УДК 539.107

A PERMUTATION GENERATION DNA-BASED ALGORITHM – AN EXAMPLE OF A DNA COMPUTING KILLER APPLICATION

© Vojislav Stojkovic, Hongwei Huo

Стойкович В., Хьюо Х. Алгоритм генерации перестановок, основанный на ДНК. В работе представлен алгоритм генерации перестановок, заимствованный из ДНК. Такой алгоритм превосходит любой известный последовательный или параллельный фон Неймановский алгоритм для значительного числа элементов. Алгоритм может быть воспроизведен на ДНК-лаборатории или на ДНК-компьютере, а также на фон Неймановском компьютере для небольшого количества элементов.

INTRODUCTION

The permutation generation problem is a motivating computational puzzle, an interesting example of an application of computer science in combinatorial mathematics, one of the first nontrivial nonnumeric problems attacked by mathematicians and computer scientists. The permutation generation problem is to generate all possible ways of rearranging n, $n \geq 1$, distinct items. The permutation generation problem is simply stated, but not easily solved. The permutation generation problem has a long and distinguished history. Over one hundred Permutation Generation algorithms have been published during the past twenty years. The most well-know surveys of the field are D.H. Lehmer [6] from 1960, R.J. Ord-Smith [8, 9] from 1970-1971, and R. Sedgewick [10] from 1977. In 1956, C. Tompkins [15] wrote a paper describing a number of practical areas where permutation generation was being used to solve problems.

The study of existing and development of new methods for permutation generation is still important today because they illustrate the relationship between counting, recursion, and iteration.

Table 1

APPROXIMATE TIME TO GENERATE PERMUTATIONS OF n-ELEMENTS (1/msec per permutation)

n	$n!$	Time
1	1	
2	2	
3	6	
4	24	
...		
9	362880	
10	3628800	3 seconds
11	39916800	40 seconds
12	479001600	8 minutes
13	6227020800	2 hours
14	87178291200	1 day
15	1307674368000	2 weeks
16	20922789888000	8 months
17	*355689428096000*	*10 years*

The permutation generation problem has big the inherent limitation difficulty. Without computers – for $n \geq 10$ the permutation generation problem is practically not solvable.

Table 1 shows the values of $n!$ and the computation time. For $n > 15$ the computation time is too long.

Table 2

APPROXIMATE DISTRIBUTION OF PERMUTATION GENERATION ALGORITMS

90%	sequential algorithms	(targeting one-processor digital computers)
9%	parallel algorithms	(targeting multi-processor digital computers)
1%	parallel algorithms	(targeting DNA, quantum, ..., and others non vonNeumann computers)

2. PERMUTATIONS

A permutation, also called an "arrangement number" or "order," is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself.

The number of permutations on a set of n elements is $n!$
Example

There are 2!=2*1=2 permutations of {1, 2}, namely {1, 2} and {2, 1}.

There are 3!=3*2*1=6 permutations of {1, 2, 3}, namely {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, and {3, 2, 1}.

3. DNA

DNA, deoxyribonucleic acid, is a molecule found in every living cell, which directs the formation, growth, and reproduction of cells. DNA consists of nucleotides. Nucleotides contain compounds called phosphate, deoxyribose, and base. Within all nucleotides, phosphate and deoxyribose are the same, however, the bases vary. The four

distinct bases are: adenine (A), guanine (G), thymine (T), and cytosine (C). The exact amount of each nucleotide and the order in which they are arranged are unique for every kind of living organism. DNA represents information as a pattern of molecules on a DNA strand. A DNA strand is a string of the alphabet {A, C, G, T}. The length of a DNA strand is equal to the length of the string that represents the DNA strand.

4. DNA COMPUTER

A DNA computer is a chemical instrument consisting of a system of connected test tubes and other auxiliary units. DNA computers use the chemical properties of DNA molecules by examining the patterns of combination or growth of the molecules or strings. DNA computers can do this through the manufacture of enzymes, which are biological catalysts that could be considered the 'software' used to execute the desired DNA computation. DNA computers represent information in terms of DNA. In DNA computers, deoxyribonucleic acids serve as the memory units that can take on four possible positions (A, C, G, or T). DNA computers do not have the vonNeumann architecture. DNA computers are massively parallel and are considered promising for complex problems that require multiple simultaneous computations. DNA computers perform computations by synthesizing particular sequences of DNA and allowing them to react in test tubes. The task of the DNA computer is to check each possible solution and remove those that are incorrect, using restrictive enzymes. When the chemical reactions are complete, the DNA strands can then be analyzed to find the solution.

