Costas S. Iliopoulos   Alessio Langiu (Eds.)

ICABD 2014

2nd International Conference on Algorithms for Big Data

Palermo, Italy, April 7-9, 2014

Proceedings

KING'S College LONDON

© 2014 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners.

Editors' addresses:
King's College London
Department of Informatics
Strand
London WC2R 2LS | England | United Kingdom

{Costas.Iliopoulos | Alessio.Langiu}@kcl.ac.uk

Preface

In recent years, there is a growing interest in applying mathematical theories and methods (algorithms, combinatorics, codes, etc.) to describe and analyse scientific regularities of massive, complex, and fast changing data produced via Next-Generation-Sequencing technology. Various algorithms and data structures were devised to efficiently solve bioinformatics problems concerning comparing, searching, analysing, storing, compressing, and modelling this kind of data. These sequences are characterised in being massive and high-repetitive collections of nucleotides or amino acid sequences plus some metadata like quality score values.

Following the success of the Royal Society meeting on the Storage and Indexing of Massive Data, held last year in Chicheley Hall, UK, this second meeting intended to gather international researchers mainly from the fields of bioinformatics, computer science, and mathematical as well as R&D industry fellows in order to present scientific papers or survey articles on the algorithmic advancements in Big Data technology.

In this edition, sponsored by the the Algorithms and Bioinformatics Group at the Informatics Department of King's College London, UK, and the Words and Automata Research Group at Mathematics and Informatics Department of University of Palermo, Italy, 9 original research papers have been accepted for presentation and an invited talk has been given by Prof Filippo Mignosi, dealing with new mathematical theories, methodologies, algorithms, and data structures for Big Data.

We thanks all the participants and also we wish to thank all the members of the Program Committee for their collaboration in the reviewing process of the papers, and the colleagues of the Organizing Committee for their resourceful cooperation.

April 2014 Costas S. Iliopoulos, Alessio Langiu

Program Committee

Anthony J. Cox, Illumina Cambridge Ltd., UK
Maxime Crochemore, King's College London, UK, and Université Paris-Est, France
Raffaele Giancarlo, University of Palermo, Italy
Roberto Grossi, University of Pisa, Italy
Costas S. Iliopoulos, King's College London, UK, and Curtin University, Australia (Chair)
Juha Kärkkäinen, University of Helsinki, Finland
Gregory Kucherov, CNRS and Université Paris-Est, France
Alessio Langiu, King's College London, UK, and University of Palermo, Italy (Co-Chair)
Thierry Lecroq, Université de Rouen, France
Moshe Lewenstein, Bar Ilan University, Israel
Filippo Mignosi, University of L'Aquila, Italy
Antonio Restivo, University of Palermo, Italy

Organizing Committee

Carl Barton, King's College London, UK
Gabriele Fici, University of Palermo, Italy
Alessio Langiu, King's College London, UK, and University of Palermo, Italy
Giovanna Rosone, University of Palermo, Italy

Contents

Compressing Big Data: when the rate of convergence to the entropy matters
Filippo Mignosi **7**

Optimal Computation of all Repetitions in a Weighted String
Carl Barton and Solon Pissis **9**

On-line String Matching in Highly Similar DNA Sequences
Nadia Ben Nsira, Thierry Lecroq and Mourad Elloumi **16**

A Text Transformation Scheme for Degenerate Strings
Jacqueline Daykin and Bruce Watson **23**

Block Graphs in Practice
Travis Gagie, Christopher Hoobin and Simon Puglisi **30**

Compressed Spaced Suffix Arrays
Travis Gagie, Giovanni Manzini and Daniel Valenzuela **37**

On Representations of Ternary Order Relations in Numeric Strings
Jinil Kim, Amihood Amir, Joong Chae Na, Kunsoo Park and Jeong Seop Sim **46**

Engineering a Lightweight External Memory Suffix Array Construction Algorithm
Juha Kärkkäinen and Dominik Kempa **53**

Faster Average Case Low Memory Semi-External Construction of the Burrows-Wheeler Transform
German Tischler **61**

ASSP; the Antibody Secondary Structure Profile search tool
Dimitrios Vlachakis, Alexandros Armaos, Ioannis Kasampalidis, Arianna Filntisi and Sophia Kossida **69**

Compressing Big Data: when the rate of convergence to the entropy matters

Filippo Mignosi

Computer Science Department, University of L'Aquila, Italy
Filippo.Mignosi@univaq.it

Abstract

In this talk we discuss of the rate of convergence to the entropy of dictionary based compressors. A faster rate of convergence to the theoretical compression limit should correspond to better compression in practice, but constants also matters. Therefore in the analysis of the rate of convergence one must also analyse the "transient" phase.

Concerning dictionary based compressors, it is known that LZ78-alike compressors have a faster convergence than LZ77-alike compressors, when the texts to be compressed are generated by a memoryless source. In practice instead it seems that LZ77-alike performs better. This seems due to the effect of a strategy of Optimal Parsing (that can be applied in both LZ77 and LZ78 cases) rather then to the fact that the texts are generated by a memoryless source. To our best knowledge there are no theoretical results concerning the rate of convergence to the entropy of both LZ77 and LZ78 case when it is used a strategy of Optimal Parsing.

We discuss some experimental results on LZ78 that show that the rate of convergence to the entropy presents a kind of *wave* effect that become bigger and bigger as the entropy of the memoryless source decrease. It can be a *tsunami* for a zero entropy source.

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

Optimal Computation of all Repetitions in a Weighted String

Carl Barton Solon P. Pissis

Department of Informatics, King's College London, London, UK
{Carl.Barton|Solon.Pissis}@kcl.ac.uk

Abstract

A *repetition* in a string of letters consists of exact concatenations of identical factors of the string. Crochemore's repetitions algorithm, usually also referred to as Crochemore's partitioning algorithm, was introduced in 1981, and was the first optimal $\mathcal{O}(n \log n)$-time algorithm to compute all repetitions in a string of length n. A *weighted string* is a string in which a set of letters may occur at each position with respective probabilities of occurrence. In this article, we present a new variant of Crochemore's partitioning algorithm for weighted strings, which requires optimal time $\mathcal{O}(n \log n)$, thus improving on the best known $\mathcal{O}(n^2)$-time algorithm for computing *all* repetitions in a weighted string of length n.

1 Introduction

A fundamental structural characteristic of a string of letters is its periodicity. Closely related to periodicity is the notion of repetition. Repetitions in strings are highly periodic factors, that is, two or more adjacent identical factors. For instance, string `abab` is a repetition in string `aababba`. In 1981, it was shown by Crochemore that there could be $\mathcal{O}(n \log n)$ repetitions in a string of length n and an $\mathcal{O}(n \log n)$-time, thus optimal, algorithm was presented [1].

Single nucleotide polymorphisms, as well as errors from wet-lab sequencing platforms during the process of DNA sequencing, can occur in some positions of a DNA sequence. In some cases, these errors can be accurately modelled as a *don't care* letter. However, in other cases the errors can be more subtly expressed, and, at each position of the sequence, a probability of occurrence can be assigned to each letter of the nucleotide alphabet; this process gives rise to a *weighted string*.

Recently, the authors of [4] proposed an $\mathcal{O}(n^2)$-time algorithm for computing all repetitions in a weighted string x of length n. The efficiency of the proposed algorithm relies on the assumption of a given constant, the *cumulative weight threshold*, defined as the minimal probability of occurrence of factors in x.

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

Our Contribution

We present the first optimal algorithm for computing all repetitions in a weighted string. We improve on the best-known algorithm for computing all repetitions in a weighted string of length n from time $\mathcal{O}(n^2)$ to an optimal $\mathcal{O}(n \log n)$.

2 Preliminaries

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

The *concatenation* of two strings x and y is the string of the letters of x followed by the letters of y. It is denoted by xy. A string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$. Consider the strings x, y, u, and v, such that $y = uxv$. If $u = \varepsilon$, then x is a *prefix* of y. If $v = \varepsilon$, then x is a *suffix* of y. Let x be a non-empty string and y be a string. We say that there exists an *occurrence* of x in y, or, more simply, that x *occurs in* y, when x is a factor of y. Every occurrence of x can be characterised by a position in y. Thus we say that x occurs at the *starting position*.

A weighted string x on an alphabet Σ is a finite sequence of n sets. Every $x[i]$, for all $0 \leq i < n$, is a set of ordered pairs $(s_j, \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having letter s_j at position i. Formally, $x[i] = \{(s_j, \pi_i(s_j)) | s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}$. A letter s_j occurs at position i of a weighted string x if and only if the *occurrence probability* of letter s_j at position i, $\pi_i(s_j)$, is greater than 0. A string u of length m is a factor of a weighted string if and only if it occurs at starting position i with *cumulative occurrence probability* $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) > 0$. Given a *cumulative weight threshold* $1/z \in (0,1]$, we say that factor u is *valid*, or equivalently that factor u has a valid occurrence, if it occurs at starting position i and $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) \geq 1/z$.

For every string x and every natural number n, we define the nth power of the string x, denoted by x^n, by $x^0 = \varepsilon$ and $x^k = x^{k-1}x$, for all $1 \leq k \leq n$. A string is said to be *primitive* if it cannot be written as v^e, where $e \geq 2$. A repetition in x is a non-trivial power of a primitive string occurring in x. Formally, a *repetition* u^e, $e \geq 2$, in x is defined as a triple (i, p, e) such that: $u = x[i \mathinner{.\,.} i+p-1] = x[i+p \mathinner{.\,.} i+2p-1] = \ldots = x[i+(e-1)p \mathinner{.\,.} i+ep-1]$; u^{e+1} does not occur at position i; and u is primitive. A repetition is maximal if $i - p < 0$ or u^e does not occur at $x[i-p]$. The integers p and e are called the *period* and the *exponent* of the repetition, respectively. If $e = 2$ the repetition is called *square*.

A *repetition* $v = u^e$, $e \geq 2$, in a weighted string x is defined as a quadruple (i, p, b, e) such that $u = v[0 \mathinner{.\,.} p-1] = v[p \mathinner{.\,.} 2p-1] = \ldots = v[(e-1)p \mathinner{.\,.} ep-1]$, where v is a factor of length ep of x occurring at position i, and each occurrence of u in v is a valid factor of x; u^{e+1} does not occur at position i; u is primitive; and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$. A

repetition is maximal if $i - p < 0$ or u^e does not occur at $x[i - p]$. In this article, we are mainly concerned with the following problem.

Problem 2.1 *Given a weighted string x of length n and a cumulative weight threshold $1/z$, find all repetitions in x.*

3 Algorithm

We first perform a colouring stage on x, similar to the one before the construction of the weighted suffix tree [3], which assigns a colour to every position in x according to the following scheme: mark position i *black*, if *none* of the possible letters at position i has probability of occurrence greater than $1 - 1/z$; mark position i *grey*, if *one* of the possible letters at position i has probability of occurrence greater than $1 - 1/z$; mark position i *white*, if *one* of the possible letters at position i has probability of occurrence 1.

Lemma 3.1 ([3]) *A valid factor of x contains at most $\lceil \log z / \log(\frac{z}{z-1}) \rceil$ black positions.*

We then perform a generation stage, as the one performed during the construction of the weighted suffix tree, where a set of factors of x is generated. We refer to this set as *extended factors* (for a definition, see [3]).

Lemma 3.2 ([3]) *A valid factor of x occurs in at least one of its extended factors.*

An *extended repetition* is a repetition occurring in an extended factor of x. A *valid repetition* $v = u^e$, $e \geq 2$, in x is defined as a quadruple (i, p, b, e) such that $u = v[0 \,..\, p - 1] = v[p \,..\, 2p - 1] = \ldots = v[(e - 1)p \,..\, ep - 1]$, where v is a *valid* factor of length ep of x occurring at position i; u^{e+1} is not a valid factor of x; u is primitive; and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$. We define the following subproblem.

Problem 3.3 *Given a weighted string x of length n and a cumulative weight threshold $1/z$, find all valid repetitions in x.*

Lemma 3.4 *Every valid repetition in x occurs in at least one extended factor.*

For each generated extended factor, we run Crochemore's partitioning algorithm for maximal repetitions; the result is all the maximal extended repetitions in x. After computing all the maximal extended repetitions we cannot simply report all of these as valid repetitions. All valid factors must occur in an extended factor but extended factors may contain factors which are not valid. This is a consequence of treating grey positions as white during the generation of extended factors [3]. Since not all maximal extended repetitions are valid repetitions, we must therefore break up these maximal extended repetitions into valid repetitions to solve Problem 3.3.

In order to break up the maximal extended repetitions, we must compute some additional information. To determine how long any valid repetition should be, we must know, for each position i in an extended factor, the length of the longest valid factor starting at position i. The computation is based on the observation that

the longest factor with probability greater than or equal to $1/z$ for the position $i+1$ has length greater than or equal to that of position i. To compute this, we maintain an additional cumulative weight threshold π. We store the computed lengths in an array LF of integers.

We start with the first position in an extended factor and naively compute the longest factor within the threshold by multiplying together the probability of the letters we encounter and storing this in π. If multiplying the probability of some position $j > 0$ causes $\pi < 1/z$ we set $\mathsf{LF}[0] := j-1$. To proceed, we remove by division the occurrence probability of the first letter from π. If $\pi < 1/z$ then $\mathsf{LF}[1] = j-1$; otherwise, we continue as before multiplying the probability of $j+1, j+2$, and so on, until the threshold is once again violated. For each extended factor this takes time and space proportional to its length. The sum of lengths of the extended factors is linear in n by the following statement.

Lemma 3.5 ([3]) *The sum of lengths of the extended factors of x is $\mathcal{O}(n)$.*

The next step is to determine the set b for each maximal extended repetition. This can be done in constant time per maximal extended repetition. We compute an array NB of integers of size n, such that for each position i in x, $\mathsf{NB}[i]$ stores the index of the leftmost black position $j > i$; this can be done in linear time in n. For each maximal extended repetition u^e, we check all black positions in the first occurrence of u. There can only be a constant number of black positions in u; finding the black positions using NB takes time proportional to their number. It is now a simple case of recording the position and the letter present in the extended factor; this takes constant time per maximal extended repetition, so time proportional to the number of maximal extended repetitions in total.

Given all the maximal extended repetitions, we can now begin to break them up into valid repetitions. To achieve this, we can check the length of the longest factor starting at position i of the extended factor, and then determine the longest possible repetition starting from i. We can continue checking the maximal extended repetition in this manner reporting the length as we go. Note that in the worst case, for each maximal extended repetition u^e, we may check the starting position of each occurrence of u. As we show later (Lemma 3.7), this can be done efficiently. We now establish the maximal number of extended repetitions in x. Note that the work done by the algorithm so far is no more than the maximal number of extended repetitions.

Lemma 3.6 *There could be $\mathcal{O}(n \log n)$ extended repetitions in x.*

As previously mentioned, whilst breaking some maximal extended repetition u^e into valid repetitions, we may need to check up to e positions. The maximum number of checks required will be the sum of the exponents of all maximal extended repetitions returned by the partitioning. Now we establish the maximal sum of the exponents of maximal extended repetitions in a weighted string.

Lemma 3.7 *The sum of exponents of maximal extended repetitions in x is $\mathcal{O}(n \log n)$.*

Note that an analogous version of Lemma 3.6 holds for valid repetitions.

Theorem 3.8 *Problem 3.3 can be solved in optimal time $\mathcal{O}(n \log n)$.*

At this point, we have solved the subproblem which forms the basis for our solution. Intuitively, the subproblem finds repetitions $v = u^e$, where factor v occurs with probability $\geq 1/z$. The idea behind our solution to Problem 2.1 is based on the observation that a repetition of exponent $e \geq 3$ is composed of overlapping occurrences of smaller repetitions. We intend to compute smaller repetitions and, from this, derive larger ones. Part of the process of computing valid repetitions was to break up maximal extended repetitions below the threshold into smaller valid repetitions. To determine the repetitions specified in Problem 2.1, we reverse this process and *compose* longer repetitions from small valid repetitions.

In order to solve Problem 2.1, we start by solving Problem 3.3 for threshold $k = 1/z^2$. The number of valid repetitions reported for k can be shown to be $\mathcal{O}(n \log n)$ by the same argument as for Lemma 3.6; and the number of black positions in a valid factor is only a constant amount higher than for the original threshold by a similar argument to the proof of Lemma 3.1. We pick $k = 1/z^2$ as we wish to guarantee that we will at least find squares such that each half may have probability greater than or equal to $1/z$. We may also find repetitions with a higher exponent and repetitions which have a probability less than $1/z$, but we will explain how to filter these out using the same techniques as for Problem 3.3.

We alter the solution to Problem 3.3 to simplify the solution to Problem 2.1. Instead of breaking up maximal extended repetitions into valid repetitions, we break them into all their valid overlapping squares. There are no more than $\mathcal{O}(n \log n)$ valid squares by [2]. This can be shown by an almost identical argument as Lemma 3.6. To split maximal extended repetitions into their valid overlapping squares, we process them one by one and create a new square for each overlapping square in the maximal extended repetition. We only need to perform this on maximal extended repetitions of exponent $e \geq 3$, and this will take time proportional to the sum of the exponents which, by Lemma 3.7, is $\mathcal{O}(n \log n)$.

To perform the filtering step, we must check if both halves of the square are above the threshold $1/z$. To check each half, we compute, for each position i in an extended factor, the length of the longest valid factor starting at position i. During the generation of extended factors for the threshold k, we at the same time determine the longest factor with probability greater than or equal to $1/z$ by computing an array LF' which stores the analogous information. Filtering the squares in time proportional to their number can be done by checking that the length stored in the array is greater than or equal to the period of the square.

After the filtering step, we have a set of quadruples (i, p, b, e) representing all primitive squares such that each half of a square has a probability of occurrence at least $1/z$. Now, for every position i in x, we declare an array A_i of linked lists, such that the linked list $\mathsf{A}_i[f_i(j)]$, $f_i : [1, \lfloor n/2 \rfloor] \to [0, \mathcal{O}(\log_\phi n)]$, stores all the squares which occur at position i with period $j \in [1, \lfloor n/2 \rfloor]$. We now wish to establish the size of A_i and the size of the linked lists stored at any $\mathsf{A}_i[f_i(j)]$, but first we state a property of valid factors required to show properties of A_i.

Lemma 3.9 *A valid factor of x is in $\mathcal{O}(1)$ extended factors of x.*

Lemma 3.10 *A_i is of size $\mathcal{O}(\log_\phi n)$, where $\phi = (1+\sqrt{5})/2$, and the size of any linked list $\mathsf{A}_i[f_i(j)]$ is $\mathcal{O}(1)$.*

We can now construct the repetitions specified in Problem 2.1. For each position i, we iterate through the linked lists of array A_i. We iterate through each linked list $\mathsf{A}_i[f_i(p)]$, where p is the considered period. We process each square element $(i, p, b, e) \in \mathsf{A}_i[f_i(p)]$ to extend the corresponding square as much as possible, by checking for an occurrence of the square at position $i + p$. For a linear string, it is simple to determine this. For each pair of overlapping squares, the second half of the first square is the first half of the second square; so it suffices to check whether there exists a square at position $i + p$ with the same period.

Example Consider y = `ababab` that contains the following primitive squares: $(0, 2, 2)$, $(1, 2, 2)$, and $(2, 2, 2)$; we wish to find the repetition $(0, 2, 3)$. We start at position 0 of y with $(0, 2, 2)$ and check if there is a square of period 2 starting at position 2. A matching square exists so we extend the repetition and check position 4. There is no square at position 4 so we report the repetition $(0, 2, 3)$.

For weighted strings the approach is very similar, with the addition of a few, constant-time, checks. We must check, for each pair of overlapping squares, if the black positions from the first square match with the black positions from the second square. There is a constant number of black positions so this takes constant time. Each time we find such overlapping squares, we extend our repetition and delete the square at position $i + p$ from the corresponding list. As soon as we find a position where we cannot extend the repetition we stop. We continue doing this until we have found all repetitions.

Each time we iterate through a linked list, a square may be added to the repetition we are extending; this takes constant time per list by Lemma 3.10. After each square is added to the repetition, it is deleted so is not considered again. There are $\mathcal{O}(n \log n)$ squares in the array and from the above description we can see that each square is considered a constant number of times. It is clear that we construct no more repetitions than there are primitive squares, so the number of constructed repetitions is also $\mathcal{O}(n \log n)$. These repetitions will be maximal, and to report repetitions specified in Problem 2.1, we may check the start of each occurrence in the repetition and report them. This takes no more than the sum of exponents which is $\mathcal{O}(n \log n)$. We can now state the main result of this article.

Theorem 3.11 *Problem 2.1 can be solved in optimal time $\mathcal{O}(n \log n)$.*

References

[1] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[2] M. Crochemore, L. Ilie, and W. Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227 – 5235, 2009. Mathematical Foundations of Computer Science (MFCS 2007).

[3] C. S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inf.*, 71(2,3):259–277, Feb. 2006.

[4] H. Zhang, Q. Guo, and C. S. Iliopoulos. Locating tandem repeats in weighted sequences in proteins. *BMC Bioinformatics*, 14(S-8):S2, 2013.

On-line String Matching in Highly Similar DNA Sequences

Nadia Ben Nsira[1,2], Thierry Lecroq[1,*], Mourad Elloumi[2]

[1]LITIS EA 4108, Normastic FR3638, University of Rouen, France
[2]LaTICE, University of Tunis El Manar, Tunisia
[*]Thierry.Lecroq@univ-rouen.fr

Abstract

We consider the problem of on-line exact string matching of a pattern in a set of highly similar sequences. This can be useful in cases where indexing the sequences is not feasible. We present a preliminary study by restricting the problem for a specific case where we adapt the classical Morris-Pratt algorithm to consider borders with errors. We give an original algorithm for computing borders at Hamming distance 1. We exhibit experimental results showing that our algorithm is much faster than searching for the pattern in each sequences with a very fast on-line exact string matching algorithm.

1 Introduction

High-throughput sequencing or *Next Generation Sequencing* (NGS) technologies allow to produce a great amount of DNA sequences with a high rate of similarity. For instance, the 1000 genomes project[1] aimed at sequencing a large amount of individual whole human genomes. This generates massive amounts of sequences (3 billion letters A, C, G, T) which are identical more than 99% to the reference human genome. The generated data form a collection of sequences where each differs from another by a few number of differences such as *substitutions* or *single nucleotide variants* (SNVs), *indels*, *copy number variations* (CNVs) or *translocations* to name a few.

