Parikh vector we use the number of bits that the integer represent and one more to use as a carry. Since it is necessary to know each of the elements of the Parikh vector when using the inverted index table we just use shiftings in order to get this values in constant time.

We use the bit vectors mainly to make a speed up in three operations used on the ESR algorithm: addition, substraction and comparation. These operations were defined component-wise, then what it is done at the end is adding, substracting and comparing all the elements of the Parikh vectors, but in parallel, using the properties of operations on bit words. The addition and the substraction are still being equal to the operations in integers, it is just necessary to verify before substracting, that the result will be non-negative. Moreover, for the comparation the function with bit vectors is:

$$\leq (a, b) := (b \,|\, carries) - a) \,\&\, carries) \neq carries) \tag{8}$$

Basically, we are 'adding' the carries to the element who is suposed to be greater or equal, namely b. In case that $a \leq b$, when substracting a to b, all the carries in the result should be setting to 1, in other case, then at least one of the positions of the carries will be 0. in Figure 3 it is showed an example of the use of bit vectors and the operations of addition and comparation.

Parikh Vectors				Bit Vectors															
				c =	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
a =	3	1	3	a' =	0	0	0	1	1	0	0	0	0	1	0	0	0	1	1
b =	2	2	3	b' =	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1
$a + b$	5	3	6	$a' + b'$	0	0	1	0	1	0	0	0	1	1	0	0	1	1	0
				$b'' = b' \mid c$	1	0	0	1	0	1	0	0	1	0	1	0	0	1	1
				$a'' = b'' - a'$	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0
$a_i \leq b_i$	F	T	T	$a'' \,\&\, c$	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
$a \leq b$	False			$a' \leq b'$	False														

Figure 3. Bit vectors corresponding to the Parikh vectors $a = \{3, 1, 3\}$ and $b = \{2, 2, 3\}$, the carry used is the bit vector c. Highlighted are the results of the binary operations addition and less or equal.

As it can be deduced, using of bit vectors instead of integer Parikh vectors can help to get a better performance of the operations which are executed many times in

a run of the algorithm. Many of the properties of bits arquitecture from computers are used for the algorithm.

3.2 All Matchings

So far it has been showed how to calculate the maximal matchings on a query, however, it can be of interest to find all the possible matches on a text. Therefore, in this section it is showed how the algorithm can be used to make this possible.

First, note that just using the functions *Expand* and *Shrink* the positions $L+1$ and R where the query has possibly a match are obtained, additionally, if after using both functions, $(L+1,\ R)$ is a match, then it is the shortest possible match starting at position $L+1$.

Second, note that the function *Refine* is used in order to make the match a maximal one, then after *Refine* it is known that there are no possible matchings starting at position $L+1$ and ending after position R, this is the longest possible match starting at position $L+1$.

Finally, using this two properties of the functions, we keep track of the position R after the function *Expand*, so if we have a match after *Shrink*, we use *Refine* to get the position R' and all the substrings starting at position $L+1$ and finishing between positions R and R' inclusive are matches, note that $R' - R \leq 2 * |\delta|$. After this we make a new cycle of *Expand* starting at position $L+2$ and ending at position R.

In Figure 4 the pseudocode of the All Matching algorithm is presented.

Input: A Parikh vector q a vector of possible errors δ
Output: A set $Matches$ with all the ocurrences of q in s allowing the error δ

```
1: L ← 0, R ← 0, R' ← 0, Matches ← ∅
2: while L < n − |q − δ| and R < n do
3:     if q − δ ⋦ p(s[L + 1, R]) then
4:         R ← Expand(L, q − δ)
5:     L ← Shrink(R, q + δ)
6:     if p(s[L + 1, R] ≥ q − δ) then
7:         R' ← Refine(L, q + δ)
8:         for j = R to R' do
9:             add (L + 1, j) to Matches
10:        L ← L + 1
11: return Matches
```

Figure 4. Pseudocode of the algorithm to calculate all the matchings for the δ - Approximate Jumbled Pattern Matching

3.3 Min-Error Matching

In some cases it is not of interest to find the maximal matchings nor all the possible matchings in a text, but the closest matchings to the query q, this is what we call a Min-Error Matching.

More formally, an occurrence $(i,\ j)$ is said to have minimum error, if neither $(i-1,\ j)$, $(i+1,\ j)$, $(i,\ j-1)$ nor $(i,\ j+1)$ is a match or is an occurrence of some Parikh vector q' such that $|q - q'| \leq |p(s_i, \ldots, s_j) - q|$, in other words, if when we extend or shrink the window of the occurrence the error is bigger. this can be made finding all the possible matchings with the algorithm and then calculating the error

of each of them, this would have a cost of $O(2n|\delta|\sigma)$. If $|\delta|$ were of $O(n)$, then the number of possible matchings in worst case would be $O(n^2)$, nevertheless, in practice $|\delta|$ is $O(1)$, so the time worst case running of finding the Min-Error matches is still $O(n\sigma)$.

Example 3. Consider again the alphabet $\Sigma = \{a, b, c\}$ and the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ over the string $s = ccabbcbaaccbbaaccbabab$. It has 6 min-error occurrences, namely $(6, 11)$, $(8, 12)$, $(8, 14)$, $(9, 16)$, $(11, 17)$, and $(14, 19)$ with errors 1, 2, 2, 1, 2 and 1 respectively. In Figure 5 the results of finding the Maximal Matchings are compared with he results finding the Min-Error Matchings of the query in s.

Max Match	Parikh Vector	Error
(5, 11)	{2, 2, 3}	2
(6, 12)	{2, 2, 3}	2
(8, 17)	{4, 2, 4}	3
(13, 19)	{3, 2, 2}	2
(14, 21)	{4, 2, 2}	3

Min-Err Match	Parikh Vector	Error
(6, 11)	{2, 1, 3}	1
(8, 12)	{2, 1, 2}	2
(8, 14)	{3, 2, 2}	2
(9, 16)	{3, 2, 3}	1
(11, 17)	{2, 2, 3}	2
(14, 19)	{3, 1, 2}	1

Figure 5. Maximal occurrences and Min-Error ocurrences (left and right) of the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ in the text *ccabbcbaaccbbaaccbabab* and the respectively error of each match.

4 Experimental Results

All the experiments were run on a computer, with an intel i5 1.70 GHz CPU with 4 GB of RAM and running Ubuntu 14.04 LTS 64-Bit. The codes were writen uniformelly in C++ and compiled with Codeblocks.

The datasets were made taking randomly generated texts of different lengths with alphabets of size 2, 4, 8, 26 and 94. Each query was made also random and was tested with diferent possible deltas. Although it is known that random texts and random queries are not closed to the reality, it a good approximation to test the performance in average of the algorithm. [10]

Essentially, we compared our implementation against the implementation based on Wavelet trees using the most efficient implementation of these [15]. For both implementations, the tests were run several times and the relative differences on the average time from each test were taken.

In Figure 6 are presented the results of the tests, here it can be seen the improvement of our algorithm over other solutions, as it can be seen that there are cases where the results are even 3.5 times faster. In addition, it can be seen that the best results are for larger alphabets, this can be clearly deduced because of the advantages of the bit vectors that were used in our implementation of the algorithm.

5 Conclusions

We presented a new implementation of an algorithm to solve the δ - Approximate Jumbled Pattern Matching, this implementation speeds up searchings using tables of

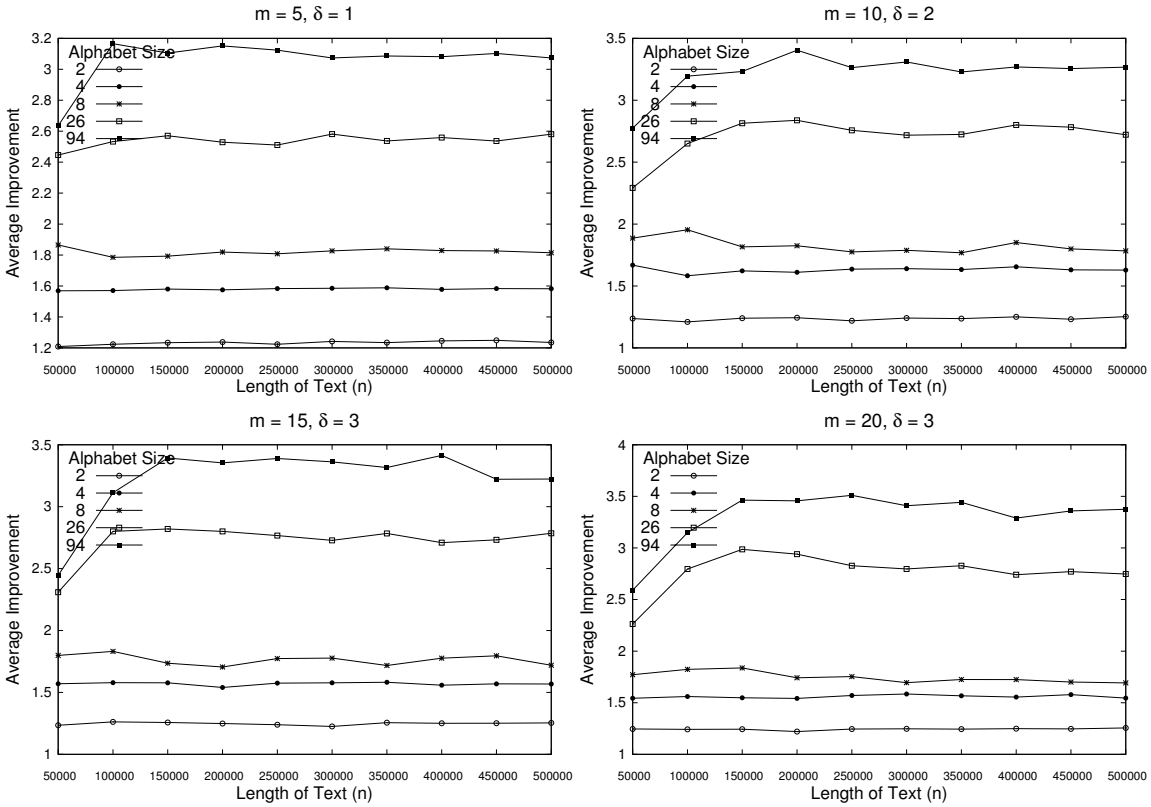


Figure 6. Average relative speed up of our implementation in random texts against efficient Wavelet tree implementation.

indexes and bit vectors. Our solution has a better performance when the alphabet is bigger. Also we showed how the algorithm can be used to calcuate all possible matchings and with it calculate also the Min-Error matchings in a text, though we are interested in a development of an algorithm which can find Min-Error matchings in a text without calculating the error for all possible matchings.

Although for binary alphabets our solution makes a practical improvement on the δ - Approximate Jumbled Pattern Matching problem, several improvements can still be done because of the many properties Parikh vectors have with binary alphabets, and does not have on general alphabets [1,13]. Even though the binary problem has been studied before on the Exact version, it has not been done on the Approximate Version.

References

1. G. Badkobeh, G. Fici, S. Kroon, and Zs. Lipták: *Binary jumbled string matching for highly run-length compressible texts*. Information Processing Letters, 113(17) 2013, pp. 604–608.
2. G. Benson: *Composition Alignment*, in Algorithms in Bioinformatics, vol. 2812 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 447–461.
3. S. Böcker: *Sequencing from Compomers: Using Mass Spectrometry for DNA de novo Sequencing of 200+ nt*. Journal of Computational Biology, 11(6) 2004, pp. 1110–1134.
4. S. Böcker: *Simulating multiplexed SNP discovery rates using baes-specific cleavage and mass spectrometry*, in European Conference on Computational Biology 2006 (ECCB 2006), vol. 23, 2006, pp. e5–e11.

5. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching*, in Fun with Algorithms, vol. 6099 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 89–101.
6. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On Approximate Jumbled Pattern Matching in Strings.* Theory of Computing Systems, 50(1) 2012, pp. 35–51.
7. A. Butman, R. Eres, and G. M. Landau: *Scaled and permuted string matching.* Information Processing Letters, 92(6) 2004, pp. 293–297.
8. D. Cantone and S. Faro: *Efficient online Abelian pattern matching in strings by simulating reactive multi-automata*, in Proceedings of the Prague Stringology Conference 2014, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2014, pp. 30–42.
9. F. Cicalese, G. Fici, and Zs. Lipták: *Searching for jumbled patterns in strings*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 105–117.
10. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt: *Fast Pattern Matching in Strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
11. L.-K. Lee, M. Lewenstein, and Q. Zhang: *Parikh Matching in the Streaming Model*, in String Processing and Information Retrieval, vol. 7608 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 336–341.
12. J. Mendivelso and Y. Pinzón: *A Novel Approach to Approximate Parikh Matching for Comparing Composition in Biological Sequences*, in Proceedings of the 6th International Conference on Bioinformatics and Computational Biology (BICoB 2014), 2014.
13. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Information Processing Letters, 110(18–19) 2010, pp. 795–798.
14. G. Navarro: *Multiple Approximate String Matching by Counting*, in Proc. WSP'97, Carleton University Press, 1997, pp. 125–139.
15. G. Navarro: *Wavelet Trees for All*, in Combinatorial Pattern Matching, vol. 7354 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 2–26.

Tuning Algorithms for Jumbled Matching

Tamanna Chhabra, Sukhpal Singh Ghuman, and Jorma Tarhio

Department of Computer Science
Aalto University
P.O. Box 15400, FI-00076 Aalto, Finland
firstname.lastname@aalto.fi

Abstract. We consider the problem of jumbled matching where the objective is to find all permuted occurrences of a pattern in a text. Besides exact matching we study approximate matching where each occurrence is allowed to contain at most k wrong or superfluous characters. We present online algorithms applying bit-parallelism to both types of jumbled matching. Most of our algorithms are variations of earlier algorithms. We show by practical experiments that our algorithms are competitive with the previous solutions.

Keywords: jumbled matching, Abelian matching, permutation matching, approximate string matching, comparison of algorithms, counting algorithm, algorithm engineering

1 Introduction

String matching [17] is a common problem in Computer Science. Let $T = t_1t_2 \cdots t_n$ and $P = p_1p_2 \cdots p_m$ be text and pattern respectively, over a finite alphabet Σ of size σ. The task of exact string matching is to find all the occurrences of P in T, i.e. all positions i such that $t_it_{i+1} \cdots t_{i+m-1} = p_1p_2 \cdots p_m$. In approximate string matching the objective is to find all substrings of T whose edit distance to P is at most k or which have at most k mismatches with P, where $0 \leq k < m$.

Jumbled matching [5,7] (also known as Abelian matching or permutation matching) is an interesting variation of string matching. The task is to find all substrings of T which are permutations of P. For instance, a permutation of `abcb` occurs in `cdfbbacda`. Jumbled matching can be formalized with Parikh vectors [19]. The Parikh vector $p(S)$ of a string S over a finite ordered alphabet is defined as the vector of multiplicities of the characters, for example $p(S)$ is (1,2,1,0) for $S =$ `abcb` in $\Sigma = \{a, b, c, d\}$.

Jumbled matching has applications in many areas such as alignment of strings [1], SPN discovery [3], discovery of repeated patterns [10], in the interpretation of mass spectrometry data [2]. In case of discovery of repeated patterns [10], jumbled matching algorithms can be used to solve the problem of local alignment of genes. Also the problem of matching of protein domain clusters [10] can be solved by these algorithms as the clusters have common functionality even though appear in different orders. In the field of interpretation of mass spectrometer [2], permutation matching is used to find the strings having the same spectra. Mass spectra are simulated for every potential sequence and comparing the resulting simulated spectra against the measured mass spectrum. In addition to that, permutation matching can be used in SNP [3] and mutation discovery which are based on base-specific cleavage of DNA or RNA and mass spectrometry. Composition alignment [1] is a process of matching strings whether they have equal length and same nucleotide content.

Tamanna Chhabra, Sukhpal Singh Ghuman, Jorma Tarhio: Tuning Algorithms for Jumbled Matching, pp. 57–66.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

Simple counting solutions [13,15,16] for jumbled matching work in linear time. The idea is to scan the text forward while maintaining counts of characters in a sliding alignment window of T. Originally, these counting algorithms were developed as filtration methods for online approximate string matching, but they recognize jumbled patterns as a side-effect when no errors are allowed. Also many other algorithms [4,6,9,12] have been introduced for jumbled matching. In this paper, we introduce new algorithms for jumbled matching and compare their efficiency with the previous solutions. Most of our algorithms are variations of earlier algorithms.

Besides traditional jumbled matching, we also consider approximate jumbled matching. We define an approximate permutation as follows. The string P' is a k-approximate permutation of P, $0 \leq k < m$, if $|P'| = |P| = m$ holds and

$$\sum_{c \in set(P')} \max(cc(P', c) - cc(P, c), 0) \leq k,$$

where $set(P')$ is the set of characters in P' and $cc(u, c)$ is the number of occurrences of a character c in a string u. Our definition of approximate jumbled matching is different from the one presented by Burcsi et al. [5]. Ejaz [9] considers also other cost models. Note that according to our definition, a 0-approximate permutation is an exact permutation. We will present linear and sublinear algorithms for both exact and approximate jumbled matching. By sublinear we mean those algorithms which are on average able to skip a part of the text.

In the pseudocodes of the algorithms, we use C-like notations '&' and '>>' representing bitwise operations AND and right shift, respectively. The size of the computer word is denoted by w. All the bitvectors and bit masks contain w bits.

Our main emphasis is on the practical efficiency of the algorithms which is demonstrated by experimental results. In almost every tested case, the best of our algorithms was faster than any previous solution, and in many cases even doubling the speed of the best previous solution.

The paper is organized as follows. Section 2 describes previous solutions for jumbled matching, Section 3 presents our solutions, Section 4 presents and discusses the results of practical experiments, and Section 5 concludes the article.

2 Previous solutions

Grossi & Luccio's and Navarro's solutions [13,15,16] are based on the frequency of characters occurring in the pattern and in an alignment window. These methods solve this problem in linear time. Navarro's counting algorithm is based on a sliding window approach. Alg. 1 presents the main loop of Navarro's algorithm called Count in the following. The variable C holds the additive inverse of the number of wrong or extra characters in an alignment window of m characters. The initial value of C is $-m$. The array A maintains the character counts of the alignment window such that $A[c]$ is $cc(t_{i-m+1} \cdots t_i, c) - cc(P, c)$. Before the main loop, the character counts of the first alignment window are collected to A, and the variable C is updated respectively.

Alg. 1 (The main loop of Count)
while $i \leq n$ do
  if $C \geq 0$ then report occurrence
  $A[t_{i-m}] \leftarrow A[t_{i-m}] + 1$
  if $A[t_{i-m}] > 0$ then $C \leftarrow C - 1$
  if $A[t_i] > 0$ then $C \leftarrow C + 1$
  $A[t_i] \leftarrow A[t_i] - 1$
  $i \leftarrow i + 1$

Grossi & Luccio's solution maintains a queue of characters which grows with acceptable characters until the length is m which means that an occurrence of P has been found. Another counting algorithm has been proposed by Grabowski et al. [12].

In addition to exact jumbled matching, Navarro's and Grossi & Luccio's algorithms can directly be applied to approximate jumbled matching as well, because a match candidate for the k mismatches problem is a k-approximate permutation of P. The initial value of C in Count is $k - m$ in the approximate case.

Besides a single pattern algorithm, Navarro [16] also presented a multipattern variation for patterns of equal length. Alg. 2 shows the main loop of this algorithm called Mcount in the following. Each pattern has a count variable (or a bin) of its own, and a field of $d + 1$ bits is allocated for it in D, a bitvector of w bits. The bitvector $E[c]$ holds a field of $d+1$ bits for each pattern. The initial values of fields in $E[c]$ and D are $2^d + cc(P_j, c) - 1$ and $2^d - (m - k)$, respectively, for the jth pattern. Before the main loop, the character counts of the first alignment window are collected to E, and D is updated respectively. During scanning, the value for the field of $E[c]$ for the jth pattern is $2^d + cc(P_j, c) - cc(t_{i-m+1} \cdots t_i, c) - 1$. The bit mask F holds one in the most significant bit of every field in D. The bit mask I holds one in the least significant bit of every field in $E[c]$. The operation $(E[c] >> d)\,\&\,I$ extracts the most significant bits of $E[c]$. The condition $D\,\&\,F \neq 0$ means that at least one overflow bit is set in D, i.e. $m - k$ acceptable characters of at least one pattern have been found. In the case of a single pattern, this is enough to recognize an occurrence. In the case of two or more patterns, a verification step is needed because the condition $D\,\&\,F \neq 0$ does not specify which pattern matches.

Alg. 2 (The main loop of Mcount)
while $i \leq n$ do
  if $D\,\&\,F \neq 0$ then verify occurrence
  $c \leftarrow t_{i-m}$
  $E[c] \leftarrow E[c] + I$
  $D \leftarrow D - (E[c] >> d)\,\&\,I$
  $c \leftarrow t_i$
  $D \leftarrow D + (E[c] >> d)\,\&\,I$
  $E[c] \leftarrow E[c] - I$
  $i \leftarrow i + 1$

Cantone and Faro [6] presented the BAM algorithm (Bit-parallel Abelian Matcher) which applies bit-parallelism and backward scanning of the alignment window. Alg. 3 shows the main loop of BAM. A field of $g(c) = \lceil \log cc(P, c) \rceil + 2$ bits is reserved for each character c appearing in P. As in Mcount, the most significant bit of each field is a kind of overflow bit. The initial value of the field is $2^{g(c)-1} - cc(P, c) - 1$ which means that $cc(P, c) + 1$ occurrences trigger the overflow bit. The adaptive width of bit fields make possible to handle longer patterns than a fixed width. Moreover, there

is a special field of one bit for characters not present in P. The bit mask $M[c]$ holds one in the least significant bit of the field of character c. The bit mask I holds the initial values of the fields, and the bit mask F holds one at each overflow bit.

Alg. 3 (The main loop of BAM)
while $i \leq n - m$ do
  $D \leftarrow I$; $j \leftarrow i + m - 1$
  while $j \geq i$ do
    $D \leftarrow D + M[t_j]$
    if $D \,\&\, F \neq 0$ then break
    $j \leftarrow j - 1$
  if $j < i$ then
    report occurrence
    $i \leftarrow i + 1$
  else $i \leftarrow j + 1$

Ejaz [9] proposed several algorithms for jumbled matching. One of them utilizes backward scanning of the alignment window. Moreover, Burcsi et al. [4] introduced a light indexing approach with linear construction time and with sublinear expected query time.

3 New solutions

We have designed various solutions for the exact and approximate jumbled matching. We explain them in the following subsections. Most of the algorithms are variations of Count, Mcount, or BAM.

3.1 Variations of BAM

If the pattern is long, w bits is not enough to hold a distinct bin for each character appearing in the pattern. We made BAMs, a variation of BAM where some bins are shared. In BAMs, characters for bins are selected circularly from the pattern in the right-to-left order. This is a kind of alphabet reduction. Then instead of "report occurrence" in Alg. 3, each match candidate should be verified.

Then we present two other algorithms that are modifications of BAM (Alg. 3). Alg. 4 is approximate BAM (ABAM for short) and Alg. 5 is enhanced BAM with 2-grams (BAM2 for short), respectively. In ABAM, $F[c]$ is the overflow bit of character c. The variable C counts errors. The width of the field for the character c is $\lceil \log(\max(cc(P, c), k)) \rceil + 2$. The width of the field for characters not present in P is $\lceil \log k \rceil + 2$. $M[c]$ and I are the same as in BAM. The enxpression "if $D \,\&\, F[t_j] \neq 0$ then 1 else 0" can be implemented as $(D \,\&\, F[t_j])$ && 1 in C.

Alg. 4 (The main loop of ABAM)
while $i \leq n - m$ do
  $D \leftarrow I$; $C \leftarrow 0$; $j \leftarrow i + m - 1$
  while $j \geq i$ do
    $D \leftarrow D + M[t_j]$
    $C \leftarrow C +$ (if $D \,\&\, F[t_j] \neq 0$ then 1 else 0)
    if $C > k$ then break
    $j \leftarrow j - 1$
  if $j < i$ then
    report occurrence
    $i \leftarrow i + 1$
  else $i \leftarrow j + 1$

Alg. 5 shows the main loop of BAM2 for patterns of even length. BAM2 handles a 2-gram at a time. BAM2 has a separate loop for patterns of even and odd lengths. The loop for patterns of odd length has two lines more because the remaining leftmost character of the alignment window must be handled in a different way. Typically q-grams are used in string matching to process the right end of the alignment window. BAM2 processes the whole window with 2-grams (except the leftmost character in the case of odd m). This is beneficial because the alignment window is scanned on average further to the left in jumbled matching than in ordinary string matching. Moreover, 2-grams instead of single characters are read in our implementation of BAM2.

Alg. 5 (The main loop of BAM2)
while $i \leq n - m$ do
  $j \leftarrow i + m - 3$
  $D \leftarrow I + M_2[t_{j+1}, t_{j+2}]$
  do
    $D \leftarrow D + M_2[t_{j-1}, t_j]$
    if $D \,\&\, F = 0$ then break
    $j \leftarrow j - 2$
  until $j \geq i$
  if $j < i$ then
    report occurrence
    $i \leftarrow i + 1$
  else $i \leftarrow j$

BAM2 reads four characters before testing D. As a consequence, the minimum width of a bit field is four bits instead of two. The width of the field for characters not present in P is three bits. The array M_2 is precomputed as follows: $M_2[c_1, c_2] = M[c_1] + M[c_2]$.

For small alphabets we use BAM2 as presented in Alg. 5. For large alphabets we use BAM2 with the same bin sharing technique as applied in BAMs.

3.2 Other variations

Alg. 6 presents the main loop of EBL (short for "Exact Backward for Large alphabets"). EBL is based on SBNDM2 [8], which is a sublinear bit-parallel algorithm for exact string matching. Instead of representing occurrence vectors of characters, the array B states if the character c is present in the pattern: $B[c] = 1$ if c is present, otherwise $B[c] = 0$. As in SBNDM2, two characters are read before the first test in an alignment window. The update step of the state variable D is simply $D = D \,\&\, B[t_{i+j-1}]$. When the alignment window contains only acceptable characters,

the window is a match candidate, which should be verified. Whenever a forbidden text character is found, the alignment window is moved forward over that text position.

Alg. 6 (The main loop of EBL)
while $i \leq n - m$ do
  $j \leftarrow m - 1$
  $D = B[t_{i+j}] \,\&\, B[t_{i+j+1}]$
  while $D \neq 0$ and $j > 0$ do
    $D \leftarrow D \,\&\, B[t_{i+j-1}]$
    $j \leftarrow j - 1$
  if $D = 1$ then verify occurrence
  $i \leftarrow i + j + 1$

Alg. 7 presents the main loop of EFS (short for "Exact Forward for Small alphabets"). Like in Count and other algorithms of forward type, the first alignment window is processed before the main loop. The bitvector D has a field of d bits[1] initially $2^{d-1} - cc(P, c) - 1$ for each character c appearing in P. The characters not in P have a joint field of one bit. Like in BAM, D is tested with a mask F which has one at each overflow bit. $M[c]$ is a bit mask having one at the least significant bit of the field of character c.

Alg. 7 (The main loop of EFS)
while $i \leq n$ do
  if $D \,\&\, F = 0$ then report occurrence
  $D \leftarrow D + M[t_i] - M[t_{i-m}]$
  $i \leftarrow i + 1$

Alg. 8 presents the main loop of AFL (short for "Approximate Forward for Large alphabets"). AFL is a modification of Mcount tuned for a single pattern. The array E is the same as in Mcount in the case of a single pattern as well as the offset d. The initial value of the counter C is $k - m$. Like in the other algorithms of forward type, the first alignment window is processed before the main loop.

Alg. 8 (The main loop of AFL)
while $i \leq n$ do
  if $C \geq 0$ then report occurrence
  $E[t_{i-m}] \leftarrow E[t_{i-m}] + 1$
  $C \leftarrow C + (E[t_i] >> d) - (E[t_{i-m}] >> d)$
  $E[t_i] \leftarrow E[t_i] - 1$
  $i \leftarrow i + 1$

Alg. 9 presents the main loop of ABS (short for "Approximate Backward for Small alphabets"). The bitvector D holding the counters (or bins) of characters is initialized for each alignment window. D has a field of d bits initially $2^{d-1} - cc(P, c) - 1$ for each character c appearing in P and a joint field for characters not in P. The offset $o[c]$ is used to move the overflow bit of the corresponding counter to the right end of a word. $M[c]$ is a bit mask having one at the least significant bit of the field of character c.

[1] All the algorithms of Section 3.2 were implemented before the appearance of [6]. So we use here bit fields of fixed width. When shared bins are used, the benefit of adaptive width is smaller than without them.

Alg. 9 (The main loop of ABS)
while $i \leq n$ do
  $D \leftarrow I$; $C \leftarrow 0$; $j \leftarrow i - m$
  while $C \leq k$ and $i > j$ do
    $D \leftarrow D + M[t_i]$
    $C \leftarrow C + (D >> o[t_i]) \,\&\, 1$
    $i \leftarrow i - 1$
  if $C \leq k$ then report occurrence
  $i \leftarrow i + m + 1$

ABL (short for "Approximate Forward for Large alphabets") is a slight modification of ABS. If there are not enough bins for all the characters of the pattern, we apply the same sharing technique as in BAMs. Then instead of "report occurrence" on the last but one line of Alg. 9, each match candidate should be verified.

4 Experiments

m	k	Count	Mcount	BAM	BAMs	BAM2a	ABAM	EBL	AFL	ABL
5	0	2.370	1.960	1.183	1.206	0.749	1.420	0.739	1.781	1.482
10	0	2.370	1.960	0.861	0.863	0.297	1.021	0.638	1.778	1.067
20	0	2.376	1.962	0.564	0.582	0.247	0.689	0.544	1.779	0.701
30	0	2.373	1.959	0.449	0.427	0.261	0.514	0.488	1.778	0.514
50	0	2.377	1.958	—	0.301	0.234	—	0.524	1.778	0.413
100	0	2.378	1.964	—	0.204	0.157	—	1.360	1.779	0.360
5	1	2.373	1.960	—	—	—	3.500	—	1.779	4.231
10	1	2.373	1.963	—	—	—	1.844	—	1.783	2.230
20	1	2.377	1.968	—	—	—	1.073	—	1.777	1.257
30	1	2.377	1.961	—	—	—	—	—	1.779	0.978
50	1	2.374	1.960	—	—	—	—	—	1.774	0.780
100	1	2.378	1.961	—	—	—	—	—	1.778	0.736
5	2	2.373	1.961	—	—	—	6.763	—	1.779	9.438
10	2	2.370	1.961	—	—	—	3.372	—	1.777	5.070
20	2	2.376	1.964	—	—	—	1.554	—	1.778	2.510
30	2	2.374	1.966	—	—	—	—	—	1.779	1.944
50	2	2.380	1.960	—	—	—	—	—	1.779	1.582
100	2	2.379	1.964	—	—	—	—	—	1.779	1.596
5	3	2.370	1.964	—	—	—	8.698	—	1.781	14.790
10	3	2.374	1.959	—	—	—	5.840	—	1.780	11.043
20	3	2.376	1.956	—	—	—	—	—	1.779	5.747
30	3	2.376	1.958	—	—	—	—	—	1.780	4.604
50	3	2.374	1.961	—	—	—	—	—	1.779	3.563
100	3	2.379	1.962	—	—	—	—	—	1.779	3.520

Table 1. Execution times of algorithms (in seconds) for English data.

The tests were run on Intel 2.70 GHz i7 processor with 16 GB of memory. All the algorithms were implemented in C and run in the 64-bit mode in the testing framework of Hume and Sunday [14]. Three types of texts were used for testing: English and protein representing large alphabets as well as DNA representing small alphabets.

m	k	Count	Mcount	BAM	BAM2	ABAM	EFS	AFL
5	0	2.724	2.321	3.279	1.559	3.987	1.138	2.150
10	0	2.722	2.326	2.851	1.761	3.511	1.118	2.151
20	0	2.721	2.324	2.419	1.626	3.184	1.118	2.154
30	0	2.722	2.330	2.091	1.430	2.902	1.126	2.159
50	0	2.720	2.324	2.060	1.297	3.074	1.117	2.153
100	0	2.727	2.327	2.240	1.276	3.632	1.111	2.160
5	1	2.723	2.378	—	—	8.250	—	2.154
10	1	2.721	2.326	—	—	7.483	—	2.144
20	1	2.718	2.323	—	—	6.318	—	2.154
30	1	2.719	2.330	—	—	5.204	—	2.160
50	1	2.721	2.323	—	—	4.833	—	2.158
100	1	2.719	2.324	—	—	4.841	—	2.158
5	2	2.720	2.322	—	—	9.907	—	2.146
10	2	2.720	2.324	—	—	11.593	—	2.157
20	2	2.724	2.326	—	—	10.857	—	2.146
30	2	2.723	2.329	—	—	8.836	—	2.159
50	2	2.721	2.323	—	—	7.762	—	2.158
100	2	2.712	2.324	—	—	6.734	—	2.153
5	3	2.727	2.322	—	—	8.638	—	2.154
10	3	2.720	2.324	—	—	14.146	—	2.154
20	3	2.723	2.323	—	—	15.888	—	2.154
30	3	2.712	2.327	—	—	13.558	—	2.159
50	3	2.724	2.322	—	—	11.582	—	2.154
100	3	2.720	2.324	—	—	9.443	—	2.153

Table 2. Execution times of algorithms (in seconds) for DNA data.

The English text is the KJV Bible (3.9 MB), the DNA text is 4.4 MB long and the protein text is 3.6 MB long. All the texts were taken from the Smart corpus [11]. For each text we had six sets of 200 patterns with lengths: $m = 5, 10, 20, 30, 50$, and 100.

As reference methods we used Count and Mcount [16] as well as BAM [6]. Mcount was run only with a single pattern. We have not shown the results of Grossi & Luccio's algorithm [13] because it was clearly slower than Count. Likewise, we did not test GFG [12] and SBA [9], because they were mostly slower than BAM in tests of [6].

Tables 1, 2 and 3 represent the average execution times in seconds for English, DNA, and protein data respectively, for $k = 0, 1, 2$, and 3. The results were obtained as an average of nine runs. The best time for each combination of m and k has been boxed. An empty cell means that 64 bits was not enough to hold the necessary counters at least for one of the 200 patterns.

From the results for English data in Table 1, it can be seen that EBL is fastest for shorter pattern length and BAM2a is fastest for remaining pattern lengths in the exact case ($k = 0$). For $k = 1, 2$, AFL is the fastest for short patterns and ABL for long patterns. As an exception, ABAM is fastest for $k = 1$ and $m = 20$. For $k = 3$, AFL is the fastest. Note that EBL gets its best time for $m = 30$. Its speed is decreasing for longer patterns because longer patterns produce more false matches which increase verification time.

