Università di Pisa
Dipartimento di Informatica

Technical Report : TR-97-07

# A survey on DNA computing

*Nadia Pisanti*

April 16, 1997

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

# A survey on DNA computing*

*Nadia Pisanti*

TR-97-07
Department of Computer Science
University of Pisa
Corso Italia, 40
56125 Pisa, Italy
e-mail: pisanti@di.unipi.it

April 1997

**Abstract**

The aim of this report is to make a review on *DNA computing*: molecular biology is used to suggest a new way of solving a $\mathcal{NP}$-complete problem. The idea (due to Leonard Adleman in [Adl94]) is to use strands of DNA to encode the (instance of the) problem, and to manipulate them using techniques commonly available in any molecular biology laboratory, in order to simulate operations that select the solution of the problem, if it exists. After Adleman's paper—appeared on *Science* in November 1994—many authors have been interested in DNA computing. We will try to give a description of the discussions and the results which appeared in literature.

*This work was supported in part by National Research Council of Italy.

# Contents

# 1 Introduction

DNA computing must not be confused whit biocomputing. Usually, biocomputing means everything that computer scientists can do to help biologists to study genes. For example, algorithms and data structures have been developed to investigate the properties of the sequences of nucleotides in DNA or RNA, and those of amino acids in the primary structure of a protein. In DNA computing, instead, molecular biology is suggested to solve a problem for computer scientists.

We will summarize basic biochemistry in section 2. This should be sufficient to allow computer scientists to understand the remainder of this work. In section 3, DNA-computing is introduced with the description of Adleman's experiment and Lipton's molecular solution of the SAT problem. Section 4 addresses the serious danger of errors that can take place in a DNA computation. In section 5, two molecular implementations of Turing Machines are presented, as a more powerful demonstration of the computational properties of molecules. After that, section 6 underlines some crucial aspects of DNA computing that take to its particular interest in storing and *associating* information. Section 7 presents a high level model for DNA computations and some examples of algorithms described by that model. Finally—in section 8—three other models and their characteristics are described: the Sticker Model that suggests a different way of storing information, the PAM Model that underlines molecular parallelism, and the Splicing Systems that represent DNA computations using the formal languages theory.

# 2 A brief biological background

For a more detailed description of the notions contained in this brief section, the reader can consult (some chapters of) [WHR$^+$87].

## 2.1 DNA and proteins

Proteins are the fundamental agents of life; every human being contains something like $100,000$ different proteins. Their properties and their interaction between each other determine the way we are. Enzymes are proteins, and they are the molecules that cause all the chemical transformations that take place in a cell. The information that defines the primary structure[1] of every

[1] The first product of the protein synthesis. The final structure of the protein (that is the one that determines its function) is the result of the interactions between the primary

protein is encoded in the DNA (deoxyribonucleic acid). In fact, the protein (in its primary structure) is a linear sequence of a combination of 20 different amino acids. DNA is a double stranded sequence of 4 nucleotides, and a portion of DNA sequence (a gene) encodes the information that determines the sequence of amino acids of the protein encoded by that particular gene. It is for this reason that scientists use the expression *genetic code*. In fact, DNA contains the genetic information that defines the proteins, the *engines* of life.

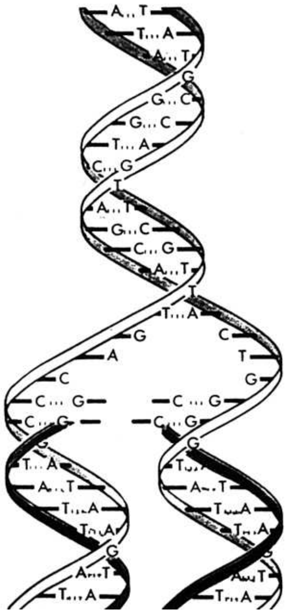

Figure 1: Autoreplication of DNA.

The four nucleotides that compose a strand of DNA are: adenine (A), guanine (G), cytosine (C) and thymine (T); they are often called *bases*. The chemical structure of DNA (the famous double helix) was discovered by James Watson and Francis Crick in 1953. It consists in a particular bond of two linear sequences of bases. This bond follows a property of complementarity: adenine bonds with thymine and viceversa, and cytosine bonds with guanine and viceversa. This is known as *Watson-Crick complementarity* and it is also denoted by bars: $\overline{A} = T$, $\overline{T} = A$, $\overline{C} = G$, $\overline{G} = C$. Each DNA strand has two different ends that determine its polarity: the $3'$ end, and the $5'$ end. The double helix is an antiparallel (two strands of opposite polarity) bonding of two complementary strands (see figure 1, that also shows how

structure and its environment. Understanding this process in details is an open problem for biologists.

DNA autoreplicates itself).

## 2.2 Recombinant DNA

In the last years, many techniques have been developed in order to study and manipulate DNA in a lab, for various biological applications. The advances in molecular biology technology allow recombinant DNA operations to be routine in all the molecular biology laboratories, while these techniques were once considered very sophisticated.

We now describe some of the *operations* that are available on DNA strands; the aim of the following description (that involves various kinds of operation) is to give the reader an idea of the possible implementation of the algorithms that appear in DNA computing literature, which we will describe in the next sections:

- *Annealing* and *melting*: the hydrogen bonding between two complementary sequences is weaker than the one that links nucleotides of the same sequence. It is possible to pair (*anneal*) two antiparallel and complementary single strands and it is possible to separate (*melt*) them, obtaining two single strands from a double one. These operations are realized creating adequate conditions of temperature, pH, etc.

- *Synthesis*: A desired strand of DNA can be *synthesized* in lab. This is possible for strands up to a certain length. Longer 'random' strands are available. They consist in DNA sequences which have been cloned from many different organisms. They can be easily produced in large quantities.

- *Extraction*: Given a test-tube $T$ and a strand $s$, it is possible to *extract* all the strands in $T$ that contain $s$ as a subsequence, and to separate them from those that do not contain it.

- *Union*: Given two or more test-tubes, say $T_1, \ldots, T_n$, it is possible to put in a new test-tube the *union* of all the strands contained in $T_1, \ldots, T_n$.

- *Amplify*: The content of a test-tube can be *amplified*, giving a duplication of all the strands in it.

- *Polimerize*: A single strand that has a portion of double stranded subsequence, can be *polimerized*. The result is an entire double stranded molecule (the opportune complementary sequences are synthesized and annealed to the original strand).

- *Ligation*: There is an enzyme called *Ligase*, that causes the concatenation of two sequences in a unique strand.
- *PCR*: It is an acronym for *Polymerase Chain Reaction*; it is not just an operation, but a machine that allows to realize some operations on DNA strands. One of them is the *amplify*, that we have already described; another one allows to select, in a test-tube, all the strands that begin (or end) with a given subsequence.
- *Detect*: Given a test-tube $T$, we denote with *detect* the lab operation that checks if $T$ contains at least a DNA strand.
- *Length*: With a technique that uses a particular gel, it is possible to separate strands of different *length*.
- *Restriction emzymes*: There are, as we have introduced in 2.1, many kind of enzymes that are molecules capable of operating on other molecules. Some of them are the *restriction enzymes* that cut DNA double strands where specific subsequences appear.

