12/15/2016

Packed suffix array

Farhaan Jalia[1] *, Kanishta Agarwal[2], Amit Gupta[2], Gaurav B G[2]

[1]Department of Computer Science, fjalia@cs.stonybrook.edu, [2]Department of Computer Science.

*To whom correspondence should be addressed

Abstract

Motivation: The suffix array and its variants are text-indexing data structures that have become indispensable in the field of bioinformatics. A recent paper on, RapMap, uses a suffix array (among other data structures) to quickly look through the transcriptome for matching substrings from the reads. For this purpose, the suffix array is initially built on the transcriptome, in an indexing phase, before the mapping process begins. Each character in the suffix array occupies one byte in memory. However, the DNA/RNA alphabet contains only 4 characters [A, T, G, C] which can be encoded using 2 bits of data. Binary encoding the sequence could reduce the space used by the suffix array by 75%.

Results: - We successfully completed the packed suffix array implementation and verified the accuracy on large datasets. As a result of this 2-bit packed suffix array implementation we have reduced the string representation size by 75%.

Availability: https://github.com/Kanisht/CompactSuffixArray

Contact:fjalia@cs.stonybrook.edu

1 Introduction

The problem of finding the occurrences of a pattern string in a given text is one of the most fundamental computational tasks in bioinformatics. In most bioinformatics applications, the text is a huge database onto which a large volume of pattern queries are thrown. In such cases, precomputing an indexed data structure of the text allows efficient processing of pattern searches.

One simple and effective data structure is a suffix array, which informally is a list of the starting positions of the suffixes of the text, sorted by their alphabetical order. Suffix arrays are easy to understand and implement and form the basis for a host of other sophisticated indexing techniques.

The popularity of suffix arrays in bioinformatics is evident from their application in a range of tasks such as pairwise sequence alignment [1–3], error correction of reads from high-throughput sequencers [4, 5], prefix–suffix match finding for genome assembly [6, 7], *k*-mer counting [8] and sequence clustering [9], as well as the development of suffix array software explicitly aimed at bioinformatics applications.

A recent paper, RapMap, also uses suffix array (among other data structures) to quickly look through the transcriptome for matching substrings from the reads. For this purpose, the suffix array is initially built on the transcriptome, in an indexing phase, before the mapping process begins. Each character in the suffix array occupies one byte in memory. The DNA/RNA alphabet contains only 4 characters [A,T,G,C] which can be encoded using 2 bits of data. Binary encoding the sequence will reduce the space used by the suffix array by 75%.

However, real sequencing data (when the suffix array are used in practice) contains a delimiter between separate segments of the sequence (called transcripts). To overcome this, we are building a generalized suffix array, and encoding this delimiter using a rank data structure to keep the suffix array size to a minimum.

Thus, in this project, we have devised an efficient way to include an encoding of the delimiter in our suffix array without compromising on the space used by the suffix array. This project entails programming a suffix array using binary encoding, along with associated functions, such as construction, building a rank data structure, search, and timing results for these operations.

2 Methods

Two important factors to consider in creation of packed Suffix array are:

1) Representation of nucleotides
2) Delimiter between reads

Representation of nucleotides: There are only 4 characters (A, C, G, T) to represent in a genomic sequence. Hence, we only need 2 bits to represent each character. A is represented as 00, T is represented as 01, G is represented as 10 and C is represented 11. A typical genome has thousands of reads of over 2000 characters each. These reads are concatenated as 1 string to form the input for suffix array. This concatenated string is 2bit encoded string.

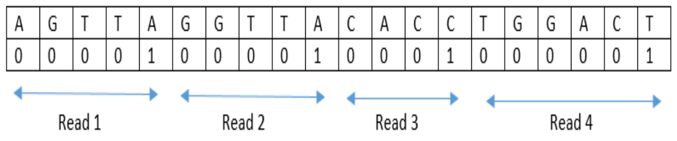
Delimiter between reads

When the reads are concatenated, we also need to represent where one read ends and the other starts. This is necessary because, when we query for a substring in the suffix array, we should make sure not to return positive results for substrings overlapping one read with other. The classical suffix array implementation adds a delimiting character at the end of each read. This character is usually a character which is not a character in the search domain (For e.g. $ symbol). If we use a different character to act as a delimiter, we cannot use the 2-bit logic to represent the nucleotides. Hence, we are using a *Rank* data structure. It is a static bit sequence data structure which answers arbitrary length rank queries in O(1) time, and provides implicit compression. The operation *rank*(*i*) is defined as the number of set bits (1s) in the range [0, *i*]. In the bit string figure shown below, the answer to *rank* (10) is 2. It is what is known as a succinct data structure, which means that even though it is compressed, we don't need to decompress the whole thing it to operate on it efficiently. We have used the *rank9* library used in *rapmap* for this purpose. Also, the additional memory used for this data structure is only 1 bit per character. Thus, this solves the problem of having a delimiter character between every read, efficiently.