From the results for DNA data in Table 2, it can be seen that EFS is clearly the fastest in the exact case and AFL in the approximate case. EFS works in a double speed when compared with the previous algorithms. Observe also that BAM2 is faster than BAM, even with a wide margin.

m	k	Count	Mcount	BAM	BAMs	BAM2a	ABAM	EBL	AFL	ABL
5	0	1.928	1.596	0.711	0.733	0.581	0.853	0.471	1.451	0.909
10	0	1.932	1.601	0.591	0.611	0.191	0.702	0.481	1.452	0.764
20	0	1.934	1.599	0.427	0.582	0.168	0.521	0.662	1.451	0.582
30	0	1.934	1.592	0.321	0.331	0.196	0.426	1.939	1.451	0.542
50	0	1.934	1.598	—	0.254	0.198	—	—	1.451	0.691
100	0	1.934	1.598	—	0.247	0.197	—	—	1.449	0.231
5	1	1.931	1.599	—	—	—	2.131	—	1.451	2.346
10	1	1.933	1.602	—	—	—	1.233	—	1.452	1.461
20	1	1.931	1.597	—	—	—	0.791	—	1.451	1.071
30	1	1.931	1.602	—	—	—	0.630	—	1.451	1.116
50	1	1.938	1.598	—	—	—	—	—	1.451	1.591
100	1	1.932	1.602	—	—	—	—	—	1.449	3.449
5	2	1.932	1.602	—	—	—	4.351	—	1.454	5.179
10	2	1.929	1.598	—	—	—	2.077	—	1.451	2.826
20	2	1.933	1.606	—	—	—	1.104	—	1.448	2.067
30	2	1.938	1.649	—	—	—	—	—	1.451	2.301
50	2	1.938	1.598	—	—	—	—	—	1.448	3.396
100	2	1.938	1.599	—	—	—	—	—	1.451	3.163
5	3	1.931	1.603	—	—	—	6.466	—	1.454	8.754
10	3	1.931	1.601	—	—	—	3.377	—	1.456	5.907
20	3	1.933	1.598	—	—	—	—	—	1.453	4.338
30	3	1.936	1.599	—	—	—	—	—	1.453	4.756
50	3	1.939	1.601	—	—	—	—	—	1.444	6.737
100	3	1.937	1.598	—	—	—	—	—	1.453	10.737

Table 3. Execution times of algorithms (in seconds) for protein data.

The results in Table 3 for protein data do not differ much from Table 1. EBL was very slow for $m > 30$ (results not shown), because then the number of forbidden characters is low.

The current implementation of ABAM does not contain shared bins. The test results suggest that ABAM with shared bins could be the winner with some new parameter combinations.

For all types of data, Mcount is considerably faster than Count. The obvious reason is that the main loop of Mcount contains only one if statement whereas the main loop of Count contains three. Relatively, the conditional instructions are slow in modern processors.

5 Concluding remarks

We introduced new variations jumbled matching algorithms. All the forward algorithms are clearly linear. The speed of their approximate versions do not depend on the value of k. It is not difficult to show that the backward algorithms are sublinear on average for small k and large m. The experimental results show that our algorithms are competitive with previous solutions. In almost every tested case, the best of our algorithms was faster than any previous solution, and in many cases even doubling the speed of the best previous solution. Especially the technique of shared bins showed to be useful for jumbled matching. We believe that there is still room to improve our results. E.g. more sophisticated character selection for shared bins may lead to faster solutions.

References

1. G. Benson: *Composition alignment*, in Proc. of the 3rd International Workshop on Algorithms in Bioinformatics 2003, pp. 447–461.
2. S. Böcker: *Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt.* Journal of Computational Biology, 11 (6) 2004, pp. 1110–1134.
3. S. Böcker: *Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry.* Bioinformatics, 23 (2) 2007, pp. 5–12.
4. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *Algorithms for jumbled pattern matching in strings.* Int. J. Found. Comput. Sci. 23 (2) 2012, pp. 357–374.
5. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On approximate jumbled pattern matching in strings.* Theory Comput. Syst. (MST) 50 (1) 2012, pp. 35–51.
6. D. Cantone and S. Faro: *Efficient online Abelian pattern matching in strings by simulating reactive multi-automata*, in J. Holub and J. Žďárek, eds., Proc. PSC 2014, pp. 30–42.
7. F. Cicalese, G. Fici, and Zs. Lipták: *Searching for jumbled patterns in strings*, in J. Holub and J. Žďárek, eds., Proc. PSC 2009, pp. 105–117.
8. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Information Processing Letters 110 (4) 2010, pp. 148–152.
9. E. Ejaz: *Abelian pattern matching in strings.* Ph.D. Thesis, Dortmund University of Technology 2010, `http://d-nb.info/1007019956`.
10. R. Eres, G. M. Landau, and L. Parida: *Permutation pattern discovery in biosequences.* Journal of Computational Biology, 11 (6) 2004, pp. 1050–1060.
11. S. Faro and T. Leqroc: *Smart: string matching algorithms research tool*, 2015, `http://www.dmi.unict.it/~faro/smart/`
12. S. Grabowski, S. Faro, and E. Giaquinta: *String matching with inversions and translocations in linear average time (most of the time).* Information Processing Letters 111 (11) 2011, pp. 516–520.
13. R. Grossi and F. Luccio: *Simple and efficient string matching with k mismatches.* Information Processing Letters 33 (3) 1989, pp. 113–120.
14. A. Hume and D. Sunday: *Fast string searching.* Software–Practice and Experience, 21 (11) 1991, pp. 1221–1248.
15. P. Jokinen, J. Tarhio, and E. Ukkonen: *A comparison of approximate string matching algorithms.* Software–Practice and Experience 26 (12) 1996, pp. 1439–1458.
16. G. Navarro: *Multiple approximate string matching by counting*, in R. Baeza-Yates, ed., Proc. 4th South American Workshop on String Processing 1997, pp. 125–139.
17. G. Navarro and M. Raffinot: *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, New York, NY, 2002.
18. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in Proc. 10th International Symposium on String Processing and Information Retrieval, vol. 2857 of Lecture Notes in Computer Science, 2003, pp. 80–93.
19. A. Salomaa: *Counting (scattered) subwords.* Bulletin of the European Association for Theoretical Computer Science (EATCS) 81, 2003, pp. 165–179.

Enhanced Extraction from Huffman Encoded Files

Shmuel T. Klein[1] and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@ariel.ac.il

Abstract. Given a file T, and the Huffman encoding of its elements, we suggest using a pruning technique for Wavelet trees that enables direct access to the i-th element of T by reordering the bits of the compressed file and using some additional space. When compared to a traditional Wavelet tree for Huffman Codes, our different reordering of the bits usually requires less additional storage overhead by reducing the need for auxiliary rank structures, while improving processing time for extracting the i-th element of T.

1 Introduction and previous work

Research in Lossless Data Compression was originally concerned with finding a good balance between the competing efficiency criteria of compressibility of the input, processing time and additional auxiliary storage for the involved data structures. Working directly with compressed data is now a popular research topic, including not only classical text but also various useful data structures, and with a wide range of possible applications. One of the fundamental components of these structures is known as a *Wavelet tree*, suggested by Grossi et al. [11], which has meanwhile become a subject of investigation in its own right, as ever more of its useful properties are discovered [7]. It is on enhancing the usefulness of the extract operation of Wavelet trees when applied to Huffman encoded text that we wish to concentrate in this paper.

The simple way to encode our digital data is by using some standard fixed length code, like ASCII. This has many advantages, for example, allowing direct access to the ith codeword for any i, which might be useful when partial or parallel decoding is required. However, fixed length codes are wasteful from the storage point of view, and have therefore been replaced in many applications by variable length codes. This may improve the compression performance, but at the price of losing the simple random access, because the beginning position of the ith codeword is the sum of the lengths of all the preceding ones.

A possible solution to allow random access to variable length codes is to divide the encoded file into blocks of size b codewords, and to use an auxiliary vector to indicate the beginning of each block. The time complexity of random access depends on the size b, as we can begin from the sampled bit address of the $\frac{i}{b}$th block to retrieve the ith codeword. This method thus suggests a processing time vs. memory storage tradeoff, since direct access requires decoding $i - \lfloor \frac{i}{b} \rfloor b$ codewords, i.e., less than b.

Brisaboa et al. [4] introduced directly accessible codes (DACs), based on Vbyte coding [20], in which the codewords represent integers. The Vbyte code splits the

Shmuel T. Klein, Dana Shapira: Enhanced Extraction from Huffman Encoded Files, pp. 67–77.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

$\lfloor \log x_i \rfloor + 1$ bits needed to represent an integer x_i in its standard binary form into blocks of b bits and prepends each block with a flag-bit as follows. The highest bit is 0 in the extended block holding the most significant bits of x_i, and 1 in the others. Thus, the 0 bits act as a comma between codewords. In the worst case, the Vbyte code loses one bit per b bits of x_i plus b bits for an almost empty leading block, which is worse than Elias-δ encoding. Using a space overhead of $O(\frac{n \log\log n}{b \log n})$, DACs achieve direct access to the i^{th} codeword in $O(\frac{\log(M)}{b})$ processing time, where M is the maximum integer to be encoded, and n is the size of the encoded file.

Another line of investigation led to the development of Wavelet trees, which allow direct access to any codeword, and in fact recode the compressed file into an alternative form. Wavelet trees can be defined for any prefix code and the tree structure is inherited from the tree usually associated with the code. The internal nodes of the Wavelet tree are annotated with bitmaps. The root holds the bitmap obtained by concatenating the *first* bit of each of the sequence of codewords in the order they appear in the compressed text. The left and right children of the root hold, respectively, the bitmaps obtained by concatenating, again in the given order, the *second* bit of each of the codewords starting with 0, respectively with 1. This process is repeated similarly on the next levels: the grand-children of the root hold the bitmaps obtained by concatenating the *third* bit of the sequence of codewords starting, respectively, with 00, 01, 10 or 11, if they exist at all, etc.

The data structures associated with a Wavelet tree for general prefix codes require some amount of additional storage (compared to the memory usage of the compressed file itself). Given a text string of length n over an alphabet Σ, the space required by Grossi et al.'s implementation can be bounded by $nH_h + O(\frac{n \log\log n}{\log_{|\Sigma|} n})$ bits, for all $h \geq 0$, where H_h denotes the hth-order empirical entropy of the text, which is at most $\log |\Sigma|$; processing time is just $O(m \log |\Sigma| + \text{polylog}(n))$ for searching any pattern sequence of length m. Multiary Wavelet trees replace the bitmaps by sequences over sublogarithmic sized alphabets in order to reduce the $O(\log |\Sigma|)$ height of binary Wavelet trees, and obtain the same space as the binary ones, but their times are reduced by an $O(\log \log n)$ factor. If the alphabet Σ is small enough, say $|\Sigma| = O(\text{polylog}(n))$, the tree height is a constant and so are the query times.

Brisaboa et al. [3] used a variant of a Wavelet tree on Byte-Codes. This induces a 128 or 256-ary tree, rather than a binary one, and the root of the Wavelet tree contains the first *byte*, rather than the first bit, of all the codewords, in the same order as they appear in the original text. The second level nodes then store the second byte of the corresponding codewords, and so on. The reordering of the compressed text bits becomes an implicit index representation of the text, which is empirically shown to be better than explicit main memory inverted indexes, built on the same collection of words, when using the same amount of space.

Külekci [15] suggested the usage of Wavelet trees for *Elias* and *Rice* variable length codes. The method is based on handling separately the unary and binary parts of the codeword in different strings so that random access is supported in constant time. As an alternative, the usage of a Wavelet tree over the lengths of the unary section of each Elias or Rice codeword is proposed, while storing their binary section, allowing direct access in time $\log r$, where r is the number of distinct unary lengths in the file.

Recently Klein and Shapira [14] adapted the Wavelet tree to Fibonacci Codes, so that in addition to supporting direct access to the Fibonacci encoded file, the compression savings when compared to the original Fibonacci compressed file are

increased. We use a similar approach in this paper and prune the traditional Wavelet trees for general prefix codes without losing the direct access property. The topology of the reduced Wavelet tree is a *Skeleton Huffman* tree suggested by Klein [13], so that there are fewer internal nodes, and shorter paths from the root to the leaves, resulting in better processing time and less memory storage. This compact representation of Huffman trees was also used for improving the processing time for compressed pattern matching [19].

The skeleton Huffman tree used herein groups alphabet symbols according to their frequencies. A similar, yet different, alphabet partition according to frequencies has already been suggested by Gagie et al. [8], who study the problem of efficient representation of prefix codes, under the assumption that the maximum codeword length is $O(w)$, where w is the length of a machine word. They divide the alphabet into frequent and rare characters according to their Huffman codeword length, and store information just for the frequent ones, while the rare ones are lexicographically sorted. Using a multiary Wavelet tree, constant time encoding and decoding is achieved for small enough alphabets, at the price of increasing the codeword length of the rare characters, hurting the optimality of the Huffman code. Our approach is designed for all sizes of alphabets and the optimality of the Huffman codewords is retained at the price of slower processing.

Another data structure based on partitioning the alphabet into group of characters of similar frequencies is due to Barbay et al. [1]. This data structure stores the text in $nH_0 + o(n)(H_0 + 1)$ bits and supports operations in worst-case time $O(\log\log|\Sigma|)$ and average time $O(\log H_0)$. The sequences of sub-alphabet identifiers are stored in a multiary Wavelet tree, while the subsequences corresponding to each group are stored in uncompressed format.

Many of the data structures mentioned above use efficient access to bit vectors based on fast implementations of operations known as rank and select. These are defined for any bit vector B and bit $b \in \{0, 1\}$ as:

$\mathsf{rank}_b(B, i)$ − returns the **number** of occurrences of b up to and including position i;
$\mathsf{select}_b(B, i)$ − returns the **position** of the ith occurrence of b in B.

Note that $\mathsf{rank}_{1-b}(B, i) = i - \mathsf{rank}_b(B, i)$, thus, only one of the two, say, $\mathsf{rank}_1(B, i)$ needs to be computed. However, for the select operation the structures for both $\mathsf{select}_0(B, i)$ and $\mathsf{select}_1(B, i)$ are necessary [16]. Jacobson [12] showed that rank, on a bit-vector of length n, can be computed in $O(1)$ time using $n + O(\frac{n \log\log n}{\log n}) = n + o(n)$ bits.

It is important to stress that the overhead $o(n)$ of the rank and select data structures for a bitmap of size, say, $n = 2^{32}$ is about $0.66n$, which is not at all negligible. We suggest to reduce the size of the Wavelet tree without hurting the direct access capabilities. Methods proposed in [10] suggest practical implementations for rank and select, reducing the storage overhead to merely a few percent, at the price of losing the constant time access but with only a negligible increase in processing time. By applying our suggested strategy, these implementations can further be improved.

The $\mathsf{select}_b(B, i)$ operation can be done by applying binary search on the index j so that $\mathsf{rank}_b(B, j) = i$ and $\mathsf{rank}_b(B, j-1) = i-1$. As for the constant time solution for select [5], the bitmap B is partitioned into blocks, similar to the solution for the rank operation. Other efficient implementations are due to Raman et al. [18], Okanohara and Sadakane [17], Barbay et al. [2] and Navarro and Providel [16]. We refer to the thesis of Clark [5] for more details.

The rest of the paper is organized as follows. Section 2 deals with random access to Huffman encoded files, using Wavelet trees especially adapted to Huffman compressed files. Section 3 improves the self-indexing data structure by pruning the Wavelet tree using a skeleton Huffman tree. Section 4 further improves the overhead storage by pruning the Wavelet tree even further by means of a reduced skeleton tree. Finally, Section 5 concludes.

2 Random Access to Huffman Encoded Files

Recall that the binary tree T_C corresponding to a prefix code C is defined as follows: we imagine that every edge pointing to a left child is labeled 0 and every edge pointing to a right child is labeled 1; each node v is associated with the bit string obtained by concatenating the labels on the edges on the path from the root to v; finally, T_C is defined as the binary tree for which the set of bit strings associated with its leaves is the code C.

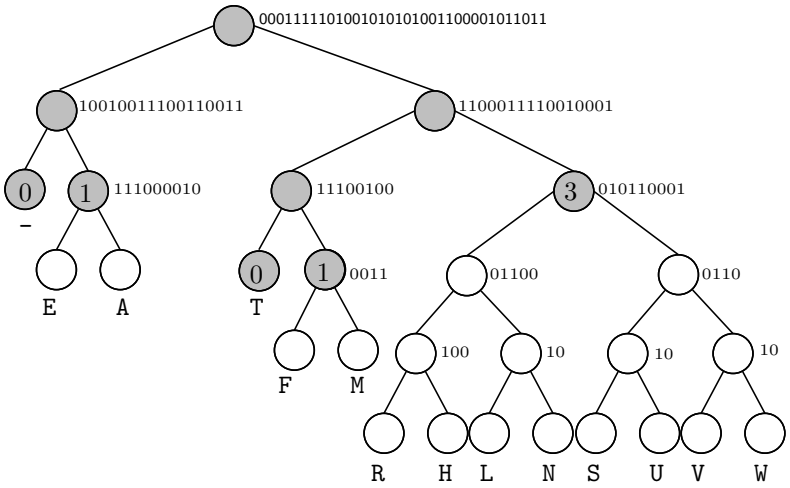


Figure 1. The Wavelet tree induced by the canonical Huffman tree corresponding to the frequencies {8,5,4,4,2,2,2,1,1,1,1,1,1,1} of {`-`,`E`,`A`,`T`,`F`,`M`,`R`,`H`,`L`,`N`,`S`,`U`,`V`,`W`}, respectively, assigned to the leaves, left to right.

A Huffman tree is *canonical* if, when scanning its leaves from left to right, they appear in non-decreasing order of their depth. To build a canonical tree, Huffman's algorithm is only used for generating the lengths ℓ_i of the codewords, and the ith codeword then consists of the first ℓ_i bits immediately to the right of the "binary point" in the infinite binary expansion of $\sum_{j=1}^{i-1} 2^{-\ell_j}$, for $1 \leq i \leq n$ [9].

As mentioned above, the nodes of the Wavelet tree are annotated by bitmaps. These bitmaps can be stored as a single bit stream by concatenating them in order of any predetermined top-down tree traversal, such as depth-first or breadth-first. No delimiters between the individual bitmaps are required, since we can restore the tree topology along with the bitmaps lengths at each node once the size n of the text is given in the header of the file. Figure 1 depicts the canonical Huffman tree for the example text $T =$ `A--HUFFMAN--WAVELET--TREE--MATTERS`. The Wavelet tree of our running example is the entire figure including the annotating bitmaps. It should be noted that the shape of the traditional Wavelet tree is not restricted to the underlying canonical Huffman tree. For any distribution, there are many different Huffman trees, and for some distributions, there might even exist Huffman trees of different depths.

Different topologies would imply different Wavelet trees and for convenience, we refer to the canonical one for the discussion in the next sections.

The algorithm for extracting the i-th element of the text T by means of a Huffman Wavelet tree rooted by v_{root} is given in Figure 2, using the function call extract(v_{root},i). B_v denotes the bitmap belonging to vertex v of the Wavelet tree, and $\cdot$ denotes concatenation. Computing the new index in the following bitmap is done by the rank operation in lines 2.1.3 and 2.2.3. The decoding of the codeword cw in line 3 by means of the decoding function $\mathcal{D}$ can be done by a preprocessed lookup table.

```
extract(v, i)
1     cw ⟵ ϵ
2     while v is not a leaf
2.1       if B_v[i] = 0 then
2.1.1         v ⟵ left(v)
2.1.2         cw ⟵ cw · 0
2.1.3         i ⟵ rank_0(B_v, i)
2.2       else
2.2.1         v ⟵ right(v)
2.2.2         cw ⟵ cw · 1
2.2.3         i ⟵ rank_1(B_v, i)
3     return D(cw)
```

Figure 2. Extracting the i-th element of T from a Wavelet tree rooted at v.

3 Enhanced Direct Access

A *Skeleton* Huffman tree [13], or sk-tree for short, is a canonical Huffman tree from which all full subtrees of depth $h \geq 1$ have been pruned. Thus, a path from the root to a leaf of an sk-tree may correspond to a prefix of several codewords of the original Huffman tree. The prefix is the shortest necessary in order to identify the length of the current codeword. A leaf, v, of the skeleton tree contains the height, $h(v)$, of the subtree that has been pruned ($h(v) = 0$ for leaves that were also leaves in the canonical Huffman tree). In Figure 1, the sk-tree nodes are colored in gray, and the numbers $h(v)$ are given in the leaves of the sk-tree.

We adjust the Wavelet tree to Huffman skeleton codes in the following way. The shape of the Wavelet tree will be that of the sk-tree, to which the children of those nodes have been added, which were leaves in the sk-tree but not in the original Huffman tree, that is, the leaves v for which $h(v) \geq 1$. Bitmaps will be stored for the internal nodes of the Wavelet tree, as well as for the leaves that are children of leaves v of the sk-tree for which $h(v) > 1$, albeit the nature of these latter bitmaps will be different. The internal nodes will store the bitmaps as in the original Wavelet tree, whereas the annotated leaves will store the binary strings obtained by the concatenation of the suffixes of length $h-1$ of the corresponding codewords, in the same order as they appear in the compressed text. That is, each such suffix appears the same number of times as the number of occurrences of the corresponding alphabet symbol $\sigma \in \Sigma$ in T.

Continuing with the running example, the resulting pruned Wavelet tree is given in Figure 3. Consider the node labeled 3; it refers to the prefix 11 of several codewords, and the bitmap stored in it relates to the third bit of these codewords, which are all

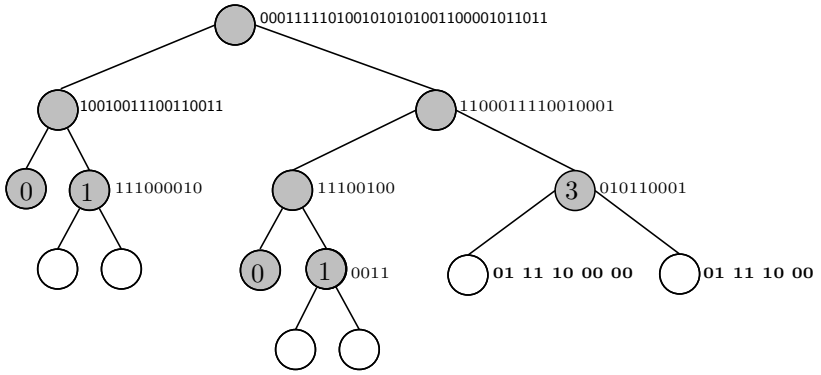


Figure 3. Pruned Huffman Wavelet tree for the text $T =$ `A--HUFFMAN--WAVELET--TREE--MATTERS`

of length 5. We thus eliminate the 3 bits that were already taken care of (110 for the left child and 111 for the right one), and consider only the remaining suffixes of size 2. In our example, the left child corresponds to the codewords {11000, 11001, 11010, 11011}, prefixed by 110 and refer to the symbols {`R,H,L,N`} of Figure 1, respectively. Their suffixes occur in the bitmap in the same order they appear in T, namely 01 11 10 00 00, corresponding to the order `HNLRR`. A similar idea to this collapsing strategy is applied on suffix or position trees in order to attain an efficient *compacted* suffix trie [6], and has also been applied on Fibonacci Wavelet trees, producing a compact Wavelet tree in [14].

The algorithm for extracting the i-th element of T from a pruned Huffman Wavelet tree requires some adjustments for concatenating the pruned parts. Figure 4 is the suitable extract function. Line 2.2.1 concatenates the fixed length suffix of size $h(v)-1$ bits to the end of the codeword. The correct suffix can be accessed directly using the computed index i by simply extracting the substring of B_v starting at position $(h(v)-1)i$ and ending at position $(h(v)-1)(i+1)-1$. We use the notation $B[x\,..\,y]$ to denote the substring from position x to, and including, position y of a bit-string B.

```
extract(v, i)
1     cw ⟵ ϵ
2     while v is not a leaf
2.1       if h(v) = 0 then
2.1.1         if B_v[i] = 0 then
2.1.1.1           v ⟵ left(v)
2.1.1.2           cw ⟵ cw · 0
2.1.1.3           i ⟵ rank_0(B_v, i)
2.1.2         else
2.1.2.1           v ⟵ right(v)
2.1.2.2           cw ⟵ cw · 1
2.1.2.3           i ⟵ rank_1(B_v, i)
2.2       else // h(v) ≠ 0
2.2.1         cw ⟵ cw · B_v[(h(v) − 1)i .. (h(v) − 1)(i + 1) − 1]
3     return D(cw)
```

Figure 4. Extracting the i-th element of T from the pruned Huffman Wavelet tree.

The following discussion refers to the select operation, however, a similar approach could be applied in order to answer the rank operation. Computing $\mathsf{select}(x, i)$ for selecting the i^{th} occurrence of x is done in the traditional Wavelet tree by processing

the tree upwards. One starts from the leaf, ℓ, representing the Huffman codeword $c(x)$ of x, initializes v to be the father of ℓ, and works its way up to the root. In each iteration, i is assigned a new value $\mathsf{select}_0(B_v, i)$ or $\mathsf{select}_1(B_v, i)$, depending on ℓ being a left or right child of v, respectively. The node v then proceeds to its father for the following stage. The running time for $\mathsf{select}(x, i)$ is $O(|c(x)|)$.

Taking a closer look at our suggested data structure, the nodes that store the values $h(v)$ induce a partition of the alphabet into several equivalence classes. Some of these classes are singletons, while the others are of size 2^k for some k. The skeleton Huffman tree does not have the ability to distinguish between elements of the same class. Thus, when applying $\mathsf{select}(x, i)$ on our pruned data structure, only partial information is attained. Instead of returning the i^{th} occurrence of x, x becomes a representative of its class, and the i^{th} occurrence of elements which are in the same class as x is returned.

However, the classes are formed according to the probabilities of their elements, which does not necessarily imply any other connection. Nevertheless, whereas the exact values cannot be calculated using the original $\mathsf{select}(x, i)$ algorithm, this algorithm can still be used to derive a *lower bound* on the index of the i^{th} occurrence of x. If $\mathsf{select}(x, i) = j$, then the index of the i^{th} occurrence of x is $\geq j$. It is equal to j if all occurrences of elements belonging to the class of x correspond only to occurrences of x itself. If $\mathsf{extract}(v_{root}, j) \neq x$, a larger lower bound can be computed by applying select again with increasing i, until $\mathsf{extract}(v_{root}, j) = x$.

Although the select query cannot be answered in constant time using the pruned Wavelet tree, the exact value can still be derived iteratively. For example, finding the index of the *first* occurrence of x can be done in the following way: if $\mathsf{select}(x, 1) = j$ and $\mathsf{extract}(v_{root}, j) = x$, the first occurrence of x is found at index j. If $\mathsf{extract}(v_{root}, j) \neq x$, but $\mathsf{select}(x, 2) = k$ and $\mathsf{extract}(v_{root}, k) = x$, the first occurrence of x is found at index k. Otherwise the process continues until there exists some ℓ for which $\mathsf{select}(x, \ell) = m$ and $\mathsf{extract}(v_{root}, m) = x$. For larger i, the $\mathsf{select}(x, i)$ query can be computed as follows:

```
1     counter ⟵ 0; ℓ ⟵ 1; m ⟵ 0;
2     while counter < i and m ≤ n
3           m ⟵select(x, ℓ);
3.1         if extract(v_root, m) = x
3.1.1             counter++
3.2         ℓ++
```

It should be noted that the negative impact of using the pruned Wavelet tree on the select queries is not as bad as it might seem on the first sight. The equivalence classes of the codewords that have been pruned may be quite large, as can be seen, for example, in Figure 5 below, but the large classes correspond to the smaller probabilities. There is, of course, no knowledge about which elements will have to be retrieved, and we might be asked to perform a $\mathsf{select}(x, \ell)$ query for any x. Nonetheless, a reasonable assumption would be to assume that the appearance of codewords x in such queries will be according to their probability of occurrence in the text. In that case, the weighted average size of the equivalence classes will be quite small, so that an iterative search as suggested above is not such a burden. An indication for this asymmetric behavior of skeleton trees can be found by comparing the savings they imply on the space and time complexities: while the number of nodes can be

reduced by 95% or more on large distributions, the weighted average path length for the same distributions is only shortened to about half, see the examples in [13].

The extract operation is much easier to apply on fixed length codes than on variable length codes. In our pruned data structure, nodes v with $h(v) > 0$ store fixed length suffixes, hence, the improvement of the extract operation on our data structure over Wavelet trees for Huffman codes is clear. However, this is not the case when processing fixed length codes in order to locate and count the occurrences of a given codeword. Counting occurrences or locating the i^{th} occurrence of a given codeword in the pruned data structure requires to perform a rank or select operation on the fixed length suffixes stored in the leaves of the pruned Wavelet tree. It seems, that if no auxiliary structure is used, then the rank and select queries must be performed sequentially, and the advantage of using fixed length suffixes disappears.

One could ask, therefore, whether rank and select queries can be done in a more efficient way for fixed length than for variable length codes. If this is the case, we can apply such a strategy on the fixed length suffixes of our data structure and support efficient rank and select queries as well, gaining faster processing time since the lengths of many of the codewords are shortened.

Note that the bits in the bitmaps stored in the leaves of the pruned Wavelet tree are the same as for the original Wavelet tree, only their order may have changed. In our example, the 18 bits appearing in boldface in Figure 3 in the subtree rooted by the node labeled 3 are the same bits as those appearing in the bitmaps of the nodes in the corresponding subtree of Figure 1, that has been pruned. The savings of the pruned Huffman Wavelet tree as compared the original one of Section 2 stem thus from the fact that the rank and select data structures corresponding to the nodes are not all necessary for gaining the ability of direct access, because the bits corresponding to codeword suffixes are stored explicitly, and need not be extracted from bitmaps. The processing time is improved by accessing a smaller number of nodes. To evaluate the savings induced by the pruning (restricting the analysis only to the rank function), we introduce the following notations. For an internal node v of the canonical Huffman tree, define $\mathsf{pref}(v)$ as the prefix of all the codewords corresponding to this node. So, $\mathsf{pref}(root) = \Lambda$, denoting the empty string, and in Figure 1, if t is the node on level 3 annotated by the bitmap 0011, then $\mathsf{pref}(t) = 110$. Let C be the set of all the codewords. For a codeword $c \in C$ denote by $x(c)$ the corresponding character of the alphabet, and let $\mathsf{freq}(x)$ be the number of occurrences of x in the text. The length of the bitmap B_v stored at node v of the Wavelet tree is then given by

$$|B_v| = \sum_{\{c \in C \mid \mathsf{pref}(v) \text{ is a prefix of } c\}} \mathsf{freq}\big(x(c)\big).$$

In particular, if v is the root, we get that $|B_v|$ is the sum of the frequencies of all the elements of the alphabet, which is equal to the length of the text in characters.

Summing the lengths of all the bitmaps in the Wavelet tree gives the size, in bits, of the compressed file:

$$\text{Size of compressed file} = \text{lengths of all bitmaps} = \sum_{\{v \mid v \text{ is an internal node}\}} |B_v|.$$

Let $\mathcal{R}(n)$ denote the size of the data structures required by the rank function for a bitmap of size n. This could be $O(\frac{n \log\log n}{\log n})$ to allow constant time, and although this size is $o(n)$, we mentioned above that it is still not negligible, even for very large n.

As alternative, $\mathcal{R}(n)$ can be reduced to $\frac{n}{20}$, at the price of increased processing time. The overall size, RSW, required by the rank structure of the original Wavelet tree is thus

$$\text{RSW} = \sum_{\{v \mid v \text{ is an internal node}\}} \mathcal{R}(|B_v|).$$

When using the pruned version, the rank structures for the bitmaps corresponding to pruned subtrees are not needed. Denote by T_w the subtree rooted at the node w and by SKL the set of leaves of the sk-tree. The number of bits saved for the rank structures by the pruning process, RSW', is given by

$$\text{RSW'} = \sum_{\{w \mid w \in \text{SKL} \wedge h(w)>1\}} \sum_{\{v \mid v \in T_w \wedge v \neq w\}} \mathcal{R}(|B_v|).$$

For example, for the tree in Figure 4, the outer summation refers to all the leaves of the sk-tree, which are the gray nodes labeled by the numbers $h(v)$, but only for one node, the condition $h(v) > 1$ is satisfied. The inner summation goes over all the internal nodes, except the root of the subtree.

It follows that the savings depend on the shape of the canonical tree and the corresponding sk-tree. In the worst cases, the skeleton tree yields no savings at all, but this happens only for highly skewed distributions implying a depth of $\Omega(|\Sigma|)$ for the Huffman tree, which is extremely rare for large alphabets. In general, the number of pruned nodes is substantial, and the overhead for the rank structures, RSW−RSW', will be significantly smaller for the pruned version of the Wavelet tree.

4 Reduced skeleton trees

Extending the pruning idea, we wish to prune the Huffman tree even more, possibly suggesting a tradeoff between space efficiency and processing time. However, it is not clear that processing time would be hurt by this further reduction, since less internal nodes would be processed. The idea is replacing the Skeleton tree topology of the Wavelet tree by a *Reduced Skeleton tree* suggested in [13]. The Reduced Skeleton tree prunes the Skeleton Huffman tree at some internal node at which the length of the current codeword is only partially determined. That is, when getting to a leaf of a Reduced Skeleton Tree, it is not yet possible to deduce the length of the current codeword, but some partial information is already available: the possible lengths belong to a set of size at most 2.

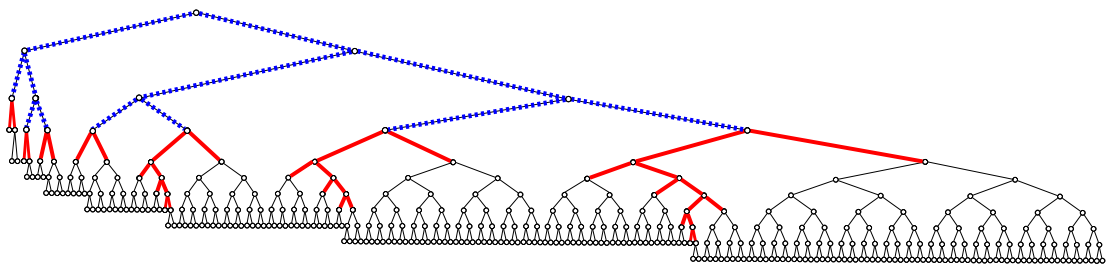


Figure 5. Canonical Huffman tree, sk-tree (bold, red and blue) and reduced sk-tree (broken lines, blue) for 200 elements of a Zipf distribution, defined by the weights $p_i = 1/(i\,H_n)$, for $1 \le i \le n$, where $H_n = \sum_{j=1}^{n}(1/j)$ is the n-th harmonic number.

Consider, for example, the canonical Huffman tree given in Figure 5. It corresponds to the probability distribution of $n = 200$ elements implied by Zipf's law [21], which is believed to govern the distribution of the most common words in a large natural language text. The bold (red or blue) edges are the corresponding sk-tree, and the subset of the bold edges, those with broken lines (blue), are the reduced sk-tree. For instance, when one gets to the leaf of the reduced sk-tree corresponding to 110, one already knows that the codeword will be of length 8 or 9, so a single comparison suffices to decide it.

The algorithm for extracting the i-th element of T when the Wavelet tree is constructed according to the reduced skeleton tree is similar to the algorithm presented earlier in Figure 4, and is given in Figure 6. We now need a *flag field* for each leaf v, with $flag(v) = 0$ if v is also a leaf in the skeleton Huffman tree (i.e., the length of the codeword is known when getting to this leaf while traversing the tree with an encoded string starting at the root; note that no leaf of the reduced sk-tree in Figure 5 has this property, but for other distributions, such leaves do exist), and $flag(v) = 1$ otherwise. In the latter case, the suffixes rooted at v are not of the same length, and we adjust the shorter suffixes to be of the length of the longer ones by padding them at their right end with a single 0. We then concatenate all these equal sized reconstructed suffixes in the same order as they appear in the text, as in skeleton Wavelet trees. The value $h(v)$ now stores the length of the suffix of the longer codeword if v is a leaf, and 0 if v is an internal node.

When a leaf v is reached, the current suffix is initialized as having length $h(v)$. This is the correct setting when $flag(v) = 0$. When $flag(v) = 1$, we compare the integer value j obtained by using the retrieved suffix with that of the first codeword of length $|cw|$. If j is smaller or equal, we know that the length of the codeword is $|cw| - 1$, hence we remove the trailing 0 from the current codeword.

```
...
4     else // h(v) ≠ 0
4.1         cw ⟵ cw · B_v[(h(v) − 1)i . . (h(v) − 1)(i + 1) − 1]
4.2         if flag(v) = 1 then
4.2.1           if cw ≤ first codeword of length |cw| then
4.2.1.1             remove trailing 0 from cw
5     return D(cw)
```

Figure 6. Extracting the i-th element of T from a Wavelet tree based on a reduced skeleton tree.

