THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

BIOINFORMATICS
Research article

DYNAMIC BLOCK ALLOCATION FOR BIOLOGICAL SEQUENCES

PAUL GAGNIUC[1] and CONSTANTIN IONESCU-TIRGOVISTE[2]

[1] First author affiliation Institute of Genetics, University of Bucharest, Romania
[2] National Institute of Diabetes, Nutrition and Metabolic Diseases "N.C. Paulescu", Romania
Corresponding author: P. Gagniuc, E-mail: paul_gagniuc@acad.ro

Received October 11, 2011

Dynamic Data Block Allocation (DDBA) represents a novel and flexible method for partitioning string sequences into data blocks taking into account different rules imposed by a function. We present two versions of this algorithm, namely DBFA (Double Brute Force Algorithm) and MBFA (Multi Brute Force Algorithm). A series of tests were performed, mostly related to integer numbers where blocking and block size planning was used. Comparisons made between these two versions of the algorithm have shown different allocation distributions of data blocks, both for small sequences and for very large sequences, suggesting that DBFA provides more rigid solutions than MBFA. We suggest possible applications in biology, but also a range of applications in computer science, such as storage efficiency or memory optimization. We used this algorithm in biology in order to divide DNA sequences into data blocks for further processing by a data mining function.

Key words: Dynamic block allocation, Brute force, DNA periodicity patterns, Optimal storage.

INTRODUCTION

Here we describe a method for handling biological sequences (strings) in some particular cases. In order to perform our analysis on strings we used DNA sequences. The genetic information is stored in DNA (deoxyribonucleic acid). DNA consists of two long polymers of simple units called nucleotides. The four bases found in DNA are adenine (abbreviated A), cytosine (C), guanine (G) and thymine (T)[1]. In 2001, the human genome sequence was published and the order of these molecules was available as text[2]. A vast majority of algorithms used in bioinformatics are designed for text data mining[3]. Some functions used in bioinformatics or in computational biology accept structured data types of a certain length. For instance, every analysis tool uses a specific set of file formats[4]. FASTA format is frequently used. This format can encompass several types of sequences, such as DNA/RNA (ribonucleic acid) sequences or amino acid sequences. Files using FASTA format may contain up to 3Gb - 6Gb of information. In order to process this information, first, one must extract smaller sequences (data blocks). For *in silico* DNA-related analysis, data blocks are string data type sequences having a nominal length (a block size). DNA sequences structured in this manner are said to be blocked (*i.e.* GenBank format[5]). Blocked data are commonly read by a function or procedure a whole block at a time. For example, we consider a function which has two parameters, a minimum input length (n) and a maximum input length (m). Therefore, the alleged function accepts only data blocks containing a number of nucleotides larger or equal to n and lower than or equal to m. Nevertheless, the chances for a number of nucleotides larger than n in the last data block depend on the total length of the sequence (length-dependent). Our method ensures this minimum input limitation (minimum content) for the last data block by planning in advance the size of data blocks. Thus, we can avoid an error inside the function. Therefore, for processing long

DNA sequences which are mandatory for *de novo* genomic analysis[6], *dynamic block allocation* represents a possible method of choice for retrieving information from large files using buffers. One of the fundamental problems of modern biology is related to post-translational modifications of proteins in the endoplasmic reticulum where these processes occur, such as folding, twisting or splitting of pro-molecules[7]. In order to understand these processes, a dynamic block allocation may also be used for a better prediction of protein folding, especially for amino acid sequences showing some periodicity patterns.

Continuing the work presented in[8] with additional results and several refinements, we show two distinct algorithms for dynamic allocation of data blocks, namely Double Brute Force and Multi Brute Force.

MATERIALS AND METHODS

We consider a function *f* that accepts as input only segments of data larger than or equal to *n* nucleotides. Let *L* be the length of an input sequence. We call any string made of *L* characters from a set {A,T,C,G} a *L-string*. In actual practice, these algorithms will take only integer input sizes and we consider number three ($n \geq 3$) as the minimum input for a function *f* (as an example). A division of *L* by a constant integer will not always result in equal data blocks. Therefore, the last block of data may contain fewer nucleotides than the minimum limit accepted by *f* function (*e.g.* $n \geq 3$).