# 3 Adleman and Lipton's model

## 3.1 Adleman's experiment

As we have introduced in section 1, Adleman's experiment—described in [Adl94]—suggests the feasibility of carrying out computations at the molecular level: a 'biological solution' for an instance of the *direct Hamiltonian path problem*[2] is presented. Figure 2 shows a graph where, if $v_{in} = 0$ and $v_{out} = 6$, there is the Hamiltonian path $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$.

As known, the problem of the Hamiltonian path is an $\mathcal{NP}$-complete one. Thus, as long as nobody demonstrates that $\mathcal{P} \neq \mathcal{NP}$, it does not exist a polinomial deterministic algorithm that solves it. Anyway, there are nondeterministic polinomial algorithms; the one that follows is one of them:
Given a graph $G = (V, E)$ and two designated vertices $v_{in}, v_{out} \in V$,

[2]A directed graph $G$ with designated vertices $v_{in}$ and $v_{out}$ is said to have an Hamiltonian path if and only if there exists a path that begins at $v_{in}$, ends at $v_{out}$, and enters every other vertex exactly once.

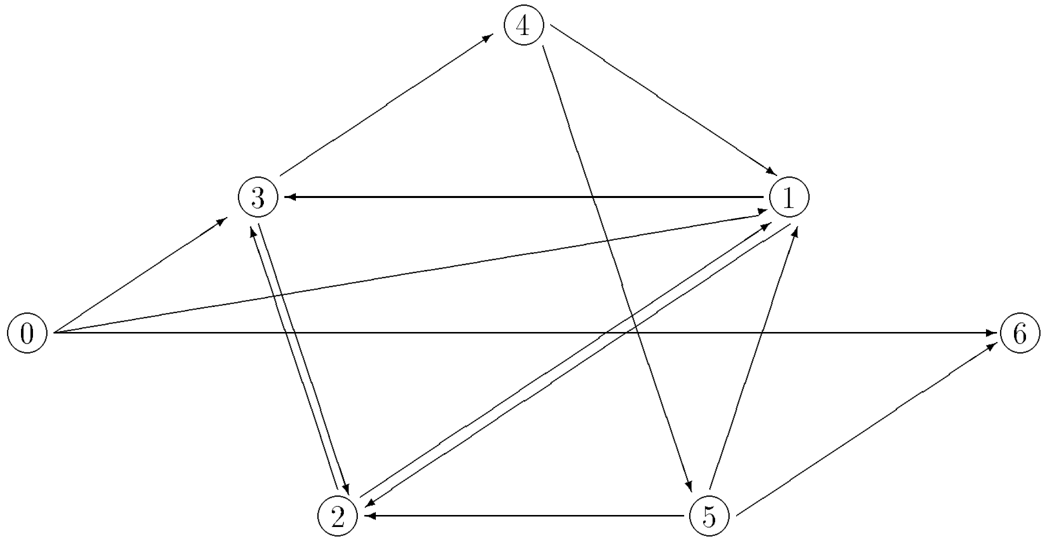


Figure 2: A directed graph with $n$ vertices.

1. Generate random path through $G$.
2. Keep only those paths that begin in $v_{in}$ and end in $v_{out}$.
3. Keep only the paths that enter exactly in $n$ vertices (where $n = \#V$).
4. Keep only paths that enter each vertex at least once.
5. Detect if any path survived the selection. If so, the answer is 'yes', otherwise the answer is 'no'.

This algorithm is the one that Adleman realized in a molecular biology lab. The trick of his implementation lies in the way he suggests to encode the instance of the problem: for every vertex $i$ in the graph $G$, a random strand of DNA of length 20 is synthesized and denoted with $O_i$. Then, for every edge $(i \to j)$ in $G$, an opportune strand in synthesized: the $3'$ half of $O_i$ and the $5'$ half of $O_j$ are concatenated, as shown in figure 3(a). In this way, when we put in a test-tube, for example, copies of the oligonucleotides $O_{i \to j}$, $O_{j \to k}$ and $\overline{O}_j$, the strands will anneal, as it is shown in figure 3(b), creating double strands that encode paths through the graph (one of the two strands is the concatenation of the substrands that encode adjacent edges, and the other one is the concatenation of the complementary oligos associated with the vertices which the path enters).

(a)

$$
\begin{array}{ll}
O_2 & \overbrace{TATCGGATCG}^{x_2}\overbrace{GTATATCCGA}^{y_2} \\
O_3 & \overbrace{GCTATTCGAG}^{x_3}\overbrace{CTTAAAGCTA}^{y_3} \\
O_4 & \overbrace{GGCTAGGTAC}^{x_4}\overbrace{CAGCATGCTT}^{y_4} \\
O_{2\to 3} & \overbrace{GTATATCCGA}^{y_2}\overbrace{GCTATTCGAG}^{x_3} \\
O_{3\to 4} & \overbrace{CTTAAAGCTA}^{y_3}\overbrace{GGCTAGGTAC}^{x_4} \\
 & \\
\overline{O}_3 & CGATAAGCTCGAATTTCGAT
\end{array}
$$

(b)

$$
\begin{array}{c}
\overbrace{GTATATCCGAGCTATTCGAG}^{O_{2\to 3}}\overbrace{CTTAAAGCTAGGCTAGGTAC}^{O_{3\to 4}} \\
\underbrace{CGATAAGCTCGAATTTCGAT}_{\overline{O}_3}
\end{array}
$$

Figure 3: Adleman's idea to encode the graph.

Now we can describe the biological realization of the nondeterministic algorithm we presented at the beginning of this section; we will explain the way Adleman suggested to realize each of the five steps of the algorithm:

1. For each vertex $i \in V$ and for each edge $(i \to j) \in E$, an opportune number of copies[3] of $\overline{O_i}$ and $O_{i\to j}$ are mixed together with the Ligase enzyme (see section 2.2). So, the ligation reaction results in the formation of DNA strands that encode for random paths through the graph.

2. The molecules created in the previous step are amplified by polymerase chain reaction using *primers* $O_0$ and $\overline{O_6}$[4].

[3]Adleman used 50 pmol of each oligonucleotide that means $3\times 10^{13}$ copies; this quantity exceeded what was necessary for the graph in figure 2 (the instance solved in Adleman's experiment), so this condition ensures that many DNA molecules encoding the Hamiltonian path were created.

[4]In this way, DNA molecules that encode for paths that begin at $v_0$ (that is $v_{in}$) and ends at $v_6$ (that is $v_{out}$) are selected.