A super DNA computer is a programmable DNA computer.

5. DNA COMPUTING

In 1961 Feynman [4] predicts in 1994 Adleman [1] realized computations at a molecular level computing with DNA. DNA computing began in 1994 when Adleman showed that DNA computing was possible by solving the Traveling Salesman Problem on a DNA computer. Adelman [2] used DNA polymerase and Watson-Crick complementary strands to do DNA computation. Since then, it has been a surge of research in the DNA computation field. DNA computation has emerged in an exciting new research field at the intersection of computer science, biology, mathematics, and engineering. DNA computation has been demonstrated to have the capability to solve problems considered to be computationally difficult for von Neumann machines. After the Hamiltonian Path problem was solved, several researchers proposed solutions to a spectrum of NP-complete problems (such as Lipton [7]) dealing with satisfiability, cryptography, as well as other search oriented problems.

Adleman's [3] work has greatly influenced our work, however, our approach is different. Adleman's approach was biochemical-oriented, while our approach is computer science-oriented: (program+DNA)-oriented (based on super DNA computer and/or modeling and simulation of biochemical processes using the Easel or Prolog programming languages). Stojkovic [12, 13, 14] and Steele [11].

DNA computing is a field that holds promise for ultra-dense systems that pack megabytes of information into devices the size of a silicon transistor. Each molecule of DNA is roughly equivalent to a computer chip. With DNA computing, in order to find a solution, DNA molecules are primed to generate different chemical states. These molecules can be examined to determine whether the molecules have combined to form DNA strands or whether there is a separation of DNA strands. Most of the possible solutions are incorrect; however, one or a few may be correct.

We have assumed that DNA computations are error-free, i.e., they work perfectly without any errors. However, in reality DNA computations can be faulty because some DNA operations can introduce errors.

DNA operations are constrained by biological feasibility.

DNA operations may be:

– realized by the present biotechnology or

– implemented by simulation on the conventional von-Neumann computers.

6. DNA COMPUTATION MODEL

As computer components become smaller and/or more compact, scientists and engineers dream of a chemical, multi-processor computer, whose processors are individual molecules involved in chemical processes.

Following this thinking, we propose DNA computation model that involves the following three operations levels:

– Basic DNA operations (DNA molecular interactions);

– Test tube operations (proposed in 1996 by Gibbons, Amos, and Hodgson [5]) such as: remove, union, copy, select, and etc.

– High level operations

A selection of Easel/C-like programming language statements such as:

– begin-end (for grouping)

– if-then-else (for selection)

– for (for loop)

The basic DNA operations level is the chemical interactions between DNA-s. It may be seen as machine programming and may be interpreted as executions of machine code. The basic DNA operations can be implemented at DNA computers or simulated at vonNeumann machines.

The test tube operations level is an interface level that serves as an interface between von-Neumann machine and DNA machine. It may be seen as the hardware of a DNA computer. The test tube operations can be implemented at DNA computers or simulated at vonNeumann machines.

The high level operations – the programming language level can be implemented using vonNeumann machines with standard processors, operating systems, and programming languages processors.

In the last twelve years DNA computation has emerged as an exciting, fascinating, and important new research field at the intersection of computer science, mathematics, biology, chemistry, bioinformatics, and engineering.

The main reasons for the interest in DNA-computations are:

– size and variety of available DNA molecular "tool boxes"

– massive parallelism inherent in laboratory and chemical operations on DNA molecule

– feasible and efficient models

– physical realizations of the models

– performing computations in vivo.

Unfortunately it is still not clear whether DNA computing can compete (or will be able to compete in the near future) with existing electronic-digital computing. We propose that in the near future it will be possible to join von-Neumann and DNA computer in a functional super biocomputer. We are confident that in 10–20 years our desktop computers will be evolved into biocomputers. These machines will be able to perform calculations in seconds that take today's PCs hours, and solve in hours problems that take today's PCs years.

A computational substrate – a substance that is acted upon by the implementation of DNA computational model is DNA. DNAs are represented by strings. DNA computational model operates upon sets of strings. A DNA computation starts and ends with a single set of strings.