With the large mass of available data, storing, indexing and support for fast pattern matching have become important research topics.

Pattern matching can be carried out in two ways: off-line by using an index or on-line when indexing is not possible. Although the first kind of solutions seems

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

[1] http://www.1000genomes.org

to be more suitable, the index issue might not seem significant in some cases even if it is compressed. The main problem we can face is to have insufficient storage space to build the index. Thus one may have to scan the whole sequence rather than index it.

In this paper, we focus on offering a preliminary study that allows to find the exact occurrences of a given pattern of length m in a set of highly similar sequences. We propose a solution that follows a tight analysis of the Morris-Pratt algorithm. We point out occurrences of the pattern by performing a left to right traversal over the reference sequence and at the same time we take into account variations contained in other sequences. Our approach makes a simplistic assumption that sequences include variations only of type *substitutions* and that there exists at most only one variation in a window of length m.

The rest of the paper is organized as follows. Section 2 presents related works. We set up notations and formalize the problem in Sect. 3. We give our new algorithm in Sect. 4 while experimental results are exhibited in Sect. 5. Finally we give our conclusions in Sect. 6.

2 Related work

Storing genetic sequences of many individual of the same species is a major challenge in biological research. Basic structures usually store redundant information which lead to a memory requirement proportional to the total length of input data. Recently, several works focusing on indexing similar sequences were implemented to allow building data structures taking advantage from high similarity between the considered data. These works aim to reduce the memory requirement from the length of all input sequences to the length of a single sequence (reference sequence) plus the number of variations.

Huang *et al.* [10] propose a solution which assumes that the input set of DNA sequences can be divided into common segments and non-common segments. Their solution assumes that every sequence differs on m' positions from the reference and the designed data structure requires $O(n \log \sigma + m' \log m')$ bits where n is the length of the reference sequence and σ is the size of the alphabet. Though this data structure greatly reduces the memory usage and allows fast pattern matching, the adopted model is restricted to a specific type of similar sequences. In [3] a solution based on the use of word level operations on bit vectors is presented. In similar way as [10], the general scheme of this technique store the entire of a reference sequence with only differences between the remaining sequences. The authors build a suffix array together with an Aho-Corasick automaton [1] to store identical segments and the non-common segments are converted into a binary word using 2 bits per base. Due to the use of the Aho-Corasick the memory usage depends on a $\log n$ factor. In [7] a compressed index is proposed based on the Lempel-Ziv compression scheme [12]. Both [6, 2] propose 2 level indexes for highly repetitive sequences. In [6] the authors implement an index based on suffix tree and traditional q-grams. The concept of the *suffix tree of alignment* was proposed by [14]. It satisfies the same properties as the classical generalized suffix tree by adding a new one: common suffixes of two sequences are stored in an identical leaf. This result has been

extended to the suffix array of an alignment [15].

All these results are concerned with off-line string matching but to the best of our knowledge there exists no result for on-line string matching in a set of highly similar sequences.

3 Preliminaries

In what follows, we consider a finite *alphabet* Σ={A, C, T, G} for DNA sequences. A *string* or a *sequence* is a succession of zero or more symbols of the alphabet. The empty string is denoted by ε. The set of all non empty strings over Σ is denoted by Σ^+. All strings over the alphabet Σ are element of $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$. The string w of length m is represented by $w[0\,..\,m-1]$ where $w[i] \in \Sigma$ and $0 \leq i \leq m-1$. The length of w is denoted by $|w|$.

A string x is a *factor* (substring) of y if there exist u and v such $y = uxv$, where $u, v, x, y \in \Sigma^*$. Let $0 \leq j \leq m-1$ be the starting position of x in y, thus $x = y[j\,..\,j+|x|-1]$. A factor x is a *prefix* of y if $y = xv$, $v \in \Sigma^*$. Similarly a factor x is a *suffix* of y if $y = ux$, for $u \in \Sigma^*$. A factor u is a *border* of x, if it is both a prefix and a suffix of x, then there exist $v, w \in \Sigma^*$ such $x = vu = uw$. The *reverse* of the string x is denoted by $x^\sim$. The longest common prefix between two strings u and v is denoted by $lcp(u, v)$.

The exact pattern matching problem consists in finding all the occurrences of a pattern x in a string y. That is, all possible j such that $y[j+i] = x[i]$ holds for all $0 \leq i \leq m-1$. This problem can be extended in a very interesting way by considering a set of sequences and find whether a given pattern occurs distributed horizontally where different parts of the pattern can be located in consecutive positions of different texts. More formally, given a set of sequences $Y = \{y_0, \ldots, y_{r-1}\}$ of equal length n, point out all positions $0 \leq j \leq n-m+1$, such that for $0 \leq i \leq m-1$ we have $x[i] = y_g[j+i]$ for some $g \in [0; r-1]$. This latter problem is known as distributed pattern matching [11].

The problem we focus on in this paper is formally defined as follows. Let $y_0, y_1, \ldots, y_{r-1}$ be r highly similar sequences with the same length n defined over the alphabet Σ. Let y_0 be the reference sequence. The sequences $y_1, y_2, \ldots, y_{r-1}$ are represented by variations over y_0. Thus, we consider the set $Z = \{(\mathcal{G}, j, c)\}$, such that $c = y_g[j] \neq y_0[j]$ for all $0 \leq j \leq n-1$, $g \in \mathcal{G}$ where $1 \leq g \leq r-1$ and $c \in \Sigma$. Furthermore, for $(\mathcal{G}, j, c), (\mathcal{G}', j', c') \in Z$ we have $|j - j'| > M$ for some integer M. We wish to find all occurrences of an arbitrary pattern x of length $m \leq M$ in y_g where $0 \leq g \leq r-1$. This problem can be viewed as an hybrid between distributed pattern matching and approximate string matching with k mismatches [4].

4 A new algorithm

We offer an algorithm to solve the problem described above in the same fashion as the Morris-Pratt (MP) algorithm [13] using a sliding window mechanism to scan the text. Hence we need to preprocess the query pattern before the search phase. We adopt the same strategy of *forward prefix scan* presented by MP by extending

the problem to the search in highly similar data. For that we need to consider borders at Hamming distance 0 (as in MP) and borders at Hamming distance 1.

Given the pattern x of length m, we consider three cases when a prefix $x[0\,..\,i]$ for $0 \le i \le m-1$, is recognized when scanning the r sequences at position j:

Case 1 $x[0\,..\,i] = y_0[j\,..\,j+i]$ and $\not\exists(\mathcal{G}, k, c) \in Z$ such that $j \le k \le j+i$. This means that $x[0\,..\,i]$ matches on y_0 and there is no variation in all others sequences in the current window then $x[0\,..\,i]$ matches equally in all sequences.

Case 2 $x[0\,..\,i] = y_0[j\,..\,j+i]$ and $\exists(\mathcal{G}, k, c) \in Z$ such that $j \le k \le j+i$. This means that $x[0\,..\,i-1]$ matches all sequences except y_g.

Case 3 $x[0\,..\,i] = y_g[j\,..\,j+i]$ and $\exists(\mathcal{G}, k, c) \in Z$ such that $j \le k \le j+i$ and $g \in \mathcal{G}$. Then $x[0\,..\,i-1]$ matches only sequence y_g.

4.1 Preprocessing phase

The preprocessing phase consists in precomputing arrays storing positions of borders for each prefix of the pattern. Borders at Hamming distance 0 are computed as in the MP algorithm and stored in an array called *mpNext*. For borders at Hamming distance 1 we will use the two following arrays.

The classical array pref_x is the array of prefixes of the pattern x: $\mathit{pref}_x[i]$ is the length of the longest prefix of x starting at position i, for $0 \le i \le m-1$.

The array $\mathit{pref}_x^{\sim}$ stores for each position i, the length of the longest common prefix starting at position i when reading the pattern from right to left with each position $i' < i$ where $\mathit{lcp}(x[0\,..\,i]^{\sim}, x[0\,..\,i']^{\sim}) \neq 0$. It is defined for all $0 \le i \le m-1$ and $i' < i$ by $\mathit{pref}_x^{\sim}[i] = \{(i',\ \ell) \mid i' < i, x[i] = x[i'] \text{ and } 0 < \ell = |\mathit{lcp}(x[0\,..\,i]^{\sim},\ x[0\,..\,i']^{\sim}|)\}$. Then $\mathit{pref}_x^{\sim}[i+1]$ can be easily computed from $\mathit{pref}_x^{\sim}[i]$.

For $0 \le i \le m$, $B[i]$ contains the suffix starting positions of borders of $x[0\,..\,i]$ with one change. More formally $B[i] = \{i' \mid \mathit{Ham}(x[0\,..\,i-i'], x[i'\,..\,i]) = 1\}$.

Proposition 4.1 *For* $1 \le i \le m-1$, *if* $i' \in B[i]$ *then* $x[0\,..\,i-i']$ *is a border of* $x[0\,..\,i' + \mathit{pref}_x[i'] - 1]x[\mathit{pref}_x[i']]x[i' + \mathit{pref}_x[i'] + 1\,..\,i]$.

Then $B[i] = \{i' \mid \exists(i-i',\ \ell) \in \mathit{pref}_x^{\sim}[i] \text{ and } i-i' = \mathit{pref}_x[i'] + \ell\} \cup \{i \mid \mathit{pref}_x[i] = 0\}$.

Proposition 4.2 *The number of elements in the array* B *is* $O(m)$.

4.2 Searching phase

The searching phase consists in scanning the sequences from beginning to end. A general situation is the following: a prefix $x[0\,..\,i]$ of the pattern matches at least one sequence of Y at position j. Then position $j+1$ is scanned and a decision has to be made in accordance with the three cases.

Case 1 If $x[i+1] = y_0[j+i+1]$ then if there is no variation at position $j+i+1$ in the other sequences we remain in Case 1 otherwise we move to case 2. If $x[i+1] \neq y_0[j+i+1]$ then if there is no variation in the other sequences at

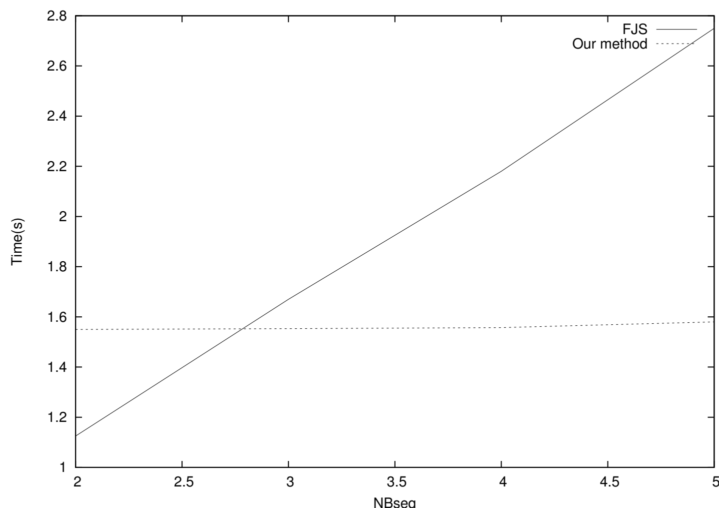

Figure 1: Experimental results: our algorithm vs the FJS algorithm.

position $j + i + 1$ then a shift is performed as in the MP algorithm otherwise if the variant symbol c is equal to $x[i+1]$ we move to Case 3 otherwise a shift has to be performed using $mpNext$ and B.

Case 2 If $x[i+1] = y_0[j+i+1]$ then we remain in Case 2. If $x[i+1] \neq y_0[j+i+1]$ then a shift has to be performed using B.

Case 3 If $x[i+1] = y_0[j+i+1]$ then we remain in Case 3. If $x[i+1] \neq y_0[j+i+1]$ then a shift has to be performed using B.

When a shift is performed using B, the case can change according to the length of the shift and the position of the variation.

Theorem 4.3 *The searching phase runs in* $O(n)$ *time.*

5 Experiments

We use simulated data of length 150 MB and mutation rate 0.25% among them 30% to 50% are at the same position in different sequences. We first generate a reference text, then mutate it at random positions with random nucleotides. We compare our solution with FJS [9] which is one of the most efficient exact string matching algorithm in this setting (see [8]). For the patterns, we randomly select them from the reference text with varying length from 8 to 128. For each pattern length, we repeat tests 100 times and compute the average searching time. Experiments were conducted on a machine with 12 GB RAM and 4-core CPU with 2.27 GHz.

As mentioned above our algorithm takes into account the reference sequence with positions of variations. We consider variations every 500 positions. For FJS algorithm we launch the execution texts one by one. We ran the FJS algorithm on

each sequence successively. Our goal is to demonstrate that from a certain number of sequences, it is more efficient to use our solution than a classical exact string matching solution. Results are shown in Figure 1 and show that our solution is faster (in our settings) when considering more that 3 highly similar sequences.

6 Discussion and conclusions

We have presented a new algorithm for searching in a set of similar sequences. The design of the algorithm follows a tight analysis of the Morris and Pratt algorithm. Recall that we use a suitable representation of data such that we take into account a reference sequence with only positions of variations of the other sequences. The searching time of our algorithm depends on the size of the reference text. However, our solution works with a particular model, since we are limited to a certain gap between consecutive variations. We will to improve our algorithm to overcome this point. On the other hand, we aim to use variants of the Boyer-Moore algorithm [5] for greater efficiency. The next steps include to take into account any kind of variation between the reference sequence and the other sequences and to be able to perform approximate pattern matching.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] A. Alatabbi, C. Barton, and C. S. Iliopoulos. On the repetitive collection indexing problem. In *IEEE International Conference on Bioinformatics and Biomedicine Workshops*, pages 682–687, 2012.

[3] A. Alatabbi, C. Barton, C. S. Iliopoulos, and L. Mouchard. Querying highly similar structured sequences via binary encoding and word level operations. In *Artificial Intelligence Applications and Innovations*, pages 584–592. Springer, 2012.

[4] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[6] X. Cao, S. C. Li, and A. K. Tung. Indexing DNA sequences using q-grams. In *Database Systems for Advanced Applications*, pages 4–16. Springer, 2005.

[7] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative lempelziv self-index for similar sequences. *Theoretical Computer Science*, 2014. To appear.

[8] S. Faro and T. Lecroq. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, 45(2):13, 2013.

[9] F. Franek, C. G. Jennings, and W. F. Smyth. A simple fast hybrid pattern-matching algorithm. *Journal of Discrete Algorithms*, 5(4):682–695, 2007.

[10] S. Huang, T. Lam, W. Sung, S. Tam, and S. Yiu. Indexing similar DNA sequences. In *Algorithmic Aspects in Information and Management*, pages 180–190. Springer, 2010.

[11] C. S. Iliopoulos, L. Mouchard, and M. S. Rahman. A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. *Mathematics in Computer Science*, 1(4):557–569, 2008.

[12] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *String Processing and Information Retrieval*, pages 201–206. Springer, 2010.

[13] J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

[14] J. C. Na, H. Park, M. Crochemore, J. Holub, C. S. Iliopoulos, L. Mouchard, and K. Park. Suffix tree of alignment: An efficient index for similar data. In *Internat. Workshop On Combinatorial Algorithms*, pages 337–348, 2013.

[15] J. C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, and K. Park. Suffix array of alignment: A practical index for similar data. In *String Processing and Information Retrieval*, pages 243–254. Springer, 2013.

A Text Transformation Scheme For Degenerate Strings

Jacqueline W. Daykin[1,2], Bruce Watson[1,3]

[1]Department of Informatics, King's College London, UK
[2]Department of Computer Science, Royal Holloway, University of London, UK
[3]Information Science Department, Stellenbosch University, South Africa
Jackie.Daykin@kcl.ac.uk bwwatson@sun.ac.za

Abstract

The Burrows-Wheeler Transformation computes a permutation of a string of letters over an alphabet, and is well-suited to compression-related applications due to its invertability and data clustering properties. For space efficiency the input to the transform can be preprocessed into Lyndon factors. We consider scenarios with uncertainty regarding the data: a position in an *indeterminate* or *degenerate* string is a set of letters. We first define *Indeterminate Lyndon Words* and establish their associated unique string factorization; we then introduce the novel *Degenerate Burrows-Wheeler Transformation* which may apply the indeterminate Lyndon factorization. A core computation in Burrows-Wheeler type transforms is the linear sorting of all conjugates of the input string - we achieve this in the degenerate case by applying lex-extension ordering. Indeterminate Lyndon factorization, and the degenerate transform and its inverse, can all be computed in linear time and space with respect to total input size of degenerate strings.

1 Introduction

This paper focuses on strings involving uncertainty – such strings are known as *indeterminate*, or equivalently, *degenerate* strings and consist of nonempty *subsets* of letters over an alphabet Σ[1] Algorithms for indeterminate strings have been described in [10].

Motivation for degenerate strings arises from applications such as interface data entry and bioinformatics. With degenerate biological strings, nucleotide sequences

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

[1]Terminology: *indeterminate* is common in theoretical computer science; *degenerate* is used in molecular biology.

are often written using the five letter alphabet $\{A, T, G, C, N\}$, where N denotes an unspecified nucleotide. For instance, $ANTAG$ may correspond to four different interpretations: $AATAG$, $ATTAG$, $AGTAG$ and $ACTAG$. Such degenerate strings can express polymorphisms in DNA/RNA sequences. Longest common subsequence computations apply to determining the homology of two biological sequences [20]; pattern matching techniques honed to degenerate DNA/RNA sequences are designed in [11].

A *Lyndon word* is defined as a (generally) finite word which is strictly minimal for the lexicographic order of its conjugacy class; the set of Lyndon words permits the unique maximal factorization of any given string [3].

In 1994, Burrows and Wheeler [2] introduced a transformation for textual data demonstrating, not only data clustering properties, but also suitability for block sorting compression. The Burrows-Wheeler Transform (BWT) operates by permuting the letters of a given text to obtain a modified text which may be more suitable for compression – the transform is therefore used by many text compression or compression-related applications, and some self-indexing data structures [1]. Space saving techniques with the BWT can be achieved by first factoring the input text or string into Lyndon words [13].

In Next-Generation Sequencing (NGS), large unknown DNA sequences are fragmented into small segments (a few dozen to several hundreds of base pairs long). This process generates masses of data, typically several million "short reads". Alignment programs attempt to align or match these reads to a reference genome; alignment was initially performed by applying hashing or the suffix tree/array data structures – subsequently, efficiency in memory requirement was achieved by using the BWT. Motivated by the degeneracy associated with genome sequencing, we introduce here a collection of novel and related concepts: a linear *Degenerate Burrows-Wheeler Transform*, an *Indeterminate Suffix Array*, *Indeterminate Conjugacy* and *Indeterminate Lyndon Words*.

2 Definitions and Preliminaries

A *string* (*word*) is a sequence of zero or more characters or letters over a totally ordered alphabet Σ. The set of all non-empty strings over Σ is denoted by Σ^+. The empty string is indicated by $\boldsymbol{\varepsilon}$; we write $\Sigma^* = \Sigma^+ \cup \boldsymbol{\varepsilon}$. Strings will be identified in mathbold such as $\boldsymbol{w}$, $\boldsymbol{x}$. We will use standard terminology from stringology: border, border-free, prefix, suffix, primitive, conjugate, etc. – see [19].

An *indeterminate* string $\boldsymbol{x} = \boldsymbol{x}[1 \dots m]$ on an alphabet Σ is a sequence of nonempty subsets of Σ; $\boldsymbol{x}$ is equivalently known as a *degenerate* string. Specifically, an indeterminate string $\boldsymbol{x}$ has the form $\boldsymbol{x} = \boldsymbol{x_1 x_2} \cdots \boldsymbol{x_m}$, where each $\boldsymbol{x_i}$ is a set of letters over Σ, and while $|\boldsymbol{x}| = m$, computationally we will be accounting for the total size of the string, that is $||\boldsymbol{x}|| = n = \sum_{i=1}^{m} |\boldsymbol{x_i}|$; if some $|\boldsymbol{x_i}| = 1$ then this is the usual case of a single letter in a string denoted as x_i. So a typical instance of a degenerate string may have the form $\boldsymbol{u} = \boldsymbol{u_1} u_2 u_3 \boldsymbol{u_4} u_5 \cdots \boldsymbol{u_{m-1}} u_m$; in a regular string all sets are unit size. Moreover, with degeneracy we can allow the $\boldsymbol{x_i}$ to be multisets. We also write the sets in degenerate strings in mathbold (unless they

are known to be unit size) - there is no ambiguity as regular and degenerate strings are used in different contexts here.

3 The Burrows-Wheeler Transform

The Burrows-Wheeler text transformation scheme was invented by Michael Burrows and David Wheeler in 1994 [2], and has become widely applied [1].

The basic BWT algorithm permutes an input string T (text) of n characters into a transform in three conceptual stages: first the n rotations (cyclic rotations or conjugates) of T are formed; these rotations are then sorted lexicographically giving the $n \times n$ BWT matrix M; finally the last (right-most) character of each of the rotations, that is the last column of the matrix M, is extracted into a string L (last). In addition to L, the algorithm computes the index i of the occurrence of the original text T in the sorted list of rotations. The pair (L, i) is known as the *transform*, that is $\mathrm{BWT}(T) = (L, i)$. Furthermore, the BWT can be constructed efficiently since the heart of the computation is sorting the rotations which, by applying a fast suffix-sorting technique such as [12], can be achieved in linear time. It is the data clustering properties of this transform, usually exhibiting long runs of identical characters, together with the fact that it is invertible, that has sparked so much interest.

Given only L and the index i, the original text T can be reconstructed in linear time [2]. Observe that the first column F of M can be obtained by lexicographically sorting the characters of L. By constructing a Hamiltonian cycle of L and F, the Last-First Mapping, the input can be recovered.

A simple observation shows that, since by definition a Lyndon word is the strictly least amongst its conjugates, if the input text forms a Lyndon word, then the index i will be 1 and therefore redundant, thus offering a space saving of $O(\log n)$ bits. Accordingly, BWT variants have been considered: Scott followed by Kufleitner introduced the bijective Multi-Word BWT; Kufleitner also proposed the bijective Sort Transform initiated earlier by Schindler − these variants are based on the Lyndon factorization of the input [9, 13, 18].

The BWT has also been implemented in bioinformatics: to reduce the memory requirement with hashing-based sequence alignment, BWT-based alignment utilities were developed including SOAP2 [15] and BOWTIE [14].

