Learning Biological Algorithms of Playing Tic-Tac-Toe Game on a Biological Computer

Weng-Long Chang[1], Athanasios V. Vasilakos[2], Ying-Chi Lin[3],
Chun-Wei Tung[4] and Chia-Chi Wang[5]

[1]Department of Computer Science and Information Engineering,
National Kaohsiung University of Applied Sciences
No 415, Chien Kung Road, Kaohsiung 807, Taiwan, R. O. C.
E-mail: changwl@cc.kuas.edu.tw

[2]Department of Computer Science, Electrical and Space Engineering
Luleå University of Technology
SE-931 87 Skellefteå, Sweden
E-mail: th.vasilakos@gmail.com

[3, 4, 5]School of Pharmacy,
Kaohsiung Medical University
100 Shih-Chuan 1st Road, Kaohsiung 807, Taiwan, R. O. C.
E-mail: [3]yclin@kmu.edu.tw, [4]cwtung@kmu.edu.tw and [5]chiachiwang@kmu.edu.tw

Abstract

A tic-tac-toe is a two-person game in which there is a three-by-three array of blank squares. Players occupy squares alternately, marking the square occupied by an O or an X, respectively. The first player who attains a horizontal, vertical, or diagonal sequence of three of his symbols wins. If the whole array is filled without either player's attaining such a sequence, the game is a draw. Newell and Simon in [1] proposed the strategy of playing tic-tac-toe as a production system. In this paper, it is demonstrated that biological operations can be used to learn how to implement the strategy of playing tic-tac-toe proposed by Newell and Simon where each O and each X are encoded as DNA strands. In order to achieve this goal, biological algorithms are proposed to play a tic-tac-toe with one person. Furthermore, this work offers clear evidence of the ability of molecular computing to learn human's intelligence.

Keywords – Molecular Computing, Biological Algorithms, Tic-Tac-Toe

I. INTRODUCTION

Playing games is the behavior of human's intelligence. A tic-tac-toe in [1] is a two-person game in which there is a three-by-three array of blank squares and players occupy squares alternately, marking the square occupied by an O or an X, respectively. The first player who attains a horizontal, vertical, or diagonal sequence of three of his symbols wins. If the whole array is filled without either player's attaining such a sequence, the game is a draw.

Feynman [2] in 1961 first presented molecular computation, but his idea was not implemented by experiment until a few decades later. In 1994 Adleman [3] succeeded in solving an instance of the Hamiltonian path problem in a test tube, just by handling DNA strands. Many famous biological algorithms have been proposed for solving many difficult problems in [4-5]. An interesting open question is asking whether bio-molecular operations and DNA strands are able to learn the behavior of human's intelligence (for example, playing tic-tac-toe that is the simplest game with human together) or not.

Our major contributions in this paper are as follows.

- **We show that biological operations and DNA strands are able to learn the behavior of human's intelligence.**
- **We also demonstrate that the proposed biological method that is made of biological operations and DNA strand**s **is able to play tic-tac-toe with human together.**

The rest of the paper is organized as follows: in Section II, DNA model of computation is introduced. In Section III, the motivation of this work is given. In Section IV, the development of molecular computing is illustrated. In Section V, the strategy of playing tic-tac-toe as a production system proposed by Newell and Simon in [1] is introduced. In Section VI, based on learning the Newell-Simon strategy, the biological algorithms of playing tic-tac-toe with human together are proposed. In Section VII, assessment of complexity to the proposed biological algorithms is given. In Section VIII, a brief conclusion is given.

II. DNA MODEL OF COMPUTATION

The genetic information of cellular organisms is encoded by DNA (deoxyribonucleic acid) in [4, 5]. DNA includes polymer chains which are commonly regarded as DNA strands. By means of an automated process, DNA strands may be synthesized to order. Each strand may be made of a sequence of nucleotides, or bases, attached to a sugar-phosphate “backbone”. The four DNA nucleotides are adenine, guanine, cytosine and thymine, commonly abbreviated to *A*, *G*, *C* and *T*, respectively. By chemical convention, each strand has a 5’ end and a 3’ end. Because one end of the single strand has a free (i.e., unattached to another nucleotide) 5’ phosphate group, and the other has a free 3’ deoxyribose hydroxyl group, therefore, any single strand has a natural orientation, as described in [4].

The classical double helix of DNA is formed when two separate single strands bond. Bonding occurs by the pairwise attraction of bases: *A* bonds with *T* and *G* bonds with *C*. The pairs (*A*, *T*) and (*G*, *C*) are therefore known as complementary base pairs in [4]. Double-stranded DNA may be denatured into single strands by heating the solution to a temperature determined by the composition of the strand in [4]. Heating breaks the hydrogen bonds between complementary strands (Figure 2-1) in [4]. Beca-

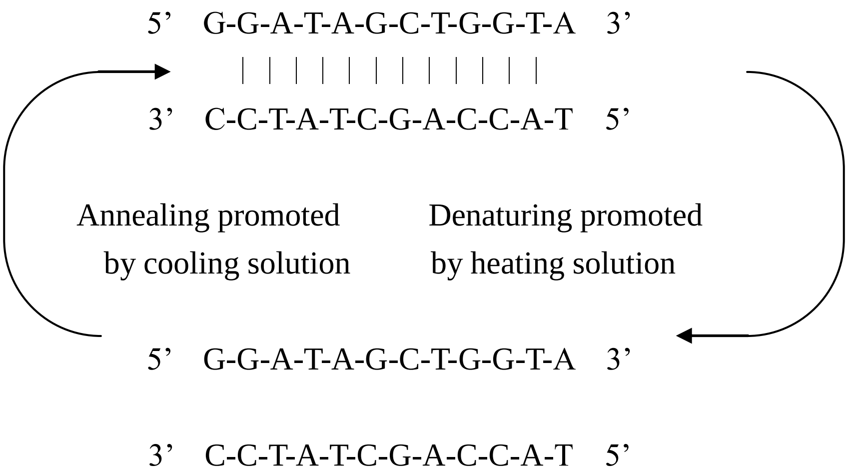

Figure 2-1: DNA denaturing and annealing.

use a $G - C$ pair is joined by three hydrogen bonds, the temperature required to break it is slightly higher than that for an $A - T$ pair, joined by only two hydrogen bonds in [4]. This factor must be taken into account when designing sequences to represent computational elements. Annealing is the reverse of melting, whereby a solution of single strands is cooled, and allowing complementary strands to bind together (Figure 2-1) in [4]. In double-stranded DNA, if one of the single strands contains a discontinuity (i.e., one nucleotide is not bonded to its neighbor) then this may be repaired by DNA ligase in [4]. This allows us to create a unified strand from several bound together by their respective complements.

The following bio-molecular operations cited in [3, 5, 7, 8, 9] will be applied to learn how human play a tic-tac-toe. From [4], the implementation of eight biological operations that are denoted in **Definition 2-1** through **Definition 2-8** is described below. Each implementation illustrates only one possible way to perform the computational behavior of one biological operation. Future improvements in laboratory techniques may well yield more efficient and error-resistant implementations of biological operations, but this does not diminish the theoretical power of the model. We simply offer descriptions of the implementation in order to show the feasibility, in principle, of executing biological operations in vitro (that is to say, every biological operation is completely feasible using existing laboratory techniques). From a biological standpoint, all sequences generated to represent bits must be checked to ensure that the DNA strands that they encode do not form unwanted secondary structures with one another (i.e., strands remain separate at all times, and only bind together when this is required). The problem of strand design for DNA-based computing has been addressed at length, and we use the methods described in [4] to minimize the possibility of unwanted binding.

Definition 2-1: Given set $X = \{x_n\, x_{n-1} \ldots x_2\, x_1 | \;\forall\, x_d \in \{0, 1\}$ for $1 \le d \le n\}$ and a bit x_j, the bio-molecular operation "Append-Head" appends x_j onto the head of every element in set X. The formal representation is written as Append-Head(X, x_j) = $\{x_j\, x_n\, x_{n-1} \ldots x_2\, x_1 | \;\forall\, x_d \in \{0, 1\}$ for $1 \le d \le n$ and $x_j \in \{0, 1\}\}$.

Definition 2-2: Given set $X = \{x_n\, x_{n-1} \ldots x_2\, x_1 | \;\forall\, x_d \in \{0, 1\}$ for $1 \le d \le n\}$ and a bit x_j, the bio-molecular operation, "Append-Tail", appends x_j onto the end of every element in set X. The formal representation is written as Append-Tail(X, x_j) = $\{x_n\, x_{n-1} \ldots x_2\, x_1\, x_j | \;\forall\, x_d \in \{0, 1\}$ for $1 \le d \le n$ and $x_j \in \{0, 1\}\}$.

Two strands (labeled *S* and *T* in Figure 2-2) may be concatenated by the following process: create a *linker* strand, which has a sequence that is the complement of *S* followed by the complement of *T*. This linker strand is affixed to a surface with a magnetic bead (Figure 2-2(a)). Strand *S* is then added to the solution, and anneals with the linker strand in the appropriate position (Figure 2-2(b)). Strand *T* is then added to the solution, and this also anneals with the linker strand, at a position immediately adjacent to strand *S* (Figure 2-2(c)). The ligase enzyme is then added to the solution to seal the "nick" between S and T, forming a single strand which may be freed by heating the solution to break its bonds with the linker strand (Figure 2-2(d)). The implementation of the *concatenate*() operation defined above may easily be used to append a specific sequence, s, to the head of each strand in a tube X. The sequence s corresponds, in this case, to the strand S defined in Figure 2-2, and strand T in Figure 2-2 corresponds to the beginning sequence of every strand in the tube X. In this case, only the beginning sequence of every strand anneals to the linker strand. Clearly, then, after a series of *append-head*() operations denoted in **Definition 2-1** has been performed on a strand, its sequence will be made up of a number of sequences representing bit-strings. A similar implementation can be used to complete the *append-tail*() operation denoted in **Definition 2-2.**

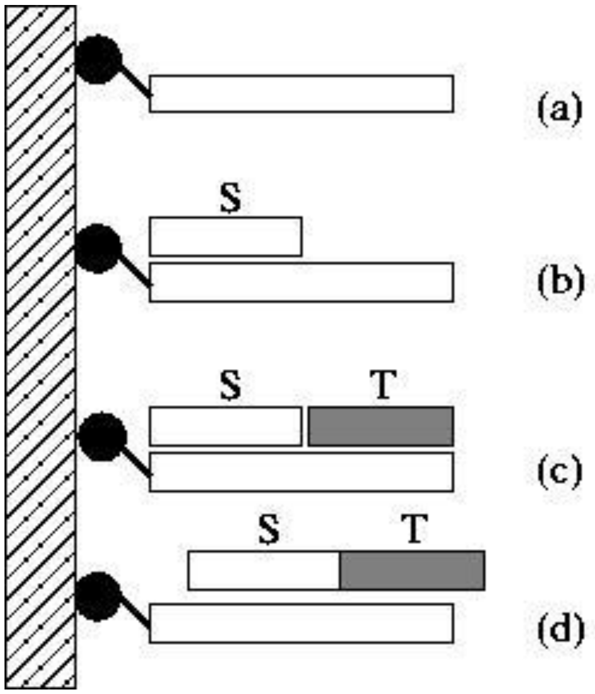

Figure 2-2: Concatenation process: (a) Linker strand affixed to surface. (b) S anneals to linker strand. (c) T anneals to linker strand, adjacent to S. (d) S and T ligated to form a single strand, which is then freed by heating the solution.

Definition 2-3: Given set $X = \{x_n\ x_{n-1}\ \ldots\ x_2\ x_1 |\ \forall\ x_d \in \{0, 1\}$ for $1 \le d \le n\}$, the bio-molecular operation "Discard(X)" sets X to be an empty set and can be represented as "$X = \varnothing$".

The implementation of the *Discard*(X) operation denoted in **Definition 2-3** is to discard the content of a tube X, and the tube X is replaced by a new, empty tube. Since the number of tubes will generally be one, this is considered to be a constant-time operation.

Definition 2-4: Given set $X = \{x_n\ x_{n-1}\ \ldots\ x_2\ x_1 \mid \forall\ x_d \in \{0, 1\}$ for $1 \le d \le n\}$, the bio-molecular operation "Amplify(X, $\{X_i\}$)" creates a number of identical copies X_i of set X, and then "Discard(X)".

The implementation of the *Amplify*(X, $\{X_i\}$) operation denoted in **Definition 2-4** is that the polymerase chain reaction (PCR) is used with its initial input being a tube X. This reaction is used to massively amplify (possibly small) amounts of DNA that begin and end with specific primer sequences. As every strand in the tube X is delimited by these sequences, they are all copied by the reaction. The result of the PCR is then divided equally between the specified number of tubes (the number of PCR cycles may therefore be adjusted to ensure a constant DNA volume per tube, regardless of the number of tubes).

Definition 2-5: Given set $X = \{x_n\ x_{n-1}\ \ldots\ x_2\ x_1 \mid \forall\ x_d \in \{0, 1\}$ for $1 \le d \le n\}$ and a bit, x_j, if the value of x_j is equal to one, then the bio-molecular *extract* operation creates two new sets, $+(X, x_j^1) = \{x_n\ x_{n-1}\ \ldots\ x_j^1\ \ldots\ x_2\ x_1 \mid \forall\ x_d \in \{0, 1\}$ for $1 \le d \ne j \le n\}$and $-(X, x_j^1) = \{x_n\ x_{n-1}\ \ldots\ x_j^0\ \ldots\ x_2\ x_1 \mid \forall\ x_d \in \{0, 1\}$ for $1 \le d \ne j \le n\}$. Otherwise, it produces another two new sets, $+(X, x_j^0) = \{x_n\ x_{n-1}\ \ldots\ x_j^0\ \ldots\ x_2\ x_1 \mid \forall\ x_d \in \{0, 1\}$ for $1 \le d \ne j \le n\}$and $-(X, x_j^0) = \{x_n\ x_{n-1}\ \ldots\ x_j^1\ \ldots\ x_2\ x_1 \mid \forall\ x_d \in \{0, 1\}$ for $1 \le d \ne j \le n\}$.