Standard brute force exemplification

We implemented an informatic strategy based on a brute-force search of an integer that meets indirectly the input conditions of some external function. To find an integer which represents the smallest multiple for *L*, we apply a brute force search method. For example, when looking for the divisors of an integer, these procedure for brute force search should take as parameters: the integer *L*, which represents the length of a DNA sequence, a boolean variable *P*, which stops the search when the result is found, and *A* variable which represents the divisor of *L*. Inside the loop, *A* variable is incremented every cycle, and a condition checks if the division result of *L* by *A* is an integer number. Thus, the boolean variable *P* (initially false), becomes true and the function returns the last increment of *A*.

```
P = False
L = 143
A = 1
Do Until P = True
  A = A + 1
  If (L/A)= (integer number) THEN P =
True
Loop
  return A
```

In the example above, the first integer result from a division of *L* by *A* is number *11*, which represents the smallest integer number that can divide *L* into another integer. Usually, the main disadvantage for brute-force method is that the number of possibilities is excessivly large for many problems. However, we managed to optimize the

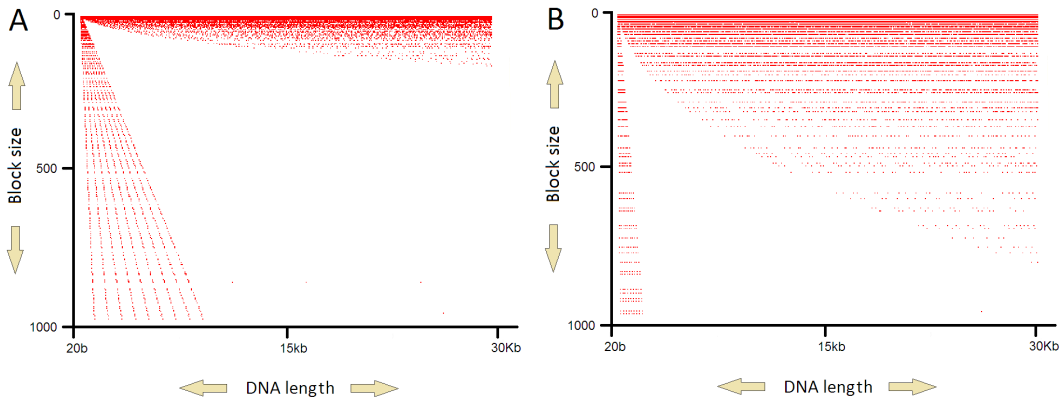

Fig. 1. Distribution of values generated by Double Brute Force and Multi Brute Force method. (**A**) Double Brute Force long distribution of data blocks lengths for *L-string* sequences starting from *5b* up to *30Kb*. (**B**) Multi Brute Force long distribution of data blocks lengths for *L-string* sequences starting from *5b* up to *30Kb*. Represented on y-axis is the block size range and on the x-axis the length of the *L-string* sequences.

process by using the *modulo* operator. Starting from this example, we take advantage of the *modulo* operator in order to implement a brute force search of an integer which can be used as a data block length (in most programming languages *modulo* operator is written Mod or %%).

Double Brute Force method

In order to obtain a desired number of nucleotides in the last block through *double brute force method*, we search for a remainder t which will represent the elements from the last block of data. The operation of finding the remainder is sometimes referred to as the modulo operation and in our case is definded by

$$t = L - \left[\frac{L}{A}\right] \times A$$

where *[L/A]* represents the largest integer less than or equal to *L/A*. For programming environments lacking on modulo opertator, it can be calculated as follows:

$$\mathrm{mod}(L, A) = L - A \times \mathrm{int}\left(\frac{L}{A}\right)$$

where *int()* represents a rounding function. Next we modify the pseudocode example from above by introducing the modulo operator. As shown below, by removing the condition inside the loop we obtain the same result:

```
t = 1
L = 143
A = 1
Do Until t = 0
  A = A + 1
  t = L - A * int(L/A)
Loop
return A
```

We create a condition in which we declare two limits. The first limit, t, is responsible for the minimum length of a data block and the second limit, m (the upper limit), is responsible for the maximum length of a data block. Next we begin to increment a new variable a, and at each incrementation of this variable we calculate the t value as the result of the expression:

$$L - a \times \mathrm{int}\left(\frac{L}{A}\right), \text{ reduced to } L \bmod a$$