4 Indeterminate Lyndon Words

A *Lyndon word* is a primitive and border-free word which is strictly minimal for the lexicographical order of its conjugacy class [17] – let $\mathcal{L}$ denote the set of Lyndon words over the totally ordered alphabet Σ. These patterned words exhibit many interesting properties [16], including:

Proposition 4.1 *[8] A word $\boldsymbol{w} \in \Sigma^+$ is a Lyndon word if and only if it is lexicographically less than each of its nonempty proper suffixes.*

Proposition 4.2 *[8] A word $\boldsymbol{w} \in \Sigma^+$ is a Lyndon word if and only if either $\boldsymbol{w} \in \Sigma$ or $\boldsymbol{w} = \boldsymbol{uv}$ with $\boldsymbol{u}$, $\boldsymbol{v} \in \mathcal{L}$, $\boldsymbol{u} < \boldsymbol{v}$.*

Importantly, the set $\mathcal{L}$ of Lyndon words permits the unique maximal factorization of any given string, hence useful for applications.

Theorem 4.3 *[3] Any word $\boldsymbol{w} \in \Sigma^+$ can be written uniquely as a non-increasing product $\boldsymbol{w} = \boldsymbol{u}_1\boldsymbol{u}_2\cdots\boldsymbol{u}_k$ of Lyndon words.*

In 1983, Duval [8] developed an algorithm for factorization that runs in linear time and constant space.

We now introduce the set $\mathcal{IL}$ of *Indeterminate Lyndon Words* – given an indeterminate string $\boldsymbol{x} = \boldsymbol{x_1}\boldsymbol{x_2}\cdots\boldsymbol{x_m}$, the first step in defining these new Lyndon words is to assign an order to each of the sets $\boldsymbol{x_i}$ (which are not necessarily distinct). So for each $1 \leq i \leq m$, let $\underline{\boldsymbol{x_i}}$ denote the lexicographic ordering of $\boldsymbol{x_i}$ (the letters are lined up in the given alphabet order) written as a string. For example, if $\boldsymbol{x_i} = \{c, a, t, g\}$ then $\underline{\boldsymbol{x_i}} = acgt$. Hence, under the convention that the order of elements in a set doesn't matter, we have a bijective mapping $\mathcal{G} : \boldsymbol{x_i} \rightarrow \underline{\boldsymbol{x_i}}$ for $1 \leq i \leq m$, or simply $\mathcal{G} : \boldsymbol{x} \rightarrow \underline{\boldsymbol{x}}$. Furthermore, we can allow multisets under this mapping. Note that if $||\boldsymbol{x}|| = n$, and if we assume an integer alphabet, that is, if the range of letters in the alphabet is $O(n)$, an array of length $|\Sigma|$ suffices to map the given alphabet onto an integer alphabet $\{1, 2, ..., k\}, k \leq n$. Therefore each of the sets $\boldsymbol{x_i}$ can be sorted in time $O(|\boldsymbol{x_i}|)$; hence the total time to compute $\underline{\boldsymbol{x}}$ is $O(n)$.

We can now state a required definition, *lex-extension order*, for the lexicographic order of given indeterminate strings $\boldsymbol{u}, \boldsymbol{v}$ over Σ mapped to $\underline{\boldsymbol{u}}, \underline{\boldsymbol{v}}$.

Definition [4, 6] Suppose that according to some factorization $\mathcal{F}$, two strings $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^+$ are expressed in terms of nonempty factors:
$\boldsymbol{u} = \boldsymbol{u}_1\boldsymbol{u}_2\cdots\boldsymbol{u}_m, \boldsymbol{v} = \boldsymbol{v}_1\boldsymbol{v}_2\cdots\boldsymbol{v}_n$. Then $\boldsymbol{u} <_{LEX(\mathcal{F})} \boldsymbol{v}$ if and only if one of the following holds:
(1) $\boldsymbol{u}$ is a proper prefix of $\boldsymbol{v}$ (that is, $\boldsymbol{u_i} = \boldsymbol{v_i}$ for $1 \leq i \leq m < n$); or
(2) for some $i \in 1..min(m, n), \boldsymbol{u_j} = \boldsymbol{v_j}$ for $j = 1, 2, ..., i-1$, and $\boldsymbol{u_i} < \boldsymbol{v_i}$ (in lexicographic order).

In the case of an indeterminate string $\boldsymbol{u} = \boldsymbol{u_1}\boldsymbol{u_2}\cdots\boldsymbol{u_m}$, the factorization $\mathcal{F}$ is given by the sets $\boldsymbol{u_1}\boldsymbol{u_2}\cdots\boldsymbol{u_m}$ mapped to $\underline{\boldsymbol{u_1}}\underline{\boldsymbol{u_2}}\cdots\underline{\boldsymbol{u_m}}$; for brevity we will write $\boldsymbol{u} <_{LEX} \boldsymbol{v}$. However, if all sets are unit size then the factorization $\mathcal{F}$ of regular strings is the individual letters, each $\underline{x_i}$ is x_i, and $\boldsymbol{u} <_{LEX} \boldsymbol{v}$ is simply the usual lexicographic order of strings $\boldsymbol{u} < \boldsymbol{v}$.

We can now proceed to clarify the concept of conjugacy for an indeterminate string.

Definition An indeterminate string $\boldsymbol{y} = \boldsymbol{y_1}\boldsymbol{y_2}\cdots\boldsymbol{y_m}$ is a *conjugate* (or cyclic rotation) of an indeterminate string $\boldsymbol{x} = \boldsymbol{x_1}\boldsymbol{x_2}\cdots\boldsymbol{x_m}$ if $\boldsymbol{y}[1\ldots m] = \boldsymbol{x}[i\ldots m]\boldsymbol{x}[1\ldots i-1]$ for some $1 \leq i \leq m$ (for $i = 1, \boldsymbol{y} = \boldsymbol{x}$).

Definition An indeterminate string $\boldsymbol{x}$ over Σ^+ is an *Indeterminate Lyndon Word* if it is strictly minimal for the lex-extension order of its congugacy class under the mapping $\mathcal{G} : \boldsymbol{x} \rightarrow \underline{\boldsymbol{x}}$.

Similarly to each letter being a Lyndon word for regular strings, each single set of letters is likewise an indeterminate Lyndon word. Clearly Duval's linear Lyndon factorization algorithm [8] extends directly to the indeterminate case via lex-extension order and linear comparison of the substrings $\underline{\boldsymbol{x_i}}$. We can also trivially derive results analogous to those for the classic case – we give some examples, where $\mathcal{IL}$ is the set of Indeterminate Lyndon Words.

Proposition 4.4 *An indeterminate word $\boldsymbol{w} \in \Sigma^+$ is an indeterminate Lyndon word if and only if it is less in lex-extension order than each of its nonempty proper suffixes.*

Proposition 4.5 *An indeterminate word $\boldsymbol{w} \in \Sigma^+$ is an indeterminate Lyndon word if and only if either $\boldsymbol{w}$ is a single set of letters or $\boldsymbol{w} = \boldsymbol{uv}$ with $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{IL}$, $\boldsymbol{u} <_{LEX} \boldsymbol{v}$.*

A subset $\mathcal{W}$ of Σ^+ is known as a factorization family (FF) if and only if for every nonempty string $\boldsymbol{x}$ on Σ there exists a factorization of $\boldsymbol{x}$ over $\mathcal{W}$ – note that $\Sigma \subseteq \mathcal{W}$. We proceed to show that the set of indeterminate Lyndon words forms an UMFF *(unique maximal factorization family)* [4].

Lemma 4.6 *(The* **xyz** *Lemma [4]) An FF $\mathcal{W}$ is an UMFF if and only if whenever $\boldsymbol{xy}, \boldsymbol{yz} \in \mathcal{W}$ for some nonempty $\boldsymbol{y}$, then $\boldsymbol{xyz} \in \mathcal{W}$.*

Lemma 4.7 *[7] The set $\mathcal{IL}$ of Indeterminate Lyndon Words forms an UMFF.*

Furthermore, as detailed in [7] we are introducing here a new circ-UMFF [5], namely the set $\mathcal{IL}$ of *Indeterminate Lyndon Words.*

5 A Degenerate Burrows-Wheeler Transform

The *degenerate Burrows-Wheeler Transform* - denoted D-BWT - is a very simple extension of the original transformation, which relies only on further use of lexicographic ordering. Given a degenerate string $\boldsymbol{x} = \boldsymbol{x}[1 \dots m] = \boldsymbol{x_1 x_2} \cdots \boldsymbol{x_m}$, to construct the D-BWT, we first perform all the mappings $\mathcal{G} : \boldsymbol{x_i} \rightarrow \underline{\boldsymbol{x_i}}$ specified in Section 4 in linear time. As in the original BWT transformation, we will generate the sorted rotations – the D-BWT matrix – of the input string. To do this we apply a fast suffix-sorting algorithm, such as that of Ko and Aluru [12], tweaked to handle substrings, thus forming an *indeterminate suffix array.* Note that an indeterminate suffix of $\underline{\boldsymbol{x}}$ has the form $\underline{\boldsymbol{x_i}}\ \underline{\boldsymbol{x_{i+1}}} \cdots \underline{\boldsymbol{x_m}}$, and the indexes in the array will be a subset of $\{1, 2, \dots, n\}$.

Given $\underline{\boldsymbol{x}}$, we first perform a pre-sorting of the substrings $\underline{\boldsymbol{x_1}},\ \underline{\boldsymbol{x_2}}, \cdots, \underline{\boldsymbol{x_m}}$ into lex-extension order, resulting in a re-labelling $\pi_1 \pi_2 \cdots \pi_m$ of $\underline{\boldsymbol{x}}$, where each π_i is just a letter or ordinal number. For example, $\underline{\boldsymbol{x}} = \{abc\}\{e\}\{ad\}\{abc\}\{bce\} \rightarrow ADBAC$ or 14213. This can be achieved using Bucket Sort on the finite ordered alphabet Σ (assumed in Section 4), with the buckets labelled by the characters in Σ. This process is repeated in each bucket where the length of each $\underline{\boldsymbol{x_i}}$ is $O(n)$ - hence $O(n)$ overall.

The indeterminate string $\boldsymbol{x}$ has now been re-labelled as a string of letters $\pi_1\pi_2\cdots\pi_m$ each according to their lex-extension order in $\boldsymbol{x}$. Therefore we can straightforwardly apply an existing linear letter-based suffix-sorting technique to yield a suffix array for the indexes $i \in \{1\ldots m\}$. A trivial mapping of each array element $i \rightarrow \sum_{j=1}^{i-1} |\boldsymbol{x_j}| + 1$ then gives the required indeterminate suffix array. The overall linear - $O(n)$ - time and space complexities follow from the original $O(m)$ method (for instance [12]) along with $O(n)$ total string length.

Given the D-BWT matrix, in the degenerate case the transform is the last right-most column of ordered sets, specifically a permutation of $\underline{\boldsymbol{x}} = \underline{\boldsymbol{x_1}\boldsymbol{x_2}\cdots\boldsymbol{x_m}}$, together with the index of the given text in the matrix. Using the re-labelling to letters π_i, the transform can be encoded as letters and the inverse achieved using the classic linear Last-First mapping. Finally the inverse mappings $\pi_i \rightarrow \underline{\boldsymbol{x_j}} \rightarrow \boldsymbol{x_j}$ reconstruct the original degenerate string, hence overall linear.

Furthermore, if we assume that the input text has been factored into indeterminate Lyndon words, then this avoids an index to the rotation in the matrix which is the input text. Once factored, and again using the re-labelling to letters $\underline{\boldsymbol{x_j}} \rightarrow \pi_i$, the bijective multi-word BWT described by Kufleitner [13] can be applied directly, followed by inverse mappings from the π_i to recover the indeterminate subsets in the input text.

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[2] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[3] K. T. Chen, R. H. Fox, and R. C. Lyndon. *Free differential calculus IV — The quotient groups of the lower central series*, volume 68. Ann. Math., 1958.

[4] D. E. Daykin and J. W. Daykin. Lyndon-like and v-order factorizations of strings. *J. Discrete Algorithms*, 1(3-4):357–365, 2003.

[5] D. E. Daykin and J. W. Daykin. Properties and construction of unique maximal factorization families for strings. *Int. J. Found. Comput. Sci.*, 19(4):1073–1084, 2008.

[6] J. W. Daykin and W. . F. Smyth. A bijective variant of the burrows-wheeler transform using v-order. 2013. Submitted.

[7] J. W. Daykin and B. Watson. Indeterminate string factorizations and degenerate text transformations. 2013. Submitted.

[8] J.-P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[9] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.

[10] J. Holub and W. F. Smyth. Algorithms on indeterminate strings. In *Proc. 14th Australasian Workshop on Combinatorial Algs.*, pages 36–45, 2003.

[11] C. S. Iliopoulos, L. Mouchard, and M. S. Rahman. A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. *Math. in Computer Science*, 2(4), 2008.

[12] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM'03, pages 200–210, Berlin, Heidelberg, 2003. Springer-Verlag.

[13] M. Kufleitner. On bijective variants of the Burrows-Wheeler Transform. In J. Holub and J. Zdárek, editors, *Stringology*, pages 65–79. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009.

[14] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, 10(3):R25, 2009.

[15] R. Li, C. Yu, Y. Li, T. W. Lam, S. M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[16] M. Lothaire. *Combinatorics on Words (Cambridge Mathematical Library).* Cambridge University Press; 2nd Edition, 1997.

[17] R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77:202–215, 1954.

[18] M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Proceedings of the Conference on Data Compression*, volume 469, 1997.

[19] W. Smyth. *Computing Patterns in Strings.* Addison-Wesley, 2003.

[20] Y.-T. Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173 – 176, 2003.

Block Graphs in Practice

Travis Gagie[1,*], Christopher Hoobin[2] Simon J. Puglisi[1]

[1]Department of Computer Science, University of Helsinki, Finland
[2]School of CSIT, RMIT University, Australia
[*]Travis.Gagie@cs.helsinki.fi

Abstract

Motivated by the rapidly increasing size of genomic databases, code repositories and versioned texts, several compression schemes have been proposed that work well on highly-repetitive strings and also support fast random access: e.g., LZ-End, RLZ, GDC, augmented SLPs, and block graphs. Block graphs have good worst-case bounds but it has been an open question whether they are practical. We describe an implementation of block graphs that, for several standard datasets, provides better compression and faster random access than competing schemes.

1 Introduction

Advances in DNA sequencing technology have led to massive genomic databases, the open-source movement has led to massive code repositories, and the popularity of wikis has led to massive versioned textual databases. Fortunately, all these datasets tend to be highly repetitive and, thus, highly compressible. Compressing them is only useful, however, if we can still access them quickly afterwards. Although many papers have been published about compression schemes with fast random access (see [6] for a recent survey), most have been about schemes such as Huffman coding, LZ78, CSAs or BWT-based coding. Only relatively recently have researchers started proposing schemes with random access and LZ77-like compression, which is better suited highly-repetitive strings.

One approach uses variants of LZ77 itself. Kreft and Navarro's LZ-End [7] is practical but lacks good worst-case bounds, for both the compression and the random-access time. Kuruppu, Puglisi and Zobel's Relative Lempel-Ziv (RLZ) [8, 9] or Deorowicz and Grabowski's Genome Differential Compressor (GDC) [4] are also practical but are not general-purpose: we can apply them only when we have a good reference sequence, or can construct one. Even then, Deorowicz, Danek and

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

Grabowski [3] have observed that when compressing genomes, "the key to high compression is to look for similarities across the whole collection, not just against one reference sequence".

Another approach uses straight-line programs (SLPs). An SLP for a string s is a context-free grammar in Chomsky normal for that generates s and only s. Rytter [13] and Charikar et al. [2] showed that if s has length n and its LZ77 parse consists of z phrases, then we can build a SLP for s with $\mathcal{O}(z \log n)$ rules and height $\mathcal{O}(\log n)$. If we store the resulting SLP together with the size of each non-terminal's expansion, which takes a total of $\mathcal{O}(z \log n)$ space, then we can extract any substring of length ℓ in $\mathcal{O}(\log n + \ell)$ time. This extraction time nearly matches a lower bound by Verbin and Yiu [14]. Bille et al. [1] showed how we can store any SLP with r rules, regardless of the height, in $\mathcal{O}(r)$ space with the same time bound for extraction. These data structures are not practical, however. The most recent and practical SLP-based scheme of which we are aware is Maruyama, Tabei, Sakamoto and Sadakane's Fully-Online LCA (FOLCA) [10] but, apart from needing less resources for construction, even this is less practical than LZ-End.

In a previous paper [5] we introduced a third approach, called block graphs, and showed that they also use $\mathcal{O}(z \log n)$ space and $\mathcal{O}(\log n + \ell)$ extraction time. We did not implement these data structures for that paper, however, so it has been an open question whether they are practical. In this paper we describe an implementation that, for several standard datasets, provides better compression and faster random access than LZ-End; thus, block graphs are competitive in both theory and practice. In Section 2 we review the definition of the block graph for a string. In Section 3 we describe some details of our implementation. Finally, in Section 4 we report on our experiments.

2 Block Graphs

The block graph of the string $s[1..n]$ is a directed acyclic graph (DAG) in which each node in-degree up to 2 and out-degree up to 3. For simplicity, assume n is a power of 2. Each node of the block graph corresponds to a substring, called that node's block: the root corresponds to the whole of s; if they exist, the children of a node v correspond to the first half v's substring, the middle half of v's substring (which overlaps the first and last halves), and the last half of v's substring. In general, v shares its left and right children with its left and right siblings, respectively, because v's left child's block is both the last half of v's left sibling's block and the first half of v's left sibling's block, and v's right child's block is both the last half of v's block and the first half of v's right sibling's block. If n is not a power of 2, then we pad it so that it is, build the block graph, and prune redundant nodes. Figure 1 — which is copied from our earlier paper — shows the block graph for the eighth Fibonacci string, abaababaabaababaababa, truncated at depth 3.

We mark as an *internal node* each node whose block is the first occurrence of that substring (shown in Figure 1 as ovals). We mark as a *leaf* all nodes whose block is not unique and whose parents are internal nodes (shown in Figure 1 as rectangles). We remove leaves' children (and their descendants) and replace them by pointers, as explained in our earlier paper:

Figure 1: The block graph for the eighth Fibonacci string, abaababaabaababaababa, truncated at depth 3.

> Suppose a leaf u at depth d had a child $\langle i..j \rangle$... Consider the first occurrence $s[i'..j']$ in s of the substring $s[i..j]$. Notice that $s[i'..j']$ is completely contained within some block at depth d — this is one reason why we use overlapping blocks — and, since $s[i'..j']$ is the first occurrence of that substring in s, that block is associated with an internal node v. We replace the pointer from u to $\langle i..j \rangle$ by a pointer to v and the offset of i' in v's block.

At some depth, we store all internal nodes' blocks instead of their children; this depth is 3 in Figure 1.

In our previous paper we proved that

1. the block graph for s has $\mathcal{O}(z \log n)$ nodes and, thus, takes $\mathcal{O}(z \log n)$ space;
2. we can extract any substring of length ℓ in $\mathcal{O}(\log n + \ell)$ time;
3. if we store all of the nodes at depth d but removing all nodes above that depth, we change the space usage to $\mathcal{O}\big(z(\log n - d) + 2^d\big)$ and decrease the extraction time to $\mathcal{O}(\log n - d)$.

For more details, we refer readers to the proofs and discussion it contains.

3 Implementation

We now describe an implementation of block graphs which is efficient in practice. The main idea is to represent the shape of the graph (the internal nodes and their pointers) using bitvectors and operations from succinct data structures, and to carefully allocate space for the leaf nodes depending on their distance from the root. Below we make use of two familiar operations for bitvectors: *rank* and *select*. Given a bitvector B, a position i, and a type of bit b (either 0 or 1), $rank_b(B, i)$ returns the number of occurrences of b before position i in B and $select_b(B, i)$ returns the position of the ith b in B. Efficient data structures supporting these operations have been extensively studied (see, e.g., [11, 12]).

Recall, each level of the block graph consists of a number of nodes, either internal nodes, or leaves. Let B_d be a bitvector which indicates whether the ith node (from the left) at depth d is a leaf, $B_d[i] = 0$, or an internal node, $B_d[i] = 1$. We define another bitvector R_d, where $R_d[i] = 1$ if and only if $B_d[i] = 1$ and $B_d[i+1] = 1$ for $i < n-1$. That is, we mark a 1 bit for each instance of two adjacent internal nodes in B_d, otherwise $R_d[i] = 0$. Let L_d be an array that holds leaf nodes at depth d. The structure of a leaf node is discussed below. Finally, let T be the concatenation of the textual representation (ie. the corresponding substrings) of all internal nodes at the truncated depth, d'. As adjacent text blocks share $2^{\log n - d' - 1}$ characters, we concatenate only the last half of a new adjacent block to T. Non-adjacent blocks are fully concatenated. We utilize an R_d bitvector at this level; however, we mark $R_d[i] = 1$ if the ith node at the truncated depth is a text block.

Navigating the block graph

The main operation is to traverse from an internal node to one of its three children. Say we are currently at the jth internal node at depth d of the block graph — that is, we are at $B_d[i]$, where $i = \mathit{select}_1(B_d, j)$. Each internal node has three children. If these children were independent then locating the left child of the current node would be simply three times the node's position on its level, that is $3j = 3 \cdot \mathit{rank}_1(B_d, i)$. However, in a block graph, adjacent internal nodes share exactly one child, so we correct for this by subtracting the number of adjacent internal nodes at this depth prior to the current node — this is given by $\mathit{rank}_1(R_d, i)$. To find the position corresponding to the left child of a node in B_{d+1} we compute $\text{leftchild}(B_d, i) = 3 \cdot \mathit{rank}_1(B_d, i) - \mathit{rank}_1(R_d, i)$.

Given the address of the left child it is easy to find the center or right child by adding 1 or 2, respectively. If $B_d[i] = 0$ then we are at a leaf node, and its leaf information is at $L_d[\mathit{rank}_0(B_d, i)]$. Once we reach the truncated depth we access the text of an internal node by computing its offset in T as $T[(\mathit{rank}_1(B_d, i) \cdot 2^{\log n - d' - 1}) - (\mathit{rank}_1(R_d, i) \cdot 2^{\log n - d' - 1})]$.

