Determining gapped palindrome density in RNA using suffix arrays

Sjoerd J. Henstra

Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands

Abstract

DNA and RNA strings contain certain structured sequences, called microRNA, which are of interest. This paper describes our attempt to develop an algorithm that can predict where in a string such sequences may be found. This is done by searching for gapped palindromes, called inverted repeats in genetics, which indicate that DNA can fold in on itself. The algorithm that is described makes use of suffix arrays and longest common prefix arrays. From the current algorithm, which is still in development, it is not fully clear how to predict the location of microRNA. The algorithm does quickly find all potentially maximal gapped palindromes in a given string.

1 Introduction

A palindrome is a word that reads the same forward and backword. For example, if we reverse the word RACECAR it remains exactly the same.

The notion of a palindrome can also be applied to DNA and RNA. DNA and RNA sequences are made up of the nucleobases cytosine, guanine, adenine, thymine and uracil. We will refer to these bases by their first letter (C, G, A, T and U). DNA is made up of bases C, G, A and T. RNA is transcribed from DNA and in that process U replaces T. The difference between DNA and RNA is not important for the methods described in this paper, but results apply to RNA rather than DNA.

DNA consists of two strands of nucleotides which are connected by hydrogen bonds between bases in either strand. A pair of bases involved in such a bond is called a base pair. These base pairs always consist of a C and a G, or an A and a T. RNA, on the other hand, is single stranded. Instead of forming base pairs with a second strand, RNA forms pairs within itself, resulting in structures which are potentially interesting. Much effort has gone into finding ways to predict the actual structure of RNA, but the goal in this paper is merely to identify regions in a large RNA sequence which are likely to be highly structured.

We will describe a method which efficiently computes a measure of gapped palindrome density on a sliding window which moves across a long string. We do so using suffix arrays and longest common prefix arrays as described by Manders and Myer [6], which have been used before to find maximal gapped palindromes [5]. If palindrome density proves to be a useful measure to identify known interesting RNA substrings this method could be used to quickly identify other potentially interesting RNA substrings which merit further investigation.

The remainder of this paper is organized as follows. In Section 2 we provide some basic definitions regarding palindromes in RNA strings. Section 3 describes suffix arrays and Section 4 describes longest common prefix arrays. In Section 5 we discuss the actual algorithm and in Section 6 we discuss how an individual match is weighed. Section 7 contains the results of our experiments. Section 8 contains our conclusions and possibilities for future work.

I would like to thank Dr. Kai Ye from Leiden University for his suggestions and information on microRNA. I also thank Dr. Walter A. Kosters and Dr. Hendrik Jan Hoogeboom from Leiden University for their support and their thoughts on suffix trees and suffix arrays.

2 Definitions

Palindromes take the form uvu^r where u is a string of characters, v is an optional single character and u^r is u reversed. In the case of RACECAR, u is RAC and v is E. An example without the optional character in the middle is ABBA, where u is AB. It's also possible for u to be empty since both the empty string and single characters remain unchanged when reversed.

A gapped palindrome takes the form uvu^r where u is a string of characters with minimal length 1 and v is a string of characters. We are not interested in cases where u is empty, since that would include all possible strings as gapped palindromes. While, strictly speaking, strings where v is empty are not gapped palindromes since there is no gap, we will allow such cases in this paper. An example of a gapped palindrome is DETESTED, where u is DET and v is ES.

When working with RNA, we slightly change our definition of a gapped palindrome. Consider the RNA string CGUAAAACG. The three leftmost bases can form three base pairs with the three rightmost bases. The first C matches up to the second G, the first G to the second C and the U to the last A. The two As in the middle remain unmatched.

The entire RNA sequence CGUAAAACG is essentially a gapped palindrome in which the last part has had its Cs replaced by Gs, its Gs by Cs, its As by Us and its Us by As. We define a gapped palindrome in an RNA sequence as a sequences uvu^{cr} where u is a non-empty sequence of bases, v is a sequence of bases and u^{cr} is the reversed complement of u: the reverse of u with all bases replaced by their potential base pair partners.

The length of u is referred to as the *arm length* and the length of v is referred to as the *gap size*. A gapped palindrome with long arms may be considered more relevant than one with very short arms. Likewise the gap size may be of interest as well. A non-gapped palindrome can also be considered a gapped palindrome with a gap size of zero or one.

