DisLex: a Transformation for Discontiguous Suffix Array Construction

Paul B. Horton[1*], Szymon M. Kiełbasa[2], and Martin C. Frith[1]

[1]AIST, Computational Biology Research Center,
2-42 Aomi, Koutou-ku, Tokyo
[2]Max Planck Institute for Molecular Genetics,
Ihnestraße 73
D-14195 Berlin, Germany
*Corresponding Author
horton-p@aist.go.jp, szymon.kielbasa@molgen.mpg.de, martin@cbrc.jp
http://last.cbrc.jp/

Proceedings of *the workshop on Knowledge, Language, and Learning in Bioinformatics, KLLBI, pp. 1-11*, Hanoi, Vietnam, December 2008.

Abstract. Algorithms for local sequence similarity search have been extensively studied. However, advances in computer hardware and the astounding throughput of next generation DNA sequencers continues to demand improved computational methods. Theory and experience has demonstrated the utility of a search strategy in which *discontiguous seeds* (i.e. short, exact matches, skipping certain fixed positions when comparing) are first identified and only those sequence regions are further explored. In theory, *variable length* seeds are desirable when repetitive elements are present, but fixed length seeds are easier to manage. Thus current popular programs use fixed length seeds, and typically ignore repetitive elements with "repeat masking". Fortunately, recent increases in memory size and advances in suffix array related algorithms have made variable length seeds technically feasible. The one element which has been missing is a simple technique to enable the use of *discontiguous* variable length seeds.

We describe a transformation, *DisLex*, for use with *discontiguous suffix arrays*. Discontiguous suffix arrays hold the suffices of a text in a sorted order which skips predetermined don't care positions in a repeated pattern, corresponding to a variable length discontiguous seed. DisLex uses the technique of lexical naming to construct a derived *lexText*, which mirrors the original text. The mapping is done in such a way that the ordinary suffix array constructed on lexText is the same as the *discontiguous* suffix array constructed on the original text.

Using whole human chromosomes, we show that discontiguous suffix arrays can be practically constructed with existing standard suffix array construction programs and that the overhead incurred by DisLex itself is negligible – less than 30 seconds for the biggest human chromosomes.

Key words: Genome Alignment, Suffix Arrays, Spaced seeds

1 Background

Nearly 20 years ago, the development of BLAST[3] revolutionized genomic sequence similarity search and rapidly became an ubiquitous tool. A decade later, PatternHunter[11] formally introduced "spaced seeds", which improve on the original BLAST technique of hashing fixed length substrings by hashing discontiguous subsequences based on a given pattern of don't care positions.

Spaced seeds have both practical and theoretical advantages for detecting sequence similarities. The practical advantage is that protein-coding DNA tends to exhibit substitutions at every third position, since such substitutions often preserve the encoded amino acids (fig. 1). The theoretical advantage is that, even with uniform random substitutions, spaced seeds are more likely to find at least one hit. This fact is related to the statistics of word counts[5]. For example, the expected count in a random sequence of the words "`gggg`" and "`gagc`" is the same, but the variance of "`gggg`" is larger (because it may occur in overlapping clumps), and the probability of getting "`gggg`" zero times is larger. The benefits of spaced seeds have been demonstrated in practical alignment tools other than PatternHunter as well[9].

These invaluable tools enable a variety of useful searches against genomic data. However the need for new sequence comparison methods for huge datasets is, perhaps surprisingly, as great as ever. This is because DNA sequencing technology continues to advance beyond what existing analysis tools can easily handle. Recent sequencing technologies such as 454, Solexa-Illumina, and SOLiD have enabled new approaches to probing chromatin structure, transcriptomes, and genetic polymorphisms, by massive sequencing of DNA tags. Genomes and metagenomes are being sequenced at an ever increasing rate. The fact is, it is not easy to compare such datasets using BLAST or any other existing tool.

The recent explosion in the availability of genomic sequence data, coupled with nearly exponential growth in the size of computer memory; has created both the need and the opportunity for new algorithms and data structures to support approximate matching of genomic sequences. Suffix trees are powerful data structures for string search with many applications in computational biology[7]. Suffix arrays[12] are more compact data structures which can replace suffix trees in most applications.