Variable r represents the difference $L - t$ and is an integer multiple of a. The a variable is called the modulus of the congruence, in other words, both numbers have the same remainder when divided by a. The imposed condition is that t, must result in an integer greater than three in order to ensure at least four nucleotides in the last data block. However, the value of L may store a prime number. To avoid a prime number[9], we introduce a new variable v whose value must be equal to zero in order to meet the imposed condition. Knowing that prime numbers are divided by number one or by themselves, we define variable v as the result of the expression

$$L - 2 \times \mathrm{int}\left(\frac{L}{2}\right)$$

The result of this expression is number zero if L is not a prime number (variable v having a minimum role of primality tester[10,11]). If those two conditions are fulfilled, meaning that $t > 3$ and $v = 0$, then we say that the algorithm, up to this point, managed to find a number a, which provides t variable a value larger than three. The next step consists in finding the optimal length for data blocks in accordance with t variable. Variable r is a multiple of a, thus the difference $L - t$ will ensure a number divisible at least by three integers. The maximum length of a data block is declared through m variable (m = 10). To find the optimal length for data blocks, we must find an integer m which is a multiple of r. If we consider a new variable called b, which is the result of the expression *r mod m*, then we impose a condition by which the data block length is found when b is equal to zero.

```
Function DBFA(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

DBFA = m
End Function
```

At every iteration, m variable is incrementing. If the value assigned to m variable in the

incrementation process does not ensure a result equal to zero for *b* variable, then we consider that this method has not provided a valid result for dividing *L* by specified parameters. Thus, *L* becomes the length of a data block. Above we show the source code of the Double Brute Force Alocation function, syntactically compatible with VBA, VBScript, Visual Basic 4,5,6[12], Visual Basic NET and Visual Basic 2005.

Multi Brute Force method

Unlike *Double Brute Force* algorithm, this method is based on a brute force search for every decrementation of *L* variabile. In order to ensure a certain number (*n*) of nucleotides in the last block of data, we subtract the desired number of nucleotides from *L*. We define *q* as the result of the expression *L – n*, where *n* represents the number of nucleotides in the last block. We start a brute force search of an integer, called *block*, which can be a multiple for *q* variable. By noting *t* as the result of the expression *q mod block*, we could impose a condition by which the length of the block is found only if *t* is equal to zero. The *block* variable is incremented from the minimum value (number nine) which we define as a parameter, to an arbitrary value. If *block* variable passes the arbitrary value (ie. one thousand), the increment stops, *q* decreases by one, the minimum length of the *block* variable is redefined from the baseline (a reset) and the search begins again for a multiple of *q*.

```
Function MBFA(ByVal L As Variant) As Integer
Dim block As Integer
Dim t, q, n As Integer
last_block = 3
block = 9
t = 1
q = L - n
1:
Do Until t = 0 Or block > 1000
   block = block + 1
   t = q Mod block
Loop

If block > 1000 Or q < 9 Then
   q = q - 1
   block = 9
   GoTo 1
End If

MBFA = block
End Function
```

The search process continues until a result is found. Essentially, dynamic allocation of data blocks consists in dividing a *L-string* into smaller segments of equal length, with respect to the *L-string* total length. Above is the source code of the Multi Brute Force Alocation function, syntactically compatible with VBA, VBScript, Visual Basic 4,5,6, Visual Basic NET and Visual Basic 2005.

RESULTS AND DISCUSSION

In order to obtain an exact visualization of a data block length distribution, we conducted an experiment on virtual sequences (Fig. 2A). For larger DNA sequences (up to 30Kb), both methods exhibit two distinct areas of distribution. The first area of distribution is located at the top of the diagram whereas the second area of distribution is located on the left side of the diagram (Fig. 1A,B). For small sequences up to 300b (bases), the Double Brute Force method assigns lengths between 6b and 12b for each data
block set (Fig. 2B).

These allocations for data blocks, aim to obtainin a certain number of nucleotides in the last block, higher than the lowest limit imposed by the programmer, but less than the total length of a data block (Fig. 2D). For instance, a small DNA sequence of 22b can be divided manually into 10b for each block of data. First and second block of data having 10b each, and the third block only 2b. We consider a function whose input requirements are DNA sequences between 3b and 10b (Fig. 2D). Since the third block of data contains only 2b and the minimum input consists of 3b, function *f* can not accept this data block. Instead, the algorithm will provide data blocks of 6b, therefore a 22b sequence will be divided into four data blocks. The first three data blocks will contain 6b each and the fourth will contain 4b, which is compatible with the minimum 3b input of the *f* function. The second experiment was conducted to see how the algorithm behaves for very large sequences.