5 Conclusion

We have presented a new data structure for reducing the space overhead of a Huffman shaped Wavelet tree when used to support extract queries to the underlying text by means of a Skeleton Huffman tree. The running time is expected to be improved as compared to the running time of the traditional Wavelet tree, since shorter paths outgoing the root down to the leaves are processed. We intend to implement the pruned data structure and include experimental results in the full version of this paper.

References

1. J. Barbay, F. Claude, T. Gagie, G. Navarro, Y. Nekrich, Efficient Fully-Compressed Sequence Representations, *Algorithmica* **69**(1) (2014) 232–268.
2. J. Barbay, T. Gagie, G. Navarro, Y. Nekrich, Alphabet partitioning for compressed rank/select and applications, *Algorithms and Computation,* Lecture Notes in Computer Science LNCS, **6507** (2010) 315–326.
3. N.R. Brisaboa, A. Fariña, S. Ladra, G. Navarro, Reorganizing compressed text, *Proc. of the 31th Annual Internetional ACM SIGIR Conference on Research and Developing in Information Retrieval (SIGIR)* (2008) 139–146.
4. N.R. Brisaboa, S. Ladra, G. Navarro, DACs: Bringing direct access to variable length codes, *Information Processing and Management*, **49**(1) (2013) 392–404.
5. D. Clark, Compact Pat Trees, Ph.D. Thesis, University of Waterloo, Canada, (1996).
6. M. Crochemore, W. Rytter, *Jewels of Stringology*, World Scientific (2002).
7. T. Gagie, G. Navarro, S.J. Puglisi, New algorithms on Wavelet trees and applications to Information Retrieval, *Theoretical Computer Science* **426** (2012) 25–41.
8. T. Gagie, G. Navarro, Y. Nekrich, Fast and Compact Prefix Codes. *Proc. SOFSEM'10*, (2010) 419–427.
9. E.N. Gilbert, E.F. Moore, Variable-length binary encodings, *The Bell System Technical Journal*, **38** (1959) 933–968.
10. R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, *Poster Proceedings of 4th Workshop on Efficient and Experimental Algorithms (WEA05)*, Greece (2005) 27–38.
11. R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA)* (2003) 841–850.
12. G. Jacobson, Space efficient static trees and graphs, *Proc. Foundations of Computer Science (FOCS)* (1989), 549–554.
13. S.T. Klein, Skeleton trees for the efficient decoding of Huffman encoded texts, in the *Special issue on Compression and Efficiency in Information Retrieval* of the *Kluwer Journal of Information Retrieval* **3** (2000) 7–23.
14. S.T. Klein, D. Shapira, Random access to Fibonacci Codes, *The Prague Stringology Conference PSC-2014* (2014) 96–109.
15. M.O. Külekci, Enhanced Variable-Length Codes: Improved Compression with efficient random access, *Proc. Data Compression Conference DCC–2014*, Snowbird, Utah (2014) 362–371.
16. G. Navarro, E. Providel, Fast, small, simple rank/select on bitmaps, *Experimental Algorithms,* Lecture Notes in Computer Science (LNCS), **7276** (2012) 295–306.
17. D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, *Proc. ALENEX, SIAM* (2007).
18. R. Raman, V. Raman, S. Rao Satti, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, *Transactions on Algorithms (TALG)* (2007) 233–242.
19. D. Shapira, A. Daptardar, Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts, *Information Processing and Management, IP & M* **42**(2) (2006) 429–439.
20. H.E. Williams, J. Zobel, Compressing integers for fast file access. *The Computer Journal* **42**(30) (1999) 192–201.
21. G.K. Zipf, *The Psycho-Biology of Language*, Boston, Houghton (1935).

Controlling the Chunk-Size in Deduplication Systems

Michael Hirsch[1,5], Shmuel T. Klein[2], Dana Shapira[3], and Yair Toaff[4,5]

[1] Mazkeret Batya, Israel,
[2] Computer Science Department, Bar Ilan University, Israel
[3] Computer Science Department, Ariel University, Israel
[4] Givat Shmuel, Israel
[5] the work was done while these authors were affiliated with Diligent, an IBM Company
{mikizvi,yair.toaff}@gmail.com, tomi@cs.biu.ac.il, shapird@ariel.ac.il

Abstract. A special case of data compression in which repeated chunks of data are stored only once is known as deduplication. The input data is cut into chunks and a cryptographically strong hash value of each (different) chunk is stored. To restrict the influence of small inserts and deletes to local perturbations, the chunk boundaries are usually defined in a data dependent way, which implies that the chunks are of variable length. Usually, the chunk sizes may spread over a large range, which may have a negative impact on the storage performance. This may be dealt with by imposing artificial lower and upper bounds. This paper suggests an alternative by which the chunk size distribution is controlled in a natural way. Some analytical and experimental results are given.

1 Introduction

Research in Lossless Data Compression has concentrated for years on improving the compression ratio, the speed of the encoding and decoding procedures and the necessary auxiliary storage. Yet some files, like purely random data, cannot be compressed at all. There are, however, applications in which even such incompressible files, if they appear more than once, may yield some savings. An example could be a large backup system, in which the entire available electronic storage of some corporation has to be copied and saved at regular time intervals for security reasons and to prevent the loss of data. The special feature of such backup data is that only a small fraction of it differs from the previously stored backup. This calls for a special form of data compression, known as *deduplication*: trying to store duplicates only once. The challenge is, of course, to locate as much of the duplicated data as possible.

A standard deduplication system achieves its goal in the following way. Partition the input database, which is often called the *repository*, into fixed or variable sized blocks, called *chunks*, apply a cryptographically strong hash function on each of these input chunks, and store the different hash values, along with the address of the corresponding chunk, in a fast-to-access data structure, like a hash table or a B-Tree [10,12]. When a fresh copy of the data is given, e.g., for a weekly or even daily backup, the new data, often called a *version*, is also partitioned into similar chunks. The hash value of each of these new chunks is searched for in the table, and if it is found, one may conclude that the new chunk is an exact copy of a previous one, so all one needs to store is a pointer to the earlier occurrence. There are also approaches to deduplication which relax the request for identical chunks and replace one chunk by another even if they are only *similar*, adding of course also the (few) differences to enable the recovery of the original data [1,2,11,7].

Michael Hirsch, Shmuel T. Klein, Dana Shapira, Yair Toaff: Controlling the Chunk-Size in Deduplication Systems, pp. 78–89.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

A simple approach would be to choose the chunk size as a constant. This would, however, result in a high sensitivity to small insertions and deletions. Indeed, even a single added or omitted byte could shift all subsequent chunk boundaries accordingly, invalidating the hash approach. The solution is to let the boundary of the chunk to be dependent on the content itself, which implies variable length chunks.

A general paradigm for cutting the data string consisting of a sequence of bytes $s_1 s_2 \cdots$ into pieces was to use a rolling hash, which calculates a hash value for any consecutive sequence of k bytes. Such a sequence will be called a *seed*. Each byte, starting with the byte indexed k and onwards, can be considered as the last of a seed. The condition for deciding whether the last byte of the seed, s_j, will also be the last byte of the current chunk, is that

$$h(s_{j-k+1} s_{j-k+2} \cdots s_j) = C,$$

where h is the hash function and C is some constant chosen from the set of values $\{h(i)\}$. Since hash functions are supposed to return uniformly distributed values, the probability of this occurring is $1/M$, where M is the size of the set of possible hash values, and it is independent of the specific value C chosen. The expected size of the chunks is then M. However, in practice, the sizes of the chunks may greatly vary, which is why it is necessary to impose lower and upper limits. For example, if we aim at an average size of 4K, we might not even check at the beginning, thereby assuring that the chunk size will not be below, say, 1K. Similarly, if the condition has not been fulfilled by any seed and we reach already a chunk size of, say, 8K, we might just cut the chunk at this point, regardless of the hash value.

While this strategy will indeed force the chunk size to be between 1K and 8K in our example, these extreme values are "artificial" cutoff points. They impose breaks in the flow of data that are not robust and not reproducible in the case of relatively small inserts or deletes. In general, the distribution of segment sizes is geometric. Cutting off an arbitrary section at the start actually eliminates a very large number of potential segment boundaries. Chopping the tail at an arbitrary size cuts a tail of infinite length, affecting the mean segment size more than would be expected.

Furthermore, segmentation techniques based on these rules produce a very inconvenient distribution of segment sizes because of their geometric distribution. There are a very large number of very small segments and a significant number of very large segments. This stresses the storage subsystem of a program that must store and index these segments.

The problem of segmentation has been the subject of much literature, one of the first being [9]. A brief survey can be found in [4]. Some of the approaches, e.g. [3] are more rigorous. A good description of segmentation appears in the text of [8].

Here we suggest a method that tries to rectify the shortcomings of minimal and maximal segment size, while also providing segment sizes that are bunched around the mean size. The basic idea is a new way of text segmentation, in which the probability of declaring a segment boundary changes with the number of bytes read since the previously declared segment boundary. This enables us to control the segment size distribution with much greater accuracy than what is possible with existing segmentation techniques.

Initially, it is highly unlikely (but still possible) that a boundary will be declared. This means that there are very few small segments, and hence no need to impose an artificial minimum segment size. As more bytes pass since the end of the previous segment, the criterion for declaring a segment is relaxed. By relaxing the criterion even-

tually completely, we encourage the distribution to tail off as sharply or as loosely as we need. This means that no artificial maximal segment size is needed. This property is especially important, because data may contain very long sequences (e.g., stretches of blanks or zeros) that may not trigger declaring a segment boundary. These can safely be chopped at an artificial maximal size without affecting deduplication.

This relaxation of the segmentation criteria is strictly defined as a family of functions such that each later member "includes" all the previous ones. This provides robustness to inserts and deletes. By tuning this relaxation, we are able to produce approximately any segment size distribution we prefer. We may choose one tailored to the needs of the storage subsystem that must store the unique segments.

In the next section, we present the details of the proposed method, and extend the ideas in Section 3 to the usage of fractional bits. Finally, Section 4 brings some experimental results.

2 New segmentation procedure

Instead of working with a single hash function h and a single constant C, we shall use a sequence of functions and constants h_i and C_i, $i = 1, 2, \ldots, n$, fulfilling the following conditions:

1. All functions are easy to calculate;
2. there exists an increasing sequence of probabilities $p_1, p_2, \ldots, p_n$ such that for any seed S of fixed length k, $\Pr(h_i(S) = C_i) = p_i$, where $\Pr()$ denotes the probability function;
3. the conditions are inclusive in the sense that

$$\forall S \quad \forall j > i \qquad h_i(S) = C_i \quad \longrightarrow \quad h_j(S) = C_j.$$

The sequence of functions h_i is then used to partition the potential chunk that is being built into three regions, delimited by the four values A_L, P_L, P_U, A_U, corresponding to the absolute lower, preferred lower, preferred upper and absolute upper limits for the occurrence of the (right) chunk boundary, as depicted in Figure 1 below. The target value of the expected size, E, is indicated by the black bar. Preferably, we want this value to fall between P_L and P_U, however, we might tolerate exceeding these limits, but not below A_L and not above A_U. This is achieved by choosing one of the indices j_0, $1 < j_0 < n$, and setting $p_{j_0} = 1/E$. Recall that our procedure for cutting the chunk being built at the current position is checking whether $h_j(S) = C_j$, where S is the seed extending up to the current position, and repeating this test for every byte, i.e., considering overlapping seeds. We shall use the same function h_{j_0} while the chunk size is in the preferred (grey) zone, between P_L and P_U. However, the range between A_L and P_L will be partitioned into sub-intervals in which the hash functions used are, in order, $h_1, h_2, \ldots, h_{j_0-1}$, and similarly, the range between P_U and A_U will be partitioned into sub-intervals in which the hash functions used are, in order, $h_{j_0+1}, \ldots, h_n$. An absolute upper size of the chunk can be imposed by defining $p_n = 1$, that is, the first seed considered when getting to the last function will be declared as being the last seed of the current chunk. Since the test for h_n has then probability 1 to succeed, A_U is indeed an upper limit.

The main advantage of the proposed method is then that the chunk size needs no artificial lower or upper limits, because these limits are obtained in a natural

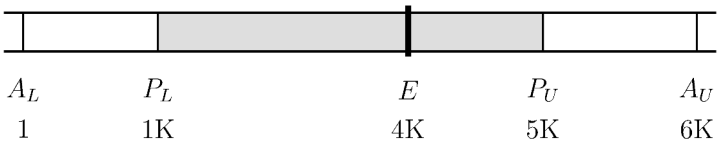


Figure 1. Possible regions for chunk boundaries.

and consistent way, so that the chunking mechanism can be applied without all the drawbacks mentioned above.

The method works because of the chosen conditions on the sequence of hash functions. The first condition is a basic requirement of any hash function. The second condition lets us define the cut-off condition differently depending on the number of the already accumulated bytes in the current chunk: we shall start with a very low probability of setting the boundary of the chunk, so that very small chunks will almost surely not appear. The closer we get to the target size, say 4K, the larger the probability will get, and within a range to be chosen around the ideal chunk size, say, between 1K and 5K, the probability will be constant. Once we have passed this upper limit, the cutoff probability will start rising, so that it will get increasingly difficult to extend the chunk further. Since $p_n = 1$, the last function will be used only once, and the chunk size will not exceed A_U.

The third condition deals with inserts and deletes. This is best explained by considering Figure 2 below. The top line represents two consecutive chunks of the original data. Suppose now that a short sequence of new bytes is inserted, as in the middle line of the figure. There is of course the possibility that one of the newly added seeds will fulfill the cut-off condition, but if the inserted block is small, this possibility might be negligible. If no new boundary has been declared, the seed S which ended at position A in the original layout has been pushed further to position B, which implies that the test applied on it is $h_j(S) = C_j$ for some $j > i$, therefore S will be declared as boundary and subsequent chunks will not be affected.

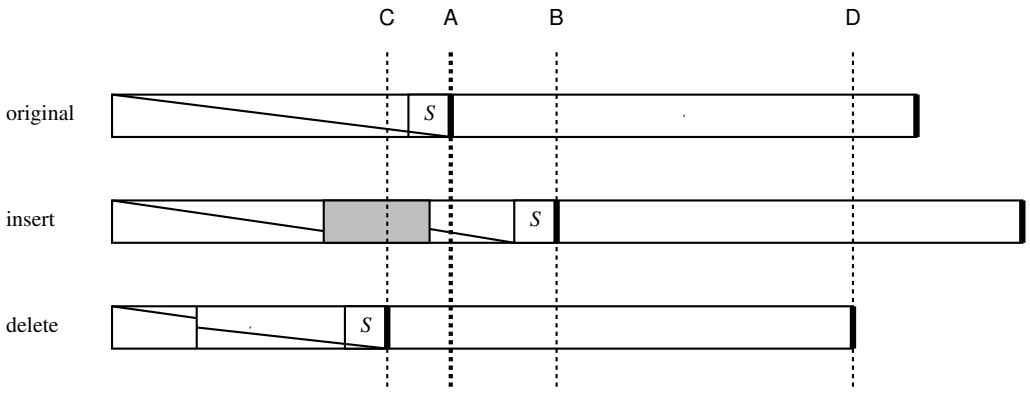


Figure 2. Schematic representation of the effect of insert and delete.

If some bytes have been deleted from the first chunk, as displayed in the lowest line of the figure, the seed S is moved to an earlier position C, so the condition checked on it, $h_t(S) = C_t$ for some $t \leq i$, might be stricter than before. It is thus possible that the boundary at level C will be missed. But depending on the number of deleted bytes, the condition might also be the same (if $t = i$), or, if $i - t$ is small, the probability of getting even this cut-off point might not be too low. In any case,

even if this chunk limit is lost, chances are good that the next one, which has now been moved backwards to position D, is still to the right of A, so it will be caught.

A possible implementation could set the limiting values as shown in the bottom line of Figure 1. To define the sequence of functions h_i, we first choose a random large prime number P. In practice, since arithmetic operations will be performed modulo P and given that typical CPUs at present mostly have 64-bit capabilities, it will be convenient to restrict ourselves to 64-bit operations, implying $P < 2^{64}$. If we were to choose a new random prime in every calculation, as is done for the Karp-Rabin pattern matching [6], there would be no need to impose also a lower limit on P, since the probability of repeatedly choosing small primes in negligible. But in our case, since the intention is to use a single prime for the entire system, we should prevent a bad choice by imposing also, say, that $P > 2^{60}$. This assures that P has at least 60 significant bits, without being too restrictive, since the number of primes in the given range is of the order of 2^{58}. Let $r_1, r_2, \ldots, r_n$ be a decreasing sequence of integers, subject to the constraints

$$32 = r_1 > r_2 > \cdots > r_{j_0-1} > r_{j_0} = \log_2 E > r_{j_0+1} > \cdots > r_{n-1} > r_n = 0,$$

the functions h_i, for $i = 1, 2, \ldots, n$, will then be defined as

$$h_i(S) = (S \bmod P) \bmod 2^{r_i},$$

in other words, $h_i(S)$ are the r_i rightmost bits of the remainder of S modulo P.

The next step is to choose a random 32-bit constant C, and to define

$$C_i = C \bmod 2^{r_i},$$

that is, the C_i are the r_i rightmost bits of C. Theoretically, we could have chosen the C_i at random, if indeed the hash functions gave uniformly distributed values. Practically, it will be convenient to have all the C_i as suffixes of different lengths of the same binary string, which enables us to fulfill condition 3.

In our particular implementation, we chose the following parameters:

$$n = 18, \quad j_0 = 11, \quad r_{11} = 12,$$
$$(r_1, \ldots, r_{10}) = (32, 30, 28, 26, 24, 22, 20, 18, 16, 14), \quad (r_{12}, \ldots, r_{18}) = (11, 9, 7, 5, 3, 1, 0).$$

Figure 3 is a plot of the number of bits involved in the hashing (which is minus the $\log_2$ of the probability of declaring the current position as a boundary point) as function of the current size of the chunk being built. We see that we start with a very low probability, 2^{-32}, which gradually gets larger (i.e., the number of bits decreases). The sizes of the corresponding ranges start with 2 bytes for 32 bits and 2 bytes for 30 bits, and then double at each step (4 bytes for 28 bits, 8 bytes for 26 bits,..., 512 bytes for 14 bits). This corresponds to the range from A_L to P_L and spans exactly 1K. Then from 1K to 5K we stay with 12 bits, that is, probability 2^{-12}, and then continue increasing the probabilities, this time on ranges that start with 512 bytes for 11 bits, then halving to 256 bytes for 9 bits, up to 64 bytes for 5 bits, 32 bytes for 3 bits and 31 bytes for 1 bit. There is also a possibility for 0 bits, but a range of only 1 byte is assigned, since it guarantees success at the first try. Denote by w_i the number of times the procedure is applied with r_i if it still continues, that is, no boundary for the current chunk has yet been set. We then have for this example setting:

$$(w_1, \ldots, w_n) = (2, 2, 4, 8, \ldots, 512, \mathbf{4096}, 512, 256, 128, 64, 32, 31, 1),$$

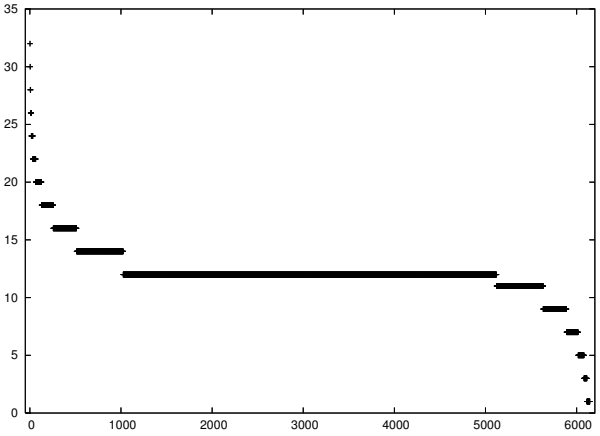


Figure 3. Plotting the number of bits used to declare a boundary as function of chunk size.

where w_{11} corresponding to r_{j_0} has been boldfaced.

Denote the length of a given chunk by L, which is a random variable whose expected value we are interested in. To evaluate the expected size of the chunk for the given settings, we shall use the formula $E(L) = \sum_{M=1}^{A_U} \Pr(L \geq M)$. The probability of getting a chunk size L which is $\geq M$ is the probability of getting failures on the first M trials, and can be evaluated as follows. Let ℓ be the index of the range to which the current size M belongs, that is, given M, we find ℓ which satisfies

$$\sum_{t=1}^{\ell-1} w_t < M \leq \sum_{t=1}^{\ell} w_t.$$

We can then calculate the probability as:

$$\Pr(L \geq M) = \Big[\prod_{t=1}^{\ell-1} \left(1 - 2^{-r_t}\right)^{w_t}\Big] \quad \left(1 - 2^{-r_\ell}\right)^{M - \sum_{t=1}^{\ell-1} w_t},$$

from which we can derive

$$\Pr(L = M) = \Pr(L \geq M) - \Pr(L \geq M - 1).$$

For our example distribution, we get as expected value for the chunk size: $E(L) = 3744$.

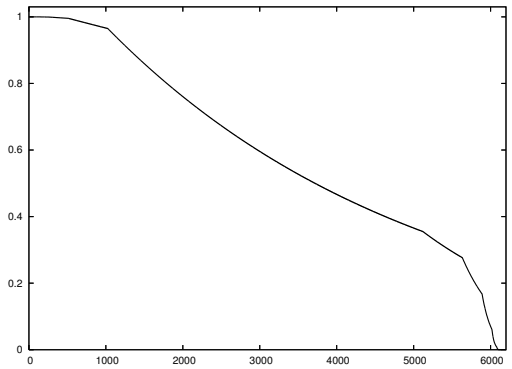


Figure 4. Cumulative probabilities $\Pr(L \geq M)$.

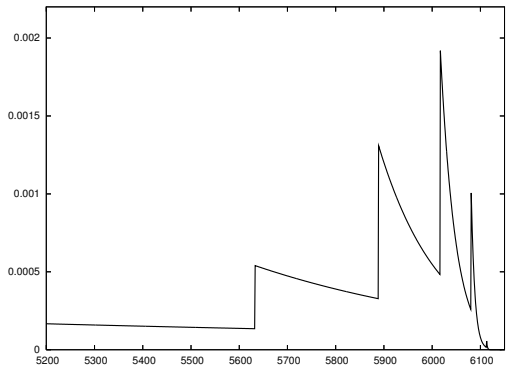


Figure 5. Individual probabilities $\Pr(L = M)$.

The hedgehog shaped graph in Figure 5 gives these probabilities for our example distribution, displayed in Figure 4. The spikes in this plot are due to the discrete nature of the distribution: using an integral number of bits for every test, the resulting probability function will not be continuous at integer points. If we prefer getting a continuous bell shaped Gaussian curve, we need to perform the tests with r_i bits without restricting the r_i to be integers. This calls for trying to deal with fractional bits or at least to simulate the behavior of the probability function as if fractional bits could be compared. This is done in the following section, dealing with the theoretical framework of using fractional bits; this has not been implemented in the experiments reported in Section 4.

3 Cutting chunks using fractional bits

The hash functions used are of the form $S \bmod P$, where S is a sequence of k consecutive bytes considered as the binary representation of one large integer of length $8k$ bits, and P is some large prime number that has been chosen arbitrarily, but is fixed throughout the process. To simulate the fractional bits, let us first decide how fine grained the resolution has to be. This is done by deciding on a step size ε, where the discrete steps correspond to $\varepsilon = 1$, and we could impose, e.g., $\varepsilon = 10^{-3}$. We thus need $\lceil -\log_2 \varepsilon \rceil$ additional bits in our hash values. Suppose we want to simulate the hashing as if it were working on ℓ bits, where ℓ is not an integer. Define the fractional part of ℓ as $f = \ell - \lfloor \ell \rfloor$, then $0 < f < 1$. We shall use either $\lfloor \ell \rfloor$ or $\lceil \ell \rceil$ bits, by first comparing just the $\lfloor \ell \rfloor$ first bits, and checking also the $\lfloor \ell \rfloor + 1$st bit with probability f'. This probability f' will be chosen as follows. Since we are simulating a sequence of Bernoulli trials, we want the probability of failure to be

$$2^{-\ell} = 2^{-(\lfloor \ell \rfloor + f)} = 2^{-\lfloor \ell \rfloor} \cdot 2^{-f}.$$

On the other hand, comparing only $\lfloor \ell \rfloor$ bits, and the additional bit with probability f', we get as probability for failure

$$(1 - f')2^{-\lfloor \ell \rfloor} + f'2^{-\lfloor \ell \rfloor - 1}.$$

Equating the two, we can derive f' as function of f:

$$f' = 2 - 2^{-f+1}.$$

Figure 6 plots the value of f' as function of f and shows that f' is only slightly larger. For example, to simulate a comparison on $\lfloor \ell \rfloor + \frac{1}{2}$ bits, we should compare the additional bit with probability $f' = 2 - \sqrt{2} = 0.586$.

A first thought about how to implement the comparison of the $\lfloor \ell \rfloor + 1$st bit with probability f' could be to generate a random number r between 0 and 1, and then perform the additional comparison if $r \leq f'$. Such a strategy would, however, hurt the consistency of the chunking procedure: if the same chunk of a certain length $\lfloor \ell \rfloor + 1$ reoccurs, this would not guarantee the same decision at the last comparison for both occurrences, so the system could fail in detecting a chunk that might be deduplicated. To rectify this, instead of r, one should rather use a pseudo-random number r' depending solely on the currently processed chunk. For example, consider an arbitrary, yet constant, subsequence of the bits currently forming the processed chunk S, denote the number represented by this subsequence as S', choose a random

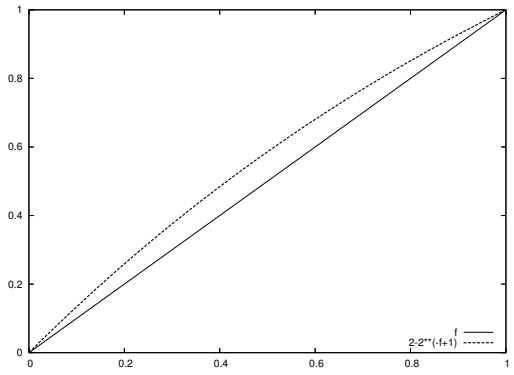


Figure 6. Plotting $f' = 2 - 2^{-f+1}$.

large prime Q, which is different from the prime P chosen earlier, and then set the threshold probability to be

$$r' = \frac{(S' + |S|) \bmod Q}{Q},$$

where the current length of the chunk has been added to avoid a bias in the case of long stretches of zeros.

Once the question of how to process fractional bits has been handled, the next step is to define the number of bits used in the sequence of hash functions as a continuous decreasing function. The first option would be to decrease the number of bits linearly from 32 to 0, in 4K steps. This, however, gives a quite narrow distribution of the chunk sizes, which all fall between roughly 2K and 3600, with average 3026. Starting with less than 32 bits, but leaving the 4K steps, reduces the average and broadens the bell shaped distribution. If we aim at getting an average chunk size of 2K, we should start at 18.3 bits. Decreasing this number in 4K regular steps to 0 yields then the solid line plots in the graphs of Figures 7(a)–(c). Figure 7(a) shows the decrease in the number of bits used in the hashing function, as a continuous function of the number of bytes in the current chunk. Figures 7(b) and 7(c) are the corresponding cumulative and individual probabilities for the possible chunk sizes, i.e., $\Pr(L \geq M)$ and $\Pr(L = M)$ for a size M of a chunk, $1 \leq M \leq 5000$.

As alternative, the decrease of the number of bits could be chosen proportional to the harmonic sum rather than linearly, as would be suggested by Zipf's law [13], which is supposed to describe the distribution of many real-life phenomena. If B_i denotes the (not necessarily integral) number of bits used to decide if the cutoff point should be after the i-th byte, then we have, for example, $B_1 = 32$, and for $i \geq 1$,

$$B_{i+1} = B_i - \frac{32}{i \cdot H_n},$$

where H_n is the n-th harmonic number, equal to $\ln n - 0.577$. For $n = 4K = 4096$, we have $H_n = 8.895$. This would exhibit a steeper decrease at the beginning but the difference between consecutive steps would be decreasing by itself.

The plots corresponding to the harmonic decrease appear as dashed lines in the graphs of Figures 7(a)–(c). Using again 4K steps to decrease the number of bits harmonically from 32 to 0 gives a nicely symmetrical bell shaped curve for the distribution of the chunk lengths, but the average is low at 487, and practically all the

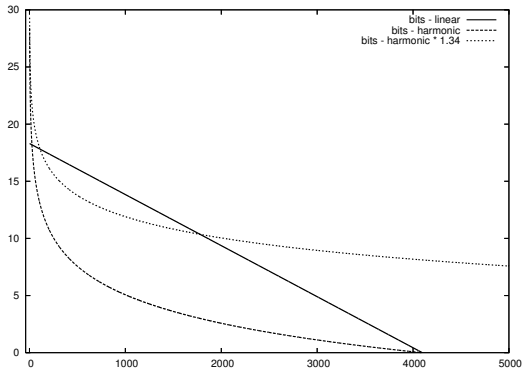


Figure 7(a). Continuous number of bits in hash function.

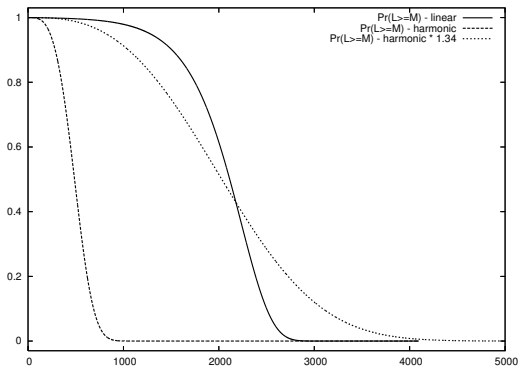


Figure 7(b). Cumulative probabilities for continuous decrease.

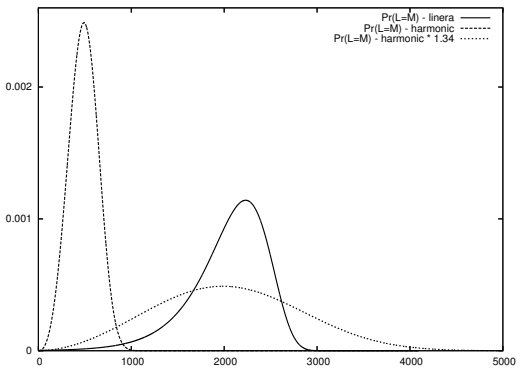


Figure 7(c). Individual probabilities for continuous decrease.

values are smaller than 1K. To move the average further up and broaden the curve, the decreasing steps can be multiplied by some constant $\alpha > 1$, so that one gets

$$B_{i+1} = B_i - \frac{32}{\alpha \cdot i \cdot H_n}.$$

The dashed line in the plots correspond to $\alpha = 1$ and the dotted lines to $\alpha = 1.34$, which yields an average chunk size of 2K. The first few elements of the B_i sequence are then 32, 29.32, 27.97, 27.08, 26.41, etc, but even after 5900 steps, the number of bits used is still about 7.14.

We also experimented with other decreasing functions than the harmonic sum, e.g., having the difference $B_n - B_{n+1}$ between consecutive bit-sizes proportional to $1/\sqrt{n}$, $\log n/n$, and others, but the harmonic decrease with parameter α gave the most appealing results.

4 Experimental results

The setup for deduplication experiments in order to get some idea on the performance of new suggested ideas is problematic. While there are well established test cases which have been agreed upon in the compression community, like the Calgary or the Canterbury [5] corpora, there is no equivalent for deduplication tests. The reason is mainly that the performance does not depend on the nature of the files, but rather

on their repetitiveness. Thus even an individual file containing random data, which cannot be compressed, may still profit from deduplication if it or any of its sub-parts appear more than once in the repository.

The other problem is that for deduplication to be interesting, there is a need to handle huge corpora. As there is no possibility to find data that could be deemed to be representative, the experimental results are presented as examples only, without claiming that one could extrapolate from them information on the performance in general. For the same reason, we did not implement the more involved techniques with fractional bits in our tests. Nevertheless, the results on our real-life tests may be considered as support, if not as evidence, for the feasibility of our approach.

Our test files were a collection of gold Virtual Machine images for a variety of different Linux OS variants and versions of total size 33.42 GB. This repository was first processed by a chunking procedure using a constant probability for setting the boundaries, aiming at an average chunk size of about 2K. Then the experiment was repeated with the varying cutoff conditions proposed herein. For both settings, a seed size of 48 bytes = 384 bits was chosen. The maximal length was set to $A_U = 6$K. Table 1 gives some statistical details. The first three columns relate to the full system including all the chunks. The last columns correspond to the chunks that have been stored, that is, without duplicates.

chunking strategy	All chunks			Unique chunks		
	number in million	average size in bytes	standard deviation	number in million	average size in bytes	standard deviation
constant	15.7	2127	2347	5.5	2502	2568
variable probability	15.8	2176	1014	5.9	2273	1081

Table 1. Details on the different chunking procedures.

As can be seen, average and standard deviation are very close for the constant variant, as is expected for an exponential distribution. For the variable probabilities, the standard deviation is much smaller, indicating that most values are closer to the mean, which is about 2K in both cases. The plots in Figures 8(a) and 8(b) are histograms showing the distribution of the chunk sizes obtained by these procedures for the unique chunks, Figure 8(a) using the constant cutoff condition, and Figure 8(b) corresponding to the procedure proposed in this work based on varying probabilities to declare a chunk boundary. The y axis gives the number of chunks as a function of a given size x on the x-axis. Although the average chunk size was close to 2K, there was a very long tail in the distribution with the constant condition, and we display here only the values up to a size of 12K, where there were still around 200 occurrences for any chunk size. In spite of the fluctuations due to various anomalies of the real-life input data, the exponentially decreasing trend of the function in Figure 8(a) is clearly noticeable.

By contrast, the distribution in Figure 8(b) corresponding to varying cutoff conditions is hedgehog shaped with an underlying Gaussian bell curve. We intend in future work to extend the tests also to the models using fractional bits.

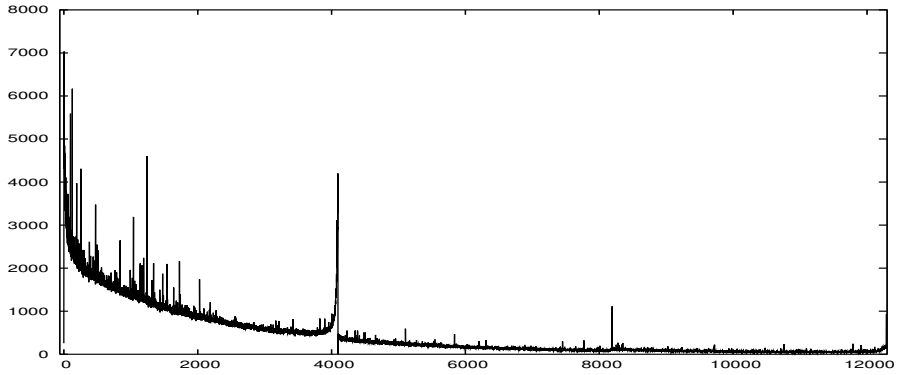


Figure 8(a). Chunk distribution with constant cutoff probability.