Finally a maximal gapped palindrome is a gapped palindrome whose arms are as long as possible. In the earlier example we can state that GUAAAC is a gapped palindrome with arm length 1 and gap size 4. However, as part of the string CGUAAAACG, it is not a maximal gapped palindrome as the arms can be extended both inward and outward. Likewise DETESTED can be considered a gapped palindrome where u is DE and v is TEST. This is also not a maximal gapped palindrome as u can be extended to DET and v reduced to ES.

3 Suffix arrays

A *suffix array* [6] contains all suffixes of some string, sorted in lexicographical order. If the original string remains available we can simple store the indexes at which suffixes start, rather than storing the actual suffixes. For example, consider the DNA string ACACACATACACA. It has the following suffixes:

index	suffix
1	ACACACATACACA
2	CACACATACACA
3	ACACATACACA
4	CACATACACA
5	ACATACACA
6	CATACACA
7	ATACACA
8	TACACA
9	ACACA
10	CACA
11	ACA
12	CA
13	A

If we sort these suffixes in lexicographical order we get:

index	suffix
13	A
11	ACA
9	ACACA
1	ACACACATACACA
3	ACACATACACA
5	ACATACACA
7	ATACACA
12	CA
10	CACA
2	CACACATACACA
4	CACATACACA
6	CATACACA
8	TACACA

The suffix array for this string is $(13, 11, 9, 1, 3, 5, 7, 12, 10, 2, 4, 6, 8)$.

Constructing a suffix array is basically the same as sorting a list of words. We can easily do this in $O(n \log n)$ string comparisons. However, the string comparisons each take $O(n)$ character comparisons, meaning we need to do $O(n^2 \log n)$ character comparisons. It is also possible to first construct a suffix tree [10] in linear time using one of many linear time algorithms [7, 9] and then convert it into a suffix array in linear time. Several linear time algorithms that construct suffix arrays directly, without first constructing a suffix tree, have also been developed [1, 3, 4].

4 Longest common prefix arrays

A *longest common prefix array* [6] contains the lengths of the longest common prefix of every two subsequent strings in some array. Longest common prefix arrays can be used to improve the performance of certain algorithms on suffix arrays. Continuing with the previous example, we compute the length of the longest common prefix of each string in the array and the string below it.

index	suffix	lcp	lcp length
13	A	A	1
11	ACA	ACA	3
9	ACACA	ACACA	5
1	ACACACATACACA	ACACA	5
3	ACACATACACA	ACA	3
5	ACATACACA	A	1
7	ATACACA	-	0
12	CA	CA	2
10	CACA	CACA	4
2	CACACATACACA	CACA	4
4	CACATACACA	CA	2
6	CATACACA	-	0
8	TACACA		

The resulting longest common prefix array is (1,3, 5, 5, 3, 1, 0, 2, 4, 4, 2, 0).

The most intuitive way to construct a longest common prefix array is to compare each pair of subsequent suffixes in the suffix array and compare them character by character. This leads to a worst case complexity of $O(n^2)$, but there are more efficient ways to do this. One particularly useful property we can use is that when any two suffixes in the array have a longest common prefix of ℓ characters, the longest common prefix of all pairs of suffixes between the two suffixes is at least ℓ characters long.

A linear time method to compute a longest common prefix array given a suffix array is described by Kasai et al. [2].

5 Finding gapped palindrome density

We need some measure of *gapped palindrome density*. We will move a sliding window across the input and for each frame calculate some value that represents the number and size of gapped palindromes we found in that frame. In order to do so we first create suffix arrays and longest common prefix arrays for both the frame and its reversed complement.

We will consider each suffix in the frame a possible arm of a gapped palindrome. We then look for the longest matching arm in the reversed complement by finding the longest common prefix between that specific suffix and all suffixes in the reversed complement of the frame.

As an example, consider the string CGUAAACG and its reversed complement CGUUUACG. If we look for the longest common prefix that the suffix GUAAACG shares with any suffix of the CGUUUACG, we find it in the suffix GUUUACG. These two suffixes have a longest common prefix of length 2. We can also determine the gap size using the index at which the two suffixes start, which in this case shows a gap size of 2.

If we repeat this process for all suffixes in the frame we find the following gapped palindromes (where * denotes a single unmatched gap character):

CGU**ACG
GU**AC
UA
UA
U*A
CGU**ACG
CG****CG
C******G

As is clear, non-maximal gapped palindromes (whose arms are not maximal or gap size is not minimal) may be found and some gapped palindromes are found twice. We can remedy these things in some way, for example by checking whether the matching arm is located to the right of the initial arm in the original string or checking whether the arms can be extended. However, experimentation has shown that these awkward results have little impact on the relative palindromic weight of frames.