Even for large sequences the algorithm can find the optimal size for data blocks in the specified parameters (Fig. 2B).

Comparison between methods

DBFA (Double Brute Force Algorithm) and MBFA (Multi Brute Force Algorithm) functions were tested in parallel for sequences up to 300b in size. Apparently the block size allocation is treated differently by both versions of the algorithm (Fig. 2C).

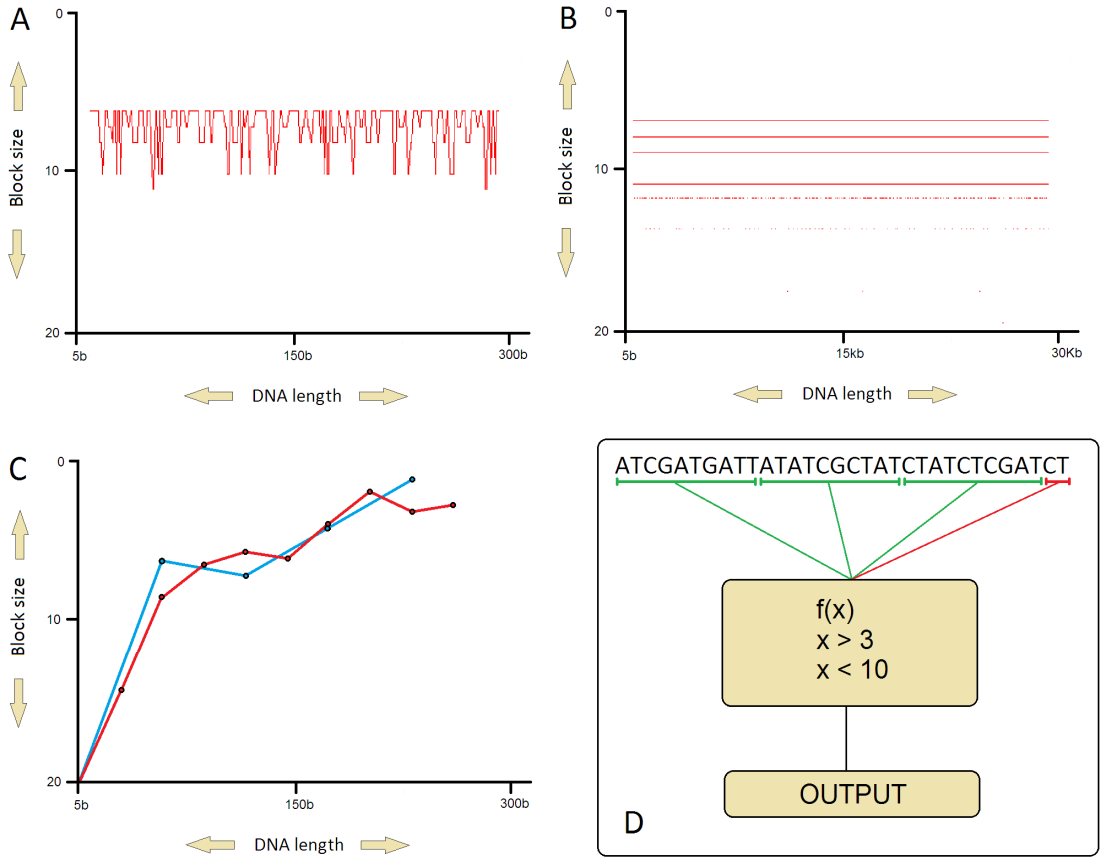

Fig. 2. MBFA and DBFA tests and comparisons. (**A**) short distribution of block sizes by Double Brute Force method ranging from *3b* to *20b*, for *L-string* sequences starting from *5b* up to *300b*. Represented on y-axis is the block size range and on the x-axis the length of the *L-string* sequences. The result is plotted by red lines for a better visualization of the distribution, (**B**) Double Brute Force method - long distribution of block sizes ranging from *3b* to *20b*, for *L-string* sequences starting from *5b* up to *30Kb*. *X* and *Y* axes have the same meaning as in section A. Data blocks are represented by red dots for a distinguishable distribution of the field, (**C**) allocation of data blocks for *L-string* ascending sequences, from *20b* up to *300b*. Represented on y-axis is the block size range and on the x-axis the length of the *L-string* sequences. Red line shows the distribution of data blocks for Multi Brute Force method whereas the blue line shows the distribution of data blocks for Double Brute Force method, (**D**) shows a schematic representation for *f* function and a random sequence of 32 nucleotides. The green horizontal segments signify the compatibility of a data block with function *f*, while the red segment signify the incompatibility of a data block with function *f*.