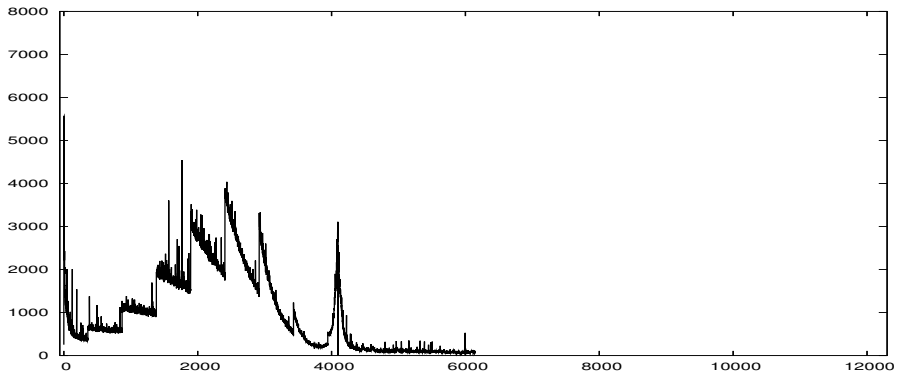


Figure 8(b). Chunk distribution with varying cutoff probability.

References

1. L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein: *The Design of a Similarity Based Deduplication System*, in Proc. SYSTOR'09, Haifa 2009, pp. 1–14.
2. L. Aronovich, R. Asher, D. Harnik, M. Hirsch, S. T. Klein, and Y. Toaff: *Similarity Based Deduplication with Small Data Chunks*, in Proc. of the Prague Stringology Conference PSC 2012, Prague 2012, pp. 3–17.
3. N. Bjørner, A. Blass, and Y. Gurevich: *Content-Dependent Chunking for Differential Compression, The Local Maximum Approach*. Journal of Computer and System Sciences 76(3–4), 2010, pp. 154–203.
4. B. Cai, F. L. Zhang, and C. Wang: *Research on Chunking Algorithms of Data De-Duplication*, in Proc. of the 2012 International Conference on Communication, Electronics and Automation Engineering, Xi'an, China, Advances in Intelligent Systems and Computing 181, 2013, pp. 1019–1025.
5. http://corpus.canterbury.ac.nz/
6. R. M. Karp and M. O. Rabin: *Efficient Randomized Pattern-Matching Algorithms*. IBM Journal of Research and Development 31(2), 1987, pp. 249–260.
7. M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble: *Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality*, in Proc. of 7th USENIX Conference on File and Storage Technologies, FAST'09, San Francisco, CA, 2009, pp. 111–123.
8. G. H. Moulton and S. B. Whitehill: *Hash file system and method for use in a commonality factoring system*. U.S. Pat. No. 6,704,730, issued March 9, 2004.
9. A. Muthitacharoen, B. Chen, and D. Mazières: *A Low-bandwidth Network File System*, in Proc. of the 18th ACM Symposium on Operating System Principles, Banff, Alberta 2001, pp. 174–187.

10. S. Quinlan and S. Dorward: *Venti: A New Approach to Archival Storage*, in Proc. of FAST'02, the 1st USENIX Conference on File and Storage Technologies, Monterey, CA, 2002, pp. 89–101.
11. P. Shilane, G. Wallace, M. Huang, and W. Hsu: *Delta Compressed and Deduplicated Storage Using Stream-Informed Locality*, in Proc. 4th USENIX Workshop on Hot Topics in Storage and File Systems, Boston, MA, 2012, p. 10.
12. B. Zhu, K. Li, H. Patterson: *Avoiding the Disk Bottleneck in the Data Domain Deduplication File System*, in Proc. FAST'08, the 6th USENIX Conference on File and Storage Technologies, San Jose, CA, 2008, pp. 279–292.
13. G. K. Zipf: *The Psycho-Biology of Language*, Boston, Houghton 1935.

A Formal Framework for Stringology

Neerja Mhaskar[1] and Michael Soltys[2]

[1] McMaster University
Dept. of Computing & Software
1280 Main Street West
Hamilton, Ontario L8S 4K1, CANADA
pophlin@mcmaster.ca
[2] California State University Channel Islands
Dept. of Computer Science
One University Drive
Camarillo, CA 93012, USA
michael.soltys@csuci.edu

Abstract. A new formal framework for Stringology is proposed, which consists of a three-sorted logical theory $\mathcal{S}$ designed to capture the combinatorial reasoning about finite words. A witnessing theorem is proven which demonstrates how to extract algorithms for constructing strings from their proofs of existence. Various other applications of the theory are shown. The long term goal of this line of research is to introduce the tools of Proof Complexity to the analysis of strings.

Keywords: proof complexity, string algorithms

1 Introduction

Finite strings are an object of intense scientific interest. This is due partly to their intricate combinatorial properties, and partly to their eminent applicability to such diverse fields as genetics, language processing, and pattern matching. Many techniques have been developed over the years to prove properties of finite strings, such as suffix arrays, border arrays, and decomposition algorithms such as Lyndon factorization. However, there is no unifying theory or framework, and often the results consist in clever but ad hoc combinatorial arguments. In this paper we propose a unifying theory of strings based on a three sorted logical theory, which we call $\mathcal{S}$. By engaging in this line of research, we hope to bring the richness of the advanced field of Proof Complexity to Stringology, and eventually create a unifying theory of strings.

The great advantage of this approach is that proof theory integrates proofs and computations; this can be beneficial to Stringology as it allows us to extract efficient algorithms from proofs of assertions. More concretely, if we can prove in $\mathcal{S}$ a property of strings of the form: "for all strings V, there exists a string U with property α," i.e., $\exists U \leq t\alpha(U, V)$, then we can mechanically extract an actual algorithm which computes U for any given V. For example, suppose that we show that $\mathcal{S}$ proves that every string has a certain decomposition; then, we can actually extract a procedure from the proof for computing such decompositions.

For a background on Proof Complexity see [3] which contains a complete treatment of the subject; we follow its methodology and techniques for defining our theory $\mathcal{S}$. We also use some rudimentary λ-calculus from [6] to define string constructors in our language.

Neerja Mhaskar, Michael Soltys: A Formal Framework for Stringology, pp. 90–103.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

2 Formalizing the theory of finite strings

We propose a three sorted theory that formalizes the reasoning about finite strings. We call our theory $\mathcal{S}$. The three sorts are *indices*, *symbols*, and *strings*. We start by defining a convenient and natural language for making assertions about strings.

2.1 The language of strings $\mathcal{L}_{\mathcal{S}}$

Definition 1. *$\mathcal{L}_{\mathcal{S}}$, the language of strings, is defined as follows:*

$$\mathcal{L}_{\mathcal{S}} = [0_{\text{index}}, 1_{\text{index}}, +_{\text{index}}, -_{\text{index}}, \cdot_{\text{index}}, \text{div}_{\text{index}}, \text{rem}_{\text{index}},$$

$$\mathbf{0}_{\text{symbol}}, \sigma_{\text{symbol}}, \text{cond}_{\text{symbol}}, ||_{\text{string}}, e_{\text{string}}; <_{\text{index}}, =_{\text{index}} <_{\text{symbol}}, =_{\text{symbol}}, =_{\text{string}}]$$

The table below explains the intended meaning of each symbol.

Formal	Informal	Intended Meaning
		Index
0_{index}	0	the integer zero
1_{index}	1	the integer one
$+_{\text{index}}$	$+$	integer addition
$-_{\text{index}}$	$-$	bounded integer subtraction
$\cdot_{\text{index}}$	$\cdot$	integer multiplication (we also just use juxtaposition)
$\text{div}_{\text{index}}$	div	integer division
$\text{rem}_{\text{index}}$	rem	remainder of integer division
$<_{\text{index}}$	$<$	less-than for integers
$=_{\text{index}}$	$=$	equality for integers
		Alphabet symbol
$\mathbf{0}_{\text{symbol}}$	$\mathbf{0}$	default symbol in every alphabet
σ_{symbol}	σ	unary function for generating more symbols
$<_{\text{symbol}}$	$<$	ordering of alphabet symbols
$\text{cond}_{\text{symbol}}$	cond	a conditional function
$=_{\text{symbol}}$	$=$	equality for alphabet symbols
		String
$\|\|_{\text{string}}$	$\|\|$	unary function for string length
e_{string}	e	binary fn. for extracting the i-th symbol from a string
$=_{\text{string}}$	$=$	string equality

Note that in practice we use the informal language symbols as otherwise it would be tedious to write terms, but the meaning will be clear from the context. When we write $i \leq j$ we abbreviate the formula $i < j \vee i = j$.

2.2 Syntax of $\mathcal{L}_{\mathcal{S}}$

We use metavariables $i, j, k, l, \ldots$ to denote indices, metavariables $u, v, w, \ldots$ to denote alphabet symbols, and metavariables $U, V, W, \ldots$ to denote strings. When a variable can be of any type, i.e., a meta-meta variable, we write it as $x, y, z \ldots$ We are going to use t to denote an index term, for example $i + j$, and we are going to use s to denote a symbol term, for example $\sigma\sigma\sigma\mathbf{0}$. We let T denote string terms. We are going to use Greek letters $\alpha, \beta, \gamma, \ldots$ to denote formulas.

Definition 2. *$\mathcal{L}_{\mathbb{S}}$-Terms are defined by structural induction as follows:*

1. *Every index variable is a term of type index (index term).*
2. *Every symbol variable is a term of type symbol (symbol term).*
3. *Every string variable is a term of type string (string term).*
4. *If t_1, t_2 are index terms, then so are $(t_1 \circ t_2)$ where $\circ \in \{+, -, \cdot\}$, and* $\mathrm{div}(t_1, t_2)$, $\mathrm{rem}(t_1, t_2)$.
5. *If s is a symbol term then so is σs.*
6. *If T is a string term, then $|T|$ is an index term.*
7. *If t is an index term, and T is a string term, then $e(T, t)$ is a symbol term.*
8. *All constant functions $(0_{\text{index}}, 1_{\text{index}}, \mathbf{0}_{\text{symbol}})$ are terms.*

We are going to employ the lambda operator λ for building terms of type string; we want our theory to be constructive, and we want to have a method for constructing bigger strings from smaller ones.

Definition 3. *Given a term t of type index, and given a term s of type symbol, then the following is a term T of type string:*

$$\lambda i \langle t, s \rangle. \tag{1}$$

The idea is that T is a string of length t and the i_0-th symbol of the string is obtained by evaluating s at i_0, i.e., by evaluating $s(i_0/i)$. Note that $s(i_0/i)$ is the term obtained by replacing every *free* occurrence of i in s with i_0. Note that (1) is a λ-term, meaning that i is considered to be a bound variable. For examples of string constructors see Section 2.4.

Definition 4. *$\mathcal{L}_{\mathbb{S}}$-Formulas are defined by structural induction as follows:*

1. *If t_1, t_2 are two index terms, then $t_1 < t_2$ and $t_1 = t_2$ are atomic formulas.*
2. *If s_1, s_2 are symbol terms, then $s_1 < s_2$ and $s_1 = s_2$ are atomic formulas.*
3. *If T_1, T_2 are two string terms, then $T_1 = T_2$ is an atomic formula.*
4. *If α, β are formulas (atomic or not), the following are also formulas:*

$$\neg\alpha, (\alpha \wedge \beta), (\alpha \vee \beta), \forall x \alpha, \exists x \alpha.$$

We are interested in a restricted mode of quantification. We say that an index quantifier is bounded if it is of the form $\exists i \leq t$ or $\forall i \leq t$, where t is a term of type index and i does not occur free in t. Similarly, we say that a string quantifier is bounded if it is of the form $\exists U \leq t$ or $\forall U \leq t$, where this means that $|U| \leq t$ and U does not occur in t.

Definition 5. *Let Σ_0^B be the set of $\mathcal{L}_{\mathbb{S}}$-formulas without string or symbol quantifiers, where all index quantifiers (if any) are bounded. For $i > 0$, let Σ_i^B (Π_i^B) be the set of $\mathcal{L}_{\mathbb{S}}$ formulas of the form: once the formula is put in prenex form, there are i alternations of bounded string quantifiers, starting with an existential (universal) one, and followed by a Σ_0^B formula.*

Given a formula α, and two terms s_1, s_2 of type symbol, then $\mathrm{cond}(\alpha, s_1, s_2)$ is a term of type symbol. We want our theory to be strong enough to prove interesting theorems, but not too strong so that proofs yield feasible algorithms. For this reason we will restrict the α in the $\mathrm{cond}(\alpha, s_1, s_2)$ to be Σ_0^B. Thus, given such an α and

assignments of values to its free variables, we can evaluate the truth value of α, and output the appropriate s_i, in polytime – see Lemma 8.

The alphabet symbols are as follows, $\mathbf{0}$, $\sigma\mathbf{0}$, $\sigma\sigma\mathbf{0}$, $\sigma\sigma\sigma\mathbf{0}$, ..., that is, the unary function σ allows us to generate as many alphabet symbols as necessary. We are going to abbreviate these symbols as $\sigma_0, \sigma_1, \sigma_2, \sigma_3, \ldots$ In a given application in Stringology, an alphabet of size three would be given by $\Sigma = \{\sigma_0, \sigma_1, \sigma_2\}$, where $\sigma_0 < \sigma_1 < \sigma_2$, inducing a standard lexicographic ordering. We make a point of having an alphabet of any size in the language, rather than a fixed constant size alphabet, as this allows us to formalize arguments of the type: given a particular structure describing strings, show that such strings require alphabets of a given size (see [2]).

2.3 Semantics of $\mathcal{L}_\mathcal{S}$

We denote a structure for $\mathcal{L}_\mathcal{S}$ with $\mathcal{M}$. A structure is a way of assigning values to the terms, and truth values to the formulas. We base our presentation on [3, §II.2.2]. We start with a non-empty set M called the universe. The variables in any $\mathcal{L}_\mathcal{S}$ are intended to range over M. Since our theory is three sorted, the universe $M = (I, \Sigma, S)$, where I denotes the set of indices, Σ the set of alphabet symbols, and S the set of strings.

We start by defining the semantics for the three 0-ary (constant) function symbols:

$$0^{\mathcal{M}}_{\text{index}} \in I, \quad 1^{\mathcal{M}}_{\text{index}} \in I, \quad 0^{\mathcal{M}}_{\text{symbol}} \in \Sigma,$$

for the two unary function symbol:

$$\sigma^{\mathcal{M}}_{\text{symbol}} : \Sigma \longrightarrow \Sigma, \quad |\,|^{\mathcal{M}}_{\text{string}} : S \longrightarrow I,$$

for the six binary function symbols:

$$+^{\mathcal{M}}_{\text{index}} : I^2 \longrightarrow I, \quad -^{\mathcal{M}}_{\text{index}} : I^2 \longrightarrow I, \quad \cdot^{\mathcal{M}}_{\text{index}} : I^2 \longrightarrow I$$

$$\text{div}^{\mathcal{M}}_{\text{index}} : I^2 \longrightarrow I, \quad \text{rem}^{\mathcal{M}}_{\text{index}} : I^2 \longrightarrow I, \quad e^{\mathcal{M}}_{\text{string}} : S \times I \longrightarrow \Sigma.$$

With the function symbols defined according to $\mathcal{M}$, we now associate relations with the predicate symbols, starting with the five binary predicates:

$$<^{\mathcal{M}}_{\text{index}} \subseteq I^2, \quad =^{\mathcal{M}}_{\text{index}} \subseteq I^2, \quad <^{\mathcal{M}}_{\text{symbol}} \subseteq \Sigma^2, \quad =^{\mathcal{M}}_{\text{symbol}} \subseteq \Sigma^2, \quad =^{\mathcal{M}}_{\text{string}} \subseteq S^2,$$

and finally we define the conditional function as follows: $\text{cond}^{\mathcal{M}}_{\text{symbol}}(\alpha, s_1, s_2)$ evaluates to $s_1^{\mathcal{M}}$ if $\alpha^{\mathcal{M}}$ is true, and to $s_2^{\mathcal{M}}$ otherwise.

Note that $=^{\mathcal{M}}$ must always evaluate to true equality for all types; that is, equality is hardwired to always be equality. However, all other function symbols and predicates can be evaluated in an arbitrary way (that respects the given arities).

Definition 6. *An* object assignment τ *for a structure* $\mathcal{M}$ *is a mapping from variables to the universe* $M = (I, \Sigma, S)$*, that is,* M *consists of three sets that we call* indices*,* alphabet symbols*, and* strings*.*

The three sorts are related to each other in that S can be seen as a function from I to Σ, i.e., a given $U \in S$ is just a function $U : I \longrightarrow \Sigma$. In Stringology we are interested in the case where a given U may be arbitrarily long but it maps I to a relatively small set of Σ: for example, binary strings map into $\{0, 1\} \subset \Sigma$. Since the range of U is relatively small this leads to interesting structural questions about the mapping: repetitions and patterns.

We start by defining τ on terms: $\tau^{\mathcal{M}}[\sigma]$. Note that if $m \in M$ and x is a variable, then $\tau(m/x)$ denotes the object assignment τ but where we specify that the variable x must evaluate to m.

We define the evaluation of a term t under $\mathcal{M}$ and τ, $t^{\mathcal{M}}[\tau]$, by structural induction on the definition of terms given in Section 2.1. First, $x^{\mathcal{M}}[\tau]$ is just $\tau(x)$, for each variable x. We must now define object assignments for all the functions. Recall that t, t_1, t_2 are index terms, s is a symbol term and T is a string term.

$$(t_1 \circ_{\text{index}} t_2)^{\mathcal{M}}[\tau] = (t_1^{\mathcal{M}}[\tau] \circ_{\text{index}}^{\mathcal{M}} t_2^{\mathcal{M}}[\tau]),$$

where $\circ \in \{+, -, \cdot\}$ and

$$(\text{div}(t_1, t_2))^{\mathcal{M}}[\tau] = \text{div}^{\mathcal{M}}(t_1^{\mathcal{M}}[\tau], t_2^{\mathcal{M}}[\tau]),$$

$$(\text{rem}(t_1, t_2))^{\mathcal{M}}[\tau] = \text{rem}^{\mathcal{M}}(t_1^{\mathcal{M}}[\tau], t_2^{\mathcal{M}}[\tau]).$$

and for symbol terms we have:

$$(\sigma s)^{\mathcal{M}}[\tau] = \sigma^{\mathcal{M}}(s^{\mathcal{M}}[\tau]).$$

Finally, for string terms:

$$|\mathrm{T}|^{\mathcal{M}}[\tau] = |(\mathrm{T}^{\mathcal{M}}[\tau])|.$$

$$(\mathrm{e}(T, t))^{\mathcal{M}}[\tau] = \mathrm{e}^{\mathcal{M}}(T^{\mathcal{M}}[\tau], t^{\mathcal{M}}[\tau]).$$

Given a formula α, the notation $\mathcal{M} \vDash \alpha[\tau]$, which we read as "$\mathcal{M}$ satisfies α under τ" is also defined by structural induction. We start with the basis case:

$$\mathcal{M} \vDash (s_1 <_{\text{symbol}} s_2)[\tau] \iff (s_1^{\mathcal{M}}[\tau], s_2^{\mathcal{M}}[\tau]) \in <_{\text{symbol}}^{\mathcal{M}}.$$

We deal with the other atomic predicates in a similar way:

$$\mathcal{M} \vDash (t_1 <_{\text{index}} t_2)[\tau] \iff (t_1^{\mathcal{M}}[\tau], t_2^{\mathcal{M}}[\tau]) \in <_{\text{index}}^{\mathcal{M}},$$

$$\mathcal{M} \vDash (t_1 =_{\text{index}} t_2)[\tau] \iff t_1^{\mathcal{M}}[\tau] = t_2^{\mathcal{M}}[\tau],$$

$$\mathcal{M} \vDash (s_1 =_{\text{symbol}} s_2)[\tau] \iff s_1^{\mathcal{M}}[\tau] = s_2^{\mathcal{M}}[\tau],$$

$$\mathcal{M} \vDash (T_1 =_{\text{string}} T_2)[\tau] \iff T_1^{\mathcal{M}}[\tau] = T_2^{\mathcal{M}}[\tau].$$

Now we deal with Boolean connectives:

$$\mathcal{M} \vdash (\alpha \wedge \beta)[\tau] \iff \mathcal{M} \vDash \alpha[\tau] \text{ and } \mathcal{M} \vDash \beta[\tau],$$

$$\mathcal{M} \vdash \neg\alpha[\tau] \iff \mathcal{M} \nvDash \alpha[\tau],$$

$$\mathcal{M} \vdash (\alpha \vee \beta)[\tau] \iff \mathcal{M} \vDash \alpha[\tau] \text{ or } \mathcal{M} \vDash \beta[\tau].$$

Finally, we show how to deal with quantifiers, where the object assignment τ plays a crucial role:

$$\mathcal{M} \vDash (\exists x \alpha)[\tau] \iff \mathcal{M} \vDash \alpha[\tau(m/x)] \text{ for some } m \in M,$$

$$\mathcal{M} \vDash (\forall x \alpha)[\tau] \iff \mathcal{M} \vDash \alpha[\tau(m/x)] \text{ for all } m \in M.$$

Definition 7. *Let $\underline{\mathbb{S}} = (\mathbb{N}, \Sigma, S)$ denote the standard model for strings, where $\mathbb{N}$ are the standard natural numbers, including zero, $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \ldots\}$ where the alphabet symbols are the ordered sequence $\sigma_0 < \sigma_1 < \sigma_2, \ldots$, and where S is the set of functions $U : I \longrightarrow \Sigma$, and where all the function and predicate symbols get their standard interpretations.*

Lemma 8. *Given any formula $\alpha \in \Sigma_0^B$, and a particular object assignment τ, we can verify $\underline{\mathbb{S}} \vDash \alpha[\tau]$ in polytime in the lengths of the strings and values of the indices in α.*

Proof. We first show that evaluating a term t, i.e., computing $t^{\underline{\mathbb{S}}}[\tau]$, can be done in polytime. We do this by structural induction on t. If t is just a variable then there are three cases: i, u, U. $i^{\underline{\mathbb{S}}}[\tau] = \tau(i) \in \mathbb{N}$, $u^{\underline{\mathbb{S}}}[\tau] = \tau(u) \in \Sigma$, and $U^{\underline{\mathbb{S}}}[\tau] = \tau(U) \in S$. Note that the assumption is that computing $\tau(x)$ is for free, as τ is given as a table which states which free variable gets replaced by what concrete value. Recall that all index values are assumed to be given in unary, and all the function operations we have are clearly polytime in the values of the arguments (index addition, subtraction, multiplication, etc.).

Now suppose that we have an atomic formula such as $(t_1 < t_2)^{\underline{\mathbb{S}}}[\tau]$. We already established that $t_1^{\underline{\mathbb{S}}}[\tau]$ and $t_2^{\underline{\mathbb{S}}}[\tau]$ can be computed in polytime, and comparing integers can also be done in polytime. Same for other atomic formulas, and the same holds for Boolean combinations of formulas. What remains is to consider quantification; but we are only allowed bounded index quantification: $(\exists i \leq t\alpha)^{\underline{\mathbb{S}}}[\tau]$, and $(\exists i \leq t\alpha)^{\underline{\mathbb{S}}}[\tau]$. This is equivalent to computing:

$$\bigvee_{j=0}^{t^{\underline{\mathbb{S}}}[\tau]} \alpha^{\underline{\mathbb{S}}}[\tau(j/i)], \text{ and } \bigwedge_{j=0}^{t^{\underline{\mathbb{S}}}[\tau]} \alpha^{\underline{\mathbb{S}}}[\tau(j/i)].$$

Clearly this can be done in polytime. □

2.4 Examples of string constructors

The string 000 can be represented by:

$$\lambda i\langle 1 + 1 + 1, \mathbf{0}\rangle.$$

Given an integer n, let $\hat{n}$ abbreviate the term $1 + 1 + \cdots + 1$ consisting of n many 1s. Using this convenient notation, a string of length 8 of alternating 1s and 0s can be represented by:

$$\lambda i\langle \hat{8}, \mathrm{cond}(\exists j \leq i(j + j = i), \mathbf{0}, \sigma\mathbf{0})\rangle. \tag{2}$$

Note that this example illustrates that indices are going to be effectively encoded in unary; this is fine as we are proposing a theory for strings, and so unary indices are an encoding that is linear in the length of the string. The same point is made in [3], where the indices are assumed to be encoded in unary, because the main object under investigation are binary strings, and the complexity is measured in the lengths of the strings, and unary encoded indices are proportional to those lengths.

Also note that there are various ways to represent the same string; for example, the string given by (2) can also be written thus:

$$\lambda i\langle \hat{2} \cdot \hat{4}, \mathrm{cond}(\exists j \leq i(j + j = i + 1), \sigma\mathbf{0}, \mathbf{0})\rangle. \tag{3}$$

For convenience, we define the empty string ε as follows:

$$\varepsilon := \lambda i\langle 0, \mathbf{0}\rangle.$$

Let U be a binary string, and suppose that we want to define $\bar{U}$, which is U with every 0 (denoted $\mathbf{0}$) flipped to 1 (denote $\sigma\mathbf{0}$), and every 1 flipped to 0. We can define $\bar{U}$ as follows:

$$\bar{U} := \lambda i\langle |U|, \mathrm{cond}(e(U,i) = \mathbf{0}, \sigma\mathbf{0}, \mathbf{0}\rangle.$$

We can also define a string according to properties of positions of indices; suppose we wish to define a binary string of length n which has one in all positions which are multiples of 3:

$$U_3 := \lambda i\langle \hat{n}, \mathrm{cond}(\exists j \leq n(i = j + j + j), \sigma\mathbf{0}, \mathbf{0})\rangle.$$

Note that both $\bar{U}$ and U_3 are defined with the conditional function where the formula α conforms to the restriction: variables are either free (like U in $\bar{U}$), or, if quantified, all such variables are bounded and of type index (like j in U_3).

Note that given a string W, $|W|$ is its length. However, we number the positions of a string starting at zero, and hence the last position is $|W| - 1$. For $j \geq |W|$ we are going to define a string to be just $\mathbf{0}$s.

Suppose we want to define the reverse of a string, namely if $U = u_0u_1 \cdots u_{n-1}$, then its reverse is $U^R = u_{n-1}u_{n-2} \cdots u_0$. Then,

$$U^R := \lambda i\langle |U|, e(U, (|U| - 1) - i)\rangle,$$

and the concatenation of two strings, which we denote as "$\cdot$", can be represented as follows:

$$U \cdot V := \lambda i\langle |U| + |V|, \mathrm{cond}(i < |U|, e(U,i), e(V, i - |U|))\rangle. \tag{4}$$

2.5 Axioms of the theory $\mathcal{S}$

We assume that we have the standard equality axioms which assert that equality is true equality — see [1, §2.2.1]. So we won't give those axioms explicitly.

Since we are going to use the rules of Gentzen's calculus, LK, we present the axioms as Gentzen's sequents, that is, they are of the form $\Gamma \to \Delta$, where Γ, Δ are coma-separated lists of formulas. That is, a sequent is of the form:

$$\alpha_1, \alpha_2, \ldots, \alpha_n \to \beta_1, \beta_2, \ldots, \beta_m,$$

where n or m (or both) may be zero, that is, Γ or Δ (or both) may be empty. The semantics of sequents is as follows: a sequent is valid if for any structure $\mathfrak{M}$ that satisfies all the formulas in Γ, satisfies at least one formula in Δ. Using the standard Boolean connectives this can be state as follows: $\neg \bigwedge_i \alpha_i \vee \bigvee_j \beta_j$, where $1 \leq i \leq n$ and $1 \leq j \leq m$.

The index axioms are the same as 2-BASIC in [3, pg. 96], plus we add four more axioms (B7 and B15, B8 and B16) to define bounded subtraction, as well as division and remainder functions. Keep in mind that a formula α is equivalent to a sequent $\to \alpha$, and so, for readability we sometimes mix the two.

Index Axioms	
B1. $i+1 \neq 0$	B9. $i \leq j, j \leq i \to i = j$
B2. $i+1 = j+1 \to i = j$	B10. $i \leq i+j$
B3. $i+0 = i$	B11. $0 \leq i$
B4. $i+(j+1) = (i+j)+1$	B12. $i \leq j \vee j \leq i$
B5. $i \cdot 0 = 0$	B13. $i \leq j \leftrightarrow i < j+1$
B6. $i \cdot (j+1) = (i \cdot j)+i$	B14. $i \neq 0 \to \exists j \leq i(j+1 = i)$
B7. $i \leq j, i+k = j \to j-i = k$	B15. $i \not\leq j \to j-i = 0$
B8. $j \neq 0 \to \mathrm{rem}(i,j) < j$	B16. $j \neq 0 \to i = j \cdot \mathrm{div}(i,j) + \mathrm{rem}(i,j)$

The alphabet axioms express that the alphabet is totally ordered according to "$<$" and define the function cond.

Alphabet Axioms
B17. $u \lneqq \sigma u$
B18. $u < v, v < w \to u < w$
B19. $\alpha \to \mathrm{cond}(\alpha, u, v) = u$
B20. $\neg\alpha \to \mathrm{cond}(\alpha, u, v) = v$

Note that α in cond is a formula with the following restrictions: it only allows bounded index quantifiers and hence evaluates to true or false once all free variables have been assigned values. Hence cond always yields the symbol term s_1 or the symbol term s_2, according to the truth value of α.

Note that the alphabet symbol type is defined by four axioms, B17–B20, two of which define the cond function. These four axioms define symbols to be ordered "place holders" and nothing more. This is consistent with alphabet symbols in classical Stringology, where there are no operations defined on them (for example, we do not add or multiply alphabet symbols).

Finally, these are the axioms governing strings:

String Axioms
B21. $\lvert\lambda i\langle t, s\rangle\rvert = t$
B22. $j < t \to e(\lambda i\langle t, s\rangle, j) = s(j/i)$
B23. $\lvert U\rvert \leq j \to e(U, j) = \mathbf{0}$
B24. $\lvert U\rvert = \lvert V\rvert, \forall i < \lvert U\rvert e(U, i) = e(V, i) \to U = V$

Note that axioms B22–24 define the structure of a string. In our theory, a string can be given as a variable, or it can be constructed. Axiom B21 defines the length of the constructed strings, and axiom B22 shows that if j is less than the length of the string, then the symbol in position j is given by substituting j for all the free occurrences of i in s; this is the meaning of $s(j/i)$. On the other hand, B23 says that if j is greater or equal to the length of a string, then $e(U, j)$ defaults to $\mathbf{0}$. The last axioms, B24, says that if two strings U and V have the same length, and the corresponding symbols are equal, then the two strings are in fact equal.

In axiom B24 there are three types of equalities, from left to right: index, symbol, and string, and so B24 is the axiom that ties all three sorts together. Note that formally strings are infinite ordered sequences of alphabet symbols. But we conclude that they are equal based on comparing finitely many entries ($\forall i < \lvert U\rvert e(U, i) = e(V, i)$). This works because by B23 we know that for $i \geq \lvert U\rvert$, $e(U, i) = e(V, i) = \mathbf{0}$ (since $\lvert U\rvert = \lvert V\rvert$ by the assumption in the antecedent). A standard string of length n is an object of the form:

$$\sigma_{i_0}, \sigma_{i_1}, \ldots, \sigma_{i_{n-1}}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \ldots,$$

i.e., an infinite string indexed by the natural numbers, where there is a position so that all the elements greater than that position are **0**.

A rich source of insight is to consider non-standard models of a given theory. We have described $\mathbb{S}$, the standard theory of strings, which is intended to capture the mental constructs that Stringologists have in mind when working on problems in this field. It would be very interesting to consider non-standard strings that satisfy all the axioms, and yet are not the "usual" object.

2.6 The rules of $\mathcal{S}$

We use the Gentzen's predicate calculus, LK, as presented in [1].

Weak structural rules

exchange-left: $\dfrac{\Gamma_1, \alpha, \beta, \Gamma_2 \to \Delta}{\Gamma_1, \beta, \alpha, \Gamma_2 \to \Delta}$ exchange-right: $\dfrac{\Gamma \to \Delta_1, \alpha, \beta, \Delta_2}{\Gamma \to \Delta_1, \beta, \alpha, \Delta_2}$

contraction-left: $\dfrac{\alpha, \alpha, \Gamma \to \Delta}{\alpha, \Gamma \to \Delta}$ contraction-right: $\dfrac{\Gamma \to \Delta, \alpha, \alpha}{\Gamma \to \Delta, \alpha}$

weakening-left: $\dfrac{\Gamma \to \Delta}{\alpha, \Gamma \to \Delta}$ weakening-right: $\dfrac{\Gamma \to \Delta}{\Gamma \to \Delta, \alpha}$

Cut rule $\dfrac{\Gamma \to \Delta, \alpha \quad \alpha, \Gamma \to \Delta}{\Gamma \to \Delta}$

Rules for introducing connectives

$\neg$-left: $\dfrac{\Gamma \to \Delta, \alpha}{\neg\alpha, \Gamma \to \Delta}$ $\neg$-right: $\dfrac{\alpha, \Gamma \to \Delta}{\Gamma \to \Delta, \neg\alpha}$

$\wedge$-left: $\dfrac{\alpha, \beta, \Gamma \to \Delta}{\alpha \wedge \beta, \Gamma \to \Delta}$ $\wedge$-right: $\dfrac{\Gamma \to \Delta, \alpha \quad \Gamma \to \Delta, \beta}{\Gamma \to \Delta, \alpha \wedge \beta}$

$\vee$-left: $\dfrac{\alpha, \Gamma \to \Delta \quad \beta, \Gamma \to \Delta}{\alpha \vee \beta, \Gamma \to \Delta}$ $\vee$-right: $\dfrac{\Gamma \to \Delta, \alpha, \beta}{\Gamma \to \Delta, \alpha \vee \beta}$

Rules for introducing quantifiers

$\forall$-left: $\dfrac{\alpha(t), \Gamma \to \Delta}{\forall x\alpha(x), \Gamma \to \Delta}$ $\forall$-right: $\dfrac{\Gamma \to \Delta, \alpha(b)}{\Gamma \to \Delta, \forall x\alpha(x)}$

$\exists$-left: $\dfrac{\alpha(b), \Gamma \to \Delta}{\exists x\alpha(x), \Gamma \to \Delta}$ $\exists$-right: $\dfrac{\Gamma \to \Delta, \alpha(t)}{\Gamma \to \Delta, \exists x\alpha(x)}$

Note that b must be free in Γ, Δ.

Induction rule

$$\text{Ind:} \quad \frac{\Gamma, \alpha(i) \to \alpha(i+1), \Delta}{\Gamma, \alpha(0) \to \alpha(t), \Delta}$$

where i does not occur free in Γ, Δ, and t is any term of type index. By restricting the quantifier structure of α, we control the strength of this induction. We call Σ_i^B-Ind to be the induction rule where α is restricted to be in Σ_i^B. We are mainly interested in Σ_i^B-Ind where $i = 0$ or $i = 1$.

Definition 9. *Let $\mathcal{S}_i$ to be the set of formulas (sequents) derivable from the axioms B1-24 using the rules of LK, where the α formula in* cond *is restricted to be in Σ_0^B and where we use Σ_i^B-Ind.*

Theorem 10 (Cut-Elimination). *If Φ is a $\mathcal{S}_i$ proof of a formula α, then Φ can always be converted into a Φ' $\mathcal{S}_i$ proof where the cut rule is applied only to formulas in Σ_i^B.*

We do not prove Theorem 10, but the reader is pointed to [5] to see the type of reasoning that is required. The point of the Cut-Elimination Theorem is that in any $\mathcal{S}_i$ proof we can always limit all the intermediate formulas to be in Σ_i^B, i.e., we do not need to construct intermediate formulas whose quantifier complexity is more than that of the conclusion.

