Optimizing DNA Sequences using Tetra-nucleotide RankList

Rasif Ajwad
Dept. of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT)
Gazipur, Bangladesh
ajwad.raw@gmail.com

Syed Nayem Hossain
Dept. of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT)
Gazipur, Bangladesh
nayem2155@gmail.com

Md. Abid Hasan
Dept. of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT)
Gazipur, Bangladesh
aabid@iut-dhaka.edu

***Abstract*— Recent advancement in the field of life science has caused the generation of massive amount of genomic data. Storing such huge amount of data require a lot of memory. However, by using efficient algorithm we can optimize the size of the dataset and save memory storage. In this paper we have proposed an algorithm which uses Tetra-nucleotide RankList for optimizing the storage requirement for storing DNA sequences in database which can be easily retrieved in a time efficient manner. Tetra-nucleotide RankList has been generated by testing several DNA sequences to confirm the uniformity of the RankList over all DNA sequences. The algorithm has been applied on the genome sequences of different bacteria and viruses and the space for storing those DNA sequences has been reduced up to 30%.**

Keywords—memory storage; tetra-nucleotide; RankList; DNA sequences

I. INTRODUCTION

DNA sequencing, the process of determining the precise order of nucleotides within a DNA molecule, has played a significant role in the field of life sciences. During the '70s, the Sanger sequencing method [1] was the most widely used sequencing method and was used for approximately 25 years. After the advent of array-based pyro-sequencing [2], there has been an exponential increase in the generation of DNA sequence data [3].

The sequence that are studied and understood by the scientists are stored in a particular type of database called sequence databases, which are composed of a large collection of computerized (digital) nucleic acid (both DNA & RNA) sequences, protein sequences and other polymer sequences stored on a computer. An example of such kind of database is GenBank [4]. It is a database of nucleotide and protein strings and was built by NCBI (**N**ational **C**enter for **B**iotechnology **I**nformation). Although started with a relatively less number of sequences, these databases are now growing at an exponential rate. According to the statistics, the size of GenBank has doubled every 15 months. [5]

These sequence databases can be searched using different algorithms. But most of them are in-memory algorithms [6-12], which have to scan the whole database for each query. As a result, these algorithms suffer from disk I/Os in case of large databases. These algorithms are also impractical where the database size grows faster than the available memory capacity because of the extensive memory requirement. Such adversities lead to a major issue of optimizing the sequence before storing them into the database in recent days.

A number of methods have been used to store the DNA sequences in databases. Dynamic Programming [13] is an approach with time and space complexity of $O(nm)$, for two strings of lengths n and m. So the matrix size needs to be $(n \times m)$, which will not be feasible in terms of both space and time for long sequences. The BLAST [14] (**B**asic **L**ocal **A**lignment **S**earch **T**echnique) is an algorithm for comparing primary biological sequence information. It emphasizes on finding the local similarity rather than the global similarity between the sequences. First it searches all database sequences for a fixed substring length 'l' for exact matching (at 'i'). And using a threshold 't', it continues the searching after the exact match at both direction, left and right, for distance more than 'i' and before '$i - l$' till exceed 't'. It stores pointer for location 'i'. So, space needed is more than the database size.

Suffix array [15] scans database strings using a window and counts repetition of all possible k-tuples. It stores the result at some vectors and then indexes those vectors at hierarchical binary tree. It runs 25 to 50 times faster than BLAST. But this technique allows false drops and index size increase linearly with k value. SST [16] scans the database by window w and map results to vector of size $4w$. The non-overlapping, hierarchical clusters are built using k-means algorithm, as any new query need to be processed against the database, using cluster mean and neglect clusters that are far away from the new query. Disadvantages of SST are the complexity of calculations, and false clustering.

The goal of this paper is to develop an algorithm where the large DNA sequences will be optimized before storing them into the databases to ensure better storage capacity. To observe the system's response time, we used DNA sequences of different lengths and compared them. We also observed the proposed algorithm by using multiple computers.

The rest of the paper is organized as follows. Section II discusses our proposed algorithm. Section III shows materials and methods. Section IV contains the experimental results and discussion and Conclusion included in Section V.

