DNA computers, tomorrow's reality

Lila Kari[1]

"Come forth into the light of things
Let nature be your teacher." [47]

1 Biological mathematics: the tables turned

The field usually referred to as *mathematical biology* is a highly interdisciplinary area that lies at the intersection of mathematics and biology. Classical illustrations include the development of stochastic processes and statistical methods to solve problems in genetics and epidemiology. As the name used to describe work in this field indicates, with "biology" the noun, and "mathematical" the modifying adjective, the relationship between mathematics and biology has so far been one–way. Typically, mathematical results have emerged from or have been used to solve biological problems (see [24] for a comprehensive survey). In contrast, Leonard Adleman, [1], succeeded in solving an instance of the directed Hamiltonian path problem solely by manipulating DNA strings. This marks the first instance of the connection being reversed: a mathematical problem is the end toward which the tools of biology are used. To be semantically correct, instead of categorizing the research in DNA computing as belonging to mathematical biology, we should be employing the mirror–image term *biological mathematics* for the field born in November 1994.

Despite the complexity of the technology involved, the idea behind biological mathematics is the simple observation that the following two processes, one biological and one mathematical, are analogous:

(a) the very complex structure of a living being is the result of applying simple operations (copying, splicing, etc.) to initial information encoded in a DNA sequence,

(b) the result $f(w)$ of applying a computable function to an argument w can be obtained by applying a combination of basic simple functions to w (see Section 4 or [42] for details).

If noticing this analogy were the only ingredient necessary to cook a computing DNA soup, we would have been playing computer games on our DNA laptops a long time ago! It took in fact the ripening of several factors and a

[1]Department of Mathematics, University of Western Ontario, London, Ontario, N6A 5B7 Canada, email:lkari@julian.uwo.ca

renaissance mind like Adleman's, a mathematician knowledgeable in biology, to bring together these apparently independent phenomena. Adleman realized that not only are the two processes similar but, thanks to the advances in molecular biology technology, one can use the biological to simulate the mathematical. More precisely, DNA strings can be used to encode information while enzymes can be employed to simulate simple computations, in a way described below.

DNA (deoxyribonucleic acid) is found in every living creature as the storage medium for genetic information. It is composed of units called nucleotides, distinguished by the chemical group, or base, attached to them. The four bases are *adenine, guanine, cytosine* and *thymine*, abbreviated as A, G, C, and T. Single nucleotides are linked together end–to–end to form DNA strands. The DNA sequence has a *polarity*: a sequence of DNA is distinct from its reverse. Taken as pairs, the nucleotides A and T and the nucleotides C and G are said to be *complementary*. Two complementary single–stranded DNA sequences with opposite polarity will join together to form a double helix in a process called *annealing*. The reverse process – a double helix coming apart to yield its two constituent single strands – is called *melting*.

A single strand of DNA can be likened to a string consisting of a combination of four different symbols, A, G, C, T. Mathematically, this means we have at our disposal a 4 letter alphabet $\Sigma = \{A, G, C, T\}$ to encode information, which is more than enough, considering that an electronic computer needs only two digits, 0 and 1, for the same purpose.

The simple operations that can be performed on DNA sequences are accomplished by a number of commercially available enzymes that execute some basic tasks. One class of enzymes, called *restriction endonucleases*, will recognize a specific short sequence in a strand and then "cut" the strand at that location. Another enzyme, called the *DNA ligase*, will hook together, or "ligate", the sticky end of a freshly cut DNA strand to another strand. There are many other enzymes that could potentially be useful, but for our models of computation these are sufficient.

The practical possibilities of encoding information in a DNA sequence and of performing simple bio–operations were used in [1] to solve a 7 node instance of the Directed Hamiltonian Path Problem. A directed graph G with designated vertices v_{in} and v_{out} is said to have a Hamiltonian path if and only if there exists a sequence of compatible "one–way" edges $e_1, e_2, \ldots, e_z$ (that is, a path) that begins at v_{in}, ends at v_{out} and enters every other vertex exactly once.

The following (nondeterministic) algorithm solves the problem:

Step 1. Generate random paths through the graph.

Step 2. Keep only those paths that begin with v_{in} and end with v_{end}.

Step 3. If the graph has n vertices, then keep only those paths that enter exactly n vertices.

Step 4. Keep only those paths that enter all of the vertices of the graph at least once.

Step 5. If any paths remain, say "YES"; otherwise say "NO".

To implement Step 1, each vertex of the graph was encoded into a random 20–nucleotide strand (20–letter sequence) of DNA. Then, for each (oriented) edge of the graph, a DNA sequence was created consisting of the second half of the sequence encoding the source vertex and the first half of the sequence encoding the target vertex. By using complements of the vertices as splints, DNA sequences corresponding to compatible edges were ligated, that is, linked together. Hence, the ligation reaction resulted in the formation of DNA molecules encoding random paths through the graph.

To implement Step 2, the product of Step 1 was amplified by polymerase chain reaction (PCR). Thus, only those molecules encoding paths that begin with v_{in} and end with v_{end} were amplified.

For implementing Step 3, a technique called gel–electrophoresis was used, that makes possible the separation of DNA strands by length. (The molecules are placed at the top of a wet gel, to which an electric field is applied, drawing them to the bottom. Larger molecules travel more slowly through the gel. After a period, the molecules spread out into distinct bands according to size.)