As an example of the use of S_i we outline an S_0 proof of the equality of (2) and (3). First note that by axiom B21 we have that:

$$|\lambda i\langle\hat{8}, \mathrm{cond}(\exists j \leq i(j + j = i), \mathbf{0}, \sigma\mathbf{0})\rangle| = \hat{8}$$
$$|\lambda i\langle\hat{2} \cdot \hat{4}, \mathrm{cond}(\exists j \leq i(j + j = i + 1), \sigma\mathbf{0}, \mathbf{0})\rangle| = \hat{2} \cdot \hat{4},$$

and by axioms B1-16 we can prove that $\hat{8} = \hat{2} \cdot \hat{4}$ (the reader is encouraged to fill in the details), and so we can conclude by transitivity of equality (equality is always true equality) that:

$$|\lambda i\langle\hat{8}, \mathrm{cond}(\exists j \leq i(j + j = i), \mathbf{0}, \sigma\mathbf{0})\rangle| = |\lambda i\langle\hat{2} \cdot \hat{4}, \mathrm{cond}(\exists j \leq i(j + j = i + 1), \sigma\mathbf{0}, \mathbf{0})\rangle|.$$

Now we have to show that:

$$\forall i < \hat{8}(\mathrm{cond}(\exists j \leq i(j + j = i), \mathbf{0}, \sigma\mathbf{0}) = \mathrm{cond}(\exists j \leq i(j + j = i + 1), \sigma\mathbf{0}, \mathbf{0})) \quad (5)$$

and then, using axiom B24 and some cuts on Σ_0^B formulas we can prove that in fact the two terms given by (2) and (3) are equal.

In order to prove (5) we show that:

$$i < \hat{8} \wedge (\mathrm{cond}(\exists j \leq i(j + j = i), \mathbf{0}, \sigma\mathbf{0}) = \mathrm{cond}(\exists j \leq i(j + j = i + 1), \sigma\mathbf{0}, \mathbf{0})) \quad (6)$$

and then we can introduce the quantifier with $\forall$-intro right. We prove (6) by proving:

$$i < \hat{8} \rightarrow \mathrm{cond}(\exists j \leq i(j + j = i), \mathbf{0}, \sigma\mathbf{0}) = \mathrm{cond}(\exists j \leq i(j + j = i + 1), \sigma\mathbf{0}, \mathbf{0}) \quad (7)$$

Now to prove (7) we have to show that:

$$\mathcal{S}_0 \vdash \exists j \leq i(j + j = i) \leftrightarrow \neg\exists j \leq i(j + j = i + 1),$$

which again is left to the reader. Then, using B19 and B20 we can show (7).

3 Witnessing theorem for $\mathcal{S}$

Recall that $\mathcal{S}_1$ is our string theory restricted to Σ_1^B-Ind. For convenience, we sometimes use the notation bold-face V, $\boldsymbol{V}$, to denote several string variables, i.e., $\boldsymbol{V} = V_1, V_2, \ldots, V_\ell$.

We now prove the main theorem of the paper, showing that if we manage to prove in $\mathcal{S}_1$ the existence of a string U with some given properties, then in fact we can construct such a string with a polytime algorithm.

Theorem 11 (Witnessing). *If $\mathcal{S}_1 \vdash \exists \boldsymbol{U} \leq t\alpha(U, \boldsymbol{V})$, then it is possible to compute $\boldsymbol{U}$ in polynomial time in the total length of all the string variables in $\boldsymbol{V}$ and the value of all the free index variables in α.*

Proof. We give a bare outline of the proof of the Witnessing theorem.

By Lemma 8 we know that we can evaluate any $\mathcal{L}_\mathcal{S}$-term in $\underline{\mathbb{S}}$ in polytime in the length of the free string variables and the values of the index variables. In order to simplify the proof we show it for $\mathcal{S}_1 \vdash \exists U \leq t\alpha(U, \boldsymbol{V})$, i.e., U is a single string variable rather than a set, i.e., rather than a block of bounded existential string quantifiers. The general proof is very similar.

We argue by induction on the number of lines in the proof of $\exists U \leq t\alpha(U, \boldsymbol{V})$ that U can be witnessed by a polytime algorithm. Each line in the proof is either an axiom (see Section 2.5), or follows from previous lines by the application of a rule (see Section 2.6). By Theorem 10 we know that all the formulas in the $\mathcal{S}_1$ proof of $\exists U \leq t\alpha(U, \boldsymbol{V})$ can be restricted to be Σ_1^B. It is this fundamental application of Cut-Elimination that allows us to prove our Witnessing theorem.

The Basis Case is simple as the axioms have no string quantifiers. In the induction step the two interesting cases are $\exists$-right and the induction rule. In the former case we have:

$$\exists\text{-right: } \frac{|T| \leq t, \Gamma \to \Delta, \alpha(T, \boldsymbol{V}, \boldsymbol{i})}{\Gamma \to \Delta, \exists U \leq t\alpha(U, \boldsymbol{V}, \boldsymbol{i})}$$

which is the $\exists$-right rule adapted to the case of bounded string quantification. We use $\boldsymbol{V}$ to denote all the free string variables, and $\boldsymbol{i}$ to denote explicitly all the free index variables. Then U is naturally witnessed by the function f:

$$f(\boldsymbol{A}, \boldsymbol{b}) := T^{\underline{\mathbb{S}}}[\tau(\boldsymbol{A}/\boldsymbol{V})(\boldsymbol{b}/\boldsymbol{i})].$$

Note that f is polytime as evaluating T under $\underline{\mathbb{S}}$ and any object assignment can by done in polytime by Lemma 8.

The induction case is a little bit more involved. We restate the rule as follows in order to make all the free variables more explicit:

$$\frac{U \leq t, \alpha(U, \boldsymbol{V}, i, \boldsymbol{j}) \to \exists U \leq t\alpha(U, \boldsymbol{V}, i+1, \boldsymbol{j})}{U \leq t, \alpha(U, \boldsymbol{V}, 0, \boldsymbol{j}) \to \exists U \leq t\alpha(U, \boldsymbol{V}, t', \boldsymbol{j})}$$

where we ignore Γ, Δ for clarity, and we ignore existential quantifiers on the left side, as it is quantifiers on the right side that we are interested in witnessing. The algorithm is clear: suppose we have a U such that $\alpha(U, \boldsymbol{V}, 0, \boldsymbol{V})$ is satisfied. Use top of rule to compute U's for $i = 1, 2, \ldots, t^{\underline{\mathbb{S}}}[\tau]$. □

4 Application of $\mathcal{S}$ to Stringology

In this section we state various basic Stringology constructions as $\mathcal{L}_\mathcal{S}$ formulas.

4.1 Subwords

The prefix, suffix, and subword are basic constructs of a given string V. They can be given easily as $\mathcal{L}_S$-terms as follows: $\lambda k\langle i, e(V,k)\rangle$, $\lambda k\langle i, e(V, |V| - i + 1 + k)\rangle$, and since any subword is the prefix of some suffix, it can also be given easily.

We can state that U is a prefix of V with the Σ_0^B predicate:

$$\mathrm{pre}(U,V) := \exists i \leq |V|(U = \lambda k\langle i, e(V,k)\rangle),$$

The predicates for suffix $\mathrm{suf}(U,V)$ and subword $\mathrm{sub}(U,V)$ predicates can be defined with Σ_0^B formulas in a similar way.

4.2 Counting symbols

Suppose that we want to count the number of occurrences of a particular symbol σ_i in a given string U; this can be defined with the notation $(U)_{\sigma_i}$, but we need to define this function with a new axiom (it seems that the language given thus far is not suitable for defining $(U)_{\sigma_i}$ with a term). First, define the projection of a string U according to σ_i as follows:

$$U|_{\sigma_i} := \lambda k\langle |U|, \mathrm{cond}(e(U,k) = \sigma_i, \sigma_1, \sigma_0)\rangle.$$

That is, $U|_{\sigma_i}$ is effectively a binary string with 1s where U had σ_i, and 0s everywhere else, and of the same length as U. Thus, counting σ_i's in U is the same as counting 1's in $U|_{\sigma_i}$. Given a binary string V, we define $(V)_{\sigma_1}$ as follows:

C1. $|V| = 0 \rightarrow (V)_{\sigma_1} = 0$
C2. $|V| \geq 1, e(V,0) = \sigma_0 \rightarrow (V)_{\sigma_1} = (\lambda i\langle |V| - 1, e(V, i+1)\rangle)_{\sigma_1}$
C3. $|V| \geq 1, e(V,0) = \sigma_1 \rightarrow (V)_{\sigma_1} = 1 + (\lambda i\langle |V| - 1, e(V, i+1)\rangle)_{\sigma_1}$

Having defined $(U)_{\sigma_1}$ with axioms C1-3, and $U|_{\sigma_i}$ as a term in $\mathcal{L}_S$, we can now define $(U)_{\sigma_i}$ as follows: $(U|_{\sigma_i})_{\sigma_1}$. Note that C1-3 are Σ_0^B sequents.

4.3 Borders and border arrays

Suppose that we want to define a border array. First define the border predicate which asserts that the string V has a border of size i; note that by definition a border is a (proper) prefix equal to a (proper) suffix. So let:

$$\mathrm{Brd}(V,i) := \lambda k\langle i, e(V,k)\rangle = \lambda k\langle i, e(V, |V| - i + 1 + k)\rangle \wedge i < |V|,$$

We now want to state that i is the largest possible border size:

$$\mathrm{MaxBrd}(V,i) := \mathrm{Brd}(V,i) \wedge (\neg \mathrm{Brd}(V, i+1) \vee |U| = |V| - 1).$$

Thus, if we want to define the function $\mathrm{BA}(V,i)$, which is the border array for V indexed by i, we can define it by adding the following as an axiom:

$$\mathrm{MaxBrd}(\lambda k\langle i, e(V,k)\rangle, \mathrm{BA}(V,i)).$$

4.4 Periodicity

See [4, pg. 10] for the definition of a period of a string, but for our purpose let us define $p = |U|$ to be a period of V if $V = U^r U'$ where U' is some prefix, possibly empty, of U. The Periodicity Lemma state the following: Suppose that p and q are two periods of V, $|V| = n$, and $d = \gcd(p, q)$. Then, if $p + q \leq n + d$, then d is also a period of V.

Let $\mathrm{Prd}(V, p)$ be true if p is a period of the string V. Note that U is a border of a string V if and only if $p = |V| - |U|$ is a period of V. Using this observation we can define the predicate for a period as a Σ_0^B formula:

$$\mathrm{Prd}(V, p) := \exists i < |V|(p = |V| - i \wedge \mathrm{Brd}(V, i))$$

We can state with a Σ_0^B formula that $d = \gcd(i, j)$: $\mathrm{rem}(d, i) = \mathrm{rem}(d, j) = 0$, and $\mathrm{rem}(d', i) = \mathrm{rem}(d', j) = 0 \supset d' \leq d$. We can now state the Periodicity Lemma as the sequent $\mathrm{PL}(V, p, q)$ where all formulas are Σ_0^B:

$$\mathrm{Prd}(V, p), \mathrm{Prd}(V, q), \exists d \leq p(d = \gcd(p, q) \wedge p + q \leq |V| + d) \rightarrow \mathrm{Prd}(V, d).$$

Lemma 12. $\mathcal{S}_0 \vdash \mathrm{PL}(V, p, q)$.

Proof. The proof relies on a formalization of the observation stated above linking periods and borders. □

4.5 Regular and context-free strings

We are now going to show that regular languages can be defined with a Σ_1^B formula. This means that given any regular language, described by a regular expression R, there exists a Σ_1^B formula Ψ_R such that $\Psi_R(U) \iff U \in L(R)$.

Lemma 13. *Regular languages can be defined with a Σ_1^B formula.*

Proof. We have already defined concatenation of two strings in (4), but we still need to define the operation of union and Kleene's star. All together this can be stated as:

$$\begin{aligned}
&\Psi_\cdot(U, V, W) := W = U \cdot V \\
&\Psi_\cup(U, V, W) := (W = U \vee W = V) \\
&\Psi_*(U, W) := \exists i \leq |W|(W = \lambda i \langle i \cdot |u|, e(U, \mathrm{rem}(i, |U|))\rangle)
\end{aligned}$$

Now we show that R can be represented with a Σ_1^B formula by structural induction on the definition of R. The basis case is simple as the possibilities for R are as follows: a, ε, σ, and they can be represented with $W = a, |W| = 0, 0 = 1$, respectively.

For the induction step, consider R defined from $R_1 \cdot R_2$, $R_1 \cup R_2$ and $(R_1)^*$:

$$\begin{aligned}
R = R_1 \cdot R_2 \quad & \exists U_1 \leq |W| \exists U_2 \leq |W|(\Psi_{R_1}(U_1) \wedge \Psi_{R_2}(U_2) \wedge \Psi_\cdot(U_1, U_2, W)) \\
R = R_1 \cup R_2 \quad & \exists U_1 \leq |W| \exists U_2 \leq |W|(\Psi_{R_1}(U_1) \wedge \Psi_{R_2}(U_2) \wedge \Psi_\cup(U_1, U_2, W)) \\
R = (R_1)^* \quad & \exists U_1 \leq |W| \Psi_*(U_1, W)
\end{aligned}$$

Thus, we obtain a Σ_1^B formula $\Psi_R(W)$ which is true iff $W \in L(R)$. □

Note that in the proof of Lemma 13, when we put $\Psi_R(W)$ in prenex form all the string quantifiers are bounded by $|W|$, and they can be viewed as "witnessing" intermediate strings in the construction of W.

Lemma 14. *Context-free languages can be defined with a Σ_1^B formula.*

Proof. Use Chomsky's normal form and the CYK algorithm. □

5 Conclusion and future work

We have just touched the surface of the beautiful interplay between Stringology and Proof Complexity. Lemma 8 can likely be strengthened to say that evaluating $\mathcal{L}_S$-terms can be done in $\mathbf{AC}^0$ rather than polytime. As was mentioned in the paper, the richness of the field of Stringology arises from the fact that a string U is a map $I \longrightarrow \Sigma$, where I can be arbitrarily large, while Σ is small. This produces repetitions and patterns that are the object of study for Stringology. On the other hand, Proof Complexity has studied in depth the varied versions of the Pigeonhole Principle that is responsible for these repetitions. Thus the two may enrich each other. Finally, Regular languages can be decided in $\mathbf{NC}^1$; how can this be reflected in the proof of Lemma 13? Also, prove Lemma 14.

Due to the lack of space, and the fact that it usually requires a rather lengthy construction, we did not illustrate an application of the Witnessing theorem. A very nice application can be found in the Lyndon decomposition of a string (see [4, pg. 29]). Recall that our alphabet is ordered — this was precisely so these types of arguments could be carried out naturally in our theory. Since $\sigma_0 < \sigma_1 < \sigma_2 \ldots$, we can easily define a lexicographic ordering of strings; define a predicate $U <_{\text{lex}} V$. We can define a Lyndon word with a Σ_0^B formula as follows: $\forall i < |V|(V <_{\text{lex}} \lambda k\langle i, e(V, |V|-i+1+k)\rangle)$.

Let V be a string; then $V = V_1 \cdot V_2 \cdots V_k$ is a Lyndon decomposition if each V_i is a Lyndon word, and $V_k <_{\text{lex}} V_{k-1} <_{\text{lex}} \cdots <_{\text{lex}} V_1$. The existence of a Lyndon decomposition can be proven as in [4, Theorem 1.4.9], and we assert that the proof itself can be formalized in $\mathcal{S}_1$. We can therefore conclude that the actual decomposition can be computed in polytime. As one can see, this approach provides a deep insight into the nature of strings.

References

1. S. R. Buss: *An introduction to proof theory*, in Handbook of Proof Theory, S. R. Buss, ed., North Holland, 1998, pp. 1–78.
2. S. R. Buss and M. Soltys: *Unshuffling a square is NP-hard*. Journal of Computer and System Sciences, 80(4) 2013, pp. 766–776.
3. S. A. Cook and P. Nguyen: *Logical Foundations of Proof Complexity*, Cambridge Univeristy Press, 2010.
4. B. Smyth: *Computing Patterns in Strings*, Pearson Education, 2003.
5. M. Soltys: *A model-theoretic proof of the completeness of LK proofs*, Tech. Rep. CAS-06-05-MS, McMaster University, 1999.
6. M. Soltys and S. Cook: *The proof complexity of linear algebra*. Annals of Pure and Applied Logic, 130(1–3) December 2004, pp. 207–275.

Quantum Leap Pattern Matching
A New High Performance Quick Search-Style Algorithm

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

[1] FASTAR Research Group, Department of Information Science, Stellenbosch University,
Private Bag X1, 7602 Matieland, Republic of South Africa
[2] Centre for Artificial Intelligence Research,
CSIR Meraka Institute, Republic of South Africa
[3] Department of Computer Science, Umeå University,
SE-901 87 Umeå, Sweden
{bruce,derrick,loek}@fastar.org

Abstract. Quantum leap matching is introduced as a generic pattern matching strategy for the single keyword exact pattern matching problem, that can be used on top of existing Boyer-Moore-style string matching algorithms. The cost of the technique is minimal: an additional shift table (of one dimension, for shifts in the opposite direction to the parent algorithm's shifts), and the replacement of a simple table lookup assignment statement in the original algorithm with a similar conditional assignment. Together with each of the conventional shift table lookups, the additional shift table is typically also indexed on the text character that is at a distance of z away from the current sliding window. Under conditions that are identified, the returned values from the two shift tables allow a "quantum leap" of distance more than the length of the keyword for the next matching attempt. If the conditions are not met, then there is a fall back is to the traditional shift.
Quick Search (by Sunday) is used as a case study to illustrate the technique. The performance of the derived "Quantum Leap Quick Search" algorithm is compared against Quick Search. When searching for shorter patterns over natural language and genomic texts, the technique improves on Quick Search's time for most values of z. Improvements are also sometimes seen for various values of z on larger patterns. Most interestingly, under best case conditions it performs, on average, at about *three times faster* than Quick Search.

Keywords: high-speed pattern matching single keyword matching, Boyer-Moore algorithms, Sunday's algorithm, faster pattern matching

1 Introduction

We consider the well-known single keyword string matching problem. We adopt the convention that string s is treated as an array whose length is $|s|$. Its first element is at index 0 and the element at index i is denoted by $s[i]$. A substring of length n starting at index i is denoted[1] by $s[i, i+n)$.

Given an alphabet Σ, a text string $t \in \Sigma^*$ and a single keyword or pattern string $p \in \Sigma^+$ of length $|p| = m$, the string matching problem is to find all indices of t where a *match* of p occurs. A match of p at index i *occurs* if $p[0, m) = t[i, i+m)$.

Following Cantone and Faro [1], we call $t[i, i+m)$ the *current window* of the text when $p[0]$ is aligned with $t[i]$. To solve the string matching problem, 'Boyer-Moore style' algorithms *slide* (or 'shift') the current window in t in a given direction

[1] A set of successive integers $\{i, i+1, \ldots j\}$ is commonly represented in interval notation format as one of the following: $[i, j]$, $[i, j+1)$, $(i-1, j+1)$ or $(i-1, j]$. Our motivation for this substring notation is to simplify $+1$ and -1 subscript expressions using square and round parentheses.

Bruce W. Watson, Derrick G. Kourie, Loek Cleophas: Quantum Leap Pattern Matching, pp. 104–117.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

— normally in a forward direction (from left to right), but it could also be in a backward direction (from right to left). After shifting the current window to index i a *match attempt* at i is made. If the match attempt is successful then i is recorded as a match location. Regardless of whether the attempt is successful, an offset value from i (to the right of i in classical algorithms proceeding from left to right) is found to indicate where the next match attempt should take place. Naturally, such an offset must not miss any intervening matches, and it is called a *safe* offset or shift.

The offset is usually given by a *shift table* that is precomputed from the structure of p — see [6] for examples of such a shift functions. The simplest such table is accessed by indexing by a character in the substring $t[i, m + 2)$, although certain algorithms instead use more information, such as two characters in this range and use a two-dimensional shift table. Due to its simplicity and efficiency, we specifically focus on Sunday's Quick Search shift table [9]. Shift tables are covered in books such as [2,4,8]. See [1] for a recent survey of related algorithms and shift tables, while [3] gives a *calculus* for arriving at all of the known shift tables as well as designing new ones.

When a current window at i is slid in a forward direction, a *forward shift table* provides an integer $shf \in [1, m + 2)$ indicating that the next match attempt should be at a window at $i + shf$. The heuristics used to set up the shift table guarantees that no match is missed — there is no match in the index range $[i + 1, i + shf)$. Note this must hold regardless of previous match attempts in the range $[0, i + 1)$.

Dually, if we were sliding the current window at i in a backward (right to left) direction, a *backward shift table* provides an integer $shb \in [1, m + 2)$ indicating that the next match attempt may be made at a window at $i - shb$, since no match at an index in the range $(i - shb, i)$ is possible. Again, the validity of this assertion is independent of whether match attempts have previously been made at indices in the range $[i, |t| + 1)$.

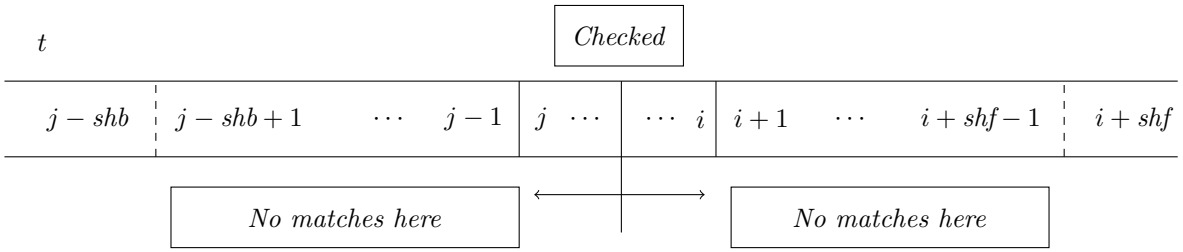
Some algorithms partition the search space over t at one or more indices of t — see [10]. Windows are placed to the right and left of such indices and, after match attempts, they are slid in a forward and backward direction respectively. The scenario is loosely depicted in Figure 1a. The figure assumes that the indices in the interval (j, i) have already been checked. It further assumes that the offset *shb* results from a match attempt at j followed by a backward table lookup, while the offset *shf* results from a match attempt at i followed by a forward table lookup.

Suppose that Figure 1b, a variant of Figure 1a, is the result of executing some abstract string matching algorithm. (Ignore for the moment the entries at the top of the figure that refer to z. They shall be addressed in Section 2.) The figure was inspired by our experience that algorithms based on Figure 1a incur penalties (including cache miss penalties) because of the bookkeeping required in respect of the partitions over t. Figure 1b assumes that a forward scan has already checked all indices in the interval $[0, i)$ for matches, thus avoiding the cache miss problem encountered by algorithms based on the latter figure. Additionally, this figure assumes that all information in Figure 1a is available and that $i < j$.

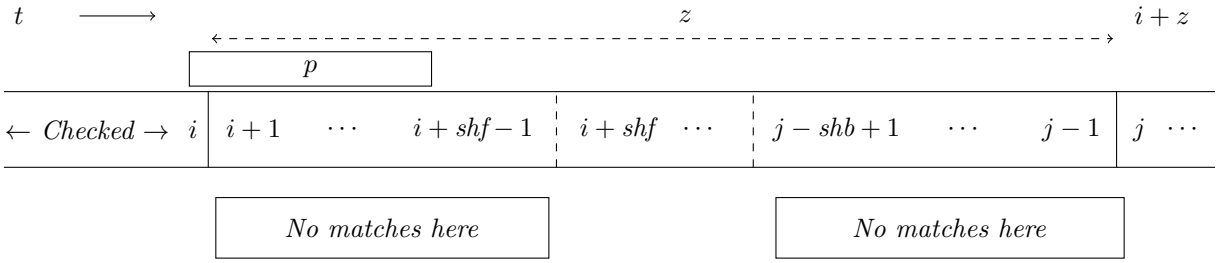
Clearly, if the predicate

$$i + shf \geq j - shb + 1 \tag{1}$$

was true, then the abstract algorithm could safely resume further processing at j, thus making a right shift that is larger than *shf*. If predicate (1) is false, then $i + shf$ serves as a fall back position from which to resume further processing.



(a) Dead Zone



(b) The Quantum Leap Matching Concept

Figure 1: Dead Zone vs Quantum Leap Matching Concept

The intuition reflected in Figure 1b served as the starting point for exploring whether and how an algorithm can be developed to exploit the possibility of *leapfrogging* over *shf* to produce larger shifts. Because this is a possibility and not a given, we call our approach the *quantum leap* strategy.

Reliance on dual shift tables may be used to update existing string matching algorithms. In Section 2 we outline how this can be done for the well-known Quick Search (QS) algorithm proposed by Sunday [9]. We call this the Quantum Leap QS algorithm (QLQS). Then Section 3 describes the empirical results yielded by this algorithm. Reflections on these results are presented in a final concluding section.

2 The QLQS Algorithm

An abstract algorithm that relies on predicate (1) is entirely generic in that it does not depend on how *shf* and *shb* are obtained. All that matters is that their values should ensure that matches are not possible in the two regions in Figure 1b marked "*No matches here*". As mentioned earlier, to derive the concrete QLQS algorithm, we decided to rely on QS's forward and backward shift tables as the dual shift tables needed for *shf* and *shb*. We compare our results against QS.

2.1 QLQS derived from QS

After a match attempt at i, QS uses the character at $t[i+m]$ as an index into its (forward) shift table to find *shf*—the offset from i for the next match attempt. It is well-known that for QS, *shf* will lie in the range $[1, m+2)$.

To proceed, an abstract algorithm based on QS would determine the value of *shb* by indexing into QS's backward shift table at the character $t[j-1]$, where j has some suitably chosen value. Unfortunately, it is not clear how to choose such a value for j. It would be pleasant if a "magical" choice of j guaranteed two conditions at *every*

match attempt: firstly, that the length of the interval $[i+1, j)$ is at a maximum; and secondly, that for the chosen j the predicate (1) continues to hold. To guarantee just the first condition would mean to have foreknowledge of the next match index and to choose j exactly at that index—something which is clearly infeasible. To guarantee the second condition without incurring the expense of additional probes between i and j is only possible for trivial choices of j. (Subsection 2.2 will examine possible ranges of j.) To avoid such computational expense, a compromise action is to rely on some fixed offset ahead of i whose compliance with predicate (1) is stochastically determined.

Let us call this offset z, and assume that $j = i + z$. This is depicted at the top of Figure 1b. In principle, at every new match attempt in a search, a different value for z could be selected according to some criterion. However, to keep things simple, our research is based on a preselected z value over the entire search. Empirical results discussed in Section 3 examine the consequences of selecting various values for z.

With elementary algebraic manipulation, predicate (1) can be rewritten in terms of z as

$$shf > z - shb \qquad (2)$$
$$\text{or equivalently} \quad shf + shb > z$$

After every match attempt, it is now necessary to check whether predicate (2) holds for the preselected value of z. To carry out this check the value of *shb* has to be obtained by looking up QS's backward shift table for the character $t[i + z - 1]$ and then evaluating the revised predicate (2). If predicate (2) turned out to be true, then the next match attempt could take place at $i + z$; otherwise the next match attempt must necessarily be at $i + shf$. Note, however, that to save on algorithmic computations during run time, the minus operation can be avoided in predicate (2) by precomputing and storing $shb' = z - shb$ instead of *shb* as the backward shift table.

A new algorithm can now be very simply derived from QS by carrying out the following four steps.

1. Select a suitable value for z;
2. Precompute the table for shb';
3. Replace the QS assignment statement that unconditionally increments i by *shf* for the next match attempt index with a conditional statement incrementing i by z if $(shf > shb')$ or by *shf* otherwise.
4. Pad the tail end of t as necessary to ensure that the reference to $t[i + z - 1]$ does not cause an array bound error in the last iteration of the main algorithm loop.

Figure 2 gives the resulting C code for this revised algorithm. It contains a counter, `count`, for the number of matches. The array `shf` stores the precomputed forward table and the array `shb` stores the precomputed values for the shb' values. The instructions differ from the conventional QS algorithm only in that the conditional assignment statement in lines 9 to 11 has replaced a conventional assignment statement, `i += shf[T[i+m]]`, that would be used in QS. The variable `z` is global to the code.

We conjecture that these simple steps, appropriately adapted, could be used to modify practically any of the common string matching algorithms.

```
1   static int search(const unsigned char *P, int m, const unsigned char *T, int n) {
2      int k, i, count;
3      i = 0;
4      count = 0;
5      while(i<=n-m) {
6          k=0;
7          while(k<m && P[i]==T[i+k]) i++;
8          if (k==m) count++;
9          i +=     (shf[T[i+m]] > shb[T[i+z-1]]
10                   ? z
11                   : shf[T[i+m]]);
12     }
13     return count;
14  }
```

Figure 2: C Code for QLQS

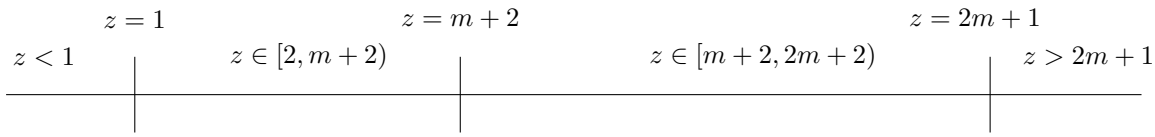


Figure 3: z Ranges Considered

2.2 Range of z Values

In QLQS outlined above, z is a constant. Variants of this algorithm might adjust the value of z dynamically to reflect text characteristics that manifest as it is being searched. It is therefore important to consider the range of values that z may legitimately and meaningfully assume, whether chosen as a constant or dynamically changed during a variant of the algorithm. In the discussion to follow, without loss of generality and for the sake of simplicity, reference to backward tables should be construed to mean those whose values are represented by *shb* and not those whose values are represented by shb'.

For given values of *shf* and *shb*, it is easy to see that $z = shf + shb - 1$ is the largest value of z that complies with predicate (2). Since the range of possible values for both *shf* and *shb* in QS shift tables is $[1, m + 2)$, the minimum and maximum of these largest possible z values are respectively attained when $shf = shb = 1$ and when $shf = shb = m + 1$. In the former instances, $shf + shb - 1$ evaluates to 1 and in the latter case $shf + shb - 1$ evaluates to $2m + 1$. The points, $z = 1$ and $z = 2m + 1$, are indicated in Figure 3, as well as various other points and ranges that will visually support the discussion that follows.

Clearly, if $z > 2m+1$ then predicate (2) cannot be satisfied for any values assumed by *shf* and *shb*, and consequently QLQS will always slide the current window ahead to $i + shf$. It will therefore execute in exactly the same way as QS, but with an additional overhead.

On the other hand, if z is selected in the range $[1, 2m + 2)$ then for some values of *shf* and *shb* within their permissible ranges predicate (2) may be satisfied, and for others, not. Whenever the predicate is satisfied, QLQS will slide its window to $i + z$, and otherwise to $i + shf$.

However, satisfying this predicate does not necessarily mean that $z > shf$ and so in these instances QLQS will not slide the window as far to the right as QS. A necessary and sufficient condition for QLQS to slide further than QS is a conjunction

of the predicates (2) and $z > shf$, namely the predicate

$$(z < shf + shb) \wedge (z > shf)$$
$$\text{or equivalently } z \in [shf + 1, shf + shb) \tag{3}$$

Since the maximal QS value for shf is $m + 1$, any selection of z in the range $[m + 2, 2m+2)$ guarantees compliance with the lower bound of the interval in predicate (3). If, in addition, the current values for shf and shb result in compliance with the upper bound—equivalently, if predicate (2) is satisfied—then QLQS will slide the window further than QS.

If $z \in [1, m + 2)$ then the current values for shf and shb may or may not render z compliant with predicate (3). If the predicate is indeed satisfied, then QLQS will slide further than QS from the current window. If predicate (3) is false but predicate (2) is satisfied, then QLQS will slide less than (or the same as, if $z = shf$) QS from the current window.

If $z = 1$, then predicate (2) is always satisfied, but predicate (3) is never satisfied. As a result, the window will always slide to $i + z = i + 1$. Thus, QLQS degenerates to the most naïve string matching algorithm—one that merely slides the window by one position in each iteration.

Finally, if $z < 1$ then inspection will confirm that predicate (2) is satisfied for all possible values of shf and shb, while predicate (3) is never satisfied. QLQS executed with $z < 1$ will therefore always slide to $i + z < i + 1$. If $z = 0$ then this means staying in the same window as before. The algorithm effectively ends up in an infinite loop, carrying out a match attempt in the same place. If $z < 0$ and the current value of $i + z \geq 0$ then the slides to the left and a region already checked before will be rechecked—the region marked *Checked* in Figure 1b. This is obviously redundant. If $z < 0$ and the $i + z < 0$, then the next match attempt will involve an out-of-range index of t. This will be the case if $z < 0$ is used in the first iteration of the algorithm. Any algorithm built around a dynamically changing value of z should account for these boundary problems.

Table 1 summarises the foregoing discussion. Columns represent differing possible choices of z as given in the column heading. The first three rows indicate whether the predicate in the row heading (on the left) is always true, always false, or possibly either depending on the specific values of shf and shb, indicated by **true**, **false** or **depends** respectively. (Note that row 2 corresponds to predicate (2) and row 3 corresponds to predicate (3).) Row 4 indicates whether the offset from i will definitely be z, or definitely shf or either one of these values, depending on whether or not predicate (2) is satisfied. The final row indicates the worst outcome for the z in each respective column.

z is	< 0	$= 0$	$= 1$	$\in [2, m + 2)$	$\in [m + 2, 2m + 2)$	$> 2m + 1$
$z > shf$	**false**	**false**	**false**	**depends**	**true**	**true**
$z < shf + shb$	**true**	**true**	**true**	**depends**	**depends**	**false**
$z \in [shf + 1, shf + shb)$	**false**	**false**	**false**	**depends**	**depends**	**false**
Offset of i	z	z	z	z or shf	z or shf	shf
Worst case outcome	Array error	Loop error	Naïve alg	$(z < shf) \wedge i = i + z$	$(z \geq shf + shb) \wedge i = i + shf$	Needless work

Table 1: Consequences of different z value choices

2.3 QLQS Behaviour

The foregoing provides a basis for theoretically assessing the behaviour of QLQS. Note that, in comparison to QS, every shift of this QLQS algorithm requires an additional table lookup and the execution of a conditional statement instead of a simple assignment statement. The potential gain for the extra computational workload is longer shifts.

For illustrative purposes, Figure 4 shows an example (taken from from [2]) of four match attempts that occur when searching for matches of $p = \texttt{GCAGAGAG}$ in a string $t \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}^{24}$. Note that t is appropriately padded at the end with `X`'s and the forward and backward shift tables are provided in Table 2.

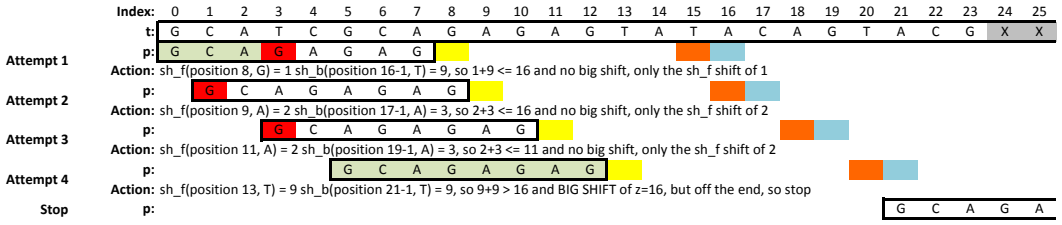


Figure 4: An example from [2]

Σ	A	C	G	T	X
shf	2	7	1	9	9
shb	3	2	1	9	9

Table 2: Shift tables for $p = \texttt{gcagagag}$

Clearly, in the best case, the maximum shift should occur at every iteration. This will be the case if z is at its maximum $(2m+1)$ and t and p are such that predicate (3) is satisfied in each iteration. Such a scenario can be constructed by choosing t and p to have disjoint character sets. In such a case, each shift of QLQS will be $2m + 1$, exceeding QS's maximal shift of $m + 1$ by m. The upper bound on the total number of shifts is $\lceil \frac{|t|}{(2m+1)} \rceil$ compared to $\lceil \frac{|t|}{(m+1)} \rceil$ for QS.

