THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

BIOINFORMATICS
Research article

GHDNA: A HASH FUNCTION FOR DNA SEGMENT-BASED ALIGMENTS AND MOTIF SEARCH

Paul GAGNIUC[1,2] and Constantin IONESCU-TÎRGOVIŞTE[2]

[1]National Institute of Pathology "Victor Babes", Romania
[2]National Institute of Diabetes, Nutrition and Metabolic Diseases "N.C. Paulescu", Bucharest, Romania
Corresponding author: Paul GAGNIUC, E-mail: paul_gagniuc@acad.ro

Accepted December 9, 2014

Hash functions were first used in computer systems for one-way encryption of passwords since the early 1960s. Although they have a long history, hash functions remained the least-understood cryptographic primitives, much less developed than encryption techniques, and over time only several dozen have been designed in total. Moreover, many of these cryptographic functions are difficult to implement in new programming languages. This paper presents an alternative proposal to some of the widely used hashing functions, such as MD5, MD2, MD4 or SHA. GHDNA hash function converts a large and variable-sized amount of DNA data into an unique integer value in order to be used for various bioinformatic analyzes. A series of tests were conducted on artificially generated DNA sequences and biological DNA sequences from NCBI website. Experimental results show that our method is efficient in generating unique keys without collisions. We used GHDNA for repetative sequence search, motif search, segment-based aligments and database implementations.

Key words: hash functions; sequence alignment; database engine; motif search; dynamic DNA block allocation.

INTRODUCTION

Informal definitions state that cryptographic hash functions are divided in two main classes, Message Authentication Code or MAC, for a hash function that uses a secret key, and Manipulation Detection Code or MDC, that does not make use of a secret key[1,2]. MDC functions can be divided in two main categories, one-way hash functions (OWHF), concept introduced by Diffe and Hellman and collision resistant hash functions (CRHF), concept introduced by Damgard[3,4]. CRHF hash functions widely used for their speed, include MD2[5-7] and its derivatives MD5[8], SHA[9], RIPEMD[10], and HAVAL [11], N-hash [12], FFT-Hash I and II[13], Snefru[14]. Designed in the late 1980s, MD or "message digest" family of hash functions, took shape from a proprietary algorithm named MD1, which was the precursor of other MDC hashing functions, such as MD2 and MD4. Other four algorithms based on MD4 design appeared shortly after, namely MD5, SHA, and RIPEMD, while HAVAL, started as an extension of MD5 function. The attack models on MD5 by Bert den Boer and Antoon Bosselaers, led to the development of Secure Hash Algorithm (SHA), proposed by National Institute for Standards and Technology, USA, in 1992[15].

A hash function is said to be attacked when a collision pattern is found, thus a collision pattern can lead to a security key falsification in computer security, or a flaw in a database engine. Nevertheless, collisions are unavoidable for large sets that are mapped to short strings or small integers. A desirable quality for a hash function seems to be a wide domain range and a uniform distribution of hash values.

The most successful approach to find a collision seems to be a differential cryptanalysis[16]. The goal of differential cryptanalysis method is to induce changes in the input sequence which do not affect the output value.

In the late 1990s, with the exponential increase in the number of DNA sequences, and their variants, coming from different research institutes, researchers began to design and use specialized hashing functions in bioinformatics, both MAC-like, and MDC-like functions.

Other algorithms that resemble GHDNA and its features are ACMES (Advanced Content Matching Engine for Sequences)[17] and SSAHA (Sequence Search and Alignment by Hashing Algorithm)[18,19], implemented for repetitive sequence searches and genomic localization[20, 21].

The operation mode of GHDNA is different from other MAC functions - CRC-like used in sequence alignments algorithms, such as LSH-ALL-PAIRS[22]. However, this deliberate imperfection of a MAC hash function makes it ideal only for sequence alignment algorithms, and less desirable for segment-based aligments.

Cryptographic hash functions can provide integrity guarantees in that they do not rely on specific error pattern assumptions. Accordingly, GHDNA may be used for segmental alignments of whole genomes and perhaps with better results than other hash functions considering the small size of the hash key.

The GHDNA key size seems to be important, both for computer memory and the speed of search. For instance, the overall size of human genome files (FASTA format assemblies) can reach up to 2-3 Gb. A direct search for text inside these large files is highly complicated and time consuming. A more direct approach consists of using short-length hash values. Initially, these files are divided into smaller sequences (*i.e.* between 500 b and 10 Kb). For each sequence a hash key is generated and stored in an array file. A search for a certain DNA sequence or even a segmental alignment of two sequences can be accomplished by comparing a hash key to the array file (or a comparison between two array files).

GHDNA is a well-defined function that returns a single integer from an output range of 1014 posible hash values. The aim of GHDNA design was to obtain uniformly distributed output values in the function output range, for any DNA sequence. Like any other hash function of this type[23,24], GHDNA is not reversible. Consequently, the same inputs always lead to the same outputs but the relation between the similarity of two inputs is unpredictable. In this regard, two DNA sequences that differ by even a single nucleotide must always lead to different hash values.

RESULTS

In order to measure the effectiveness of the proposed hash function, we divided three groups of experiments in which we considered the output range, the uniformity of hash values and data block processing (more in the methods section). For terminology, we defined all possible values that can be returned by GHDNA function as a “domain range”. Furthermore, a “collision” is a situation that occurs when two distinct pieces of data exhibit the same hash value.

The **first experiment** established the domain range of GHDNA function at 10^{14} posible hash values. In the initial phase $PHash$ (Pre-Hash) values are calculated for 9920 random DNA sequences (Figure 2). Each sequence was generated approximately in the following manner: the set of all strings over $\sum$ of length n is denoted

$$\sum^{n}\ ,\ \sum=\{A,T,C,G\}\ ,\ \sum^{*}=\bigcup_{n\in N}\sum^{n}$$

in which, for our experiments, n =1000, corresponding to each point on the x-axis.

In order to avoid a congestion on the y-axis (Figures 2-5), we generated only 10 random samples for each *n* (ie. similar to the first steps of Kleene closure applied to set of characters[25-27]).

Comparative Analysis

A comparative speed analysis conducted against other three hashing functions, MD5, SHA1 and SHA256, showed substantial qualities of GHDNA function, namely the speed and the output size (Figure 1).

In order to obtain a real and balanced result, all functions have been implemented in the same programming language (Visual Basic) and were tested on a computer featuring a 2.8 GHz processor. Each hash function generated 1000 hash values for the same sequence (a short DNA sequence of 20b), after wich the response time was measured (Figure 1).