Building Packed suffix array:

The implementation involves the following steps.

1. The program takes the path to the input file.
2. The size ‘s’ of input file is read using fstat API and a temp array of size ‘s’ is allocated to hold the concatenated string of all reads.
3. Each read string is read from the file into a buffer and appended to the temp array.
4. The rank structure is appended with 1 at the end of every read. All the other bits are set to 0. The figure below shows a snapshot of the rank structure for an arbitrary sequence of characters.



A	G	T	T	A	G	G	T	T	A	C	A	C	C	T	G	G	A	C	T
0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1

Figure 1

5. Suffix array is built using *divsufsort* library. (This is the same library used in *Rapmap* to construct the suffix array).
6. Each character of the temp array is represented as a 2bit encoded character and stored in a new array ‘S’. We pack 4 characters represented as 2 bits each in one byte. The below figure shows the difference between classical suffix array implementation and the packed suffix array implementation. The code snippet of packing the 2-bit representation of characters is shown below.

```
for (int i = 0; i < in_size; i++) {
                        if (in[i] == 'A') {
                                  data[i/4]   =   data[i/4]   |
(VALUE_A <<((i % 4) * 2));
                        }
                        else if(in[i] == 'T') {
                   data[i/4] = data[i/4] | (VALUE_T <<((i
% 4) * 2));
              }
                        else if (in[i] == 'G') {
                   data[i/4] = data[i/4] | (VALUE_G <<((i
% 4) * 2));
              }
                        else if (in[i] == 'C') {
                   data[i/4] = data[i/4] | (VALUE_C <<((i
% 4) * 2));
              }
              }
```

7. The temp array is now freed. ‘S’ array is the packed representation of the characters which our suffix array will use for substring querying.

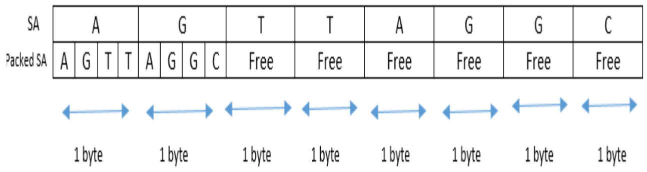


Figure 2

Querying from Packed suffix array:
Involves the following steps.

1. The user is prompted for the string to search.
2. Because the suffixes in the suffix array are all sorted, we can perform binary search over the suffix array to search for the search string.
3. We maintain the start rank when we start searching for the user search string in the suffix. This is necessary in order to not return positive results for a substring found which is an overlap of 2 contiguous reads. Thus, when the rank changes while in the process of character comparisons, we break from comparing any further characters.
4. The indexes maintained in Suffix array needs to be mapped to the appropriate character in the packed suffix array. The method used to do this mapping is as below.

```
char get_values(unsigned char *data, long index) {
        return (data[index/4] >> ((index % 4) * 2)) & 3;
}
```

5. We also maintain a global counter for the maximum match substring found and the index at which it was found. This value is returned to the user if the complete search string in not found.
6. This implementation takes O (n log m) time, where m is the length of the concatenated reads and n is the size is the of the maximum length read.

3 Testing

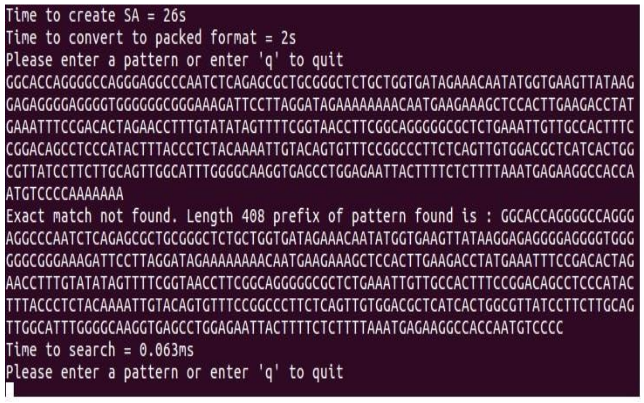
Our test cases comprise of DNA strings samples in the FASTA format. A sequence in FASTA format can contain several sentences. Each sequence in FASTA format begins with a single-line description, followed by lines of sequence data. This description line begins with a (">") symbol.

