BWT Arrays and Mismatching Trees: A New Way for String Matching with *k* Mismatches

[1]Yangjun Chen, [2]Yujia Wu, Dept. Applied Computer Science, University of Winnipeg, Canada
email: [1] y.chen@uwinnipeg.ca, [2]wyj1128@yahoo.com

Introduction

By the string matching with *k* mismatches, we mean a problem to find all the occurrences of a pattern string *r* in a target string *s* with each occurrence having up to *k* positions different between *r* and *s*. This problem is important for DNA databases to support the biological research, where we need to locate all the appearances of a read (a short DNA sequence) in a genome (a very long NDA sequence) for disease diagnosis or some other purposes. Due to polymorphisms or mutations among individuals or even sequencing errors, the read may disagree in some positions at any of its occurrences in the genome.

As an example, consider a target *s* = *ccacacagaagcc*, and a pattern *r* = *aaaaacaaac*. Assume that *k* = 4. Let us see whether there is an occurrence of *r* with *k* mismatches that starts at the third position in *s*.

```
        a a a a a c a a a c
    c c a c a c a g a a g c c
        |   |   |       |
```

At only four locations *s* and *r* have different characters, implying an occurrence of *r* starting at the third position of *s*. Note that the case *k* = 0 is the extensively studied string matching problem.

This topic has received much attention in the research community and many efficient algorithms have been proposed, such as [1, 2, 4, 6, 7, 9, 10]. Among them, [6] and [7] are two on-line algorithms (using no indexes) with the worst-case time complexities bounded by O(*kn* + *m*log*m*), where *n* = |*s*| and *m* = |*r*|. By these two methods, the *mismatch information* among substrings of *r* is used to speed up the working process. The methods discussed in [2] and [10] are also on-line strategies, but with a slightly better time complexity O(*n*log*k*). By these two methods, the *periodicity* within *r* is utilized. Only the algorithms discussed in [4, 9] are index-based. By the method discussed in [4], a (compressed) suffix tree over *s* is created. Then, a brute-force tree searching is conducted to find all the possible string matchings with *k* mismatches. Its time complexity is bounded by O($m + n + (\log n)^k / k!$), where *c* is a very large constant. For DNA databases, this time complexity can be much worse than O(*nk*) since *n* tends to be very large and *k* is often set to be larger than 10. By the method discussed in [9], *s* is transformed to a *BWT*-array (denoted *BWT*(*s*)) as an index [8]. In comparison with *suffix trees*, *BWT*(*s*) uses much less space [3, 5]. However, the time complexity of [9] is bounded by O(*mn′* + *n*), where *n′* is the number of leaf nodes of a tree (forest) produced during the search of *BWT*(*s*). Again, this time requirement can also be much worse than the best on-line algorithm for large patterns. Thus, simply indexing *s* is not always helpful for *k* mismatches. The reason for this is that in both the above index-based methods neither mismatch information nor periodicity within *r* is employed, leading to a lot of redundancy, which shadows the benefits brought by indexes. However, to use such information efficiently and effectively in an indexing environment is very challenging since in this case *s* will no longer be scanned character by character and the auxiliary information extracted from *r* cannot be simply integrated into an index searching process.

In this paper, we address this issue, and propose a new method for the *k*-mismatch problem, based on a BWT-transformation,

Methods

1. BWT Transformation

$s = a_1\ c_1\ a_2\ g_2\ a_3\ c_2\ a_4\ \$$ ⇒ F: $\$\ a_4\ a_3\ a_1\ a_2\ c_2\ c_1\ g_1$; L: $a_4\ c_2\ g_1\ \$\ c_1\ a_3\ a_1\ a_2$

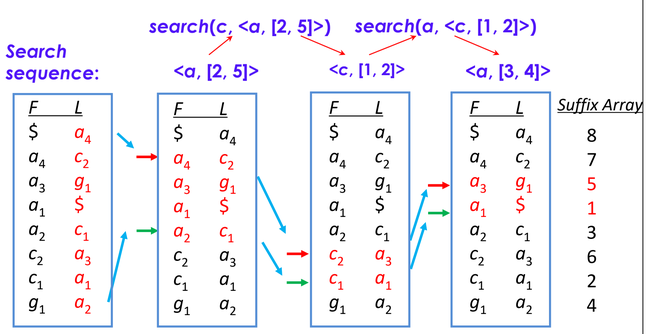
$L[i] = \$$, if $J[i] = 0$;
$L[i] = s[J[i] - 1]$, otherwise

where J is the suffix array of s.

LF mapping: $F[i] = L[i]$'s successor

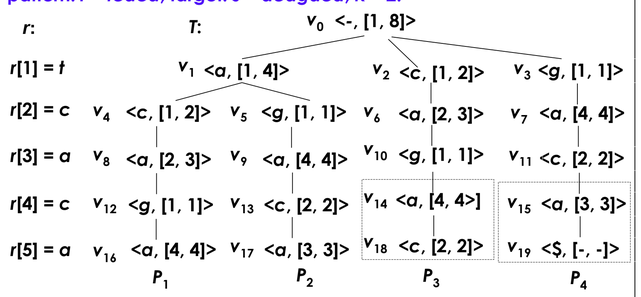
Rank correspondence: $rank_F(e) = rank_L(e)$

2. Search Sequences



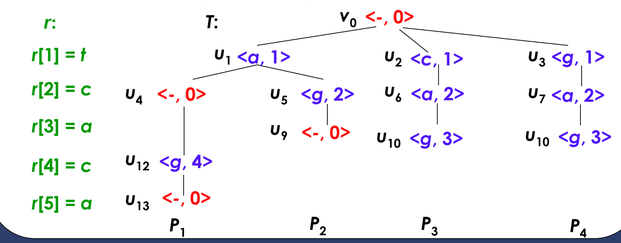
3. Search Trees

pattern: *r* = *tcaca*; target: *s* = *acagaca*; *k* = 2.