Repeated generation of hash values for the same DNA sequence proved to be a balanced test, allowing for an accurate measurement of processing time for every function separately (Figure 1). The processing time for each function can be reduced even further by using other programming languages such as assembly or C++. Nevertheless, the differences between measurements remain constant. As shown in Figure 1, GHDNA function produces the smallest key (14 digits) but the speed performance is comparable to SHA256 and SHA512.

Collision analysis

Initially, $PHash$ values [6] are not equally distributed in the output range of the function at this stage (Figure 2). In order to obtain equally distributed hash values we have used a technique whose name is chosen by the type of string operation performed, namely *digit shift*.

Digit shift method consists in transforming an integer number into a string data type, and thus applying a string concatenation. A reverse function that transforms a string into an integer can provide the new hash value for further processing (this method is described in the methods section). In **the second experiment** we tested if GHDNA hash values are uniformly distributed after *digit shift* method was applied.

As shown in Figure 3, *digit shift* method ensures the uniformity of hash values over the output range of the function. However, after applying this method, digit number eight becomes constant, and a domain range stabilization tendency appears (Figure 2). During this stage, we concluded that the constancy of digit number eight produces clusters of hash values, which cover approximately 1,000,000 hash values at every 10,000,000 possible hash values. In other words, if digit eight is equal to two, then the function output should be only between 2,000,000 and 2,999,999 real hash values at every 10,000,000 from the 10^{14} scale.

In order to avoid a domain range stabilization tendency, we apply a method by which the digit number eight is replaced from the hash value with another one-digit number (*i.e.* 0 to 9). The value of the new digit is calculated through modulo operation, $L \bmod 10$, where L represents the length of the input sequence. The expression $L \bmod 10$ can ensure a result between 0 and 9. The name for the second method is chosen due to a sequence length dependence, namely *digit uncertainty*, as we can not know which will be the digit value without knowing the length of the input sequence. If *digit shift* method ensures the uniformity over the output range of the function, *digit uncertainty* method ensures the avoidance of collisions by making the hash values even more randomized[28].

In the **third set of experiments**, we tested whether GHDNA function can work with DNA data blocks in order to avoid collisions even more. GHDNA function can not receive input sequences less than 3 nucleotides, and a fixed block division may lead to a non-equivalent splitting of the input sequence. For instance, we can meet a particular case in which, for predetermined fixed size DNA blocks, the last DNA block may contain less than three nucleotides. In order to avoid such situations we implemented a dynamic DNA block allocation, in which the length of DNA blocks is calculated in advance according to the length of the input sequence.

Avalanche test

Our approach on differential cryptanalysis consisted of an avalanche effect measurement. The avalanche test was described by Horst Feistel[29] for the first time in 1973 and measures the output changes of a cryptographic function. Any change in the input area should have a drastic change in the output value in order to avoid a predictable pattern.

Our implementation was based on a progressive hashing of eighty DNA sequences which differ from each other by a single nucleotide (eighty progressive insertions or replacements). Next, the results have been plotted on a graph (Figure 4) which shows that the hash values exhibit a random distribution without collisions between similar inputs.

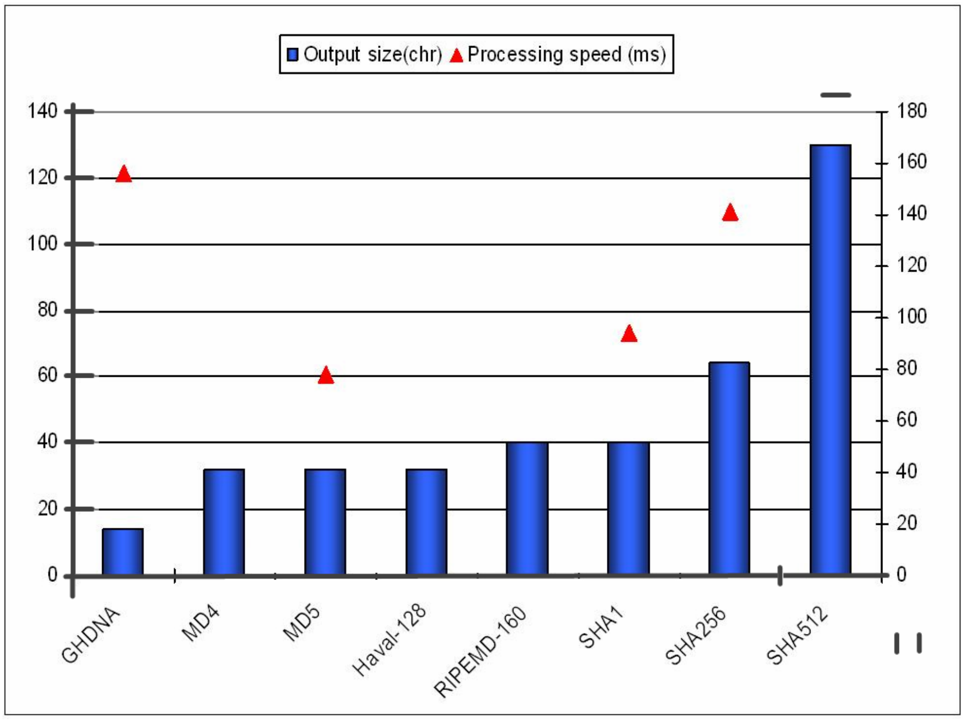

Figure 1. Comparisons made between known hashing functions. Red triangles show the response time for GHDNA, MD5, SHA1, SHA256 and blue bars show the length of hash values for GHDNA, MD4, MD5, Haval-128, RIPEMD-160, SHA1, SHA256 and SHA512.