Leaf nodes

In a block graph, leaves point to internal nodes. For each leaf we store two values, the position of the destination node on the current level, and an offset in the destination node pointing to the beginning of the leaf block. Note that we do not need to store the depth of the destination node. It is, by definition, on the level above the leaf, and we know this by keeping keep track of the depth during each step in a traversal. To improve compression we store leaf positions and offsets in two separate arrays. At depth d there are no more than $2^{d+1} - 1$ possible nodes, so we can store each position in $\log(2^{d+1} - 1)$ bits. Given that the length of a node at depth d is $b = 2^{\lceil \log n \rceil - d}$ and leaf nodes point to an internal node on the level above, we store each offset in $\log(2^{\lceil \log n \rceil - d - 1})$ bits.

Table 1: Size in MB of repetitive corpus files encoded with ASCII, gzip, xz, LZ-End and block graphs truncated at text length 4, 8, 16 and 32.

Collection	ASCII	GZIP	XZ	LZ-End	Bg4	Bg8	Bg16	Bg32
world_leaders	49	8.28	0.51	4.52	6.62	5.83	5.72	6.37
Escherichia_Coli	112	31.53	5.18	49.10	49.70	49.57	45.33	46.91
influenza	154	10.63	1.59	21.50	33.16	32.97	33.32	37.89
coreutils	205	49.92	3.70	35.88	42.80	33.19	30.43	33.00
kernel	257	69.39	2.07	19.34	21.21	15.69	13.84	14.05
para	429	116.07	6.09	57.41	72.39	72.13	67.84	70.66
cere	461	120.08	5.07	41.34	57.68	57.54	54.59	57.96
einstein.en.txt	467	163.66	0.33	2.24	3.52	3.07	3.01	3.19

4 Experiments

We have developed an implementation of block graphs[1] and tested it on texts from the Pizza-Chili Repetitive Corpus[2], a standard testbed for data structures designed for repetitive strings.

We compared compression achieved by the block graph to the LZ-End data structure by Kreft and Navarro [7], and to the general-purpose compressors gzip and xz; the results are shown in Table 1. gzip and xz were run with their highest compression setting -9, while LZ-End was executed with its default settings. Throughout our experiments we tested block graphs that were truncated such that the smallest blocks were 4, 8, 16 and 32 bytes. Note that gzip and xz provide compression only, not random access, and are included as reference points for achievable compression. We did not test extraction from bookmarks because Kreft and Navarro's data structure does not support it (nor does any other implemented data structure).

We then compared how quickly block graphs and LZ-End support extracting substrings of various lengths; the results are shown in Figure 2. The mean extraction speed with LZ-End never exceeded 9 million characters per second while, for sufficiently long extractions from the kernel file, the mean extraction speed with the block graph was over 480 million characters per second. Each run of extractions was performed across 10,000 randomly-generated queries. Experiments were conducted on an Intel Core i7-2600 3.4 GHz processor with 8GB of main memory, running Linux 3.3.4; code was compiled with GCC version 4.7.0 targeting x86_64 with full optimizations. Caches were dropped between runs with `sync && echo 1 > /proc/sys/vm/drop_caches`.

Although xz achieves much better compression, block graphs achieve better compression than gzip except on the Escherichia Coli and influenza files. Most importantly, our experiments show that block graphs generally achieve compression comparable to that achieved by LZ-End while supporting significantly faster substring extraction.

[1] Available at http://www.github.com/choobin/block-graph
[2] http://pizzachili.dcc.uchile.cl/repcorpus.html

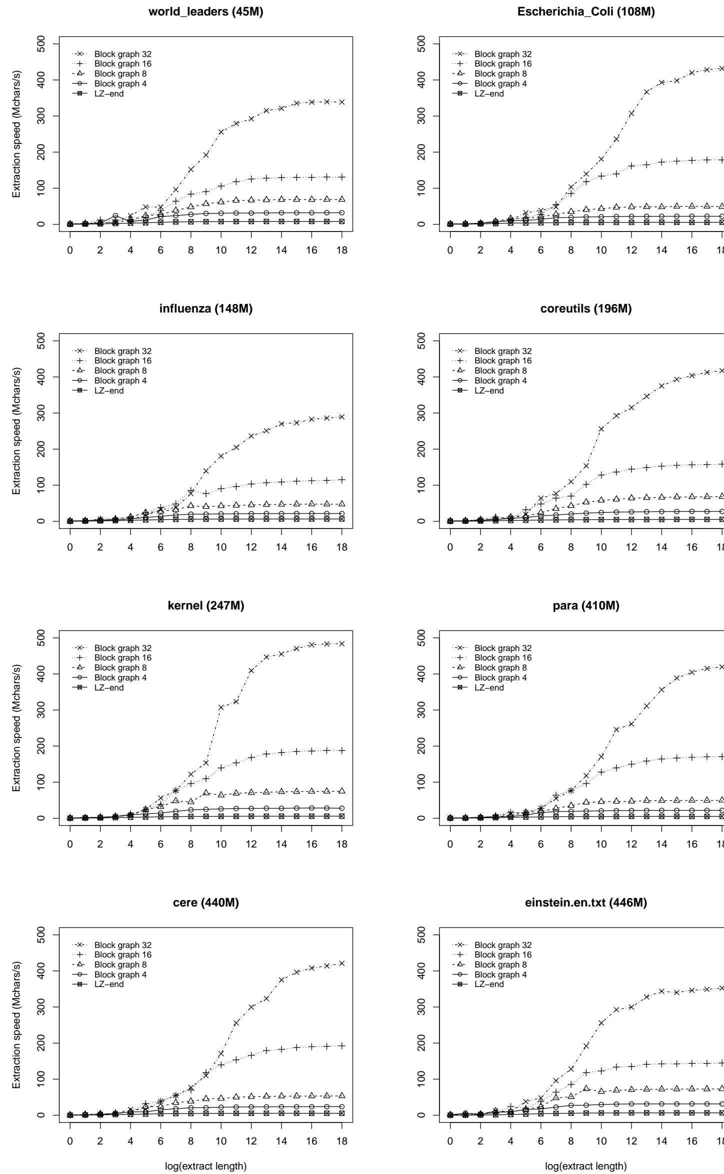

Figure 2: Extraction speeds in millions of characters per second versus the binary logarithm of the length of the extracted substring. Each data point is averaged over 10,000 random substring extractions.

References

[1] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proceedings of the 22nd Symposium on Discrete Algorithms (SODA)*, pages 373–389, 2011.

[2] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[3] S. Deorowicz, A. Danek, and S. Grabowski. Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578, 2013.

[4] S. Deorowicz and S. Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011.

[5] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 653–662, 2011.

[6] R. Grossi. Random access to high-order entropy compressed text. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, pages 199–215. Springer-Verlag, 2013.

[7] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, to appear.

[8] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.

[9] S. Kuruppu, S. J. Puglisi, and J. Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proceedings of the 34th Australasian Computer Science Conference (ACSC)*, pages 91–98, 2011.

[10] S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane. Fully-online grammar compression. In *Proceedings of the 20th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 218–229, 2013.

[11] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[12] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[13] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[14] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proceeings of the 24th Symposium on Combinatorial Pattern Matching (CPM)*, pages 247–258, 2013.

Compressed Spaced Suffix Arrays

Travis Gagie[1,*], Giovanni Manzini[2], Daniel Valenzuela[1]

[1]Department of Computer Science, University of Helsinki, Finland
[2]University of Eastern Piedmont, Italy
[*]Travis.Gagie@cs.helsinki.fi

Abstract

Spaced seeds are important tools for similarity search in bioinformatics, and using several seeds together often significantly improves their performance. With existing approaches, however, for each seed we keep a separate linear-size data structure, either a hash table or a spaced suffix array (SSA). In this paper we show how to compress SSAs relative to normal suffix arrays (SAs) and still support fast random access to them. We first prove a theoretical upper bound on the space needed to store an SSA when we already have the SA. We then present experiments indicating that our approach works even better in practice.

1 Introduction

For the problem of similarity search, we are given two texts and asked to find each sufficiently long substring of the first text that is within a certain Hamming distance of some substring of the second text. Similarity search has many applications in bioinformatics — e.g., ortholog detection, structure prediction or determining rearrangements — and has been extensively studied (see, e.g., [19]). Researchers used to first look for short substrings of the first text that occur unchanged in the second text, called seeds, then try to extend these short, exact matches in either direction to obtain longer, approximate matches. This approach is called, naturally enough, "seed and extend". The substrings' exact matches are found using either a hash table of the substrings with the right length, or an index structure such as a suffix array (SA).

Around the turn of the millenium, Burkhardt and Kärkkäinen [5] and Ma, Tromp and Li [15] independently proposed looking for short *subsequences* of the first text that have a certain shape and occur unchanged in the second text, and trying to extend those. A binary string encoding the shape of a subsequence, with

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

1s indicating positions where the characters must match and 0s indicating positions where they need not, is called a spaced seed. The total number of bits in the binary string is called the seed's length, and the number of 1s is called its weight. The subsequences' exact matches are found using either a hash table of the subsequences with the right shape, or a kind of modified SA called a spaced suffix array [12] (SSA).

Burkhardt and Kärkkäinen, Ma et al. and subsequent authors have shown that using spaced seeds significantly improves the performance of seeding and extending. Many papers have been written about how to design spaced seeds to minimize the number of errors (see, e.g., [4, 8, 11] and references therein), with the specifics depending on the model of sequence similarity and the acceptable numbers of false positives (for which the characters indicated by 1s all match but the substrings are not similar) and false negatives (for which those do not all match but the substrings are still similar) for the application in question. Regardless of the particular application, however, researchers have consistently observed that the best results are obtained using more than one seed at a time. A set of spaced seeds used in combination is called a multiple seed.

Multiple seeds are now a popular and powerful tool for similarity search, but they have a lingering flaw: we keep a hash table or SSA for each seed, and each instance of these data structures takes linear space. For example, SHRiMP2's [7] index for the human genome takes 16 GB *for each seed*. In contrast, Bowtie 2's [14] compressed SA for that genome takes only 2.5 GB. This is because a normal SA (which supports only substring matching) can be compressed such that the number of bits per character is only slightly greater than the empirical entropy of the text. Unfortunately, the techniques for compressing normal SAs do not seem to apply directly to SSAs.

In this paper we show how to compress SSAs relative to normal SAs and still support fast random access to them. Whereas the normal SA for a text lists the starting points of the suffixes of that text by those suffixes' lexicographic order, the SSA for a text and a spaced seed lists the starting points of the subsequences with the right shape by those subsequences' lexicographic order. Intuitively, if the seed starts with many 1s, the SSA will be similar to the SA. In Section 2 we formalize this intuition and prove a theoretical upper bound on the space needed to store an SSA when we already have the SA, in terms of the text's length, the alphabet's size, and the seed's length and weight.

In Section 3 we present experiments showing that our approach works even better in practice. That is, even when we implement our data structures using simpler, theoretically sub-optimal components, we achieve better compression than our upper bound predicts. In fact, in practice we can even successfully apply our approach in some cases when the assumptions underlying our upper bounds are violated. However, we still want to improve our compression and random-access times for seeds with low weight-to-length ratios.

We recently learned that Peterlongo et al. [17] and Crochemore and Tischler [6] independently defined SSAs, under the names "bi-factor arrays" and "gapped suffix arrays", for the special case in which the spaced seed has the form $1^a0^b1^c$. Russo and Tischler [18] showed how to represent such an SSA in asymptotically succinct

space such that we can support random access to it in time logarithmic in the length of the text. We note, however, that the spaced seeds used for most applications do not have this form. We also recently learned that Battaglia et al. [2] used an idea similar to that of spaced seeds in an algorithm for finding motifs with don't-care symbols. It seems possible our results could be useful in reducing their algorithm's memory usage.

2 Theory

Suppose we want to store an SSA for a text $T[0..n-1]$ over an alphabet of size σ and a spaced seed S with length ℓ and weight w. For $i < n$, let T_i be the subsequence of $T[i..n-1]$ that contains $T[j]$ if and only if $i \leq j$ and $S[j-i] = 1$. Let T'_i be the subsequence of $T[i..n-1]$ that contains $T[j]$ if and only if $S[j-i] = 0$. Let SSA be the permutation on $\{0, \ldots, n-1\}$ in which i precedes i' if either $T_i \prec T_{i'}$, or $T_i = T_{i'}$ and $T[i..n-1] \prec T[i'..n-1]$.

For example, if $T = \mathsf{abracadabra}$ and $S = 101$ then

$$
\begin{array}{lll lll lll lll}
T_0 & = & \mathsf{ar} & T_6 & = & \mathsf{db} & T'_0 & = & \mathsf{b} & T'_6 & = & \mathsf{a} \\
T_1 & = & \mathsf{ba} & T_7 & = & \mathsf{ar} & T'_1 & = & \mathsf{r} & T'_7 & = & \mathsf{b} \\
T_2 & = & \mathsf{rc} & T_8 & = & \mathsf{ba} & T'_2 & = & \mathsf{a} & T'_8 & = & \mathsf{r} \\
T_3 & = & \mathsf{aa} & T_9 & = & \mathsf{r} & T'_3 & = & \mathsf{c} & T'_9 & = & \mathsf{a} \\
T_4 & = & \mathsf{cd} & T_{10} & = & \mathsf{a} & T'_4 & = & \mathsf{a} & & & \\
T_5 & = & \mathsf{aa} & & & & T'_5 & = & \mathsf{d} & & &
\end{array}
$$

and so $\mathrm{SSA} = [10, 3, 5, 7, 0, 8, 1, 4, 6, 9, 2]$, while $\mathrm{SA} = [10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2]$.

If $T_i \preceq T_{i'}$ and $T'_i \preceq T'_{i'}$, then i precedes i' in both SSA and SA. In particular, if $T_i = T_{i'}$ or $T'_i = T'_{i'}$, then i and i' have the same relative order in SSA and SA. In our example, $T_3 = T_5 = \mathsf{aa}$, so 3 precedes 5 in both SSA and SA; $T'_2 = T'_6 = T'_9 = \mathsf{a}$, so 6 precedes 9 and 9 precedes 2 in both SSA and SA.

If we partition SSA into subsequences such that i and i' are in the same subsequence if and only if $T_i = T_{i'}$, then we can partition SA into the same subsequences. Since there are at most $\sigma^w + w$ distinct strings T_i, our partitions each consist of at most $\sigma^w + w$ subsequences. Similarly, if we partition based on T'_i and $T'_{i'}$, then our partitions each consist of at most $\sigma^{\ell-w} + \ell - w$ subsequences.

For our example, we can partition both SSA and SA into $[4, 6, 9, 2]$, for $T'_i = \mathsf{a}$; $[7, 0]$, for $T'_i = \mathsf{b}$; $[3]$, for $T'_i = \mathsf{c}$; $[5]$, for $T'_i = \mathsf{d}$; $[8, 1]$, for $T'_i = \mathsf{r}$; and $[10]$, for $T'_i = \epsilon$. In this particular case, however, we could just as well partition both SSA and SA into only two common subsequences: e.g., $[10, 7, 0]$ and $[3, 5, 8, 1, 4, 6, 9, 2]$.

Consider the permutation $\mathrm{SA}^{-1} \circ \mathrm{SSA}$, which maps elements' positions in SSA to their positions in SA, and let ρ be the minimum number of increasing subsequences into which $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ can be partitioned. Since any subsequence common to SSA and SA corresponds to an increasing subsequence in $\mathrm{SA}^{-1} \circ \mathrm{SSA}$, we have $\rho \leq \min(\sigma^w + w, \sigma^{\ell-w} + \ell - w)$. In our example, $\mathrm{SA}^{-1} \circ \mathrm{SSA} = [0, 3, 4, 1, 2, 5, 6, 7, 8, 9, 10]$ and $\rho = 2$.

Supowit [20] gave a simple algorithm that partitions $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ into ρ increasing subsequences in $\mathcal{O}(n \lg \rho) \subseteq \mathcal{O}(n \min(w, \ell - w) \lg \sigma)$ time. When applied to

$\text{SA}^{-1} \circ \text{SSA}$ in our example, Supowit's algorithm partitions it into $[0, 3, 4]$ and $[1, 2, 5, 6, 7, 8, 9, 10]$.

Barbay et al. [1] showed how, given a partition of $\text{SA}^{-1} \circ \text{SSA}$ into ρ increasing subsequences, we can store it in $(2+o(1))n \lg \rho \leq (2+o(1))n \min(w, \ell-w) \lg \sigma$ bits and support random access to it in $\mathcal{O}(\lg \lg \rho)$ time. Combining their ideas with later work by Belazzougui and Navarro [3], we can keep the same space bound and improve the time bound to $\mathcal{O}(1)$.

To do this, for $i \leq \rho$, we replace each element in the ith subsequence in $\text{SA}^{-1} \circ \text{SSA}$ by a character a_i, and store the resulting string R such that we can support random access to it and partial rank queries on it. We then permute R according to $\text{SA}^{-1} \circ \text{SSA}$ and store the resulting string R' such that we can support fast select queries on it. In our example, $R = a_1 a_2 a_2 a_1 a_1 a_2 a_2 a_2 a_2 a_2 a_2$ and $R' = a_1 a_1 a_1 a_2 a_2 a_2 a_2 a_2 a_2 a_2 a_2$.

The partial rank query $R.\text{p_rank}(i)$ returns the number of copies of $R[i]$ in $R[0..i]$, and the select query $R'.\text{select}_a(i)$ returns the position of the ith copy of a in R'. Barbay et al. noted that, for $i < n$,

$$(\text{SA}^{-1} \circ \text{SSA})[i] = R'.\text{select}_{R[i]}(R.\text{p_rank}(i)) \ .$$

Belazzougui and Navarro showed how we can store R in $(1 + o(1))n \lg \rho$ bits and support random access to it and partial rank queries on it in $\mathcal{O}(1)$ time, and store R' in $(1 + o(1))n \lg \rho$ bits and support select queries on it in $\mathcal{O}(1)$ time.

In summary, we can store $\text{SA}^{-1} \circ \text{SSA}$ in $(2 + o(1))n \min(w, \ell - w) \lg \sigma$ bits such that we can support random access to it in $\mathcal{O}(1)$ time. We will give a longer explanation in the full version of this paper. Since $\text{SSA} = \text{SA} \circ (\text{SA}^{-1} \circ \text{SSA})$, this gives us the following result:

Theorem 2.1 *Let $T[0..n-1]$ be a text over an alphabet of size σ and let S be a spaced seed with length ℓ and weight w. If we have already stored the suffix array* SA *for T such that we can support random access to* SA *in time t_{SA}, then we can store a spaced suffix array* SSA *for T and S in $(2 + o(1))n \min(w, \ell - w) \lg \sigma$ bits such that we can support random access to* SSA *in $t_{\text{SA}} + \mathcal{O}(1)$ time.*

3 Practice

Theorem 2.1 says we can store SSAs for the human Y-chromosome `chrY.fa` in FASTA format (about 60 million characters over an alphabet of size 5) and SHRiMP2's three default spaced seeds — i.e., 11110111101111, 1111011100100001111 and 1111000011001101111 — in about 560 MB, in addition to the SA, whereas storing the SSAs naïvely would take about 720 MB. Storing the SSAs packed such that each entry takes $\lceil \lg 60\,000\,000 \rceil = 26$ bits would reduce this to about 580 MB.

To test our approach, we built the SSAs as described in Section 2; computed $\text{SA}^{-1} \circ SSA$, R and R' for each SSA; and stored each copy of R or R' as a wavelet tree. We chose wavelet trees because they are simple to use and often more practical than the theoretically smaller and faster data structures mentioned in Section 2. We ran all our tests described in this section on a computer with a quad-core Intel

Xeon CPU with 32 GB of RAM, running Ubuntu 12.04. We used a wavelet-tree implementation from `https://github.com/fclaude/libcds` and compiled it with GNU `g++` version 4.4.3 with optimization flag `-O3`.

The uncompressed SA took 226 MB, and the six wavelet trees took a total of 215 MB and performed 10 000 random accesses each in 7.67 microseconds per access. That is, we compressed the SSAs into about 60% of the space it would take to store them naïvely and, although our accesses were much slower than direct memory accesses, they were fast compared to disk accesses. Thus, our approach seems likely to be useful when a set of SSAs is slightly larger than the memory and fits only when compressed.

Using the same test setup, we then compressed SSAs for the ten spaced seeds BFAST [9, Table S3] uses for 36-base-pair Illumina reads, which all have weight 18:

1. 111111111111111111
2. 11110100110111101010101111
3. 11111111111111001111
4. 1111011101100101001111111
5. 11110111000101010000010101110111
6. 1011001101011110100110010010111
7. 1110110010100001000101100111001111
8. 1111011111111111111
9. 11011111100010110111101101
10. 111010001110001110100011011111 .

Since the first seed consists only of 1s, the SSA we would build for it is the same as the SA. The uncompressed SA again took 226 MB and the 18 wavelet trees for the other nine seeds took a total of 649 MB — so instead of 2.26 GB, we used 875 MB (about 39%) for all ten seeds — and together performed 10 000 random accesses to each of the ten SSAs in about 7 microseconds per access. The left side of the top half of Figure 1 shows how many bits per character (bpc) of the text each SSA took, and the average time per access to each SSA.

We also compressed the SSAs for the ten spaced seeds BFAST uses for 50-base-pair Illumina reads, which all have weight 22:

1. 1111111111111111111111
2. 1111101110111010100101011011111
3. 1011110101101001011000011010001111111
4. 10111001101001100100111101010001011111
5. 11111011011101111011111111
6. 111111100101001000101111101110111
7. 11110101110010100010101101010111111
8. 111101101011011001100000101101001011101
9. 1111011010001000110101100101100110100111

10. 111101001011011010111001011011101 1.

Again, the first seed consists only of 1s. This time, the 18 wavelet trees for the other nine seeds took a total of 712 MB; each access took about 8 microseconds. The left side of the bottom half of Figure 1 shows how many bit per character of the text each SSA took, and the average access time per access to each SSA.

If we have a permutation π_1 on $\{0, \ldots, n-1\}$ stored and π_2 is any other permutation on $\{0, \ldots, n-1\}$, then we can store π_2 relative to π_1 using the ideas from Section 2. For example, we can store SSAs relative to other SSAs. Suppose we consider the size of each SSA (except the SA) when compressed relative to each other SSA (including the SA), build a minimum spanning tree rooted at the SA, and compress each SSA relative to its parent in the tree. This can reduce our space usage at the cost of increasing the random-access time, as shown for the BFAST seeds on the right side of Figure 1.