II. DESIGN OF PRPOSED ALGORITHM

DNA sequences are not random; they contain repeating sections, palindromes, and other features that could be represented by fewer bits than is required. Taking this property into consideration, we created a RankList by calculating the frequency of each of the possible tetra-nucleotides and assigning rank to them (Algorithm 1). We assigned lowest rank to the tetra-nucleotide which had the highest frequency, thus ensuring maximum optimization of a sequence. To calculate frequencies of these combinations, we used DNA sequences of different species to form a generalized RankList, which can be used to optimize any dataset to save memory storage.

Algorithm 1

Input: A training dataset (D) of genome sequence
Output: Frequency of combinations appeared in the dataset

1: Create two arrays (P and Q).One for all possible 4 length combination of A, C, G, T (P), the other for calculating the frequency value (Q).
2: Read 4 consecutive characters from dataset (D) and check them in the combination array (P).
3: When match found, increment the corresponding frequency value of the second array (Q).
4: Repeat step 2-3 until end of dataset (D).
5: The whole process is repeated for several dataset.
6: Sort the two arrays in descending order of frequency to create the RankList.

The proposed algorithm emphasizes on storing the DNA sequences using as less space as possible. To achieve that goal, first we divided the large sequence into sequence of tetra-nucleotides and replaced it with its corresponding rank (a positive integer) from the rank table (Algorithm 2). The rank table is developed by taking all possible combinations of A, T, C and G of length 4 from the training dataset showed in Table I.

III. MATERIALS AND METHODS

In order to create an optimized RankList, first we need to calculate the frequency of each possible tetra-nucleotide. We took DNA sequences of different lengths. The training dataset are obtained from different species of bacteria & viruses. These data are collected directly from the NCBI database. The list of these training dataset along with their respective NCBI reference numbers and the scientific names of the species appear in Table I. The number of occurrence of the tetra-nucleotides is first calculated and then sorted to obtain the rank for each tetra-nucleotide. The calculated frequencies of the tetra-nucleotides are shown on Table II. The corresponding ranks of all possible tetra-nucleotides are shown in the rightmost column.

Algorithm 2

Input: A training dataset (D) of DNA/genome sequence
Output: Optimized dataset of that DNA/genome sequence

1: Load the RankList (R) of all possible tetra-nucleotide sequences and their corresponding rank value.
2: Read 4 consecutive characters from the dataset (D) and check the value with RankList (R).
3: When match found, store the corresponding rank value in an arraylist (A).
4: Repeat step 2-3 until the end of sequence or as far as possible.
5: If bases of length less than 4 (1, 2 or3) are unread at last, read them and store their corresponding rank in the arraylist (A).
6: Store the arraylist (A) containing ranks of the combinations.

After obtaining the RankList using the training dataset, we then collect some experimental dataset from the NCBI database and apply our proposed algorithm (Algorithm 2) to those dataset (DNA sequences). The experimental dataset used for this purpose are shown on Table III.

TABLE I. TRAINING DATASET

Training Dataset	NCBI Reference Sequence	Data
Acetohalobium arabaticum DSM 5501 chromosome, complete genome	NC_014378.1	ATTATTA... TGTT
Acidaminococcus fermentans DSM 20731 chromosome, complete genome	NC_013740.1	AAAAAATC…ATGA
Acidimicrobium ferrooxidans DSM 10331 chromosome, complete genome	NC_013124.1	GACTCGTC… TGAT
Acidothermus cellulolyticus 11B chromosome, complete genome	NC_008578.1	GATTCCTA… CAGT
Actinobacillus succinogenes 130Z chromosome, complete genome	NC_009655.1	TTAGGAAC…AAAA
Actinobacillusn suis H91-0380 chromosome, complete genome	NC_018690.1	GTATTGAC….GTAT
Acidaminococcus intestini RyC-MR95 chromosome, complete genome	NC_016077.1	CAAAGTCA…TGCA
Acholeplasma laidlawii PG-8A chromosome, complete genome	NC_010163.1	TATTTGAT… TTAT
Acanthamoeba polyphaga mimivirus, complete genome	NC_014649.1	CGCCGGGG…TAAA
Acidianus hospitalis W1 chromosome, complete genome	NC_015518.1	AACATTTA…AAAC

TABLE II. FREQUENCY TABLE

Sequence	Frequency	Rank
AAAA	53647	1
TTTT	52989	2
ATTT	44314	3
⋮	⋮	⋮
CCTA	10398	254
TAGG	10269	255
CTAG	6679	256