The implementation of the *extract* operation denoted in **Definition 2-5** is that affinity purification is applied to extract any strands from a tube X containing a short strand, s, that encodes the value of a bit, x_j. This process applies a probe sequence, which is complementary to the target sequence being searched for. Probes are fixed to a surface, and capture strands through annealing any strands containing the target sequence. Captured strands may then be separated from the rest of the population by placing them in a separate solution, which is heated to break the bonds between the probes and the target sequence. The probe used is therefore the complementary sequence of s. Retained strands are placed in one new tube, $U = +(X, s)$, and the remainder are placed in another new tube, $V = -(X, s)$.

Definition 2-6: Given m sets $X_1 \ldots X_m$, the bio-molecular *merge* operation, *merge*(X_1, $\ldots$, X_m) $= \cup(X_1, \ldots, X_m) = X_1 \cup \ldots \cup X_m$.

The implementation of the *merge* operation denoted in **Definition 2-6** is that the contents of tubes (sets) $\{X_i\}$ are simply merged by pouring. The number of tubes will generally be low, so this is considered to be a *constant-time* operation.

Definition 2-7: Given set $X = \{x_n\, x_{n-1} \ldots x_2\, x_1 | \; \forall\, x_d \in \{0, 1\}$ for $1 \le d \le n\}$, the bio-molecular operation "Detect(X)" returns *true* if $X \ne \varnothing$. Otherwise, it returns *false*.

The implementation of the *detect* operation denoted in **Definition 2-7** is that a tube X is run through a gel electrophoresis process, which is generally used to sort DNA strands on length. Any DNA present in X shows up as a visible band in the gel; if DNA strands of the appropriate length are present, the operation returns *true*. If there are no visible bands corresponding to DNA of the correct length, then the operation returns *false*. The length criterion is used to ensure that the DNA fragments present do not cause a false positive result. If the DNA in the band corresponding to the contents of X is required in a subsequent processing step, the band may be excised from the gel by cutting, and then is soaked to remove the strands for further use.

Definition 2-8: Given set $X = \{x_n\, x_{n-1} \ldots x_2\, x_1 | \; \forall\, x_d \in \{0, 1\}$ for $1 \le d \le n\}$, the bio-molecular operation "Read(X)" describes any element in X. Even if X contains many different elements, the bio-molecular operation can give an explicit description of exactly one of them.

The implementation of the *read* operation denoted in **Definition 2-8** is that *gel electrophoresis* is used to sort DNA strands in a tube X by size. Electrophoresis is the movement of charged molecules in an electric field. Since DNA molecules carry a negative charge, when placed in an electric field they tend to migrate toward the positive pole. The rate of migration of a molecule in an *aqueous* solution depends on its shape and electric charge. Since DNA molecules have the same charge per unit length, they all migrate at the same speed in an aqueous solution. However, if electrophoresis is carried out in a *gel* (usually made of agarose, polyacrylamide, or a combination of the two), the migration rate of a molecule is also affected by its *size*. This is due to the fact that the gel is a dense network of pores through which the molecules must travel. Smaller molecules therefore migrate faster through the gel, thus sorting them according to size. DNA strands of the appropriate length in *base pairs* are measured.

III. MOTIVATION OF THIS WORK

After Adleman's article in [3] that is a milestone was published in 1994, many biological algorithms were proposed to solve many NP-Complete Problems with the number of bits *n* that is the size of those problems. In those methods, the first phase is to construct 2^n DNA strands as their solution space. Next, in the second phase, 2^n DNA strands were filtered through biological operations. Next, in the third phase, illegal solutions are removed and legal solutions are reserved. Finally, in the fourth phase, reading the required answer(s) is completed. For a biological algorithm, the first biggest challenge is to that the problem of DNA strand design has been addressed at length. From a biological standpoint, all sequences generated to represent bits must be checked to ensure that the DNA strands that they encode do not form unwanted secondary structures with one another (i.e., strands remain separate at all times, and only bind together when this is required). The second biggest challenge is to how 2^n DNA strands are filtered through biological operations without occurring of errors. Bonnet et al. in [6] used *intensity* of *green fluorescent protein* to encode two values '0' and '1' of a bit and implemented **AND**, **NAND**, **OR**, **XOR**, **NOR**, and **XNOR** gates. This gives another very good choice for representing two values '0' and '1' of a bit. An interesting open question is asking whether biological algorithms are able to learn the behavior of human's intelligence (for example, playing with human together a tic-tac-toe that is the simplest game) or not. Our motivation is to find the answer of the interesting open question.

IV. ILLUSTRATION OF RELATIVE WORKS ON MOLECULAR COMPUTING

From [10], Qian and Winfree showed that arbitrary chemical reaction networks can in principle be implemented with DNA strand displacement cascades was a major step toward proving the generality and universality of pure-DNA systems. From [11], Velasco et al. presented transport spectroscopy measurements of Landau level gaps in double-gated suspended bilayer graphene with high mobilities in the quantum Hall regime. From [12], they investigated the possibility of constructing an exponentially large number of sequences from a short initial sequence and simple replication rules, including those resembling genomic replication processes. From [13], a polynomial-time algorithm was proposed for that decides, given a matching that is stable under the partial preference orderings, whether that matching is stable and optimal for one side of the market under some refinement of the partial orders. From [14], Cook et al. examined that the self-assembly of structures growing at "temperature 1", meaning that no cooperativity was needed for the bonding of new elements – if a bond matched, the particle could stick. From [15], Sun et al.

characterized methods to protect linear DNA strands from exonuclease degradation in an Escherichia coli based transcription-translation cell-free system, as well as mechanisms of degradation. From [16], Sadowski et al. demonstrated the "developmental" self-assembly of a DNA tetrahedron.

From [17], Qian et al. proposed a chemical implementation of stack machines — a Turing-universal model of computation similar to Turing machines — using DNA strand displacement cascades as the underlying chemical primitive. From [18], their results suggested that DNA strand displacement cascades could be used to endow autonomous chemical systems with the capability of recognizing patterns of molecular events, making decisions and responding to the environment. From [19], using a simple DNA reaction mechanism based on a reversible strand displacement process, they experimentally demonstrated several digital logic circuits, culminating in a four-bit square root circuit that comprises 130 DNA strands. From [20], it was reported that complex molecular circuits with reliable digital behavior can be created using DNA strands. It was introduced from [21] that natural computing was concerned with human-designed computing inspired by nature as well as with computation taking place in nature. From [22], it is shown how the same principles can be applied to breaking the Data Encryption Standard. From [23], molecular algorithms of implementing bio-molecular databases on a biological computer were proposed.

From [24], it is showed that the proposed quantum algorithm of implementing Boolean circuits yielded from the DNA-based algorithm solving the vertex-cover problem in [25, 26] of any graph G with m edges and n vertices is the optimal quantum algorithm, and also is demonstrated that mathematical solutions of the same bio-molecular solutions are represented in terms of a unit vector in the finite-dimensional Hilbert space. It is indicated from the hidden variable theorem in [27] which in the case of computing states that no classical computer can simulate a quantum computer without suffering from an exponential slowdown. This also is to say that any classical computer can simulate a quantum computer in term of polynomial time is the violation of the hidden variable theorem. From [28], it is shown that arithmetical operations of complex vectors can be implemented by means of the proposed DNA-based algorithms.

V. ILLUSTRATION OF THE NEWELL-SIMON METHOD TO PLAY A TIC-TAC-TOE

A representation of a tic-tac-toe board is shown in Figure 5-1. Nine blank squares on the tic-tac-toe board in Figure 5-1 are numbered as one through nine. The *first* square, the *third* square, the *seventh* square and the *ninth* square are called *corner* squares. The *second* square, the *fourth* square, the *sixth* square and the *eighth* square are called *side* squares. The *fifth* square is called a *center* square. The first player who attains a horizontal, vertical, or diagonal sequence of three of his symbols wins. If the whole array is filled without either player's attaining such a sequence, the game is a draw.

1	2	3
4	5	6
7	8	9

Figure 5-1: The tic-tac-toe board

Newell and Simon in [1] proposed the good strategy of playing a tic-tac-toe. Because the game is a draw when viewed from a game-theoretic standpoint, *good* means here a strategy that will guarantee a draw and that will give the opponent as many opportunities as possible of making a losing mistake. The Newell-Simon method in [1] is described below. In the Newell-Simon method, it is assumed that own is a computer with a mark (X) and its opponent is one person with a mark (O).

The Newell-Simon method: Select next moving from a tic-tac-toe board.

(1) **If** one player (a computer) finds that there is a line with two of the computer's marks and one blank, **then** an X is filled into the *blank* square and the Newell-Simon method is terminated.

(2) **If** one player (a computer) checks that there is a line with two of the opponent's marks and one blank, **then** an X is filled into the blank square to protect that the opponent wins the game and the **Newell-Simon method** is terminated.

(3) **If** one player (a computer) finds that there are two lines, each with one of the computer's mark and two blanks, intersecting in a single blank square, **then** an X is filled into the single blank square to create two lines in which each line has two computer's marks and one blank, thus forking the opponent and the **Newell-Simon method** is terminated.

(4) **If** one player (a computer) checks whether in the board the *fifth* square that is called a *center* square is empty or not and the *checked* condition is satisfied, then

an X is filled into the *center* square and the **Newell-Simon method** is terminated.

(5) **If** one player (a computer) checks whether in the board the *second* square, the *fourth* square, the *sixth* square or the *eighth* square that are all called *side* squares are occupied by the ***opponent*** or not and also simultaneously checks whether the *opposite* of each *side* square is an empty square or not and the *checked* condition is satisfied, then an X is filled into the *opposite* of the *side* square and the **Newell-Simon method** is terminated.

(6) **If** one player (a computer) checks whether in the board the *first* square, the *third* square, the *seventh* square or the *ninth* square that are all called *corner* squares are occupied by the ***opponent*** or not and also simultaneously checks whether the *opposite* of each *corner* square is an empty square or not and the *checked* condition is satisfied, then an X is filled into the *opposite* of the *corner* square and the **Newell-Simon method** is terminated.

Lemma 5-1: From the **Newell-Simon method**, next moving in a tic-tac-toe board is selected.

Proof: Please refer to [1]. ■

VI. BIOLOGICAL ALGORITHMS OF PLAYING A TIC-TAC-TOE

Biological operations and DNA strands will be used to learn how to use the Newell-Simon method to together play a tic-tac-toe with human. Biological algorithms are proposed in the following subsections.

A. Data Structures of Playing a Tic-tac-toe

First we will develop a representation for the tic-tac-toe board shown in Figure 5-1. We will number the blank squares on the tic-tac-toe board shown in Figure 5-1 this way: we will use *nine* tubes (sets) to encode *nine* squares (positions) and to store the contents of each position (square) on the tic-tac-toe board shown in Figure 5-1. It is assumed that tube (set) S_k for $1 \leq k \leq 9$ is used to encode the k^{th} square (position) and to store its contents. Each tube S_k for $1 \leq k \leq 9$ is initialized as an empty tube. An empty tube S_k for $1 \leq k \leq 9$ means that the k^{th} square (position) is not occupied by players.

Two *distinct* DNA strands (sequences) in [3-5, 7-9] are designed to minimize the possibility of unwanted binding and their length is θ base pairs. One represents the value “0” for a binary variable with a *bit b* and the other represents the value “1” for it.

For the sake of convenience in our presentation, it is assumed that b^1 denotes the value of b to be 1, b^0 defines the value of b to be 0, and b defines the value of b to be 0 or 1. b^0 is applied to encode an O that is one of two marked symbols, and b^1 is employed to encode an X that is also one of two marked symbols. If tube (set) S_k for $1 \leq k \leq 9$ contains b^0 (DNA strands), then this means that the k^{th} square (position) is occupied and is filled by an O. Similarly, if tube (set) S_k for $1 \leq k \leq 9$ contains b^1 (DNA strands), then this means that the k^{th} square (position) is occupied and is filled by an X. Of course, if tube (set) S_k for $1 \leq k \leq 9$ does not have any DNA strand, then this means that the k^{th} square (position) is empty. Players that include a biological computer and his opponent can make a move by destructively changing one of the board positions from an empty content to an O (b^0) or an X (b^1).

For selecting the best move, it must have some way of analyzing the configuration of the board. It is very clear from tic-tac-toe that there are only eight ways to make three-in-a-row: three horizontally, three vertically, and two diagonally. Three *horizontal triplets* of making three-in-a-row are, respectively, (1 2 3), (4 5 6) and (7 8 9). Three *vertical triplets* are, respectively, (1 4 7), (2 5 8) and (3 6 9). Two *diagonal triplets* are, respectively, (1 5 9) and (3 5 7). This is to say that one of two player wins with that three of his symbols appear the same triplet. For example, the opponent wins with that three Os appear in the right diagonal triplet is (3 5 7), indicating the contents of elements *three*, *five*, and *seven* of a tic-tac-toe board are all Os.