4. Mismatching Trees

pattern: *r* = *tcaca*; target: *s* = *acagaca*; *k* = 2.



Experiments

In our experiments, we have tested altogether four different methods:

- *BWT-based* [9] (BWT for short),
- *Amir's method* [2] (Amir for short),
- *Cole's method* [4] (Cole for short),
- *Algorithm **A** discussed in this paper* (*A*( ) for short)

By the Cole's, a suffix tree for a target is constructed. (The code for constructing suffix trees is taken from the *gsuffix* package: http:://gsuffix.Sourceforge.net/).

All the four methods are implemented in C++, compiled by GNU make utility with optimization of level 2. In addition, all of our experiments are performed on a 64-bit Ubuntu operating system, run on a single core of a 2.40GHz Intel Xeon E5-2630 processor with 32GB RAM.

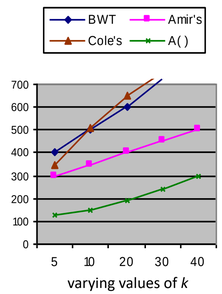
For the tests, five reference genomes are used:

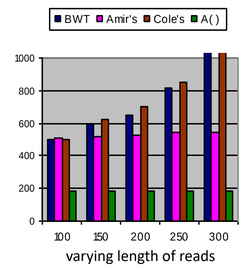
Genomes	Genome sizes (bp)
Rat chr1 (Rnor_6.0)	290,094,217
C. merolae (ASM9120v1)	16,728,967
C. elegans (WBcel235)	103,022,290
Zebra fish (GRCz10)	1,464,443,456
Rat (Rnor_6.0)	2,909,701,677

All the pattern strings are created by simulating reads from the five genomes shown in the above table, with varying lengths and amounts. It is done by using the *wgsim* program included in the *SAMtools* package [11] with default model for single read simulation.

To store *BWT*( ), we use 2 bits to represent a character ∈ {*a*, *c*, *g*, *t*} and store 4 *rankAll* values (respectively in A_a, A_c, A_g, and A_t) for every 4 elements (in *L*) with each taking 32 bits.

In the following figures, we report the average time of testing the Rat (Rnor_6.0) for matching 100 reads of length 100 to 300 bps. From this figure, we can see that Algorithm *A*( ) outperforms all the other three methods. But the Amir's method is better than the other two methods. The BWT-based and the Cole's method are comparable. However, for small *k*, the Cole's is a little bit better than the BWT-based method while for large *k* their performances are reversed.





Conclusions

In this paper, a new method to do the string matching with *k* mismatches is proposed. Its main idea is to transform the reverse of target string *s* to *BWT*( ) and use the mismatch information over a pattern string *r* to speed up the computation. Its time complexity is bounded by O(*kn′* + *n* + *m*log*m*), where *m* = |*r*|, *n* = |*s*|, and *n′* is the number of leaf nodes of a tree structure produced during the search of a *BWT*(*s*). Our experiments show that it has a better running time than any existing on-line and index-based algorithms.

Bibliography

[1] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Communication of the ACM*, Vol. 23, No. 1, pp. 333 -340, June 1975.

[2] A. Amir, M. Lewenstein and E. Porat, Faster algorithms for string matching with *k* mismatches, *Journal of Algorithms*, Vol. 50, No. 2, Feb.2004, pp. 257-275.

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

[4] R. Cole, L. Gottlieb, and M. Lewenstein, Dictionary Matching and Indexing with Errors and Don't Cares, *STOC'04*, pp. 91 – 100, 2004.

[5] P. Ferragina and G. Manzini, Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pp. 390 - 398. IEEE, 2000.

[6] Z. Galil and R. Giancarlo, Improved string matching with *k* mismatches, *ACM SIGACT News*, Vol. 17, Issue 4, Spring 1986, pp. 52b- 54.

[7] G.M. Landau and U. Vishkin, Efficient string matching with *k* mismatches, *Theoretical Computer Science*, Vol. 43, pp. 239 – 249, 1986.

[8] G.M. Landau and U. Vishkin, Efficient string matching with *k* mismatches, *Theoretical Computer Science*, Vol. 43, pp. 239 – 249, 1986.

[9] H. Li and R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler Transform, *Bioinformatics*, Vol. 25 no. 14 2009, pp. 1754–1760.

[10] M. Nicolas and S. Rajasekarian, On string matching with k mismatches, https://arxiv.org/pdf/1307.1406, 2013.

[11] H. Li, wgsim: a small tool for simulating sequence reads from a reference genome, https://github.com/lh3/wgsim/, 2014.

printed by MegaPrint Inc. www.postersession.com