Finding the longest matching arm in the reversed complement of the frame can be done by simply performing a binary search. This would take $O(n \log n)$ time. However, by using the longest common prefix array we can improve our performance to $O(n)$.

Our method is based on a simple linear search, but is significantly more efficient because we use the longest common prefix information to skip through large parts of the array. The pseudo-code in Figure 1 describes how this information is used to skip comparisons between strings or substrings which we already know cannot match.

This function takes a window W composed of the letters A, C, G and T or U. W' is the reverses complement of the window. A special final symbol \$ is appended to both W and W'. Then suffix array S and longest common prefix array L are computed for window W. Likewise, S' and L' are computed for W'. Integers i and j are used to traverse S and S' respectively while k is used to count the current number of matching characters. Finally, w is used to add up the weight of each gapped palindrome that is found.

Each suffix $S[i]$ in the window is considered and compared to the suffixes $S'[j]$ in the reversed complement of the window. If the first unmatched characters in the current suffixes $S[i]$ and $S'[j]$ are identical, they are matched on line 13. If they differ and there may be a better matching suffix for $S[i]$, j is increased by one.

```
$\mathit{WindowWeight}\,(W)$ :
    $W' \leftarrow \mathit{ReverseComplement}(W) + \$$
    $W \leftarrow W + \$$
    $S \leftarrow \mathit{SuffixArray}(W)$
    $L \leftarrow \mathit{LongestCommonPrefixArray}(W)$
    $S' \leftarrow \mathit{SuffixArray}(W')$
    $L' \leftarrow \mathit{LongestCommonPrefixArray}(W')$
    $i \leftarrow 0$
    $j \leftarrow 0$
    $k \leftarrow 0$
    $w \leftarrow 0$
    **while** $i < |W| - 1$ **do**
        **if** $W[S[i] + k] = W'[S'[j] + k]$ **and** $W[S[i] + k] \neq \$$ **then**
            $k \leftarrow k + 1$
        **else if** $W[S[i] + k] > W'[S'[j] + k]$ **and** $W[S[i] + k] \neq \$$
            **and** $k \leq L'[j]$ **and** $j < |W|$ **then**
            $j \leftarrow j + 1$
        **else**
            $w \leftarrow w + \mathit{PalindromeWeight}(W, S[i], S'[j], k)$
            $k \leftarrow \min(k, L[i])$
            $i \leftarrow i + 1$
        **fi**
    **od**
    **return** $w$
```

Figure 1: Calculating the palindromic weight of a window

This is only possible if $S[i]$ is lexicographically larger than $S'[j]$, $S[i]$ has not been completely matched and $S'[j]$ has a smaller common prefix with $S'[j+1]$ than the current match length. If any of these conditions is not met, the current match is optimal. On line 18 the weight of the match is added to the window weight, i is incremented and k is lowered to the number of characters that $S[i+1]$ will at least have in common with $S'[j]$.

The algorithm applies to the string GCAAGC as follows:

The suffix arrays, longest common prefix arrays and counters are initialized:

$W' \leftarrow$ GCTTGC\$
$W \leftarrow$ GCAAGC\$
$S \leftarrow (2, 3, 1, 5, 0, 4, 6)$
$L \leftarrow (1, 0, 1, 0, 2, 0)$
$S' \leftarrow (1, 5, 0, 4, 3, 2, 6)$
$L' \leftarrow (1, 0, 2, 0, 1, 0)$
$i \leftarrow j \leftarrow k \leftarrow w \leftarrow 0$

The **while** loop goes through the following iterations:

- $W[S[i] + k] = W[2 + 0] =$ A
  $W'[S'[j] + k] = W'[1 + 0] =$ C
  $W[S[i] + k] \neq W'[S'[j] + k] \Rightarrow$ don't increase k
  $W[S[i] + k] \not> W'[S'[j] + k] \Rightarrow$ don't increase j
  Best match found for AAGC\$: $i = 0, j = 0, k = 0$
  Match length is 0
  $k \leftarrow \min(k, L[i]) = \min(0, 1) = 0$
  $i \leftarrow i + 1 = 1$