Two *distinct* DNA strands (sequences) in [3-5, 7-9] are designed to minimize the possibility of unwanted binding and their length is θ base pairs. One represents the value “0” for a binary variable with a *bit* r and the other represents the value “1” for it. For the sake of convenience in our presentation, it is assumed that r^1 denotes the value of r to be 1, r^0 defines the value of r to be 0, and r defines the value of r to be 0 or 1. Bit r^0 is used to encode the result that indicates that there are three Os to make three-in-a-row, and bit r^1 is applied to encode the result that is that there are three Xs to make three-in-a-row. It is assumed that tube T_0 is used to store the result that is whether the contents of the board positions specified by eight triplets make three-in-a-row or not. If tube (set) T_0 contains r^0 (DNA strands), then this indicates that in the current configuration of the board there are no three Xs or three Os to make three-in-a-row. Similarly, if tube (set) T_0 contains r^1 (DNA strands), then this indicates that in the current configuration of the board there are three Xs or three Os to make three-in-a-row.

B. System Architecture of Playing a Tic-tac-toe

Now, let us look at the basic framework for playing the game. The function **Play-Tic-Tac-Toe**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) first offers to set a new, empty board as appropriate input. Then, it also offers the *opponent* the choice to go first, and then calls either **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) or **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to begin to play the game. In the function **Play-Tic-Tac-Toe**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), the first parameter that is tube T_0 stores the result that is whether the contents of the board positions specified by eight triplets make three-in-a-row or not. The second parameter to the tenth parameter that are, subsequently, tubes S_1 through S_9 store respectively the contents of nine squares (positions), and are all set to empty tubes in the function by means of calling **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).

Play-Tic-Tac-Toe(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(2) Discard(T_0).
(3) **If** (the opponent would like to go first) **Then**
    (4) **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
  Else
    (5) **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
  EndIf
EndFunction

Lemma 6-1: The function **Play-Tic-Tac-Toe**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) offers beginning of playing a tic-tac-toe.

Proof:

On the first execution of Step (1), it calls the function **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to set nine tubes to empty tubes. After it is completed, a new and empty board is obtained. Next, on the first execution of Step (2), it uses the *discard* operation to set tube T_0 to an empty tube. Next, from the execution of Step (3) if the opponent would like to go first, then the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is called by the execution of Step (4).

When the opponent goes first, the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) asks the opponent to type in a move and checks that the move is legal. The content of the board is updated by it and then the function **Computer-Move**(T_0,

S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is called. However, there are two special cases to cause not to call the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). The first special case is that if the opponent's move makes a three-in-a-row, the opponent has won and the game is over. The second special case is that if there are no empty spaces left on the board, the game has ended in a tie.

If the opponent does not want to go first, then the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is called from the execution of Step (5). When the computer goes first, the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) selects the best move. The content of the board is also updated by it and then the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is called. Similarly, there are two special cases to cause not to call the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). The first special case is that if the computer's move makes a three-in-a-row, the computer has won and the game is over. The second special case is that if there are no empty spaces left on the board, the game has ended in a tie. Therefore, it is at once inferred that the function **Play-Tic-Tac-Toe**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) offers to beginning of playing a tic-tac-toe. ■

C. Creating a New Tic-tac-toe Board for Playing a Tic-tac-toe

The following function (algorithm), **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is used to create a new tic-tac-toe board. The *first* parameter to the *ninth* parameter that are, subsequently, tubes S_1 through S_9 store respectively the contents of nine squares (positions), and are all set to empty tubes after the function **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is completed.

Make-Board(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **For** k = 1 **to** 9 **Step** 1
  (1a) Discard(S_k).
EndFor
EndFunction

Lemma 6-2: A new tic-tac-toe board can be created from the function **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).

Proof:

The function, **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is implemented by means of using the *discard* operation. It consists of one single loop. The single loop is applied to set the content of each square (position) to be empty. *Mathematical induction* is applied to complete the proof. When the value of the loop index variable, k, is equal to one, on the first execution of Step (1a) embedded in the loop, it uses the *discard* operation to set the content of the first square (position) to be empty. This is to say that the first square (position) on a new tic-tac-toe board is not occupied by players. Next, when the value of the loop index variable, k, is equal to l for $2 \leq l \leq 9 - 1$, on the l^{th} execution of Step (1a) embedded in the loop, it employs the *discard* operation to set the content of the l^{th} square (position) to be empty. This indicates that the l^{th} square (position) on a new tic-tac-toe board is not occupied by players. Next, when the value of the loop index variable, k, is equal to $(l + 1)$ for $2 \leq l \leq 9 - 1$, on the $(l + 1)^{th}$ execution of Step (1a) embedded in the loop, it applies the *discard* operation to set the content of the $(l + 1)^{th}$ square (position) to be empty. This is to say that the $(l + 1)^{th}$ square (position) on a new tic-tac-toe board is not occupied by players. Hence, it is at once inferred that a new tic-tac-toe board can be created from the function **Make-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). ■

D. The Strategy of the Move of the Opponent to Play a Tic-tac-toe

The following function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) offers the strategy of the move of the opponent to play a tic-tac-toe, calls the function **Read-A-Legal-Move**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that asks the opponent to type in a move, and checks that the move is legal and updates the board. Next, the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) calls the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). But there are two special cases where the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) should not be called. First, if the opponent's move makes a three-in-a-row, then the opponent has won and the game is over. Second, if there are no empty spaces left on the board, the game has ended in a tie. In the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), the first parameter T_0 contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are used to store the contents of nine squares (positions).

Opponent-Move(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **Read-A-Legal-Move**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(2) **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(3) **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(4) **If** (Detect(T_0) == *true*) **then**
(5) A string with 'You (the opponent) wins' is printed out.

(6) The execution of the function is terminated.

Else

(7) **If** ((Detect(S_1) == *true*) **AND** (Detect(S_2) == *true*) **AND** (Detect(S_3) == *true*) **AND** (Detect(S_4) == *true*) **AND** (Detect(S_5) == *true*) **AND** (Detect(S_6) == *true*) **AND** (Detect(S_7) == *true*) **AND** (Detect(S_8) == *true*) **AND** (Detect(S_9) == *true*)) **then**

(8) A string with 'The game has ended in a tie' is printed out.

(9) The execution of the function is terminated.

Else

(10) **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).

EndIf

EndIf

EndFunction

Lemma 6-3: The function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) offers the opponent to play the game.

Proof:

On the execution of Step (1), it calls the function **Read-A-Legal-Move**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that fills an O into the position selected by the opponent. Next, on the execution of Step (2), it calls the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that prints out the *current* configuration of the board after the opponent selected his move. Next, on the execution of Step (3), it calls the function **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that decides whether there are three Os to make a three-in-a-row or not. If there are three Os to make a three-in-a-row, then tube T_0 contains DNA sequences encoding r^1 that indicates that the condition is true. Otherwise, tube T_0 is an empty tube.

Next, on the execution of Step (4), if a *true* is returned, then a string with 'You (the opponent) wins' is printed out from the execution of Step (5) and the execution of the function is terminated from the execution of Step (6). Otherwise, if *nine detect* operations all returns a *true* from the execution of Step (7), then a string with 'The game has ended in a tie' is printed out from the execution of Step (8) and the execution of the function is terminated from the execution of Step (9). Otherwise, on the execution of Step (10) it calls the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that offers the computer to play the game. Therefore, it is at once inferred that the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) offers the opponent to play the game. ■

E. Reading a Legal Move from the Opponent to Play a Tic-tac-toe

Because nine positions in a board are subsequently numbered as one through nine, a legal move is an integer between one and nine such that the corresponding position in the board is empty. Therefore, the opponent can select one through nine as his move. The following function **Read-A-Legal-Move**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) reads a value from the opponent's selection (input) and judges whether it is a legal move or not. If not, the function **Read-A-Legal-Move**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) asks again that the opponent gives his new selection (move) and again reads another move. The opponent's selection is stored in one of tubes S_1 through S_9 that are subsequently the first parameter through the ninth parameter.

Read-A-Legal-Move(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)

(1) **For** j = 1 **to** 9 **Step** 1

  (2) The opponent's selection is read and is stored into an index variable k.

  (3) **If** (Detect(S_k) == *true*) **then**

    (4) Append-Tail(S_k, b^0).

    (5) The execution of the function is terminated.

  EndIf

EndFor

EndFunction

Lemma 6-4: The function **Read-A-Legal-Move**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) reads a legal move from the opponent's selection (input).

Proof:

Step (1) is a single loop and at most allows that the opponent selects his move nine times. On each execution of Step (2), the opponent's selection is read and is stored into an index variable k. Next, on each execution of Step (3), it uses the *detect* operation to judge whether the position selected by the opponent is not occupied or not. If a *true* is returned, then on each execution of Step (4) it appends a DNA sequence, encoding the value b^0, onto the end of every strand in tube S_k and this is to say that the corresponding square is occupied by the opponent and is filled by an O. Next, each execution of Step (5), the execution of the function is terminated.

Therefore, it is at once inferred that the function **Read-A-Legal-Move**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) reads a legal move from the opponent's selection (input). ■

F. Printing out the Configuration of the Board for Playing a Tic-tac-toe

Displaying the configuration of the board for playing a tic-tac-toe is a part of any tic-tac-toe, and also is a function to take a list of nine elements as input. Each element will be an X, an O, or an empty content. The following function, **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is used to print out the configuration of the board. Tubes S_1 through S_9 are subsequently the first parameter through the ninth parameter, and are used to store the content of nine elements.

Print-Board(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **For** j = 1 **to** 9 **Step** 3
  (2) **For** k = j **to** j + 2 **Step** 1
    (2a) **If** (Detect(S_k) == *false*) **then**
        (2b) A space and a string with ' | ' are printed out.
      Else
        (2c) S_k^{ON} = +(S_k, b^1) and S_k^{OFF} = –(S_k, b^1).
        (2d) **If** (Detect(S_k^{ON}) == *true*) **then**
            (2e) An X and a string with ' | ' are printed out.
            (2f) $S_k = \cup(S_k, S_k^{ON})$.
          Else
            (2g) An O and a string with ' | ' are printed out.
            (2h) $S_k = \cup(S_k, S_k^{OFF})$.
          EndIf
      EndIf
  EndFor
  (3) A string with '------------' is printed out if the value of k is less than seven.
  (4) A new line is printed out.
EndFor
EndFunction

Lemma 6-5: The *new* configuration of the board in a tic-tac-toe can be printed out from the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).

Proof:

The function, **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is implemented by means of using the *discard* operation, the *exact* operation and the *merge* operation. It consists of one nested loop. The nested loop is employed to print out the content of each square on a tic-tac-toe board. *Mathematical induction* is used to complete the proof. When the values of the two loop index variable, j and k, are, respectively, equal to one and j (one), on the first execution of Step (2a) embedded in the loop, it applies the *detect* operation to check whether the content of the first square (position) is empty or not. If a *false* is returned from the first execution of Step (2a), then a space and a string with ‘ | ‘ are printed out from the first execution of Step (2b). Otherwise, Step (2c) through Step (2h) is implemented.

Next, on the first execution of Step (2c), it uses the *extract* operation to form two test tubes, S_1^{ON} and S_1^{OFF} so that tube S_1 is an empty tube. The value encoded by DNA strands in tube S_1^{ON} is equal to b^1. The value encoded by DNA strands in tube S_1^{OFF} is equal to b^0. Next, on the first execution of Step (2d), it uses the *detect* operation to check whether the content of the first square (position) is an X or not. If a *true* is returned from the first execution of Step (2d), then from the first execution of Step (2e) an X and a string with ‘ | ‘ are printed out and from the first execution of Step (2f) the *merge* operation is used to pour the content of tube S_1^{ON} into tube S_1 so that tube S_1^{ON} is an empty tube. Otherwise, from the first execution of Step (2g) an O and a string with ‘ | ‘ are printed out and from the first execution of Step (2h) the *merge* operation is used to pour the content of tube S_1^{OFF} into tube S_1 so that tube S_1^{OFF} is an empty tube.

Next, when the values of the two loop index variable, j and k, are, respectively, equal to one and $j + 2$ (3), the content of the third square (position) is printed and from the first execution of Step (3) and Step (4) a string with ‘------------’ is printed out and a new line is also printed out. Similarly, when the values of the two loop index variable, j and k, are, respectively, equal to seven and $j + 2$ (9), the content of the nine square (position) is printed and from the third execution of Step (3) and Step (4) a new line is printed out. Therefore, it is at once derived that the *new* configuration of the board in a tic-tac-toe can be printed out from the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). ■

G. Checking Whether the Contents of Eight Triplets Make Three-in-a-row

For fully analyzing a board we must look at all eight triplets. The following function, **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is employed to test whether in the contents of the board positions specified by all eight triplets there are three Xs or three Os to make three-in-a-row or not. Tube T_0 that is the first parameter is initialized to an empty tube. Other nine parameters store the content of each square. Notice that if player O (the opponent) ever gets three in a row, from the function **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) r^1 that is encoded by DNA strands in tube T_0 is obtained. Similarly, if player X (the computer) manages to get three in a row, from the function **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) r^1 that is encoded by DNA strands in tube T_0 is also obtained.

Test-three-in-a-row-for-eight-triplets(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **For** k = 1 **to** 9 **Step** 3
    (1a) **Check-One-Triplet**(T_0, S_k, S_{k+1}, S_{k+2}).
    (1b) **If** (*Detect*(T_0) == *true*) **Then**
            (1c) The execution of the function is terminated.
    EndIf
EndFor
(2) **For** k = 1 **to** 3 **Step** 1
    (2a) **Check-One-Triplet**(T_0, S_k, S_{k+3}, S_{k+6}).
    (2b) **If** (*Detect*(T_0) == *true*) **Then**
            (2c) The execution of the function is terminated.
    EndIf
EndFor
(3) **Check-One-Triplet**(T_0, S_1, S_5, S_9).
    (4) **If** (*Detect*(T_0) == *true*) **Then**
            (5) The execution of the function is terminated.
    EndIf
(6) **Check-One-Triplet**(T_0, S_3, S_5, S_7).
    (7) **If** (*Detect*(T_0) == *true*) **Then**
            (8) The execution of the function is terminated.
    EndIf
EndFunction

Lemma 6-6: Testing whether in the contents of the board positions specified by all eight triplets there are three Xs or three Os to make three-in-a-row or not can be done from the function **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).