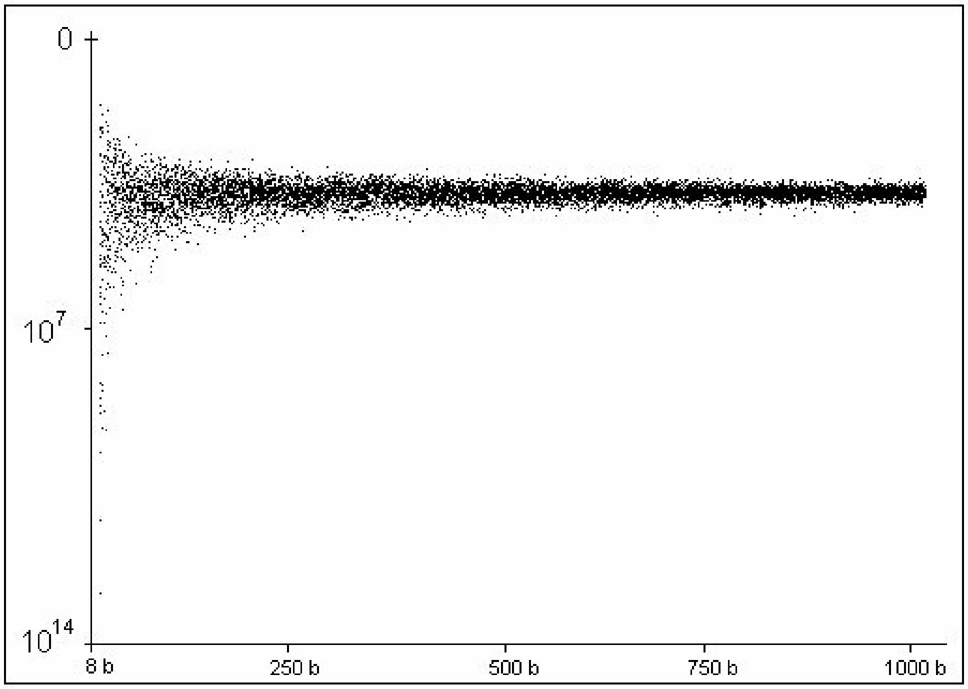

Figure 2. *PHash* values distribution. *PHash* values represent the first step for calculating the final hash value of GHDNA function. Each point in the figure represents a *PHash* value (9920 points in total). On x-axis we represent a gradual increase in length for the artificial generated sequences, on y-axis we represent the output values.

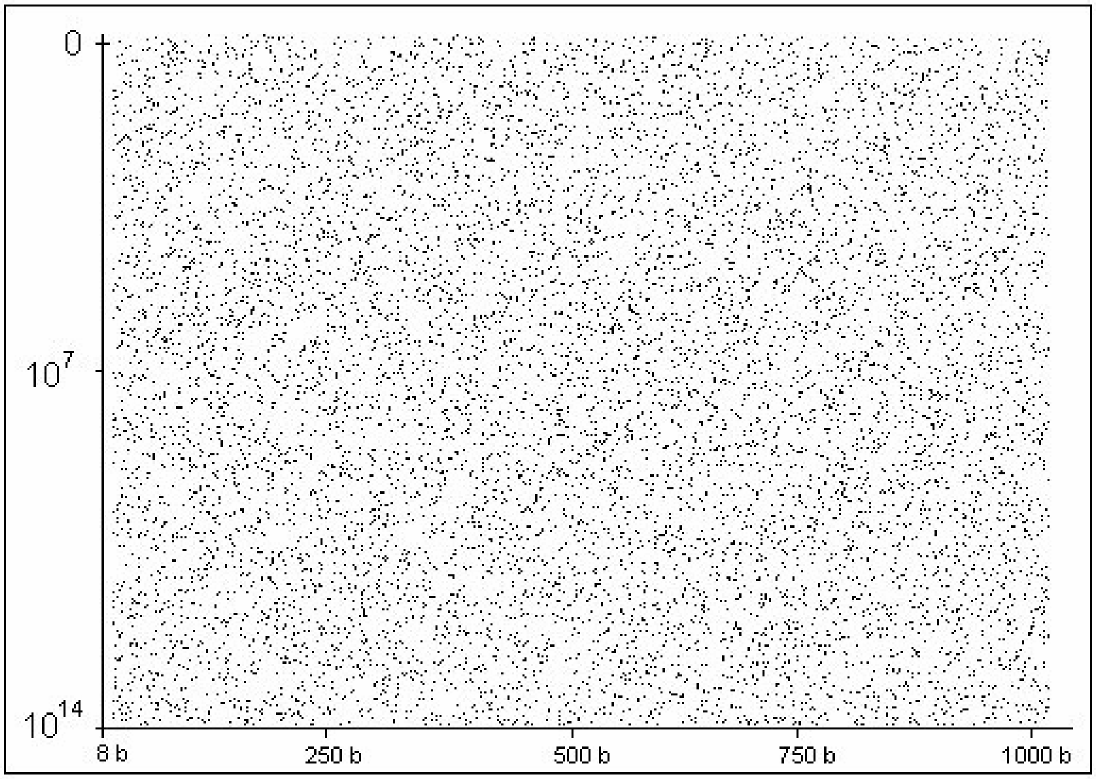

Figure 3. GHDNA domain range of hash values after *digit shift*. X-axis, Y-axis and each of the 9920 points have the same meaning as they have in Figure 2. Here we can observe the uniformity of hash values over the output range of the function after *digit shift* method was applied.

Initial avalanche tests also showed that GHDNA function operates within normal parameters only if an input sequence (of any size) exhibits an Index of Coincidence below 99%, *i.e.* sequence "AAAAAAAA" will generate an IC of 100%, while a sequence "AAATAAAA" will generate an IC of 81.87%.

General approach towards the Index of Coincidence, as described by William F. Friedman[30] in 1935, for two aligned texts, is

$$IC = \frac{\sum_{i=1}^{N}[A_i = B_i]}{N/C}$$

where sequences A and B have the same length N. Only if an A_i nucleotide from sequence A matches the B_i correspondent from sequence B, then $\sum$ is incremented by 1.

```
Function KIC(A)
N = lenght(A) - 1
for u = 1 to N
        B = A[u + 1] … A[N]
          for i = 1 to length(B)
                If A[i]= B[i]  then C = C + 1
              next i
            T = T + (C / lenght (B) × 100)
            C = 0
          next u
  IC = Round((T / N), 2)
  end function
```

With small changes, the same method for measuring the Index of Coincidence was applied for only one sequence, in which the sequence was actually compared with itself, as shown above in the source code implementation for Visual Basic family of languages[31].

Speed test

The speed test implementation for GHDNA function consisted of three modules: a random sequence generator, the algorithm for GHDNA function, and the main loop which progressively increases the input sequence length and measures the response time of the function.

Speed tests were made on a 2.8 GHz processor, both for GHDNA function in a linear fashion and for GHDNA function based on dynamic DNA block allocation. Without dynamic DNA block allocation method, GHDNA function processed on average 100 Kb/250 ms (Figure 5). For DNA blocks with a minimum length of 300 nucleotides, dynamic DNA block allocation method used about 125 ms at every 100 Kb.