There are other circumstances in which we can ignore SSAs' semantics and consider them only as permutations. For example, spaced seeds can be generalized to subset seeds [13], such as ternary strings in which 1s indicate positions where the characters must match, 0s indicate positions where they need not, and Ts indicate positions where characters must fall within the same equivalence class (such as the pyrimidines `C` and `T` and the purines `A` and `G`). It is not difficult to generalize Theorem 2.1 to subset seeds — we will do so in the full version of this paper — but it is also not necessary to obtain practical results. The `Iedera` tool (available at `http://bioinfo.lifl.fr/yass/iedera.php`) generates good subset seeds.

A more challenging change is from fixed-length seeds to repetitive seeds [12]. A repetitive seed is a string in whose repetition the digits indicate which characters must match and how. For example, with respect to the repetitive spaced seed 10110, `ATCGATCGGT` matches `ACCGTTGGGA` but not `ACCGTTGAGA`. Repetitive seeds are useful when looking for approximate matches of substrings that have been extended until they become sufficiently infrequent. It is not clear how or if we can extend Theorem 2.1 to repetitive seeds. Nevertheless, the `LAST` tool (available at `http://last.cbrc.jp`) generates SSAs for repetitive spaced or subset seeds, which we can still try to compress in practice; see also [10, 16].

Our current goal is to achieve reasonable compression and access times for a set of repetitive subset seeds that we received from Martin Frith, which have average length 19.85 and average weight about 10.44, counting "same equivalence class" digits as 0.5. Unfortunately, at the moment we use nearly 24 bits per entry in the corresponding SSAs (including the overhead for the uncompressed SA), which is only marginally better than the 26 bits we would use with simple packing. Meanwhile, random accesses take about 12 microseconds on average, which is significantly slower than access to a packed array. On the other hand, these seeds have an unusually low average weight-to-length ratio. We used `Iedera` and `LAST` to generate SSAs for a set of eight repetitive subset seeds, with average length 17.875 and average weight 12. For these, we used only 20.15 bits per entry, with random accesses taking about 10 microseconds on average.

seed	space (bpc)	time (μs)
1	32.00	0
2	11.29	9
3	4.41	4
4	9.75	8
5	11.54	9
6	13.77	11
7	13.14	10
8	3.85	3
9	10.10	7
10	13.91	11

seed	reference	space (bpc)	time (μs)
1	-	32.00	0
2	8	9.71	11
3	1	4.41	4
4	8	9.22	10
5	4	9.23	19
6	8	12.27	14
7	3	12.58	14
8	1	3.85	3
9	1	10.10	7
10	7	12.59	26

seed	space (bpc)	time (μs)
1	32.00	0
2	9.03	8
3	12.30	10
4	13.86	11
5	8.13	7
6	10.80	9
7	11.14	8
8	11.09	8
9	11.77	9
10	12.54	10

seed	reference	space (bpc)	time (μs)
1	-	32.00	0
2	1	9.03	6
3	1	12.30	10
4	2	12.59	18
5	1	8.13	6
6	1	10.80	8
7	1	11.14	8
8	1	11.09	9
9	8	11.34	18
10	8	8.94	17

Figure 1: The space usage of the SSAs of the spaced seeds BFAST uses for Illumina reads, in bits per character of the text, and the average time for a random access. On top, the seeds are for 36-base-pair reads; on the bottom, the seeds are for 50-base-pair reads. On the left, all the SSAs are compressed relative to the SA; on the right, some of the SSAs are compressed relative to other SSAs.

Acknowledgments

Many thanks to Francisco Claude, Maxime Crochemore, Matei David, Martin Frith, Costas Iliopoulos, Juha Kärkkäinen, Gregory Kucherov, Bin Ma, Ian Munro, Taku Onodera, Gonzalo Navarro, Luis Russo, German Tischler and the anonymous reviewers.

References

[1] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica.* To appear.

[2] G. Battaglia, D. Cangelosi, R. Grossi, and N. Pisanti. Masking patterns in sequences: A new class of motif discovery with don't cares. *Theoretical*

Computer Science, 410:4327–4340, 2009.

[3] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*. To appear.

[4] D. G. Brown. *Bioinformatics Algorithms: Techniques and Applications*, chapter A survey of seeding for sequence alignment, pages 126–152. Wiley-Interscience, 2008.

[5] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informicae*, 56:51–70, 2003.

[6] M. Crochemore and G. Tischler. The gapped suffix array: A new index structure for fast approximate matching. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 359–364, 2010.

[7] M. David, M. Dzamba, D. Lister, L. Ilie, and M. Brudno. SHRiMP2: Sensitive yet practical short read mapping. *Bioinformatics*, 27:1011–1012, 2011.

[8] L. Egidi and G. Manzini. Better spaced seeds using quadratic residues. *Journal of Compututer and System Sciences*, 79:1144–1155, 2013.

[9] N. Homer, B. Merriman, and S. F. Nelson. BFAST: An alignment tool for large scale genome resequencing. *PLOS One*, 4:e7767, 2009.

[10] P. Horton, S. M. Kiełbasa, and M. C. Frith. DisLex: A tranformation for discontiguous suffix array construction. In *Proceedings of the Workshop on Knowledge, Language, and Learning in Bioinformatics (KLLBI)*, pages 1–11, 2008.

[11] L. Ilie, S. Ilie, S. Khoshraftar, and A. Mansouri Bigvand. Seeds for effective oligonucleotide design. *BMC Genomics*, 12:280, 2011.

[12] S. M. Kiełbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith. Adaptive seeds tame genomic sequence comparison. *Genome Research*, 21:487–493, 2011.

[13] G. Kucherov, L. Noé, and M. A. Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. *Journal of Bioinformatics and Computational Biology*, 4:553–570, 2006.

[14] B. Langmeand and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9:357–359, 2012.

[15] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[16] T. Onodera and T. Shibuya. An index structure for spaced seed search. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 764–772, 2011.

[17] P. Peterlongo, N. Pisanti, F. Boyer, and M.-F. Sagot. Lossless filter for finding long multiple approximate repetitions using a new data structure, the bifactor array. In *Proceedings of the 12th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 179–190, 2005.

[18] L. M. S. Russo and G. Tischler. Succinct gapped suffix arrays. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 290–294, 2011.

[19] Y. Sun and J. Buhler. Designing multiple simultaneous seeds for DNA similarity search. *Journal of Computational Biology*, 12:847–861, 2005.

[20] K. J. Supowit. Decomposing a set of points into chains, with applications to permutation and circle graphs. *Information Processing Letters*, 21:249–252, 1985.

On Representations of Ternary Order Relations in Numeric Strings

Jinil Kim[1,*], Amihood Amir[2,3], Joong Chae Na[4], Kunsoo Park[1,*],

Jeong Seop Sim[5]

[1]Dep. of Computer Science and Engineering, Seoul National University, Korea
[2]Department of Computer Science, Bar-Ilan University, Israel
[3]Johns Hopkins University, USA
[4]Dep. of Computer Science and Engineering, Sejong University, Korea
[5]Dep. of Computer and Information Engineering, Inha University, Korea
[*]{jikim|kpark}@theory.snu.ac.kr

Abstract

Order-preserving matching is a string matching problem of two numeric strings where the relative orders of consecutive substrings are matched instead of the characters themselves. The order relation between two characters is a ternary relation $(>, <, =)$ rather than a binary relation $(>, <)$, but it was not sufficiently studied in previous works [5, 7, 1]. In this paper, we extend the representations of order relations by Kim et al. [5] to ternary order relations, and prove the equivalence of those representations. The *extended prefix representation* takes $\log m + 1$ bits per character, while the *nearest neighbor representation* takes $2 \log m$ bits per character. With our extensions, the time complexities of order-preserving matching in binary order relations can be achieved in ternary order relations as well.

1 Introduction

Order-preserving matching is a string matching problem of two numeric strings where the relative orders of substrings are matched instead of the characters themselves. It has many practical applications such as stock price analysis and musical melody matching. The study on this field was introduced by Kubica et al. [7] and Kim et al. [5] where Kubica et al. [7] defined order relations by order isomorphism of two strings, while Kim et al. [5] defined them explicitly by the sequence

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

of rank values (which they called the *natural representation*). Recently, variants of order-preserving matching have been studied such as order-preserving suffix trees [2], approximate matching with k mismatches [3], and a Boyer-Moore type algorithm [1].

An order relation between two characters is a ternary relation $(>, <, =)$ rather than a binary relation $(>, <)$, but the representations of order relations were not sufficiently studied for ternary order relations. Kim et al. [5] considered only binary order relations by assuming that all the characters in a string are distinct, while Kubica et al. [7] considered ternary order relations but their match condition on equal characters was faulty and it was fixed later by Cho et al. [1]. As there was no integrated study on the relationship between the representations of order relations, previous works [1, 2, 3] individually handled the cumbersome details of ternary order relations on their own works.

The number of order relations on a sequence of n elements is $n!$ in binary order relations, but it is the *ordered Bell number* [4] in ternary order relations. The ordered Bell number is the solution of the recurrence $f(n) = 1 + \sum_{j=1}^{n-1} \binom{n}{j} f(n-j)$, which is exponentially greater than $n!$. The representations of order relations should be extended for ternary order relations, which might incur some space overhead on the representations.

In this paper, we extend the representations of order relations by Kim et al. [5] to ternary order relations, and prove the equivalence of those representations. With the *extended prefix representation*, order-preserving matching can be done in $O(n \log m)$ time, and the representation of order relations takes $(\log m + 1)$-bit per character. With the *nearest neighbor representation*, the matching can be done in $O(n + m \log m)$, but the representation takes $(2 \log m)$-bit per character. The *nearest neighbor representation* is suitable for the single pattern matching while the *extended prefix representation* is space-efficient and can be useful in order-preserving multiple pattern matching [5] and order-preserving suffix trees [2].

2 Problem Formulation

Let Σ denote the set of numbers such that a comparison of two numbers can be done in constant time, and let Σ^* denote the set of strings over the alphabet Σ. For a string $x \in \Sigma^*$, let $|x|$ denote the length of x. A string x is described by a sequence of characters $(x[1], x[2], ..., x[|x|])$. Let a substring $x[i..j]$ be $(x[i], x[i+1], ..., x[j])$ and a prefix x_i be $x[1..i]$. For a character $c \in \Sigma$, let $rank_x(c) = 1 + |\{i : x[i] < c \text{ for } 1 \leq i \leq |x|\}|$, and let $exist_x(c)$ be 1 if c exists in x, and 0 otherwise. Let $\textit{ex-rank}_x(c) = (rank_x(c), exist_x(c))$. For any boolean condition *cond*, let $\delta(cond)$ be 1 if *cond* is true, 0 otherwise.

Order Isomorphism [7] For two strings x and y of length n, x and y are order-isomorphic if $\forall i, j \in [1..n]$, $x[i] \leq x[j] \Leftrightarrow y[i] \leq y[j]$.

Order isomorphism *implicitly* deals with ternary order relations because each of ternary order relations $(>, <, =)$ can be checked by variants of the proposition in Definition 2 (changing i and j, taking the contrapositive, or both). For example,

if $x[i] > x[j]$, then $y[i] > y[j]$ by the contrapositive of $x[i] \leq x[j] \Leftarrow y[i] \leq y[j]$. If $x[i] = x[j]$, then $y[i] = y[j]$ by $x[i] \leq x[j] \Rightarrow y[i] \leq y[j]$ and $x[j] \leq x[i] \Rightarrow y[j] \leq y[i]$.

The definition of order isomorphism looks simple, but it is somewhat complicated to handle in practice. The number of the order relations involved in checking order isomorphism of two strings of length n is $O(n^2)$, hence it has an inherent quadratic term if the definition is used directly for order-preserving matching.

Natural Representation [5] For a string x of length n, the natural representation of the order relations is defined as $Nat(x) = (rank_x(x[1]), rank_x(x[2]), ..., rank_x(x[n]))$.

In the natural representation, ternary order relations are *explicitly* stated in terms of ranks. For example, $x[i] > x[j]$ if and only if $Nat(x)[i] > Nat(x)[j]$, and $x[i] = x[j]$ if and only if $Nat(x)[i] = Nat(x)[j]$. That is, the order relations of two strings coincide if and only if $Nat(x) = Nat(y)$. The comparison of two natural representations takes $O(n)$ time if the natural representations are given.

These two definitions are equivalent because the natural representations of two strings are identical if and only if they are order-isomorphic. We adopt the natural representation throughout this paper because the definition itself and subsequent analysis are more intuitive.

Order-Preserving Matching [5] Given a text $T[1..n] \in \Sigma^*$ and a pattern $P[1..m] \in \Sigma^*$, P matches T at position i if $Nat(P) = Nat(T[i-m+1..i])$. Order-preserving matching is the problem of finding all positions of T matched with P.

Equivalent Representation For any two representations $R_1(\cdot)$ and $R_2(\cdot)$, R_1 is equivalent to R_2 if $R_1(x) = R_1(y) \Leftrightarrow R_2(x) = R_2(y)$ for any two strings x, y.

For KMP-based algorithms [6], the length of matches is incrementally increased when the next character of the text matches that of the pattern. Such a match operation is formalized by the *match condition* as follows.

Match Condition A match condition of a representation $R(\cdot)$ is a boolean function $Match(x, y, R(x), t+1)$ such that $Nat(x_{t+1}) = Nat(y_{t+1})$ holds if and only if $Nat(x_t) = Nat(y_t)$ and $Match(x, y, R(x), t+1)$ where x, $y \in \Sigma^*$, $|x| = |y|$ and $t \in [1..|x|-1]$.

3 Prefix Representation

Prefix Representation [5] For a string x, the prefix representation of the order relations is defined as $Pre(x) = (rank_{x_1}(x[1]), rank_{x_2}(x[2]), ..., rank_{x_{|x|}}(x[|x|]))$.

The *prefix representation* has an ambiguity between different strings in ternary order relations [1]. For example, when $x = (10, 30, 20)$, and $y = (10, 20, 20)$, the prefix representations of both x and y are $(1, 2, 2)$. The ambiguity is resolved in the extended prefix representation.

i	1	2	3	4	5	6	7	8
x	30	10	50	20	30	20	25	20
$Nat(x_6)$	4	1	6	2	4	2		
Ex-pre(x_7)	$\binom{1}{0}$	$\binom{1}{0}$	$\binom{3}{0}$	$\binom{2}{0}$	$\binom{3}{1}$	$\binom{2}{1}$	$\binom{4}{0}$	
$NN(x_7)$	$\binom{-\infty}{\infty}$	$\binom{-\infty}{1}$	$\binom{2}{1}$	$\binom{2}{1}$	$\binom{1}{1}$	$\binom{4}{4}$	$\binom{6}{5}$	
$Nat(x_7)$	5	1	7	2	5	2	4	
Ex-pre(x_8)	$\binom{1}{0}$	$\binom{1}{0}$	$\binom{3}{0}$	$\binom{2}{0}$	$\binom{3}{1}$	$\binom{2}{1}$	$\binom{4}{0}$	$\binom{2}{1}$
$NN(x_8)$	$\binom{-\infty}{\infty}$	$\binom{-\infty}{1}$	$\binom{2}{1}$	$\binom{2}{1}$	$\binom{1}{1}$	$\binom{4}{4}$	$\binom{6}{5}$	$\binom{6}{6}$
$Nat(x_8)$	6	1	8	2	6	2	5	2

Figure 1: Example of Lemma 3.1 and Lemma 4.1

Extended Prefix Representation For a string x, the extended prefix representation is defined as *Ex-pre*$(x) = ($*ex-rank*$_{x_1}(x[1])$, *ex-rank*$(x_2[2])$, ..., *ex-rank*$(x_{|x|}[|x|]))$.

For any $t \in [1..|x|-1]$, $Nat(x_{t+1})$ can be computed from $Nat(x_t)$ and *Ex-pre*(x_{t+1}).

Lemma 3.1 (Representation Conversion) *Given $Nat(x_t)$ and Ex-pre(x_{t+1}),*

$$Nat(x_{t+1})[i] = \begin{cases} a + \delta((a > c) \vee (a = c \wedge d = 0)) & \text{for } 1 \le i \le t \\ c & \text{for } i = t+1 \end{cases}$$

where $a = Nat(x_t)[i]$ and $\binom{c}{d} =$ Ex-pre$(x_{t+1})[t+1]$.

An example of Lemma 3.1 is shown in Figure 1 for $x = (30, 10, 50, 20, 30, 20, 25, 20)$. Let's consider when $t+1 = 7$. For $i = 1$, we have $Nat(x_6)[1] = a = 4$ and *Ex-pre*$(x_7)[7] = \binom{c}{d} = \binom{4}{0}$. Since $a = c$ and $d = 0$, we have $Nat(x_7)[1] = a + 1 = 5$. For $i = 2$, $\binom{c}{d}$ is the same as for $i = 1$, and we have $Nat(x_6)[2] = a = 1$. Since $a < c$, $Nat(x_7)[2] = a = 1$. Let's consider the next step when $t+1 = 8$. For $i = 4$, we have $Nat(x_7)[4] = a = 2$ and $\binom{c}{d} = \binom{2}{1}$. As $a = c$ and $d = 1$, $Nat(x_8)[4] = a = 2$.

Theorem 3.2 *Ex-pre$(\cdot)$ is equivalent to $Nat(\cdot)$.*

Theorem 3.3 (Match Condition) *Given x, y and t, the condition Ex-pre$(x_{t+1})[t+1] =$ Ex-pre$(y_{t+1})[t+1]$ is a match condition of Ex-pre$(\cdot)$.*

4 Nearest Neighbor Representation

Let $LMax_x[i] = j$ if $x[j] = \max\{x[k] : x[k] \le x[i] \text{ for } 1 \le k \le i-1\}$, or $-\infty$ if no such j. Let $LMin_x[i] = j$ if $x[j] = \min\{x[k] : x[k] \ge x[i] \text{ for } 1 \le k \le i-1\}$, or

∞ if no such j. If there are multiple j's for $LMax_x[i]$ or $LMin_x[i]$, we choose the rightmost one. In Figure 1, $LMax_x[8] = 6$ since $x[6]$ is the rightmost one among the maximum values which are less than or equal to $x[8]$ in $x[1..7]$. Similarly, $LMin_x[8] = 6$.

Nearest Neighbor Representation [5, 7, 1] For a string x, the nearest neighbor representation can be defined as

$$NN(x) = \binom{LMax_x[1]}{LMin_x[1]}\binom{LMax_x[2]}{LMin_x[2]}\cdots\binom{LMax_x[|x|]}{LMin_x[|x|]}$$

For convenience, let $x[-\infty] = -\infty$, $x[\infty] = \infty$, $Nat(x)[-\infty] = 0$ and $Nat(x)[\infty] = |x| + 1$ for any string x. Then, $Nat(x)[LMax_x[i]] \leq Nat(x)[i] \leq Nat(x)[LMin_x[i]]$ holds for any $i \in [1..|x|]$ even when $LMax_x[i] = -\infty$ or $LMin_x[i] = \infty$.

Lemma 4.1 (Representation Conversion) *Given $Nat(x_t)$ and $NN(x_{t+1})$,*

$$Nat(x_{t+1})[i] = \begin{cases} a + \delta((a > f) \vee (a = f \wedge e \neq f)) & \text{for } 1 \leq i \leq t \\ f & \text{for } i = t+1 \end{cases}$$

where $a = Nat(x_t)[i]$, $\binom{c}{d} = NN(x_{t+1})[t+1]$, $e = Nat(x_t)[c]$ and $f = Nat(x_t)[d]$.

An example of Lemma 4.1 is shown in Figure 1 for $x = (30, 10, 50, 20, 30, 20, 25, 20)$. Let us consider when $t + 1 = 7$. For $i = 1$, we have $Nat(x_6)[1] = a = 4$, $NN(x_7)[7] = \binom{c}{d} = \binom{6}{5}$, $e = 2$ and $f = 4$. Since $a = f$ and $e \neq f$, we get $Nat(x_7)[1] = a + 1 = 5$. For $i = 2$, $\binom{c}{d}$, e and f are the same as for $i = 1$, and we have $a = 1$. Since $a < e$, we have $Nat(x_7)[2] = a = 1$. For $i = 3$, we have $a = 6$ and $a > f$, and thus $Nat(x_7)[3] = a + 1 = 7$. For $i = 4$, we have $a = 2$. Since $a = e$ and $e \neq f$, we get $Nat(x_7)[4] = a = 2$. For $i = 7$, we get $Nat(x_7)[7] = f = 4$ since $i = t + 1$. Consider the next step when $t + 1 = 8$. For $i = 4$, we have $a = 2$, $NN(x_8)[8] = \binom{c}{d} = \binom{2}{1}$, $e = 2$ and $f = 2$. Since $a = e = f$, we get $Nat(x_8)[4] = a = 2$.

Theorem 4.2 *$NN(\cdot)$ is equivalent to $Nat(\cdot)$.*

A naive match condition of the nearest neighbor representation is $NN(x_{t+1})[t+1] = NN(y_{t+1})[t+1]$ as that of the extended prefix representation in Theorem 3.3, which requires computing the nearest neighbor representations of both x and y. Kubica et al. [7] proposed an efficient match condition for ternary order relations which can be checked in constant time when the nearest neighbor representation of x is given, but it was faulty. Cho et al. [1] presented a modified match condition in ternary order relations, which can produce an $O(n + m \log m)$ algorithm as in binary order relations.

Theorem 4.3 (Match Condition of Nearest Neighbor Representation [1]) *Given x, y and t, the condition $(y[c] < y[t+1] < y[d]) \vee (y[t+1] = y[c] = y[d])$ is a match condition of $NN(\cdot)$ where $\binom{c}{d} = NN(x_{t+1})[t+1]$.*

We can generalize order-preserving matching algorithms in binary order relations [5] to ternary order relations using the representations above. For single pattern matching, we can obtain an $O(n \log m)$ algorithm with the extended prefix representation, and an $O(n + m \log m)$ algorithm with the nearest neighbor representation, both of which are consistent with the results in [5, 7, 1]. For multiple pattern matching, we can obtain an $O((n + m) \log m)$ algorithm in ternary order relations using the extended prefix representation.

Acknowledgements

The work of Jinil Kim and Kunsoo Park was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2011-0029924). Amihood Amir was partly supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217. Joong Chae Na was supported by the IT R&D program of MKE/KEIT [10038768, The Development of Supercomputing System for the Genome Analysis], and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2011-0007860). Jeong Seop Sim was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2012R1A2A2A01014892), and by the Industrial Strategic technology development program (10041971, Development of Power Efficient High-Performance Multimedia Contents Service Technology using Context-Adapting Distributed Transcoding) funded by the Ministry of Knowledge Economy (MKE, Korea).