Step 4 was accomplished by iteratively using a process called affinity purification. This process permits single strands containing a given subsequence v (encoding a vertex of the graph) to be filtered out from a heterogeneous pool of other strands. (After synthesizing strands complementary to v and attaching them to magnetic beads, the heterogeneous solution is passed over the beads. Those strands containing v anneal to the complementary sequence and are retained. Strands not containing v pass through without being retained.)

To implement Step 5, the presence of a molecule encoding a Hamiltonian path was checked. (This was done by amplifying the result of Step 4 by polymerase chain reaction and then determining the DNA sequence of the amplified molecules).

A remarkable fact about Adleman's result is that not only does it give a solution to a mathematical problem, but that the problem solved is a hard computational problem in the sense explained below (see [20], [17]).

Problems can be ranked in difficulty according to how long the best algorithm to solve the problem will take to execute on a single computer. Algorithms whose running time is bounded by a polynomial (respectively exponential) function, in terms of the size of the input describing the problem, are in the "polynomial time" class P (respectively the "exponential time" class EXP). A problem is called *intractable* if it is so hard that no polynomial time algorithm can possibly solve it.

A special class of problems, apparently intractable, including P and included in EXP is the "non–deterministic polynomial time" class, or NP. The following inclusions between classes of problems hold:

$$P \subseteq NP \subseteq \text{ EXP} \subseteq \text{ Universal.}$$

NP contains the problems for which no polynomial time algorithm solving them is known, but that can be solved in polynomial time by using a non–deterministic

computer (a computer that has the ability to pursue an unbounded number of independent computational searches in parallel). The directed Hamiltonian path problem is a special kind of problem in NP known as "NP–complete". An NP–complete problem has the property that every other problem in NP can be reduced to it in polynomial time. Thus, in a sense, NP–complete problems are the "hardest" problems in NP.

The question of whether or not the NP–complete problems are intractable, mathematically formulated as "Does P equal NP?", is now considered to be one of the foremost open problems of contemporary mathematics and computer science. Because the directed Hamiltonian path problem has been shown to be NP–complete, it seems likely that no efficient (that is, polynomial time) algorithm exists for solving it with an electronic computer.

Following [1], in [25] a potential DNA experiment was described for finding a solution to another NP–complete problem, the Satisfiability Problem. The Satisfiability Problem consists of a Boolean expression, the question being whether or not there is an assignment of truth values – true or false – to its variables, that makes the value of the whole expression true. Later on, the method from [25] was used in [28], [27] and [26], to show how other NP–complete problems can be solved.

In [7], a "molecular program" was given for breaking the U.S. government's Data Encryption Standard (DES). DES encrypts 64 bit messages and uses a 56–bit key. Breaking DES means that given one (plain–text, cipher–text) pair, we can find a key which maps the plain–text to the cipher–text. A conventional attack on DES would need to perform an exhaustive search through all of the 2^{56} DES keys, which, at a rate of 100,000 operations per second, would take 10,000 years. In contrast, it was estimated that DES could be broken by using molecular computation in about 4 months of laboratory work.

The problems mentioned above show that molecular computation has the potential to outperform existing computers. One of the reasons is that the operations molecular biology currently provides can be used to organize massively parallel searches. It is estimated that DNA computing could yield tremendous advantages from the point of view of *speed, energy efficiency* and *economic information storing*. For example, in Adleman's model, [2], the number of operations per second could be up to approximately 1.2×10^{18}. This is approximately 1,200,000 times faster than the fastest supercomputer. While existing supercomputers execute 10^9 operations per Joule, the energy efficiency of a DNA computer could be 2×10^{19} operations per Joule, that is, a DNA computer could be about 10^{10} times more energy efficient (see [1]). Finally, according to [1], storing information in molecules of DNA could allow for an information density of approximately 1 bit per cubic nanometer, while existing storage media store information at a density of approximately 1 bit per 10^{12} nm^3. As estimated in [3], a single DNA memory could hold more words than all the computer memories ever made.

2 Can DNA compute everything?

The potential advantages of DNA computing versus electronic computing are clear in the case of problems like the Directed Hamiltonian Path Problem, the Satisfiability Problem, and breaking DES. On the other hand, these are only particular problems solved by means of molecular biology. They are one–time experiments to derive a combinatorial solution to a particular sort of problem. This immediately leads to two fundamental questions, posed in Adleman's article and in [20] and [28]:

(1) What kind of problems can be solved by DNA computing?

(2) Is it possible, at least in principle, to design a programmable DNA computer?

More precisely, one can reformulate the problems above as:

(1) Is the DNA model of computation computationally complete in the sense that the action of any computable function (or, equivalently, the computation of any Turing machine) can be carried out by DNA manipulation?

(2) Does there exist a universal DNA system, i.e., a system that, given the encoding of a computable function as an input, can simulate the action of that function for any argument? (Here, the notion of function corresponds to the notion of a program in which an argument w is the input of the program and the value $f(w)$ is the output of the program. The existence of a universal DNA system amounts thus to the existence of a DNA computer capable of running programs.)

Opinions differ as to whether the answer to these questions has practical relevance. One can argue as in [8] that from a practical point of view it maybe not be that important to simulate a Turing machine by a DNA computing device. Indeed, one should not aim to fit the DNA model into the Procrustean bed of classical models of computation, but try to completely rethink the notion of computation. On the other hand, finding out whether the class of DNA algorithms is computationally complete has many important implications. If the answer to it were unknown, then the practical efforts for solving a particular problem might be proven futile at any time: a Gödel minded person could suddenly announce that it belongs to a class of problems that are impossible to solve by DNA manipulation. The same holds for the theoretical proof of the existence of a DNA computer. As long as it is not proved that such a thing theoretically exists, the danger that the practical efforts will be in vane is always lurking in the shadow.