Suffix arrays are a promising basis for improved sequence comparison methods. Although in pure form simply an array of (the starting indices of) the text suffices in sorted order; with the aid of some auxiliary data, the suffix array is a data structure that allows many types of sequence queries to be answered in a theoretically efficient manner[7, 1]. Several studies have applied standard suffix arrays to genomic data[14, 2, 15].

The efficient *variable length* exact matching afforded by suffix arrays is especially promising in handling repetitive elements. Repetitive elements pose a problem for fixed length seeds, because long matches to repetitive sequences occur frequently and either overwhelm the computation or obscure meaningful shorter matches (if the seed length is increased to reduce the number of hits).

A common approach to repetitive regions has been to simply ignore them by so called "repeat masking". Unfortunately this approach may sacrifice crucial information. Repetitive sequences comprise a large fraction of the genomes of higher species (more than 50% for human) and in recent years have been found to be transcribed in many cases. Indeed, in at least one case, transcripts of repetitive sequences have been implicated in important regulatory roles[6].

Standard suffix arrays can only find contiguous matches, limiting their sensitivity for remote similarity detection. Discontiguous suffix arrays would be expected to have improved sensitivity, just as has been demonstrated for spaced seeds.

This paper presents DisLex, a transformation which allows existing suffix array algorithms and software to be used to construct discontiguous suffix arrays.

2 Methods

2.1 Outline of DisLex

DisLex consists of two linear time (in the size of the text) transformations wrapped around a standard suffix array construction step. Since suffix arrays can be constructed in linear time[10, 8], DisLex can be used to construct a discontiguous suffix array in linear time. The dislex procedure is outlined in table 1.

1. transformation of the text to a LexText*
2. construction of a standard suffix array on the LexText
3. reverse transformation of the indices of the suffix array constructed in the previous step, so that the indices refer to positions in the original text*

Table 1. The steps for constructing a discontiguous suffix array with DisLex are shown. The steps marked with an asterisk are unique to DisLex.

2.2 The Skew Algorithm

The key idea of DisLex comes from a technique involving lexical naming and suffix grouping by index modulus used in the skew algorithm[8] for construction of ordinary suffix arrays. We begin by explaining the parts of the skew algorithm which are relevant to DisLex.

2.3 The Skew Algorithm: Sorting Suffices While Skipping Every Third

The skew algorithm[8] is a linear time algorithm for ordinary suffix array construction. The technique of lexical naming used by the skew algorithm forms the

basis of DisLex. In the skew algorithm, lexical naming allows the suffices with index i, such that $i \mod 3 \neq 0$, to be sorted using an ordinary suffix array construction procedure (*i.e.* a suffix array construction procedure which does not skip every third suffix). Note that this cannot be done by simply sorting the suffices of the text with every third character deleted.

2.4 Lexical Naming in the Skew Algorithm

The lexical naming used in a subroutine of the skew algorithm, defines a one-to-one mapping between each distinct length three substring (*triple*) and integers such that the lexical order of the integers is consistent with that of the triples they represent. Adopting their terminology, the *lexical name* of a triple is the integer it corresponds to in the mapping. The particular mapping used is the rank (counting from one) of each triple, when sorted in alphabetical order. Finally, they pad each string with three special terminal characters which map to the integer zero.

2.5 Suffix Grouping by Modulus in the Skew Algorithm

As an intermediate step, the skew algorithm computes an array which we call the *lexText*; an array of lexical names which consists of the concatenation of two similar blocks, both of which mirror the original text. The first block represents the suffices with indices i, such that $i \mod 3 = 1$, (hereafter denoted as 1_{mod3} suffices while the second block represents the 2_{mod3} suffices. An example text and its corresponding lexText are shown in figure 2. For suffices in the second block, the corresponding triples spell out the 2_{mod3} suffices of the original text, while for suffices in the first block, the triples spell out the 1_{mod3} suffices followed by terminal padding and then the original text minus the first two characters. Fortunately the terminal padding insulates the effect of the extra characters when sorting. It can easily be confirmed that the sorted order of the 1_{mod3} and 2_{mod3} suffices of the original text is equivalent to the sorted order of their corresponding positions in the lexText.

2.6 The DisLex transformation

Notation Some notation used in this manuscript is shown in table 2.

The lexText computed by the skew algorithm skips the 0_{mod3} suffices, which is the whole point of its use – to recursively reduce the size of the "text" by a factor of two thirds until a base case is reached.