References

[1] S. Cho, J. C. Na, K. Park, and J. S. Sim. Fast order-preserving pattern matching. In *COCOA*, 2013.

[2] 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 *SPIRE*, pages 84–95, 2013.

[3] P. Gawrychowski and P. Uznanski. Order-preserving pattern matching with k mismatches. *CoRR*, abs/1309.6453, 2013.

[4] O. A. Gross. Preferential arrangements. *The American Mathematical Monthly*, 69:4–8, 1962.

[5] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. *To appear in Theor. Comput. Sci.*

[6] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[7] 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):430–433, 2013.

Engineering a Lightweight External Memory Suffix Array Construction Algorithm*

Juha Kärkkäinen, Dominik Kempa

Department of Computer Science, University of Helsinki, Finland
{Juha.Karkkainen|Dominik.Kempa}@cs.helsinki.fi

Abstract

We describe an external memory suffix array construction algorithm based on constructing suffix arrays for blocks of text and merging them into the full suffix array. The basic idea goes back over 20 years and there has been a couple of later improvements, but we describe several further improvements that make the algorithm much faster. In particular, we reduce the I/O volume of the algorithm by a factor $\mathcal{O}(\log_\sigma n)$. Our experiments show that the algorithm is the fastest suffix array construction algorithm when the size of the text is within a factor of about five from the size of the RAM in either direction, which is a common situation in practice.

1 Introduction

The suffix array [12, 9], a lexicographically sorted array of the suffixes of a text, is the most important data structure in modern string processing. It is the basis of powerful text indexes such as enhanced suffix arrays [1] and many compressed full-text indexes [14]. Modern text books spend dozens of pages in describing applications of suffix arrays, see e.g. [16]. In many of the applications, the construction of the suffix array is the main bottleneck in space and time, even though a great effort has gone into developing better algorithms [17].

For internal memory, there exists an essentially optimal suffix array construction algorithm (SACA) that runs in linear time using little extra space in addition to what is needed for the input text and the output suffix array [15]. However, little *extra* space is not good enough for large text collections such as web crawls, Wikipedia

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

* Supported by the Academy of Finland grant 118653 (ALGODAN).

or genomic databases, which may be too big for holding even the text alone in RAM. There are also external memory SACAs that are theoretically optimal with internal work $\mathcal{O}\Big(n \log_{M/B}(n/B)\Big)$ and I/O complexity $\mathcal{O}\Big((n/B)\log_{M/B}(n/B)\Big)$ [11, 4], where M is the size of the RAM and B is the disk block size. Furthermore, they are practical and have implementations [7, 4] that are fully scalable in the sense that they are not seriously limited by the size of the RAM. However, constant factors in practical running time and disk space usage are significant. The currently best implementation, eSAIS [4], needs $28n$ bytes of disk space, which is probably the main factor limiting its practical scalability.

In this paper, we focus on an algorithm, which we call SAscan, that lies between internal memory SACAs and eSAIS in scalability. SAscan is a true external memory algorithm in the sense that it can handle texts that do not fit in internal memory, but its time complexity $\Omega(n^2/M)$ makes it hopelessly slow when the text is much larger than the RAM. However, when the text is too large for an internal memory SACA, i.e., larger than about one fifth of the RAM size, but not too much bigger than the RAM, SAscan is probably the fastest SACA in practice. SAscan is also lightweight in the sense that it uses less than half of the disk space of eSAIS, and can be implemented to use little more than what is needed for the text and the suffix array.

The basic approach of SAscan was developed already in the early days of suffix arrays by Gonnet, Baeza-Yates and Snider [9]. The idea is to partition the text into blocks that are small enough so that the suffix array for each block can be constructed in RAM. The block suffix arrays are then merged into the full suffix array. After constructing the suffix array for each block, the algorithm scans the previously processed part of the text and determines how each suffix of the scanned text compares to the suffixes of the current block. The information collected during the scan is then used for performing the merging.

The early version of SAscan depended heavily on the text not having long repeats and thus had a poor worst case time complexity. This problem was solved by Crauser and Ferragina [6], who developed an improved version with worst case time complexity $\mathcal{O}\big((n^2/M)\log M\big)$ and I/O complexity of $\mathcal{O}\big(n^2/(MB)\big)$. The algorithm was further improved by Ferragina, Gagie and Manzini [8], who reduced the time complexity to $\mathcal{O}\big((n^2/M)\log\sigma\big)$, where σ is the size of the text alphabet, and the disk space usage to little more than what is needed for the input and the output.

In this paper, we describe several further improvements to the SAscan algorithm. The first improvement is a new merging technique that reduces the I/O complexity of SAscan to $\mathcal{O}\big(n^2/(MB\log_\sigma n) + n/B\big)$ and provides a substantial speedup in practice too. Another target of improvement is the rank data structure that plays a key role in the algorithm. In theory, we observe that the time complexity can be reduced to $\mathcal{O}\big((n^2/M)\log(2 + (\log\sigma/\log\log n))\big)$ by plugging in the rank data structure of Belazzougui and Navarro [3]. In practice, we improve the rank data structure used in the implementation by Ferragina, Gagie and Manzini by applying alphabet partitioning [2] and fixed block boosting [10]. Finally, we reduce the size of the internal memory data structures by more than one third, which allows the algorithm to use bigger and fewer blocks improving the running time significantly.

We show experimentally that our practical improvements reduce the running time of SAscan by more than a factor of three. We also show that the algorithm is faster than eSAIS when the text size is less than about six times the available RAM, at which point the disk space usage of eSAIS is already well over 150 times the available RAM.

2 Preliminaries

Let $\mathsf{X} = \mathsf{X}[0..m)$ be a string over an integer alphabet $[0..\sigma)$. For $i = 0, \ldots, m-1$ we write $\mathsf{X}[i..m)$ to denote the *suffix* of X of length $m - i$, that is $\mathsf{X}[i..m) = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[m-1]$. Similarly, we write $\mathsf{X}[i..j)$ to denote the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j-1]$ of length $j-i$. If $i = j$, the substring $\mathsf{X}[i..j)$ is the empty string, also denoted by ε.

The suffix array SA_X of X contains the starting positions of the non-empty suffixes of X in the lexicographical order, i.e., it is an array $\mathsf{SA}_\mathsf{X}[0..m)$ which contains a permutation of the integers $[0..m)$ such that $\mathsf{X}[\mathsf{SA}_\mathsf{X}[0]..m) < \mathsf{X}[\mathsf{SA}_\mathsf{X}[1]..m) < \cdots < \mathsf{X}[\mathsf{SA}_\mathsf{X}[m-1]..m)$.

The partial suffix array $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ is the lexicographical ordering of the suffixes of XY with a starting position in X, i.e., it is an array $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[0..m)$ that contains a permutation of the integers $[0..m)$ such that $\mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[0]..m)\mathsf{Y} < \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[1]..m)\mathsf{Y} < \cdots < \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[m-1]..m)\mathsf{Y}$. Note that $\mathsf{SA}_{\mathsf{X}:\varepsilon} = \mathsf{SA}_\mathsf{X}$ and that $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ is usually similar but not identical to SA_X. Also, $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ can be obtained from $\mathsf{SA}_{\mathsf{XY}}$ by removing all entries that are larger or equal to m.

3 Overview of the Algorithm

Let a string $\mathsf{T}[0..n)$ be the text. It is divided into blocks of size (at most) m, where m is chosen so that all the in-memory data structures of $\mathcal{O}(m \log n)$ bits fit in the RAM. The blocks are processed starting from the end of the text. Assume that so far we have processed $\mathsf{Y} = \mathsf{T}[i..n)$ and constructed the suffix array SA_Y. Next we will construct the partial suffix array $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ for the block $\mathsf{X} = \mathsf{T}[i-m..i)$ and merge it with SA_Y to form $\mathsf{SA}_{\mathsf{XY}}$.

The suffixes in $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ and SA_Y are in the same relative order as in $\mathsf{SA}_{\mathsf{XY}}$ and we just need to know how to merge them. For this purpose, we compute the *gap array* $\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}[0..m]$, where $\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}[i]$ is the number of suffixes of Y that are lexicographically between the suffixes $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[i-1]$ and $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[i]$ of XY. Formally, for $i \in [1..m)$,

$$\begin{aligned}
\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}[0] &= |\{j \in [0..|\mathsf{Y}|) : \mathsf{Y}[j..|\mathsf{Y}|) < \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[0]..m)\mathsf{Y}\}| \\
\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}[i] &= |\{j \in [0..|\mathsf{Y}|) : \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[i-1]..m)\mathsf{Y} < \mathsf{Y}[j..|\mathsf{Y}|) < \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[i]..m)\mathsf{Y}\}| \\
\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}[m] &= |\{j \in [0..|\mathsf{Y}|) : \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[m-1]..m)\mathsf{Y} < \mathsf{Y}[j..|\mathsf{Y}|)\}| \,.
\end{aligned}$$

The construction of $\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}$ scans Y and is the computational bottleneck of the algorithm.

The merging of $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ and SA_Y is trivial with the help of $\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}$ but involves a lot of I/O for reading SA_Y and writing $\mathsf{SA}_{\mathsf{XY}}$. The total I/O volume is $\mathcal{O}(n^2/m)$ in units of $\mathcal{O}(\log n)$-bit words. However, we can reduce the I/O by delaying the

merging. We write $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ to disk and then proceed to process the next block. Once all partial suffix arrays and gap arrays have been computed, we perform one multiway merging of the partial suffix arrays with the help of the gap arrays. The I/O volume for merging is reduced to $\mathcal{O}(n)$ words.

Suppose that during the construction of $\mathsf{SA}_{\mathsf{X:Y}}$ or $\mathsf{gap}_{\mathsf{X:Y}}$ we need to compare two suffixes of $\mathsf{SA}_{\mathsf{XY}}$, at least one of which begins in X. In the worst case, the suffixes could have a very long common prefix, much longer than m, making it impossible to perform the comparison without a lot of I/O — unless we have extra information about the order of the suffixes. In our case, that extra information is provided by a bitvector gt_{Y}, which tells whether each suffix of Y is lexicographically smaller or larger than Y itself. Formally, for all $i \in [0..|\mathsf{Y}|)$,

$$\mathsf{gt}_{\mathsf{Y}}[i] = \begin{cases} 1 & \text{if } \mathsf{Y}[i..|\mathsf{Y}|) > \mathsf{Y} \\ 0 & \text{if } \mathsf{Y}[i..|\mathsf{Y}|) \leq \mathsf{Y} \end{cases}$$

With the help of gt_{Y}, two suffixes of XY, at least one of which begins in X, can be compared in $\mathcal{O}(m)$ time. The algorithms for constructing $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ perform more complex operations than plain comparisons, but they use the same bitvector to avoid extra I/Os resulting from long common prefixes.

In summary, for each text block X we perform the following steps:

1. Given X, $\mathsf{Y}[0..m)$ and $\mathsf{gt}_{\mathsf{Y}}[0..m)$, compute $\mathsf{SA}_{\mathsf{X:Y}}$.
2. Given X, $\mathsf{SA}_{\mathsf{X:Y}}$, Y and gt_{Y}, compute $\mathsf{gap}_{\mathsf{X:Y}}$ and $\mathsf{gt}_{\mathsf{XY}}$.

The output bitvector $\mathsf{gt}_{\mathsf{XY}}$ is needed as input for the next block. The two other arrays $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ are stored on disk until all blocks have been processed. The final phase of the algorithm takes all the partial suffix arrays and gap arrays as input and produces the full suffix array SA_{T}.

4 Details and Analysis

The first stage in processing a block X is constructing the partial suffix array $\mathsf{SA}_{\mathsf{X:Y}}$. In the full paper, we show how to construct a string Z such that $\mathsf{SA}_{\mathsf{Z}} = \mathsf{SA}_{\mathsf{X:Y}}$. We can then construct the suffix array using any standard SACA; In the implementation we use Yuta Mori's divsufsort [13].

The partial Burrows–Wheeler transform [5] of X is an array $\mathsf{BWT}_{\mathsf{X:Y}}[0..m)$ defined by:

$$\mathsf{BWT}_{\mathsf{X:Y}}[i] = \begin{cases} \mathsf{X}[\mathsf{SA}_{\mathsf{X:Y}}[i]-1] & \text{if } \mathsf{SA}_{\mathsf{X:Y}}[i] > 0 \\ \$ & \text{if } \mathsf{SA}_{\mathsf{X:Y}}[i] = 0 \end{cases},$$

where \$ is a special symbol that does not appear in the text. For a character c and an integer $i \in [0..m]$, the answer to the rank query $\mathsf{rank}_{\mathsf{BWT}_{\mathsf{X:Y}}}(c, i)$ is the number of occurrences of c in $\mathsf{BWT}_{\mathsf{X:Y}}[0..i)$. Rank queries can be answered in $\mathcal{O}(\log(2 + (\log\sigma/\log\log n)))$ time using a linear space data structure [3]. In practice, we use a simpler data structure, described in the full paper, that requires $4.125m$ bytes of space.

For a string S, let $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{S})$ be the number of suffixes of XY starting in X that are lexicographically smaller than S. Let $\mathsf{C}[0..\sigma)$ be an array, where $\mathsf{C}[c]$ is the number of positions $i \in [0..m)$ such that $\mathsf{X}[i] < c$. In the full paper, we prove the following lemma.

Lemma 4.1 *Let* $k = \mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{S})$ *for a string* S*. For any symbol* c*,*

$$\mathsf{sufrank}_{\mathsf{X:Y}}(c\mathsf{S}) = \mathsf{C}[c] + \mathsf{rank}_{\mathsf{BWT}_{\mathsf{X:Y}}}(c,k) + \begin{cases} 1 & \textit{if } \mathsf{X}[m-1] = c \textit{ and } \mathsf{Y} < \mathsf{S} \\ 0 & \textit{otherwise} \end{cases} .$$

Note that when $\mathsf{S} = \mathsf{Y}[j..|\mathsf{Y}|)$, we can replace the comparison $\mathsf{Y} < \mathsf{S}$ with $\mathsf{gt}_{\mathsf{Y}}[j] = 1$. Thus, given $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{Y}[j..|\mathsf{Y}|))$, we can easily compute $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{Y}[j-1..|\mathsf{Y}|))$ using the lemma, and we only need to access $\mathsf{Y}[j-1]$ in Y and $\mathsf{gt}_{\mathsf{Y}}[j]$ in gt_{Y}. Hence, we can compute $\mathsf{sufrank}_{\mathsf{X:Y}}(\mathsf{Y}[j..|\mathsf{Y}|))$ for $j = |\mathsf{Y}|-1,\dots,0$ with a single sequential pass over Y and gt_{Y}. This is all that is needed to compute $\mathsf{gap}_{\mathsf{X:Y}}$ and $\mathsf{gt}_{\mathsf{XY}}$, which are the output of the second stage of processing X.

The final phase of the algorithm is merging the partial suffix arrays into the full suffix array SA_{T}. For $k \in [0..\lceil n/m \rceil)$, let $\mathsf{X}_k = \mathsf{T}[km..(k+1)m)$, $\mathsf{Y}_k = \mathsf{T}[(k+1)m..n)$, $\mathsf{SA}_k = \mathsf{SA}_{\mathsf{X}_k:\mathsf{Y}_k}$ and $\mathsf{gap}_k = \mathsf{gap}_{\mathsf{X}_k:\mathsf{Y}_k}$. The algorithm shown below moves suffixes from the input suffix arrays to the output suffix array in ascending lexicographical order.

1: **for** $k = 0$ **to** $\lceil n/m \rceil - 1$ **do** $i_k \leftarrow 0$
2: **for** $i = 0$ **to** $n-1$ **do**
3: $\quad k \leftarrow 0$
4: $\quad$ **while** $\mathsf{gap}_k[i_k] > 0$ **do**
5: $\quad\quad \mathsf{gap}_k[i_k] \leftarrow \mathsf{gap}_k[i_k] - 1$
6: $\quad\quad k \leftarrow k+1$
7: $\quad \mathsf{SA}_{\mathsf{T}}[i] \leftarrow \mathsf{SA}_k[i_k] + km$
8: $\quad i_k \leftarrow i_k + 1$

The correctness of the algorithm is based on the following invariants maintained by the algorithm: (1) i_k is the number of suffixes already moved from SA_k to SA_{T}, and (2) $\mathsf{gap}_k[i_k]$ is the number of suffixes remaining in SA_{k+1}, SA_{k+2}, ..., $\mathsf{SA}_{\lceil n/m \rceil - 1}$ that are smaller than $\mathsf{SA}_k[i_k]$.

Theorem 4.2 *SAscan can be implemented to construct the suffix array of a text of length* n *over an alphabet of size* σ *in* $\mathcal{O}\left(\frac{n^2}{M}\log\left(2+\frac{\log\sigma}{\log\log n}\right)\right)$ *time and* $\mathcal{O}\left(\frac{n^2\log\sigma}{MB\log n} + \frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$ *I/Os in the standard external memory model (see [18]) with RAM size* M *and disk block size* B*, both measured in units of* $\Theta(\log n)$*-bit words. Under the reasonable assumption that* $M \ge B\log_\sigma n$*, the I/O complexity is* $\mathcal{O}\left(\frac{n}{B}\left(1+\frac{n\log\sigma}{M\log n}\right)\right)$.

In the full paper, we describe an implementation that needs $5.2m$ bytes of RAM and at most $11.5n$ bytes of disk space, and could be implemented to use just $6.5n$ bytes of disk space.

5 Experimental Results

We performed experiments on a machine with a 3.16GHz Intel Core 2 Duo CPU with 6144KiB L2 cache running Linux (Ubuntu 12.04, 64bit, kernel 3.2). All

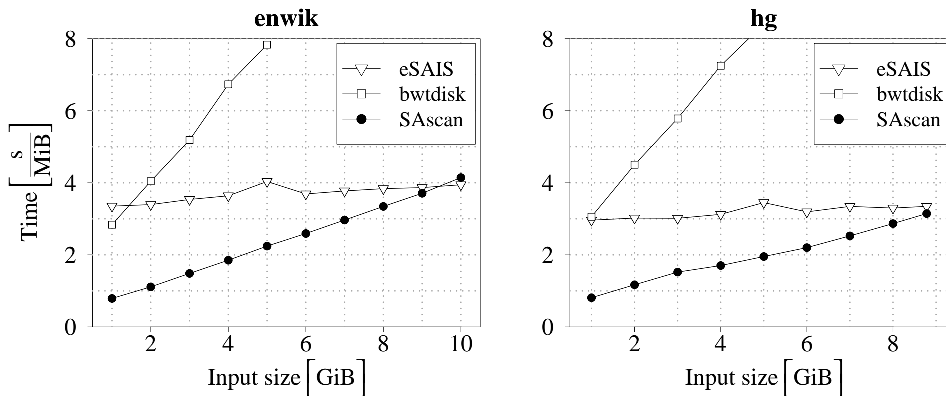

Figure 1: Scalability of SAscan compared to eSAIS and bwtdisk.

programs were compiled using g++ version 4.6.4 with -O3 -DNDEBUG options. To reduce the time to run the experiments, we artificially restricted the RAM size to 2GiB using the Linux boot option mem, and the algorithms were allowed to use at most 1.5GiB. We used two big test files: a concatenation of three different Human genomes[1,2,3] (hg) and a prefix of the English Wikipedia dump[4] (enwik).

The first experiment measures the scalability of our new algorithm. We computed the suffix array for varying length prefixes of each testfile using our algorithm and compared to eSAIS – currently the fastest algorithm for building SA in external memory. In addition, we also show the runtimes of bwtdisk[5], which is essentially the Ferragina–Gagie–Manzini version of SAscan though it constructs the BWT instead of the suffix array. The suffix array version of bwtdisk would be slower as it needs more I/O during merging. The results are given in Figure 1. SAscan is faster than eSAIS up to input sizes of about 9GiB, which is about six times the size of the RAM available to the algorithms. It is this ratio between the input size and the RAM size that primarily determines the relative speeds of SAscan and eSAIS. Note that the main limitation to the scalability of eSAIS is the disk space requirement, which is about 170 times the available RAM at the crossing point. The runtime of bwtdisk is always at least 3.5 times the runtime of SAscan, showing the dramatic effect of our improvements. In the second experiment, we take a closer look at the effect of our improvements on the runtime. More precisely, we show a detailed runtime breakdown after turning on individual improvements one by one: the fast merging of partial suffix arrays, the space-efficient representation of the gap array (which reduces the RAM usage from $8m$ to $5.2m$ bytes), and the optimized rank data structure. The results are presented in Figure 2. Each of the improvements produces a significant speedup. The combined effect is more than a factor of three.

[1] http://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/

[2] ftp://public.genomics.org.cn/BGI/yanhuang/fa/

[3] ftp.ncbi.nlm.nih.gov/genbank/genomes/Eukaryotes/vertebrates_mammals/Homo_sapiens/

[4] http://dumps.wikimedia.org/enwiki/

[5] http://people.unipmn.it/manzini/bwtdisk/

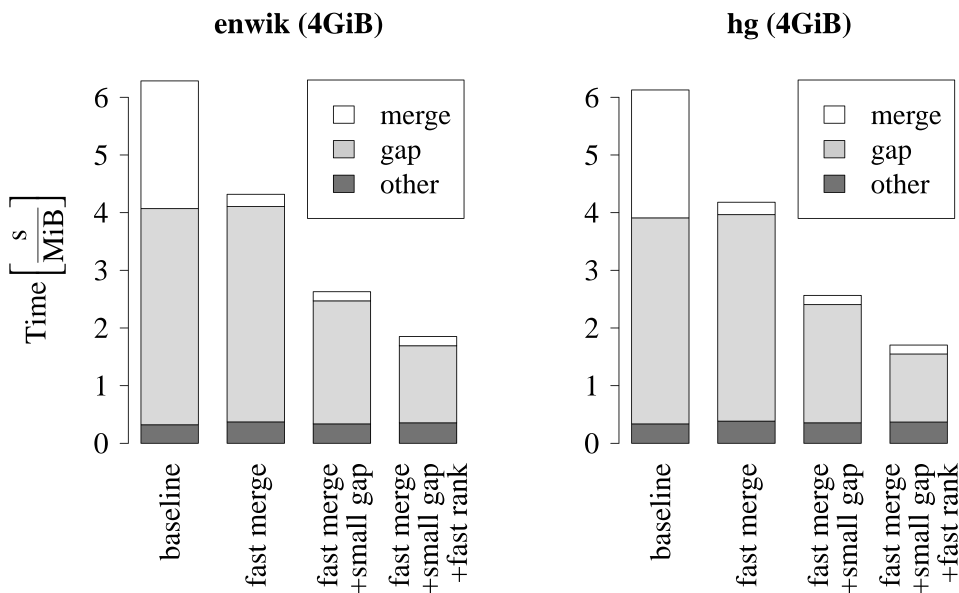

Figure 2: Effects of various optimizations on runtime. We separated the runtime into three components: merging suffix arrays, gap array construction and other ($\mathcal{O}(n)$ time) computations.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[2] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. ISAAC*, pages 315–326, 2010.