A DNA algorithm is composed of a sequence of operations upon one or more sets of strings. At the end of the DNA algorithm's execution, a solution to the given problem is encoded as a string in the final set.

Characterization of DNA computations using traditional measures of complexity, such as time and space is misleading due to the nature of the laboratory implementation of DNA computation. One way to quantify the time complexity of a DNA-based algorithm is to count the required numbers of "biological steps" to solve the problem. The biological steps include the creation of an initial library of strands, separation of subsets of strands, sorting strands by length, chopping and joining strands, and etc.

7. BASIC DNA OPERATIONS

An assignment is a finite sequence of unit assignments. An unit assignment is coded by a DNA strand. All unit assignments of an assignment have the same length.

The most important basic DNA operations are:

– Append (Concatenate, Rejoined) – appends two DNA strands with 'sticky ends'

– Melt (Anneal, Renaturation) – breaks two DNA strands with complementary sequences

– Cut -- cuts a DNA strand with restriction enzymes.

Append Operation: append(alpha, beta, gama)

Input:

– the unit assignment alpha and

– the unit assignment beta.

Output:

– the unit assignment gama.

Append operation appends the unit assignment alpha with the unit assignment beta.

The unit assignment beta can be appended at the beginning or at the end of the unit assignments alpha.

gama = beta . alpha or gama = alpha . beta

The default is at the beginning.

Melt Operation: melt(alpha, beta, gama)

Input:

– the unit assignment alpha and

– the unit assignment beta.

Output:

– the unit assignment gama.

Melt operation melts the unit assignment alpha with the unit assignment beta.

Unit assignment alpha can be melted from the beginning or from the end.

Default is from the beginning.

Cut Opperation: cat(alpha, i, beta)

Input:

– the unit assignment alpha and

– the non negative integer i.

Output:

– the unit assignment beta.

Cut operation cuts the unit assignment alpha for i-places.

Unit assignment alpha can be cutted from the beginning or from the end.

Default is from the beginning.

If cut length i is equal to 0, cut operation has no effect.

If cut length i is greater than the maximum length of unit assignment alpha, the result will be empty.

8. TEST TUBE OPERATIONS

A test tube contains an assignment.

The most important test tube operations are:

– Union (Merge, Create) – pours the context of more tubes into one tube

– Copy (Duplicate, Amplify) – makes copies of a tube

– Separate – separates an assignment into a finite sequence of assignments sorted by the length of unit assignments

– Detect – confirms presence or absence of an unit assignment in a tube

– Select – selects from an assignment an unit assignment on the uniformly random way

– Append (Concatenate, Rejoined) – appends an unit assignment to each unit assignment of an assignment

– Melt (Anneal, Renaturation) – melts each unit assignment of an assignment with an unit assignment

– Extract – extracts the context of one tube into two tubes using a pattern unit assignment

– Remove – removes unit assignments that contain occurrence(s) of other unit assignments

– Cut – cuts each unit assignment of an assignment for the given length.

Union (Merge, Create) Operation:

– *Union*($\{T_1, ..., T_i, ..., T_m\}$, T) or

– *Union*($\{T_m\}$, T)

Input: the finite sequence of tubes $\{T_1, ..., T_i, ..., T_m\}$.

Output: the tube T that contains the content of tubes T_i, where $i = 1, ..., m$.

Copy (Duplicate, Amplify) Operation:

– *Copy*(T, $\{T_1, ..., T_i, ..., T_m\}$) or

– *Copy*(T, $\{T_m\}$)

Input: the tube T.

Output: the finite sequence of tubes $\{T_1, ..., T_i, ..., T_m\}$. The tube T_i, where $i = 1, ..., m$, contains the content of the tube T.

Separate Operation:

– *Separate*(T, $\{T_1, ..., T_i, ..., T_m\}$) or

– *Separate*(T, $\{T_m\}$)

Input:

– the tube T.

Output:

– the finite sequences of tubes $\{T_1, ..., T_i, ..., T_m\}$, where $m \leq$ max(length(DNA strand)). The tube T_i, where $i = 1, ..., m$, contains DNA strands of the length i, where $i \leq m$.

Separate operation separates an assignment into a finite sequence of assignments sorted by the length of unit assignments.

Detect Operation: Detect(T)

Input:

– the tube T.

Output:

– true, if the tube T contains at least one unit assignment;

– false, if the tube T contains zero unit assignments.