Figure 5 illustrates how windows slide under best conditions, both for QS and QLQS . Text $t = \texttt{a}^{23}$ is padded at the end with `X`'s and $p = \texttt{01234}$ is used. The four match attempts required by QS are shown, where *shf* is is looked up at $t[5], t[11], t[17]$ and $t[23]$. QLQS requires two match attempts. The first needs *shf* and *shb* lookups at $t[5]$ and $t[11]$ respectively; and the second at $t[16]$ and $t[22]$ respectively.

Worst case behaviour is manifested in both QLQS and QS if every window in t matches p. This is the case for both algorithms when $|\Sigma| = 1$. We have already pointed out that this worst case behaviour will also be exhibited if QLQS is run with z chosen as 1. In such instances, both algorithms can slide ahead by only one position at each iteration, each therefore execute $(|t| - |p|)$ iterations.

For randomly chosen t and p when $|\Sigma| > 1$, behaviour will be consistent with the analysis given in Subsection 2.2 and summarised in Table 1. Randomness implies that the values for *shf* and *shb* are randomly distributed over the interval $[1, m + 2)$—i.e. choosing a random character from Σ and indexing either shift table on this character, is equally likely to return any of the integers in the interval $[1, m + 2)$.

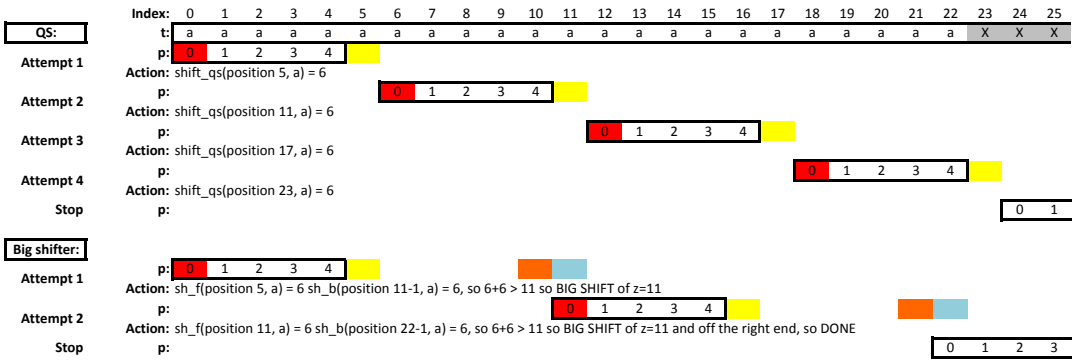


Figure 5: Example of best case behaviour

Noting that *shf* and *shb* are bound from above, and that predicate (2) is satisfied if their sum exceeds z, it is clear that the larger z that is chosen, the smaller is the probability that $shf + shb > z$ will hold and thus, the smaller the chance that z will be used as the offset for the next match attempt. This observation mitigates against using large z values in the presence of randomness. On the other hand, as indicated in Table 1, if $z < m + 2$ then $z < shf$ may be selected as the offset. The smaller the value of z that is chosen, the more likely this is to happen, whereas it definitely does not happen if $z \in [m + 2, 2m + 2)$. This observation mitigates against using small values of z.

It is beyond the scope of this research to take further these observations about random behaviour and derive symbolic expressions for statistical metrics such as the expected value or standard deviation. Instead, Section 3 reports on the empirical analysis undertaken in respect of the performance of QLQS on commonly used benchmarking data.

2.4 Degenerate Forms of QLQS

It is instructive to consider in more detail the behaviour of QLQS in the degenerate case when $|p| = 1$. Table 3 reflects this behaviour. It shows the shifts when $z = q, \ldots 4$, the pattern is the single character c and the current window in t is at i and $t[i, i+4) = c_1c_2c_3c_4$. Under these circumstances, *shf* and *shb* can ony assume the values 1 (when indexed by c) or 2 (when indexed by anything other than c). The table then shows the various possible outcomes.

- As pointed out above, when $z = 1(= m)$ the shift is always 1. Sometimes this will be when QS could do a shift by 2 and so a non-optimal shift will occur.
- When $z = 2(= m + 1)$ then QLQS shifts always correspond to QS shifts.
- When $z = 3(= 2m + 1)$ then QLQS in one instance shifts ahead by 3 positions—something that QS could not do.
- When $z = 4(= 2m + 2)$ then behaviour is as expected: only conventional QS shifts can be made, at slightly additional computational expense.

z	$shf = 1 \wedge shb = 1$	$shf = 1 \wedge shb = 2$	$shf = 2 \wedge shb = 1$	$shf = 2 \wedge shb = 2$
$z = 1$ Shift	$c_2 = c \wedge c_1 = c$ z	$c_2 = c \wedge c_1 \neq c$ z	$c_2 \neq c \wedge c_1 = c$ z Non-optimal	$c_2 \neq c \wedge c_1 \neq c$ z
$z = 2$ Shift	$c_2 = c$ *shf*	Infeasible z?	Infeasible z?	$c_2 \neq c$ z
$z = 3$ Shift	$c_2 = c \wedge c_3 = c$ *shf*	$c_2 = c \wedge c_3 \neq c$ *shf*	$c_2 \neq c \wedge c_3 = c$ *shf*	$c_2 \neq c \wedge c_3 \neq c$ z
$z = 4$ Shift	$c_2 = c \wedge c_4 = c$ *shf*	$c_2 = c \wedge c_4 \neq c$ *shf*	$c_2 \neq c \wedge c_4 = c$ *shf*	$c_2 \neq c \wedge c_4 \neq c$ *shf*

Table 3: QLQS shifts for $m = 1$, $p = c$ and $t[i, i + 4) = c_1c_2c_3c_4$.

3 The Results

3.1 Experimental Design

The hardware platform used for this study is a 17-inch Macbook Pro (early 2011), 2.2 GHz (peaks at 3.0 GHz turbo) Intel Core i7 Quad-core (with another 4 virtual cores), 8 GB of 1333 MHz DDR3 RAM, 256 KB of L2 Cache per core, 6 MB of L3 Cache. C Code was compiled with Gnu g++ Apple LLVM version 6.1.0 using optimisation -O3 and also optimisations `unroll-loops` and `unit-at-a-time` for performance.

A software framework reads the text, t, from a specified file. Each pattern, p, needed in the test is a substring of t of a designated length that starts at an index whose value is determined by a pseudo-random number generator. An arbitrary string matching function for a given t and p can be plugged into the framework, as well as routines to set up forward and backward shift tables based on p. The framework allows for specifying the number runs to take over the same data, for specifying the number of random patterns of a given length to generate and for specifying a range of different pattern lengths to use.

The framework was configured to always execute 5 runs over the same data and to generate 30 randomly selected patterns for each specified length. This configuration is in line with findings reported in [7] about appropriate statistical sample sizes and appropriate times to repeat a run over given data. It was additionally configured to generate patterns of length $m = 1, \ldots, 32, 256, 1024$.

C-coded versions of both QS and of QLQS (slightly altered for display purposes in Figure 2) were provided. In the case of QLQS, runs were executed for all z values in the range $[m, 2m+3)$. This provides data about scenarios described in the last two columns of Table 1, headed "$\in [m + 2, 2m + 2)$" and "$> 2m + 1$" respectively. It also provides for limited data about scenarios described in the preceding two columns of the table, namely those headed "$= 1$" and "$\in [2, m + 2)$". Although the boundary cases (i.e. when $z = m$ and $z = m + 1$) are always covered, data for $z \in [1, m)$ is only available for limited values of m.

We realised *ex post facto* that more complete data for $z \in [1, m)$ over *all* pattern lengths could also be of interest (albeit somewhat marginal) to investigate empirically how frequently QLQS selects shifts smaller than QS for such z values. Such data will be included in future investigations. Of course, there is no practical value in empirically investigating QLQS behaviour when $z < 1$ since that will lead to algorithmic errors — as indicated in the first two columns of Table 1.

As text data sources, the Bible file (approximately 4MB) and the Ecoli file (approximately 4MB) from the SMART corpus were used [5] were used. The distribution of alphabet symbols over the indices of the text from the latter file is conjectured to be random and so may be characterised as a random text. We shall refer to it as t_s to indicate that its alphabet is relatively small. The text based on the Bible file could be said to approximate randomness with respect to English text, but not with respect to the distribution of its alphabet symbols of text index positions. Since it contains 63 different symbols — roughly eight times more than the Ecoli file — it serves to represent QLQS behaviour over a (relatively) large alphabet and so we refer to it as t_ℓ. For convenience, the shift tables needed for QS and QLQS were implemented as arrays of size 256 for both these texts. It is conjectured that the speed and space implications for these overly large tables are negligible.

In addition to the foregoing arrangements for measuring "random" behaviour, the behaviour of QLQS and QS on best case data was also measured. Such data for both algorithms is easily constructed by using disjoint alphabets for p and t. Here a 4MB text was used and pattern lengths ranged over $m \in [1, 257)$. The theoretical best case performance for QLQS, is when $z = 2m + 1$. However, as a sanity check, times were taken for all z values in the range $[1, 2m + 2)$.

All timing data is gathered in nanoseconds but for reporting purposes these times are converted into milliseconds. In all our reporting we use the *minimum time* over the 5 runs on the same data item to eliminate possible outlier timings caused by unscheduled operating system effects.

The timing for these runs *excludes* precomputational time required to set up the shift tables — in contrast to benchmarking frameworks such as SMART. In practical use-cases, such as network security, antivirus, etc., the precomputation of the shift tables is done once (per keyword, often offline on a server), while the string matching is conducted repeatedly over large (sometimes unending) input strings. As a result, the time required is quickly overwhelmed by the string processing time for a large input string.

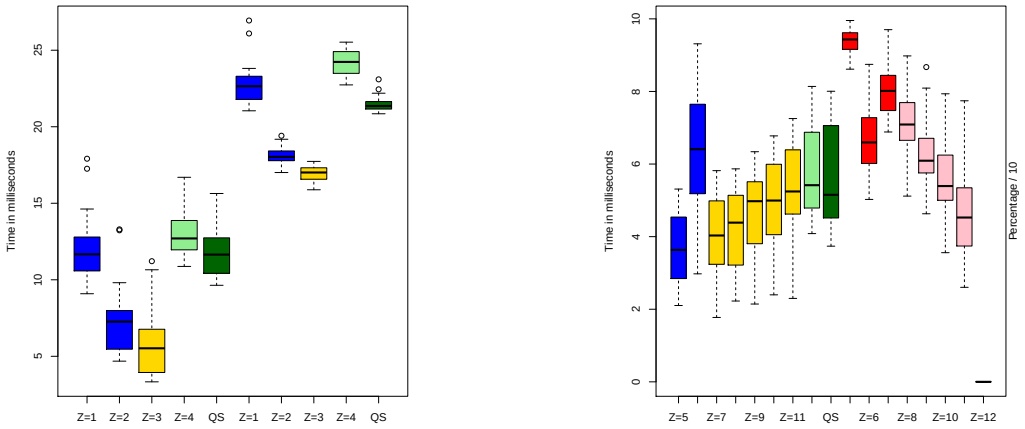
3.2 Outcomes: The Broad Picture

The subfigures of Figures 6 and 7 have been selected to illustrate one or more representative features of the data. Each subfigure contains, for a specified pattern length, several box-and-whisker plots. Such a plot indicates the median, quartiles and outlier regions in relation to measurements (in y-axis units) over a sample. Generally these measurements pertain to a sample of QLQS data for the z value given on the x-axis. Where the measurements relate instead to QS, this is also indicated on the x-axis. In the present instance, the sample is the set of 30 randomly generated patterns of the length under consideration in the subfigure. These lengths are $m = 1, 5, 1024$ and 1024 for Subfigures 6a, 6b, 7a and 7b respectively.

The plots in Subfigure 6a on the left hand side refer to time performance of QLQS and QS for runs over t_ℓ. The same plots are given on the right hand side for runs over t_s. In each case $m = 1$ and $z = 1, \ldots, 4$. Subfigures 6b and 7a give similar plots for time performance for QLQS and QS, but for $m = 5$ and $m = 1024$ respectively. Subfigures 6b and 7b incorporate plots for data described in the captions as %QLQS shifts. By this is meant the percentage of all shifts in a run to the window $t[i + z]$ instead of the conventional QS shift to $t[i + shf]$. (Note that the

data in Subfigure 6b has been scaled by a factor of 10 to keep within the range of the performance data in the same subfigure.

As a visual aid, plots of particular interest are coloured according to the following guidelines: plots relating to time performance are in blue, gold or green and plots relating to %QLQS shift data are in red or pink. The blue and red plots relate to z values that merit particular attention. Plots for $z = 2m + 2$ are coloured in light green and QS plots are coloured in dark green. The plots in Subfigure 6a are for



(a) t_ℓ vs t_s times
$m = 1$ and $z = 1, \ldots, 4$

(b) Time compare to % QLQS shifts
$m = 5$ and $z = 5, \ldots, 12$

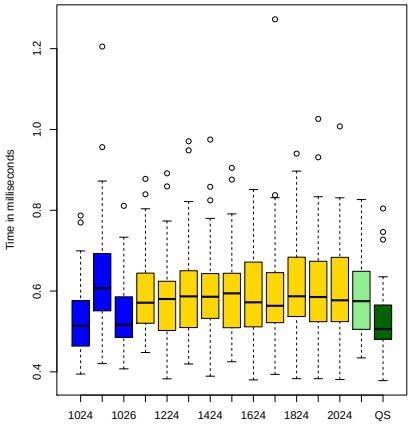
Figure 6: QLQS performance for smaller patterns

patterns of length 1. There are undoubtedly more efficient single character search algorithms than these degenerate instances of QS and QLQS. Nevertheless, the data usefully illustrates a number of relevant broader trends and issues. Median times for t_ℓ range between about 50 and 130 milliseconds compared to a range of about 160 to 240 milliseconds for t_s. Also, t_s plots show data spread over shorter ranges than corresponding t_ℓ plots. These differences in behaviour in respect of strings from small and large alphabets is consistent with other string matching algorithms and can be explained from first principles. However, the subfigure also illustrates that, despite these differences, trends in the data from t_s to t_ℓ are similar. For example, the ranking of the medians for $z = 1, \ldots 4$ and QS is exactly the same for the t_s and t_ℓ data. Comparison of the other t_s and t_ℓ data confirmed this broad correspondence. Consequently, nothing of interest is missed by limiting further discussion here to the t_ℓ-derived data.

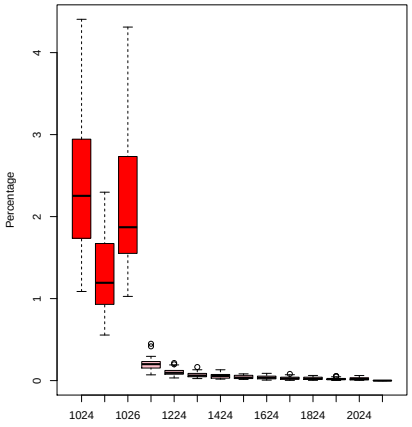
Subfigure 6a shows that QLQS significantly improves on the speed of QS for $z = 2$ and 3, more than doubling it for $z = 3$. Its speed is more or less the same as QS for $z = 1$ and worse for $z = 4$. This is consistent with the following generalisation about data from all pattern lengths:

> QLQS's *best performance over all z values* tends to be better than QS for relatively small m but that advantage is eventually lost as m increases.

By the time $m = 1024$, Subfigure 7a confirms that QS outperforms QLQS for all values of z. (At $m = 32$, QLQS outperforms QS for several z values. This data is not shown here.)



(a) Time for $m = 1024$, $z = 1024, 1025, 1026, 1124, \ldots, 2024, 2050$



(b) % QLQS shifts for $m = 1024$, $z = 1024, 1025, 1026, 1124, \ldots, 2024, 2050$

Figure 7: QLQS performance for larger patterns

Visual inspection of the subfigures that the median of the light green plot is always greater than that of the dark green plot. This points to a general feature that is evident in all the data, namely that QS always outperforms QLQS when $z = m+2$. Of course, this is to be expected because, as observed in the last column of Table 1, when $z > m + 1$ no QLQS shifts are made. Thus the QLQS algorithm behaves just as QS, but incurs additional computational complexity.

The %QLQS plots in Subfigures 6b and 7b show explicitly that there are no QLQS shifts when $z = 2m + 2$. Furthermore, both these subfigures show that as z increases, the probability diminishes of doing a %QLQS shift. There is one exception to this trend and that is when $z = m+1$. Once again the subfigures are typical of all pattern sizes. The general trend is explicable. As z increases it becomes less and less likely to comply with predicate (2), i.e. large values of z are less likely to be smaller than $shf + shb$, and therefore less likely to be selected for the next shift.

When $z = m+1$, there is a significant dip in %QLQS shifts, and there is an accompanying worsening of QLQS time performance. This peculiar behaviour is manifested in all the data for QLQS when $m \geq 4$. It can be seen in the performance plots in Subfigures 6b and 7a as well as in the %QLQS plots in Subfigures 6b and 7b.

To explain the dip in %QLQS shifts, note that when $z = m + 1$ then the same character, say c, in t is being used to index into the forward and backward shift tables and retrieve a value for shf and shb. If c does not occur in p then $shf = shb = m + 1$ so that $z < shf + shb = 2m + 2$. Since predicate (2) holds, z will be used as the offset for the next move.

Suppose c occurs one or more times in p. By definition shf is the length of the suffix in p that begins at the leftmost occurrence of c and shb the length of the prefix of p that has the leftmost occurrence of c as its last element. The maximal value of $shf + shb = m + 1$ and occurs when there is only one instance of c in p. In this case predicate (2) is not satisfied (equality holds) and so the offset used for the next move is shf.

We conjecture that the dip in time performance relates to the way in which the operating system and compiler handle the aliasing that arises in line 9 of

the code in Figure 2 — the same location in text vector `T` is referenced by two different index expressions.

The foregoing means that all the plots for %QLQS shifts when $z = m + 1$ reflects the percentage of times that a character does not appear in the pattern but appears just to the right of a window used in the text.

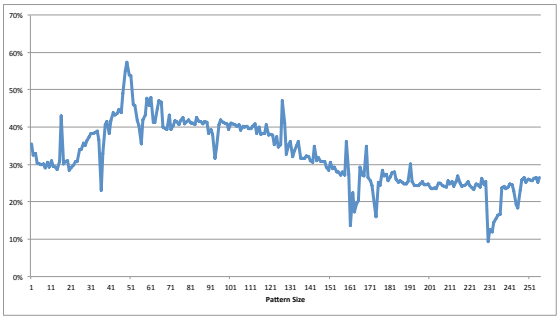


Figure 8: Best Case Performance Ratios: $\frac{QLQS}{QS}\%$ for $m \in [1, 257)$

Figure 8 graphs the performance ratio of QLQS time divided by QS time under the best case data previously outlined — disjoint pattern and text alphabets and $z = 2m + 1$. The average and standard deviation of these observations is 23% and 9% respectively, with maximum and minimum values of about 58% and 9%. Thus, QLQS on average performs at about one third of QS's speed over all pattern lengths tested.

4 Conclusion

We have presented a new algorithm for single-keyword string pattern matching. The algorithm has a number of interesting properties:

- It outperforms Sunday's QS algorithm in most cases with an appropriate choice of z.
- QLQS significantly outperforms QS when the pattern consists of letters not appearing in the input text. In the best case, the two subalphabets are disjoint and the QLQS double's QS's performance, making half the number of match attempts.
- While large z choices appear to violate the principle that safe shifts larger than $m + 1$ are not possible, QLQS in fact makes the same number of table lookups as QS — though uses them considerably more efficiently thanks to instruction-level parallelism.
- Significant instruction-level parallelism is used by modern processors (in this case Intel i7) to enable simultaneous shift lookups and simple arithmetic.
- The algorithm structure is as simple as Sunday's QS, and considerably simpler than many similar recent algorithms.
- The shift tables are easily computed and closely related to Sunday's QS.
- This appears to be the first left to right algorithm using a backward shift distance.
- QLQS is an example of a *speculative execution* (take a Quantum Leap/shift, then check if it was valid) algorithm.

There are several possible enhancements to this algorithm, as well as other areas to use the Quantum Leap principle:

- Simplify QLQS to be a probabilistic algorithm in which the validity of a z shift is not checked. Measure such an algorithm over a range of z to determine the probability of missed matches.
- Explore opportunities for coarse-grained parallelism in this style of algorithm.
- Benchmark QLQS using two dimensional shift tables (as opposed to two one dimensional tables).
- Characterize the performance of QLQS on processors unable to use instruction-level parallelism or with vastly different cache memory sizes (compared to the i7).
- Apply the Quantum Leap principle to Boyer-Moore style algorithms in other pattern matching areas such as multiple-keyword, regular expression, tree, and multi-dimensional pattern matching.
- The shift tables used in QLQS should be formally derived in a correctness-by-construction algorithm formalism.

References

1. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics*, in Proceedings of the Prague Stringology Conference 2013, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2013, pp. 92–106.
2. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's College Publications, 2004.
3. 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.
4. M. A. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific Publishing Company, 2003.
5. S. Faro and T. Lecroq: *2001–2010: Ten years of exact string matching algorithms*, in Proceedings of the Prague Stringology Conference 2011, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2011, pp. 1–2.
6. A. Hume and D. Sunday: *Fast string searching.* Software — Practice & Experience, 21(11) 1991, pp. 1221–1248.
7. D. G. Kourie, B. W. Watson, T. Strauss, L. Cleophas, and M. Mauch: *Empirically assessing algorithm performance*, in Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014, SAICSIT '14, New York, NY, USA, 2014, ACM, pp. 115–125.
8. W. F. Smyth: *Computing Patterns in Strings*, Addison-Wesley, 2003.
9. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) Aug. 1990, pp. 132–142.
10. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in IWOCA, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.

Parameterized Matching: Solutions and Extensions

Juan Mendivelso[1] and Yoan Pinzón[2]

[1] Fundación Universitaria Konrad Lorenz, Bogotá, Colombia
juanc.mendivelsom@konradlorenz.edu.co
[2] Universidad Nacional de Colombia, Bogotá, Colombia
ypinzon@unal.edu.co

Abstract. Parameterized matching is a string searching variant in which two equal-length strings parameterized-match if there exists a bijective function g for which every text symbol in one string is equal to the image under g of the corresponding symbol in the other string. Baker was the first researcher to have addressed this problem [15], and many others since have followed Baker's work. She did, indeed, open up a wide field of extensive research. Over the years, other lines of research that have been pursued are: parameterized matching under edit and Hamming distance, multiple parameterized matching, 2-dimensional parameterized matching, structural matching and function matching. This accelerated research could only be justified by the usefulness of its practical applications such as in software maintenance, image processing and bioinformatics. In this paper, we present an overview of the most notable contributions in this area.

Keywords: parameterized matching, string matching, software maintenance, function matching

1 Introduction

String searching is inarguably one of the foremost computational primitives [9]. The input to the *string matching problem* consists of two strings: the pattern $P = P_{1...m}$ and the text $T = T_{1...n}$. The output should list all the occurrences of the pattern in the text. Given that, for specific applications, it is useful to find inexact occurrences of the pattern, many variants of the string matching problem have been proposed. One of these variants is *parameterized matching*. Two equal-length strings *parameterized-match* if there exists a bijective function g for which every text symbol in one string is equal to the image under g of the corresponding symbol in the other string. The symbols in the strings are drawn from two alphabets: the constant alphabet Σ_C and the parameter alphabet Σ_P. The mapping of the symbols from Σ_C must be identity.

More formally, two length–m strings $X = X_{1...m}$ and $Y = Y_{1...m}$, defined over $(\Sigma_C \cup \Sigma_P)^*$, are said to be a *parameterized–match*, or a *p–match*, if there exists a bijective function $g : \Sigma_C \cup \Sigma_P \mapsto \Sigma_C \cup \Sigma_P$ such that $g(Y_i) = X_i,\ 1 \leq i \leq m$ so that g is identity for the the symbols from Σ_C. Note that, in the worst case, g can be chosen from $|\Sigma_P|!$ different possible mapping functions. For instance, given the constant alphabet $\Sigma_C = \{b\}$ and the parameter alphabet $\Sigma_P = \{x, y, z\}$, let us consider two strings $X = xbyyxbx$ and $Y = zbxxzbz$ defined over $\Sigma_C \cup \Sigma_P$. We can say that X and Y are a parameterized–match given that we can map Y to $Y' = xbyyxbx = X$ by means of the bijective function $r : (b, x, y, z) \mapsto (b, y, z, x)$. Notice that the mapping of the symbol b, the only symbol from the constant alphabet in this example, is identity.

Furthermore, two equal-length strings X and Y that parameterized-match have the same structure. Let us suppose that i and j are the only occurrences of the symbol α in Y. Then, the existence of a bijective function g that maps the symbols in Y to the

Juan Mendivelso, Yoan Pinzón: Parameterized Matching: Solutions and Extensions, pp. 118–131.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

symbols in X implies that $g(\alpha) = X_i = X_j = \beta$ and that β has no other occurrences in X. As this applies for all the distinct symbols α in Y, we can conclude that the following facts hold: (i) X and Y have the same number of distinct symbols; (ii) the first occurrence of each distinct symbol α in Y takes place in the same position of the first occurrence of the symbol $g(\alpha)$ in X; and (iii) the relative distances among the different occurrences of each α in Y are the same relative distances among the occurrences of $g(\alpha)$ in X. Therefore, two strings that parameterized-match are the same except for a systematic change of the symbols.

In this sense, parameterized matching has important applications in different areas. However, it was initially defined as a tool for software maintenance. This was motivated by the observation that programmers introduce duplicate code into large software systems when they add new features or fix bugs. Instead of adapting working sections of code, programmers prefer to copy and slightly modify new instances of those sections in order to avoid making major revisions and introducing new bugs. They do it especially when the working sections were written by another programmer. Then, the code is considered as a sequence of tokens (variables, constants, operands, reserved keywords and procedure names) where the constant alphabet Σ_C is comprised by the operands and the reserved keywords while the parameter alphabet Σ_P is comprised by the variables, constants and procedures' names [13].

With time, the amount of duplicate code is highly increased and the code gets larger, more complex and more difficult to maintain. For instance, when a new issue in a determined part of the program is fixed, it will not be automatically fixed in the other copies of that section of code and sometimes they may be hard to find. Experimental results on a large subsystem of over a million lines of code showed that 22% of the lines was involved in parameterized matching [13]. This is a great amount of duplicate code, given that a proportional percentage of the code could be shrunk by using better programming techniques like procedures and functions. A reduction of this magnitude would make the code much more simple and easier to maintain.

In this paper, we review, organize and summarize some of the most important works on parameterized matching. The outline of the article is as follows. The definitions of the different parameterized matching problems are presented in Section 2. The solutions of parameterized matching are reviewed in Section 3 and its extensions are presented in Section 4. Finally, some of the most important applications of this pattern matching variant are shown in Section 5 and the conclusions are drawn in Section 6.

2 Basic Problems

A *parameterized string* or a *p–string* is defined as a string of symbols in $(\Sigma_C \cup \Sigma_P)^*$. Then, two length–$m$ p–strings $X = X_{1...m}$ and $Y = Y_{1...m}$ are said to be a *parameterized–match* or a *p–match*, if one p–string can be transformed into the other by bijectively renaming its parameters. The basic parameterized matching problem is *parameterized pattern matching*. It consists of finding all the parameterized–matches of a pattern $P = P_{1...m}$ in a text $T = T_{1...n}$. Note that, at each position i of T, a different g_i can be considered to determine the existence of a parameterized-match between the pattern and the text window starting at position i. This problem is also referred as *Parameterized Fixed Pattern Matching (PFPM)* [54].

Some other problems related to parameterized matching have been defined to be able to support more applications. One of them is finding the maximal p–matches of

a p–string text $T = T_{1...n}$ over a threshold length t. A maximal p–match is a p–match between a pair of p-substrings (of T) that is neither *left-extensible* nor *right-extensible*. Let us consider a p–match between $T_{i...i+k}$ and $T_{j...j+k}$. This p–match is said to be *left–extensible* if $T_{i-1...i+k}$ and $T_{j-1...j+k}$ are a p–match and is *right–extensible* if $T_{i...i+k+1}$ and $T_{j...j+k+1}$ are a p–match, where $1 \leq i \leq i+k \leq n$, $1 \leq j \leq j+k \leq n$ and $i \neq j$. Note that the length of the strings in a p–match is $k+1$. Because the output of the problem is the set of maximal p–matches whose length is at least t, then $k+1 \geq t$ for each match.

On the other hand, parameterized matching has been extended to search for multiple parameterized patterns [54]. For a given fixed set D of p–string patterns over $\Sigma_C \cup \Sigma_P$, the *Parameterized Multiple Pattern Matching (PMPM)* problem consists of preprocessing D as an aid to later determine the p–matches (for all of the patterns in D) in a query text T. A dynamic variant of this problem, called *Parameterized Dynamic Dictionary Matching (PDDM)*, has also been considered [54]. In this problem, a dictionary D of p–string patterns is preprocessed and maintained with available operations of inserting/deleting patterns into/from D and searching a query text T for p–matches for the patterns currently in D.

3 Solutions

The problem of finding the maximal p–matches of a p–string text over a threshold length was the first parameterized matching problem to ever be considered. Baker tackled this problem motivated by the observation that there was a considerable amount of duplicate code in large software systems. Therefore, she presented a program, called DUP [13], as an aid to find all the duplicate sections of code with a minimum length specified by the user. DUP simplifies the problem to an exact matching problem by replacing all the parameters with a determined symbol and then looks for the p–matches among the exact matches found. The algorithm is based on recursions over the suffix tree of the text. Experiments with real data showed that the algorithm is inefficient given that just a few of the exact matches found correspond to p–matches. For this reason, the same author proposed a more elaborate theory [15,17].

A procedure called *prev* was defined to yield efficient solutions for parameterized matching [17]. Given a length–m p–string $X = X_{1...m}$ defined over $\Sigma_C \cup \Sigma_P$, $prev(X)$ is a string in $(\Sigma_C \cup \mathbb{N})^*$ where every constant symbol in X remains the same in $prev(X)$ but the parameters are replaced by non-negative integers: the leftmost occurrence of a determined parameter is represented by a 0 and the other occurrences are represented by the difference in position compared to the previous occurrence of this parameter. The numbers that represent difference in position are called *parameter pointers*. The time complexity of the computation of *prev* is $O(m)$ and the space complexity is $O(|\Sigma_P|)$ by means of a table containing the last occurrence position of each parameter.

Notice that $prev(X)$ is calculated in such a way that it does not matter what the parameters of X are; what is really relevant is the relative distance among the different occurrences of the same parameter (represented by the parameter pointers) which provides valuable information about the structure of the p–string. Thus, two p–strings X and Y are a p–match, iff $prev(X) = prev(Y)$. For example, given $\Sigma_C = \{b\}$, $\Sigma_P = \{x, y, z\}$, $X = xbyyxbx$ and $Y = zbxxzbz$, we find that $prev(X) = 0b014b2 = prev(Y)$ and therefore X and Y are a p–match. The *prev* of any substring of a p–string

X can be calculated from $prev(X)$. This is because any symbol of the substring is the same as in $prev(X)$ except when it is a parameter pointer that points to a position before i; in such case, it will correspond to the first occurrence of the parameter in the substring so it must replaced by a 0.

Reminiscing about the use of suffix trees for exact matches in DUP, Baker defined a new data structure called *parameterized–suffix tree*, or *p–suffix tree*, to aid in directly searching for parameterized–matches [17]. The p–suffix tree of a p–string $X = X_{1...m}$ is a compacted trie that stores the *p–suffices* of X. The i–th *p–suffix* of a X is defined as $\text{psuffix}(X, i) = prev(X_{i...m})$, for $1 \leq i \leq m$ [17]. So we can calculate each p–suffix, just like the *prev* of any substring of X, by copying the corresponding symbols of $prev(X)$ except when they are parameter pointers that point to a symbol outside the substring (in which case they are replaced by 0).

An algorithm to construct p–suffix trees, called LAZY, was proposed [17]. It is based on McCreight's algorithm for constructing suffix trees [65]. This algorithm is linear in the p–string length in both time and space for fixed alphabets. For variable alphabets, the time complexity is $O(n(|\Sigma_P| log(|\Sigma_C| + |\Sigma_P|)))$. Later, Baker proposed a new algorithm to build p–suffix trees, called EAGER, that improved the time complexity for variable alphabets to $O(n(|\Sigma_P| + log(|\Sigma_C| + |\Sigma_P|)))$ [15]. The time complexity of both LAZY and EAGER can be reduced to $O(n \log n)$, for the variable alphabet case, by using auxiliary data structures like concatenable queues [2] and Sleator-Tarjan dynamic trees [75]. However, the use of these structures makes the algorithms not practical. Then, Kosaraju proposed an algorithm whose time complexity is $O(n\, log(|\Sigma_P| + |\Sigma_C|))$ [58]. Other authors devised randomized algorithms [36,60,61].

Two solutions for the parameterized matching problem that use p–suffix trees were developed [15]. Let us consider the pattern p–string $P = P_{1...m}$ and the text p–string $T = T_{1...n}$. One of the algorithms consists of following the path determined by the symbols of $prev(P)$ on the p–suffix tree of T to find out if $prev(P)$ is identical to a length–m substring of T. Retrieving all the positions in T where there is a p–match with P, for fixed alphabets, takes $O(m + occ)$ time and $O(n)$ space, where *occ* is the number of p–matches. The time complexity of this operation is $O(m\, log(|\Sigma_C| + |\Sigma_P|) + occ)$ for variable alphabets. The other algorithm consists of searching in a p–suffix tree for P through an adaptation of the corresponding algorithm for strings [35]. Its space complexity is $O(m)$ and its time complexity is $O(n)$ for fixed alphabets; for variable alphabets, its time complexity is $O(n\,(|\Sigma_P| + log(|\Sigma_C| + |\Sigma_P|)))$. Nevertheless, it could also be improved to $O(n\, log(|\Sigma_C| + |\Sigma_P|))$ by using some auxiliary data structures for computing lowest common ancestors [47,73].

On the other hand, an algorithm, called PDUP, for finding the maximal p–matches over a threshold length of a text $T = T_{1...n}$ was devised [17]. PDUP is similar to DUP, but constructs a p–suffix tree of the text instead of a suffix tree. This algorithm generalizes to p–strings the algorithm for finding maximal p–matches over a threshold length in a string [14]. In this generalization, it is necessary to augment the p–suffix tree with lists that store data that makes possible to determine whether there is left–extensibility in the p–matching substrings. The time complexity of PDUP is $O(n + occ)$ even for variable alphabets. The efficiency of this algorithm to detect duplicate code was evaluated through an experiment in [19].

In order to improve the memory usage and access locality provided by p–suffix trees, *parameterized suffix arrays*, or *p–suffix arrays*, were defined [41]. Specifically, p–suffix arrays are defined with respect to p–suffix trees in an analogous manner as traditional suffix arrays are defined with respect to suffix trees. It is well-known

that the combination of suffix arrays and the *longest common prefix* (LCP) yields an efficient solution for traditional string matching [64,46,76,1]. This also applies for p–strings [26]. Then, most of the operations on a p–suffix tree can be simulated with the use of the corresponding p–suffix array and an array that contains the lengths of the longest common prefixes of the p–suffixes. The latter, called *parameterized longest common preffix array*, is denoted as *p–LCP*. Thus, parameterized pattern matching can be solved using p–suffix arrays and p–LCP by means of a binary search in $O(m + \log n + occ)$ [41].