Proof:

The function, **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7,

S_8, S_9), is implemented by means of using the *extract* operation, the *detect* operation, the *append-tail* operation and the *merge* operation. *Mathematical induction* is used to complete the proof. Step (1) consists of one single loop, and is used to test whether in the contents of the board positions specified by *three horizontal* triplets with positions (1 2 3), (4 5 6) and (7 8 9) there are three Xs or three Os to make three-in-a-row or not. When the value of the loop index variable, k, is equal to one, on the first execution of Step (1a), it calls the function, **Check-One-Triplet**(T_0, S_1, S_2, S_3). After the first execution of Step (1a) is implemented, if the contents of the board positions specified by the *first horizontal* triplet with positions (1 2 3) make three-in-a-row, then tube T_0 contains DNA strands encoding r^1. Otherwise, tube T_0 still is an empty tube. Next, on the first execution of Step (1b), if a *true* is returned from the *detect* operation, then the execution of the function is terminated from the first execution of Step (1c) and the contents of the board positions specified by the *first horizontal* triplet with positions (1 2 3) make three-in-a-row. Otherwise, the resting operations will continue to be executed.

Similarly, when the value of the loop index variable, k, is equal to four, from the second execution of Step (1a), the function, **Check-One-Triplet**(T_0, S_4, S_5, S_6) is called and implemented. If the contents of the board positions specified by the *second horizontal* triplet with positions (4 5 6) make three-in-a-row, then tube T_0 contains DNA strands encoding r^1. Otherwise, tube T_0 still is an empty tube. Next, on the second execution of Step (1b), if a *true* is returned from the *detect* operation, then the execution of the function is terminated from the second execution of Step (1c) and the contents of the board positions specified by the *second horizontal* triplet with positions (4 5 6) make three-in-a-row. Otherwise, the resting operations will continue to be executed.

Next, when the value of the loop index variable, k, is equal to seven, from the third execution of Step (1a), the function, **Check-One-Triplet**(T_0, S_7, S_8, S_9) is called and implemented. If the contents of the board positions specified by the *third horizontal* triplet with positions (7 8 9) make three-in-a-row, then tube T_0 contains DNA strands encoding r^1. Otherwise, tube T_0 still is an empty tube. Next, on the third execution of Step (1b), if a *true* is returned from the *detect* operation, then the execution of the function is terminated from the third execution of Step (1c) and the contents of the board positions specified by the *third horizontal* triplet with positions (7 8 9) make three-in-a-row. Otherwise, the resting operations will continue to be executed.

Next, the same operations that are implemented by the first execution through the third execution of Step (2a) through Step (2c) judge whether the contents of the board positions specified by *three vertical* triplets with positions (1 4 7), (2 5 8) and (3 6 9) make three-in-a-row or not. If one of *three vertical* triplets makes three-in-a-row, then the execution of the function is terminated. Otherwise, the resting operations will continue to be executed.

Next, the same operations that are implemented by the first execution of Step (3) through Step (8) judge whether the contents of the board positions specified by *two diagonal* triplets with positions (1 5 9) and (3 5 7) make three-in-a-row or not. If one of *two diagonal* triplets makes three-in-a-row, then the execution of the function is terminated and one player wins the game. Otherwise, the game will continue to be played. Therefore, it is at once derived that testing whether in the contents of the board positions specified by all eight triplets there are three Xs or three Os to make three-in-a-row or not can be done from the function **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). ■

H. Testing Whether the Contents of One of Eight Triplets Make Three-in-a-row

The following function, **Check-One-Triplet**(T_0, U_0, V_0, W_0) is used to check whether the contents of the board positions specified by that triplet make three-in-a-row or not. Tube T_0 that is the first parameter is used to store the result that indicates whether there are three Xs or three Os to make three-in-a-row or not. The second, third and fourth parameters U_0, V_0 and W_0 are all empty tubes, and they are used to subsequently store the contents of three elements in that triplet.

Check-One-Triplet(T_0, U_0, V_0, W_0)
(1) $U_0^{ON} = +(U_0, b^1)$ and $U_0^{OFF} = -(U_0, b^1)$.
(2) $V_0^{ON} = +(V_0, b^1)$ and $V_0^{OFF} = -(V_0, b^1)$.
(3) $W_0^{ON} = +(W_0, b^1)$ and $W_0^{OFF} = -(W_0, b^1)$.
(4) **If** ((Detect(U_0^{ON}) == *true*) **AND** (Detect(V_0^{ON}) == *true*) **AND** (Detect(W_0^{ON}) == *true*)) **Then**
    (5) Append-Tail(T_0, r^1).
(6) **ElseIf** ((Detect(U_0^{OFF}) == *true*) **AND** (Detect(V_0^{OFF}) == *true*) **AND** (Detect(W_0^{OFF}) == *true*)) **Then**
    (7) Append-Tail(T_0, r^0).
EndIf
EndFunction

Lemma 6-7: Checking whether the contents of the board positions specified by that triplet make three-in-a-row or not can be done from the function **Check-One-Triplet**(T_0, U_0, V_0, W_0).

Proof:

The function, **Check-One-Triplet**(T_0, U_0, V_0, W_0), is implemented by means of using the *exact* operation and the *detect* operation. On each execution of Step (1), it uses the *extract* operation to form two test tubes, U_0^{ON} and U_0^{OFF} so that tube U_0 is an empty tube. The value encoded by DNA strands in tube U_0^{ON} is equal to b^1. The value encoded by DNA strands in tube U_0^{OFF} is equal to b^0. This is to say that an X in the first element of that triplet appears in tube U_0^{ON} or an O in the *first* element of that triplet appears in tube U_0^{OFF}.

Next, on each execution of Step (2), it also applies the *extract* operation to form two test tubes, V_0^{ON} and V_0^{OFF} so that tube V_0 is an empty tube. The value encoded by DNA strands in tube V_0^{ON} is equal to b^1. The value encoded by DNA strands in tube V_0^{OFF} is equal to b^0. This indicates that an X in the *second* element of that triplet appears in tube V_0^{ON} or an O in the *second* element of that triplet appears in tube V_0^{OFF}. Next, on each execution of Step (3), it also employs the *extract* operation to form two test tubes, W_0^{ON} and W_0^{OFF} so that tube W_0 is an empty tube. The value encoded by DNA strands in tube W_0^{ON} is equal to b^1. The value encoded by DNA strands in tube W_0^{OFF} is equal to b^0. This implies that an X in the *third* element of that triplet appears in tube W_0^{ON} or an O in the *third* element of that triplet appears in tube W_0^{OFF}.

Next, on each execution of Step (4), it uses six *detect* operations to check whether the content of each element in that triplet is an X, an O or empty or not. If the *front three* detect operations all return true, then this is to say that three Xs make three-in-a-row. If the *last three* detect operations all return true, then this indicates that three Os make three-in-a-row. Hence, on each execution of Step (5), it appends a DNA sequence, encoding the value r^1, onto the end of every strand in tube T_0 and this indicates that the contents of three elements in that triplet make three-in-a-row. Therefore, it is at once derived that checking whether the contents of the board positions specified by that triplet make three-in-a-row or not can be done from the function **Check-One-Triplet**(T_0, U_0, V_0, W_0). ■

I. The Strategies of the Movement of the Computer

Because the analysis of selecting the best move to two players is more complex, we shall use biological operations and DNA strands to learn how to make use of the

Newell-Simon method in which the very good strategies are provided. The function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is similar to that function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), except the player is X instead of O, and instead of reading a move from the opponent's selection (input), how learning a good strategy in the Newell-Simon method to the computer is proposed. Because the game is a draw when viewed from a game-theoretic standpoint, *good* means here a strategy that will guarantee a draw and that will give the opponent as many opportunities as possible of making a losing mistake. The function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) calls several other functions to choose the best move and to update the configuration of the board. Next, the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) calls the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). But there are two special cases where the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) should not be called. First, if the computer's move makes a three-in-a-row, then the computer has won and the game is over. Second, if there are no empty spaces left on the board, the game has ended in a tie. In the function **Computer-Move(**T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), the first parameter T_0 contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are used to store the contents of nine squares (positions).

Computer-Move(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **If** (**Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)) **then**
    (1a) **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(2) **Else If** (**Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)) **then**
    (2a) **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(3) **Else If** (**Finding-Intersetion**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)) **then**
    (3a) **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(4) **Else If** (**Finding-Center**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)) **then**
    (4a) **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(5) **Else If** (**Opponent-on-Side**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)) **then**
    (5a) **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(6) **Else If** (**Opponent-on-Corner**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)) **then**
    (6a) **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
EndIf
(7) **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).
(8) **If** (Detect(T_0) == *true*) **then**

(8a) A string with ‘I (the computer) wins’ is printed out.

(8b) The execution of the function is terminated.

Else

(9) **If** ((Detect(S_1) == *true*) **AND** (Detect(S_2) == *true*) **AND** (Detect(S_3) == *true*) **AND** (Detect(S_4) == *true*) **AND** (Detect(S_5) == *true*) **AND** (Detect(S_6) == *true*) **AND** (Detect(S_7) == *true*) **AND** (Detect(S_8) == *true*) **AND** (Detect(S_9) == *true*)) **then**

(9a) A string with ‘The game has ended in a tie’ is printed out.

(9b) The execution of the function is terminated.

Else

(9c) **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9).

EndIf

EndIf

EndFunction

Lemma 6-8: The function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to make use of the good strategies in the Newell-Simon method for winning the game.

Proof:

On the first execution of Step (1), it calls the function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to find whether there is a line with two of the computer’s marks and one blank or not. If the condition is satisfied, then an X is filled into the *blank* square and a *true* is returned. If a *true* is returned, then on the first execution of Step (1a) it calls the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that prints out the *current* configuration of the board after the computer selected his move. Otherwise, on the first execution of Step (2) it invokes the function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to check whether there is a line with two of the opponent’s marks and one blank or not. If the condition is satisfied, then an X is filled into the blank square to protect that the opponent wins the game, and a *true* is returned.

If a *true* is returned, then on the first execution of Step (2a) it calls the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that prints out the *current* configuration of the board after the computer selected his move. Otherwise, on the first execution of Step (3) it calls the function **Finding-Intersetion**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to find whether there are two lines, each with one of the computer's mark and two blanks, intersecting in a single blank square. If the condition is satisfied, then an X is filled into the single blank square to create two lines in which each line has two computer's marks and one blank, thus forking the opponent, and a true is returned.

If a true is returned, then on the first execution of Step (3a) it calls the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that prints out the *current* configuration of the board after the computer selected his move. Otherwise, on the first execution of Step (4) it calls the function **Finding-Center**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to test whether in the board the *fifth* square that is called a *center* square is empty or not. If the condition is satisfied, then an X is filled into the *center* square and a *true* is returned.

If a true is returned, then on the first execution of Step (4a) it calls the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that prints out the *current* configuration of the board after the computer selected his move. Otherwise, on the first execution of Step (5) it calls the function **Opponent-on-Side**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to check whether in the board the *second* square, the *fourth* square, the *sixth* square or the *eighth* square that are all called *side* squares are occupied by the opponent or not and to check whether the *eighth* square, the *sixth* square, the *fourth* square or the *second* square are empty or not. If the condition is satisfied, then an X is filled into the *opposite* of each *side* square and a *true* is returned.

If a true is returned, then on the first execution of Step (5a) it calls the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that prints out the *current* configuration of the board after the computer selected his move. Otherwise, on the first execution of Step (6) it calls the function **Opponent-on-Corner**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to check whether in the board the *first* square, the *third* square, the *seventh* square or the *ninth* square are occupied by the opponent or not and to find whether the *opposite* of the *first* square, the *third* square, the *seventh* square or the *ninth* position is empty or not. If the condition is satisfied, then an X is filled into the *opposite* of the *corner* square and a *true* is returned. If a true is returned, then on the first execution of Step (6a) it calls the function **Print-Board**(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that prints out the *current* configuration of the board after the computer selected his move.

Next, on the first execution of Step (7), it calls the function **Test-three-in-a-row-for-eight-triplets**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) to decide whether there are three Xs to make a three-in-a-row or not. If there are three Xs to make a three-in-a-row, then tube T_0 contains DNA sequences encoding r^1 that indicates that the condition is true. Otherwise, tube T_0 is an empty tube. Next, on the first execution of Step (8), if a *true* is returned, then a string with ‘I (the computer) wins’ is printed out from the first execution of Step (8a) and the execution of the function is terminated from the first execution of Step (8b). Otherwise, if *nine detect* operations all returns a *true* from the first execution of Step (9), then a string with ‘The game has ended in a tie’ is printed out from the first execution of Step (9a) and the execution of the function is terminated from the first execution of Step (9b). Otherwise, on the first execution of Step (9c) it calls the function **Opponent-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) that offers the opponent to play the game. Therefore, it is at once inferred that the function **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to make use of the good strategies in the Newell-Simon method for winning the game. ■

J. Biological Algorithms of the Winning Strategies to the Movement of the Computer

In the Newell-Simon method the *first* strategy is if one player (a computer) finds that there is a line with two of the computer’s marks and one blank, then an X is filled into the *blank* square and a three-in-a-row is made. Therefore, the following function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is applied to find whether there is a line with two of the computer’s marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square and a *true* is returned. Otherwise, a *false* is returned. The first parameter T_0 contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are used to store the contents of nine squares (positions).