TABLE III. EXPERIMENTAL DATASET

Experimental Dataset	NCBI Reference Sequence:	Data
Acaryochloris marina MBIC11017 chromosome, complete genome	NC_009925.1	AATAAATA...CCAC
Acetobacterium woodii DSM 1030 chromosome, complete genome	NC_016894.1	TTATTTGG...TGAT
Acidithiobacillus caldus SM-1 chromosome, complete genome	NC_015850.1	ATGAGTAG...CATC
Acidovorax ebreus TPSY chromosome, complete genome	NC_011992.1	TAACTCCT...TGGA
Acidovorax citrulli AAC00-1 chromosome, complete genome	NC_008752.1	ATAACCCC...GTGG
Acidovorax avenae ATCC 19860 chromosome, complete genome	NC_015138.1	TTATCACA...GCGA
Achromobacter xylosoxidans A8 chromosome, complete genome	NC_014640.1	ATGAAAGA...CGAC
Achromobacter xylosoxidans NBRC 15126 = ATCC 27061, complete genome	NC_023061.1	ATGAAAGA...CGAC
Acetobacter pasteurianus 386B, complete genome	NC_021991.1	AATGGGTA...CTAG
Acetobacter pasteurianus IFO 3283-01, complete genome	NC_013209.1	ACTGCAGG...AGAA

IV. RESULTS AND DISCUSSION

A. Obtaining Results

After applying our proposed algorithm to the DNA sequences, we find our desired output sequences, which are sequence of integer numbers (varied from 1 to 256), are separated by spaces. These space-separated integers are the ranks of the tetra-nucleotides which were split and read from the original long sequences. In the output we see these ranks of the tetra-nucleotides of the original long sequence. However, there might be some case where the original long sequence is not completely divisible by 4. In that case, the remaining subsequences will be of length less than 4 (1, 2 or 3). To manage these subsequences, all possible 1-length, 2-length and 3-length combinations of A, T, C, G are taken and added to the rank table sequentially (from 257 to 356). This ensures the conversion of the remaining subsequences to integers according to the proposed algorithm. The results obtained from the DNA sequences are shown in the Table IV.

TABLE IV. OPTIMIZED RESULTS

Original Data	Optimized Data						
	Col 1	*Col 2*	*Col 3*	*Col 4*	⋯	*Col n − 1*	*Col n*
AATAAATA… CCAC	8	8	194	122	⋯	68	178
TTATTTGG…T	16	139	196	162	⋯	53	356
ATGAGTAG…C	47	233	46	169	⋯	46	354
TAACTCCT…A	191	93	204	123	⋯	165	353
ATAACCCC….GTGG	15	237	53	12	⋯	118	165
TTATCACA …GA	16	248	116	79	⋯	7	345
ATGAAAGA …GAC	47	21	66	79	⋯	136	306
ATGAAAGA…GAC	47	21	66	79	⋯	136	306
AATGGGTA …TAG	41	215	138	187	⋯	79	323
ACTGCAGG …GAA	171	122	52	250	⋯	223	305

For example, let our experimental DNA sequence be CCTATTTTATTT. If we apply Algorithm 2 on this sequence, our output will be 254 2 3. As these ranks are space-separated, the original DNA sequence can easily be retrieved from the output sequence using the RankList (Table II). The space between each of the ranks in the output sequence ensures that each integer number (Rank) is considered individually and there is no overlap between them. As a result 254 will not be wrongly decoded as 2 or 25.

B. Performance Evaluation

To measure the performance of our proposed algorithm, we compared the size of the input file with that of the output file, which is a sequence of integer numbers (ranks). The comparison shows that for a sequence of medium length, after replacing the tetra-nucleotides with their corresponding ranks, the size of the original sequence is decreased up to 30%. The comparison results are shown in Fig. 1.

The optimized dataset can easily be decoded with the help of the RankList. As this RankList is universal for all experimental dataset, we can again replace the integer values (ranks) with their corresponding tetra-nucleotides to retrieve the original sequence.