- $W[S[i] + k] = W[3 + 0] =$ A
  $W'[S'[j] + k] = W'[1 + 0] =$ C
  $W[S[i] + k] \neq W'[S'[j] + k] \Rightarrow$ don't increase k
  $W[S[i] + k] \not> W'[S'[j] + k] \Rightarrow$ don't increase j
  Best match found for AGC\$: $i = 1, j = 0, k = 0$
  Match length is 0
  $k \leftarrow \min(k, L[i]) = \min(0, 0) = 0$
  $i \leftarrow i + 1 = 2$

- $W[S[i] + k] = W[1 + 0] =$ C

$W'[S'[j]+k] = W'[1+0] = $ C
$W[S[i]+k] = W'[S'[j]+k] \wedge W[S[i]+k] \neq \$ \Rightarrow$ match one character
$k \leftarrow k+1 = 1$

- $W[S[i]+k] = W[1+1] = $ A
  $W'[S'[j]+k] = W'[1+1] = $ T
  $W[S[i]+k] \neq W'[S'[j]+k] \Rightarrow$ don't increase k
  $W[S[i]+k] \not> W'[S'[j]+k] \Rightarrow$ don't increase j
  Best match found for CAAGC\$: $i=2, j=0, k=1$
  *C**G*
  This is a gapped palindrome, though it is not maximal.
  $k \leftarrow \min(k, L[i]) = \min(1,1) = 1$
  Note that $L[i]$ and $L[i+1]$ share a common prefix which the algorithm does not have to match again.
  $i \leftarrow i+1 = 3$

- $W[S[i]+k] = W[5+1] = \$$
  $W'[S'[j]+k] = W'[1+1] = $ T
  $W[S[i]+k] = \$ \Rightarrow$ don't increase k
  $W[S[i]+k] = \$ \Rightarrow$ don't increase j
  Best match found for C\$: $i=3, j=0, k=1$
  ****GC
  This palindrome has no gap.
  $k \leftarrow \min(k, L[i]) = \min(1,0) = 0$
  $i \leftarrow i+1 = 4$

- $W[S[i]+k] = W[0+0] = $ G
  $W'[S'[j]+k] = W'[1+0] = $ C
  $W[S[i]+k] \neq W'[S'[j]+k] \Rightarrow$ don't increase k
  $W[S[i]+k] > W'[S'[j]+k] \wedge W[S[i]+k] \neq \$ \wedge k \leq L'[j] \wedge j < |W| \Rightarrow$ move to next suffix in S'
  $j \leftarrow j+1 = 1$

- $W[S[i]+k] = W[0+0] = $ G
  $W'[S'[j]+k] = W'[5+0] = $ C
  $W[S[i]+k] \neq W'[S'[j]+k] \Rightarrow$ don't increase k
  $W[S[i]+k] > W'[S'[j]+k] \wedge W[S[i]+k] \neq \$ \wedge k \leq L'[j] \wedge j < |W| \Rightarrow$ move to next suffix in S'

$j \leftarrow j + 1 = 2$

- $W[S[i] + k] = W[0 + 0] =$ G
  $W'[S'[j] + k] = W'[0 + 0] =$ G
  $W[S[i] + k] = W'[S'[j] + k] \wedge W[S[i] + k] \neq \$ \Rightarrow$ match one character
  $k \leftarrow k + 1 = 1$

- $W[S[i] + k] = W[0 + 1] =$ C
  $W'[S'[j] + k] = W'[0 + 1] =$ C
  $W[S[i] + k] = W'[S'[j] + k] \wedge W[S[i] + k] \neq \$ \Rightarrow$ match one character
  $k \leftarrow k + 1 = 2$

- $W[S[i] + k] = W[0 + 2] =$ A
  $W'[S'[j] + k] = W'[0 + 2] =$ T
  $W[S[i] + k] \neq W'[S'[j] + k] \Rightarrow$ don't increase k
  $W[S[i] + k] \not> W'[S'[j] + k] \Rightarrow$ don't increase j
  Best match found for GCAAGC\$: $i = 4, j = 2, k = 2$
  GC**GC
  This is a maximal gapped palindrome.
  $k \leftarrow \min(k, L[i]) = \min(2, 2) = 2$
  Once again the longest common prefix array allows us to carry the current match length over to the next suffix.
  $i \leftarrow i + 1 = 5$

- $W[S[i] + k] = W[4 + 2] = \$$
  $W'[S'[j] + k] = W'[0 + 2] =$ T
  $W[S[i] + k] = \$ \Rightarrow$ don't increase k
  $W[S[i] + k] = \$ \Rightarrow$ don't increase j
  Best match found for GCAAGC\$: $i = 5, j = 2, k = 2$
  ****GC
  In this case the two arms overlap completely. Such cases are discussed in the next section.
  $k \leftarrow \min(k, L[i]) = \min(2, 0) = 0$
  $i \leftarrow i + 1 = 6$

The **while** loop ends because $i = |W| - 1$.
The algorithm finally returns the total weight of all the gapped palindromes that were found.