The construction of a p–suffix array can be achieved by traversing the corresponding p–suffix tree. Deguchi et.al. were the first to directly construct p–suffix arrays, i.e. without constructing the p–suffix tree [41]. Specifically, they provided a linear algorithm to construct p-suffix arrays and p–LCP for binary alphabets. Then, I et.al. proposed an algorithm for constructing p–suffix arrays and p–LCP for non-binary alphabets [51]. Moreover, they were the first to consider the *p–suffix sorting problem*, which consists of sorting all the p–suffixes of a p–string in lexicographic order. It is important to remark that traditional suffix sorting techniques cannot be applied to p–suffixes because of their dynamic nature. In particular, the first two algorithms that addressed p–suffix sorting are based in QuickSort and RadixSort; they take $O(n^3)$ and $O(n^2)$, respectively [51]. Later, Beal and Adjeroh devised a solution that generates and lexicographically sorts fingerprints and arithmetic codes that correspond to the p–suffixes [22,25]. This algorithm has expected linear time and $o(n^2)$ worst-case time complexity, which improves the complexity given in [51].

However, the definition of p–suffix arrays took place in recent years. Chronologically, Baker's solution based on p–suffix trees was followed by other works. For instance, Amir et.al. defined a related model called *mapped matching* which is a special case of parameterized matching where all symbols are in the parameter alphabet [5]. Through this model, an algorithm that extends the KMP algorithm [57] to parameterized matching, and runs in $O(n \log \min(m, |\Sigma_P|))$ time, was proposed [5]. It was proven that the $\log \min(m, |\Sigma_P|)$ factor is inherent to any algorithm for parameterized matching in the comparison model and, consequently, that the provided algorithm is optimal. This demonstration was achieved through a reduction from the element distinctness problem to parameterized matching.

This new research may have motivated Baker to look for parameterized matching solutions based on classical exact string matching algorithms [16]. Given that the Boyer–Moore algorithm [33] is one of the most efficient, she attempted to generalize it to p–strings but found out that its worst case performance was poor. Therefore she turned to one of its variants: TurboBM [37]. Her non-trivial generalization of TurboBM to p–strings, called PtURBOBM, runs in $O(n \log \min(m, |\Sigma_P|)))$ time and $O(n)$ space; the preprocessing time is $O(m \log \min(m, |\Sigma_P|))$. Its time complexity is the same as the generalization of KMP complexity so it is optimal [5]. Nevertheless some experiments show that PtURBOBM works better for long patterns over different alphabet sizes. Anyhow, for variable alphabets, both of these algorithms are notably better than then p–suffix tree based parameterized matching algorithms.

Other important contributions were made by Idury and Schäffer who proposed some variants of the basic problem (see Section 2) and solutions for all of them [54]. For the Parameterized Multiple Pattern Matching Problem, they proposed an algorithm that uses a modified Aho–Corasick automaton and runs in $O(n \log(|\Sigma_C| + |\Sigma_C|) + occ)$ time. As for the Parameterized Dynamic Dictionary Problem, they devised an automaton algorithm that supports different operations with the following

time complexity: (i) $O((n+occ)(log(|\Sigma_C|+|\Sigma_P|)+log\, d)$ for searching the p–string patterns of the dictionary in a p–string text $T = T_{1...n}$; (ii) $O(m(log(|\Sigma_C|+|\Sigma_P|))+log^2\, d)$ for inserting a new pattern $P = P_{1...m}$ into the dictionary; and (iii) $O(m(log(|\Sigma_C| + |\Sigma_P|)) + log\, d)$ for deleting a pattern $P = P_{1...m}$ from the dictionary, where d is the total size of all the patterns.

The adaptation of the Aho–Corasick algorithm to p–strings proposed by Idury and Schäffer leads to the definition of *parameterized border arrays*, or *p–border arrays*, which constitute the parameterized version of traditional *border arrays*. In particular, this adaptation modifies the *goto* and *fail* functions with their respective parameterized versions: *pgoto* and *pfail*. When only a single pattern is considered, the *pfail* function can be implemented by a p–border array. A p-border array can be computed in linear time, as presented in [54]. In more recent works, I et.al. proposed three related algorithms for the binary alphabet case [53,52]: (i) a linear time algorithm to verify if an integer array is a valid p–border array; (ii) a linear time algorithm to compute all the p–strings that share a given p–border array; and (iii) an algorithm that computes all the p–border arrays shorter than a given threshold length. The latter is linear in the number of p–border arrays reported.

The same authors proposed an algorithm to verify if a length-n integer array is a valid p–border array for the case of unbounded alphabets [53,52]. Its time complexity is $O(n^{1.5})$ and its space complexity is $O(n)$. This algorithm is more efficient than the previous solution, which takes time proportional to the n–th Bell number $\frac{1}{e}\sum_{k=0}^{\infty}\frac{k^n}{k!}$. Furthermore, it is shown that the enumeration of all p–border arrays shorter than a threshold length n can be performed in $O(B^n n^{2.5})$ where B^n denotes the number of length-n p–border arrays [53].

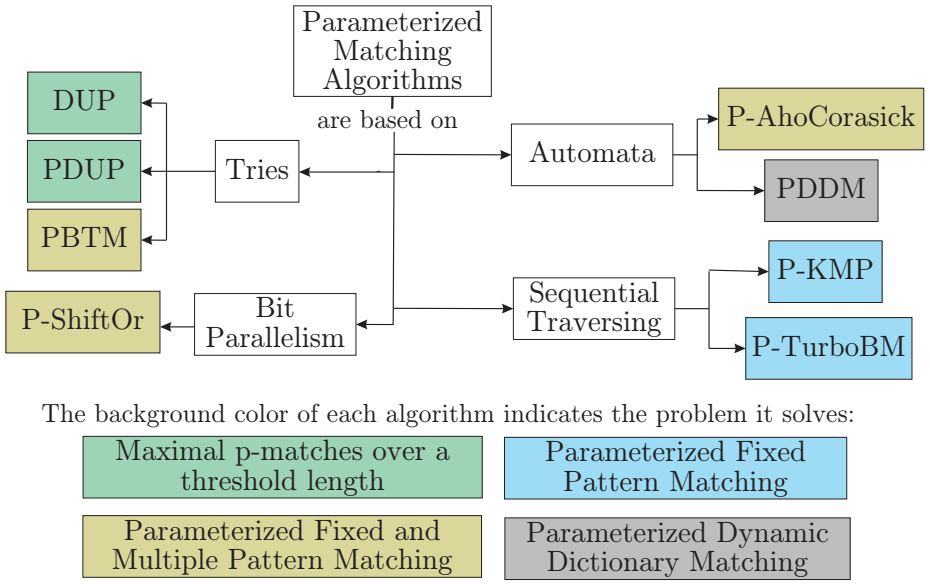


Figure 1. Concept map of the algorithms for solving the main parameterized matching problems organized by the nature of their approaches.

On the other hand, Fredriksson and Mozgovoy proposed two new solutions for both the single and multiple parameterized matching problems [44]. Both of them make use of Baker's lemma to compute the *prev* of a text substring through the *prev* of the container p–string [17]. One of them is a bit–parallelism based algorithm called P–Shift–Or. It is a generalization of the Shift–Or algorithm [11] to p–strings and

runs in $O(n\lceil m/w\rceil)$ worst-case time and $O(n)$ average time. The other solution, called Parameterized Backward Trie Matching (PBTM) [44], is based on the Backward DAWG Matching (BDM) algorithm [32,37]. The average time complexity of PBTM is $O(n\,log(m)/m)$. This algorithm could also make use of a suffix array [64] instead of a trie, in which case it is called Parameterized Backward Array Matching (PBAM). These parameterized matching algorithms are the first for which an average time complexity analysis has been made. They have optimal average-case running time as confirmed by experimental results. Other algorithms with sublinear average–case complexity were proposed in [72,71]; they are based on the BoyerMoore algorithm.

The diagram in Figure 1 shows the algorithms for solving the different parameterized matching problems organized by the nature of their approaches.

4 Extensions

Parameterized matching has been studied in many directions. For instance, it has a close relation with palindromes. It was shown that two strings drawn from an alphabet of size at most 3 have the same set of maximal palindromes if and only if they are a p–match [77]. On the other hand, an investigation about the periodicity of parameterized strings was done [10]. They attempted to generalize to p–strings two of the periodicity lemmas of strings: the Lyndon and Schitzenberger lemma (referred as *Weak Version*) [62], and the Fine and Wilf lemma [43]. They found out that only the Weak Version holds for p–strings only when the two mappings inducing the periodicity commute. These results and some other studies about the repetitions in p–strings showed considerable differences between p–strings and ordinary strings. Nevertheless, binary p–strings behave in a very similar way as ordinary strings with respect to periodicity and repetitions.

Furthermore, parameterized matching was extended to the two dimensional case by considering matrices of symbols instead of p–strings. *Two–dimensional parameterized matching* consists of finding all the p–matches of a pattern of size $m \times m$ in a text of size $n \times n$. An algorithm for the problem that runs in $O(n^2 + m^{2.5}\,polylog\,m)$ time was proposed [48]. Other solutions include a $O(n^2\,log^2 m)$ deterministic algorithm and a $O(n^2\,log\,n)$ randomized algorithm that reports all the p–matches [4]. Nevertheless, it may report a mismatch as match with probability of $1/n^k$, where k is a given constant.

Other topic that arose as a matter of interest was the calculation of similarity between two p–strings. In particular, Baker defined the *parameterized edit distance* or *p–edit distance* of two p–strings as the cost of a minimal edit script, called *p–edit script*, that transforms one p–string into the other [18]. The valid operations are insertions, deletions and parameterized replacements (the replacement of a substring with a p–string that p–matches it). Moreover, Baker proposed an algorithm [18] for calculating the p–edit distance D of two *prev*–encoded p–strings, $X = X_{1...m}$ and $Y = Y_{1...n}$, by generalizing Myers's algorithm for finding the LCS of two strings [70]. The algorithm runs in $O(D\,(n+m))$ time and $O(n+m)$ space. Furthermore, a divide-and-conquer based algorithm for reporting the minimal p–edit script was proposed [18]. It also runs in $O(D\,(n+m))$ and $O(n+m)$ space.

There have been some works about *approximate parameterized problem under hamming distance*. In particular, the π–match between two p–strings $X = X_{1...m}$ and $Y = Y_{1...m}$ was defined as the number of matches between $\pi(Y_i)$ and X_i, for $1 \leq i \leq m$ [8]. For two equal–length p–strings, the *approximate parameterized matching*

problem, also called *parameterized matching with mismatches*, consists of finding a π of maximal π–match. Given a p–string pattern $P = P_{1...m}$ and a p–string text $T = T_{1...n}$, the *approximate parameterized searching problem under hamming distance* consists of computing the approximate parameterized matching between P and every length–m p–substring of T. It is not necessary to choose the same π for every text window. Furthermore, a linear algorithm to solve this problem, for the case where both P and T are run–length encoded and one of them is a binary p–string, was devised [8].

Further studies about parameterized matching and hamming distance have been developed [49,48]. Specifically, a related problem, called *parameterized matching with a threshold of k mismatches*, was proposed. Its goal is finding all the p–matches of a pattern $P = P_{1...m}$ in a text $T = T_{1...n}$ with at most k mismatches. Furthermore, for two p–strings $X = X_{1...m}$ and $Y = Y_{1...m}$, they proposed a $O(m + k^{1.5})$ time algorithm and a $O(m^{1.5})$ time algorithm for the cases when k is and is not considered, respectively. These solutions are based on maximum matching algorithms; furthermore, it was demonstrated that the maximum matching problem is reducible to the approximate parameterized matching problem. For a p–string pattern $P = P_{1...m}$, a p–string text $T = T_{1...n}$ and a given k, a $O(nk^{1.5} + mk \log m)$ time algorithm for the parameterized matching with k mismatches problem was also proposed. It is shown that this could be extended to the two dimensional case in $O(n^2mk^{1.5} + m^2k \log m)$ time.

Another approximate version of parameterized matching is based on δ– and γ–distances. Two equal-length integer strings are said to $\delta\gamma$–match if (i) the difference between their corresponding symbols is at most δ; and (ii) the sum of such differences is at most γ. Note that constants δ and γ are bounds for the local and global errors, respectively, on the difference between the corresponding symbols of the strings. Thus, these distances are used to search for all similar but not necessarily identical occurrences of a given pattern [34]. Then, the $\delta\gamma$–approximate parameterized matching problem was defined [59]. Specifically, given two equal-length integer strings $X = X_{1...m}$ and $Y = Y_{1...m}$, string X is said to $\delta\gamma$–parameterized match string Y if X can be transformed into a string X', via a bijection π (i.e., $X'_i = \pi(X_i)$ for $1 \leq i \leq m$), such that X' $\delta\gamma$–matches Y. Moreover, a $O(nm)$ algorithm to report the $\delta\gamma$–parameterized matches of a pattern $P = P_{1...m}$ in a text $T = T_{1...n}$ was proposed [59]. In particular, this algorithm is based on a reduction to the Maximum Weight Perfect Matching problem in bipartite graphs [66].

The parameterized matching problem under the LCS distance problem has also been considered. The *longest common parameterized subsequence (LCPS)* for two p–strings $X = X_{1...m}$ and $Y = Y_{1...n}$ was defined as the pair of sequences I and J of maximum length, such that I is a subsequence of the p–string X, J is a subsequence of the p–string Y, and I and J are a p–match [56]. It is important to remark that it is not required that the symbols in I and J are consecutive in X and Y. The LCPS could be useful as a similarity measure between code sections; nevertheless, this problem has been proven to be NP–hard. Then, an approximate algorithm was proposed [56].

The longest previous factor (LPF) on traditional strings [38] has important applications in string compression [79] and for detecting runs [63]. It was extended to p–strings as the parameterized longest previous factor, or *p–LPF*, by Beal and Adjeroh [26,23]. In particular, given a p–string $T = T_{1...n}$, the p–LPF of a p–suffix in T, starting at position i, is the longest p–suffix starting at position h such that $1 \leq h < i$.

The p–LPF is useful to detect study duplication and compression in p–strings. An algorithm to compute the p–LPF in expected linear time was also proposed in [26]; it utilizes p–suffix arrays. This solution can be used to calculate the p–LCP, LCP and LPF due to the general definition of parameterized matching.

This generality was further exploited by the same authors in [27]. They proposed a taxonomy of classes for longest previous factor problems that allows them to show the relation between p–LPF and traditional data structures. Specifically, they show that the p–LCP can be used to linearly construct the p–border array and the border array, which are quite relevant for pattern matching. Moreover, the concept of permuted LCP is extended to p–strings. Also, motivated by the variants of the traditional LPF problem [39,40], they defined the counterpart versions for p–strings: *parameterized longest not-equal factor (p–LneF)*, *parameterized longest reverse factor (p–LrF)* and *parameterized longest factor (p-LF)*. The same framework of the p-LPF solution can be used to compute all these structures by changing the preprocessing and postprocessing phases. Applications of these data structures include clone detection, pattern substitution, LZ decomposition, periodicity study and biological sequence compression and analysis.

Another parameterized paradigm called *parameterized pattern queries*, that is closely related to the theory developed by Baker, was proposed [42]. They use a set of symbols and a set of variables that correspond to Baker's constant alphabet and parameter alphabet. They also defined a concept of *valuation* that could be associated with the mapping bijection and the p–match definition. This paradigm was conceived as an extension of traditional pattern expressions to enhance the querying and clustering operations over sequence databases. Thus, the definition of a set of predicates on the variables (constraints) is also permitted under this new model. Furthermore, a KMP–based algorithm for this problem is also proposed. Experimental results showed that it notably decreases the query evaluation time compared to a naive approach.

One of the most important extensions of parameterized matching is *structural matching* (or *s–matching* for short). Shibuya defined it as parameterized matching but taking into account an injective complementary relation among a subset of the parameters. This relation is used to establish an additional constrain in the matching: if parameter x is mapped to y, then the complement of x is also mapped to the complement of y in the bijection [74]. The motivation for this definition is the application for matching RNA and single-stranded DNA sequences as they contain complementary bases: adenine with uracil or thymine, and cytosine with guanine. Then, two sequences that s–match are likely to have similar structure and, therefore, similar functions [20]. The solution proposed to solve this problem involves the utilization of a *structural suffix tree*, also called *s–suffix tree*, which is a generalization of Baker's p–suffix trees. An on-line algorithm to construct the s–suffix tree in $O(n(\log|\Sigma_C| + \log|\Sigma_P|))$ is also presented in [74]. This is the first on-line algorithm that constructs p–suffix trees. It performs in linear time for RNA and DNA sequences. Moreover, it is important to remark that this was a novel approach to the problem of comparing RNA and DNA sequences; other solutions include [50,12,3,45,55,78].

Given that the practical space requirement for s–suffix trees is high, Beal and Adjeroh recently defined the *structural suffix array*, or *s–suffix array* for short, and the *structural longest common preffix array*, denoted as *s–LCP*, to solve the s–matching problem [21,31]. They exploit the flexibility of these data structures to address diverse variants of the RNA matching problem with slight modifications in the solution. The

same authors also proposed another data structure to solve the s–matching problem: the *structural border array (s–border array)* [30,24]. A linear time algorithm to construct the s–border array is also presented in [30]; it is based on special properties of the s–border data structure. Furthermore, it is shown how to modify the alphabets so that the algorithm constructs the p–border and the traditional border as well. Due to the recent interest on parameterized matching in compressed strings [8,7,28], the authors also show how to tackle parameterized matching on run-length encoded strings. Another data structure with applications in RNA matching is the *forward stem matrix (FSM)* [29]. This structure efficiently represents the length-k options, for $k \in K$, within a length-n RNA sequence; its size is $O(n|K|)$.

In order to support other applications, parameterized matching was generalized to *function matching* by allowing the mapping function to be of any type, and not just bijections as in parameterized matching [4]. In other words, many symbols of the pattern can be mapped to the same text symbol. A deterministic solution for the function matching problem, that runs in $O(n|\Sigma_P| \log m)$ time, was devised [4]. Furthermore, they proposed a Monte Carlo algorithm that runs in $O(n \log m)$ time with failure probability of $1/n^k$, where k is a given constant. Function matching was also extended for the two–dimensional case and a randomized algorithm that runs in $O(kn^2 \log n)$ time was proposed [4]. This algorithm has a $1/n^k$ probability of reporting a false positive. An approximate version of function matching based on the $\delta\gamma$- distances was developed [69]. Given two integer strings, $X = X_{1...m}$ and $Y = Y_{1...m}$, and two given constants, δ and γ, there is a match from X to Y if X can be transformed into a string X', by means of a function f, such that X' is $\delta\gamma$–equal to Y. A $O(nm)$ algorithm to find the $\delta\gamma$–function matches of a pattern $P = P_{1...m}$ in a text $T = T_{1...n}$ was proposed [69].

To support even a much wider range of applications, function matching was extended to the *generalized function matching with don't cares* problem [6]. In this problem, the image of the mapping function can be any substring in $(\Sigma_C \cup \Sigma_P)^*$ and not just a single symbol as in function matching. Furthermore, an extra symbol ϕ, called the *don't care* character, can be present in the strings. A ϕ in the text matches any pattern symbol; a ϕ in the pattern matches any text substring. This problem represents many pattern searching types but, as a consequence, it is much more complex. A polynomial algorithm for the finite alphabet case was presented; for the case of infinite alphabets, it was demonstrated that the problem is *NP–hard* [6]. This is the first problem, so far, for which there is a polynomial solution for the finite alphabet case and there is not one for the infinite alphabet case.

5 Applications

Besides its applications in software maintenance, parameterized matching is useful in image processing [49,4]. The Human–Computer Interaction Lab at the University of Maryland tackled the problem of searching for an icon in the screen. If the colors are fixed, the problem can be solved with an exact two-dimensional pattern matching algorithm. Nevertheless, sometimes the pattern image appears in other ranges of colors within the text, which makes impossible for exact–matching algorithms to find these occurrences. In this kind of cases it is proper to use two dimensional *parameterized matching* algorithms. However, images often have some errors resulting from distortion and loss of resolution, so such occurrences of a pattern image could not be reported by parameterized matching algorithms either. But occurrences with

these errors can indeed be found by taking either a function matching approach [4,69] or an approximate parameterized matching approach under the hamming, p–edit, or $\delta\gamma$ distance [18,48,49,59].

On the other hand, parameterized matching has applications in databases. For instance, in a database that contains URLs of the pages visited by different users, parameterized pattern queries can be used to retrieve useful information for improving the ergonomy of the site and finding the best places for advertisement ads [42]. For example, given the symbol a and the variable x where both represent URLs, the query of the parameterized pattern expression axa would retrieve the set of URLs that the users have visited before coming back to the previously visited page represented by a. In a similar fashion, this idea can be used in computational biology to retrieve all the amino acids substrings that follow a determined structure where the presence of determined amino acids at certain positions are a constraint. This is also applicable to databases of any type, where the analysis over the sequential occurrence of elements is a matter of interest.

In recent works, parameterized matching has been utilized as a mechanism to solve the graph isomorphism problem [68]. Graph isomorphism is the problem of determining if the topology of two graphs is the same. More formally, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijection $f : V_1 \mapsto V_2$ such that $(u, v) \in E_1$, for $u, v \in V_1$, if and only if $(f(u), f(v)) \in E_2$. Given that parameterized matching is defined for linear structures, the concept of *graph linearization* was defined to represent the topology of a graph as a walk that traverses all its nodes and edges. Then, two graphs are isomorphic if and only if there exists a walk in one of the graphs that parameterized-matches a linearization of the other graph.

Specifically, the solution for graph isomorphism under this approach has two main steps: (i) representing G_1 by means of a linearization p; and (ii) determining if there exists a walk in G_2 that parameterized-matches the linearization p [68]. For the former, an efficient linearization algorithm that generates short linearizations with an approximation guarantee was proposed; it requires $O(|E_1| + d|V_1| \lg d)$ time and $O(|E_1|)$ space, where d is the maximum node degree. For the latter, a DFS-based algorithm that prunes the search space by using vertex degrees and previous assignments was developed; it requires $O(|V_2|d^{\lfloor \ell/2 \rfloor})$ time and $O(|V_1| + |E_1|)$ space, where ℓ is the length of the linearization p. This solution was experimentally evaluated on graphs of different types and sizes. It was compared to the performance of VF2, which is a prominent algorithm for graph isomorphism. Empirical measurements show that graph linearization finds a matching graph faster than VF2 in numerous cases, especially in Miyazaki-constructed graphs which are known to be one of the hardest cases for graph isomorphism algorithms [67].

6 Conclusions

Parameterized matching is a string searching variant that allows to find strings with the same structure. Thus, it is useful in any area where patterns are defined in terms of structural correlation across the positions. Its applications in areas like software maintenance, plagiarism detection and image processing have motivated extensive research for more than two decades. In particular, different problems, solutions, extensions and properties have been studied. New insights on parameterized matching in recent research works include: (i) the definition of new data structures to yield more

efficient solutions; (ii) the generalization to s–matching as a mechanism to match RNA sequences; and (iii) its use to solve the graph isomorphism problem.

References

1. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*, Springer Science & Business Media, 2008.
2. A. V. Aho and J. E. Hopcroft: *Design & Analysis of Computer Algorithms*, Pearson Education India, 1974.
3. J. Allali and M.-F. Sagot: *A new distance for high level rna secondary structure comparison.* IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), 2(1) 2005, pp. 3–14.
4. A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat: *Function matching: Algorithms, applications, and a lower bound*, in Proceedings of the 30th International Colloquium on Automata, Languages and Programming, 2003.
5. A. Amir, M. Farach, and S. Muthukrishnan: *Alphabet dependence in parameterized matching.* Information Processing Letters, 49(3) 1994, pp. 111–115.
6. A. Amir and I. Nor: *Generalized function matching.* Journal of Discrete Algorithms, 5(3) 2007, pp. 514–523.
7. A. Apostolico, P. L. Erdős, and A. Jüttner: *Parameterized searching with mismatches for run-length encoded strings.* Theoretical Computer Science, 454 2012, pp. 23–29.
8. A. Apostolico, P. L. Erdős, and M. Lewenstein: *Parameterized matching with mismatches.* Journal of Discrete Algorithms, 5(1) 2007, pp. 135–140.
9. A. Apostolico and Z. Galil: *Pattern matching algorithms*, Oxford University Press, USA, 1997.
10. A. Apostolico and R. Giancarlo: *Periodicity and repetitions in parameterized strings.* Discrete Applied Mathematics, 156(9) 2008, pp. 1389–1398.
11. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, p. 82.
12. V. Bafna, S. Muthukrishnan, and R. Ravi: *Computing similarity between rna strings*, in Combinatorial Pattern Matching, Springer, 1995, pp. 1–16.
13. B. S. Baker: *A program for identifying duplicated code*, in Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface, 1992.
14. B. S. Baker: *On finding duplication in strings and software*, tech. rep., AT&T Laboratories, 1993.
15. B. S. Baker: *A theory of parameterized pattern matching: Algorithms and applications*, in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, 1993.
16. B. S. Baker: *Parameterized pattern matching by boyer-moore-type algorithms*, in Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 1995, p. 550.
17. B. S. Baker: *Parameterized duplication in strings: Algorithms and an application to software maintenance.* SIAM Journal on Computing, 26(5) 1997, pp. 1343–1362.
18. B. S. Baker: *Parameterized diff*, in Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1999, pp. 854–855.
19. B. S. Baker: *Finding clones with dup: Analysis of an experiment.* Software Engineering, IEEE Transactions on, 33(9) 2007, pp. 608–621.
20. R. T. Batey, R. P. Rambo, and J. A. Doudna: *Tertiary motifs in rna structure and folding.* Angewandte Chemie International Edition, 38(16) 1999, pp. 2326–2343.
21. R. Beal and D. Adjeroh: *Efficient pattern matching for rna secondary structures.* Theoretical Computer Science. 2015.
22. R. Beal and D. Adjeroh: *p-suffix sorting as arithmetic coding*, in Combinatorial Algorithms: 22th International Workshop, IWOCA 2011, Victoria, Canada, July 20-22, 2011, Revised Selected Papers, vol. 7056, Springer, 2011, p. 44.
23. R. Beal and D. Adjeroh: *Parameterized longest previous factor*, in Combinatorial Algorithms: 22th International Workshop, IWOCA 2011, Victoria, Canada, July 20-22, 2011, Revised Selected Papers, vol. 7056, Springer, 2011, p. 31.
24. R. Beal and D. Adjeroh: *Border array for structural strings*, in Combinatorial Algorithms: 23rd International Workshop, IWOCA 2012, Krishnankoil, India, July 19-21, 2012, Revised Selected Papers, vol. 7643, Springer, 2012, p. 189.

25. R. Beal and D. Adjeroh: *p-suffix sorting as arithmetic coding.* Journal of Discrete Algorithms, 16 2012, pp. 151–169.
26. R. Beal and D. Adjeroh: *Parameterized longest previous factor.* Theoretical Computer Science, 437 2012, pp. 21–34.
27. R. Beal and D. Adjeroh: *Variations of the parameterized longest previous factor.* Journal of Discrete Algorithms, 16 2012, pp. 129–150.
28. R. Beal and D. Adjeroh: *Compressed parameterized pattern matching*, in Data Compression Conference (DCC), 2013, IEEE, 2013, pp. 461–470.
29. R. Beal, D. Adjeroh, and A. Abbasi: *The forward stem matrix: An efficient data structure for finding hairpins in rna secondary structures*, in Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics, ACM, 2013, p. 575.
30. R. Beal and D. A. Adjeroh: *The structural border array.* Journal of Discrete Algorithms, 23 2013, pp. 98–112.
31. R. A. Beal: *Parameterized strings: Algorithms and data structures*, Master's thesis, West Virginia University, 2011.
32. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.-T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theoretical Computer Science, 40(1) 1985, pp. 31–55.
33. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications ACM, 20(10) 1977, pp. 762–772.
34. E. Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, and Y. Pinzon: *Algorithms for computing approximate repetitions in musical sequences.* International Journal of Computer Mathematics, 79(11) 2002, pp. 1135–1148.
35. W. I. Chang and E. L. Lawler: *Approximate string matching in sublinear expected time*, in Proceedings of the 31st Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1990.
36. R. Cole and R. Hariharan: *Faster suffix tree construction with missing suffix links.* SIAM Journal on Computing, 33(1) 2004, pp. 26–42.
37. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: *Speeding up two string-matching algorithms.* Algorithmica, 12(4) 1994, pp. 247–267.
38. M. Crochemore and L. Ilie: *Computing longest previous factor in linear time and applications.* Information Processing Letters, 106(2) 2008, pp. 75–80.
39. M. Crochemore, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń: *Efficient algorithms for three variants of the lpf table.* Journal of Discrete Algorithms, 11 2012, pp. 51–61.
40. M. Crochemore and G. Tischler: *Computing longest previous non-overlapping factors.* Information Processing Letters, 111(6) 2011, pp. 291–295.
41. S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, and M. Takeda: *Parameterized suffix arrays for binary strings.*, in Stringology, 2008, pp. 84–94.
42. C. Du Mouza, P. Rigaux, and M. Scholl: *Parameterized pattern queries.* Data & Knowledge Engineering, 63(2) 2007, pp. 433–456.
43. N. J. Fine and H. S. Wilf: *Uniqueness theorems for periodic functions.* Proceedings of the American Mathematical Society, 16 1965, pp. 109–114.
44. K. Fredriksson and M. Mozgovoy: *Efficient parameterized string matching.* Information Processing Letters, 100(3) 2006, pp. 91–96.
45. J. Gramm, J. Guo, and R. Niedermeier: *Pattern matching for arc-annotated sequences*, in Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, 2002, pp. 182–193.
46. D. Gussfield: *Algorithms on strings, trees, and sequences.* Computer Science and Computional Biology (Cambrigde, 1999), 1997.
47. D. Harel and R. E. Tarjan: *Fast algorithms for finding nearest common ancestors.* SIAM Journal on Computing, 13 1984, p. 338.
48. C. Hazay: *Parameterized matching*, Master's thesis, Bar-Ilan University, 2004.
49. C. Hazay, M. Lewenstein, and D. Sokol: *Approximate parameterized matching.* ACM Transactions on Algorithms (TALG), 3(3) 2007, p. 29.
50. S. Heyne, S. Will, M. Beckstette, and R. Backofen: *Lightweight comparison of rnas based on exact sequence-structure matches.* Bioinformatics, 25(16) 2009, pp. 2095–2102.
51. T. I, S. Deguchi, H. Bannai, S. Inenaga, and M. Takeda: *Lightweight parameterized suffix array construction*, in Combinatorial Algorithms: 20th International Workshop, IWOCA 2009, Hradec nad Moravicí, Czech Republic, June 28–July 2, 2009, Revised Selected Papers, vol. 5874, Springer, 2009, p. 312.

52. T. I, S. Inenaga, H. Bannai, and M. Takeda: *Counting parameterized border arrays for a binary alphabet*, in Proceedings of the 3rd International Conference on Language and Automata Theory and Applications, Springer-Verlag, 2009, pp. 422–433.
53. T. I, S. Inenaga, H. Bannai, and M. Takeda: *Verifying and enumerating parameterized border arrays.* Theoretical Computer Science, 412(50) 2011, pp. 6959–6981.
54. R. M. Idury and A. A. Schäffer: *Multiple matching of parameterized patterns.* Theoretical Computer Science, 154(2) 1996, pp. 203–224.
55. T. Jiang, G.-H. Lin, B. Ma, and K. Zhang: *The longest common subsequence problem for arc-annotated sequences*, in Combinatorial Pattern Matching, Springer, 2000, pp. 154–165.
56. O. Keller, T. Kopelowitz, and M. Lewenstein: *On the longest common parameterized subsequence.* Theoretical Computer Science, 410(51) 2009, pp. 5347–5353.
57. D. E. Knuth, J. H. Morris Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6 1977, p. 323.
58. S. R. Kosaraju: *Faster algorithms for the construction of parameterized suffix trees*, in Proceedings of the 36th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Washington, DC, USA, 1995.
59. I. Lee, J. Mendivelso, and Y. J. Pinzón: *$\delta\gamma$–parameterized matching.* Lecture Notes in Computer Science, String Processing and Information Retrieval, 5280 2008, pp. 236–248.
60. T. Lee, J. C. Na, and K. Park: *On-line construction of parameterized suffix trees*, in String Processing and Information Retrieval, Springer, 2009, pp. 31–38.
61. T. Lee, J. C. Na, and K. Park: *On-line construction of parameterized suffix trees for large alphabets.* Information Processing Letters, 111(5) 2011, pp. 201–207.
62. R. C. Lyndon and M.-P. Schützenberger: *The equation $a^m = b^n c^p$ in a free group.* The Michigan Mathematical Journal, 11 1962, pp. 289–298.
63. M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Mathematics, 25(1) 1989, pp. 145–153.
64. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22 1993, p. 935.
65. E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of the ACM (JACM), 23(2) 1976, pp. 262–272.
66. J. Mendivelso: *Definition and solution of a new string searching variant termed $\delta\gamma$–parameterized matching*, Master's thesis, Universidad Nacional de Colombia, 2010.
67. J. Mendivelso: *The graph pattern matching problem through parameterized matching*, PhD thesis, Universidad Nacional de Colombia, 2015.
68. J. Mendivelso, S. Kim, S. Elnikety, Y. He, S.-w. Hwang, and Y. Pinzón: *Solving graph isomorphism using parameterized matching.* Lecture Notes in Computer Science, String Processing and Information Retrieval, 8214 2013, pp. 230–242.
69. J. Mendivelso, I. Lee, and Y. J. Pinzón: *Approximate function matching under δ-and γ-distances.* Lecture Notes in Computer Science, String Processing and Information Retrieval, 7608 2012, pp. 348–359.
70. E. W. Myers: *An $O(ND)$ difference algorithm and its variations.* Algorithmica, 1(1) 1986, pp. 251–266.
71. L. Salmela and J. Tarhio: *Sublinear algorithms for parameterized matching*, in Combinatorial Pattern Matching, Springer, 2006, pp. 354–364.
72. L. Salmela and J. Tarhio: *Fast parameterized matching with q-grams.* Journal of Discrete Algorithms, 6(3) 2008, pp. 408–419.
73. B. Schieber and U. Vishkin: *On finding lowest common ancestors: Simplification and parallelization.* SIAM Journal on Computing, 17 1988, p. 1253.
74. T. Shibuya: *Generalization of a suffix tree for rna structural pattern matching.* Algorithmica, 39(1) 2004, pp. 1–19.
75. D. D. Sleator and R. Endre Tarjan: *A data structure for dynamic trees.* Journal of Computer and System Sciences, 26(3) 1983, pp. 362–391.
76. B. Smyth: *Computing patterns in strings*, Pearson Education, 2003.
77. I. Tomohiro, S. Inenaga, H. Bannai, and M. Takeda: *Counting and verifying maximal palindromes*, in String Processing and Information Retrieval, Springer, 2010, pp. 135–146.
78. K. Zhang, L. Wang, and B. Ma: *Computing similarity between rna structures*, in Combinatorial Pattern Matching, Springer, 1999, pp. 281–293.
79. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on information theory, 23(3) 1977, pp. 337–343.

Refined Tagging of Complex Verbal Phrases for the Italian Language*

Simone Faro[1] and Arianna Pavone[2]

[1] Dipartimento di Matematica e Informatica, Università di Catania,
Viale A. Doria 6, I-95125 Catania, Italy
[2] Dipartimento di Scienze Umanistiche, Università di Catania
Piazza Dante 32, I-95124 Catania, Italy
faro@dmi.unict.it, pavone@unict.it

Abstract. A verb phrase is a syntactic unit consisting of one verbal form, combined with any other elements, representing the verbal part of the speech. In Italian, as in many other languages, the verb phrase is the central element in a sentence. In this paper, we investigate the problem of the automatic recognition of complex verb phrases in the Italian language, where the wide variety of syntactic units and the complexity of morphology make the problem more difficult to solve than in English. In particular we propose an hybrid approach which faces the recognition and the disambiguation of Italian verb phrases by using language generation. We provide also a web tool[1] for testing and querying our method. The level of accuracy and the grade of detail reached by our solution is higher than any other known approach.