3. With a technique that uses *agarose gel* (see the operation *length* in section 2.2) the molecules that encode paths that enter exactly $n$ vertices are selected. In Adleman's instance, it means to extract the molecules which are 140 base pair[5] long.

4. For each vertex $i$ of the graph, only paths that enter $v_i$ at least once are kept: the double stranded product of step 3 is *melted* creating single strands and, for each $i$, copies of $\overline{O_i}$ are synthesized and mixed with them, keeping only the molecules that anneal (they are those that contain the substrand $O_i$, that is they encode paths that enter $v_i$).

5. If any DNA molecule survives step 4, then it encodes a Hamiltonian path in the graph; the *detect* operation verifies if there is such a solution.

This experiment demonstrates the feasibility of a biological solution for the Hamiltonian path problem; this computation—executed by Adleman in lab—required a week of lab work. This fact appears discouraging, but another fact is more comforting: the number of required lab operations grows linearly with $n$, that is the number of vertices of the graph. The number of different molecules required for the computation is linear in the number $m$ of the edges of $G$. In fact, step 1 requires the synthesis of $n + m$ different oligos (a linear quantity); step 4 requires to repeat a procedure $n$ times, and the other three steps take constant time to be executed.

The generation of molecules that encode entire paths is obtained *gratis* by molecular parallelism (it takes few hours of incubation in a test-tube). We remind our readers that each operation described in section 2.2 can be applied to an entire test-tube and—thus—simultaneously to all the molecules it contains. In particular—for example—the amplify operation that duplicates the content of a test-tube, allows to generate $2^n$ copies of one or more oligos in $n$ steps. DNA molecules are manipulated in test-tubes where they lie—for example—in water. So, if an algorithm needs an exponential number of strands to encode the data, we have to fix an upper bound to the dimension of the input in order to evaluate the 'feasibility' of DNA algorithms: in [BDLS95], the authors assume that $10^{21}$ is the upper bound to the number of DNA strands that an algorithm can treat[6]. So, as suggested in [Har95b], in molecular computing the exponential barrier is the *weight* barrier, because it

[5] With $n = 7$ and oligos of 20 nucleotides, the paths that enter exactly $n$ vertices are encoded by molecules of length $7 \times 20 = 140$ bp.

[6] $10^{21} \approx 2^{70}$. These data consider strands 20bp long; obviously, longer strands would take more space. Anyway, the length of the strands usually grows linearly with the size of the problem, while their number can grow exponentially.

is the weight that impose a limit to the volume of a test-tube. Thus, Adleman's experiment could theoretically be executed for a graph with up to 70 vertices, but—as shown in [Har95b]—if we had to execute the experiment for a graph with 200 vertices, we would need to manipulate DNA molecules for a weight heavier than that of our planet. Anyway, the potential of molecular computation is impressive because storing information in molecules of DNA allows for an information density of approximately 1 bit per cubic nanometer, improving over existing storage media (such as videotapes) which store information at a density of approximately 1 bit per $10^{12}\text{nm}^3$ ([Adl94]).
Adleman's solution of the Hamiltonian path problem interested many researchers. They arranged a meeting at Princeton University on April 1995 (and a second one in 1996). Adleman detected ([Adl95]) two types of obstructions to creating a practical molecular computer:

- Physical obstructions due to difficulty in dealing with large scale systems with errors.
- Logical obstructions arising the versatility of molecular computers and their capacity to efficiently solve a wide variety of computational problems.

We will treat the problem of errors in section 4 and we have already told about weight/volume upper bounds. As far as logical obstructions are concerned, the remainder of this report will try to describe the proposals that are presented in literature in order to address them. We begin describing Lipton's solution to the SAT problem, a first example of an extension of Adleman's DNA computation.

## 3.2 Lipton's solution to SAT problem

Although the Hamiltonian path problem is an $\mathcal{NP}$-complete one, Adleman's result of [Adl94] does not demonstrate the feasibility of solving all instances of $\mathcal{NP}$-complete problems. Lipton ([Lip95b]) extended Adleman's idea to solve the problem of the satisfiability of a boolean formula (usually called SAT problem). The SAT problem consists in finding boolean values to $n$ variables $x_1, \ldots, x_n$ that make true a formula $F$. In Lipton's algorithm $F$ is assumed to be of the form

$$F = C_1 \wedge \ldots \wedge C_m$$

where each $C_i$ is a clause of the form $v_1 \vee \ldots \vee v_k$, and each $v_j$ is one of the $n$ variables or its negation. Lipton suggests a molecular way of solving this problem. The solution uses an initial set of DNA strands that—given $n$, the number of the variables—is the same for every possible instance of the problem. In this sense, Lipton's algorithm is more *general* than Adleman's one. The initial test-tube $T_0$ contains DNA strands that encode all the $2^n$ possible assignments of values to the $n$ variables. We will not describe in details the way they are encoded (the interested reader can consult [Lip95b]), but this initialization is possible and it takes linear time to be done. The content of $T_0$—that is the solution space of the SAT problem—corresponds to the set of strands that encode all the Hamiltonian paths of the graph $G_n$ illustrated in figure 4 (given that if the path enters $x'_i$ to go from $x_i$ to $x_{i+1}$, then it means that the variable $x_{i+1}$ is assumed to be $true$ and, viceversa, if the path enters $x_i$ then it means that $x_{i+1} = false$).

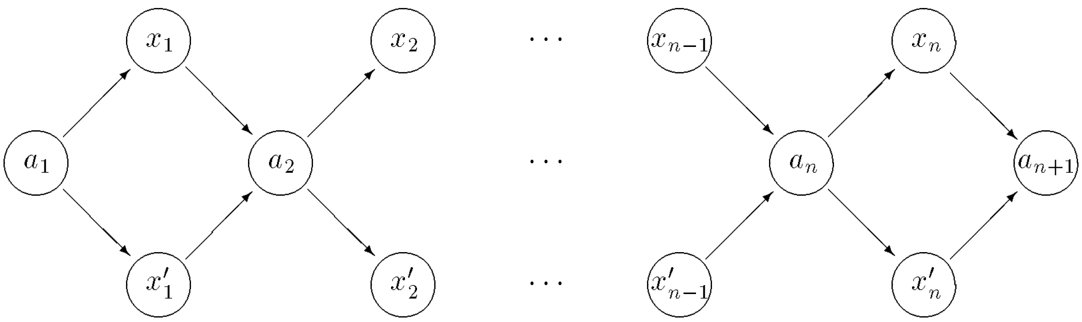


Figure 4: The graph $G_n$.