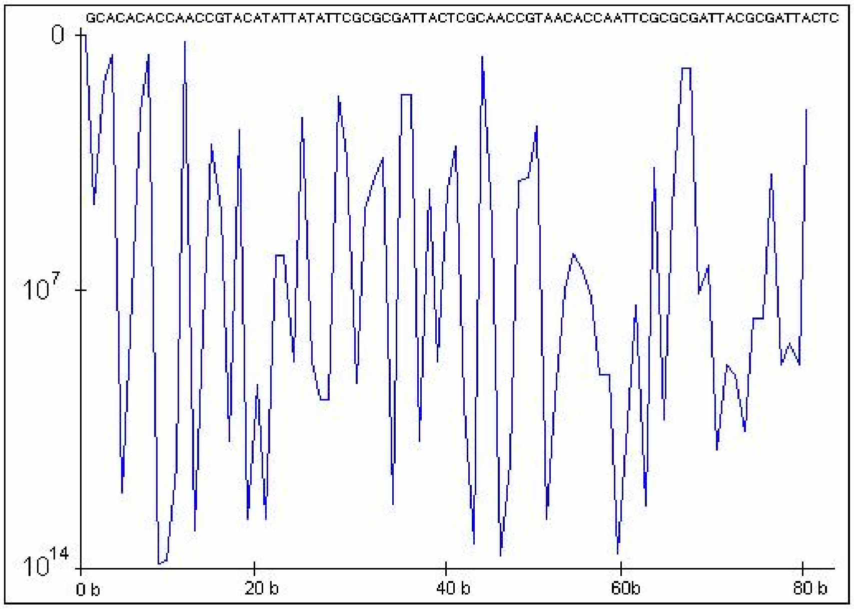

Figure 4. GHDNA avalanche test. In silico, thymine nucleotide is gradually inserted into a random DNA sequence, from nucleotide number one to nucleotide number eighty. On X-axis we represent thymine position, on Y-axis we represent GHDNA output values. Red bars show a random positioning of hash values.

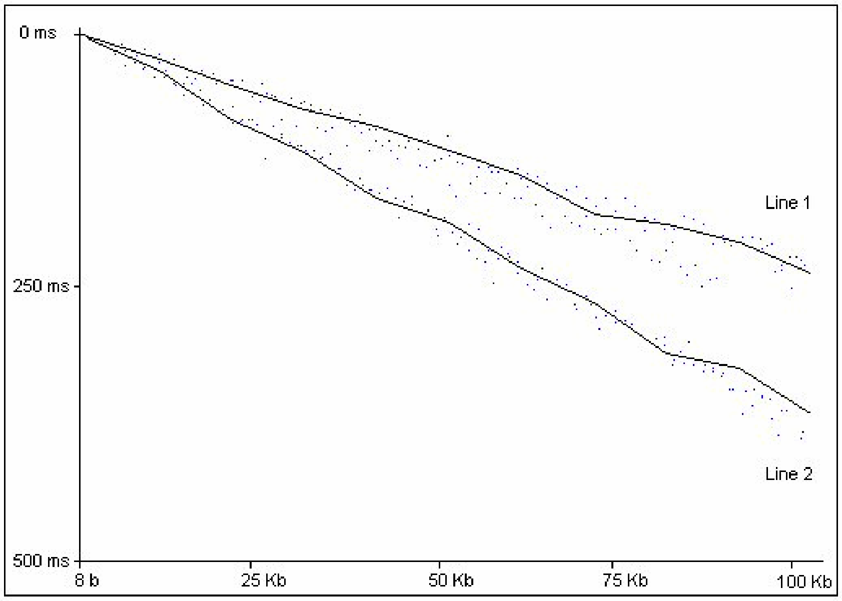

Figure 5. GHDNA processing time. On the x-axis we represent the DNA sequence length (from 1 to 100 Kb), on Y-axis we represent the response time in milliseconds, from 0 ms to 500 ms. Line 1 - shows the response time for GHDNA function without the use of dynamic DNA block allocation method. Line 2 - shows the response time for GHDNA function based on dynamic DNA block allocation.

GHDNA function takes into account all the elements from the input sequence. Nevertheless, as can be observed in equation (3) and (5), the algorithm optimization reaches approximately 50% since the function performs only $(L/2)$ cycles.

METHODS

A vast majority of hash functions are deterministic procedures that take arbitrary blocks of data and return strings of fixed lengths. GHDNA function can operate in two modes. In a first version, makes a continuous calculation of the elements from the input sequence, thereby gaining speed ahead of other types of hash functions. In a second version, which represents a continuation of the first, we use a new method which we call "dynamic DNA block allocation". Although this method takes longer to compute is intended to provide a collision free result.

Notation

Let f be a function, Let L be the length of the input sequence. We call any string made of L characters from a set $\{A,T,C,G\}$ a $L-string$. Letting $S=N_1\ldots N_L$, ($N_i \in \{A,T,C,G\}$, $i=1,\ldots,L$), be a $L-string$. In terms of computer science, we will redefine $\sum$ as a specialized construct for iterating a specific number of times, often called in programming languages a "for each loop". Therefore, considering $\sum$ an overloaded symbol, we declare variables i and u as iterators. Usually, C variable contains an irrational number, representing the preliminary value for building the final hash and is directly used to calculate equation (6). The result of equation (6) is an integer which we will call a "*PHash*" value.

Implementation

A numerical sequence representation of DNA sequences is introduced. There exists a one-to-one correspondence between a DNA sequence and its numerical sequence representation as Yu et al. proposed[32]. We provide a numeric value to each nucleotide molecule as follows, *Adenine* is associated with the number *3*, *Thymine* is associated with the number *5*, *Cytosine* is associated with the number *7*, *Guanine* is associated with the number *11*. The preliminary calculation of C value, for each 3-tuple in the DNA input sequence, is made according to the numeric values associated to the next two elements in front of the first element. The final result for C variable is made by summing all the results from each 3-tuples calculation, as can be seen in the first part of expression (5). Before continuing, we first define the association of values with the elements from the set, using a function f

$$f=\begin{cases} A \mapsto 3 \\ T \mapsto 5 \\ C \mapsto 7 \\ G \mapsto 11 \end{cases} \tag{1}$$

Function f returns the value of the non-numerical element in the DNA sequence. The value of N_i is calculated according to N_{i+1} and N_{i+2} (Figure 6). Consequently, the minimum length (n≥3) of an input sequence is dictated by the 3-tuple computation method. Equation (2) ensures that t is the largest integer less than or equal to L that is divisible by 2. Variables β and t are particularly important because they provide a calculation on 3-tuples, step two, as shown in Figure 6 and equation (5). If L is divisible by two, then t will be equal to L and the equation (4) will have zero iterations, ie. for a DNA sequence of 32 nucleotides, $L=32$, $(L \bmod 2)$ equals 0, and $(L-0)$ equals 32. We know now that $(L-(L \bmod 2))$ equals also 32, which is divisible by two.