Nevertheless, we observe that by prepending a third block, representing the 0_{mod3} suffices to the lexText computed by the skew algorithm, the lexText could be used to indirectly sort *all* of the original text.

For ordinary suffix array construction this indirection is meaningless, but it does allow some flexibility in the way suffices are sorted. In particular, by computing lexical names with respect to a don't care pattern *mask*, the lexText can be used to transform the discontiguous suffix array construction problem to ordinary suffix array construction. An example is shown in figure 3.

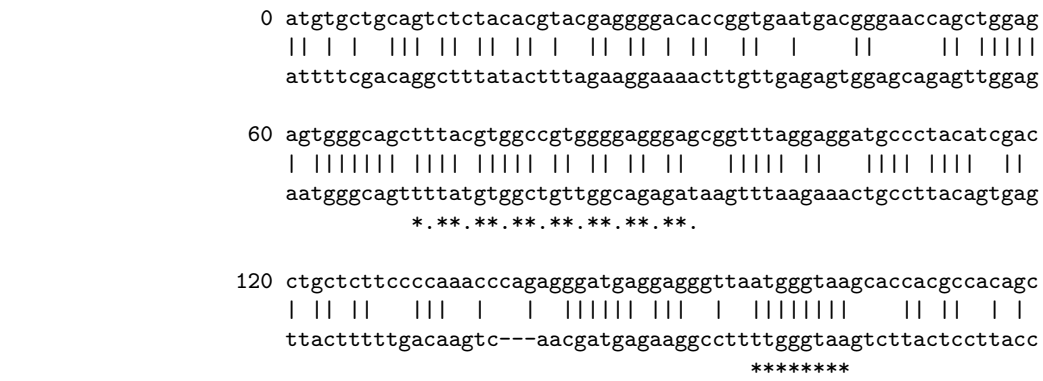

Fig. 1. An alignment of the human and fugu DCDC2 genes is shown. The longest exact match (length 8) and match under the codon mask "101" (length 23, including 15 matching nucleotides) are indicated with asterisks.

notation	quantity	numerical relation
l_m	mask	
l_s	text	
l_y	text plus mask	$l_s + l_m$
l_p	terminal padding	$2 * l_m - (l_y \,\%\, l_m) - 1$
l_z	text with terminal padding	$l_s + l_p$
l_x	lexText	l_z - l_m + 1
l_b	lexText modulo block	l_x / l_m

Table 2. Notation for the length of various strings or text regions is shown. "% l_m" denotes the remainder after dividing by l_m.

Terminal Padding A sufficient number of terminal padding characters needs to be appended to the text to ensure that each modulo block is insulated from the following blocks by at least one terminal character. To ease the reverse tranformation described in the following section, it is convenient to append extra terminal characters as necessary so that each modulo block has the same length. The exact formula for the number of terminal characters is given in third column of table 2.

2.7 Reverse Transformation

The conventional suffix array of the lexText contains the desired information regarding the order of the text suffices sorted as discontiguous subsequences with the given mask. The remaining step is to reverse transform the indices of the lexText suffix array to indices into the original text. This can be computed