Beginning with $T_0$, the computation goes on generating the test-tubes $T_1, \ldots, T_m$ where $T_i$ contains only the strands that encode assignments that verify all the clauses $C_1, \ldots, C_i$. This is realized in the following way: suppose we have $T_{i-1}$ that contains all the assignments of boolean values that verify $C_1 \wedge \ldots \wedge C_{i-1}$, and suppose that—for example—$C_i = x_p \vee \neg x_q$. $T_i$ is generated as follows: strands corresponding to assignments in which $x_p = true$ are extracted from $T_{i-1}$ (after this one is duplicated) and they are kept in a temporary test-tube $T_1$. Similarly, strands that encode assignments in which $x_q = false$, are put in a new test-tube $T_2$. Then $T_i = T_1 \cup T_2$, and so on. Obviously, the final $T_m$ contains strands that encode the solutions to the SAT problem, if they exist. At the end of the computation, a detect operation verifies this condition.

This computation requires linear time to be executed because the generation of $T_0$—as we said—takes linear time, and then a number of steps proportional to the number of literals of the formula is required, plus one that is the final detect.

### 3.3 Other solutions of $\mathcal{NP}$ problems

After the papers of Adleman and Lipton, many other authors suggested DNA algorithms to solve $\mathcal{NP}$-problems. Adleman himself also suggested—in [Adl95]—a linear solution (in the number of the edges of the graph) for the *3-colorability* problem, using the same few operations of the solution of the Hamiltonian path problem. Equally, Lipton—in the same paper in which the solution of the SAT problem appeared—suggested how to extend his method to solve the SAT problem for contact networks. Different models of DNA computations have been proposed to solve various kinds of problem. We will describe some of them in section 8. Some of the proposals made in order to solve $\mathcal{NP}$ problems are summarized in table 3.1, where we indicate—for each result—the reference and the two parameters that evaluate the algorithm: first, the number of required biological operations (*bio-steps*) and—second—the number of distinct strands that may appear in the same test-tube during the computation.

| problem | bio-steps | strands | reference |
|---|---|---|---|
| (1) Direct Hamiltonian path | $O(n)$ | $n!$ | [Adl94] |
| (2) 3-colorability problem | $O(m)$ | $3^n$ | [Adl95] |
| (3) FNC formula satisfiability | $O(s)$ | $2^n$ | [Lip95b] |
| (4) Contact network satisf. | $O(s)$ | $2^n$ | [Lip95b] |
| (5) Circuit satisfiability | $O(s)$ | $2^n$ | [BDLS95] |
| (6) MAX-circuit satisfiability | $O(s)$ | $2^n$ | [BDLS95] |
| (7) Regular circuit satisf. | $O(s)$ | $2^n$ | [BDLS95] |
| (8) 1-tape NTM | $O(t)$ | $2^N$ | [Bea95] |
| (9) Cellular automata | 1 | $t * S$ | [Win95b] |

**Table 1.1**

For each result appearing in table 1.1, a brief explanation follows:

(1) It is Adleman's experiment, already described in section 3.1. With $n$ we indicate the number of vertices of the graph.

(2) It is another result due to Adleman. The biological operations used in this algorithm are simply those of the *restricted model*[7] (extraction, union

[7]See [Adl95].

and detect, beyond annealing and melting). The number of required steps grows with the number $m$ of the edges of the graph. Finally, with $n$ we indicate the number of vertices of the graph.

(3) It is Lipton's result. We have described it in section 3.2.

(4) In the same paper where the previous result appeared, Lipton suggested how to extend it to the solution of the SAT problem for a contact network, that is a directed graph with two special vertices $s$ (source) and $t$ (sink). Each edge is labeled with either $x$ or $\overline{x}$, where $x$ is a variable. Given an assignment of values to the boolean variables, an edge is considered to be connected if the edge's formula has value *true*. The solution needs a number of steps that is linear in the size of the formula.

(5) This result appears in [BDLS95]. It improves Lipton's results, since the SAT for circuits is a more general problem. As in (6) and (7), $s$ is the number of gates of the circuit and $n$ is the number of input variables.

(6) It is a solution to an optimization problem, that is finding a satisfying assignment for a boolean circuit in which the largest number of variables is set to *true*.

(7) This result increases the efficiency of the two previous algorithms. Details about this algorithm can be found in [BDLS95].

(8) The result we refer to is due to Donald Beaver ([Bea94], [Bea96]). It is only one of the proposals of realizing a Turing Machine via DNA ([Rei95], [RW95], [Rot96]). We will see these kinds of result in section 5 and in section 7.2.

(9) The most interesting aspect of this result is the constant number of required biological operation. The trick of this efficiency is the following: the cellular automata is encoded associating each rule of the form $(x,y) \longrightarrow (f(x,y), g(x,y))$ to a particular complex DNA molecule. It has a planar structure with four sticky ends associated to $x, y$ at one side, and to $f(x,y)$ and $g(x,y)$ at the other. The computation goes on by itself *in vitro* because—after an opportune initialization—such molecules anneal between each other simulating the application of the rules of the automata. The number of different DNA molecules that may appear in the test-tube grows proportionally to the product of $S$ (the space used by the automata) and $t$ (the number of generations for which it is run).

# 4 Errors

## 4.1 The operation of extraction

We have described biological computations assuming that the operations used by Adleman and Lipton models were *perfect*; this is not true. Errors take place during the manipulation of DNA strands. In this sense, the most *dangerous* operation—among those used by Adleman and Lipton—is the extraction. Let the reader imagine what would happen if a 'good' path were lost during one of the extraction operations in step 4 of Adleman's experiment; it is for this reason that Adleman suggests to amplify (more than once) the content of the test-tube at each step of his experiment; in this way, the loss of a strand that encodes a 'good' path would not be catastrophic, because one of its copies should survive. The opposite error—to take a 'bad' path as if it were a 'good' one—is less dangerous, because the solution could be verified at the end of the computation.

Formally, let the extraction operation be the following:

$$\text{Extraction} :(T, s) \longrightarrow \{+T, -T\}$$

that is, given a test-tube $T$ and a strand $s$, the extraction operation gives two test-tubes: $+T$ contains all the strands in $T$ that have $s$ as a substrand, and $-T$ contains all the others. Let $\epsilon$ be the probability that a strand that should end in $+T$, ends in $-T$ and—viceversa—let $\gamma$ be the probability that a strand ends erroneously in $+T$. In [Adl95], Adleman—analyzing in details the procedure that realizes the extraction in lab—estimated the values for these two probabilities: $\epsilon \approx 10^{-1}$ and $\gamma \approx 10^{-6}$. Obviously, they are too high to be acceptable. A first possible way of decreasing $\epsilon$'s value—the most worrying—was suggested in [Adl95]. The idea is to repeat the operations in the following way:

$$
\begin{array}{lll}
(1) & T_1 := +T & T'_1 := -T \\
(2) & T_2 := +T'_1 & T'_2 := -T'_1 \\
. & & \\
. & & \\
. & & \\
(n) & T_n := +T'_{n-1} & T'_n := +T'_{n-1} \\
(n+1) & T^+ := \cup(T_1, T_2, \ldots, T_n) & \\
 & T^- := T'_n &
\end{array}
$$

After these $n + 1$ steps, the result is the realization of the operation

$$\text{Superextract} : \ (T, s) \longrightarrow \{T^+, T^-\}$$

where the final $T^+$ contains $(1-\epsilon^n)$ of the elements that should end in $+T$ (the test-tube obtained with the normal extraction operation). With an opportune choice of $n$, we can have the desired probability that all the strands which contain $s$ appear in $T^+$. If it were—for example—$n = 6$, we would have $\epsilon = 10^{-6}$ and $\gamma = 6/10^6$. In this way, we could assume $\gamma = \epsilon = 6/10^6$.