Select Operation: Select(T, alpha$_i$)

Input:

– the tube T that contains the finite sequence of unit assignments {alpha$_m$}.

Output:

– the unit assignment alpha$_i$.

Select operation selects from the tube T that contains the finite sequence of unit assignments {alpha$_m$} an unit assignment alpha$_i$ on the uniformly random way.

If the tube T is empty, then the empty unit assignment will be returned.

Append Operation: Append(S, beta$_i$, T)

Input:

– the tube S that contains the finite sequence of unit assignments {alpha$_m$} and

– the unit assignment beta$_i$.

Output:

– the tube T that contains all unit assignments alpha$_m$ of the tube S concatenated to the unit assignment beta$_i$.

The unit assignment beta$_i$ can be appended at the beginning or at the end of the unit assignments alpha$_m$.

tube T = {beta$_i$. alpha$_m$} or tube T = {alpha$_m$. beta$_i$}

The default is at the beginning.

Melt Operation: Melt(S, beta$_n$, T)

Input:

– the tube S that contains the assignment alpha – that is

– the finite sequence of unit assignments {alpha$_m$} and

– the unit assignment beta$_n$.

Output:

– the tube T that contains all unit assignments alpha$_m$ from the tube S melted with the unit assignment *beta$_n$*.

The unit assignment alpha$_m$ can be melted from the beginning or from the end.

The default is from the beginning.

Extract Operation: Extract(alpha, T, T_1, T_2)

Input:

– the unit assignment alpha and

– the tube T.

Output:

– the tube T_1 consisting of DNA strands from the tube T that contains the unit assignment alpha as substrand and

– the tube T_2 consisting of DNA strands from the tube T that does not contain the unit assignment alpha as substrand.

Extract operation extracts using the given pattern DNA strands alpha the tube T into the tube T_1 and the tube T_2.

Remove Operation: Remove(T_1, T_2, T_3)

Input:

– the tube T_1 and

– the tube T_2.

Output:

– the tube T_3.

The tube T_3 contains the finite sequence of all unit assignments from the tube T_1 that do not contain occurrences of unit assignments from the tube T_2.

Cut Operation: Cut(T_1, i, T_2)

Input:

– the tube T_1 that contains the finite sequence of unit assignments {alpha$_m$} and

– the cut length i, $i \geq 0$.

Output:

– the tube T_2 that contains all unit assignments of tube T_1 cut for the length i.

Cut operation cuts each unit assignment of an assignment from the beggining for the given length.

DNA strands can be cut with restriction enzymes.

The test tube operations allow us to solve problems - code DNA-based algorithms and write the appropriate programs.

Test Tube Programming Language was proposed by Lipton [7] and developed by Adleman [3] and then discussed at many places.

9. DNA REPRESENTATIONS

DNA representation of a string $c_1 \ldots c_m$ is a sequence $c[1] \ldots c[m]$, where $c[i]$ is the character at the position i, where $i = 1, \ldots, m$. Characters are uniquely encoded by DNA strands.

If an unsigned integer number is not used for numerical calculations, then the unsigned integer number may be represented as a string of digits.

DNA representation of an unsigned integer number $d_1 \ldots d_m$ is a sequence $d[1] \ldots d[m]$, where $d[i]$ is the digit at the position i, where $i = 1, \ldots, m$. Digits are uniquely encoded by DNA strands.

If an unsigned integer number is used for numerical calculations, then the given DNA representation of an unsigned integer number is not suitable because it does not care on carries what complicates implementations of arithmetic operations with unsigned integer numbers.

If $0 \leq m \leq 10$, then a permutation of the integers $\{1, \ldots, m\}$ may be represented by unsigned integer numbers.

If $10 < m$, then DNA representation of a permutation of the integers $\{1, \ldots, m\}$ is $p[1]v[1] \ldots p[i]v[i] \ldots p[m]v[m]$ where $p[i]$ is the position i and $v[i]$ is the value at the position i, where $i = 1, \ldots, m$. Positions and values must be uniquely encoded by DNA strands.

10. A PERMUTATION GENERATION DNA-BASED ALGORITHM

Permutation generation algorithm generates the set of all permutations

$\{P_1, \ldots, P_m \mid P_m$ is m-th permutation of the integers $\{1, \ldots, m\}$ and $1 \leq m \leq m!\}$.