One more indication of the relevance of the questions concerning computational completeness and universality of DNA–based devices is that they have been addressed for most models of DNA computation that have so far been proposed.

The existing models of DNA computation are based on various combinations of a few primitive *biological operations*:

– *Synthesis* of a desired polynomial–length strand ([1], [2], [6], [5]);

– *Separation* of the strands by length ([1], [2], [8], [5], [6]);
– *Merging*: pour two test tubes into one to do union ([1], [2], [28]);
– *Extraction*: extract those strands containing a given pattern as a substring ([1], [2], [28], [8], [6]);
– *Melting/Annealing*: break apart/bond together two single DNA strands with complementary sequences ([8], [40], [46]);
– *Amplifying*: make copies of DNA strands by using the Polymerase Chain Reaction ([1], [2], [28], [8], [5], [6], [40]);
– *Cutting*: cut DNA strands by using restriction enzymes ([8], [5], [6], [21], [37], [40]);
– *Ligation*: paste DNA strands with complementary sticky ends by using ligases ([5], [6], [46], [21], [37], [40]);
– *Detection*: given a tube, say "yes" if it contains at least one DNA strand, and "no" otherwise ([1], [2], [28], [8]).

These operations are then used to write "programs" which receive a tube containing DNA strands as input and return as output either "yes" or "no" or a set of tubes. A computation consists of a sequence of tubes containing DNA strands.

There are pro's and con's for each model (combination of operations). The ones using operations similar to Adleman's have the obvious advantage that they could already be successfully implemented in the lab. The obstacle preventing the large scale automatization of the process is that most bio–operations rely on mainly manual handling of tubes. In contrast, the model introduced by Tom Head in [21] aims to be an "one–pot" tube with all the operations carried out in principle by enzymes. Moreover, it has the theoretical advantage of being a mathematical model with all the claims backed up by mathematical proofs. Its disadvantage is that the current state of art in molecular biology has not allowed yet practical implementation. Overall, the existence of different models with complementing features shows the versatility of DNA computing and increases the likelihood of practically constructing a DNA–computing–based device.

In the sequel we will restrict our attention to the *splicing system* model of DNA recombination that has been introduced in the seminal article of Tom Head, [21], already in 1987. A formal definition of the *splicing* operation (a combination of cut and paste), that can be used as the sole primitive for carrying out a computation, is given in Section 3. We will then prove in Section 4 that for the DNA model based on splicing we can affirmatively answer both questions posed at the beginning of this section.

3 A mathematical model: splicing systems

As described in Section 1, a DNA strand can be likened to a string over a four letter alphabet. Consequently, a natural way to model DNA computation is within the framework of formal language theory, which deals with letters and

strings of letters. We specify here only the notions and notations necessary for our exposition. For further formal language notions the reader is referred to [38].

An alphabet is a finite nonempty set; its elements are called *letters* or *symbols*. Σ^* denotes the free monoid generated by the alphabet Σ under the operation of catenation (juxtaposition). The elements of Σ^* are called *words* or *strings*. The empty string (the null element of Σ^*) is denoted by λ. A *language* over the alphabet Σ is a subset of Σ^*. For instance, if $\Sigma = \{a, b\}$ then $aaba, aabbb = a^2b^3$ are words over Σ, and the following sets are languages over Σ: $L_1 = \{\lambda\}$, $L_2 = \{a, ba, aba, abbaa\}$, $L_3 = \{a^p |\ p \text{ prime }\}$.

Since languages are sets, we may define the set–theoretic operations of union, intersection, difference, and complement in the usual fashion. The catenation of languages L_1 and L_2, denoted L_1L_2, is defined by $L_1L_2 = \{uv |\ u \in L_1, v \in L_2\}$.

A finite language can always be defined by listing all of its words. Such a procedure is not possible for infinite languages and therefore other devices for the representation of infinite languages have been developed. One of them is to introduce a *generative device* and define the language as consisting of all the words generated by the device. The basic generative devices used for specifying languages are *grammars*.

A *generative grammar* is an ordered quadruple

$$G = (N, T, S, P),$$

where N and T are disjoint alphabets, $S \in N$ and P is a finite set of ordered pairs (u, v) such that u, v are words over $N \cup T$ and u contains at least one letter of N. The elements of N are called *nonterminals* and those of T *terminals*; S is called the axiom. Elements (u, v) of P are called rewriting rules and are written $u \longrightarrow v$. If $x = x_1ux_2$, $y = x_1vx_2$ and $u \longrightarrow v \in P$, then we write $x \Longrightarrow y$ and say that x derives y in the grammar G. The reflexive and transitive closure of the derivation relation $\Longrightarrow$ is denoted by $\Longrightarrow^*$. The language generated by G is

$$L(G) = \{w \in T^* |\ S \Longrightarrow^* w\}.$$

Intuitively, the language generated by the grammar G is the set of words over the terminal alphabet that are derived from the axiom by repeatedly applying the rewriting rules.

Grammars are classified by imposing restrictions on the forms of productions. A grammar is called of *type–0* if no restriction (zero restrictions) is applied to the rewriting rules and is called *regular* if each rule of P is of the form $A \longrightarrow aB$, $A \longrightarrow a$, $A, B \in N$, $a \in T$. The family of finite languages will be denoted by FIN, the family of languages generated by regular grammars by REG and the family of languages generated by type–0 grammars by $\mathcal{L}_0$.

Using these formal language theory prerequisites, we can proceed now to define the *splicing operation*.