To reach a two step calculation, we divide 32 by 2, which will reveal the final value of $\beta = 16-1=15$, therefore in equation (5) we have 15 iterations.

$$t=(L-(L \bmod 2)) \tag{2}$$

$$\beta=\left(\frac{t}{2}\right)-1 \tag{3}$$

$$R=\sum_{u=t}^{L}\frac{(f(N_u)-(L-\sqrt{u}))}{f(N_u)} \tag{4}$$

The relationship (4) has the task of calculating the elements which are not in the range of β variable. Variable R is the smallest time-consuming variable, R can only make one or two iterations, because $(L \bmod 2)$ can not take a value greater than 1, for any given number. Variable R is valuable for generating hash values for sequences of the same size, that exceed the range of β variable. Relation $(L-t)$ will provide the number of iterations for R.

$$C=\left(\sum_{i=1}^{\beta}\frac{(f(N_{2i-1})-\sqrt{(i \bmod 2)+1})\times(f(N_{2i})+\sqrt{(i \bmod 3)+1})}{f(N_{2i+1})}\right)-R \tag{5}$$

The C variable is the core of GHDNA function, is designed to generate a number according to the order in which elements are arranged in the DNA sequence, ie. if we consider the sequence "AGTTAGGACG" shown in Figure 6, whose length is equal to 10, we can illustrate step by step the method of calculation for equation (5). We see in equation (5) that i can take a value from 1 up to β. To find β, we must first solve equation (3).

$$\beta = \left(\frac{(L-(L \bmod 2))}{2}\right) - 1$$

$$\beta = \left(\frac{(10-(10 \bmod 2))}{2}\right) - 1$$

$$\beta = 4$$

Knowing the value of β and the values taken by i = 1 to 4, we can show how equation (5) unfolds, by following expressions below.

$$\begin{cases} N[2i-1] = N[2\times 1-1] = N[1] & i=1 \\ N[2i] = N[2\times 1] = N[2] & i=1 \\ N[2i+1] = N[2\times 1+1] = N[3] & i=1 \end{cases}$$

$$\begin{cases} N[2i-1] = N[2\times 2-1] = N[3] & i=2 \\ N[2i] = N[2\times 2] = N[4] & i=2 \\ N[2i+1] = N[2\times 2+1] = N[5] & i=2 \end{cases}$$

$$\begin{cases} N[2i-1] = N[2\times 3-1] = N[5] & i=3 \\ N[2i] = N[2\times 3] = N[6] & i=3 \\ N[2i+1] = N[2\times 3+1] = N[7] & i=3 \end{cases}$$

$$\begin{cases} N[2i-1] = N[2\times 4-1] = N[7] & i=4 \\ N[2i] = N[2\times 4] = N[8] & i=4 \\ N[2i+1] = N[2\times 4+1] = N[9] & i=4 \end{cases}$$

At each iteration of i, we obtain three values that identify a 3-tuple in the DNA sequence. Each of these three values are identifiers for N in the DNA sequence. When the corresponding non-numerical element from the DNA sequence is passing through function f, at each iteration of i, is returning the corresponding number of this non-numerical element (section **c**. from Figure 6). A calculation on more than 3-tuples slows down the function. Using less than 3-tuples prevents equation (5) from generating a collision-free identifier for the hash value.

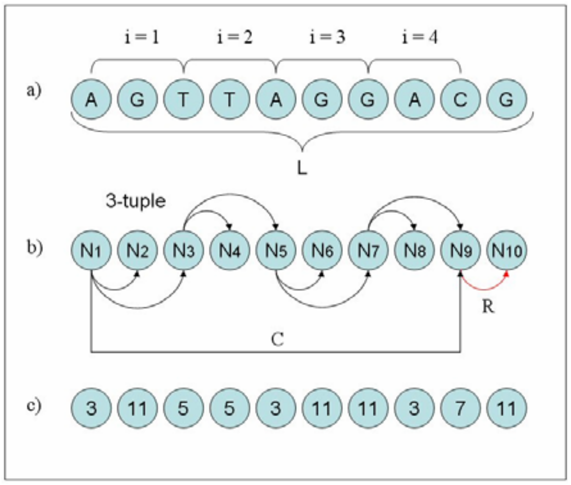

Figure 6. GHDNA calculation method. a. shows the analyzed DNA sequence and the location of interest for each iteration of i in equation (5). b. describes a 3-tuples calculation, in step two. c. shows the prime numbers associated by function f with each nucleotide molecule.

After processing C variable, the $PHash$ value that produces the data shown in Figure 2 is

$$PHash = \left[\left(\frac{L}{C}\right)\times 10^{14}\right] - L \qquad (6)$$

In the equation above (6), the bracketed term stands for a rounding function, negative L stands for an error correction variable and (L/C) represents an identifier value for the input sequence.

String operations

As shown in Figure 2, $PHash$ values are not evenly distributed at this stage and must be processed by two other methods to achieve uniformity. For further processing we convert $PHash$ integers into a string data type in order to perform *digit shift* and *digit uncertainty* methods through string operations. In *digit shift* method, we permute the first seven digits with the last seven digits, ie. after *digit shift* the $PHash$ value "24583480330876" becomes "03308762458348".

After *digit shift* method was applied, digit number eight tends to stabilize more quickly than other digits. By using *digit uncertainty* method we replace this digit from the hash value. For instance, the hash value for "G G A T A A T A G T G G G G A A G G G A" sequence will be 03308762458348. After digit replacement the new hash value will be 0330876X458348, where "X" is equal to $(L \bmod 10)$. In the case above, L is equal to 20, therefore the value of the new digit is zero. At choice, the new string can be converted into an integer data type, or can be used directly as a string data type. However, if converted to an integer, the constant number of digits can not be guaranteed by the GHDNA function, due to *digit shift* implications.

With gradual increase of the sequence length, eventually all digits tend to stabilize. For increasingly large amounts of information, *digit uncertainty* method gradually loses the properties. However, is the last method to stand even after digit stabilization.

Block calculus

As we previously specified, GHDNA function performs a continuous calculation for the entire input sequence. To avoid digit stabilization, as mentioned in the previous chapter, a calculation on data blocks[33,34] will offer a series of hash values, which can be reduced to a single hash value.

We can not choose fixed data blocks, ie. fixed data blocks of 16-tuples for instance, will split a DNA sequence of 64 nucleotides in 4 blocks of data. GHDNA function will generate a hash value for each block of data. If this DNA sequence would contain 65 nucleotides, fixed data blocks of 16-tuples will split this new DNA sequence in five data blocks. The last block of data would contain only a single nucleotide, which makes it incompatible with the 3-tuple calculation from GHDNA function. The solution to this problem was a dynamic DNA block allocation, where the length of DNA blocks varies from input sequence to input sequence.