The danger of errors in the extraction operation was also examined by Richard Karp and other authors in [KKW95]. Their result is the following: a computation that needs the execution of $t(n)$ extraction operations, can be executed in

$$O(t(n) \cdot \left\lceil \log_\epsilon \frac{\delta}{t(n)} \right\rceil \cdot \left\lceil \log_\gamma \frac{\delta}{t(n)} \right\rceil)$$

steps, with a probability $\delta$ of error (in fact, $\delta$ is the desired *accuracy* of the computation—the execution of the algorithm fails with a probability lower than $\delta$—with $\delta$ arbitrarily low). In this case as well, the idea is to construct a sequence of *faulty* Extracts that simulates a highly reliable Extract operation.

In [BL96], the following classification of DNA algorithms was suggested: algorithms may be *decreasing volume* if the number of required strands decreases as the execution goes on, they may be *constant volume* if the number of strands does not change during the computation and—finally—they are *mixed* if they do not fall in any of the two other classes. The authors suggested a way of transforming a decreasing volume algorithm in a opportune constant volume one which is much more resistant to errors. The required number of steps is almost the same (only some amplify operations are added), and the increased volume required is—anyway—the one that was necessary at the beginning of the computation in the original (decreasing volume) version of the algorithm. Taking as examples Adleman or Lipton's computation (they are both decreasing volume), the idea is simply to amplify—during the computation—the content of the test-tube that contains the space solution, filling the space that the selection of each step leaves. In this way—during the computation—it would be very difficult to lose a strand that encodes a solution to the problem. Another modification is suggested about the final detect: it could be replaced by another operation that (i) takes one of the strands that remained in the final test-tube, (ii) checks whether or not it is a solution and (iii) if it is not, chooses and tries another strand.

## 4.2 Undesired annealings

Most of the causes of errors in biological computations—not only in the extraction operation—are related to the possibility of undesired annealings: a strand $s$ could anneal with one that is similar to $\overline{s}$, but it is not the right one. In this case, we talk about *partial matches*. Another kind of undesired matching could happen between two 'shifted' strands: for example, a strand $xy$ could partially anneal with a strand $\overline{y}z$. Finally, a strand could anneal with itself, losing its linear structure. The probability of all these undesired annealings could be decreased with an opportune choice of the strands used to encode the data of the problem. In [GFB+96], the authors realized some experiments in this sense: given a combinatorial set of DNA molecules—candidate to store information for a hypothetical DNA computation—they addressed the problem of analyzing all the possible undesired matches between the molecules of the set. Anyway, some of these partial matches cannot be eliminated: all combinatorial sets have inherent partial matches, but—obviously—the choice of the set can minimize their number. The authors of [GFB+96] wrote a program that—given a DNA template[8]—calculates the annealing interaction of each member of the template with all the molecules in the set as well as with all the molecules in the complementary set. Their program also calculates the values of opportune parameters that concur to evaluate the quality of a template.

# 5 Simulations of Turing Machines

DNA solutions for $\mathcal{NP}$ problems demonstrate that molecules hide interesting computational properties, but such results would not be so interesting if a molecular computer did not exist. In particular, the reader could ask himself if DNA computing is capable of universal computation. The design of a Turing Machine based on DNA molecules is a step in this direction because it defines a molecular computing system capable of maintaining a state and a memory and of executing an indefinite number of transitions. In this section, we describe two of the proposals made in order to show how DNA computations can simulate the behavior of a Turing Machine.

## 5.1 Beaver's realization of a nondeterministic Turing Machine

In [Bea95] and [Bea96], Donald Beaver designed a Turing Machine consisting of a single DNA strand. At each step of the computation that we are going to describe, a DNA molecule encodes a configuration of the Turing Machine: the content of the tape, the current state and the head position. Each state

[8]For example an oligonucleotide 15 base pair long with 10 fixed positions and 5 variable bases.

transition requires $O(1)$ laboratory steps to be carried out. Finally, we will see how such method can be extended to perform the computation of a nondeterministic Turing Machine.

Let $M = (Q, \Sigma, \delta, s_0)$ be a deterministic Turing Machine where $Q$ is the state set (containing the *halt* state $h$), $\Sigma$ is a finite alphabet, $\delta$ is the state transition function, and $s_0$ is the start state. First, states and symbols are codified using the alphabet of the four nucleotides: $\Sigma_{DNA} = \{A, C, T, G\}$. Since $Q$ and $\Sigma$ are finite sets, such encoding is obviously possible and simple. We denote with the string $\alpha q \sigma \beta$ the configuration of the Turing Machine $M$ that is in the state $q \in Q$, whose head points to the beginning of a string $\sigma \in \Sigma^*$ that has the string $\alpha$ on its left end the string $\beta$ on its right. Let $S(n)$ be the polynomial length of $M$'s tape. The entire configuration of $M$ is encoded in a DNA single strand in the following way: the presence of the substrand $u = E(i, \sigma_i)$ indicates that the symbol $\sigma_i \in \Sigma$ is in the $i^{th}$ position of the tape[9]. In the same way, the presence of the substrand $v = E(j, q)$ indicates that $M$ is in the state $q \in Q$ and that the head points at the $j^{th}$ position of the tape. Thus—for example—the configuration

$$\sigma_1 \sigma_2 \ldots \sigma_{i-1} q \sigma_i \sigma_{i+1} \ldots \sigma_{S(n)}$$

is codified by the DNA strand

$$E(1, \sigma_1) E(2, \sigma_2) \ldots E(i-1, \sigma_{i-1}) E(i, q) E(i+1, \sigma_{i+1}) \ldots E(S(n), \sigma_{S(n)})$$

where each $E(k, \sigma)$ or $E(k, q_h)$ (with $k \in \{1, 2, \ldots, S(n)\}, \sigma \in \Sigma, q_h \in Q$) is an opportune random substrand.

In order to perform a state transition—once a DNA single strands encodes a configuration—the idea is of substituting the portion of the DNA strand that is interested by the transition. Before describing the way a transition of $M$ is executed, we explain the realization of the biological operation that substitutes a portion of a DNA strand.