Computer-Winning-Strategy(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **For** k = 1 **to** 9 **Step** 3
 (1a) **If** (**Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_k, S_{k+1}, S_{k+2})) **Then**
  (1b) Return a *true* to the caller and terminate the execution of the function.
 EndIf
EndFor
(2) **For** k = 1 **to** 3 **Step** 1
 (2a) **If** (**Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_k, S_{k+3}, S_{k+6})) **Then**
  (2b) Return a *true* to the caller and terminate the execution of the function.
 EndIf
EndFor

(3) **If** (**Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_1, S_5, S_9)) **Then**

(3a) Return a *true* to the caller and terminate the execution of the function.

EndIf

(4) **If** (**Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_3, S_5, S_7)) **Then**

(4a) Return a *true* to the caller and terminate the execution of the function.

EndIf

(5) Return a *false* to the caller and terminate the execution of the function.

EndFunction

Lemma 6-9: The function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to make use of the *first* strategy in the Newell-Simon method to select one movement of winning the game.

Proof:

Step (1) is one single loop and is used to test whether three *horizontal* lines (1 2 3), (4 5 6) and (7 8 9) contain two of the computer's marks and one blank or not. On the first execution of Step (1a), it calls the function **Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_k, S_{k+1}, S_{k+2}) to check whether the *first* horizontal line (1 2 3) contains two of the computer's marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square and a *true* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). If a true is returned, then on the first execution of Step (1b) it returns a true to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Otherwise, a *false* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the same processing from the *second* execution through the *third* execution of Steps (1a) and (1b) is used to check whether the *second horizontal* line (4 5 6) and the *third horizontal* line (7 8 9) include two of the computer's marks and one blank or not.

Next, Step (2) is one single loop and is employed to judge whether three *vertical* lines (1 4 7), (2 5 8) and (3 6 9) contain two of the computer's marks and one blank or not. On the first execution of Step (2a), it calls the function **Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_k, S_{k+3}, S_{k+6}) to decide whether the *first vertical* line (1 4 7) contains two of the computer's marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square and a *true* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). If a true is returned, then on the first execution of Step (2b) it returns a true to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Otherwise, a *false* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the same processing from the *second* execution through the *third* execution of Steps (2a) and (2b) is used to check whether the *second vertical* line (2 5 8) and the *third vertical* line (3 6 9) include two of the computer's marks and one blank or not.

Next, On the first execution of Step (3), it calls the function **Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_1, S_5, S_9) to judge whether the *first diagonal* line (1 5 9) contains two of the computer's marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square and a *true* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). If a true is returned, then on the first execution of Step (3a) it returns a true to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Otherwise, a *false* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the same processing from the *first* execution of Steps (4) and (4a) is used to check whether the *second diagonal* line (3 5 7) includes two of the computer's marks and one blank or not. If the condition above is not satisfied, then from the first execution of Step (5) a false is returned to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Therefore, it is at once inferred from the statements above that the function **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to make use of the *first* strategy in the Newell-Simon method to select one movement of winning the game. ■

K. Biological Algorithms of Finding a Line with Two of the Computer's Marks and One Blank

The following function, **Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_d, S_e, S_f), learns how to find a line with two of the computer's Marks and one blank. If the line satisfying the condition above is found, then an X is filled into the *blank* square and a *true* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). Because eight lines (*triplets*) are respectively (1 2 3), (4 5 6), (7 8 9), (1 4 7), (2 5 8), (3 6 9), (1 5 9) and (3 5 7), the first parameter through the three parameter (d, e, f) is respectively the first element, the second element and the third element in one of eight lines. Tube T_0 that is the fourth parameter contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes (S_d, S_e, S_f) that are the fifth parameter through the seventh parameter are subsequently used to store the contents of three squares in one of eight lines.

Find-A-Line-With-Two-Xs-One-Blank(T_0, S_d, S_e, S_f)

(1) $S_d^{ON} = +(S_d, b^1)$ and $S_d^{OFF} = -(S_d, b^1)$.

(2) $S_e^{ON} = +(S_e, b^1)$ and $S_e^{OFF} = -(S_e, b^1)$.

(3) $S_f^{ON} = +(S_f, b^1)$ and $S_f^{OFF} = -(S_f, b^1)$.

(4) **If** ((Detect(S_d^{ON}) == *true*) **AND** (Detect(S_e^{ON}) == *true*) **AND** (Detect(S_f^{ON}) == *false*) **AND** (Detect(S_f^{OFF}) == *false*)) **Then**

(4a) Append-Tail(S_f, b^1).

(4b) $S_d = \cup(S_d^{ON}, S_d^{OFF})$ and $S_e = \cup(S_e^{ON}, S_e^{OFF})$.

(4c) Return a *true* to the caller and terminate the execution of the function.

(5) **Else If** ((Detect(S_d^{ON}) == *true*) **AND** (Detect(S_f^{ON}) == *true*) **AND** (Detect(S_e^{ON}) == *false*) **AND** (Detect(S_e^{OFF}) == *false*)) **Then**

(5a) Append-Tail(S_e, b^1).

(5b) $S_d = \cup(S_d^{ON}, S_d^{OFF})$ and $S_f = \cup(S_f^{ON}, S_f^{OFF})$.

(5c) Return a *true* to the caller and terminate the execution of the function.

(6) **Else If** ((Detect(S_e^{ON}) == *true*) **AND** (Detect(S_f^{ON}) == *true*) **AND** (Detect(S_d^{ON}) == *false*) **AND** (Detect(S_d^{OFF}) == *false*)) **Then**

(6a) Append-Tail(S_d, b^1).

(6b) $S_e = \cup(S_e^{ON}, S_e^{OFF})$ and $S_f = \cup(S_f^{ON}, S_f^{OFF})$.

(6c) Return a *true* to the caller and terminate the execution of the function.

(7) **Else**

(7a) $S_d = \cup(S_d^{ON}, S_d^{OFF})$, $S_e = \cup(S_e^{ON}, S_e^{OFF})$ and $S_f = \cup(S_f^{ON}, S_f^{OFF})$.

(7b) Return a *false* to the caller and terminate the execution of the function.

EndIf

EndFunction

Lemma 6-10: The function **Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_d, S_e, S_f) learns how to find a line with two of the computer's Marks and one blank.

Proof:

On each execution of Step (1) through Step (3), they respectively use three *extract* operations to form six test tubes, S_d^{ON}, S_d^{OFF}, S_e^{ON}, S_e^{OFF}, S_f^{ON} and S_f^{OFF}. DNA strands in tubes S_d^{ON}, S_e^{ON} and S_f^{ON} encodes b^1 representing an *X*, and DNA strands in tubes S_d^{OFF}, S_e^{OFF} and S_f^{OFF} encodes b^0 representing an *O*. Next, on each execution of Step (4), it uses four *detect* operations to test whether the first square, the second square and the third square in a line that is one of eight triplets are subsequently an *X*, an *X* and a blank or not. If the condition above is satisfied by each *detect* operation, then an *X* is filled into the blank square from each execution of Step (4a), tubes S_d^{ON} and S_d^{OFF} are poured into tube S_d from each execution of Step (4b), tubes S_e^{ON} and S_e^{OFF} are poured into tube S_e from each execution of Step (4b) and from each execution of Step (4c) it returns a *true* to the caller and the execution of the function is terminated.

Otherwise, next, on each execution of Step (5), it also applies four *detect* operations to check whether the first square, the second square and the third square in a line that is one of eight triplets are subsequently an *X*, a blank and an *X* or not. If the condition above is satisfied by each *detect* operation, then an *X* is filled into the blank square from each execution of Step (5a), tubes S_d^{ON} and S_d^{OFF} are poured into tube S_d from each execution of Step (5b), tubes S_f^{ON} and S_f^{OFF} are poured into tube S_f from each execution of Step (5b) and from each execution of Step (5c) it returns a *true* to the caller and the execution of the function is terminated.

Otherwise, next, on each execution of Step (6), it uses four *detect* operations to check whether the first square, the second square and the third square in a line that is one of eight triplets are subsequently a blank, an *X* and an *X* or not. If the condition above is satisfied by each *detect* operation, then an *X* is filled into the blank square from each execution of Step (6a), tubes S_e^{ON} and S_e^{OFF} are poured into tube S_e from each execution of Step (6b), tubes S_f^{ON} and S_f^{OFF} are poured into tube S_f from each execution of Step (6b) and from each execution of Step (6c) it returns a *true* to the caller and the execution of the function is terminated.

Otherwise, next, on each execution of Step (7a), tubes S_d^{ON} and S_d^{OFF} are poured into tube S_d, tubes S_e^{ON} and S_e^{OFF} are poured into tube S_e, tubes S_f^{ON} and S_f^{OFF} are poured into tube S_f and from each execution of Step (7b) it returns a *false* to the caller and the execution of the function is terminated. Therefore, it is at once inferred from the statements above that the function **Find-A-Line-With-Two-Xs-One-Blank**(T_0, S_d, S_e, S_f) learns how to find a line with two of the computer's Marks and one blank. ■

M. Biological Algorithms of Protecting the Opponent That Wins the Game

In the Newell-Simon method the *second* strategy is if one player (a computer) checks that there is a line with two of the opponent's marks and one blank, then an X is filled into the blank square to protect that the opponent wins the game. Therefore, the following function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is used to find whether there is a line with two of the opponent's marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square to protect that the opponent wins the game and a *true* is returned. Otherwise, a *false* is returned. The first parameter T_0 contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are used to store the contents of nine squares (positions).

Opponent-Winning-Strategy(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **For** k = 1 **to** 9 **Step** 3
(1a) **If** (**Find-A-Line-With-Two-Os-One-Blank**(k, $k + 1$, $k + 2$, T_0, S_k, S_{k+1}, S_{k+2})) **Then**
(1b) Return a *true* to the caller and terminate the execution of the function.
EndIf
EndFor
(2) **For** k = 1 **to** 3 **Step** 1
(2a) **If** (**Find-A-Line-With-Two-Os-One-Blank**(k, $k + 3$, $k + 6$, T_0, S_k, S_{k+3}, S_{k+6})) **Then**
(2b) Return a *true* to the caller and terminate the execution of the function.
EndIf
EndFor
(3) **If** (**Find-A-Line-With-Two-Os-One-Blank**(1, 5, 9, T_0, S_1, S_5, S_9)) **Then**
(3a) Return a *true* to the caller and terminate the execution of the function.
EndIf
(4) **If** (**Find-A-Line-With-Two-Os-One-Blank**(3, 5, 7, T_0, S_3, S_5, S_7)) **Then**
(4a) Return a *true* to the caller and terminate the execution of the function.
EndIf
(5) Return a *false* to the caller and terminate the execution of the function.
EndFunction

Lemma 6-11: The function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to protect that the opponent wins the game.

Proof:

Step (1) is one single loop and is employed to judge whether three *horizontal* lines (1 2 3), (4 5 6) and (7 8 9) consists of two of the opponent's marks and one blank or not. On the first execution of Step (1a), it calls the function **Find-A-Line-With-Two-Os-One-Blank**(k, $k + 1$, $k + 2$, T_0, S_k, S_{k+1}, S_{k+2}) to test whether the *first* horizontal line (1 2 3) includes two of the opponent's marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square to protect that the opponent wins the game and a *true* is returned to the caller **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). If a true is returned, then on the first execution of Step (1b) it returns a true to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Otherwise, a *false* is returned to the caller **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the same processing from the *second* execution through the *third* execution of Steps (1a) and (1b) is applied to decide whether the *second horizontal* line (4 5 6) and the *third horizontal* line (7 8 9) include two of the opponent's marks and one blank or not.

Next, Step (2) is one single loop and is used to test whether three *vertical* lines (1 4 7), (2 5 8) and (3 6 9) contain two of the opponent's marks and one blank or not. On the first execution of Step (2a), it calls the function **Find-A-Line-With-Two-Os-One-Blank**(k, $k + 3$, $k + 6$, T_0, S_k, S_{k+3}, S_{k+6}) to judge whether the *first vertical* line (1 4 7) consists of two of the opponent's marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square to protect that the opponent wins the game and a *true* is returned to the caller **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). If a true is returned, then on the first execution of Step (2b) it returns a true to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Otherwise, a *false* is returned to the caller **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the same processing from the *second* execution through the *third* execution of Steps (2a) and (2b) is used to check whether the *second vertical* line (2 5 8) and the *third vertical* line (3 6 9) contain two of the opponent's marks and one blank or not.

Next, On the first execution of Step (3), it calls the function **Find-A-Line-With-Two-Os-One-Blank**(1, 5, 9, T_0, S_1, S_5, S_9) to judge whether the *first diagonal* line (1 5 9) includes two of the opponent's marks and one blank or not. If the condition above is satisfied, then an X is filled into the *blank* square to protect that the opponent wins the game and a *true* is returned to the caller **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). If a true is returned, then on the first execution of Step (3a) it returns a true to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Otherwise, a *false* is returned to the caller **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the same processing from the *first* execution of Steps (4) and (4a) is used to check whether the *second diagonal* line (3 5 7) includes two of the opponent's marks and one blank or not. If the condition above is not satisfied, then from the first execution of Step (5) a false is returned to the caller **Computer-Move**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) and the execution of the function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) is terminated. Therefore, it is at once inferred from the statements above that the function **Opponent-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to protect that the opponent wins the game. ■

N. Biological Algorithms of Finding a Line with Two of the Opponent's Marks and One Blank

The following function, **Find-A-Line-With-Two-Os-One-Blank**(d, e, f, T_0, S_d, S_e, S_f), learns how to find a line with two of the opponent's Marks and one blank. If the line satisfying the condition above is found, then an X is filled into the *blank* square to protect that the opponent wins the game and a *true* is returned to the caller **Computer-Winning-Strategy**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9). Since eight lines (*triplets*) are respectively (1 2 3), (4 5 6), (7 8 9), (1 4 7), (2 5 8), (3 6 9), (1 5 9) and (3 5 7), the first parameter through the three parameter (d, e, f) is respectively the first element, the second element and the third element in one of eight lines. DNA strands in tube T_0 that is the fourth parameter encode the result of predicating whether there is any a three-in-a-row or not. Tubes (S_d, S_e, S_f) that are the fifth parameter through the seventh parameter are subsequently used to store the contents of three squares in one of eight lines.