6 Gapped palindrome weights

Whenever a best match is found for a given suffix of the window, a weight is calculated using the current values of $S[i]$, $S'[j]$ and k. The first arm starts at position $S[i]$ and the second arm starts at position $|S| - S'[j] - k$. The arm length is k and the gap size is $|S| - S[i] - S'[j] - 2k$.

With this information we can create any number of weight functions. For the purpose of this paper, gapped palindromes weigh $(k - c)^2$ if $k > c$ and 0 if $k \leq c$ where c is a predefined constant.

The given algorithm finds the longest matching string for each suffix of the window. However, those matches may not be maximal. The two arms that are found may even overlap, or the second arm may be to the left of the arm. Some of these problems can be solved fairly easily.

Overlapping arms and reversed arms are easily found by checking whether the second arm starts after the position where the first arm ends. If overlap is found, the match can be ignored, or the size of the overlap can be subtracted from the arm length.

Palindromes with non-maximal arm length can be recognized by comparing the match of the current suffix to the suffix that starts one position earlier. As suffixes in the window are handled in alphabetical order, it is necessary to save matches as they are found and only weigh them after all suffixes have been matched. If the match found for the suffix starting at position i is smaller than that found for the suffix starting at position $i - 1$, the gapped palindrome at position i that is found is either non-maximal, or it overlaps with a different gapped palindrome that has longer arms. Either way, the match is ignored.

7 Experiments

The following experiments were performed on build 36 of human chromosome 14 [8].

Given a large input consisting of the letters A, C, G and T or U, our program reads a window, calculates a palindromic weight and slides the window forward by a fixed amount to repeat this process until the entire input has been processed. This produces output like this:

100 558 050	130
100 558 100	123
100 558 150	100
100 558 200	80
100 558 250	87
100 558 300	79
100 558 350	59
100 558 400	93
100 558 450	58
100 558 500	68

The number on the left is the position of the window and the number on the right is the weight that has been calculated for that window.

The graph in Figure 2 shows the palindromic weight of windows in a small part of human chromosome 14. The graph covers positions 100 558 000 to 100 563 000 with window size 200, shift size 1 and weight function $(k-5)^2$ where k is the length of the maximal gapped palindrome. The palindromic weight of the windows is marked in red. The blue areas mark known microRNA locations.

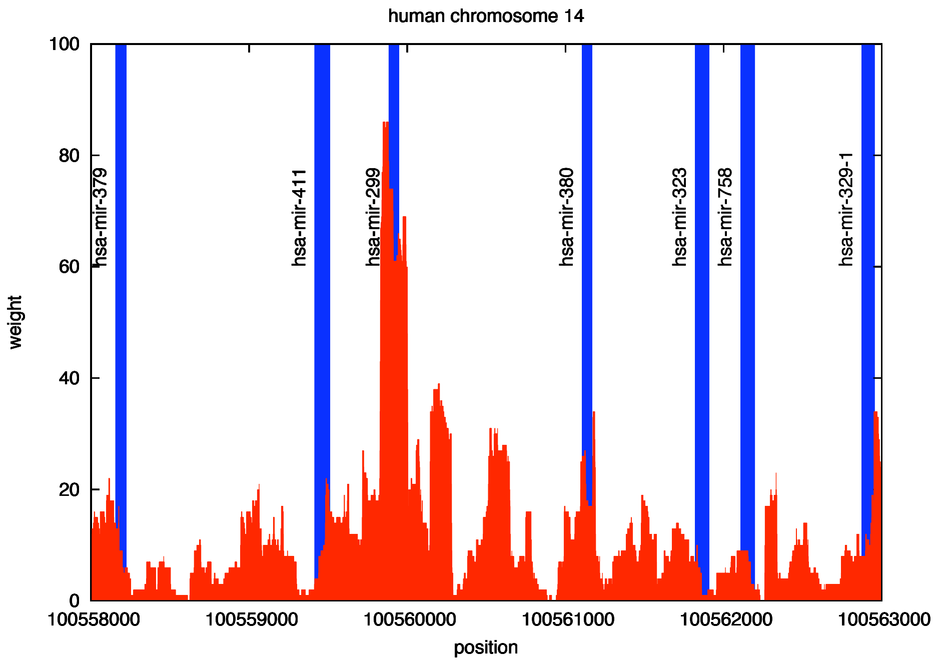

Figure 2: Palindromic weights and microRNA locations

While some known microRNA locations match up to peaks in palindromic window weights as calculated by our algorithm, hsa-mir-299 in particular, other such locations have very low weights.