Let's suppose we have a strand $M = LAXBR$ and a strand $Y$ and we want to generate a new strand $M' = LAYBR$. Obviously, $A, B, L, R, X, Y \in \Sigma_{DNA}^*$. To implement the substitution, we operate in the following way ([Bea95]): first, the single strands $\overline{AYB}$ and $\overline{R}$ are sinthesized and annealed with $M$. The result will be a partially double stranded DNA molecule where the right side ($R$) is annealed with its complementary, the left side ($L$) is single stranded and the central part is such that the molecule $\overline{AYB}$ is partially

[9] Let $E$ be a generic encoding function.

bonded with $AXB$. Note that, the substrands $\overline{B}$ and $\overline{R}$ are not linked yet between themselves, in spite of they annealed with the same single strand $M$. In the next step two enzymes are added: *DNA Polymerase* and *DNA Ligase*. The former—using the fragment $\overline{R}$ as a starting point—polymerazes the molecules: the strand $\overline{L}$ is created and annealed with $M$. The latter seals the nicks, linking in an entire single strand all the substrands that are annealed to $M$. This new single strand is exactly the complementary of $M'$, that is the strand we want to obtain. This double strand is still incompletely paired with the original $M$. At this point, the molecule is denatured, so the two single strands are separated and the molecule $L$ is added. In this way, the single strand $\overline{M'}$ anneals with $L$. The other single strand—the original molecule—is destroyed using the enzyme *S1 Nucleasi*. The remaining strand is polymerized starting from $L$ and the molecule $M'$ is thus generated. The last step is simply that of denaturing the double stranded molecule, obtaining the single strand $M'$.

Now we can show the way Beaver suggests to realize the state transition: let the reader suppose that the Turing Machine is in the state $q$, the head position is $i$ where the symbol $\sigma$ is pointed. That is, the encoding of the configuration contains the substrand[10]

$$E(\sigma_{--}, i-2)E(\sigma_{-}, i-1)E(q,i)E(\sigma,i)E(\sigma_{+}, i+1)$$

Moreover, the state transition that must be performed is

$$\delta(q,\sigma) = (q', \sigma', left)$$

where $q, q' \in Q$ and $\sigma, \sigma_{--}, \sigma_{-}, \sigma_{+}, \sigma' \in \Sigma$. The substitution that performs this state transition is $LAXBR \longrightarrow LAYBR$, where

- $A = E(\sigma_{--}, i-2)$,
- $X = E(\sigma_{-}, i-1)E(q,i)E(\sigma,i)$,
- $B = E(\sigma_{+}, i+1)$,
- $Y = E(q', i-1)E(\sigma_{-}, i-1)E(\sigma', i)$.

[10]The encoding function $E$ codifies pairs of elements, associating a tape position to a symbol (or a state):

$$E: \ ((Q \cup \Sigma) \times \{1, \ldots, S(n)\}) \to \Sigma_{dna}^{O(\log n)}.$$

while $L$ and $R$ are—respectively—the left and the right side of (the encoding of) the rest of the configuration. In this way a new configuration is encoded by the DNA strand that contains the new substrand

$$E(\sigma_{--}, i-2)E(q', i-1)E(\sigma_{-}, i-1)E(\sigma', i)E(\sigma_{+}, i+1)$$

In order to give the input to the Turing Machine, a initial test-tube $T_0$ must be created. Let us suppose that the start state is $q_0$ and the tape contains the word $x$ as an input, the initial configuration would be $q_0 x 0^{S(|x|)-|x|}$ and its encoding should be the content of $T_0$. Moreover, at the end of the computation the final state must be recognized (we can suppose that there is only one *halt* state) and a result (accept or reject of the input) must be given. We can suppose that the input has been accepted if—and only if—the symbol '1' is in the first position of the tape at the end of the computation (and—viceversa—if the first symbol is '0', then the input is rejected). Thus, the biological operation

$$Detect(Extract(T, E(halt, 1)E(1, 1)))$$

realizes the termination. It extracts all the strands that contain the substrand $E(halt, 1)E(1, 1)$, that are the configurations of the Turing Machine that reached the *halt* state and accepted the input (an analogous extraction with $E(0, 1)$ instead of $E(1, 1)$ would extract final configurations where the input is rejected). If at least one of such configurations exists (that is, the detect operations finds there are DNA strands that have been extracted), the computation terminates and the opportune result is given.

Finally, Beaver suggests to extend his method to simulate nondeterministic Turing Machine computations: the strands (that encode configurations) can be amplified at each step reaching a sufficient number to perform all the possible state transitions. That is, if the current state is $q$ and the pointed symbol is $\sigma$, the strands are amplified and separated in $|\delta(q, \sigma)|$[11] different test-tubes. In this way, in each test-tube an opportune state transition is performed. Beaver asserts—in [Bea95]—that *'such unconstrained parallelism allows to perform any PSPACE computation, far beyond $\mathcal{NP}$'*. Anyway, he recognizes that molecular massive parallelism is not exponential parallelism. This is caused by the practical restrictions that we introduced in section 3.1: the solution of $\mathcal{NP}$-complete problems requires an exponential number of molecules, because each candidate for the solution needs at

[11] $\delta$ is the transition function: $\delta : (Q \times \Sigma) \longrightarrow (Q \times \Sigma \times \{left, right\})$.

$$\begin{array}{ll} 5' & 3' \\ \ldots ACTGCAGT \ldots & \\ \ldots TGACGTCA \ldots & \\ 3' & 5' \end{array}$$

Figure 5: A restriction site of a class II endonucleases enzyme.

least one molecule to be encoded. Anyway, such limitations do not change that Beaver's method allows to simulate nondeterministic Turing Machines computations.

## 5.2 A restriction enzyme implementation of a Turing Machine

In [Rot96], a DNA and restriction enzyme implementation of a Turing Machine is suggested. Restriction enzymes are employed in nature by bacteria to cut DNA double strands[12] at specific substrands called *restriction sites*. Many of them cut DNA in two pieces leaving *sticky ends* on the two new strands. An example is the enzyme *EcoRI* that cuts DNA in the following way:

$$\begin{array}{l} \ldots GAATTC \ldots \\ \ldots CTTAAG \ldots \end{array} \longrightarrow \begin{array}{l} \ldots G \\ \ldots CTTAA \end{array} + \begin{array}{r} AATTC \ldots \\ G \ldots \end{array}$$

The restriction site is the one at the left of the arrow and the product of the restriction are the two strands showed on the right. They have sticky ends in the sense that they remain partially single stranded at the extremity and they can link with a complementary sticky end. The enzyme *EcoRI* belongs to the class *II endonucleases*. One of the properties of this type of enzymes is that the two single strands that compose the restriction site are identical: since each DNA strand has a polarity, and since two complementary single strands are antiparallel, if we read—for example—the restriction site of the enzyme *EcoRI* (see figure 5) from the 5' end to the 3' end (or viceversa), we can see that they are the same strand: $GAATTC$[13]. Consequently, also the two sticky ends generated by the cut are identical. This symmetry permits that a sticky end of a DNA strand cut by the enzyme can anneal to a sticky end of another DNA strand that has been cut by the same enzyme. This

[12]Bacteria use restriction enzymes to damage foreign DNA—like that of viruses—which enters the bacterium.