To reduce two hash values, A and B, to a single one, A is positioned above B to create a 2-by-14 matrix. We resorted again to modulo operation in order to reduce each column to a single integer, lower than 10.

$$\sum_{i=1}^{14}(A_i + B_i)\bmod 10$$

For each of the fourteen columns we obtain a new number between 0 and 9, which will build the new vector B. A new hash value A, for another block of data, will be positioned above B to make a new reduction. The reduction of multiple one-block messages ends for the last block of data. Thus, collisions between hash values resulting from the reduction of data blocks are negligible in the final hash value.

Dynamic DNA block allocation

GHDNA function can not receive input sequences smaller than 3 nucleotides. For fixed data blocks, we can meet a particular case in which, sequences are not divided exactly in fixed blocks. Moreover, the number of nucleotides that remain after this division may be less than three, which is not desirable. Notwithstanding the notation used so far, first we ask a "Block Alocation" function to search for a remainder t, larger than three, from the division of L by a variable. Expression $(L-t)$ ensures a number divisible by 2, thus avoiding a prime number. If t variable is greater than three and r is a number divisible by two, t and r variables meet the imposed conditions, allowing a subsequent search for a number m, greater than ten, which divides r into an integer.

```
    Function Block_Alocation(ByVal L As Variant)
As Integer
    Dim a, t, b, m As Integer

     a = 1
     t = 1
     b = 1
     m = 10

    Do Until t > 3 And v = 0
     a = a + 1
     t = (L Mod a)
     r = (L - t)
     v = r Mod 2
    Loop

    Do Until b = 0 Or m >= 999
     m = m + 1
     b = r Mod m
    Loop

    Block_Alocation = m
    End Function
```

Above we show the source code of Block Alocation function, syntactically compatible with VBA, VBScript, Visual Basic 4,5,6, Visual Basic NET and Visual Basic 2005. Number ten is a starting point for the size of a DNA block. Once found, m will contain the length of a DNA block. In the source code for "GHDNA DATA BLOCK" function, the initial hash value is "12345678912345". This initial hash value can be any 14 digit number, converted into a string data type.

```
    Function GHDNA(ByVal sequence As String)
As String
        Dim correction As Variant
        Dim N(1 To 3) As String
        Dim Prehash As Variant
        Dim hash As Variant
        Dim alfa As Variant
        Dim beta As Variant
        Dim C As Variant
        Dim x As Integer
        Dim i As Integer
        Dim u As Integer

    t = (Len(sequence) - (Len(sequence) Mod 2))
    beta = ((Len(sequence) - (Len(sequence) Mod
2)) / 2) - 1

        For i = 1 To beta
            N(1) = Mid(sequence, 2 * i - 1, 1)
            N(2) = Mid(sequence, 2 * i, 1)
        N(3) = Mid(sequence, 2 * i + 1, 1)
        C1 = (f(N(1)) - Sqr((i Mod 2) + 1))
        C2 = ((f(N(2)))) - Sqr((i Mod 3) + 1)
        C3 = f(N(3))
        C = C + ((C1 * C2) / C3)
    Next i

  For u = t To Len(sequence)

  N(1) = Mid(sequence, u, 1)
  C = C + (f(N(1)) - (Len(sequence) - Sqr(u))) /
f(N(1))

  Next u

  ID = Len(sequence) / C
  Prehash  =  Round(ID  *  10  ^  14)  -
Len(sequence)
  DS = Mid(Prehash, 8, 7) & Mid(Prehash, 1, 7)
  x = Len(sequence) Mod 10
  DU = Mid(DS, 1, 7) & x & Mid(DS, 9, 6)

  GHDNA = DU
  End Function

  Function  f(ByVal  nucleotide  As  String)  As
Integer

        If nucleotide = "A" Then f = 3
        If nucleotide = "T" Then f = 5
        If nucleotide = "C" Then f = 7
        If nucleotide = "G" Then f = 11

  End Function

  Function          GHDNA_DATA_BLOCK(ByVal
sequence _
  As String) As Variant
  Dim a, b, C As String
  Dim i, BlockSize As Long
  Dim EA, EB, u As Integer

  BlockSize = Block_Alocation(Len(sequence))
  b = "12345678912345"

  For i = 1 To Len(sequence) Step BlockSize
    a = GHDNA(Mid(sequence, i, BlockSize))
    For u = 1 To 14
        EA = Val(Mid(a, u, 1))
        EB = Val(Mid(b, u, 1))
        C = C & (Val(EA + EB) Mod 10)
    Next u
    b = C
    C = ""
  Next i

  GHDNA_DATA_BLOCK = b
  End Function
```

Above we show the source code of GHDNA function, and its extension, "GHDNA DATA BLOCK" function, syntactically compatible with VBA, VBScript, Visual Basic 4,5,6, Visual Basic NET and Visual Basic 2005. Actual source code implementation of GHDNA function is made in the integrated design environment of Visual Basic 6.0.

DISCUSSION

Real DNA sequences are far from random. Nevertheless, GHDNA function treats any DNA sequence without discrimination. Therefore using random DNA sequences for testing can be sufficient. Given the small number of component letters for a DNA sequence (*i.e.* A,T,C,G), the probability of collision is small. A regular cryptographic function must digest normal text which is composed of 255 possible character types. In conclusion, the set of all possible texts arising from the combination of 255 characters, provides a higher probability of collision while the set of all texts derived from the combination of only 4 characters is much smaller. Therefore, GHDNA hash key can be smaller than that of other functions, without any collisions.

We also tested some simpler methods (*i.e.* MOD or XOR operators). For instance, we performed a modulo by prime operation. Nevertheless, we concluded that such methods generate frequent collisions and can not be reliable.

Digit shift and *digit uncertainty* methods may be modified for a more optimal domain range distribution. However, using *digit shift* or *digit uncertainty* methods for an alternative hash function will certainly exhibit an undesirable outcome in most cases.

The 3-tuple limitation of GHDNA function may be overcome by padding the sequence with three or more fixed characters (*i.e.* "AAA"). However, by padding different characters for two GHDNA implementations, we obtain different hash values. In order to preserve a compatibility between different GHDNA implementations we used dynamic DNA block allocation method.