An example sequence in FASTA format is:

```
>ENST00000367585
CAGCGCTCCCGAGGCCGCGGGAGCCTGCAGAGA
GGACAGCCGGCCTGCGCCGGGACATGCGGCCCC
AGGAGCTCCCCAGGCTCGCGTTCCCGTTGCTGCT
GTTGCTGTTGCTGCTGCTGCCGCCGCCGCCGTGCC
CTGCCCACAGCGCCACGCGCTTCGACCCCACCTG
GGAGTCCCTGGACGCCCGCCAGCTGCCCGCGTGG
TTTGACCAGGCCAAGTTCGGCATCTTCATCCACT
GGGGAGTGTTTTCCGTGCCCAGCTTCGGTAGCGA
GTGGTTCTGGTGGTATTGGCAAAAGGAAAAGATA
CCGAAGTATGTGGAATTTATGAAAGATAATTACC
CTCCTAGTTTCAAATATGAAGATTTTGGACCACT
ATTTACAGCAAAATTTTTTAATGCCAACCAGTGG
GCAGATATTTTTCAGGCCTCTGGTGCCAAATACA
TTGTCTTAACTTCCAAACATCATGAAGCAACTTGT
AGAGACAGTTTCATGTGGAGGAAATCTTTTGATG
AATATTGGGCCCACACTAGATGGCACCATTTCTG
TAGTTTTTGAGGAGCGACTGAGGCAAATGGGGTC
CTGGCTAAAAGTCAATGGAGAAGCTATTTATGAA
ACCCATACCTGGCGATCCCAGAATGACACTGTCA
```

To validate the proper working of the code we created the sample set of positive and negative queries and ran it on our code.
We then ran it on these sets to fine tune our parameters in the algorithm.

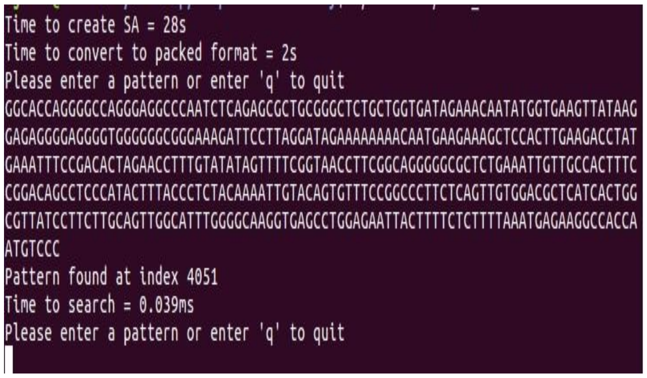
Below are the sample results that we found on running queries from both positive and negative sets.
Figure 3 shows the running time and partial pattern found during the search



```
Time to create SA = 26s
Time to convert to packed format = 2s
Please enter a pattern or enter 'q' to quit
GGCACCAGGGGCCAGGGAGGCCCAATCTCAGAGCGCTGCGGGCTCTGCTGGTGATAGAAACAATATGGTGAAGTTATAAG
GAGAGGGGAGGGGTGGGGGGCGGGAAAGATTCCTTAGGATAGAAAAAAAACAATGAAGAAAGCTCCACTTGAAGACCTAT
GAAATTTCCGACACTAGAACCTTTGTATATAGTTTTCGGTAACCTTCGGCAGGGGGCGCTCTGAAATTGTTGCCACTTTC
CGGACAGCCTCCCATACTTTACCCTCTACAAAATTGTACAGTGTTTCCGGCCCTTCTCAGTTGTGGACGCTCATCACTGG
CGTTATCCTTCTTGCAGTTGGCATTTGGGGCAAGGTGAGCCTGGAGAATTACTTTTCTCTTTTAAATGAGAAGGCCACCA
ATGTCCCCAAAAAAA
Exact match not found. Length 408 prefix of pattern found is : GGCACCAGGGGCCAGGG
AGGCCCAATCTCAGAGCGCTGCGGGCTCTGCTGGTGATAGAAACAATATGGTGAAGTTATAAGGAGAGGGGAGGGGTGGG
GGGCGGGAAAGATTCCTTAGGATAGAAAAAAAACAATGAAGAAAGCTCCACTTGAAGACCTATGAAATTTCCGACACTAG
AACCTTTGTATATAGTTTTCGGTAACCTTCGGCAGGGGGCGCTCTGAAATTGTTGCCACTTTCCGGACAGCCTCCCATAC
TTTACCCTCTACAAAATTGTACAGTGTTTCCGGCCCTTCTCAGTTGTGGACGCTCATCACTGGCGTTATCCTTCTTGCAG
TTGGCATTTGGGGCAAGGTGAGCCTGGAGAATTACTTTTCTCTTTTAAATGAGAAGGCCACCAATGTCCCC
Time to search = 0.063ms
Please enter a pattern or enter 'q' to quit
```

