Implementing efficient suffix sorting algorithms

Nurgissa Umatay

Abstract

My aim is to provide an implementation of a suffix sorting algorithm[2] and rough draft of possible enhancements to the algorithm in order to make it memory and time efficient. I provide evidence of potential improvements however failed to present a practical algorithm for efficient suffix array construction.

Introduction

Suffix trees and suffix arrays are widely used data structures to process sequences and strings. Although both of them produces same results, suffix trees are not widely used in practice. The reason is complexness of implementation and maintenance. This paves the way to suffix arrays that are actually more space efficient and easy to maintain. In this way, very large genomes now can be analyzed and indexed way efficiently. *Difference cover 3 (DC3)[2]* is a very elegant algorithm for constructing suffix arrays in linear time. It is quite fast on dealing with small and big data sets. I intend to explore of efficiency of this algorithm and find a bottleneck if there is so. I would like to experiment on bigger dataset and demonstrate that the algorithm could be enhanced on the scale of constant factors. I also plan to optimize my implementation of the algorithm it to achieve a good speedup on multi core CPU on ugradx cluster.

Prior work

The research on suffix arrays thrived during mid 1990s and early 2000s. Initially Manber and Myers introduced this data structure with an algorithm that runs in O(n log n). After that, a race for linear construction of suffix arrays has begun. Mostly the best algorithms to construct suffix arrays in linear time were build on suffix trees. But they were space demanding and inefficient regarding computer architectures.

The race has stopped in 2003 when the problem of constructing suffix arrays in linear time were solved by multiple group of researchers.[2, 3, 4] I chose DC3 as a basis of this paper for several reasons. It is simple(skew algorithm is very tricky that requires high entrance barrier) yet powerful. It can also be efficiently parallelized.[5] The algorithm requires a lot of memory for the input sequences that are very large. From previous implementations, I learned that there is an open hitting "out of memory" problem due to usage of recursion in first step of the algorithm. Developers deal very carefully with allocating and freeing memory in their programs.

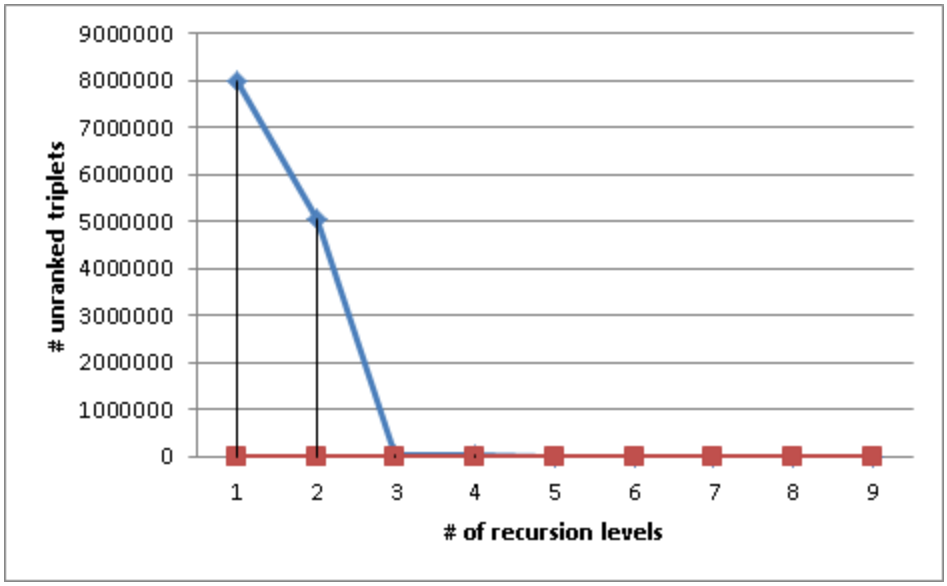
Methods and software

In this project I used two different versions of implementation of DC3 algorithm. One of them I implemented in python by myself with respect to OOP design. Other version of the it is direct translation from C++ code to python[8]. Both of them accomplish a task of constructing suffix arrays differently in terms of time and memory usage. Throughout the project I implemented different test versions of skew algorithm. All of the programs were very memory consumptive. After short research it turned out that the algorithm's effectiveness decreases drastically after a certain threshold. For bigger data sets it has many triplet collisions per round that results to call itself recursively. A below table shows an assessment result of the algorithm on DNA Chromosome of Drosophila. A third column shows a number of unresolved ranks per round.

# of repeated triplets	#of unique ranks per round	# of unresolved ranks	Level of recursion
7999960	66	7999894	0
5333307	261808	5071499	1
3555538	3518654	36884	2
2370359	2355181	15178	3
1580240	1575284	4956	4
1053494	1052242	1252	5
702330	702110	220	6
468220	468219	1	7
312147	312147	0	8

Table 1. Rank comparison results

The skew algorithm works great when the input text has many characters. It works by slicing input into two different sets such as sample and non-sample. Then each set results of generating triplets of suffices. It guarantees to shrink the original input by two-third when it finds a collision of triplets. This is a great approach when we deal with small data set and number of collided triplet is big . However, it becomes very inefficient on later stages of indexing big dataset when collided triplet decreases. For instance, on the sixth round of the recursion (Table 1.) there are only 220 unresolved triplets when size of the input is 720 330 nucleotides. Since the logic of DC3 does not encounter this problem, it just passes the input to the next round of recursion.