As described in [21] and modified in [19], given an alphabet Σ and two strings x and y over Σ, the splicing of x and y according to the splicing rule r consists of two steps: *(i)* cut x and y at certain positions determined by the splicing rule r, and *(ii)* paste the resulting prefix of x with the suffix of y, respectively the prefix of y with the suffix of x. Using the formalism introduced in [29], a *splicing rule* r over Σ is a word of the form $\alpha_1\#\beta_1\$\alpha_2\#\beta_2$, where $\alpha_1, \beta_1, \alpha_2, \beta_2$ are strings over Σ and $\#, \$$ are markers not belonging to Σ.

We say that z and w are obtained by *splicing* x and y according to the splicing rule $r = \alpha_1\#\beta_1\$\alpha_2\#\beta_2$, and we write

$$(x, y) \longrightarrow_r (z, w)$$

if and only if

$$\begin{array}{ll} x = x_1\alpha_1\beta_1 x_1' & \\ y = y_2\alpha_2\beta_2 y_2' & \text{and} \end{array} \quad \begin{array}{l} z = x_1\alpha_1\beta_2 y_2' \\ w = y_2\alpha_2\beta_1 x_1', \end{array}$$

for some $x_1, x_1', y_2, y_2' \in \Sigma^*$, as shown in Fig.1.

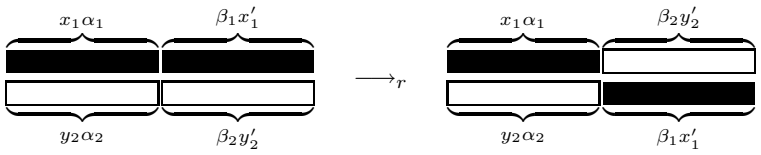

Fig.1. Splicing $x = x_1\alpha_1\beta_1 x_1'$ and $y = y_2\alpha_2\beta_2 y_1'$ according to the rule $r : \alpha_1\#\beta_1\$\alpha_2\#\beta_2$.

The words $\alpha_1\beta_1$ and $\alpha_2\beta_2$ are called *sites* of the splicing, while x and y are called the *terms* of the splicing. The splicing rule r determines both the sites and the positions of the cutting: between α_1 and β_1 for the first term and between α_2 and β_2 for the second. Note that the site $\alpha_1\beta_1$ can occur more than once in x while the site $\alpha_2\beta_2$ can occur more than once in y. Whenever this happens, the sites are chosen nondeterministically. As a consequence, the result of splicing x and y can be a set containing more than one pair (z, w).

We illustrate the way splicing works by using it to simulate the addition of two positive numbers, n and m. If we consider the alphabet $\Sigma = \{a, b, c\}$ and the splicing rule $r = a\#b\$c\#a$, then the splicing of $x = a^n b$ and $y = ca^m$ according to r yields the words a^{n+m} and cb. Indeed,

$$(a^n b, ca^m) = (x, y) =$$

$$(\underbrace{a^{n-1}}_{x_1}\underbrace{a}_{\alpha_1}\underbrace{b}_{\beta_1}\underbrace{\lambda}_{x_1'}, \underbrace{\lambda}_{y_2}\underbrace{c}_{\alpha_2}\underbrace{a}_{\beta_2}\underbrace{a^{m-1}}_{y_2'}) \longrightarrow_r (\underbrace{a^{n-1}}_{x_1}\underbrace{a}_{\alpha_1}\underbrace{a}_{\beta_2}\underbrace{a^{m-1}}_{y_2'}, \underbrace{c}_{y_2\alpha_2}\underbrace{b}_{\beta_1 x_1'}) =$$

$$= (a^{n+m}, cb) = (z, w).$$

In the case of DNA sequences, the alphabet consists of the four letters, A, T, C and G, the cutting could in principle be done by restriction enzymes and the catenation by ligases. For more biological motivations of splicing, see [21] and [22].

The splicing operation can be used as a basic tool for building a generative mechanism, called *splicing system*. Given a set of strings (axioms) and a set of splicing rules, the generated language will consist of the strings obtained as follows. Starting from the set of axioms, we iteratively use the splicing rules to splice strings from the set of axioms and/or strings obtained in preceding splicing steps.

If the classical notion of a set is used, we implicitly assume that, after splicing x and y and obtaining z and w, we may use again x or y as terms of the splicing, that is, the strings are not "consumed" by splicing. Similarly, there is no restriction on the number of copies of the newly obtained z and w. More realistic is the assumption that some of the strings are only available in a limited number of copies. In mathematical terms this translates in using, instead of sets, the notion of *multisets*, where one keeps track of the number of copies of a string at each moment.

In the style of [13], if $\mathbf{N}$ is the set of natural numbers, a multiset of Σ^* is a mapping $M : \Sigma^* \longrightarrow \mathbf{N} \cup \{\infty\}$ where for a word $w \in \Sigma^*$, $M(w)$ represents the number of occurrences of w. Here $M(w) = \infty$ is taken to mean that there are unboundedly many copies of the string w. The set $supp(M) = \{w \in \Sigma^* |\ M(w) \neq 0\}$ is called the *support* of M. With this modification of the notion of a set, we are now ready to introduce splicing systems.