Find-A-Line-With-Two-Os-One-Blank(d, e, f, T_0, S_d, S_e, S_f)
(1) $S_d^{ON} = +(S_d, b^0)$ and $S_d^{OFF} = -(S_d, b^0)$.
(2) $S_e^{ON} = +(S_e, b^0)$ and $S_e^{OFF} = -(S_e, b^0)$.
(3) $S_f^{ON} = +(S_f, b^0)$ and $S_f^{OFF} = -(S_f, b^0)$.

(4) **If** ((Detect(S_d^{ON}) == *true*) **AND** (Detect(S_e^{ON}) == *true*) **AND** (Detect(S_f^{ON}) == *false*) **AND** (Detect(S_f^{OFF}) == *false*)) **Then**

(4a) Append-Tail(S_f, b^1).

(4b) $S_d = \cup(S_d^{ON}, S_d^{OFF})$ and $S_e = \cup(S_e^{ON}, S_e^{OFF})$.

(4c) Return a *true* to the caller and terminate the execution of the function.

(5) **Else If** ((Detect(S_d^{ON}) == *true*) **AND** (Detect(S_f^{ON}) == *true*) **AND** (Detect(S_e^{ON}) == *false*) **AND** (Detect(S_e^{OFF}) == *false*)) **Then**

(5a) Append-Tail(S_e, b^1).

(5b) $S_d = \cup(S_d^{ON}, S_d^{OFF})$ and $S_f = \cup(S_f^{ON}, S_f^{OFF})$.

(5c) Return a *true* to the caller and terminate the execution of the function.

(6) **Else If** ((Detect(S_e^{ON}) == *true*) **AND** (Detect(S_f^{ON}) == *true*) **AND** (Detect(S_d^{ON}) == *false*) **AND** (Detect(S_d^{OFF}) == *false*)) **Then**

(6a) Append-Tail(S_d, b^1).

(6b) $S_e = \cup(S_e^{ON}, S_e^{OFF})$ and $S_f = \cup(S_f^{ON}, S_f^{OFF})$.

(6c) Return a *true* to the caller and terminate the execution of the function.

(7) **Else**

(7a) $S_d = \cup(S_d^{ON}, S_d^{OFF})$, $S_e = \cup(S_e^{ON}, S_e^{OFF})$ and $S_f = \cup(S_f^{ON}, S_f^{OFF})$.

(7b) Return a *false* to the caller and terminate the execution of the function.

EndIf

EndFunction

Lemma 6-12: The function **Find-A-Line-With-Two-Os-One-Blank**($d, e, f, T_0, S_d, S_e, S_f$) learns how to find a line with two of the opponent's Marks and one blank to protect that the opponent wins the game.

Proof:

On each execution of Step (1) through Step (3), they respectively use three *extract* operations to form six test tubes, S_d^{ON}, S_d^{OFF}, S_e^{ON}, S_e^{OFF}, S_f^{ON} and S_f^{OFF}. DNA strands in tubes S_d^{ON}, S_e^{ON} and S_f^{ON} encodes b^0 representing an *O*, and DNA strands in tubes S_d^{OFF}, S_e^{OFF} and S_f^{OFF} encodes b^1 representing an *X*. Next, on each execution of Step (4), it uses four *detect* operations to test whether the first square, the second square and the third square in a line that is one of eight triplets are subsequently an *O*, an *O* and a blank or not. If the condition above is satisfied by each *detect* operation, then an *X* is filled into the blank square from each execution of Step (4a), tubes S_d^{ON} and S_d^{OFF} are poured into tube S_d from each execution of Step (4b), tubes S_e^{ON} and S_e^{OFF} are poured into tube S_e from each execution of Step (4b) and from each execution of Step (4c) it returns a *true* to the caller and the execution of the function is terminated.

Otherwise, next, on each execution of Step (5), it also applies four *detect* operations to check whether the first square, the second square and the third square in a line that is one of eight triplets are subsequently an *O*, a blank and an *O* or not. If the condition above is satisfied by each *detect* operation, then an *X* is filled into the blank square from each execution of Step (5a), tubes S_d^{ON} and S_d^{OFF} are poured into tube S_d from each execution of Step (5b), tubes S_f^{ON} and S_f^{OFF} are poured into tube S_f from each execution of Step (5b) and from each execution of Step (5c) it returns a *true* to the caller and the execution of the function is terminated.

Otherwise, next, on each execution of Step (6), it applies four *detect* operations to check whether the first square, the second square and the third square in a line that is one of eight triplets are subsequently a blank, an *O* and an *O* or not. If the condition above is satisfied by each *detect* operation, then an *X* is filled into the blank square from each execution of Step (6a), tubes S_e^{ON} and S_e^{OFF} are poured into tube S_e from each execution of Step (6b), tubes S_f^{ON} and S_f^{OFF} are poured into tube S_f from each execution of Step (6b) and from each execution of Step (6c) it returns a *true* to the caller and the execution of the function is terminated.

Otherwise, next, on each execution of Step (7a), tubes S_d^{ON} and S_d^{OFF} are poured into tube S_d, tubes S_e^{ON} and S_e^{OFF} are poured into tube S_e, tubes S_f^{ON} and S_f^{OFF} are poured into tube S_f and from each execution of Step (7b) it returns a *false* to the caller and the execution of the function is terminated. Therefore, it is at once inferred from the statements above that the function **Find-A-Line-With-Two-Os-One-Blank**(d, e, f, T_0, S_d, S_e, S_f) learns how to find a line with two of the opponent's marks and one blank to protect that the opponent wins the game. ■

O. Biological Algorithms of Finding That There Are Two Lines with One of the Computer's Marks and Two Blank and Intersecting in a Single Blank Square

The following function, **Finding-Intersetion**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is used to check whether there are two lines, each with one of the computer's mark and two blanks, intersecting in a single blank square or not. If the condition is satisfied, then an X is filled into the single blank square to create two lines in which each line has two computer's marks and one blank, thus forking the opponent and a *true* is returned. Otherwise, a *false* is returned. The first parameter T_0 contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are used to store the contents of nine squares (positions).

Finding-Intersetion(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)

(1) **If** (**Intersection-In-Two-Lines**(S_1, S_2, S_3, S_4, S_7)) **Then**

(1a) Return a *true* to the caller and terminate the execution of the function.

(2) **If** (**Intersection-In-Two-Lines**(S_1, S_2, S_3, S_5, S_9)) **Then**

(2a) Return a *true* to the caller and terminate the execution of the function.

(3) **If** (**Intersection-In-Two-Lines**(S_1, S_4, S_7, S_5, S_9)) **Then**

(3a) Return a *true* to the caller and terminate the execution of the function.

(4) **ElseIf** (**Intersection-In-Two-Lines**(S_3, S_1, S_2, S_6, S_9)) **Then**

(4a) Return a *true* to the caller and terminate the execution of the function.

(5) **ElseIf** (**Intersection-In-Two-Lines**(S_3, S_1, S_2, S_5, S_7)) **Then**

(5a) Return a *true* to the caller and terminate the execution of the function.

(6) **ElseIf** (**Intersection-In-Two-Lines**(S_3, S_6, S_9, S_5, S_7)) **Then**

(6a) Return a *true* to the caller and terminate the execution of the function.

(7) **ElseIf** (**Intersection-In-Two-Lines**(S_7, S_8, S_9, S_1, S_4)) **Then**

(7a) Return a *true* to the caller and terminate the execution of the function.

(8) **ElseIf** (**Intersection-In-Two-Lines**(S_7, S_8, S_9, S_3, S_5)) **Then**

(8a) Return a *true* to the caller and terminate the execution of the function.

(9) **ElseIf** (**Intersection-In-Two-Lines**(S_7, S_1, S_4, S_3, S_5)) **Then**

(9a) Return a *true* to the caller and terminate the execution of the function.

(10) **ElseIf** (**Intersection-In-Two-Lines**(S_9, S_7, S_8, S_3, S_6)) **Then**

(10a) Return a *true* to the caller and terminate the execution of the function.

(11) **ElseIf** (**Intersection-In-Two-Lines**(S_9, S_7, S_8, S_1, S_5)) **Then**

(11a) Return a *true* to the caller and terminate the execution of the function.

(12) **ElseIf** (**Intersection-In-Two-Lines**(S_9, S_3, S_6, S_1, S_5)) **Then**

(11a) Return a *true* to the caller and terminate the execution of the function.

(13) **ElseIf** (**Intersection-In-Two-Lines**(S_2, S_1, S_3, S_5, S_8)) **Then**

(13a) Return a *true* to the caller and terminate the execution of the function.

(14) **ElseIf** (**Intersection-In-Two-Lines**(S_4, S_5, S_6, S_1, S_7)) **Then**

(14a) Return a *true* to the caller and terminate the execution of the function.

(15) **ElseIf** (**Intersection-In-Two-Lines**(S_6, S_4, S_5, S_3, S_9)) **Then**

(15a) Return a *true* to the caller and terminate the execution of the function.

(16) **ElseIf** (**Intersection-In-Two-Lines**(S_8, S_7, S_9, S_2, S_5)) **Then**

(16a) Return a *true* to the caller and terminate the execution of the function.

(17) **ElseIf** (**Intersection-In-Two-Lines**(S_5, S_4, S_6, S_2, S_8)) **Then**

(17a) Return a *true* to the caller and terminate the execution of the function.

(18) **ElseIf** (**Intersection-In-Two-Lines**(S_5, S_4, S_6, S_1, S_9)) **Then**

(18a) Return a *true* to the caller and terminate the execution of the function.

(19) **ElseIf** (**Intersection-In-Two-Lines**(S_5, S_4, S_6, S_3, S_7)) **Then**

(19a) Return a *true* to the caller and terminate the execution of the function.

(20) **ElseIf** (**Intersection-In-Two-Lines**(S_5, S_2, S_8, S_1, S_9)) **Then**

(20a) Return a *true* to the caller and terminate the execution of the function.

(21) **ElseIf** (**Intersection-In-Two-Lines**(S_5, S_2, S_8, S_3, S_7)) **Then**

(21a) Return a *true* to the caller and terminate the execution of the function.

(22) **Else**

(22a) Return a *false* to the caller and terminate the execution of the function.

EndIf

EndFunction

Lemma 6-13: The function **Finding-Intersetion**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to find that there are two lines, each with one of the computer's mark and two blanks, intersecting in a single blank square.

Proof:

Each execution of Step(1) through Step (21) subsequently checks twenty-one pairs of two lines: (1 2 3) and (1 4 7), (1 2 3) and (1 5 9), (1 4 7) and (1 5 9), (1 2 3) and (3 6 9), (1 2 3) and (3 5 7), (3 6 9) and (3 5 7), (7 8 9) and (1 4 7), (7 8 9) and (3 5 7), (1 4 7) and (3 5 7), (7 8 9) and (3 6 9), (7 8 9) and (1 5 9), (3 6 9) and (1 5 9), (1 2 3) and (2 5 8), (4 5 6) and (1 4 7), (4 5 6) and (3 6 9), (7 8 9) and (2 5 8), (4 5 6) and (2 5 8), (4 5 6) and (1 5 9), (4 5 6) and (3 5 7), (2 5 8) and (1 5 9), and (2 5 8) and (3 5 7). If the condition is satisfied, then an X is filled into the single blank square to create two lines in which each line has two computer's marks and one blank, thus forking the opponent and a *true* is subsequently returned to the caller and the execution of the function is terminated from each execution of Step(1a) through Step (21a). Otherwise, on each execution of Step (22a), a *false* is returned to the caller and the execution of the function is terminated. Therefore, it is inferred at once from the statements above that the function **Finding-Intersetion**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to find that there are two lines, each with one of the computer's mark and two blanks, intersecting in a single blank square.

P. Biological Algorithms of Checking Whether One of Twenty-One Pairs of Two Lines Has One of the Computer's Marks and Two Blank and Intersecting in a Single Blank Square

The following function, **Intersection-In-Two-Lines**(S_a, S_b, S_c, S_d, S_e), is applied to test whether for one of *twenty-one* pairs in which each pair contains two lines the two lines have one of the computer's marks and two blank and intersecting in a single blank square or not. If the condition is satisfied, then an X is filled into the single blank square to create two lines in which each line has two computer's marks and one blank, thus forking the opponent and a *true* is returned to the caller. Otherwise, a *false* is returned to the caller. Tube S_a is used to store the content of an intersectional empty square, and tubes S_b, S_c, S_d and S_e are subsequently used to store the content of other four squares.

Intersection-In-Two-Lines(S_a, S_b, S_c, S_d, S_e)

(1) $S_b^{ON} = +(S_b, b^1)$ and $S_b^{OFF} = -(S_b, b^1)$, and $S_c^{ON} = +(S_c, b^1)$ and $S_c^{OFF} = -(S_c, b^1)$.

(2) $S_d^{ON} = +(S_d, b^1)$ and $S_d^{OFF} = -(S_d, b^1)$, and $S_e^{ON} = +(S_e, b^1)$ and $S_e^{OFF} = -(S_e, b^1)$.

(3) **If** (Detect(S_a) == *false*) **Then**

(4) **If** ((Detect(S_b^{ON}) == *true*) **AND** (Detect(S_c^{ON}) == *false*) **AND** (Detect(S_c^{OFF}) == *false*) **AND** (Detect(S_d^{ON}) == *true*) **AND** (Detect(S_e^{ON}) == *false*) **AND** (Detect(S_e^{OFF}) == *false*)) **Then**

(4a) Append-Tail(S_a, b^1).