Initially, during design the "avalanche test" showed that for two DNA sequences containing only one difference, a constant length and a large Index of Coincidence, GHDNA function sometimes generated the same hash value, presenting a subtle pattern. Our solution consisted of two additional expressions introduced inside equation (5), namely $\sqrt{(i \bmod 2)+1}$ and $\sqrt{(i \bmod 3)+1}$. These two expressions can provide a small number, which does not exceed the lowest number provided by the f function for a nucleotide molecule, as for example is *Adenine*, whose value is three. The value of $i \bmod 2$ will generate for the entire process (6), at each iteration of i, a repeated sequence of numbers 1,0,1,0, … ,1,0, whereas the value of $i \bmod 3$ will provide a repeated sequence of numbers such as 1,2,0,1,2,0, … ,1,2,0. When processed by GHDNA function, this imbalance ensures a unique value for a DNA sequence. Furthermore, to highlight the relationship between sequence length(L), t, β and the number of iterations for R, we show the following source code implementation for Visual Basic family of languages.

```
Function Relationship()
 For L = 1 To 100
  t = (L - (L Mod 2))
  b = (t / 2) - 1
  Print "L=" & L & ", t=" & t & ", b=" & b
  Print "R iterations" & L - t
 Next L
End Function
```

The algorithm presented above reveals the relationship between t, β, L and the number of iterations of R, for each number from 1 to 100.

GHDNA function can be used through two separate methods. The first method involves a direct call of GHDNA function while the second method uses an indirect call of “GHDNA DATA BLOCK” function. Nevertheless, “GHDNA DATA BLOCK” function is the final deliverable mode for GHDNA algorithm.

Database implementation

While the details will differ in other database engines, the fundamental principles are usually unanimous regardless of the configuration used. Database design seems to be one of many critical parts of an application. In bioinformatics, databases often contain large amounts of information, such as DNA motifs, repeated sequences, palindrome sequences and other DNA sequences with important biological role.

Hashing is an effective method for accessing data using a key value. For instance, we considered a database engine that uses text files to store data. To provide an example, we used a list of 814 hexamers identified by FGA [35] for acceptor splice-site prediction and donor splice-site prediction. Our simple database example is composed of a series of records, which are seperated by double colon delimeters, whose structure consists in a hash value followed by a DNA sequence from which the hash value was originaly calculated. While entering new sequences in the database, repetative sequences will have the same hash value, and they will be mapped to a single record, for a fast and optimal search.

In our implementation, a search for hexamers within a DNA sequence (*i.e.* whole genome files), is made by using sliding windows across the DNA sequence. The sliding window step may vary. However, for segmental alignment of sequences the sliding window step should be equal to the sliding window size.

GHDNA function calculates a hash value for each sliding window, as shown in Figure 7. Multiple sliding windows with the same content, will have the same hash value. Each content derived from the sliding window, is then passed through a restriction filter, to avoid searching the same hash value inside the database. If the hash value has not been searched previously, then passes the restriction filter, and the search takes place in memory on a array structure, filled in advance with hash values from the database text file. The database engine can easily be changed for searching repeated sequences. However, in the current implementation, repeated sequences are filtered.

CONCLUSIONS

There is a fifty years history of cryptographic hash functions in which only a few have been built. GHDNA is a different and reliable approach, dedicated for hashing DNA sequences. In this manuscript we described the operating mode and the optimal parameters of GHDNA function. In order to avoid collisions, we established the domain range at 1014 hash values and we obtained a uniform distribution of these values across the domain range of the function. Although written in a high-level language, the algorithm for GHDNA obtained a good speed and low memory requirements, due to its relative simplicity and dedicated purpose. The issue in using a more complex hashing algorithm over a simpler and dedicated one remains the hash size and the additional time it takes to compute.

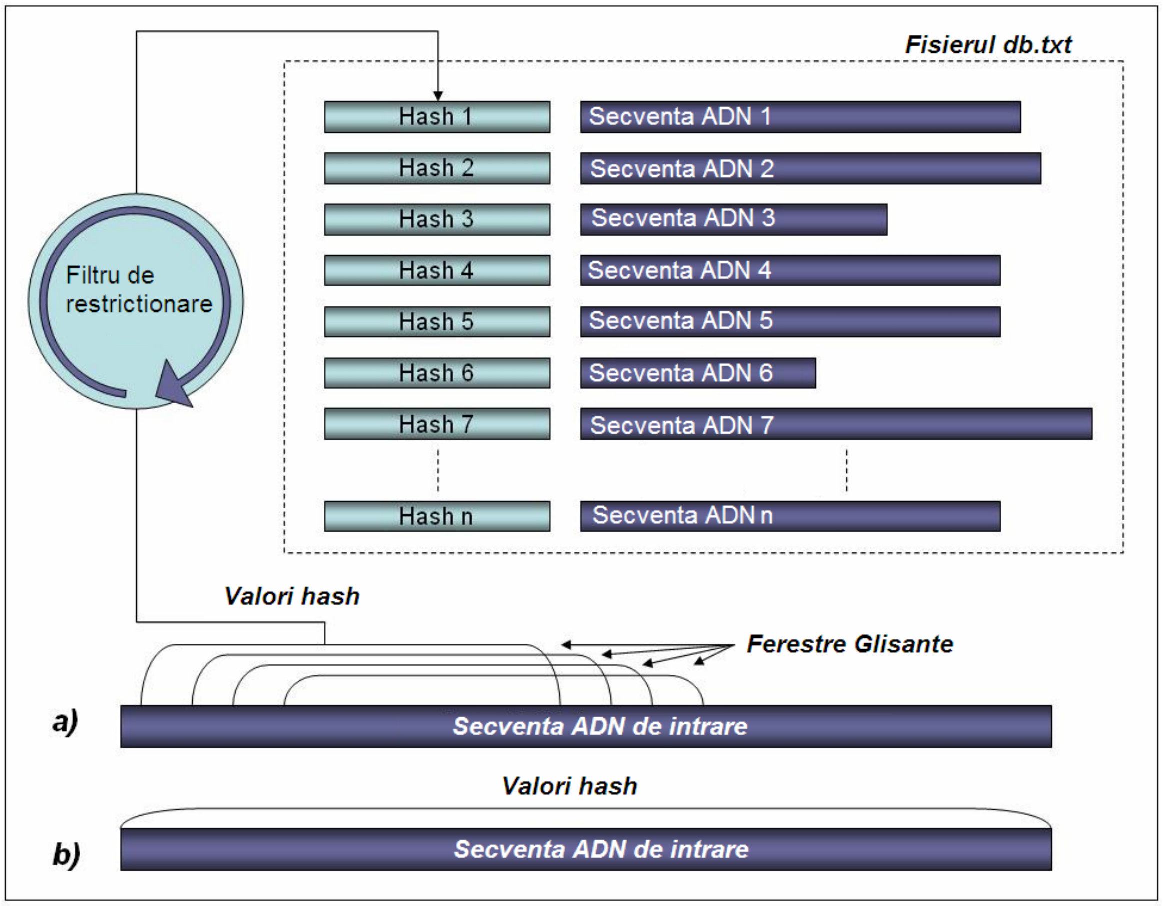

Figure 7. Database structure. A database structure composed of (a) records stored in a text file and (b) the restriction filter linked to GHDNA function. Section c. refers to a motif search inside a DNA sequence using sliding windows and section d. refers to a DNA motif search by direct match.