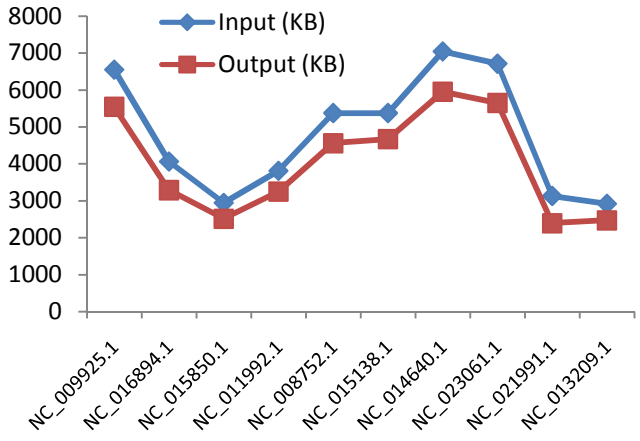

Fig. 1. Comparison between input and output files (in size)

According to our proposed algorithm, the worst case scenario for the output occurs when all the tetra-nucleotides correspond to comparatively higher ranks containing 3 digits each (>99) in the output file. In this case, for an input file of length n, the space complexity C of the output will be:

$$C = \left\lceil \left(\frac{n}{4} \right) \right\rceil \times 3$$

V. CONCLUSION

The main advantage of the proposed system is that it requires fewer bytes than the original sequence to represent each sequence in the database, thus saving I/O bandwidth and disk size. It is to be noted that, similar optimized results can be obtained by applying the proposed algorithm to RNA sequences. As all RNA sequences are combinations of A, U, C, G, we can easily modify the RankList by replacing 'T' with 'U'. In future, are interested to work with the frequent patterns in the DNA sequences, which can be used for disease detection, criminal forensics analysis and protein analysis.

REFERENCES

[1] F. Sanger, S. Nicklen and A. R. Coulson, "DNA sequencing with chain terminating inhibitors", Proc. Natl. Acad. Sci. 74:5463–5467, December 1977.

[2] M. Margulies, M. Egholm, W. E. Altman, S. Attiya, J. S. Bader, L. A. Bemben, et al., "Genome sequencing inmicrofabricated high-density picolitre reactors", Nature437: 376–380, September 2005.

[3] L. D. Stein, "The case for cloud computing in genome informatics", Genome Biology. 11:207.doi: 10.1186/gb-2010-11-5-207, May 2010.

[4] D.A. Benson, M.S. Boguski, D.J. Lipman, J. Ostell, and B.F. Ouellette,"Genbank.", Nucleic Acids Research, 26(1):1–7, 1998.

[5] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, B.A. Rapp, and D.L. Wheeler, "Genbank.", Nucleic Acids Research, January 2000.

[6] R. A. Baeza-Y ates and G. Navarro, "Faster approximate string matching", Algorithmica, 23(2):127–158, 1999.

[7] R. A. Baeza-Yates and C. H. Perleberg, "Fast and practical approximate string matching", In Combinatorial Pattern Matching, Third Annual Symposium, pages 185–192, 1992.

[8] D. Gusfield, "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology", Cambridge UniversityPress, 1 edition, January 1997.

[9] E. Myers, "An O(ND) difference algorithm and its variations", Algorithmica, pages 251–266, 1986.

[10] T.F. Smith and M.S. Waterman, "Identification of common molecular subsequences", Journal of Molecular Biology, March 1981.

[11] S. Uliel, A. Fliess, A. Amir, and R. Unger, "A simple algorithm for detecting circular permutations in proteins", Bioinformatics,15(11):930–936, 1999.

[12] S. Wu and U. Manber, "Fast text searching allo wing errors", Communications of the ACM, 35(10):83–91, 1992.

[13] T. Kahveci and A. K. Singh, "An efficient index structure for string databases", CA 93106 , 2001.

[14] T. Kahveci, A. Singh, "MAP: searching large genome databases", Pacific Symposium on Biocomputing 8:303-314, 2003.

[15] S. Muthukrishnan and S. C. Sahinalp, "Approximate nearest neighbor and sequence comparison with block operations", STOC '00 Proceedings of the thirty-second annual ACM symposium on Theory of Computing, 416-424, 2000.

[16] E. Giladi, M. G. Walker, J. Z. Wang and W. Volkmuth, "SST: An algorithm for finding nearexact sequence matches in time proportional to the logarithm of the database size", Bioinformatics 18, 873–877, 2002.

,