The input set T is (the tube T contains) a finite sequence of unit assignments (DNA strands) that represents candidates for permutations.

The output set T is (the tube T contains) a finite sequence of unit assignments (DNA strands) that represents permutations.

The input set T may be created using the following CreateInputSet(m, T) algorithm.

```
procedure CreateInputSet(m, T)   // m is input; T is output
{
        T = Ø;                   // empty
        for (i = 1; i <= m; i++)
        {
           Copy(T, { T[m] });
           for(j = 1; j<=m; j++) { Append(T[j], j, T[j]); }
           Union({ T[m] }, T);
        }
}
```

The output set T may be created using the following PermutationGeneration(T) algorithm.

```
PermutationGeneration(T)          // T is input and output
{
        for (i = 1; i <= m – 1; i++)
        {
           Copy(T, {T[m]});
           for(j = 1; j <= m; j++) // may be executed
                                              in parallel
           {
                 S[j] = {i ¬j};   // k >= i
                 for (k = i+1; k <= m; k++) { S[j] = S[j]
                                              U {k j}; };
                 Remove(T[j], S[j], T[j]);
        };
        Union({T[m]}, T);
   };
}
```

The frame of the algorithm is sequential.
Test tube operations execute in parallel.
The whole algorithm is semi-parallel.

Explanations

j means the unsigned integer number j from the range $1 .. m$.

$\neg j$ means all unsigned integer numbers from the range $1 .. m$, not equal to j.

$\neg j = \{1, ..., m\} \setminus \{j\} = \{1, ..., j-1, j+1, ..., m\}$.

$i\ j$ means the unsigned integer number j from the range $1 .. m$ at the position i.

$i\ \neg j$ means all unsigned integer numbers from the range $1 .. m$, not equal to j at the position i.

Remove($T[j]$, $\{i\ \neg j,\ k\ j\}$) removes from the tube $T[j]$ all DNA strands which contain at least one occurrence of the DNA substrands $i\ \neg j$ and/or $k\ j$.

Remove($T[j]$, $\{i\ \neg j,\ k\ j\}$) saves in the tube $T[j]$ only DNA strands which contain at the position i the value j and does not contain at other positions the value j.

At the end of the computation each of the surviving strings will contain exactly one occurrence of each unsigned integer number from the set $\{1, ..., m\}$ and so represents one of the possible permutations.

Complexity

Complexity of Permutation Generation DNA-based algorithm is $O(m)$ parallel-time.

Program Execution

Permutation Generation DNA-based program may be executed:

– step by step in a DNA-lab or on a DNA-computer

– automatically on a super DNA-computer or on an electronic-digital computer

(for small (less then 10) number of elements).

Test Example

The purpose of the test example is to "visualize" execution of Permutation Generation DNA-based algorithm.

The number of elements is 3.

CreateInputSet(3, T);

T
111 112 113 121 122 123 131 132 133
211 212 213 221 222 223 231 232 233
311 312 313 321 322 323 331 332 333

Copy(T, {T[1], T[2], T[3]});

T[1]	T[2]	T[3]
111 112 113	111 112 113	111 112 113
121 122 123	121 122 123	121 122 123
131 132 133	131 132 133	131 132 133
211 212 213	211 212 213	211 212 213
221 222 223	221 222 223	221 222 223
231 232 233	231 232 233	231 232 233
311 312 313	311 312 313	311 312 313
321 322 323	321 322 323	321 322 323
331 332 333	331 332 333	331 332 333

Remove(T[1], {1 ¬1, 2 1, 3 1}, T[1]);
Remove(T[2], {1 ¬2, 2 2, 3 2}, T[2]);
Remove(T[3], {1 ¬3, 2 3, 3 3}, T[3]);

T[1]	T[2]	T[3]
122 123 132 133	211 213 231 233	311 312 321 322

Union({T[1], T[2], T[3]}, T);

T
122 123 132 133
211 213 231 233
311 312 321 322

Copy(T, {T[1], T[2], T[3]});

T[1]	T[2]	T[3]
122 123 132 133	122 123 132 133	122 123 132 133
211 213 231 233	211 213 231 233	211 213 231 233
311 312 321 322	311 312 321 322	311 312 321 322

Remove(T[1], {2¬1, 3 1}, T[1]);
Remove(T[2], {2¬2, 3 2}, T[2]);
Remove(T[3], {2¬3, 3 3}, T[3]);

T[1]	T[2]	T[3]
213 312	123 321	132 231

Union({T[1], T[2], T[3]}, T);

T
213 312
123 321
132 231

Set T is the output - the result.