Definition 3.1 *A splicing system is a quadruple $\gamma = (\Sigma, T, A, R)$, where Σ is an alphabet, $T \subseteq \Sigma$ is the terminal alphabet, A is a multiset over Σ^*, and $R \subseteq \Sigma^*\#\Sigma^*\$\Sigma^*\#\Sigma^*$ is the set of splicing rules.*

A splicing system γ defines a binary relation $\Longrightarrow_\gamma$ on the family of multisets of Σ^* as follows. For multisets M and M', $M \Longrightarrow_\gamma M'$ holds iff there exist $x, y \in supp(M)$ and $z, w \in supp(M')$ such that:

(i) $M(x) \geq 1$, $M(y) \geq 1$ if $x \neq y$ (resp. $M(x) \geq 2$ if $x = y$);

(ii) $(x, y) \longrightarrow_r (z, w)$ according to a splicing rule $r \in R$;

(iii) $M'(x) = M(x) - 1, M'(y) = M(y) - 1$ if $x \neq y$ (resp. $M'(x) = M(x) - 2$ if $x = y$);

(iv) $M'(z) = M(z) + 1, M'(w) = M(w) + 1$ if $z \neq w$ (resp. $M'(z) = M(z) + 2$ if $z = w$).□

Informally, having a "set" of strings with a certain number (possibly infinite) of available copies of each string, the next "set" is produced by splicing two of the existing strings (by "existing" we mean that both strings have multiplicity at least 1). After performing a splicing, the terms of the splicing are consumed (their multiplicity decreases by 1), while the products of the splicing are added to the "set" (their multiplicity increases by 1).

The *language generated* by a splicing system γ is defined as

$$L(\gamma) = \{w \in T^* |\ \ A \Longrightarrow_\gamma^* M \text{ and } w \in \text{supp}(M)\},$$

where A is the set of axioms and $\Longrightarrow_\gamma^*$ is the reflexive and transitive closure of $\Longrightarrow_\gamma$.

For two families of type–0 languages, F_1, F_2, denote

$$H(F_1, F_2) = \{L(\gamma) \mid \gamma = (\Sigma, T, A, R), A \in F_1, R \in F_2\}.$$

(The notation $H(F_1, F_2)$ comes from both the initial of Tom Head, who first introduced the notion of splicing, and the resemblance of Fig. 1 to a letter H viewed sideways.)

For example, $H(FIN, REG)$ denotes the family of languages generated by splicing systems where the set of axioms is finite and the set of rules is a regular language. A splicing system $\gamma = (\Sigma, T, A, R)$ with $A \in F_1, R \in F_2$, is said to be *of type* (F_1, F_2).

Splicing systems have been extensively studied in the literature. For example, the generative power of different types of splicing systems has been studied in [21], [11], [29], [31], [30], [32], [18], [9], [16]. Decidability problems have been tackled in [13]. Moreover, variations of the model have been considered: splicing systems with permitting/forbidding contexts in [9], linear and circular splicing systems in [22], [34], [44], splicing systems on graphs in [15], distributed splicing systems in [10], [12]. For a survey of the main results on splicing the reader is referred to [33].

4 The existence of DNA computers

Having defined a mathematical model of DNA computation, we now proceed to answer – for this model – the questions raised in Section 2. We start by showing that the splicing systems are computationally complete. By computational completeness of splicing we mean that every algorithm (effective procedure) can be carried out by a splicing system. It is obvious that this is not a mathematical definition of computational completeness. For an adequate definition, the intuitive notion of an algorithm (effective procedure) must be replaced by a formalized notion.

Since 1936, the standard accepted model of universal computation has been the Turing machine introduced in [41]. The Church–Turing thesis, the prevailing paradigm in computer science, states that no realizable computing device can be more powerful than a Turing machine. One of the main reasons that Church–Turing's thesis is widely accepted is that very diverse alternate formalizations of the class of effective procedures have all turned out to be equivalent to the Turing machine formalization. These alternate formalizations include Markov normal algorithms, Post normal systems, type–0 grammars, (which we have already considered in Section 3) as well as "computable" functions.

Showing that the splicing systems are computationally complete amounts thus, for example, to showing that the action of any computable function can be realized by a splicing system, where the term of computable function is detailed below (see [42]) .

Mappings of a subset of the Cartesian power set $\mathbf{N}^n$ into $\mathbf{N}$, where $n \geq 1$ and $\mathbf{N}$ is the set of natural numbers, are referred to as *partial functions*. If the domain of such a function equals $\mathbf{N}^n$, then the function is termed *total*. Examples of total functions (for different values of n) are:

The zero function: $Z(x_0) = 0$, for all $x_0 \in \mathbf{N}$.

The successor function: $S(x_0) = x_0 + 1$, for all $x_0 \in \mathbf{N}$.

The projection functions, for all i, n and $x_i \in \mathbf{N}$, $0 \leq i \leq n$:

$$U_i^{n+1}(x_0, x_1, \ldots, x_n) = x_i.$$

The class of partial recursive functions can be defined as the smallest class which contains certain basic functions and is closed under certain operations.

An $(n+1)$–ary function f is defined by the *recursion scheme* from the functions g and h if:

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(n+1, x_1, \ldots, x_n) &= h(f(n, x_1, \ldots, x_n), n, x_1, \ldots, x_n). \end{aligned}$$

The operation of *composition* associates to the functions $h, g_0, \ldots, g_k$ the function f defined by:

$$f(x_0, x_1, \ldots, x_n) = h(g_0(x_0, \ldots, x_n), \ldots, g_k(x_0, \ldots x_n)),$$

which is defined exactly for those arguments $(x_0, \ldots, x_n)$ for which each of g_i, $0 \leq i \leq k$, as well the corresponding value of f is defined.