1 Introduction

The recognition of terms and phrases which compose a text is one of the main problems concerning with the automatic information extraction from natural language texts. This process is also at the base of a large area of applications, such as semantic analysis of natural language texts, automatic paraphrase, knowledge bases construction, automatic spelling and part of speech tagging. The process of recognition, analysis and paraphrase of the components of a natural language text is certainly more complex than the reverse process used in the automatic generation of the language itself. The complexity of such recognition process is due to the possible presence of a large number of variants, concerning the syntax and the grammar, that must be taken into account in the parsing process of the text. In addition it is also necessary to determine the appropriate syntactic and semantic features to be applied to it. Differently these details are prearranged in the process of automatic generation of natural language text.

However, there is considerable commercial interest in natural language recognition, mainly due to its numerous applications in various fields such as information extraction, categorization of texts, storage and analysis of large-scale content. The recognition of parts of speech (PoS) also finds application as a component in tools for grammatical spell-correction of texts. Such tools are currently unable to recognize the correctness of complex verb phrases as *se ne era accorta*[⋆2], but are limited to the

⋆ This work has been supported by project PRISMA PON04a2 A/F funded by the Italian Ministry of University and Research within the PON 2007-2013 framework.

[1] The web tool is provided to the reviewers in order to establish the effectiveness of our solution. It can be accessed at http://www.dmi.unict.it/~faro/tagger/voci_verbali.php .

[2] Along the paper we will present several examples of Italian verb phrases, together with the corresponding English translation, where necessary. Each example is also an anchor (identified by the symbol ⋆) which links to the recognition tool activated for the corresponding verb phrase.

Simone Faro, Arianna Pavone: Refined Tagging of Complex Verbal Phrases for the Italian Language, pp. 132–145.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

correction of the terms composing the phrase. These instruments are not therefore able to recognize some types of grammatical errors, such as those which we can find in the sentence *l'aveva stato*⋆, in which the error is in the choice of the auxiliary relative to the main verb. Solutions to these problems would find application in many tools such as word processors, e-mail clients, electronic dictionaries, and search engines.

In this paper we address the problem of recognition and disambiguation of Italian phrases, with particular reference to the recognition of verb phrases. This work is part of a more complex project named A.R.I.ANN.A. (Automatic Refined Italian ANNotation Approach) whose aim is to produce a refined syntactic and logical tagger for the Italian language.

The paper is organized as follows. In Section 2 we will give a general view of the state of the art and we will briefly describe related works known in literature. Then we will introduce the new tool for the recognition of complex verb phrases for the Italian language. In particular we will discuss separately the features of the tool and the recognized verbal forms (Section 3) and the recognition scheme (Section 4). We will draw our conclusions in Section 5.

2 Related Work

The analysis of the parts of speech (PoS Tagging problem), with reference to the English language, is considered a simple problem today. The experimental results obtained by Tsuruoka and Tsujii [2] show that the PoS tagging solutions available for the English language can reach an accuracy up to 97%. Such solutions are generally based on machine learning techniques such as dependency networks [3], perceptrons [4], support vector machines (SVM, also known as support vector networks) [5] or hidden Markov models (HMM) [6]. Such problem consists in analyzing a natural language text and in associating each part of the speech to a tag, selected from a predetermined set of tags. Such tag set could be more or less refined.

The reference tag set used in PoS Tagging for the English language is the Penn Treebank tag set [7], which divides the parts of speech in 36 categories. The same problem, with reference to the Italian language, has been dealt with in the context of EVALITA (Evaluation of NLP and Speech Tools for Italian[3]) an initiative aimed at the evaluation of the tools for the analysis of natural language, with reference to the Italian language.

In the course of the competition proposal in 2007 [8] the set of tags consisted in 32 lexical categories proposed by Treebank tag-set of the University of Pennsylvania, adapted to the Italian language, whose 6 categories were devoted to description of verb phrases. In that case the best solution achieved an accuracy of 98%.

In the course of the competition revived in 2009 [9] a set of lexical classes was used, widened to 37 elements with different morphological variants allowing the identification of 336 different elements. The set of reference tags is TANL (Text Analytics and Natural Language), made in accordance with the EAGLES guideline [10], a standard for English language, recognized by the community in natural language processing. The TANL tag set includes three levels of accuracy of tag, of which the highest level consists of 14 categories. those relating to the verb phrases are listed in Table 2, in

[3] Information related to different competitions proposals under EVALITA can be found at the web page www.evalita.it

Tag	**Description**	**Examples (Italian)**
VB	verb, lemma	*leggere, conoscere, andare*
VBD	verb, past	*leggevo, conobbi, andasti*
VBG	verb, gerund or present participle	*leggendo, conoscente, andando*
VBN	verb, past participle	*letto, conosciuta, andati*
VBP	verb, present, non-third singular person	*leggevamo, conosco, vai*
VBZ	verb, present, third singular person	*legge, conosce, va*

Table 1. The Treebank tag-set relative to verb phases.

Tag	**Description**	**Examples**
V	verb	*leggere, conosco, andato*
VA	auxiliary verb	*sono, eravamo, hanno*
VM	modal verb	*volevo, posso, dobbiamo*
Suffix	**Description**	**Examples**
-m	masculine	*letto, conosciuti, andato*
–f	feminine	*lette, conosciuta, andata*
–n	not specified	*leggo, conoscere, vanno*
–s	singular	*letto, conosci, va*
–p	plural	*lette, conoscevano, vanno*
–n	not specified	*leggere, conoscere, andare*
–1	first person	*leggevo, conosco, andammo*
–2	second person	*leggi, conoscevi, andrete*
–3	third person	*legge, conobbe, vanno*
–i	indicative	*leggo, conosceva, andavamo*
–m	imperative	*leggi, conosca, andate*
–c	subjective	*legga, conoscano, andassimo*
–d	conditional	*leggerei, conoscerebbe, andresti*
–g	gerund	*leggendo, conoscendo, andando*
–f	infinitive	*leggere, conoscere, andare*
–p	participle	*letto, conosciuta, andato*
–p	present	*leggo, conosco, vai*
–i	present perfect	*leggevo, conoscevi,*
–s	past	*lessi, conoscesti, andarono*
–f	future	*leggerà, conoscerete, andranno*
–c	clitics	*leggendocele, conoscilo*

Table 2. The TANL tag-set relative to verb phrases.

which are also shown suffixes that can be integrated to the main tag in order to describe form, tense, mood, number, person and also the possible presence of clitics.

Most of these solutions are able to recognize the parts of speech by associating the terms in the text with the entries in some lexical Knowledge Base (KB), as WordNet [12], Multi-WordNet [13], Euro-WordNet [14], BabelNet [15] or similar ones. Apart from WordNet, which contains only lemmas for the English language, other lexical

resources also contain lemmas of the Italian language, as well as those of many other languages. These lemmas include nouns, verbs, adjectives, adverbs etc. Each lemma or phrasal term in a KB, is associated to its sense, usually identified with one of the synsets related to the given term.

One of the most difficult challenges in the recognition of phrases in a natural language text is that these phrases are often composed of several terms. In WordNet 3.0, for instance, over 40% of the items are compound phrases, while the Italian version of MultiWordNet 1.5 the number of such phrases is 15%. The compound phrases are difficult to be accurately recognized for three main reasons:

a) In the first place, the terms which compose a compound phrase are themselves voices of the KB. For example, the verb phrase *essere caduto** (*to have fallen*, past infinitive) is composed by two separate verb phrases, *essere** (*to be*, present infinitive) and *caduto** (*fallen*, past participle). This is typically the output produced by the PoS tagging solutions described above, which ignore the issues of compound phrases splitting the entire term in the constituent subterms.
b) Secondly, the terms composing a compound phrase may not appear contiguously in the text. For example the verb phrase *essere improvvisamente caduto** (*to have suddenly fallen*) contains the verb phrase *essere caduto** (*to have fallen*) which is separated by the modal adverb *improvvisamente* (*suddenly*).
c) Finally, the conjugation of the terms contained in a compound verbal phrase may lead to a difficult recognition. For instance the verb phrase *esserle caduta addosso** (*to have fallen on top of her*) contains the verb phrase *esserle caduta** (*to have fallen on her*, past infinitive, clitic form, singular) with difficult recognition because of its pronominal form.

None of the above described problems are solved by state of the art PoS tagging solutions for the Italian language. Only very recently Del Corro *et al.* [1] addressed some of the above problems introducing a tool that allows to make jointly the recognition of the phrase and its disambiguation in Italian. The solution we describe in this paper solves a) and c) and could be adapted to solve b) as well.

Online there are many facilities for the generation of verb phrases, but limited only to the conjugation of verbs. Among the services for the Italian language the most used are Italian-Verbs[4], Coniugazione.it[5], it.bab.la[6], WordReference.com[7] e Virgilio.it[8]. Most of these services offer the possibility to conjugate verbs, not only in their active form, but also in the passive and reflective, if available.

3 A New Tool for the Recognition of Italian Verb Phrases

In linguistics a syntagma is a unit of varying syntactic complexity and autonomy, which is between the word and sentence. The verb phrase is a syntagma consisting of a verbal form together with any other elements, but it is still the verbal part of the speech. In this section we describe the features of our tool and the different verb forms which it is able to detected. We also focus our attention on some problems related to Italian syntax which make the recognition a difficult task.

[4] http://www.italian-verbs.com
[5] http://www.coniugazione.it
[6] http://it.bab.la/coniugazione/italiano/
[7] http://www.wordreference.com/conj/Itverbs.aspx
[8] http://parole.virgilio.it/parole/verbi_italiani/

In Italian, as in other languages, the verb phrase is the variable part of the speech and indicates an action, a state or a becoming in relation to a subject, expressed or implied, that does or undergoes an action. Some examples of verb phrases recognized by our tool are:

mangio⋆	(*I eat*)
sono andato⋆	(*I went*)
mi fu concesso⋆	(*I was allowed*)
le è stato mandato⋆	(*it was sent to her*)
mi pettino⋆	(*I comb my hair*)

The head of the verb phrase is the verb, the more complex part of speech under the grammatical aspect, which may vary according to different categories of reference. In Table 3 we show the tag set used in our solution. It reflects the level of detail of the recognition process. It includes 3 head tags and and 30 feature tags, beginning with a symbol ":", which can be added to any head tag in order to increase the level of detail. The tag set allows the identification of more than 10 000 different verb forms.

Regarding their value, verbs can be transitive (tag TR) or intransitive (tag IN); regarding their form or diathesis (describing the relationship between actor and action) a verb can be active (tag VSA), passive (tag VSP), reflexive (tag VPR). In addition, regarding the subject which they refers to, we can have verbs of first (tag 1), second (tag 2) or third person (tag 3). By number they can be singular (tag S) or plural (tag P). Regarding the performance of the action, verbal forms can vary according to a range of tenses and moods, as described below.

Our system is based on a manually annotated KB containing a set of more than 5.700 verb lemmas, including 151 intransitive reflexive forms. Verbal lemmas are also categorized according to their values. In particular about 180 verbal lemmas are recognized as intransitive verbs. The number of verbs accepting the auxiliary *avere*⋆ (*to have*) is about 3.650, while about 500 verbal entries accept the auxiliary *essere*⋆ (*to be*). There is also a third class of items that accepts both verbal auxiliaries, consisting of about 310 items in our KB.

3.1 Recognition of tenses and verbal moods

The verbal moods express attitudes that the speaker establishes against the interlocutor. In the Italian language, we distinguish the following moods: *indicative*, *subjunctive*, *conditional*, *imperative*, *gerund* and *participle*. Each of these verbal moods consist of some simple and some compound times. The latter ones are formed by combining the auxiliary verbs, *essere*⋆ or *avere*⋆, with the past participle of the verb itself.

The *indicative* (IND tag) shows the reality of a fact, which can be true or false. This verbal mode is very used in main clauses, i.e. independent grammatical clauses. For instance *mangio una mela*⋆ (*I eat an apple*) is an objective remark. For the conjugation of verbs, the indicative has four simple tenses (present, imperfect, far past and future) and four compound tenses (present perfect, past perfect, distant past perfect and future perfect).

The *subjunctive* (CNG tag) indicates a situation for which it is not possible to propose a real judgment of truth, because it concerns a desire, a possibility or a supposition. It consists of two simple tenses and two compound tenses.

The *conditional* (CND tag) indicates the presence of a real or unreal conditioning of the reality of facts, of an action or process. The conditional consists in a single simple tense (present) and a single compound tense (past).

The *imperative* tense (IMP tag) indicates an exhortation and a command. It has a single tense, the present, and only two forms: the second person (singular) and second person (plural). For other person, the imperative, borrows forms of the subjunctive, and in this case becomes exhortative subjunctive.

The *gerund* (tag GER) is a verbal mode which has just two tenses: simple and compound, present and past. It is used in subordinate clauses and establishes a relationship of contemporaneity to the action expressed by the verb in the main clause.

The *participle* (PAR tag) has two simple tenses: *present* and *past* participle. The participle is a mood participating in both the category names (from which it draws the conjugation, distinguishing between voice and aspect).

The *infinitive* (INF tag), denoted by the lemma of the verb, has a simple tense, the present, and a composed tense, the past.

All the grammatical forms of the verb relating to mood, tense and person, number and diathesis, constitute the conjugation. The Italian has three conjugations, distinguished by the infinite endings: *-are*, *-ere*, *-ire*: each conjugation has its paradigm, consisting in a series of endings and suffixes, by which, starting from the theme of the verb, the verbs are formed depending on different moods and tenses;

3.2 Recognition of pronominal verbal forms

In Italian there are particular verbal forms with particles, called *clitics*. These clitics attach themselves to a word and they form a single unit. For instance, *leggerla** (legger-la, *to read it*), *leggerne** (legger-ne, *to read some of them*) and *leggerci** (legger-ci, *to read to us*). Some of these verbs incorporate two clitics together, in these cases they are bi-pronominal verbs. Some examples are *leggersela** (legger-se-la, *to read it to himself*), *leggersene** (legger-se-ne, *to read some of them to himself*) and *leggerceli** (legger-ce-li, *to read them to ourselves*).

The pronominal verbs are divided into classes, distinguished by the clitic and the meaning. Specifically, we distinguish the following forms:

Verb forms including an direct object.
They are built with the particles *-mi -ti -lo -la -li -le -ci* and *-vi*, where the particle assumes the function of direct object (with the meaning, respectively, of *me*, *you*, *him her*, *us*, *you* and *them*). When the subject and the object are the same, the verbs in these forms indicate that the action expressed by the verb is reflexive, and it is related with the subject itself that performs the action. It is important to distinguish the different cases, for which we can not speak of reflexive constructions, as in the cases listed above. If the particles *-lo -la -li -le* are prefixed to the verb beginning with a vowel, the elision of the vowel is common: thus *l'amo** is equivalent to *la amo** (*I love her*).

Other examples are:

1. *lo porti** (*you bring it*)
2. *portarmi** (*to bring me*)
3. *se l'avessi portata** (*if you had brought it*)

Forms	**Description**	**Examples**
VSA	standard active	*capisco*⋆
VSP	standard passive	*sono capito*⋆
VPA	pronominal active	*avendolo capito*⋆
VPP	pronominal passive	*avendomi capito*⋆
VPR	pronominal reflexive	*essendomi capito*⋆
Values	**Description**	**Examples**
:TR	transitive	*capissi*⋆
:IN	intransitive	*andassi*⋆
Tenses	**Description**	**Examples**
:IND	indicative	*avevo capito*⋆
:CNG	subjective	*avessi capito*⋆
:CND	conditional	*avrei capito*⋆
:IMP	imperative	*capisci*⋆
:GER	gerund	*avendo capito*⋆
:PAR	participle	*capente*⋆
:INF	infinitive	*capire*⋆
Moods	**Description**	**Examples**
:PRE	present	*capisco*⋆
:PAS	past	*capivo*⋆
:FUT	future	*capirò*⋆
:IMP	present perfect	*avevo capito*⋆
:PRM	past perfect	*ebbi capito*⋆
:TRA	distant past perfect	*avessi capito*⋆
:FAN	future perfect	*avrò capito*⋆
Gender	**Description**	**Examples**
:M	male	*è stato capito*⋆
:F	female	*è stata capita*⋆
:N	neuter	*abbiamo capito*⋆
Number	**Description**	**Examples**
:S	singular	*capisci*⋆
:P	plural	*capiamo*⋆
:I	invariable	*capire*⋆
Person	**Description**	**Examples**
:P0	impersonal	*aver capito*⋆
:P1	first person	*abbiamo capito*⋆
:P2	second person	*avete capito*⋆
:P3	third person	*hanno capito*⋆
Clitic	**Description**	**Examples**
:COC	object complement	*avermi portato*⋆
:CTC	term complement	*avergli portata*⋆
:CPC	place complement	*averci portati*⋆
:CPF	partitive complement	*averne portate*⋆

Table 3. The new tag-set introduced in this paper. On top the list of head tags is listed. any of all other tags in the list, beginning with a symbol ":", can be added to the head tag in order to increas the level of detail. The set allows the identification of more than 10 000 different verb forms.

Verb forms including an indirect object.
Some pronominal forms use the particles *-mi* and its conjugations in gender and number, *-ti -gli -le -ci -vi*. In this case the pronominal particle is used as an indirect object (with the meaning of *to me*, *to you*, *to him*, *to her*, etc). This form is used with both transitive and intransitive verbs.

Other examples are:

1. *gli porti** (*you bring to him*)
2. *portarmi** (*to bring to me*)
3. *le avessi portata** (*you had brought to her*)

Verb forms including an adverb of place.
They are built by using the pronominal particle *-ci* or *-ne*, which have the function of adverb of place. The particle *-ci* is used with the meaning of *in that/this place* while the particle *-ci* is used with the meaning of *from that/this place*. In this context, the verb phrase *andarci** (*to go there**) can be paraphrased as *andare in quel luogo** (*to go in that place**). Other examples are:

1. *arrivarci** (*to reach that place*)
2. *ne vengo ora** (*I came now from there*)
3. *lui ci viene** (*he came here*)

Verb forms including a partitive complement.
The particle *-ne** can be used also with the meaning *of that/this/them* with a partitive function. It can be applied to transitive and to intransitive verbs as well. Example of these verb phrases are:

1. *parlarne** (*to speak about that*)
2. *ne avevamo spesi** (*we spent some of them*)
3. *ne porterò due** (*I will bring two of them*)

Bi-pronominal verb forms.
Many of the particles used for the composition of pronominal verb forms listed above can, in general, composed to create bi-pronominal verb forms. The particles obtained in this way, can be listed in the following forms:

1. adverbof place + direct object *ci + (lo/la/li/le)*
2. direct object + adverbof place *(me/te/se/ve) + ci*
3. adverb of place + partitive complement *ci + ne*
4. indirect object + partitive complement *(me/te/se/ce/ve) + ne*
5. indirect object + direct object *(me/te/se/ce/ve) + (lo/la/li/le)*

Example of these verb phrases are, respectively:

1. *portarcelo** (*to bring it in that place*)
2. *portarmici** (*to bring me in that place*)
3. *portarcene** (*to bring there some of them*)
4. *portarmene** (*to bring some of them to me*)
5. *portarvelo** (*to bring it to you*)

3.3 Recognition of irregular verbal forms

In the Italian language there are many verbs that do not follow the whole regular paradigm related with their conjugation and for this reason they are called irregular; these verbs are used quite common, for instance, the verbs *essere** (*to be*), *avere** (*to have*), *andare** (*to go*), *fare** (*to do*), etc. In relation to the conjugation, we distinguish defective verbs, i.e. without some forms, such as *vertere** and overabundant verbs, which follow different conjugations in all or in some tenses, such as *starnutare** – *starnutire** (*to sneeze*), *adempiere** – *adempire** (*to fulfill*). In our system verbal entries are divided according to three main conjugations. Irregular forms are listed based on the class of regularity.

For instance, the verbs *abbassare** (*to decrease*) and *appellare** (*to appeal*) belong to the same class because they have the same irregularities in their conjugation. In particular belong to the first conjugation more than 4 200 entries, of which about 1 200 are irregular forms, divided into 6 classes of irregularities; belong to the second conjugation approximately 500 entries, of which 440 are irregular forms, divided into 19 classes; the voices belonging to the third conjugation are about 520, of which about 480 irregular forms, divided into 12 classes of irregularities.

3.4 Ambiguity of the recognition of compound tenses

The compound tenses consist in (at least) two terms: an auxiliary verb, *essere** (*to be*) or *avere** (*to have*), conjugates in a simple tense, and a main verb conjugated in the past participle. In this context the past participle can be composed depending on the number or on the gender. The correct recognition (and the consequent tagging) of this verbal form creates some problems since in the Italian language the compound verbs can be composed in different ways.

In particular the question of the correspondence of the past participle is one of the most difficult chapter of Italian syntax. The main errors that usually arise in the correspondence, and that we addressed in our solution, can be summarized as follows: **a)** if the verb is transitive and accepts the auxiliary *avere**, then it is possible to accord the participle of the verb, both in masculine or feminine, and also with the object complement, even if the first form is more used than the second. Thus the sentence *I chose the best solutions* can be translated as:

a1. *ho scelto le migliori soluzioni**
a2. *ho scelte le migliori soluzioni**

b) If the verb is transitive and accepts the auxiliary *avere** (*to have*) and the compound verb is preceded by a personal or a relative then it is possible to accord the participle of the verb with the prefixed object. Thus the sentence *He has cheated us* can be translated as:

b1. *ci ha ingannato**
b2. *ci ha ingannati**

c) If the verb accepts the auxiliary *essere** (*to be*) then it is possible to accord the participle of the verb with the subject or with the predicate complement. Thus the sentence *it was a news* can be translated as:

c1. *lo è stato una novità**
c2. *lo è stata una novità**

d) When the verb phrase is in pronominal transitive form, the participle of the verb can be accorded with the subject or with the object complement, even if it is prefixed to the verb. Thus, the sentence *since we set ourselves that goal* can be translated as:

d1. *essendocelo prefissati*⋆
d2. *essendocelo prefissato*⋆

The possibilities of choice among the points reported above have always existed in the Italian language and the restrictions indicated by some grammarian are considered to be unfounded. Our system recognizes as accurate all previous combinations, providing the correct interpretation of the correspondence of participle.

3.5 Actives, passives and reflexives forms

In the Italian language, and also in other languages, verbal entries can take various forms: *active*, *passive*, *reflexive* and *pronominal*.

A verb is in active form, when the subject performs the action:

lei guarda⋆ (*she looks*)

A verb is in passive form when it is the subject who undergoes the action:

lei è guardata⋆ (*she is looked*)

The passive form is characterized by the auxiliary *essere*⋆ (*to be*) followed by the past participle of the verb. Only transitive verbs can take the passive form. Reflexive verbs are accompanied by a reflexive pronoun (*mi*⋆, *ti*⋆, *si*⋆, *ci*⋆, *vi*⋆) which comply with the subject. The presence of the reflexive pronoun, which can be prefixed or postponed to the verb, makes a phrase in pronominal form.

Examples are:

mi guardo⋆ (*I look at me*)
guardatevi⋆ (*look at yourselves*)

There are different types of reflexive verbs. The wider class of reflexive verbs is obtained by entries that admit both transitive reflexive forms, and active (*io lavo*, *io mi lavo*). There are also reflexive verbs that are used with reciprocal value, that allow a reading for which an event, which has at least two promoters subjects, is realized when the effects produced by the first fall on the second, and the effects produced by the second fall on the first, as in:

amarsi⋆ (*to love each other*)
sposarsi⋆ (*to marry*)
spingersi⋆ (*to push each other*)

4 The recognition process

In this section we briefly describe the recognition process on which our solution is based. As we noticed above, in general, the process of recognition of the components of a natural language text is more complex than the reverse process of language generation. This is particularly true in the case of Italian verb phrases, where the grammar is hardly structured and allows phrases as complex compound sequences of terms.

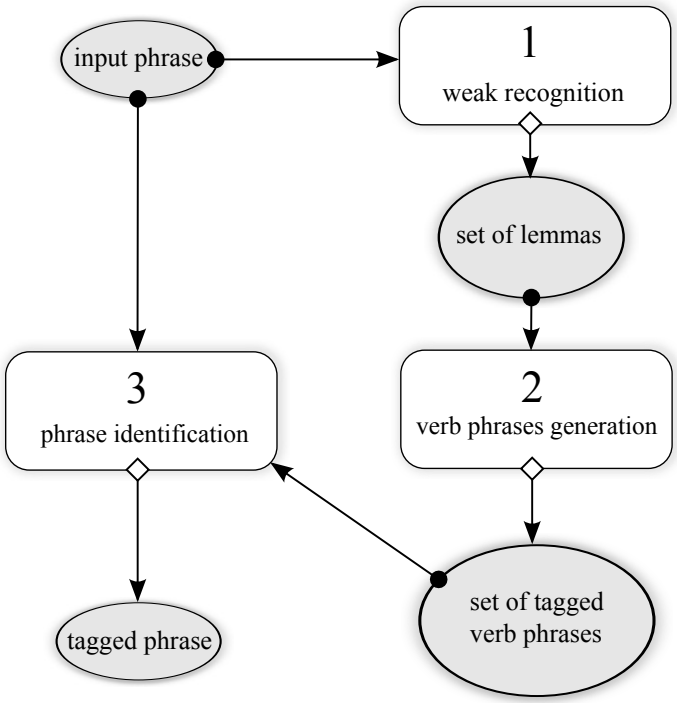


Figure 1. The scheme which describes the recognition process. Data are represented by grey circles, while recognition steps are represented by white rectangles. Input data starts with a black circle, while output data come from white square.

The tool takes as input a text, which consists in a sequence of terms. It identifies all the verb phrases contained in such a text, assuming that each term of the phrases can participate only of a single verb phrase.

In order to simplify the correct and detailed recognition of verb phrases we mix a ruled based weak recognition approach with a robust finite model approach for language generation. The process is then finalized by using a fast string matching subroutine. Specifically the recognition process is divided in three steps.

- *weak recognition:* the input text is tokenized and each term is associated with a (possibly empty) set of verb lemmas;
- verb phrases generation: each lemma is processed and a set of possible (already tagged) matching verb phrases is generated;
- final identification: the input phrases are matched against the candidate verb phrases in order to perform the correct association.

Figure 1 shows the scheme which describes the recognition process. Data are represented by grey circles, while recognition steps are represented by white rectangles. Input data starts with a black circle, while output data come from white square.

In what follows we briefly describe each step of the process of Figure 1. We suppose that the input phrase t is a sequence of n terms $t = \langle x_1 x_2 \cdots x_n \rangle$, with $n \geq 1$.

I. Weak recognition step.
During the first step the input phrase is tokenized and each term x_i is analyzed, for $i = 1, \ldots, n$. In particular the algorithm uses a weak recognition process in order to estabilish whether a term x_i is a verb or not. At this level the tool is unable to identify the correct features of the term, like tense, mood, person and gender. Such a process allows only the identification of the lemma (or the list of lemmas) associated with the term, if the latter is a verb phrase. For instance if x_i is the term*andassimo* the tool will associate it with the lemma *andare*.

Specifically, each term x_i is decomposed in two substrings p_i (a prefix) and s_i (a suffix) such that $x_i = p_i \cdot s_i$. Any possible decomposition of the type $x_i = p_i \cdot s_i$ is taken into account, with $|p_i| > 0$ and $|s_i| > 0$. If we find a prefix p_i which is equal to the radix of a verb v in our KB then we investigate if the corresponding suffix s_i could be a desinence of v. In such a case the verb v is returned as a lemma of x_i.

Observe that in some cases two or more lemmas can be associated to a single term. For instance the term *stato* can be associated to both lemmas *essere* (*to be*) and *stare* (*to stay*). If the input phrase is *ce lo avevano portato** (*they had brought it to us*) then the tool recognizes the following set of lemmas:

1. *ce*	$\emptyset$
2. *lo*	$\emptyset$
3. *avevano*	$\{avere^\star\}$
4. *portato*	$\{portare^\star\}$

while the set of lemmas relative to the input phrase *le era stato dato** (*it has been given to her*) are:

1. *le*	$\emptyset$
2. *era*	$\{essere^\star\}$
3. *stato*	$\{essere^\star, stare^\star\}$
4. *dato*	$\{dare^\star\}$

II. Verb phrases generation step.
During the second step of the recognition process the algorithm generates all possible verb phrases which are connected to the lemmas which have been identified at the previous step. Specifically, let x_i a term of the input text t, and let $\{\ell_1, \ell_2, \ldots, \ell_m\}$ the set of lemmas associated to x_i. The algorithm generates all possible verb phrases which are licensed by lemma ℓ_j, for $j = 1, \ldots, m$, by using a finite state model based on conjugation details stored in our KB.

As stated above, the set of verb phrases generated from a single lemma ℓ_i could contain more than 10 000 elements, even if, in most practical cases it is not larger than 9 000 elements, including active, passive, pronominal, simple and compound verb forms.

In addition, during the generation process, each produced verb phrase is associated with high precision to the correct tag. This can be done since form features are stored in the KB together with the conjugation details.

For example, some of the tagged verb phrases generated from the lemma *portare** (*to bring*) are

portare* $\rightarrow$ {	*porto**,	(VSA:TR:IND:PRE:N:S:P1)
	porti*	(VSA:TR:IND:PRE:N:S:P2)
	porta*	(VSA:TR:IND:PRE:N:S:P3)
	...	
	avessi portati*	(VSA:TR:CNG:TRA:N:S:P2)
	avesse portati*	(VSA:TR:CNG:TRA:N:S:P3)
	...	
	eravate state portate*	(VSP:TR:IND:IMP:F:P:P2)
	erano state portate*	(VSP:TR:IND:IMP:F:P:P3)
	...	
	ce lo avessi portato*	(VSA:TR:CNG:TRA:N:S:P2:COC:CTC)
	ce lo avesse portato*	(VSA:TR:CNG:TRA:N:S:P3:COC:CTC)
	... }	

III. Final identification step.
During the final step of the process the algorithm identifies any possible verb phrase in the input text t by using information generated at the previous step. Let x_i be a term in t and let ℓ_j a lemma associated to x_i during the first step. Moreover let V_j be the set of all possible verb phrases which are licensed by lemma ℓ_j, generated at the previous step. Notice that each verb phrase $v \in V_j$ is a sequence of terms $v = \langle y_1 y_2 \cdots y_k \rangle$, with $k \geq 1$.

In order to identify all verb phrases the algorithm checks whenever each sequence $v \in V$ is equal to any subsequence of length k in t which involves the term x_i. More formally the sequence p is compared with the subsequence $\langle x_h x_{h+1} \cdots x_{h+k} \rangle$, for $h = \max(1, i-k) \cdots \min(n, i+k)$.

Since each term can be involved in a single verb phrase, if two overlapping subsequences of t are recognized as verb phrases, then only the longest one is taken into account. For instance, in the sentence *ce lo avevano portato** ($t = \langle x_1 \cdots x_5 \rangle$) the tool will recognize the following verb phrases:

verb phrase	lemma	tag	position
1. *ce lo avevano**	*avere**	VSA:TR:IND:PAS:N:P:P3:CPC:COC	$\langle x_2 \ldots x_4 \rangle$
2. *lo avevano**	*avere**	VSA:TR:IND:PAS:N:P:P3:COC	$\langle x_3 \ldots x_4 \rangle$
3. *ce lo avevano portato**	*portare**	VSA:TR:IND:IMP:N:P:P3:CTC:COC	$\langle x_2 \ldots x_5 \rangle$
4. *ce lo avevano portato**	*portare**	VSA:TR:IND:IMP:N:P:P3:CPC:COC	$\langle x_2 \ldots x_5 \rangle$
5. *avevano**	*avere**	VSA:TR:IND:PAS:N:P:P3	$\langle x_3 \ldots x_3 \rangle$
6. *avevano portato**	*portare**	VSA:TR:IND:IMP:N:P:P3	$\langle x_4 \ldots x_5 \rangle$
7. *portato**	*portare**	VSA:TR:PAR:PAS:M:S:P1	$\langle x_5 \ldots x_5 \rangle$
8. *portato**	*portare**	VSA:TR:PAR:PAS:M:S:P2	$\langle x_5 \ldots x_5 \rangle$
9. *portato**	*portare**	VSA:TR:PAR:PAS:M:S:P3	$\langle x_5 \ldots x_5 \rangle$

However only the verb phrases n.3 and n.4 are returned as output since they overlaps all other choices.

5 Conclusions and Future Works

In this paper we presented a web tool for the recognition of complex verb phrases in the Italian language. Our solution is able to recognize a refined set of verbal forms, including passive, reflexive and pronominal forms. In addition the new proposed tool

is able to recognize compound verbs and to associate such forms to their right features, which include mood, tense, person, number, gender and type of the clitics, if present.

Future work will be devoted to increase the number of details recognized by our solution, allowing the identification of features related to the direct object or indirect object referred by the verb phrase, as number and gender. Moreover, the tool framework is general enough to be adapted to other languages like English, French or Spanish. In addition the recognition process could be also integrated in a more general solution to the PoS tagging problem for the Italian language.

References

1. L. Del Corro, R. Gemulla, and G. Weikum: *Werdy: Recognition and Disambiguation of Verbs and Verb Phrases with Syntactic and Semantic Pruning*, in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, 2014, pp. 374–385.
2. Y. Tsuruoka and J. Tsujii: *Bidirectional Inference with the Easiest-First Strategy for Tagging Sequence Data*, in Proceedings of HLT-EMNLP, 2005, pp. 467–474.
3. K. Toutanova, D. Klein, C. Manning, and Y. Singer: *Feature-rich part-of-speech tagging with a cyclic dependency network*, in Proceedings of HLT-NAACL, 2003, pp. 505–512.
4. M. Collins: *Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms*, in Proceedings of EMNLP, 2002, pp. 1–8.
5. J. Gimenez and L. Marquez: *Fast and accurate part-of-speech tagging: the SVM approach revisited*, in Proceedings of RANLP, 2003, pp. 158–165.
6. T. Brants: *TnT: a statistical part-of-speech tagger*, in Proceedings of the 6th Applied NLP Conference, 2000, pp. 224–231.
7. M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz: *Building a Large Annotated Corpus of English: The Penn Treebank*. Computational Linguistics, vol. 19, issue 2, 1993, pp. 313–330.
8. F. Tamburini: *EVALITA 2007: The Part-of-speech Tagging Task*. IA-Intelligenza Artificiale, Anno IV, issue 2, 2007, pp. 4–7.
9. G. Attardi and M. Simi: *EVALITA 2009: The Part-of-speech Tagging Task*, 2009.
10. M. Monachini: *ELM-IT: An Italian Incarnation of the EAGLES-TS. Definition of Lexicon Specification and Classification Guidelines*. Technical report, Pisa, 1995.
11. S. Montemagni et al.: *Building the Italian Syntactic-Semantic Treebank*, in Abeillé, ed., Building and using Parsed Corpora, Language and Speech series. Kluwer, Dordrecht, 2003, pp. 189–210.
12. G. A. Miller: *WordNet: A Lexical Database for English*. Communications of the ACM Vol. 38, No. 11, 1995, pp. 39–41.
13. E. Pianta, L. Bentivogli, and C. Girardi: *MultiWordNet: developing an aligned multilingual database*, in Proceedings of the First International Conference on Global WordNet, 2002, pp. 21–25.
14. P. Vossen: *EuroWordNet: A Multilingual Database with Lexical Semantic Networks*. Kluwer Academic Publishers, Dordrecht, 1998.
15. R. Navigli, S. P. Ponzetto: *BabelNet: Building a Very Large Multilingual Semantic Network*, in Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, 2010, pp. 216–225.

Author Index

Proceedings of the Prague Stringology Conference 2015
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, Praha 6, 160 00, Czech Republic.

ISBN 978-80-01-05787-2

URL: http://www.stringology.org/
E-mail: psc@stringology.org Phone: +420-2-2435-9811

Printed by Česká technika – Nakladatelství ČVUT
Zikova 4, Praha 6, 166 36, Czech Republic