[3] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. ESA*, pages 181–192, 2012.

[4] T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Proc. ALENEX*, pages 103–112, 2013.

[5] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[6] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[7] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM J. Experimental Algorithmics*, 12:Article 3.4, 2008.

[8] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[9] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and Pat arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice–Hall, 1992.

[10] J. Kärkkäinen and S. J. Puglisi. Fixed-block compression boosting in FM-indexes. In *Proc. SPIRE*, pages 174–184, 2011.

[11] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[12] U. Manber and G. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.

[13] Y. Mori. libdivsufsort, a C library for suffix array construction. `http://code.google.com/p/libdivsufsort/`.

[14] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

[15] G. Nong. Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):Article 15, 2013.

[16] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

[17] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.

[18] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

Faster Average Case Low Memory Semi-External Construction of the Burrows-Wheeler Transform

German Tischler*

The Wellcome Trust Sanger Institute, Wellcome Trust Genome Campus
Hinxton, Cambridge, CB10 1SA, United Kingdom
german.tischler@sanger.ac.uk

Abstract

The Burrows Wheeler transform has applications in data compression as well as full text indexing. Despite its important applications and various existing algorithmic approaches the construction of the transform for large data sets is still challenging. In this paper we present a new semi external memory algorithm capable of constructing the transform in time $O(n \log^2 \log n)$ on average if sufficient internal memory is available to hold a fixed fraction of the input text. In the worst case the run-time is $O(n \log n \log \log n)$.

1 Introduction

The Burrows-Wheeler transform (BWT) was introduced to facilitate the lossless compression of data (cf. [3]). It has an intrinsic connection to some data structures used for full text indexing like the suffix array (cf. [11]) and is at the heart of some compressed full text self indexes like the FM index (see [8]). The FM index requires no more space than the k'th order entropy compressed input text plus some asymptotically negligible supporting data structures. Many construction algorithms for the BWT are based on its relation to the suffix array, which can be computed from the input text in time linear in the length of that text (see e.g. [10, 13]). While these algorithms run in linear time and are thus theoretically optimal they require $O(n \log n)$[1] bits of space for the uncompressed suffix array given an input text of

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

* Supported by the Wellcome Trust

[1] by log we mean $\log_2$ in this paper

length n while the text itself can be stored in a space of $n\lceil\log\sigma\rceil$ bits for an alphabet of size σ where we often have $\sigma \ll n$ and in most applications σ is constant. Algorithms for computing the suffix array in external memory have been proposed (see e.g. [2, 5]) but these algorithms require large amounts of space and input/output in external memory. An asymptotically optimal internal memory solution concerning time and space has been proposed [9]. However the space usage of this algorithm is $O(n)$ bits for constant alphabets, where an inspection of the algorithm suggests that the actual practical memory usage of the algorithm is several times the size of the text in bits. The practical space usage of the algorithm subsequently presented in [14] is lower (i.e. the involved constants are smaller) while theoretically not linear. It however still requires multiple times as much space as the input text. A sample implementation given by the authors only works for input sizes of up to 2^{32} (see [1]) and only handles a single level of the recursive algorithm. Given the implementation complexity of the algorithm it remains unclear if it would scale well. Crochemore et al present an algorithm computing the BWT in quadratic time with constant additional space (see [4]). In [1] Beller et al propose a semi external algorithm for the construction of the BWT based on induced sorting. An algorithm is called semi external if it uses non negligible amounts of internal as well as external memory. According to the authors the algorithm scales to arbitrary input sizes and uses about one byte (i.e. 8 bits) per input symbol in internal memory. An algorithm constructing the BWT of a text by block-wise merging using a finite amount of internal memory is presented in [7]. The algorithm partitions the text into a set of fixed size blocks. The run-time is $O(n^2/b)$ for a block size of b and a text length of n. It requires an amount of internal memory which is roughly sufficient to compute the suffix array of a single of these block. In particular the amount of internal memory used can be smaller than the space required for the text. In this paper we modify this algorithm to run in time $O(n\log n\log\log n)$ in the worst case and $O(n\log^2\log n)$ on average for the case where we are able to keep a fixed fraction of the text in memory. Assuming the practically common case of a finite alphabet the algorithm in [7] uses blocks of size $O(n/\log n)$ blocks when provided with $O(n)$ bits of space in internal memory so its total run-time for this setting is $O(n\log n)$. In consequence our algorithm is faster on average and slower by $O(\log\log n)$ for a very unlikely worst case. Compared to the algorithm presented in [1] our algorithm can work with less internal memory. For DNA for instance the complete text can be held in memory using about 2 bits per symbol which is significantly less than a full byte (8 bits) per character.

2 Definitions

For a string $s = s_0s_1s_2\ldots s_{m-1}$ of length $|s| = m$ we define $s[i] = s_i$ for $0 \leq i < m$ and for $s = s_0s_1\ldots$ we define $s[i] = s_i$ for $0 \leq i$. For a finite word u and a finite or infinite word v we write their concatenation as uv. For any finite words u, x and finite or infinite words w, v such that $w = uxv$ we call u a prefix, v a suffix and x a factor of w. The empty word consisting of no symbols is denoted by ϵ. For a string s and indices $0 \leq i \leq j < |s|$ we denote the factor $s[i]s[i+1]\ldots s[j]$ by $s[i, j]$. For any i, j such that $i > j$ the term $s[i, j]$ denotes the empty word. A finite word w

has period p iff $w[i] = w[i+p]$ for $i = 0, \ldots, |w| - p - 1$ and an infinite word w has period p iff $w[i] = w[i+p]$ for $i = 0, 1, \ldots$. For a finite word u and $k \in \mathbb{N}$ the k'th power u^k of u is defined by $u^0 = \epsilon$ and $u^{i+1} = u^i u$ for $i \in \mathbb{N}$. A word w is primitive if it is not a power of a word u such that $|u| < |w|$. A word u is a root of w if $w = u^k$ for some $k \in \mathbb{N}$. A word w is a square if there is a word u such that $w = u^2$. Throughout this paper let $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ denote a finite alphabet for some $\sigma > 0$ and let $t = t_0 t_1 \ldots t_{n-1} \in \Sigma^n$ denote a finite string of length $n > 0$. We define the semi infinite string $\tilde{t}$ by $\tilde{t}[i] = t[i - \lfloor i/n \rfloor n]$ for $i \geq 0$. We define the suffix $\tilde{t}_i$ of $\tilde{t}$ as $\tilde{t}_i = \tilde{t}[i]\tilde{t}[i+1]\ldots$ and $\tilde{t}_i < \tilde{t}_j$ for $i, j \in \mathbb{N}, i \neq j$ iff either $\tilde{t}_i = \tilde{t}_j$ and $i < j$ or for the smallest $\ell \geq 0$ such that $\tilde{t}[i+\ell] \neq \tilde{t}[j+\ell]$ we have $\tilde{t}[i+\ell] < \tilde{t}[j+\ell]$. The suffix array A of t is defined as the permutation of the numbers $0, 1, \ldots, n-1$ such that $\tilde{t}_{A[i]} < \tilde{t}_{A[i+1]}$ for $i = 0, 1, \ldots, n-2$ and the Burrows-Wheeler transform (BWT) $B = b_0 b_1 \ldots b_{n-1}$ of t is given by $b_i = \tilde{t}[A[i] + n - 1]$ for $i = 0, 1, \ldots, n-1$.

3 BWT construction by block-wise merging

We give a short high level description of the algorithm by Ferragina et al. in [7] as we will be modifying it. Unlike our algorithm it assumes the input string to have a unique minimal terminator symbol. Given a block size b the input string t is partitioned into $c = \lceil n/b \rceil$ blocks $T_0, T_1, \ldots, T_{c-1}$ of roughly equal size. The algorithm starts by suffix sorting the last block, computing its BWT B_{c-1} and the bit array gt_{c-1} which denotes for each suffix in T_{c-1} but the first whether it is smaller or larger than the first. The BWT of $T_i \ldots T_{c-1}$ for $i < c-1$ is computed by first computing the suffix array for the suffixes starting in T_i by using the text of T_i and T_{i+1} in memory and handling the comparison of suffixes starting in T_i but equal until both have entered T_{i+1} by using the bit vector gt_{i+1} which explicitly stores the result of this comparison. The BWTs of T_i and $T_{i+1}T_{i+2}\ldots T_{c-1}$ are merged by computing the ranks of the suffixes starting in $T_{i+1}T_{i+2}\ldots T_{c-1}$ in the sorted set of suffixes of T_i and computing a gap array G_i which denotes how many suffixes from $T_{i+1}T_{i+2}\ldots T_{c-1}$ are to be placed before the suffixes in T_i, between two adjacent suffixes in T_i and after all suffixes in T_i. This process follows a backward search of $T_{i+1}T_{i+2}\ldots T_{c-1}$ in T_i. Using the array G_i it is simple to merge the two BWTs together. For computing the rank of a suffix from $T_{i+1}\ldots T_{c-1}$ it is necessary to know whether it is smaller or larger than the one at the start of $T_{i+1}T_{i+2}\ldots T_{c-1}$ as B_i is not a conventional BWT. For further details about the algorithm the reader is referred to [7].

4 Sorting single blocks

The algorithm by Ferragina et al processes each single block relying on knowledge about the priorly fully processed following block, in case of the last block the terminator. For our algorithm we need to be able to sort a single block without knowing the complete sorted order of the next block. For this purpose we need to be able to handle repetitions, one of the major challenges along the way, efficiently. For our block sorting only repetitions with a period not exceeding the maximum block size are relevant. Consider a block of b suffixes $\tilde{t}_{i+j}$ for some $i \in \mathbb{N}^+, 0 \leq j < b$. We say it generates a repetition with period $p, 1 \leq p \leq b$ iff $\tilde{t}[b-p, b-1] = \tilde{t}[b, b+p-1]$

and propagates a repetition with period $p, 1 \leq p \leq b$ iff $\tilde{t}_i[0, b+2p-1]$ has period p. If it propagates repetitions of any periods, then there is a unique minimal period dividing all other propagated periods. This unique minimal period can then be computed in time $O(b)$ and space $O(b \log \sigma)$ bits using minor modifications of standard string algorithms. As there is a unique minimal period propagated by a block if any and we are only interested in generated periods which are propagated by the next block we can compute the relevant generation properties of a block in the same time and space bounds. Given a target block size b' we partition the given text into a set of blocks of size either $b = \lceil \frac{n}{\lceil (n/b') \rceil} \rceil \leq b'$ or $b-1$ where the first $n \bmod b$ blocks have length b and the rest length $b-1$. For the propagation of repetitions we extend the blocks of length $b-1$ to size b by adding the (circularly) next character to the right. Using this information about short period repetitions in the input string, we are able to handle the sorting of a single block of suffixes extending beyond the end of the block efficiently by reducing long repetitions.

Lemma 4.1 *A block of b circular suffixes of $\tilde{t}$ can be sorted in lexicographical order using time $O(b)$ and space $O(b \log b)$ bits using precomputed repetition propagation data.*

For forward searching using the suffix array it is useful to in addition have the longest common prefix (LCP) array. For two strings u, v let $\text{LCP}(u, v) = \text{argmax}_{l=0}^{\min\{|u|,|v|\}} u[0, \ell-1] = v[0, \ell-1]$. For a block $\tilde{t}[i, i+b-1]$ for $i, b \in \mathbb{N}, b > 0$ let $\mathcal{A}$ denote the permutation of $i, i+1, \ldots, i+b-1$ such that $\tilde{t}_{\mathcal{A}[j]} < \tilde{t}_{\mathcal{A}[j+1]}$ for $j = 0, 1, \ldots, b-2$. Then the LCP array of the block is defined by $\text{LCP}[0] = 0$ and $\text{LCP}[i] = \text{LCP}(\tilde{t}_{\mathcal{A}[i-1]}, \tilde{t}_{\mathcal{A}[i]})$ for $i = 1, 2, \ldots, b-1$. Using a repetition reduction method similar to the suffix sorting case we obtain the following result.

Lemma 4.2 *The LCP array for a block of b circular suffixes on $\tilde{t}$ can be computed in time $O(b)$ and space $O(b \log b)$ bits using precomputed repetition propagation data.*

5 Merging Pairs of Adjacent Blocks

In our modified algorithm we replace the completely skewed binary merge tree used in [7] by a balanced binary merge tree. Consequently we will need to be able to merge blocks with a block size in $\Omega(n)$. For merging two adjacent blocks we need the following components:

1. The BWT of the left and right block. These can be compressed and in external memory as they will be scanned sequentially.
2. An internal memory index of the left block suitable for backward search in $O(1)$ time per step. An FM type index using space $b_l H_k + o(n \log \sigma)$ bits can be used where b_l is the length of the left block and H_k denotes the k'th order entropy of the left block (see [12]).
3. The *gt* bit vectors for the left and right block. Scanned sequentially and thus can be read streaming from external memory.
4. The number of circular suffixes in the left block smaller than the rightmost suffix of the right block. Used as the start point for the backward search.

5. The gap array G.

The first three are equivalent to those used in [7]. The rank of the rightmost suffix in the right block relative to the suffixes of the left block can be obtained by employing forward search on one or more text blocks. If the left block is a single block which was produced by explicit suffix sorting using the method of Section 4, then the rank can be obtained using classical forward search in the suffix array while using the adjoined LCP array. This takes time $O(n + \log b)$ in the worst case (on average this can be expected to be $O(\log n + \log b)$, see [15]). If the left block was already obtained by merging c blocks together, then the desired rank can be obtained as the sum of the ranks of the suffix relative to all single blocks composing the left block in time $O(c(n + \log b))$. Assuming the blocks are merged together in a balanced binary merge tree the total time used for forward searches is $O(\frac{n}{b} \log \frac{n}{b} n)$ in the worst case and $O(\frac{n}{b} \log \frac{n}{b} \log n)$ on average. If we choose $b \in O(\frac{n}{\log n})$ then this becomes $O(n \log n \log \log n)$. The memory required for the index of the left block in internal memory will be $b_l \log \sigma + o(b_l \log \sigma)$ for a left block size of b_l assuming that the entropy compression is ineffective. This leaves us with the space required for the gap array. In the original algorithm this is a conventional array in internal memory taking space $O(b \log n)$ bits for a left block size of b. As we want to be able to merge blocks with size in $\Omega(n)$ this space requirement is too high. Using Elias γ code (cf. [6]) we can store the gap array for merging a left and right block of length b_l and b_r respectively in $O(b_l + b_r)$ bits of space. γ code however is not suitable for efficient updating as we would need it for computing the gap array. We solve this by producing partial sparse gap arrays and merging these together as needed. These sparse gap arrays are encoded using two γ coded number sequences where one encodes the indices of non-zero values in differential coding and the other encodes the non-zero values. The array G is produced by backward searching the suffixes of the right block in a suitable index of the left block. After each step exactly one element of G is incremented. The sum over the elements of G increases by exactly one for each step. For computing a complete gap array one option is to start by producing sparse arrays consisting of a single element of value 1. Whenever we have produced two partial arrays of sum s we immediately merge them together to a partial array of sum $2s$ in time $O(s)$. This method guarantees that the set of sparse gap arrays present at any one time is bounded in space by $O(b_l + b_r)$ bits. The total merging of partial gap arrays to obtain the final gap array then takes time $O(b_r \log b_r)$. If we accumulate $\frac{b_r}{\log^2 b_r}$ indices for incrementing before writing a partial gap array then we can reduce the merging time to $O(b_r \log \log b_r)$ without increasing the space used by the algorithm.

The gt array for the merged block can be composed by concatenating the gt array for the left block and an array storing the respective information for the right block computed while performing the backward search for filling the gap array. For this purpose we need to know the rank of the leftmost suffix in the left block. This can either be computed using forward search on the suffix arrays of the basic blocks or extracted from a sampled inverse suffix array which can be computed along the way. The sampled inverse suffix arrays of two blocks can just like the BWTs of the two blocks be merged using the gap array. This is also an

operation based on stream accesses, so it can be done in external memory in time $O(b)$.

6 BWT Computation by Balanced Tree Block Merging

Using the building blocks described above we can now describe the complete algorithm for computing the BWT of t by merging basic blocks according to a balanced binary tree.

1. Choose a target block size $b' \in O(\frac{n}{\log n})$ and deduce a block size $b = \lceil \frac{n}{\lceil \frac{n}{b'} \rceil} \rceil$ such that the number of blocks c satisfies $c = \lceil \frac{n}{b} \rceil = \lceil \frac{n}{b'} \rceil$ and n can be split into blocks of size b and $b-1$ only. Split t such that the blocks of size b appear before those of size b'. This step takes constant space and time.
2. Compute which blocks in t propagate repetitions of period at most b and for each block which is followed by a block propagating a repetition whether it is generating this repetition. This takes time $O(n)$ in total and space $O(b \log \sigma) = O(\frac{n \log \sigma}{\log n}) \subseteq O(n)$ bits. The result data can be stored in external memory.
3. Compute a balanced merge tree for the blocks. Start with a root representing all blocks. If a node containing a single block is considered produce a leaf and stop. Otherwise for an inner node representing $k > 1$ blocks produce a left subtree from the $\lceil \frac{k}{2} \rceil$ leftmost blocks and a right subtree from $\lfloor \frac{k}{2} \rfloor$ rightmost blocks in t. The tree has $O(\log n)$ nodes. Each node stores at most two (start and end) block indices taking $O(\log \log n)$ bits and two node pointers also taking space $O(\log \log n)$ bits. So the total tree takes space $O(\log n \log \log n)$ bits. It can be computed in time $O(\log n)$.
4. Sort the blocks and store the resulting BWT, gt and sampled inverse suffix arrays in external memory. Using the suffix and LCP arrays of the basic blocks also compute the start ranks necessary for the backward searches when merging the blocks together. This takes time $O(n \log n \log \log n)$ in the worst case and $O(n)$ on average and space $O(b \log b) = O(\frac{n}{\log n} \log \frac{n}{\log n}) = O(n)$ bits of internal memory.
5. Process the merge tree. Mark all leafs as finished and all inner nodes as unfinished. While there are unfinished nodes choose any unfinished node with only finished children, merge the respective blocks and mark the node as finished. There are $O(\log n)$ leafs and the tree is balanced, so it has $O(\log \log n)$ levels. Each single level can be processed in time $O(n \log \log n)$. So the total run time for the tree merging phase is $O(n \log^2 \log n)$. The maximum internal memory space usage appears when performing the merge operation at the root of the tree. Here we need space $b_l H_k + o(b_l \log \sigma)$ bits where b_l denotes the sum of the length of the blocks in the left subtree which is $O(n)$ and H_k denotes the k'th order entropy of the text comprising those text blocks.

Summing over all steps the run-time of the algorithm is $O(n \log n \log \log n)$ in the worst case and $O(n \log^2 \log n)$ on average. In practice this means we can compute the BWT of a text as long as we are able to hold the text (more precisely the text for the left subtree of the merge tree) in internal memory. If we can hold a fixed fraction of the text in main memory, then we can still compute the BWT of the text in the same run-time by resorting to the original iterative merging scheme

from [7]. We decompose the text into blocks of size b' such that $b' \leq \frac{n \log \sigma}{c \log n}$ where $\frac{1}{c}$ is the fixed fraction of the text we can hold in internal memory and compute a partial BWT for each of these blocks where the suffixes sorted are considered as coming from the whole text, i.e. suffix comparisons are still over $\tilde{t}$ and not limited to a single of the blocks. Then we merge these blocks along a totally skewed merge tree such that the left block always has size about b'. The size of the set of partial sparse gap arrays required at any time remains bounded by $O(n)$ bits. As the number of blocks is fixed, the total asymptotical run-time of the algorithm remains $O(n \log n \log \log n)$ in the worst case and $O(n \log^2 \log n)$ on average.

7 Conclusion

We have presented a new semi external algorithm for computing the Burrows-Wheeler transform. On average our new algorithm is faster then the algorithm of Ferragina et al published in [7]. In comparison with the algorithm by Beller et in [1] our algorithm can be applied for the case when less than 8 bits per symbol of internal memory are available. Due to space constraints proofs, parallelisation of our algorithm and the discussion of an implementation study are postponed to another paper. Sample code implementing parts of the ideas in this paper is available from the author on request.

References

[1] T. Beller, M. Zwerger, S. Gog, and E. Ohlebusch. Space-Efficient Construction of the Burrows-Wheeler Transform. In O. Kurland, M. Lewenstein, and E. Porat, editors, *SPIRE*, volume 8214 of *Lecture Notes in Computer Science*, pages 5–16. Springer, 2013.

[2] T. Bingmann, J. Fischer, and V. Osipov. Inducing Suffix and LCP Arrays in External Memory. In P. Sanders and N. Zeh, editors, *ALENEX*, pages 88–102. SIAM, 2013.

[3] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Digital Systems Research Center. *RR-124*, 1994.

[4] M. Crochemore, R. Grossi, J. Kärkkäinen, and G. M. Landau. A Constant-Space Comparison-Based Algorithm for Computing the Burrows-Wheeler Transform. In J. Fischer and P. Sanders, editors, *CPM*, volume 7922 of *Lecture Notes in Computer Science*, pages 74–82. Springer, 2013.

[5] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12, 2008.

[6] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[7] P. Ferragina, T. Gagie, and G. Manzini. Lightweight Data Indexing and Compression in External Memory. *Algorithmica*, 63(3):707–730, 2012.

[8] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[9] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *FOCS*, pages 251–260. IEEE Computer Society, 2003.

[10] J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[11] U. Manber and G. Myers. Suffix Arrays: a New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[12] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[13] G. Nong, S. Zhang, and W. H. Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *Computers, IEEE Transactions on*, 60(10):1471–1484, 2011.

[14] D. Okanohara and K. Sadakane. A Linear-Time Burrows-Wheeler Transform Using Induced Sorting. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *SPIRE*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009.