ACKNOWLEDGEMENT

This work was supported by the postdoctoral program CERO – Profil De Carieră: Cercetător Român, project number POSDRU/159/1.5/S/135760.

REFERENCES

1. Bart Preneel, The State of Cryptographic Hash Functions, Appeared in Lectures on Data Security, Lecture Notes in Computer Science 1561, I. Damgard (ed.), pp. 158-182, 1999.
2. Bart Preneel, Design principles for dedicated hash functions, Appeared in Fast Software Encryption, FSE 1993, Lecture Notes in Computer Science 809, R. J. Anderson (ed.), Springer-Verlag, pp. 71-83, 1993.
3. W. Diffe, M.E. Hellman, New directions in cryptography, IEEE Trans. on Information Theory, Vol. IT-22, No. 6, pp. 644-654, 1976.
4. I.B. Damgard, Collision free hash functions and public key signature schemes, Advances in Cryptology, Proceedings Eurocrypt'87, LNCS 304, D. Chaum and W.L. Price, Eds., Springer-Verlag, 1988, pp. 203-216.25.
5. B.S. Kaliski, The MD2 Message-Digest algorithm, Request for Comments (RFC) 1319, Internet Activities Board, Internet Privacy Task Force, April 1992.
6. R.L. Rivest, The MD4 message digest algorithm, Advances in Cryptology, Proc. Crypto’90, LNCS 537, S. Vanstone, Ed., Springer-Verlag, 1991, pp. 303–311.
7. R.L. Rivest, The MD4 message-digest algorithm, Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.
8. R.L. Rivest, The MD5 message digest algorithm, Presented at the rump session of Crypto’91.
9. Secure Hash Standard, Federal Information Processing Standard (FIPS), Draft, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., January 31, 1992.
10. Race Integrity Primitives Evaluation (RIPE): final report, RACE 1040, 1993.
11. Y. Zheng, J. Pieprzyk, and J. Seberry, HAVAL — a one-way hashing algorithm with variable length output, Advances in Cryptology, Proc. Auscrypt’92, LNCS, Springer-Verlag, to appear.
12. S. Miyaguchi, K. Ohta, and M. Iwata, 128-bit hash function (N-hash), Proc. Securicom 1990, pp. 127–137.
13. C.P. Schnorr, FFT-Hash II, efficient cryptographic hashing, Advances in Cryptology, Proc. Eurocrypt’92,

LNCS 658, R.A. Rueppel, Ed., Springer-Verlag, 1993, pp. 45–54.

14. R. Merkle, A fast software one-way hash function, Journal of Cryptology, Vol. 3, No. 1, 1990, pp. 43–58.
15. B. den Boer and A. Bosselaers, Collisions for the compression function of MD5, Advances in Cryptology, Proceedings Eurocrypt'93, LNCS 765, T. Helleseth, Ed., Springer-Verlag, 1994, pp. 293-304.
16. Bart PRENEEL, Analysis and Design of, Cryptographic Hash Functions, PhD Thesis, pp. 40-51, 2003.
17. Reneker J, Shyu CR, Zeng P, Polacco JC, Gassmann W: ACMES: fast multiple-genome searches for short repeat sequences with concurrent cross-species information retrieval, Nucleic Acids Res 2004, 32:W649-W653.
18. Ning,Z., Cox,A.J. and Mullikin,J.C. (2001), SSAHA: a fast search method for large DNA databases, Genome Res., 11, 1725–1729.
19. Ning Z, Spooner W, Spargo A, Leonard S, Rae M, Cox A, The SSAHA trace server, Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference (CSB 2004) 2004:544-545.
20. Reneker J, Shyu CR: Refined repetitive sequence searches utilizing a fast hash function and cross species information retrievals. BMC Bioinformatics 2005, 3:111.
21. Courtney A Harper *et al.*, Comparison of methods for genomic localization of gene trap sequences, BMC Genomics 2006, 7:236 doi:10.1186/1471-2164-7-236.
22. Jeremy Buhler, Efficient large-scale sequence comparison by locality-sensitive hashing, BIOINFORMATICS, Vol. 17 no. 5 2001, Pages 419–428.
23. Knuth DE, The Art of Computer Programming. Volume 3:Sorting and Searching. 2nd edition., Addison-Wesley; 1998:800.
24. Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest, Introduction to Algorithms. McGraw-Hill, New York., 2001.
25. J. Agrawal, Y. Diao, *et al.*, On supporting kleene closure over event streams, Technical Report 07-03, UMass Amherst, 2007.
26. Kozen, D., A completeness theorem for Kleene algebras and the algebra of regular events, Information and Computation 110(2) (1994) 366–390.
27. Desharnais, J., M¨oller, B., Struth, G., Modal Kleene algebra and applications, A survey. Technical Report DIUL-RR-0401, D´epartement d'informatique et de g´enie logiciel, Universit´e Laval, D-86135 Augsburg (2004).
28. Juan Soto and Lawrence Bassham, Randomness Testing of the Advanced Encryption Standard Finalist Candidates, National Institute of Standards and Technology, March 2000.
29. Horst Feistel, Cryptography and Computer Privacy, Scientific American, Vol. 228, No. 5, 1973.
30. Friedman William F., The index of coincidence and its applications in cryptanalysis, Published in 1935, G.P.O. (Washington) .
31. David I. Schneider, Computer Programming Concepts and Visual Basic, ISBN 0–536–60446–0.
32. Yu, Zuguo and Anh, Vo V. and Zhou, Yu and Zhou, Li-Qian, Numerical Sequence Representation of DNA Sequences and Methods To Distinguish Coding And Non-Coding Sequences in a Complete Genome. In: 11th World Multi-Conference on Systemics, Cybernetics and Informatics: WMSCI 2007, 8-11 July 2007, Florida, USA.
33. Jason Cong *et al.*, Buffer Block Planning for Interconnect Planning and Prediction, IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 9, NO. 6, pp. 929-937, DECEMBER 2001.
34. Le Cai and Yung-Hsiang Lu, Energy Management Using Buffer Memory for Streaming Data, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 24, NO. 2, pp. 141-152, FEBRUARY 2005.
35. Rezarta Islamaj Dogan, Lise Getoor, W John Wilbur and Stephen M Mount, Features generated for computational splice-site prediction correspond to functional elements, BMC Bioinformatics 2007, 8:410 doi:10.1186/1471-2105-8-410.