Figure 3: Partial Pattern Search

Figure 4 shows the running time on searching the pattern.



```
Time to create SA = 28s
Time to convert to packed format = 2s
Please enter a pattern or enter 'q' to quit
GGCACCAGGGGCCAGGGAGGCCCAATCTCAGAGCGCTGCGGGCTCTGCTGGTGATAGAAACAATATGGTGAAGTTATAAG
GAGAGGGGAGGGGTGGGGGGCGGGAAAGATTCCTTAGGATAGAAAAAAAACAATGAAGAAAGCTCCACTTGAAGACCTAT
GAAATTTCCGACACTAGAACCTTTGTATATAGTTTTCGGTAACCTTCGGCAGGGGGCGCTCTGAAATTGTTGCCACTTTC
CGGACAGCCTCCCATACTTTACCCTCTACAAAATTGTACAGTGTTTCCGGCCCTTCTCAGTTGTGGACGCTCATCACTGG
CGTTATCCTTCTTGCAGTTGGCATTTGGGGCAAGGTGAGCCTGGAGAATTACTTTTCTCTTTTAAATGAGAAGGCCACCA
ATGTCCC
Pattern found at index 4051
Time to search = 0.039ms
Please enter a pattern or enter 'q' to quit
```

Figure 4: Full Pattern Search

4. Results

We ran our test on the data set provided by professor. The given data set was of 303 MB. We ran this data set on both Suffix array and our packed suffix array to compare the results.
Initially, we used the full data and found out that the memory used by suffix array including input data is 1.59 GB whereas our packed suffix array used 1.36 GB as shown in figure1.

After packing the input data, the memory used by the data is reduced to 75.75 MB from 303 MB.
Below figure represents the total memory used by our program which includes the suffix array and input data.

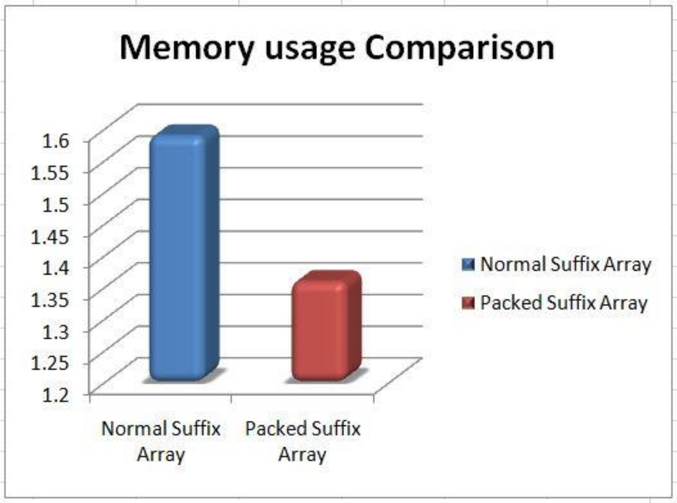


Figure 5: Memory usage comparison (in GB) for data set of size 303 MB

We then choose half of the sample to clearly distinguish the memory usage pattern among the two suffix arrays and found that for 178 MB of data the memory usage for packed suffix array was 809 MB whereas standard suffix array implementation was using 946 MB as shown in figure6.