[13]Biologists call these kind of sequences *palindromic*.

property of the class II endonucleases enzyme is the ground of Rothemund's idea of implementing a Turing Machine.

In few words, any instantaneous description of the Turing Machine is encoded in a DNA strand and the state transition is realized cutting the strand at an opportune substrand (recognized by the enzyme associated with the transition) and inserting a new (opportune) substrand between the two sticky ends to obtain the new instantaneous description.

To be more precise, the elements of the Turing Machine are distinguished in *constants* and *variables*. The transition table is constant, while the instantaneous description (also ID from now on) is variable. Each ID is encoded in a single circular DNA molecule, called *plasmid*. The state transition table—that is the constant information—is encoded permanently by oligonucleotides. These strands are opportunely chosen and each one of them realizes—with few chemical steps—the correspondent transition and it modifies the plasmid into a new one representing the new ID. This is obtained (for a more detailed description of Rothemund's implementation, the reader can consult [Rot96]) in the following way: the plasmid encodes the symbols of the tape concatenating double stranded fragments that encode each element. The current state and the head position are also encoded in the plasmid. They both appear near the encoding of the pointed symbol and they consist in subsequences that contain a restriction site. Each state has its site and—consequently—its enzyme. The site (and then the enzyme) also depends on the symbol that the head is reading; thus, there is a different site for each entry in the transition table. The table is codified by a set of double strands, one for each transition of the Turing Machine. Each one of these oligos contains the encodings of the new symbol, the one of the new state and that of the move (to the left or to the right) that the Turing Machine must realize to perform the state transition associated with that precise oligonucleotide. Finally, the oligo has two sticky ends that allow it to insert exactly in the site cut by the enzyme associated with old state, with the old pointed symbol and—thus—with the performing state transition. In fact, this substitution of that fragment of the plasmid results in a new plasmid that encodes the desired new instantaneous description, including the encoding of the new state and the new head position. This operation is realized—as we said—in few chemical steps that, first, cut the old plasmid and detach the fragment that must be removed and, then, circularize the DNA molecule with DNA ligase after the new fragment is opportunely inserted.

There is a difference with Beaver's proposal because here the encoding of the new elements are already in the molecules that encode the transition table (that Beaver did not encode) and they must not be sinthesized at each transition. In Rothemund's method, the table is encoded once at the beginning of the computation and each entry is associated with commercially available restriction enzymes[14].

Rothemund—as an example—designed the implementation of the BB-3 Turing Machine[15], that is a machine with 3 states and an alphabet of 2 symbols. Since the product of these two number is 6, just 6 different enzymes are needed because this is the number of the entries of the transition table. Rothemund asserts that his implementation of a Turing Machine is possible for machine in which this product is up to 60, since this is the number of the commercially available enzymes of the opportune class that are sufficiently different not to generate errors during the simulation of the Turing Machine's execution.

Finally, Rothemund shows how his method—that uses only commercially available reagents and perfoms operations that are routine for molecular biologists—can be applied to implement a Universal Turing Machine (UTM), since Minsky designed a UTM with 7 states and 4 symbols (and $4 \times 7 = 28 < 60$). The chemical expression of a universal model of computation improves all the demonstrations of the computational capacity of a hypothetic molecular computer.

# 6 Particular applications

In this section, two particular fields for molecular computing are described. They are the implementation of memories and the solution of some cryptographic problems. In fact, these applications would be of particular interest

[14] In reality, Rothemund suggests to use a different class of restriction enzyme. The reason is that the palindromic property of the type II restriction endonucleases enzymes can take to undesired events during this implementation of a Turing Machine. For example, since the two sticky ends generated by the cut are identical, a new annealing between the two cut strands could happen. Besides—for the same reason—having a control of the orientation of the fragment that must be inserted is not possible. For these and other reasons, the restriction enzymes that Rothemund suggests to use are a subclass of the II endonucleases, that is the *IIS restriction endonucleases*. These enzymes cut the strand *near* the restriction site, at an opportune distance. In this way the undesired events cannot take place. These enzymes are called *nonpalindromic*.

[15] The Busy Beaver problem for a Turing Machine with $N$ states (that is BB-$N$) consists in designing a Turing Machine with $N$ states and a two-symbol alphabet $\{Black, White\}$ that writes the greatest possible number of $Black$ symbols before halting.

(if a molecular computer existed!) in order to improve current technologies.

## 6.1 Associative memories

One of the most interesting properties of DNA molecules—from a computational point of view—is that storing data in DNA molecules allows for a very high information density: 1 bit per cubic nanometer ([Adl94]). This property gives molecular computation a great potential. Moreover, there is an interesting variety of available biological operations that can be applied at a time to all the molecules that are in a same test-tube. For these reasons, the idea of using molecules for implementing memories is as old as DNA computing.

In fact, the first author that suggested it, is Adleman himself: in [Adl95] the *Memory Model* is presented. The allowed operations are those of the Restricted model (that is Extract, Union and Detect), plus the *Flip* operation. This one—given a test-tube $T$ and two symbols $a_i$ and $b_i$—switches $a_i$ to $b_i$ in all the aggregates that contain $a_i$ and, viceversa, switches $b_i$ to $a_i$ where $b_i$ appears. Each aggregate in $T$ represents a memory with $n$ locations where each location $l_i$ contains a '1' if $l_i = a_i$ (viceversa, it contains a '0' if $l_i = b_i$). Thus, the Flip operation allows to change the value to a bit. The aim of this proposal was to increase the computational interest of DNA molecules, proving they are able of keeping a memory too.

After Adleman's proposal, another author suggested the realization of a memory model. We will describe the PAM model (suggested by John Reif in [Rei95]) in section 8.2. This high level model is characterized by the *PA-Match* operation which—added to other known biological operations—allows to simulate a CREW PRAM with $M$ memory cells, time bound $D$ and processor bound $P$, using $O(t)$ PA-Match operations and $O(s \log s)$ of the other operations, where $t = D + s$ and $s = O(\log(PM))$. In [Rei95], it is shown how the new PA-Match operation can be implemented in a lab, and how the PRAM is realized.