Picture 1. Rank comparison results

Lets go through an example of colliding triplets and examine how can we restore rankings for T = “yabbadabbado”. In the Appendix I gave a step by step example of DC3 algorithm upto this step.

```
R      = [abb][ada][bba][do$][bba][dab][bad][o$$]
ranks =   1    2    4    6    4    5    3    7
```

After we constructed triplets to B1 and B2 sets and sorted with radix sort, we got to the point where `[bba]` triplet was given 4th rank. But there are two same triplets and which of them should come first? This really depends on the origination of these triplets and position to each other. In this simple example we pick ranks of next triplets where `[do$]->6` and `[dab]->5`. As we see `[do$]>[dab]` will result to conclusion `[bba][do$] > [bba][dab]`. We were able to easily restore collision by simply comparing neighbors. One can argue that this is simple example and in reality we don't get perfect line up. Let’s take a look at below table.

# of collided triplets A	#of uniqe ranks B	A - B	next_to	diff_set	same_set
1999960	66	1999894	54096	64	1999830
1333307	246353	1086954	374	180867	906087
888872	881499	7373	5	1919	5454
592582	589888	2694	0	1434	1260
395055	394110	945	0	427	518
263370	263068	302	0	88	214

175580	175537	43	0	41	2
117054	117054	0	0	0	0

Table 2. Rank comparison results with additional data

The table is same as previous except it has additional fields such as diff_set, same set and next_to. **Next_to** indicates a number of same triplets that are located next to each other in a given input. **Diff_set** displays a number of collided triplets that belong to same set and **same_set** is reverse to diff_set. It presents a number of collided triplets that belong to same set. As these three value decreases the chance of restoring ranks increases.

Experimental results

The performance of the implemented algorithm is tested with following dataset.

Name	Size	Time
Drosophila.dna.chromosome	~24 400000 ntd	NA
Drosophila.dna.chromosome(half)	~12 200 000 ntd	664 sec
Drosophila.dna.chromosome(quarter)	~6 100 000 ntd	286 sec
Random.DNA	~1 200 000 ntd	65.5 sec
Repetative.letter	~1 867 000 ntd	77.6 sec

Table 2. Description of the data set

Conclusions

The main goal of this paper is project was not only implementing DC3 algorithm, but bringing novelty to this very explored area. The result of the project was implementation of a linear time algorithm for suffix sorting with integer alphabets. Although the algorithm becomes easy after implementation, but the path was not that. I was not able to achieve results that I planned initially. There were many unexpected obstacles that hindered reaching them. I plan to continue to work on this project and implement pre computation ranking and avoid redundant of recursive rounds.

References

[1] U. Manber and E. W. Myers. Suffix arrays: A new method for online string searches. SIAM Journal on Computing, 22(5):935–948, October 1993.

[2] J. K¨arkka¨inen and P. Sanders. Simple linear work suffix array construction. In Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP 2003), volume 2719 of Lecture Notes in Computer Science, pages 943–955. Springer Verlag, June 2003.

[3] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear- time construction of suffix arrays. In Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003), volume 2676 of Lecture Notes in Computer Science, pages 186–199. Springer Verlag, June 2003.

[4] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003), volume 2676 of Lecture Notes in Computer Science, pages 200–210. Springer Verlag, June 2003.

[5] Suffix Array Construction on GPU
http://www.cs.cmu.edu/afs/cs/user/chaominy/www/418_project/index.html

[6] The DC3 Algorithm Made Simple
http://spencer-carroll.com/the-dc3-algorithm-made-simple/

[7] https://github.com/Efficient-Functional-Datastructures/blob/master/yabbadabbado.txt

[8] https://code.google.com/p/pysuffix/

Appendix[7]

Step by step explanation of DC3

```
i   | 0 1 2 3 4 5 6 7 8 9 10 11 12
t_i | y a b b a d a b b a d  o  $

v = 3

 1 abb|adabbado
 2 bba|dabbado
 4 ada|bbado
 5 dab|bado
 7 bba|do
 8 bad|o
10 do
11 o

B_0 = {0, 3, 6, 9, 12}
B_1 = {1, 4, 7, 10}
B_2 = {2, 5, 8, 11}
C   = {1, 4, 7, 10, 2, 5, 8, 11}

R_1 = [abb][ada][bba][do$]
R_2 = [bba][dab][bad][o$$]

R     = [abb][ada][bba][do$][bba][dab][bad][o$$]
ranks =   1    2    4    6    4    5    3    7
```