[15] W. Szpankowski. On the Height of Digital Trees and Related Problems. *Algorithmica*, 6(1-6):256–277, 1991.

ASSP; the Antibody Secondary Structure Profile search tool

Dimitrios Vlachakis†, Alexandros Armaos†, Kasampalidis I, Arianna Filntisi, Sophia Kossida*

Bioinformatics & Medical Informatics Team, Biomedical Research Foundation
Academy of Athens, Soranou Efessiou 4, Athens 11527, Greece
*skossida@bioacademy.gr

Abstract

Antibodies constitute the first line of defense against harmful invaders. In the post genomics era the sheer size of antibody related NGS information is a major bottleneck in the quest of understanding and tackling complex genetic diseases and immunological disorders. Bioinformatics is becoming hugely involved in the processing of this data with the development of new, more accurate and efficient algorithms. However, one of the major drawbacks of modern bioinformatics is the fact that protein similarity and blast searches are still based on primary amino acid sequence rather than structural data. Primary sequence searches are inadequate, as they fail to provide a realistic fingerprint for the query protein. Antibody function is much more related to its 3D structure and physicochemical profile rather than its primary amino acid sequence. After all, structure is much more conserved than sequence in nature. In this direction, a novel platform has been developed, which is capable of performing a customized hydropathy blast using traditional sequence blast filtering and an integrated fast similarity search algorithm that uses protein secondary structure information. The Antibody Secondary Structure Profile (ASSP) tool will use secondary structural information from the PDB database when available, whereas if the query antibody is not indexed in the RCSB PDB database, it will automatically determine the secondary elements of the given antibody by performing an "on the fly" secondary structure prediction. All query antibodies are then blasted against the RCSB PDB secondary elements database. Hits are scored, ranked and returned to the user via a well-organized and user friendly graphical interface.

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

† These authors have contributed equally to this study.
* Corresponding author: Sophia Kossida, Bioinformatics & Medical Informatics Team, Biomedical Research Foundation, Academy of Athens, Soranou Efessiou 4, Athens 11527, Greece
Tel: + 30 210 6597 199, Fax: +30 210 6597 545 E-mail: skossida@bioacademy.gr

1 Introduction

The hydrophobic effect is the tendency of non-polar substances to avoid contact with water. The hydropathy of an amino acid, which is derived from the physico-chemical properties of its side chains, determines in part the orientation of its side chains in the three-dimensional protein structure. In particular, when a protein folds into a three-dimensional structure, the majority of the hydrophobic side-chains cluster together within the core of the protein. This removal of the hydrophobic side-chains out of contact with water generates sufficient free energy to maintain the folded structure of the protein. The determination of the hydrophobic or hydrophilic inclinations of a given amino acid side-chain has been approached in a number of ways. Measuring the partition coefficient of a given amino acid side-chain between water and a non-interacting, isotropic phase as well as calculating a transfer free energy from that coefficient is one such approach. Another way to calculate the hydropathy of a given side-chain is the tabulation of residue accessibilities from the atomic co-ordinates of twelve globular proteins, taking into consideration that the ensemble average of the actual locations of a side-chain should be a direct evaluation of its hydropathy. An additional approach combined those previously mentioned methods, resulting in the construction of a hydropathy scale, according to which each amino acid has been given a value reflective of its relative hydrophilicity and hydrophobicity [4, 17].

The twenty amino acids found in nature have been categorized in hydropathy classes based on the previously mentioned amino acid hydropathy index developed by Kyte and Doolittle (1982). Specifically, the amino acids with a hydropathy index equal to or more than 1.8 were defined as hydrophobic. The amino acids with a hydropathy index equal to or less than -3.3 were defined as hydrophilic, while the amino acids with a hydropathy index less than 1.8 and more than -3.3 were defined as neutral. Three classes were thus defined: the hydrophobic class (I, V, L, F, C, M, A, W), the neutral class (G, T, S, Y, P, H) and the hydrophilic class (D, N, E, Q, K, R). Tryptophan (W) was included in the hydrophobic class, its hydropathy index varying from -0.9 to 1.9, depending on the study. As a general observation, amino acids with large, nonpolar or largely nonpolar side-chains tend to be hydrophobic, while the least hydrophobic amino acids are the ones that are charged and largely polar, such as asparagine. Statistical analysis, in particular correspondence analysis (COA) and hierarchic classification (CAH), has been conducted according to those three hydropathy classes upon 2474 sequences of antibody variable regions, which were extracted from human productively rearranged sequences [17, 25, 8]. There is a significant variation among the hydrophobicities of the amino acids. Some are strongly hydrophobic, others are strongly hydrophilic, while others include both hydrophobic and hydrophilic parts and are called amphiphilic. For such amphiphilic molecules it is sometimes useful to define a hydrophobic moment, which is analogous to a dipole moment. For a single amino acid, the hydrophobic moment can be defined as a line that points from the Ca atom to the middle of the side-chain, and whose length is proportional to the hydrophobicity of the side-chain. The dipole moment of a protein, or a part of it, is obtained by summing the individual vectors (in magnitude and direction) corresponding to the amino acids

the protein is composed of. For example, an α helix located on the surface of a protein will have one side of the helix exposed to solvent and the other side facing the interior of the protein. The amino acids that comprise the buried side of the α helix will, in general, be much more hydrophobic than those on the solvent-exposed side of the helix. This asymmetry results in the α helix having a large hydrophobic moment directed towards the center of the protein [21].

The three-dimensional structure of a protein is determined by the balance between a number of destabilizing and stabilizing forces, such as conformational entropy, electrostatic interactions, hydrogen bonds, van der Waals interactions and hydrophobic interactions. However, hydropathy is considered to be the most prominent driving force responsible for the folding of proteins. Protein folding occurs in the presence of water, the properties of which are dominated by its inclination to form hydrogen bonds. Polar compounds can share hydrogen bonds with water and, for this reason, are readily soluble. In contrast, when a hydrophobic nonpolar surface is introduced into an aqueous environment, it prevents hydrogen bonding from occurring, which forces the water molecules to adopt alternative arrangements that permit hydrogen bonding to other water molecules. This inflicted restriction on the alignment of the water molecules has an energetic cost and is the physical basis of the hydrophobic effect. It has been calculated that when a protein folds, 81% of the nonpolar side-chains (Ala, Val, Ile, Leu, Met, Phe, Trp, Cys), 70% of the peptide groups, 63% of the polar side chains (Asn, Gln, Ser, Thr, Tyr) and 54% of the charged side chains (Arg, Lys, His, Asp, Glu) are buried in the interior of the protein, out of contact with water [21, 23].

2 Description of ASSP

Hydropathy is a physicochemical property known to be well conserved among antibodies, which can be explained to a large extent by the significant contribution of the hydrophobic residues to the folding of antibodies. Numerous studies on proteins and antibodies have demonstrated that the information necessary to produce a given three dimensional protein structure can be encoded by many different amino acids. In contrast, it has been demonstrated that the periodicity of polar and nonpolar amino acids is the major determinant of secondary structure in self-assembling oligomeric peptides. In fact, the choice between α-helical and β-sheet secondary structure is influenced by the sequence periodicity of polar and nonpolar amino acids. Even though amino acid residues may differ in their intrinsic preferences for one secondary structure versus another, these preferences can be overwhelmed by the drive to form amphiphilic structures capable of burying hydrophobic surface area. It can be observed that structural similarity among antibodies is reflected on the distribution of hydropathicity along their amino acid sequences, since the hydrophobicity patterns of residues match the periodicity of secondary structures.

Homologous antibodies and proteins within a antibody/protein family as well as proteins with related structures appear to have similarities in their hydropathy distributions, even when sequence similarities could not be detected [14, 2, 32, 24].

Since the hydropathy distribution along the antibody sequence has been recog-

nized as a feature useful for the characterization of protein structure in the form of hydropathy profiles, a number of methods based on hydropathy have been developed in order to explain the folding and the structural features of antibodies. The realization that protein sequence contains hydropathy patterns led to the development of reduced amino acid alphabets based on hydropathy for the prediction of secondary structure.

Hydropathy has also been utilized for the detection of analogous and distantly related proteins and the classification of new protein sequence data. The use of hydropathy profile analysis has made possible the identification of more distantly related antibodies than could be done by sequence comparison. In addition, antibody sequence databases have been analyzed using hydropathy patterns with the goal of identifying new members of functional classes [17, 24, 7, 26, 5, 33, 19, 20, 6].

Many homologous proteins share very low primary sequence identity and similarity scores amongst them. The most characteristic example of such proteins is viral enzymes. Helicases, proteases and polymerases are just few of the many examples of protein families that are structurally conserved but may share no more than 10% sequence identity with each other. Consequently, looking for homologues of a certain viral enzyme or even for a suitable template structure during homology modelling using traditional amino acid based blast searches is futile. However, careful structural analysis of any of the above enzymes reveals that those proteins are actually highly conserved in their secondary, tertiary and quaternary structures. Moreover, all evolutionary protein relationships as well as protein function analysis should also be based on searches that utilize structural information. Overall, it has been established that homologous proteins are much more conserved in their structures than in their amino acid primary sequences [4, 17].

Herein, the ASSP tool takes advantage of the full RCSB PDB secondary structure database in order to perform blast-like searches in the secondary element level amongst proteins. To date, even though long studies have been conducted in many fields of structural biology and modern bioinformatics this problem not been yet satisfactorily addressed [15, 1, 22]. This is a fact that necessitates the need to the development of such a platform.

ASSPs main-window is a menu-driven interface as well as a tab step-by-step layout. Initially the user has an option regarding the query input type that will be used. ASSP will handle both primary amino acid sequence as well as DSSP-formatted secondary element protein sequence [13]. The user can follow two main routes for the ASSP run: Firstly, the user may input either raw primary amino acid sequence for a conventional blast search or opt for a quick secondary structure prediction of the amino acid sequence using the built-in STRAP module [9]. STRAP will perform a very fast, over the internet secondary structure prediction, which will eventually return the predicted secondary element composition of the query protein [9]. Eventually a DSSP compatible secondary structure determination code will have been obtained for the actual secondary structure similarity search [13]. Secondly, an existing DSSP compatible secondary structure determination code may be used as input from the user straightaway, which will then be automatically blasted against the secondary structural index database of the RCSB Protein Databank. If a secondary element antibody sequence description is used,

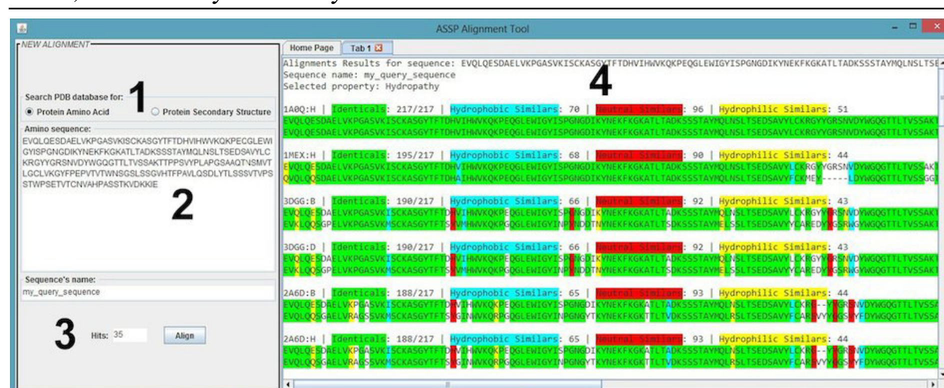

Figure 1: The graphical user interface for the ASSP platform provides an intuitive pipeline for running similarity searches that eliminate the chance for human error, while at the same time providing a graphical, easy to comprehent results output window. The graphical user interface of the ASSP platform, consists of the following main windows: 1. The blast type selection option. 2. The input query window. 3. Sequence name and number of hits selections 4. The result-output area. The alignmed regions are color-coded in accordance to the IMGT coloring scheme for hydropathy. The identical residues are colored green, the hydrophobic residues are colored blue, the neutral residues are colored red and the hydrophilic residues are colored yellow.

then ASSP will move swiftly to the actual similarity blast search.

The screening process of ASSP is broken down in two steps. First a conventional primary sequence-based blast is performed with a threshold value of 30% identity, when temporary file is created with all sequences sharing more than 30% identity with the query antibody sequence (using the blosum62 substitution matrix). Then the custom made hydropathy substitution matrix is engaged and the previously filtered entries are ranked according to their identity/similarity scores based on their hydropathy profiles.

The hydropathy matrix has been created using the antibody hydropathy index from IMGT [18]. Results in the form of alignments and similarity percentages are calculated, scored, ranked and returned to the user through the same graphical interface that has been specifically designed to simplify the task for the user and to eliminate the possibility of a user-inflicted error (see Figure 1). The output window is color-coded in accordance to the IMGT coloring scheme for hydropathy. The identical residues are colored green, the hydrophobic residues $(4,5$ to $-0,9)$ are colored blue, the neutral residues $(-0,4$ to $-3,2)$ are colored red and the hydrophilic residues $(-3,5$ to $-4,5)$ are colored yellow. The results are outputted and saved in easy to manipulate text-based text/ascii files for future analysis.

The secondary description code that ASSP has adopted is the same with the one DSSP has been using for many years now [13]. This was intentionally done for ease of use and backward compatibility issues. More specifically an eight-letter description code is used. Using just eight letters, instead of the traditional twenty

amino acid letters, makes similarity searches ever more efficient and faster than ever before. The eight letter secondary element code comprises of the following letters: H for α helix conformation, B for residues in isolated beta-bridge, E for extended strands that participate in beta ladders, G for 3/10 helices, I for pi helices, T for hydrogen bonded turns and S for bends. C is used for the blank space in the DSSP secondary structure, which represents a loop or an irregular element. Other major suites, such as the PDBFINDER suite, also adopt this convention with unstructured protein regions [11]. Same WHATIF uses C, as many times leaving a blank may be confusing, misleading and inconvenient [31, 10, 12]. A batch execultion mode has also been prepared for the ASSP suite. A simple text file is required with as many sequences as the user wishes, each one stored in a different line. The ASSP algorithm will then automatically read that file line by line and execute the antibody similarity search for as many times as the lines of the input batch file. This comes quite handy for those who wish to perform secondary structure similarity searches on large databases of protein or peptide sequences [28, 30, 29, 3, 27, 16]. Finally, an extensive manual and use-case based examples for the use of ASSP, will pop-up through the Help button, using the operating systems HTML browser application.

3 Conclusions

In conclusion, the ASSP toolkit provides a novel, quick and reliable tool for in silico antibody similarity searches in one pipelined platform under a user friendly graphical user interface. We therefore, propose that our structural similarities application described here would yield results of great interest to many antibody-related scientific disciplines. The ASSP platform is distributed as freeware under a GNU license.

4 Availability

Availability: ASSP can be freely downloaded via our dedicated server system at `http://www.bioacademy.gr/bioinformatics/assp/index.html`
ASSP is an open source, cross platform application available freely to all users under a GNU license basis. The full package, including installation scripts, figures, a full description, a detailed manual, complete tutorials as hands-on use cases, software prerequisites and various examples can be downloaded at: `http://www.bioacademy.gr/bioinformatics/assp/`. Prior to download; check the provided information on the website about software prerequisites. Please email comments and bug reports at *dvlachakis@bioacademy.gr*.

Acknowledgements

This work was partially supported by:
1. The BIOEXPLORE research project. BIOEXPLORE research project falls under the Operational Program “Education and Lifelong Learning” and is co-financed by the European Social Fund (ESF) and National Resources.
2. European Union (European Social Fund - ESF) and Greek national funds

through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

The authors would like to thank Prof. Marie-Paule LeFranc and the IMGT institute for their constructive comments and support.

References

[1] C. Berbalk, C. S. Schwaiger, and P. Lackner. Accuracy analysis of multiple structure alignments. *Protein Sci.*, 18(10):2027–2035, Oct 2009.

[2] J. U. Bowie, J. F. Reidhaar-Olson, W. A. Lim, and R. T. Sauer. Deciphering the message in protein sequences: tolerance to amino acid substitutions. *Science (New York, N.Y.)*, 247(4948):1306–1310, Mar. 1990.

[3] C. S. Carvalho, D. Vlachakis, G. Tsiliki, V. Megalooikonomou, and S. Kossida. Protein signatures using electrostatic molecular surfaces in harmonic space. *PeerJ*, 1:e185, 2013.

[4] C. Chothia. The nature of the accessible and buried surfaces in proteins. *Journal of Molecular Biology*, 105(1):1–12, July 1976.

[5] S. Y. Chung and S. Subbiah. A structural explanation for the twilight zone of protein sequence homology. *Structure*, 4(10):1123–1127, Oct. 1996.

[6] J. D. Clements and R. E. Martin. Identification of novel membrane proteins by searching for patterns in hydropathy profiles. *Eur. J. Biochem.*, 269(8):2101–2107, Apr 2002.

[7] D. Eisenberg, R. M. Weiss, and T. C. Terwilliger. The hydrophobic moment detects periodicity in protein hydrophobicity. *Proc Natl Acad Sci U S A*, 81(1):140–144, Jan. 1984.

[8] D. M. Engelman, T. A. Steitz, and A. Goldman. Identifying nonpolar transbilayer helices in amino acid sequences of membrane proteins. *Annual review of biophysics and biophysical chemistry*, 15:321–353, 1986.

[9] C. Gille. STRAP: Structure based sequences alignment program. `http://www.bioinformatics.org/strap/index2.html`.

[10] M. L. Hekkelman, T. A. Te Beek, S. R. Pettifer, D. Thorne, T. K. Attwood, and G. Vriend. WIWS: a protein structure bioinformatics Web service collection. *Nucleic Acids Res.*, 38(Web Server issue):W719–723, Jul 2010.

[11] R. Hooft, C. Sander, M. Scharf, and G. Vriend. The pdbfinder database: a summary of pdb, dssp and hssp information with added value. *Computer applications in the biosciences : CABIOS*, 12(6):525–529, 1996.

[12] R. W. W. Hooft, G. Vriend, C. Sander, and E. E. Abola. Errors in protein structures. *Nature*, 381(6580):272, May 1996.

[13] W. Kabsch and C. Sander. Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers*, 22(12):2577–2637, Dec 1983.

[14] S. Kamtekar, J. M. Schiffer, H. Xiong, J. M. Babik, and M. H. Hecht. Protein design by binary patterning of polar and nonpolar amino acids. *Science*, 262(5140):1680–1685, 1993.

[15] R. Kolodny, P. Koehl, and M. Levitt. Comprehensive evaluation of protein structure alignment methods: scoring by geometric measures. *J. Mol. Biol.*, 346(4):1173–1188, Mar 2005.

[16] E. Krissinel. Enhanced fold recognition using efficient short fragment clustering. *Journal of Molecular Biochemistry*, 1(2), 2012.

[17] J. Kyte and R. F. Doolittle. A simple method for displaying the hydropathic character of a protein. *Journal of Molecular Biology*, 157(1):105–132, May 1982.

[18] M. P. Lefranc, V. Giudicelli, C. Ginestoux, J. Bodmer, W. Muller, R. Bontrop, M. Lemaitre, A. Malik, V. Barbie, and D. Chaume. IMGT, the international ImMunoGeneTics database. *Nucleic Acids Res.*, 27(1):209–212, Jan 1999.

[19] J. S. Lolkema and D. J. Slotboom. Estimation of structural similarity of membrane proteins by hydropathy profile alignment. *Mol. Membr. Biol.*, 15(1):33–42, 1998.

[20] J. S. Lolkema and D. J. Slotboom. Hydropathy profile alignment: a tool to search for structural homologues of membrane proteins. *FEMS Microbiol. Rev.*, 22(4):305–322, Oct 1998.

[21] B. W. Matthews. *Hydrophobic Interactions in Proteins.* John Wiley & Sons, Ltd, 2001.

[22] G. Mayr, F. S. Domingues, and P. Lackner. Comparative analysis of protein structure alignments. *BMC Struct. Biol.*, 7:50, 2007.

[23] C. N. Pace, B. A. Shirley, M. McNutt, and K. Gajiwala. Forces contributing to the conformational stability of proteins. *FASEB journal*, 10(1):75–83, Jan. 1996.

[24] J. Pánek, I. Eidhammer, and R. Aasland. A new method for identification of protein (sub)families in a set of proteins based on hydropathy distribution in proteins. *Proteins*, 58(4):923–934, 03 2005.

[25] C. Pommié, S. Levadoux, R. Sabatier, G. Lefranc, and M.-P. Lefranc. Imgt standardized criteria for statistical analysis of immunoglobulin v-region amino acid properties. *Journal of Molecular Recognition*, 17(1):17–32, 2004.

[26] R. B. Russell, M. A. Saqi, R. A. Sayle, P. A. Bates, and M. J. Sternberg. Recognition of analogous and homologous protein folds: analysis of sequence and structure conservation. *J Mol Biol*, 269(3):423–439, June 1997.

[27] D. Vlachakis, D. Tsagkrasoulis, V. Megalooikonomou, and S. Kossida. Introducing drugster: a comprehensive and fully integrated drug design, lead and structure optimization toolkit. *Bioinformatics*, 29(1):126–128, 2013.

[28] D. Vlachakis, D. Tsagkrasoulis, G. Tsiliki, and S. Kossida. The future of structural bioinformatics in the post-genomic era. *EMBnet. journal*, 18(1):pp–3, 2012.

[29] D. Vlachakis, S. C. Tsaniras, C. Feidakis, and S. Kossida. An in silico 3D study of the biglycan core protein, using homology modelling techniques. *Journal of Molecular Biochemistry*, 2(2), 2013.

[30] D. Vlachakis, G. Tsiliki, D. Tsagkrasoulis, C. S. Carvalho, V. Megalooikonomou, and S. Kossida. Speeding up the drug discovery process: structural similarity searches using molecular surfaces. *EMBnet.journal*, 18(1), 2012.

[31] G. Vriend. WHAT IF: a molecular modeling and drug design program. *J Mol Graph*, 8(1):52–56, Mar 1990.

[32] H. Xiong, B. L. Buckwalter, H. M. Shieh, and M. H. Hecht. Periodicity of polar and nonpolar amino acids is the major determinant of secondary structure in self-assembling oligomeric peptides. *Proceedings of the National Academy of Sciences*, 92(14):6349–6353, 1995.

[33] X. J. Yu and D. H. Walker. Sequence and characterization of an Ehrlichia chaffeensis gene encoding 314 amino acids highly homologous to the NAD A enzyme. *FEMS Microbiol. Lett.*, 154(1):53–58, Sep 1997.