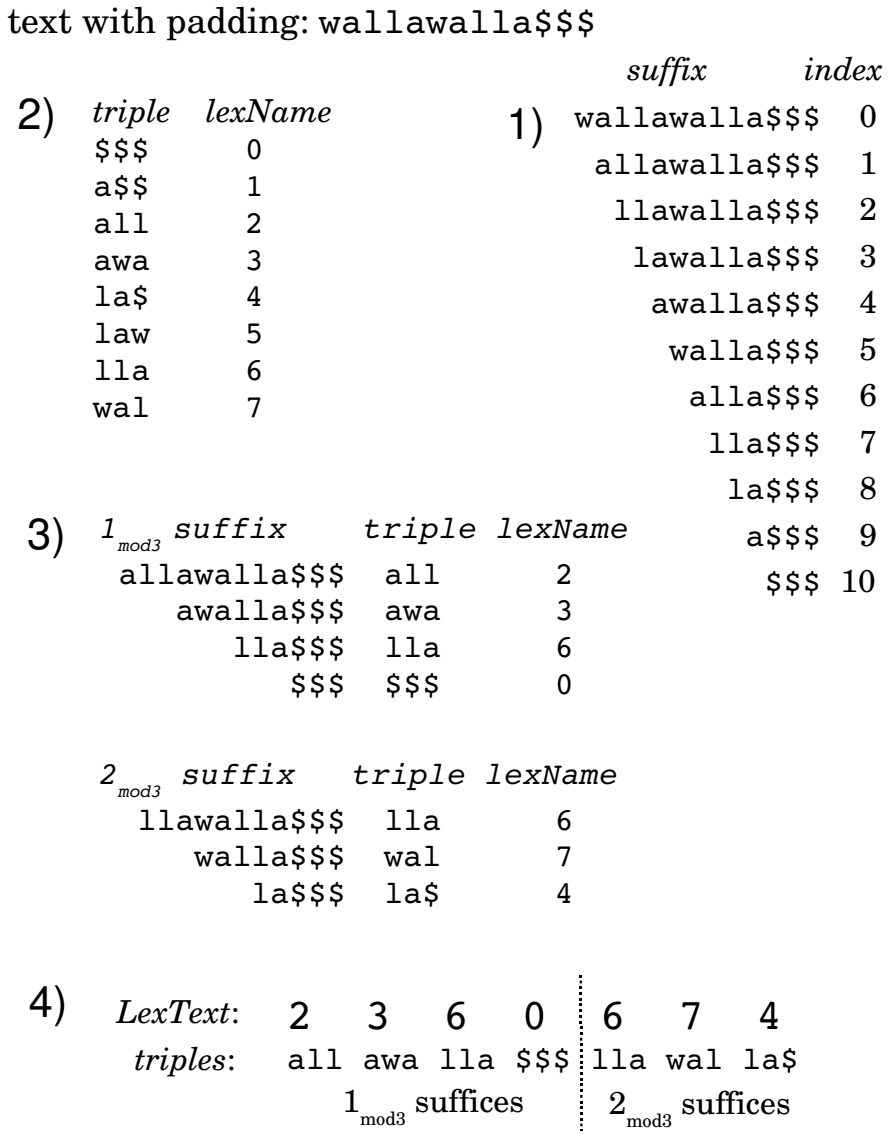

Fig. 2. The definition of the lexText used by the skew algorithm is illustrated with the text "wallawalla". 1) lists the suffices and their indices, 2) lists the lexical names of each triple, 3) lists the lexical names of the suffices equal (mod 3) to 1 and 2 respectively, 4) depicts how the lexText mirrors the original text. "LexName" denotes "lexical name".