MBFA shows greater flexibility and manages to find more solutions than DBFA algorithm, showing a relatively uniform distribution. MBFA and DBFA methods can be incorporated as they are or ported to other platforms without additional effort.

Implementation

DBFA and MBFA functions work well with other functions which require some minimum input limitation. Below we show an example, coded in Visual Basic programming language, which calculates the size of data blocks for random DNA sequences (simple strings). The syntax that comes from *Basic* family of programming languages was chosen for many similarities with the *pseudocode* expressions[13], which allow a focus on the logic of the method. The application presented below is designed to work without a GUI (Graphical user interface) for an easy portability to other platforms.

```
Private Sub main()
sequence_input = Novo_Sequence(300,
"ADN")
Last_Block = 3
Min_Block  = 9
Max_Block  = 1000
Max_Rows   = 1
Max_Cols   = 3
b = MBFA(Len(sequence_input), Last_Block,
Min_Block, Max_Block)
```

```
For i = 1 To Len(sequence_input) Step b
   Total_Blocks = Total_Blocks + 1
   Col = Col + 1
   BlockData = Mid(sequence_input, i, b)
   output = output & "|" & BlockData
   If Col >= Max_Cols Then
      Col = 0
      Max_Rows = Max_Rows + 1
      output = output & vbCrLf
   End If
Next i

MsgBox output
End Sub

Function MBFA(L, Last_Block, MinBlock,
MaxBlock) As Variant
Dim RestetBlock As Variant
Dim q, t As Variant
RestetBlock = MinBlock
t = 1
q = L - Last_Block

1:
Do Until t = 0 Or MinBlock > MaxBlock
   MinBlock = MinBlock + 1
   t = q Mod MinBlock
Loop

If MinBlock > MaxBlock Or q < RestetBlock
Then
   q = q - 1
   MinBlock = RestetBlock
   If q < RestetBlock Then GoTo 2
   GoTo 1
End If

2:
MBFA = MinBlock
End Function

Function Novo_Sequence(ByVal nr As
Variant, ByVal tip As String) As String
Dim nucleo(1 To 5) As String
   nucleo(1) = "A"
   nucleo(2) = "T"
   nucleo(3) = "G"
   nucleo(4) = "C"
   nucleo(5) = "U"

For N = 1 To nr
   If (tip = "ADN") Then
      C = Int(3 * Rnd(3))
      p = p & nucleo(C + 1)
   End If

   If (tip = "ARN") Then
      C = Int(4 * Rnd(4))
      If (C + 1 = 2) Then C = 4
      p = p & nucleo(C + 1)
   End If
Next N
Novo_Sequence = p
End Function
```

Our Supplementary material 1 contains the source codes of DBFA and MBFA algorithms.

We aimed at assessing the effectiveness of the proposed algorithms by implementing a GUI application (Fig. 3A,B). This application allows the arrangement of a DNA sequence so that an exact number of nucleotides is provided in the last block of data. The software implementation is started by calling *Novo_Sequence* function, which creates a random DNA sequence of a predefined length. Next, all parameters are defined and declared, namely *Last_Block*, *Min_Block*, *Max_Block*, *Max_Rows* and *Max_Cols*.

Next, *MBFA* function is called in order to determine the optimal length for data blocks. Depending on the number of columns and rows allowed, the spatial positioning of data blocks is processed and displayed (Fig. 3A,B). *Dynamic block allocation* may also have an important role in aligning DNA sequences by segmental sequence alignment method. Other possible applications are in areas where optimization is mandatory, such as compression functions, optimal storage for databases, new processing methods for DNA *in silico* analysis[14] or perhaps some DNA sequence storage formats[15]. Another application for DBFA and MBFA functions consists in reading information from large FASTA files through buffer methods, by planning in advance the size of data blocks, which are entering in the buffer information flow.