We say that f is defined by using the *minimization* operation on g, if

$$f(x_0, \ldots, x_n) = \begin{cases} (\mu y)[g(y, x_0, \ldots, x_n) = 0], & \text{if there is such a } y \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

whose value, for a given $(x_0, \ldots, x_n)$, is the smallest value of y for which $g(y, x_0, \ldots, x_n) = 0$, and which is undefined if no such y exists.

A function f is defined *partial–recursively* if (i) it is the zero function, the successor function, or a projection function; (ii) it is defined by composing functions which are defined partial–recursively; (iii) it is defined by the recursion scheme from functions which are defined partial–recursively; or (iv) it is defined using the minimization operation on a function that is defined partial–recursively and is total.

It was proved that a function f is partial recursive if and only if there is a Turing machine which computes the values of f. On the other hand, according to the Church–Turing thesis, everything that can be effectively computed by any kind of device, can be computed by a Turing machine. As a consequence,

partial recursive functions came to be known also under the name of *(effectively) computable functions*.

The formal language notion equivalent to the notion of a computable function is the notion of a *type–0 language*, i.e., a language generated by a grammar with no restriction imposed on its rewriting rules. One can prove that a language is of type–0 if and only if its characteristic function is computable. (By the characteristic function of a language $L \subseteq \Sigma^*$ we mean the function ϕ of Σ^* such that $\phi(w) = 1$ if $w \in L$, and ϕ is undefined otherwise.)

Recalling the notation $\mathcal{L}_0$ for the family of type–0 languages, the result proving the computational completeness of splicing systems can be formulated as follows.

Theorem 4.1 $\mathcal{L}_0 = H(FIN, FIN)$.

Informally, the theorem says that every type–0 language can be generated by a splicing system with finitely many axioms and finitely many rules. Given a type–0 grammar G generating the language $L(G)$, the proof of the inclusion $\mathcal{L}_0 \subseteq H(FIN, FIN)$ consists of two steps: (a) construct a splicing system $\gamma \in H(FIN, FIN)$, that simulates the rewriting rules of the grammar G, and (b) prove that the constructed splicing system generates the given type–0 language, i.e., show the equality $L(\gamma) = L(G)$. The reverse inclusion can be proved directly or by invoking the Church–Turing thesis. (The proof techniques used in Theorem 4.1 were suggested in [13] and first developed in [30]. The reader is referred to [16] for details.)

In terms of computable functions, Theorem 4.1 states that the work of any computable function can be carried out by a splicing system. Equivalently, Theorem 4.1 tells that everything that is Turing–computable can be computed also by this DNA model of computation. This answers the question as regards to what kinds of algorithms (effective procedures, computable functions) can be simulated by DNA computing devices based on splicing, and the answer is: all of them.

Theorem 4.1 shows that every program (computable function) can be simulated by a finite splicing system, but this does not say anything about the existence of a *programmable DNA computer* based on splicing. To this aim, it is necessary to find a *universal splicing system*, i.e., a system with all components but one fixed, able to behave as any given splicing system γ when a code of γ is introduced in the set of axioms of the universal system. Formally,

Definition 4.1 *Given an alphabet T and two families of type–0 languages, F_1, F_2, a construct*

$$\gamma_U = (\Sigma_U, T, A_U, R_U),$$

where Σ_U is an alphabet, $T \subseteq \Sigma_U$, $A_U \in F_1$, and $R_U \subseteq \Sigma_U^* \# \Sigma_U^* \$ \Sigma_U^* \# \Sigma_U^*$, $R_U \in F_2$, is said to be a universal splicing system of type (F_1, F_2), if for every $\gamma = (\Sigma, T, A, R)$ of type (F_1', F_2'), $F_1', F_2' \subseteq \mathcal{L}_0$, there exists a language A_γ such that $A_U \cup A_\gamma \in F_1$ and $L(\gamma) = L(\gamma_U')$, where $\gamma_U' = (\Sigma_U, T, A_U \cup A_\gamma, R_U)$.

Note that the type of the universal system is fixed, but the universal system is able to simulate systems of any type (F_1', F_2'), when F_1' and F_2' are families of type–0 languages, that is, languages with a computable characteristic function. Based on Definition 4.1 we are now in position to state the main universality result.

Theorem 4.2 *For every given alphabet T there exists a splicing system, with finitely many axioms and finitely many rules, that is universal for the class of systems with the terminal alphabet T.*

The proof is based on Theorem 4.1 and on the existence of universal type–0 grammars (or, equivalently, universal Turing machines). For the details of the proof the reader is referred to [16]. Another proof, based on the fact that a language generated by a Post system can be generated by a splicing system, can be found in [14].

The interpretation of Theorem 4.2 from the point of view of DNA computing is that, theoretically, there exist *universal programmable DNA computers* based on the splicing operation. A program consists of a single string to be added to the axiom set of the universal computer. The program has multiplicity one, while an unbounded number of the other axioms is available. The "fixed" axioms of the computer can be interpreted as the "energy" that has to be constantly supplied to the DNA computer for running the programs. The only bio–operations used in these computers are splicing (cut/ligate) and extraction (which in mathematical terms amounts to the intersection of the result with T^*, where T is the terminal alphabet). In the case of splicing systems, we can conclude that Theorem 4.2 provides an affirmative answer to the second question posed in Section 2 with regards to the existence of programmable DNA computers.

Results analogous to Theorem 4.1 and Theorem 4.2 have been obtained for several variants of the splicing systems model presented in Section 3. For example, similar results hold if the condition of the axiom set to be a multiset is replaced by a control condition: a splicing rule is applicable only when certain strings, called *permitting contexts*, are present in the terms of splicing (see [16]).