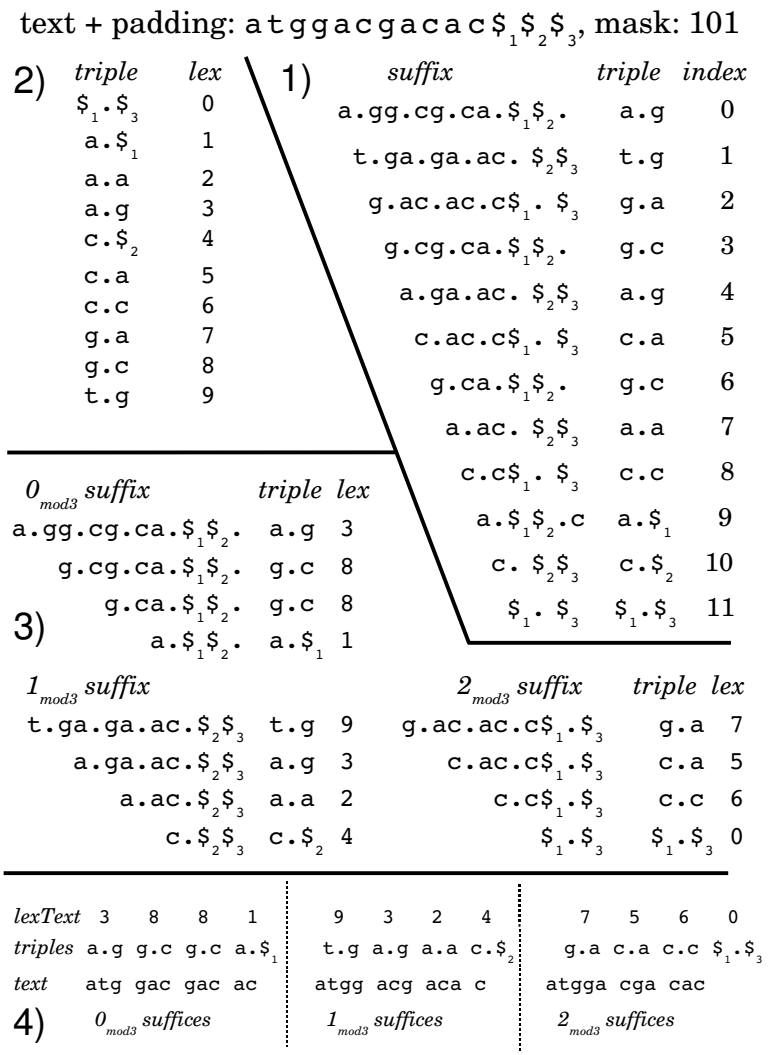

Fig. 3. The transformation computed by DisLex for the text "atggacgacac" under mask "101" is shown. Parts: 1) lists the suffices, their triples and indices, 2) lists the lexical names of each triple, 3) lists the lexical names of the suffices equal (mod 3) to 0, 1 and 2 respectively, 4) depicts how the lexText mirrors the original text. "Lex" denotes "lexical name". The terminal characters $\$_1, \$_2, \ldots$ have increasing lexical order, but less than that of any non-terminal characters.

as follows:

$$i_s = b_{num} + l_m * (i_x - l_b * b_{num}),$$
$$b_{num} = \lfloor i_x / l_b \rfloor$$

2.8 Lexical Naming Method and Computational Complexity

Lexical names of substrings of fixed length l_m can be computed based on sorted rank or arithmetically.

Rank Lexical Naming The skew algorithm assigns lexical names to substrings appearing in the text, based on the rank of their sorted order amongst the distinct length l_m substrings appearing in the text. This can be done with l_m passes of a single character radix sort in $O(l_s * l_m)$ time and $O(l_s)$ space. The textbook version of radix sort requires two working copies of the array (LexText in our case) to be sorted. Fortunately "in place" versions of radix sort which do not require extra memory are also possible[13].

Arithmetic Lexical Naming For small alphabets and relatively short masks, each possible length l_m substring can be mapped to a lexical name arithmetically, by treating the substring as a non-negative integer represented in base σ. Let b_i represent the ith character (skipping don't care positions) of a substring $b = b_1 b_2 ... b_{l_m}$. One may define the arithmetic lexical name of b as:

$$b_1 \sigma^{l_m - 1} + b_2 \sigma^{l_m - 2} + \cdots + b_{l_m}$$

When the lexical names fit in a single computer word (*e.g.* 32 bits), this is a convenient method. For *contiguous* substrings, given the lexical name of one substring, the succeeding substring can be computed in constant time by subtracting the first term, multiplying by σ, and then adding in one new term. Unfortunately this technique does not apply to discontiguous substrings and the time complexity is the same as for rank lexical naming ($O(l_s * l_m)$, although the constant factor may somewhat faster). No extra memory is required. One potential disadvantage of this approach is that the alphabet size of the resulting LexText is larger than necessary, which may adversely affect the performance of some suffix array construction algorithms.

3 Results and Conclusions

3.1 Implementation

We (P.H.) implemented DisLex in `C++` for arithmetic lexical naming up to a 32 bit lexical name and also a custom function for the important special case of the codon mask. To test the implementation we also implemented two other (non-linear time) algorithms coded by different programmers. One (coded by

S.K.) uses the `C++` standard library sort routine, and another "LAST" (coded by M.F.) uses a version of radix sort. The source code for LAST is available at `last.cbrc.jp`. The source code for DisLex will be made available upon request under an open source license.