Storage efficiency

In most software implementations a data block length is directly allocated into memory as empty space. Subsequently, this empty space will be loaded with data at the entire allocated space capacity or less than the allocated space. *In silico*, an equivalent for a nucleotide molecule is an ASCII (American Standard Code for Information Interchange) character of 8 bits. Thus, we refer at the memory space in terms of nucleotides. For instance, a DNA sequence of 33 nucleotides divided into four blocks of data (each of 10 nucleotides long), will allocate memory space for 40 nucleotides. However, the information that will be written in the allocated area is of 33 nucleotides, which leaves unused space for seven nucleotides. The obvious solution is to remove the unused space by recalculating the size of a data block. A DNA sequence of 33 nucleotides is divided by MBFA function into three blocks of data, each of 11 nucleotides long. This recalculation of a data

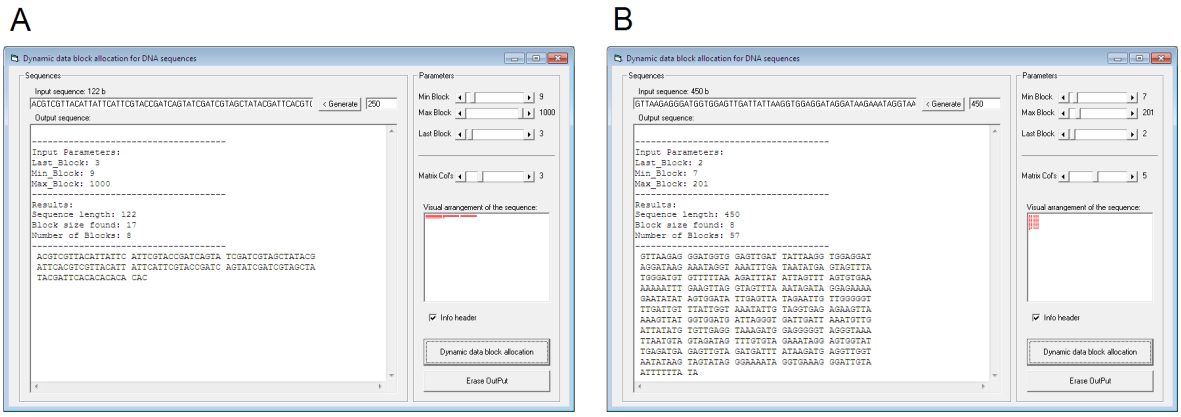

Fig. 3. MBFA application. (**A**) Multi Brute Force method - allocation of data blocks for a DNA sequence of 122b. On the right panel the parameters can be changed, namely the minimum and maximum length of a data block, the desired number of nucleotides in the last data block and the maximum number of columns in which data blocks are arranged. (**B**) Multi Brute Force method - allocation of data blocks for a DNA sequence of 450b.

block size allocates memory space for exactly 33 nucleotides, thus making an optimization of memory use. This method is applicable both to memory allocation and information storage for files. Furthermore, a dynamic allocation may protect any software implementation from buffer overflow, because the algorithm itself makes this allocation and not the programmer. The method efficiency consists in calculating the length of data blocks depending on the total length of the sequence.

In future applications we intend to use this methods for detecting different types of genomic signals using sliding window methods[16,19] or for self-organization models in proteins. Self-organization models in proteins are important for understanding the causes of certain diseases, such as diabetes and obesity[20,21]. Therefore, an accurate prediction of a 3D structure for certain hormones or prohormones, such as insulin or proinsulin (the precursor of insulin), proamylin (Proislet Amyloid Polypeptide) or leptin[22], may bring new insights in the metabolic syndrome[23].

Notice: Part of this paper was presented in a preliminary version as[8].

CONCLUSIONS

Dynamic data block allocation represents a novel and flexible method for partitioning DNA sequences into data blocks taking into account different rules imposed by a function. We showed two versions of this algorithm, namely DBFA (Double Brute Force Algorithm) and MBFA (Multi Brute Force Algorithm). We suggested possible applications in biology, but also a range of applications in computer science, such as storage efficiency or memory optimization. However, we used this algorithm in biology in order to divide DNA sequences into data blocks for further processing by a certain function responsible for data mining. Comparisons made between these two versions of the algorithm have shown that the allocation of data blocks involve different distributions, both for small sequences and for very large sequences, suggesting that DBFA provides more rigid solutions than MBFA. Nevertheless, given the role of this algorithm to partition information, we also thought of a possible application in the allocation of data packets for telecommunication. Moreover, in our supplementary material 1 we provide the source code of four applications that are based on this algorithm.