We run the algorithm with window size 100, shift size 50 and weight function $(k-2)^2$ on the full chromosome. We pay special attention to the region from position 100 558 000 to 100 603 000, which contains several microRNA sequences we know of. In addition, we apply the algorithm to a randomly generated DNA sequence. This results in the following weights:

windows	average weight
random input	77.4
selected region	88.6
windows containing known microRNA	91.1
human chromosome 14	94.4

Unsurprisingly, human DNA tends to contain more long gapped palindromes than a randomly generated string. Within the region which contains several known microRNA sequences, the windows that contain microRNA sequences have slightly higher average weights than windows that don't. However, the difference is small and the entire chromosome has an even higher average palindromic weight.

The same experiment with window size 200 and shift size 100 results in the following weights:

windows	average weight
random input	260.0
selected region	288.7
windows containing known microRNA	291.6
human chromosome 14	306.9

The same conclusions hold true with these parameters. Random input results in low weights. Within the selected region, windows which contain known microRNA sequences have slightly higher weights, but not as high as the entire chromosome.

The program was run on a 2.16 GHz Intel Core 2 Duo. The following table lists runtimes of the program processing all available data for chromosome 14, which consists of 88 290 585 base pairs:

input size	window size	shift size	number of windows	runtime
88 290 585	100	50	882 905	3m31s
88 290 585	200	100	882 904	2m56s

The following table lists runtimes of the program processing randomly generated strings of 1 000 000 base pairs:

input size	window size	shift size	number of windows	runtime
1 000 000	100	50	19 999	2.278s
1 000 000	200	100	9 999	1.947s
1 000 000	1000	500	1 999	1.801s

8 Conclusions

The algorithm presented in this paper takes a DNA or RNA string as input and calculates a palindromic weight that represents the number and size of gapped palindromes in the input. Suffix arrays and longest common prefix arrays are calculated for the input which enables quick identification of potentially maximal gapped palindromes.

Given the small differences in the average palindromic weight of windows that are known to be of interest and other windows, as well as the fact that the average weight across the entire chromosome is higher than the average weight of known microRNA locations, the algorithm does not seem to accurately predict microRNA locations. However, the algorithm does succeed in quickly finding all maximal gapped palindromes in a given string. Unfortunately, the measure of palindrome density does not yet provide statistically significant results.

A possibility for future work is experimenting with different weight functions and additional restrictions such as proper nesting of inverted repeats. Allowing small errors in matches could also be interesting, though that severely reduces the benefit of using suffix arrays and longest common prefix arrays.

The ability to quickly find the longest matching inverted repeat for every substring of a given input may also have uses outside of prediction of microRNA locations. Finally, it would be worth investigating why some microRNA sequences have extremely low palindromic weights according to this algorithm.

References

[1] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings 30th International Colloquium on Automata, Languages and Programming, ICALP '03*, number 2719 in LNCS, pages 943–955. Springer, 2003.

[2] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings Combinatorial Pattern Matching: 12th Annual Symposium, CPM 2001*, number 2089 in LNCS, pages 181–192. Springer, 2001.

[3] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proceedings Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003*, number 2676 in LNCS, pages 186–199. Springer, 2003.

[4] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003*, number 2676 in LNCS, pages 200–210. Springer, 2003.

[5] R. Kolpakov and G. Kucherov. Searching for gapped palindromes. In *Proceedings Combinatorial Pattern Matching: 19th Annual Symposium, CPM 2008*, number 5029 in LNCS, pages 18–30. Springer, 2008.

[6] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. Society for Industrial and Applied Mathematics, 1990.

[7] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[8] Website of the National Center for Biotechnology Information. `www.ncbi.nlm.nih.gov` [retrieved February 23, 2010].

[9] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[10] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory, SWAT '73*, pages 1–11, 1973.