Constructions showing how to simulate the work of a Turing machine by a DNA model of computation have also been proposed in [40], [37], [2], [8], [6], [46], [36]. In an optimistic way, one may think of an analogy between these results and the work on finding models of computation carried out in the 30's, which has laid the foundation for the design of the electronic computers. In a similar fashion, the results obtained about the models of DNA computation show that programmable DNA computers are not science fiction material, but the reality of the near future.

5 Meta–thoughts on biomathematics

We have seen in Section 2 that the bio–operations are quite different from the usual arithmetical operations. Indeed, even more striking than the quantitative differences between a virtual DNA computer and an electronic computer (the DNA computer winning the comparison on most fronts) are the qualitative differences between the two.

DNA computing is a new way of thinking about computation altogether. Maybe this is how nature does mathematics: not by adding and subtracting, but by cutting and pasting, by insertions and deletions. Perhaps the primitive functions we currently use for computation are just as dependent on the history of humankind, as the fact that we use base 10 for counting is dependent on our having ten fingers. In the same way humans moved on to counting in other bases, maybe it is time we realized that there are other ways to compute besides the ones we are familiar with.

The fact that phenomena happening inside living organisms (copying, cutting and pasting of DNA strands) could be computations in disguise suggests that life itself may consist of a series of complex computations. As life is one of the most complex natural phenomena, we could generalize by conjecturing the whole cosmos to consist of computations. The differences between the diverse forms of matter would then only reflect various degrees of computational complexity, with the qualitative differences pointing to huge computational speed–ups. From chaos to inorganic matter, from inorganic to organic, and from that to consciousness and mind, perhaps the entire evolution of the universe is a history of the ever–increasing complexity of computations.

Just imagine. Perhaps all there was in the beginning was a universal cocktail of particles. They combined randomly for millions of years, until, by chance, some patterns of beautiful mathematical symmetry started to emerge: the inorganic matter. They continued to mix and intermingle until some formations started to self-replicate (see fractals and iterated functions) and then to do computations: life appeared. The more complex the computations grew, the more complex the life forms became, until there was again a sudden leap and consciousness and mind appeared, apparently out of thin air, but in reality an inevitable corollary to complexity. Who knows what the next step could be in this infinite spiral of mathematical evolution?...

Of course, the above is only a hypothesis, and the enigma whether modern man is "homo sapiens" or "homo computans" still awaits solving. But this is what makes DNA computing so captivating. Not only may it help compute faster and more efficiently, but it stirs the imagination and opens deeper philosophical issues. What can be more mesmerizing than something that makes you dream?

To a mathematician, DNA computing tells that perhaps mathematics is the foundation of all there is. Indeed, mathematics has already proven to be an intrinsic part of sciences like physics and chemistry, of music, visual arts (see

[23]) and linguistics, to name just a few. The discovery of DNA computing, indicating that mathematics also lies at the root of biology, makes one wonder whether mathematics isn't in fact the core of all known and (with noneuclidean geometry in mind) possible reality.

Why not? Sometimes a graceful move of a dancer seems to hide the truth of a remarkable theorem, to be the fluid graph of a function with properties of amazing depth. The more profound the mathematics behind is, the more striking the beauty. I may discover a (little and insignificant) theorem once in a while, but she is able to create them by the dozen, theorem after theorem, function after function with breathtaking properties, just by moving an arm or hand, just by smiling. The beauty seems ephemeral, but is reproducible and therefore as eternal as the underlying mathematical truth.

Maybe indeed, Plato was right: Truth, Beauty and Good are one and the same. Maybe indeed, [35], the material things are mere instances of "ideas" that are everlasting, never being born nor perishing. By intimating that – besides everything else – mathematics lies at the very heart of life, DNA computing suggests we take Plato's philosophy one step further: the eternal "ideas" reflected in the ephemeral material world could be mathematical ones.

If this were the case, and the quintessence of reality is the objective world of mathematics, then we should feel privileged to be able to contemplate it.

Acknowledgements. This article has benefited from discussions with Leonard Adleman, Greg Gloor, Tom Head, Jarkko Kari, Gheorghe Paun, Clive Reis, Arto Salomaa, Gabriel Thierrin and Gary Walsh, to all of whom I wish to express my gratitude. It would be impossible to separately point out to all written sources that have influenced this paper. Adopting therefore Seneca's maxim *"Whatever is well said by anyone belongs to me.", [39]*, I want to thank all the authors in the references.

References

[1] L.Adleman. Molecular computation of solutions to combinatorial problems. *Science* v.266, Nov.1994, 1021– 1024.

[2] L.Adleman. On constructing a molecular computer, *ftp: /ftp/pub/csinfo/papers/adleman/molecular_computer.ps.*

[3] E.Baum. Building an associative memory vastly larger than the brain. *Science*, vol.268, April 1995, 583–585.

[4] D.Beaver. The complexity of molecular computation. *http://www.transarc.com/~beaver/research/alternative/molecute/molec.html.*

[5] D. Beaver. Computing with DNA. *Journal of Computational Biology*, (2:1), Spring 1995.

[6] D.Beaver. Molecular computing.
http://www.transarc.com/~beaver/research/alternative/molecute/molec.html.

[7] D.Boneh, R.Lipton, C.Dunworth. Breaking DES using a molecular computer. *http://www.cs.princeton.edu/~dabo*.

[8] D.Boneh, R.Lipton, C.Dunworth, J.Sgall. On the computational power of DNA. *http://www.cs.princeton.edu/~dabo*.