11. CONCLUSION

A Permutations Generation DNA-based Algorithm-Program is written in C/Easel-like programming language. It represents a DNA computer and computing environment based on DNA operations. This type of framework enables, facilitates, and supports the work of bioinformatics scientists and researchers in the field. Tools such as DNA computers will allow:

– Bioscientists to better understand the fundamental processes involved in biological systems and perhaps aid in predicting likely behaviors;

– Computer scientists to better understand parallelism and maybe to get the new parallel-oriented ideas.

12. FUTURE RESEARCH

Our future research will be focused on:

– Hunting/searching the new so-called "killer applications" – that is applications of DNA computation that would establish its superiority within a certain domain. Our favorite domains are: computer security & information assurance (cryptography), DNA-controlled devices, DNA-motors, and etc.

We believe that an assured future for DNA computation can only be established through the discovery of such and other applications of DNA-computations;

– Introducing through permutation generation counting, recursion, and iteration – the fundamental concepts of the classical computer science – into DNA-computation.

REFERENCES

1. *Adleman L.M.* Molecular Computation of Solutions of Combinatorial Problems. 1994. Science, 266. P. 1021-1024.
2. *Adleman L.M.* On Constructing a Molecular Computer // R. Lipton and E. Baum, editors, DNA Based Computers, Discrete Mathematics and Theoretical Computer Science Series. American Mathematical Society. 1995. V. 27. P. 1-21.
3. *Adleman L.M.* Computing with DNA // Scientific American. 1998. V. 279(2). P. 54-61.
4. *Feynman R.P.* There's plenty of room at the bottom. Miniaturization, Reinhold, 1961.
5. *Gibbons A., Amos M. and Hodgson D.* Models of DNA computation: Mathematical Foundations of Computer Science, Lecture Notes in Computer Science. Springer. 1996.
6. *Lehmer D.H.* Teaching combinatorial tricks to a computer // Proceedings of Symposium Appl. Math, Combinatorial Analysis. American Mathematical Society, Providence, R. I. 1960. V. 10. P. 179-193.
7. *Lipton R.J.* DNA solution of hard computational problems. Science. 1995. V. 268. P. 542-545.
8. *Ord-Smith R.J.* Generation of permutation sequences Part 1 // Computer. 1970. V. 13. № 3. P. 152-155.
9. *Ord-Smith R.J.* Generation of permutation sequences: Part 2 // Computer 1971. V. 14. № 2. P. 136-139.
10. *Sedgewick R.* Permutation Generation Methods // Computing Surveys 1977. V. 9. № 2. 137-164.
11. *Steele G. and Stojkovic V.* Agent-Oriented Approach to DNA Computing (6 pages poster presentation) in Proceedings, Poster, Workshops, and Demo Abstracts of CSB2004 Conference, Stanford, CA. 2004. August. P. 16-19.
12. *Stojkovic V., Huo H. and Britto E.* DNA Based Addition and Subtraction of Two Unsigned Integer Numbers Inspired by Unrestricted Grammars Implemented in Prolog Language (6 pages poster presentation) in Proceedings, Poster, Workshops, and Demo Abstracts CD of CSB2006 Conference, Stanford, CA. 2006. August. P. 14-18.
13. *Stojkovic V. and Huo H.* DNA Based Addition and Subtraction of Two Unsigned Integer Numbers Inspired by Unrestricted Grammars (1 page poster presentation) in Proceedings of DNA12 Conference, Seoul, South Korea. 2006. June. P. 5-9.
14. *Stojkovic V., Steele G. and Lupton W.* Using Easel for Modeling and Simulating the Interactions of Cells in Order to Better Understand the Basics of Biological Processes and to Predict Their Likely Behaviors (6 pages poster presentation) in Proceedings, Poster, Workshops, and Demo Abstracts of CSB2003 Conference, Stanford, CA. 2003. August. P. 11-14.
15. *Tompkins C.* Machine attacks on problems whose variables are permutations // Proceedings of Symposium Appl. Math., Numerical Analysys, N. Y., 1956. V. 6. McGraw-Hill, Inc. P. 195-211.

Поступила в редакцию 12 августа 2006 г.