REFERENCES

1. Watson J.D. & Crick F.H.C., A Structure for Deoxyribose Nucleic Acid, Nature, 171 (4356), pp. 737–738, 1953.
2. Venter J.C. *et al.*, The sequence of the human genome, Science, 16, 291(5507), pp. 1304-1351, 2001.
3. Kumar S. & Dudley J., Bioinformatics software for biologists in the genomics era, Bioinformatics, 23, pp. 1713–1717, 2007.
4. Gary R. Skuse & Chunguang, DU., Bioinformatics Tools for Plant Genomics, Int J Plant Genomics, 2008.
5. Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and Eric W. Sayers, GenBank, Nucleic Acids Res. 38(Database issue): pp. 46–51, 2010.
6. Hand, D. J., Mannila, H. & Smyth, P., Principles of Data Mining, MIT Press, Cambridge, Mass, 2000.

7. Ionescu-Tîrgovişte C. & Guja C., Proinsulin, proamylin and the beta cell endoplasmic reticulum: the key for the pathogenesis of different diabetes phenotypes, Proc. Rom. Acad., Series B, 2, pp. 113–139, 2007.
8. Gagniuc, P. *et al.*, Dynamic Block Allocation for DNA sequences, GSP 2011, 2nd International Workshop on Genomic Signal Processing, pp. 125-130, 2011.
9. Luque, B. & Lacasa L., The first-digit frequencies of prime numbers and Riemann zeta zeros, Proc. R. Soc. A, 465, pp. 2197-2216, 2009.
10. Andrew Granville, Prime Number Patterns, The mathematical association of america, pp. 279-296, 2008.
11. Rene Schoof, Four primality testing algorithms, Algorithmic Number Theory, MSRI Publications, 44, pp. 101-126, 2008.
12. David I. Schneider, Computer Programming Concepts and Visual Basic, 2000.
13. Kreher, D.L. & Stinson, D.R., Pseudocode: A LATEX Style File for Displaying Algorithms, Bulletin of ICA, 30, pp. 11-24, 2000.
14. Miller, W., Comparison of genomic DNA sequences: Solved and unsolved problems, Bioinformatics, 17, 5, pp. 391-397, 2001.
15. Jacqueline Batley & David Edwards, Genome sequence data: management, storage, and visualization, BioTechniques, 46, pp. 333-336 (Special Issue), 2009.
16. Paul Gagniuc *et al.*, A sensitive method for detecting dinucleotide islands and clusters through depth analysis, RJDNMD, 18, 2, pp. 165-170, 2011.
17. Paul Gagniuc *et al.*, Genomin: a software framework for reading genomic signals, Proc. Rom. Acad., Series B, 1, pp. 53–56, 2011.
18. Paul Dan Cristea & Rodica Tuduce: Use of Nucleotide Genomic Signals in the Analysis of Variability and Inserts in Prokaryote Genomes, BIOCOMP, pp. 241-247, 2008.
19. Paul Dan Cristea, Rodica Tuduce, Iulian Nastac, Jan Cornelis, Rudi Deklerck, Marius Andrei, Signal representation and processing of nucleotide sequences, I. J. Functional Informatics and Personalised Medicine 1, 3, pp. 253-268, 2008.
20. Constantin Ionescu-Tîrgovişte, A short personal view on the pathogenesis of diabetes mellitus, Proc. Rom. Acad., Series B, 3, pp. 219–224, 2010.
21. Constantin Ionescu-Tîrgovişte, Proinsulin as the possible key in the pathogenesis of type 1 diabetes, Acta Endocrinologica, 5, 2, pp. 233-249, 2009.
22. Alina Constantin & Gabriela Costache, The emerging role of adipose tissue-derived leptin in inflammatory and immune responses in obesity: an update, Proc. Rom. Acad., Series B, 1, pp. 3-12, 2010.
23. Nicoleta Milici, A short history of the metabolic syndrome definitions, Proc. Rom. Acad., Series B, 1, pp. 13–20, 2010.