(4b) $S_b = \cup(S_b^{ON}, S_b^{OFF})$, $S_c = \cup(S_c^{ON}, S_c^{OFF})$, $S_d = \cup(S_d^{ON}, S_d^{OFF})$, $S_e = \cup(S_e^{ON}, S_e^{OFF})$.

(4c) Return a *true* to the caller and terminate the execution of the function.

(5) **Else If** ((Detect(S_b^{ON}) == *true*) **AND** (Detect(S_c^{ON}) == *false*) **AND** (Detect(S_c^{OFF}) == *false*) **AND** (Detect(S_e^{ON}) == *true*) **AND** (Detect(S_d^{ON}) == *false*) **AND** (Detect(S_d^{OFF}) == *false*)) **Then**

(5a) Append-Tail(S_a, b^1).

(5b) $S_b = \cup(S_b^{ON}, S_b^{OFF})$, $S_c = \cup(S_c^{ON}, S_c^{OFF})$, $S_d = \cup(S_d^{ON}, S_d^{OFF})$, $S_e = \cup(S_e^{ON}, S_e^{OFF})$.

(5c) Return a *true* to the caller and terminate the execution of the function.

(6) **Else If** ((Detect(S_c^{ON}) == *true*) **AND** (Detect(S_b^{ON}) == *false*) **AND** (Detect(S_b^{OFF}) == *false*) **AND** (Detect(S_d^{ON}) == *true*) **AND** (Detect(S_e^{ON}) == *false*) **AND** (Detect(S_e^{OFF}) == *false*)) **Then**

(6a) Append-Tail(S_a, b^1).

(6b) $S_b = \cup(S_b^{ON}, S_b^{OFF})$, $S_c = \cup(S_c^{ON}, S_c^{OFF})$, $S_d = \cup(S_d^{ON}, S_d^{OFF})$, $S_e = \cup(S_e^{ON}, S_e^{OFF})$.

(6c) Return a *true* to the caller and terminate the execution of the function.

(7) **Else If** ((Detect(S_c^{ON}) == *true*) **AND** (Detect(S_b^{ON}) == *false*) **AND** (Detect(S_b^{OFF}) == *false*) **AND** (Detect(S_e^{ON}) == *true*) **AND** (Detect(S_d^{ON}) == *false*) **AND** (Detect(S_d^{OFF}) == *false*)) **Then**

(7a) Append-Tail(S_a, b^1).

(7b) $S_b = \cup(S_b^{ON}, S_b^{OFF})$, $S_c = \cup(S_c^{ON}, S_c^{OFF})$, $S_d = \cup(S_d^{ON}, S_d^{OFF})$, $S_e = \cup(S_e^{ON}, S_e^{OFF})$.

(7c) Return a *true* to the caller and terminate the execution of the function.

(8) **Else**

(8a) S_b=∪(S_b^{ON}, S_b^{OFF}), S_c=∪(S_c^{ON}, S_c^{OFF}), S_d =∪(S_d^{ON}, S_d^{OFF}), S_e =∪(S_e^{ON}, S_e^{OFF}).

(8b) Return a *false* to the caller and terminate the execution of the function.

EndIf

(9) **Else**

(9a) S_b=∪(S_b^{ON}, S_b^{OFF}), S_c=∪(S_c^{ON}, S_c^{OFF}), S_d =∪(S_d^{ON}, S_d^{OFF}), S_e =∪(S_e^{ON}, S_e^{OFF}).

(9b) Return a *false* to the caller and terminate the execution of the function.

EndIf

EndFunction

Lemma 6-14: For one of twenty-one pairs in which each pair contains two lines, the function **Intersection-In-Two-Lines**(S_a, S_b, S_c, S_d, S_e) learns how to decide whether the two lines have one of the computer's mark and two blanks, intersecting in a single blank square or not.

Proof:

On each execution of Step (1) and Step (2), four *extract* operations are used to separate tubes S_b, S_c, S_d and S_e to generate tubes S_b^{ON}, S_b^{OFF}, S_c^{ON}, S_c^{OFF}, S_d^{ON}, S_d^{OFF}, S_e^{ON} and S_e^{OFF}. DNA strands in tubes S_b^{ON}, S_c^{ON}, S_d^{ON}, and S_e^{ON} all encodes an *X*, and DNA strands in tubes S_b^{OFF}, S_c^{OFF}, S_d^{OFF}, and S_e^{OFF} also all encodes an *O*. If a *false* is returned from each execution of Step (3), then the *intersectional* square is an empty square and Step (4) through Step (8b) will be executed. Otherwise, there is no empty intersectional square, tubes S_b^{ON}, S_b^{OFF}, S_c^{ON}, S_c^{OFF}, S_d^{ON}, S_d^{OFF}, S_e^{ON} and S_e^{OFF} are subsequently poured into tubes S_b, S_c, S_d and S_e from each execution of Step (9a) and a *false* is return to the caller and the execution of the function is terminated from each execution of Step (9b).

On each execution of Step (4), Step (5), Step (6) or Step (7), six *detect* operations are used to check whether other two squares of each line are one empty square and one of the computer's mark or not. If the condition above is satisfied, then from each execution of Step (4a) through Step (4c), each execution of Step (5a) through Step (5c), each execution of Step (6a) through Step (6c), or each execution of Step (7a) through Step (7c) an *X* is filled into tube S_a (an intersectional empty square), tubes S_b^{ON}, S_b^{OFF}, S_c^{ON}, S_c^{OFF}, S_d^{ON}, S_d^{OFF}, S_e^{ON} and S_e^{OFF} are subsequently poured into tubes S_b, S_c, S_d and S_e and a *true* is return to the caller and the execution of the function is terminated. Otherwise, tubes S_b^{ON}, S_b^{OFF}, S_c^{ON}, S_c^{OFF}, S_d^{ON}, S_d^{OFF}, S_e^{ON} and S_e^{OFF} are subsequently poured into tubes S_b, S_c, S_d and S_e from each execution of Step (8a) and a *false* is return to the caller and the execution of the function is terminated from each execution of Step (8b). Therefore, it is at once inferred from the statements above that for one of twenty-one pairs in which each pair contains two lines, the function **Intersection-In-Two-Lines**(S_a, S_b, S_c, S_d, S_e) learns how to decide whether the two lines have one of the computer's mark and two blanks, intersecting in a single blank square or not. ■

Q. Biological Algorithms of Testing Whether in the Board A Center Square Is Empty

The following function, **Finding-Center**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is applied to test whether in the board the *fifth* square that is called a *center* square is empty or not. If the condition above is satisfied, then an X is filled into the *center* square. The first parameter T_0 contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are used to store the contents of nine squares.

Finding-Center(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) $S_5^{ON} = +(S_5, b^1)$ and $S_5^{OFF} = -(S_5, b^1)$.
(2) **If** ((Detect(S_5^{ON}) == *false*) **AND** (Detect(S_5^{OFF}) == *false*)) **Then**
    (2a) Append-Tail(S_5, b^1).
    (2b) Return a *true* to the caller and terminate the execution of the function.
(3) **Else**
    (3a) $S_5 = \cup(S_5^{ON}, S_5^{OFF})$.
    (3b) Return a *false* to the caller and terminate the execution of the function.
EndIf
EndFunction

Lemma 6-15: The function **Finding-Center**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to decide whether in the board the *fifth* square that is called a *center* square is empty or not.

Proof:

On each execution of Step (1), it uses the *extract* operation to generate that tube S_5^{ON} contains DNA strands encoding an X and tube S_5^{OFF} includes DNA strands encoding an O. Next, on each execution of Step (2), it applies two *detect* operations to check whether the *fifth* square in the board is not occupied by any player or not. If both of them returns a false, then an X is filled into the center square from each execution of Step (2a), and from each execution of Step (2b) a *true* is returned to the caller and the execution of the function is terminated. Otherwise, on each execution of Step (3a) it uses one *merge* operation to pour tubes S_5^{ON} and S_5^{OFF} into tube S_5 and from each execution of Step (3b) a *false* is returned to the caller and the execution of the function is terminated. Therefore, it is at once derived from the statements above that the function **Finding-Center**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to decide whether in the board the *fifth* square that is called a *center* square is empty or not. ■

R. Biological Algorithms of Judging Whether Side Squares Are Occupied by the Opponent and the Opposite of Each Side Square Is an Empty Square

The following function, **Opponent-on-Side**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is applied to check whether in the board the *second* square, the *fourth* square, the *sixth* square or the *eighth* square that are all called *side* squares are occupied by the ***opponent*** or not and to also simultaneously check whether the *opposite* of each *side* square is an empty square or not. If the condition above is satisfied, then an X is filled into the *opposite* of the *side* square. The first parameter T_0 contains DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are used to store the contents of nine squares.

Opponent-on-Side(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **For** k = 2 **to** 8 **Step** 2
  (2) $S_k^{ON} = +(S_k, b^0)$ and $S_k^{OFF} = -(S_k, b^0)$.
  (3) $S_{10-k}^{ON} = +(S_{10-k}, b^1)$ and $S_{10-k}^{OFF} = -(S_{10-k}, b^1)$.
  (4) **If** ((Detect(S_k^{ON}) == *true*) **AND** (Detect(S_{10-k}^{ON}) == *false*) **AND** (Detect(S_{10-k}^{OFF}) == *false*)) **Then**
    (4a) Append-Tail(S_{10-k}, b^1).
    (4b) $S_k = \cup(S_k^{ON}, S_k^{OFF})$.
    (4c) Return a *true* to the caller and terminate the execution of the function.
  (5) **Else**
    (5a) $S_k = \cup(S_k^{ON}, S_k^{OFF})$ and $S_{10-k} = \cup(S_{10-k}^{ON}, S_{10-k}^{OFF})$.
  EndIf
EndFor
(6) Return a *false* to the caller and terminate the execution of the function.
EndFunction

Lemma 6-16: The function **Opponent-on-Side**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to check whether in the board the *second* square, the *fourth* square, the *sixth* square or the *eighth* square that are all called *side* squares are occupied by the ***opponent*** or not and how to also check whether the *opposite* of each *side* square is an empty square or not.

Proof:

Step (1) is a single loop and is used to check whether each side square is occupied by the opponent and the *opposite* of each *side* square is an empty square or not. On each execution of Step (2) and Step (3), they use two *extract* operations to generate tubes S_k^{ON}, S_k^{OFF}, S_{10-k}^{ON} and S_{10-k}^{OFF}. In tubes S_k^{ON} and S_{10-k}^{ON}, DNA strands respectively encode an O and an X, and in tubes S_k^{OFF} and S_{10-k}^{OFF}, DNA strands respectively encode an X and an O. Next, on each execution of Step (4), it uses three *detect* operations to check whether the kth square (tube S_k^{ON}) that is a side square is occupied by the opponent and the $(10 - k)$th square (tubes S_{10-k}^{ON} and S_{10-k}^{OFF}) that is the opposite of the side square is an empty square or not. If a true and two false are returned, then an X is filled into the opposite of the side square from each execution of Step (4a), tubes S_k^{ON} and S_k^{OFF} are poured into tube S_k from each execution of Step (4b) and from each execution of Step (4c) a true is returned to the caller and the execution of the function is terminated. Otherwise, from each execution of Step (5a), tubes S_k^{ON} and S_k^{OFF} are poured into tube S_k and tubes S_{10-k}^{ON} and S_{10-k}^{OFF} are poured into tube S_{10-k}. After each operation from Step (2) through Step (5a) is all implemented, if no X is filled into the opposite of any side square, then from each execution of Step (6) a false is returned to the caller and the execution of the function is terminated. Therefore, it is at once inferred from the statements above that the function **Opponent-on-Side**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to check whether in the board the *second* square, the *fourth* square, the *sixth* square or the *eighth* square that are all called *side* squares are occupied by the ***opponent*** or not and how to also check whether the *opposite* of each *side* square is an empty square or not.
■

*S. Biological Algorithms of Deciding Whether Corner Square*s *Are Occupied by the Opponent and the Opposite of Each Corner Square Is an Empty Square*

The following function, **Opponent-on-Corner**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), is used to decide whether in the board the *first* square, the *third* square, the *seventh* square or the *ninth* square that are all called *corner* squares are occupied by the *opponent* or not and to also simultaneously check whether the *opposite* of each *corner* square is an empty square or not. If the condition above is satisfied, then an X is filled into the *opposite* of the *corner* square. The first parameter T_0 consists of DNA strands encoding the result of predicating whether there is any a three-in-a-row or not. Tubes S_1 through S_9 that are, subsequently, the second parameter through the tenth parameter are employed to store the contents of nine squares.

Opponent-on-Corner(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)
(1) **If** (**Checking-Opponent-on-Corner**(S_1, S_9)) **Then**
    (1a) Return a *true* to the caller and terminate the execution of the function.
(2) **ElseIf** (**Checking-Opponent-on-Corner**(S_3, S_7)) **Then**
    (2a) Return a *true* to the caller and terminate the execution of the function.
(3) **ElseIf** (**Checking-Opponent-on-Corner**(S_7, S_3)) **Then**
    (3a) Return a *true* to the caller and terminate the execution of the function.
(4) **ElseIf** (**Checking-Opponent-on-Corner**(S_9, S_1)) **Then**
    (4a) Return a *true* to the caller and terminate the execution of the function.
(5) **Else**
    (5a) Return a *false* to the caller and terminate the execution of the function.
EndIf
EndFunction

Lemma 6-17: The function **Opponent-on-Corner**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9) learns how to decide whether in the board the *first* square, the *third* square, the *seventh* square or the *ninth* square that are all called *corner* squares are occupied by the *opponent* or not and also learns how to decide whether the *opposite* of each *corner* square is an empty square or not.