3.2 Running time

We measured the execution time of DisLex with the codon mask `101` and the "PatternHunter" mask `111010010100110111`, proposed for use with PatternHunter[11]. The running time of DisLex and a slightly modified version of the skew algorithm program "drittel"[8] for constructing the suffix array of the lexText are shown in table 3. As expected the customized routine for the codon mask is faster than the general routine used for the PatternHunter mask. In fact, for the codon mask, the reverse transformation takes slightly longer than DisLex because it involves computing the modulus, which is a relatively slow operation.

chromosome #	DisLex transformation	lexText suffix array construction	reverse transformation
Codon Mask: `101`			
Human 1	0, 0	580, 579	3, 4
Human 2	0, 1	617, 613	4, 3
Human 3	0, 0	486, 485	3, 3
Pattern Hunter Mask: `111010010100110111`			
Human 1	21, 21	902, 899	4, 3
Human 2	21, 21	951, 952	4, 4
Human 3	17, 17	758, 756	3, 3

Table 3. Running time of two trials on an 2.66GHz Intel Core2 Quad workstation with 8 GBytes of memory. Time in seconds reported by system library "`time`" function, "0" means less than one second.

3.3 Terminal Padding

Regarding terminal padding, in figure 3 we have depicted each terminal padding character as a distinct character $\$_1, \$_2, \ldots$. Since the performance of some suffix array processing algorithms is sensitive to alphabet size, we note here that the distinct terminal padding characters are mostly an expositional device. Multiple terminal characters (or some other special convention) are necessary to give the suffices an easily understood unique order (namely the suffix with the smallest text index comes first). Consider the text "`aa$$$`" under the codon mask "101"; two suffices both spell out the suffix "`a.$`". Introducing multiple terminal characters breaks this tie. However, for most applications it is not necessary

to resolve the relative order of such suffices, and therefore our implementation of DisLex does not distinguish between different terminal characters.

Conclusion We have presented the first linear time algorithm to construct discontiguous suffix arrays. As demonstrated by the running times shown here, DisLex can be implemented efficiently enough to require negligible running time, compared to the time required to construct a conventional suffix array on the text. Combining DisLex with the skew algorithm gives the entire process a linear computation time in the length of the text, but any suffix construction program can be used to sort the lexText suffices. Indeed the simplicity of DisLex suggests it may be possible to combine it with advanced techniques such as compressed suffix arrays[4].

DisLex represents an important step forward in the quest to fully utilize genomic data.

4 Authors contributions

P.H. conceived and implemented DisLex, and wrote most of the manuscript. S.K. suggested studying the skew algorithm, which inspired DisLex and helped test the implementation. M.F. suggested discontiguous suffix arrays, helped polish the algorithm and wrote some of the manuscript.

5 Acknowledgement

P.H. was supported by a Japanese Ministry of Education, Culture, Sport, Science and Technology, Grant-in-Aid for Scientific Research (B).

References

1. MI Abouelhoda, S Kurtz, and E Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, 2004.
2. Mohamed I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, pages 449–463. Springer-Verlag, 2002.
3. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *JMB*, 215:403–410, 1990.
4. Kuo Ping Chiu, Chee-Hong Wong, Qiongyu Chen, Pramila Ariyaratne, Hong Sain Ooi, Chia-Lin Wei, Wing-Kin Ken Sung, and Yijun Ruan. PET-Tool: a software suite for comprehensive processing and managing of Paired-End diTag (PET) sequence data. *BMC Bioinformatics*, 7:390, 2006.
5. W.J. Ewens and G.R. Grant, editors. *Statistical Methods in Bioinformatics*. Springer, second edition, 2005.
6. L.S. Gunawardane, K. Saito, K.M. Nishida, K. Miyoshi, Y. Kawamura, T. Nagami, H. Siomi, and M.C. Siomi. A slicer-mediated mechanism for repeat-associated siRNA 5' end formation in Drosophila. *Science*, 315(5818):1587–1590, 2007.

7. Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge Press, New York, NY, 1997.
8. Juha Kaŕkkaïnen and Peter Sanders. Simple linear work suffix array construction. *Lecture Notes in Computer Science*, 2719:943–955, 2003.
9. W. James Kent and Alan M. Zahler. Conservation, regulation, synteny, and introns in a large-scale C. briggsae-C. elegans genomic alignment. *Genome Research*, 10:1115–1125, 2000.
10. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In R. Baeza-Yates, E. Chavez, and M. Crochemore, editors, *CPM'03*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2003.
11. Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
12. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):262–272, Oct 1993.
13. Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.
14. Kunihiko Sadakane and Tetsuo Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.
15. Tomoyuki Yamada and Shinichi Morishita. Computing highly specific and noise-tolerant oligomers efficiently. *Journal of Bioinformatics and Computational Biology*, 2:21–46, 2004.