[9] E.Csuhaj–Varju, R.Freund, L.Kari, G.Paun. DNA computing based on splicing: universality results. *First Annual Pacific Symposium on Biocomputing*, Hawaii, 1996, also *http://www.csd.uwo.ca/~lkari*.

[10] E.Csuhaj–Varju, L.Kari, G.Paun. Test tube distributed systems based on splicing, *http://www.csd.uwo.ca/~lkari*.

[11] K.Culik, T.Harju. Splicing semigroups of dominoes and DNA. *Discrete Applied Mathematics*, 31(1991) 261–277.

[12] J.Dassow, V.Mitrana. Splicing grammar systems. *Computers and AI*. To appear.

[13] K.L.Deninnghoff, R.W.Gatterdam. On the undecidability of splicing systems. *International Journal of Computer Mathematics*, 27(1989) 133–145.

[14] C.Ferretti, S.Kobayashi, T.Yokomori. DNA splicing systems and Post Systems. *First Annual Pacific Symposium on Biocomputing*, Hawaii, 1996.

[15] R.Freund. Splicing systems on graphs. *Proc.Intelligence in Neural and Biological Systems*, IEEE Press, May 1995, 189–194.

[16] R.Freund, L.Kari, G.Paun. DNA computing based on splicing: the existence of universal computers. T.Report 185–2/FR–2/95, TU Wien, Institute for Computer Languages, 1995, also *http://www.csd.uwo.ca/~lkari*.

[17] M.Garey, D.Johnson. *Computers and Intractability. A Guide to the Theory of NP–completeness.* W.H.Freeman and Company, San Francisco, 1979.

[18] R.W.Gatterdam. Splicing systems and regularity. *International Journal of Computer Mathematics*, 31(1989) 63–67.

[19] R.W.Gatterdam. DNA and twist free splicing systems, in *Words, Languages and Combinatorics II*, Eds.:M.Ito and H.Jürgensen, World Scientific Publishers, Singapore, 1994, 170–178.

[20] D.K.Gifford. On the path to computation with DNA. *Science* 266(Nov.1994), 993–994.

[21] T.Head. Formal language theory and DNA: an analysis of the generative capacity of recombinant behaviors. *Bulletin of Mathematical Biology*, 49(1987) 737–759.

[22] T.Head. Splicing schemes and DNA, in: *Lindenmayer systems – Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*. Springer Verlag, Berlin 1992, 371–383.

[23] D.Hofstadter. *Gödel, Escher, Bach: an eternal golden braid*, New York, Basic Books, 1979.

[24] F.Hoppensteadt. Getting started in mathematical biology. *Notices of the AMS*, vol.42, no.9, Sept.1995.

[25] R.Lipton. DNA solution of hard computational problems. *Science*, vol.268, April 1995, 542–545.

[26] R.Lipton. Using DNA to solve NP–complete problems. *http://www.cs.princeton.edu/~dabo*.

[27] R.Lipton. Using DNA to solve SAT. *http://www.cs.princeton.edu/~dabo*.

[28] R.Lipton. Speeding up computations via molecular biology. Manuscript.

[29] G.Paun. On the splicing operation. *Discrete Applied Mathematics*, to appear.

[30] G.Paun. On the power of the splicing operation. *International Journal of Computer Mathematics*, to appear.

[31] G.Paun. Regular extended H systems are computationally universal. *Journal of Information Processing and Cybernetics, EIK*, to appear.

[32] G.Paun, G.Rozenberg, A.Salomaa. Computing by splicing. Submitted, 1995.

[33] G.Paun, A.Salomaa. DNA computing based on the splicing operation. Submitted, 1995.

[34] D.Pixton. Linear and circular splicing systems. *Proc.Intelligence in Neural and Biological Systems*, IEEE Press, May 1995, 181–188.

[35] Plato. *Great Dialogues of Plato*. The New American Library, New York, 1956.

[36] J.Reif. Parallel Molecular Computation. To appear in SPAA'95, also at *http://www.cs.duke.edu/~reif/HomePage.html*.

[37] P.Rothemund. A DNA and restriction enzyme implementation of Turing machines. Abstract at *http://www.ugcs.caaltech.edu/~pwkr/oett.html*.

[38] A.Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[39] L.Seneca. *Letters from a stoic*, Harmondsworth, Penguin, 1969.

[40] W.Smith, A.Schweitzer. DNA computers in vitro and in vivo, *NEC Technical Report 3/20/95*.

[41] A.M.Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc.London Math Soc.*, Ser.2, 42(1936), 230–265.

[42] A.Yasuhara. *Recursive Function Theory and Logic*. Academic Press, New York, 1971.

[43] T.Yokomori, S.Kobayashi. DNA evolutionary linguistics and RNA structure modeling: a computational approach. *Proc.Intelligence in Neural and Biological Systems*, IEEE Press, May 1995, 38–45.

[44] T.Yokomori, S.Kobayashi, C.Ferretti. On the power of circular splicing systems and DNA computability. Rep. CSIM–95–01, University of Electro–Communications, Dept.of Comp.Sci. and Information Mathematics, Chofu, Tokyo 182, Japan.

[45] E.Winfree. Complexity of restricted and unrestricted models of molecular computation. *http://dope.caltech.edu/winfree/DNA.html*.

[46] E.Winfree. On the computational power of DNA annealing and ligation. *http://dope.caltech.edu/winfree/DNA.html*.

[47] W.Wordsworth. The tables turned. In *Wordsworth's Poems*, vol.1, Ed.P.Wayne, Dent, London, 1965.