Proof:

On each execution of Step (1) through Step (4), they respectively call the function **Checking-Opponent-on-Corner**(S_a, S_b) to test whether tubes (four *corner* square) S_1, S_3, S_7, or S_9 are occupied by the opponent or not and also simultaneously test whether tubes (the opposite of each corner square) S_9, S_7, S_3, or S_1 are an empty square or not. If a true is returned from the function **Checking-Opponent-on-Corner**(S_a, S_b), then from each execution of Step (1a), Step (2a), Step (3a) or Step (4a) a true is returned to the caller and the execution of the function is terminated. Otherwise, from each execution of Step (5a) a *false* is returned to the caller and the execution of the function is terminated. ■

T. Biological Algorithms of Deciding Whether One of Corner Squares Is Occupied by the Opponent and Its Opposite Is an Empty Square

The following function, **Checking-Opponent-on-Corner**(S_a, S_b), is employed to check whether one of four corner squares is occupied by the *opponent* and its opposite is an empty square or not. Tube S_a that is the first parameter is used to store the content for one of four corner squares (S_1, S_3, S_7, or S_9), and tube S_b that is the second parameter is applied to store the content for its opposite (S_9, S_7, S_3, or S_1).

Checking-Opponent-on-Corner(S_a, S_b)
(1) $S_a^{ON} = +(S_a, b^0)$ and $S_a^{OFF} = -(S_a, b^0)$ and $S_b^{ON} = +(S_b, b^1)$ and $S_b^{OFF} = -(S_b, b^1)$.
(2) **If** ((Detect(S_a^{ON}) == *true*) **AND** (Detect(S_b^{ON}) == *false*) **AND**
(Detect(S_b^{OFF}) == *false*)) **Then**
(2a) Append-Tail(S_b, b^1).
(2b) $S_a = \cup(S_a^{ON}, S_a^{OFF})$.
(2c) Return a *true* to the caller and terminate the execution of the function.
(3) **Else**
(3a) $S_a = \cup(S_a^{ON}, S_a^{OFF})$ and $S_b = \cup(S_b^{ON}, S_b^{OFF})$.
(3b) Return a *false* to the caller and terminate the execution of the function.
EndIf
EndFunction

Lemma 6-18: The function **Checking-Opponent-on-Corner**(S_a, S_b) learns how to decide whether in the board the *first* square, the *third* square, the *seventh* square or the *ninth* square that are all called *corner* squares are occupied by the *opponent* or not and also learns how to decide whether the *opposite* of each *corner* square is an empty square or not.

Proof:

On each execution of Step (1), it applies two *extract* operations to generate tubes S_a^{ON}, S_a^{OFF}, S_b^{ON} and S_b^{OFF}. In tubes S_a^{ON} and S_b^{ON}, DNA strands respectively encode an O and an X, and in tubes S_a^{OFF} and S_b^{OFF}, DNA strands respectively encode an X and an O. Next, on each execution of Step (2), it uses three *detect* operations to check whether tube S_a^{ON} that is a corner square is occupied by the opponent and tubes S_b^{ON} and S_b^{OFF} that is the opposite of the corner square is an empty square or not. If a true and two false are returned, then an X is filled into the opposite of the corner square from each execution of Step (2a), tubes S_a^{ON} and S_a^{OFF} are poured into tube S_a from each execution of Step (2b) and from each execution of Step (2c) a true is returned to the caller and the execution of the function is terminated. Otherwise, from each execution of Step (3a), tubes S_a^{ON} and S_a^{OFF} are poured into tube S_a, tubes S_b^{ON} and S_b^{OFF} are poured into tube S_b and from each execution of Step (3b) a false is returned to the caller and the execution of the function is terminated. ■

VII. ASSESSMENT OF COMPLEXITY TO THE PROPOSED BIOLOGICAL ALGORITHMS

The following lemma is used to show *volume* complexity and *time* complexity of the proposed biological algorithms to play a tic-tac-toe.

Lemma 7-1: Playing a tic-tac-toe with human together can be completed with O(1) biological operations, O(1) DNA strands, O(1) tubes and the number of the base pairs of the longest DNA strand O(1).

Proof:

From the execution of Step (1) and Step (2) in **Play-Tic-Tac-Toe**(T_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9), it takes *constant* biological operations and *constant* tubes. Because the execution of Step (3) only gives one selection who go first to play it, no biological operation are implemented. Next on the execution of Step (4), it completes one moving of the opponent with *constant* biological operations and *constant* tubes. Next, on the execution of Step (5), it completes one moving of the computer with *constant* biological operations and *constant* tubes. The opponent and the computer at most only give their five selections which can be completed with *constant* biological operations and *constant* tubes, and the contents of nine squares are encoded by constant DNA strands with *constant* length. Therefore, it is at once inferred from the statements above that playing a tic-tac-toe with human together can be completed with O(1) biological operations, O(1) DNA strands, O(1) tubes and the number of the base pairs of the longest DNA strand O(1). ■

VIII. CONCLUSIONS

Playing games is the behavior of human's intelligence, and a tic-tac-toe is one of the simplest games. Nine tubes S_1 through S_9 can be regarded nine *variables* that are used to store an O or an X of each square. Tube T_0 also can be regarded as a variable storing r^0 that predicates that there are no three Os or three Xs to make three-in-a-row or storing r^1 that predicates that there are three Os or three Xs to make three-in-a-row.

From a biological standpoint, all sequences generated to represent each bit must be checked to ensure that the DNA strands that they encode do not form unwanted secondary structures with one another (i.e., strands remain separate at all times, and only bind together when this is required). The biggest challenge of implementing the proposed method is actually to the problem of strand design that has been addressed at length to minimize the possibility of unwanted binding. However, from the implementation of the proposed method, O(1) DNA strands, O(1) tubes and the number of the base pairs of the longest DNA strand O(1) are needed. This is to say that the problem of strand design can be easily overcome.

From **Lemma 7-1**, playing a tic-tac-toe with human together can be implemented with O(1) biological operations that are a *constant* time. This is a very useful algorithm for consideration in a DNA implementation. With current biotechnology, the time for each operation is at least one second. Realistically, steps like gel electrophoresis take much longer, but for the sake of argument say each biological operation takes one second. Because from the proposed algorithm *constant* biological operations are implemented, it takes about *constant* seconds to obtain the result of who wins the game.

Bonnet et al. in [6] used *intensity* of *green fluorescent protein* to encode two values '0' and '1' of a bit and implemented **AND**, **NAND**, **OR**, **XOR**, **NOR**, and **XNOR** gates. This gives another very good choice for representing two values '0' and '1' of a bit. In the past two methods, we designed two kinds of plasmids and the required polymerases for generating *green fluorescent protein* and *blue fluorescent protein* encoding two marks 'O' and 'X'. But after checking the fluorescent induction systems of *E. coli*, we realized that it would take more than 2 hours to get a detectable level of fluorescent proteins after chemical induction. This is to say that when one of two players selects his single move, after at least two hours his mark just can be encoded. This indicates that this will be a major limitation of the biological experiment.

REFERENCES

[1] A. Newell and H. A. Simon. *Human Problem Solving*. Englewood Cliffs, N.J., Prentice-Hall, **ISBN-13: 978-0134454030 ISBN-10: 0134454030**, 1972.

[2] R. P. Feynman. "In Minaturization". D.H. Gilbert, Ed., Reinhold Publishing Corporation, New York, 1961, pp. 282-296.

[3] L. Adleman. "Molecular Computation of Solutions to Combinatorial Problems". *Science*, 266: 1021-1024, Nov. 11, 1994.

[4] M. Amos. *Theoretical and Experimental DNA Computation*. Springer, **ISBN**-13 978-3-540-65773-6, 2005.

[5] W.-L. Chang and A. V. Vasilakos. *Molecular Computing*: *Towards a Novel Computing Architecture for Complex Problem Solving*, Springer, **ISBN**-10: 3319051210 **|** **ISBN**-13: 978-3319051215, 2014.

[6] J. Bonnet, P. Yin, M. E. Ortiz, P. Subsoontorn, D. Endy. “Amplifying Genetic Logic Gates”. *Science*: Volume 340, No. 6132, pp. 599-603, May 2013.

[7] R. S. Braich, C. Johnson, P. W.K. Rothemund, D. Hwang, N. Chelyapov and L. M. Adleman. “Solution of a Satisfiability Problem on a Gel-based DNA Computer”. Proceedings of the 6th International Conference on DNA Computation in the Springer-Verlag Lecture Notes in Computer Science series, pp. 27-42, 2000.

[8] Ravinderjit S. Braich, Clifford Johnson, Paul W.K. Rothemund, Darryl Hwang, Nickolas Chelyapov and Leonard M. Adleman. “Solution of a 20-Variable 3-SAT Problem on a DNA Computer”. *Science*, Volume 296, Issue 5567, 499-502, 19 April, 2002.

[9] Leonard Adleman, Paul W. K. Rothemund, Sam Roweis, and Erik Winfree. “On applying molecular computation to the Data Encryption Standard”. The 2*nd* annual workshop on DNA Computing, Princeton University, DIMACS: series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pp. 31-44, 1999.

[10] L. L. Qian and E. Winfree. “Parallel and Scalable Computation and Spatial Dynamics with DNA-based Chemical Reaction Networks on a Surface”. *DNA Computing and Molecular Programming* (***DNA20***), *Lecture Notes in Computer Science* (***LNCS***), Volume 8727, pp 114-131, 2014.

[11] J. Velasco, Y. Lee, Z. Zhao, L Jing, P. Kratz, M. Bockrath, C. N. Lau. “Transport Measurement of Landau Level Gaps in Bilayer Graphene with Layer Polarization Control”. *Nano Letter* 14, 1324, 2014.

[12] F. Farnoud (Hassanzadeh), M. Schwartz and J. Bruck. “The Capacity of String-Replication Systems”. Paradise, ETR126 (Electronic Technical Reports), pp. 1-9, January, 2014 (http://arxiv.org/abs/1401.4634).

[13] B. Rastegari, A. Condon, N. Immorlica, R. Irving, and K. Leyton-Brown. “Reasoning about Optimal Stable Matchings under Partial Information”. *The Fifteenth ACM Conference on Electronic Commerce*, pp. 431-448, 2014.

[14] M. Cook, Y. Fu, and Robert T. “SchwellerTemperature 1 Self-Assembly: Deterministic Assembly in 3d and Probabilistic Assembly in 2d”. SIAM Symposium on Discrete Algorithms, pp. 1-40, 2011.

[15] Z. Z. Sun, E. Yeung, C. A. Hayes, V. Noireaux, and R. M. Murray. “Linear DNA for Rapid Prototyping of Synthetic Biological Circuits in an Escherichia Coli Based TX-TL Cell-free System”. ACS Synthetic Biology, 3(6), pp. 387–397 2013

[16] J. P. Sadowski, C.R. Calvert, D.Y. Zhang, N.A. Pierce, and P. Yin. “Developmental Self-Assembly of a DNA Tetrahedron”. ACS Nano, 8(4): 3251-3259, 2014.

[17] L. L. Qian, D. Soloveichik, and E. Winfree. "Efficient Turing-Universal Computation with DNA Polymers". DNA Computing and Molecular Programming, LNCS 6518: 123-140, 2011.

[18] L. L. Qian, E. Winfree, and J. Bruck. “Neural Network Computation with DNA Strand Displacement Cascades”. Nature, 475: 368-372, 2011.

[19] L. L. Qian and E. Winfree. “Scaling up Digital Circuit Computation with DNA Strand Displacement Cascades”. Science, 332: 1196-1201, 2011.

[20] Y. Benenson. “DNA Computes a Square Root”. Nature Nanotechnology, 6: 465-467, 2011.

[21] A. Ehrenfeucht, and G. Rozenberg. “Processes Inspired by the Functioning of Living Cells: Natural Computing Approach”. Proceedings Nature of Computation Logic, Algorithms, Applications - 9th Conference on Computability in Europe (CiE 2013), Lecture Notes in Computer Science, Volume 7921, pp. 120-122, 2013.

[22] D. Boneh, C. Dunworth, and R. J. Lipton. “Breaking DES Using a Molecular Computer”. In Proceedings of the 1*st* DIMACS Workshop on DNA Based Computers, 1995 and also in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 27, pp. 1-14, 1995.

[23] W.-L. Chang, and A. V. Vasilakos, “Molecular Algorithms of Implementing Bio-molecular Databases on a Biological Computer”. *IEEE Transactions on Nanobioscience*, Volume 14, NO. 1, pp. 104-111, January 2015.

[24] W.-L. Chang, T.-T. Ren, and M. Feng. “Quantum Algorithms and Mathematical Formulations of Bio-molecular Solutions of the Vertex Cover Problem in the Finite-dimensional Hilbert Space”. *IEEE Transactions on NanoBioscience*, Volume 14, NO. 1, pp 121-128, January 2015.

[25] M. R. Garey, and D. S. Johnson. *Computer and intractability*: *A Guide to the Theory of NP-Completeness*. W. H. Freeman Company, New York, 1979.

[26] R. Karp. “On the Computational Complexity of Combinatorial Problems”. *Networks,* Volume 5, pp. 45-68, 1975.

[27] Tenn., Memphis. *Hidden variable theory: Bell’s theorem, Kochen-Specker theorem, Spekkens toy model, local hidden variable theory*. L. L. C., **ISBN**-9781155784854, 2010.

[28] W.-L. Chang, A. V. Vasilakos and M. S.-H. Ho. “The DNA-Based Algorithms of Implementing Arithmetical Operations of Complex Vectors on a Biological Computer”. *IEEE Transactions on NanoBioscience*, Volume 14, NO. 8, pp 907-914, December 2015.