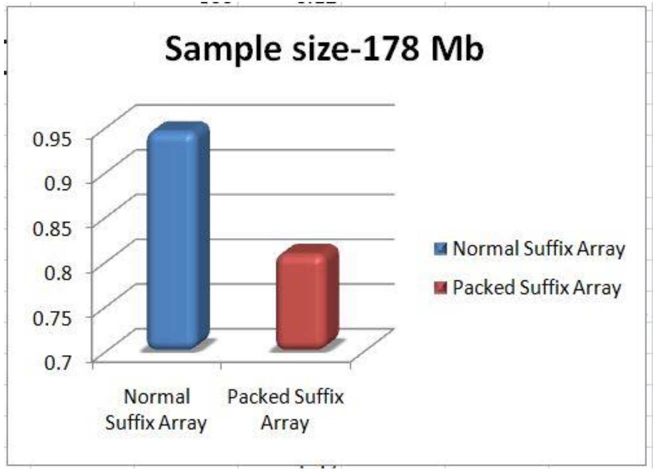
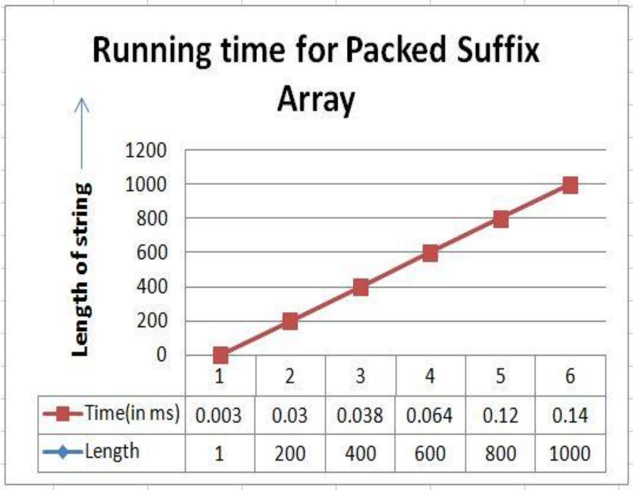


Figure 6: Memory Usage Comparison (in GB) for data set of Size 178 MB

We further tested the query time from our packed suffix array and found that querying time is linear with query size as shown in Figure7.



	1	2	3	4	5	6
Time(in ms)	0.003	0.03	0.038	0.064	0.12	0.14
Length	1	200	400	600	800	1000

Figure 7: Querying time w.r.t. query size

Our algorithm took 47 sec to create suffix array and 3 secs to pack the array for 303 MB of data.

Following the linear pattern, it took 26 secs to create the suffix array and 2 secs to pack when ran on 178 MB of data.

Acknowledgements

We are thankful to professor Rob Patro for his advice and guidance throughout the course of project. We are also grateful to Laraib Malik who was also supportive and available to help and clear our doubts.

References

Kiełbasa SM, Wan R, Sato K, et al. Adaptive seeds tame genomic sequence comparison. Genome Res 2011; 21:487-93.

Vyverman M, De Schrijver J, Criekinge WV, et al. Accurate long read mapping using enhanced suffix arrays. In: Pellegrini M, Fred A, Filipe J, Gamboa H, editors. International conference on Bioinformatics Models, Methods and Algorithms (BIOINFORMATICS 2011), Rome, Italy. Lisbon, Portugal: SciTePress; 2011. p. 102-7.

Vyverman M, De Baets B, Fack V, et al. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. Bioinformatics 2013;29(6):802-4

Ye Y, Choi J-H, Tang H.RAP Search: a fast protein similarity search tool for short reads. BMC Bioinformatics 2011; 12:159.

Ilie L, Fazayeli F, Ilie S . HiTEC: accurate error correction in high-throughput sequencing data. Bioinformatics 2011;27(3):295-302.

Schröder J, Schröder H, Puglisi SJ, et al . SHREC: a short-read error correction method. Bioinformatics 2009;25(17):2157-63.

Gonnella G, Kurtz S . Readjoiner: a fast and memory efficient string graph-based sequence assembler. BMC Bioinformatics 2012;13(1):1-19.

Hernandez D, François P, Farinelli L, et al . De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. Genome Res 2008;18(5):802-9.

Kurtz S, Narechania A, Stein JC, et al . A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. BMC Genomics 2008;9(1):1-18.

Alex. RRR – A Succinct Rank/Select Index for Bit Vectors. http://alexbowe.com/rrr/

http://bib.oxfordjournals.org/content/early/2014/01/10/bib.bbt081.full#ref-6