Finally, the paper [Bau95] is entirely dedicated to the realization—via DNA molecules–of an associative memory. As the title itself suggests (*Building an Associative Memory Vastly Larger Than the Brain*), a vessel containing DNA molecules is employed to implement an associative memory of enormous capabilities. In fact, Eric Baum shows how to realize a content addressable memory which stores binary words. Usual memories need an address to retrieve a stored word. Instead, a content addressable memory needs a partial information about the content of a cell to retrieve the entire content of it. Thus, a DNA based memory can store binary words that can be *read* looking for the molecules that contain a certain subword (that is a substrand). As we know, this operation can be realized—at a time—on all the DNA molecules that are in the vessel. Operations like *write* or *remove* are also feasible, with the opportune biological operations that we can imagine (and that are detailed in [Bau95]). Anyway we store this binary information and anyway we realize the extraction of the desired words[16], the interest of this implementation lies in the high memory density allowed by DNA molecules and in the efficient search of the words: it is not necessary to visit the cells one at a time, but the search is performed simultaneously on all the stored words. Such a memory is able to contain something like $10^{20}$ words in each vessel, that is a capability comparable to that of human brain (estimated as $10^{14}$ synapses, each storing few bits).

## 6.2 Cryptography

A field in which DNA computing appears to be particularly interesting is the cryptography. Immediately after Adleman's paper [Adl94], Donald Beaver in [Bea94] suggested a DNA solution for the problem of factoring large numbers. He suggested a molecular algorithm that solves this problem in a polynomial number of steps. As known, factoring large numbers is a relevant problem in many cryptographic applications.

In [BDL95]—whose title is *Breaking DES using a molecular computer*—and in following papers ([BL95] and [ARRW96]), the problem of attacking a cryptosystem is addressed. The *DES* is the *Data Encryption Standard*, that is the IBM's widely used encryption procedure. It uses a 56 bit key to encrypt 64 bit messages. The encryption procedure is known, and the security is based only on the secret encryption key. In [BDL95], a 'molecular program' that breaks DES is proposed. That is, one (plain-text, cipher-text) pair is given, and the key that maps the plain-text into the cipher-text is found. Conventional (silicon based) computers—using brute force—would take $10^4$ years to solve the same problem (finding the key that maps a particular pair). Special purpose computers could be built to break DES. Such a machine would solve the problem in about a year, but it would cost about $10^6$\$ ([Smi95]). Algorithm different from the brute force one exists, but they do not seem to be a really dangerous attack to DES. Thus, the molecular approach results interesting, since it is estimated—by the authors of [BDL95]—that the problem could be carried out in about 4 months of lab

[16] Various choices are examined in [Bau95].

work.[17] We just give an idea of the *tricks* of the molecular algorithm that attacks DES. A more detailed description of the procedure can be consulted directly in [BDL95].

First of all, a molecular encoding for binary strings is suggested. This encoding allow to perform some logical operations that are used in the algorithm. For example, in few chemical steps, it is possible to extract strands that codify binary strings that have assigned a certain sequence of binary values to a certain sequence of bit position. This is an extension of the Extract operation that we already know.

Let $f : \{0,1\}^n \to \{0,1\}^m$ be a function. A DNA solution that codifies all the pairs $(k, f(k))$ can be easily built. For each $k \in \{0,1\}^n$ this soup contains at least a molecule that represents the binary string $(k, f(k))$. Such a soup can be seen as a table of all the values of the function $f$. For this reason, we denote with $T_f$ the test-tube that contains it. This test-tube is the *trick*, because it allows to invert the function $f$. That is, given a value $C$, it is possible to find $k$ such that $f(k) = C$.[18]

The authors call their algorithm *Eve* (from eavesdropper). Eve's first aim is to find the key $K_0$ that maps a given plain-text $M_0$ in a given cipher-text $C_0$. The DES circuit is such that $k_0$ is unique. Let $f(k) = DES(M_0, k)$ be the DES encryption procedure for the plain-text $M_0$, using the key $k$. The tube $T_f$[19] allows Eve to invert this function and—thus—to find the key. The four months-time required is the time necessary to build the test-tube $T_f$. Once this is done, Eve needs just few time to find the key that maps the pair.

Such a procedure is sufficiently general to be used to break any cryptosystem that uses a key up to 64 bits. The reasons that make DNA computing particularly interesting in cryptography are: first, the great memory capability. Second—as it is underlined in [BL95][20]—with a test-tube like $T_f$, it is pos-

[17]In [Smi95], the feasibility in a lab of (some steps of) the algorithm is analyzed. It is estimated that a correct execution would take one year—and not only four months—of work. Anyway—as the same author of [Smi95] says—this does not change the validity of the idea.

[18]This is achieved in the following way: first, all the strands that contain the pattern that encodes $C$ are extracted from $T_f$. Second, the value of $k$ can be retrieved from the extracted strands.

[19]Since 'only' $2^{56}$ DNA strands are requested, it is possible to manipulate such a test-tube.

[20]In this paper, it is pointed out that—under some hypotheses—the DES attack could take just one day to be carried out.

sible to *batch* the computation, computing $m$ values—$f^{-1}(x_0), \ldots, f^{-1}(x_m)$—in time $O(s)+O(m)$, where $s$ is the time requested to build $T_f$, while a conventional computer would take $O(s \cdot m)$ time. So, if a molecular computer existed, cryptosystems should use keys that are much longer that 70 bits.

# 7 The DNA-Pascal

In order to suggest a high level model of molecular computation, in [RW95] a molecular programming language has been presented: the *DNA-Pascal*. It uses particular data types and (some of) the usual Pascal instructions, adding new operations and tests that correspond to biological operations executed on DNA strands. In fact—instead of using only arithmetical and boolean operations on numerical (and boolean) data—new instruction are defined. Similarly, some tests between data are proposed, to be added to the usual binary relations between numerical and boolean variables. The definition of this language allows the authors of [RW95] and [RW96] to prove results about the computational power of the different models of DNA computing that can be expressed using different sets of operations and tests. For example, in [RW95] they prove that Lipton's model solves—in polynomial time—exactly the problems belonging to the complexity class $\mathcal{P}^{\mathcal{NP}}$. In [RW96], they characterize the computational power of set operations defining some crucial properties achieved if some operations (and/or tests) are used.

The language is presented in subsection 7.1. In subsection 7.2, we will show two examples of DNA-Pascal programs. The first one is Lipton's algorithm that we described in section 3.2. The other one is another implementation of a Turing Machine.

## 7.1 The language

DNA-Pascal programs operate only on arrays and variables of non-negative integers. A new type of variable is defined consisting in finite sets of words. More in detail, natural numbers are encoded using words of a finite alphabet (in the case of DNA computing the alphabet is $\Sigma_{DNA} = \{A, C, T, G\}$)[21]. For this reason, we refer both to *natural variables* and to *word variables*. The data type *multiset* is a finite set of (eventually repeated) finite words on the alphabet. It is easy to understand that this data encodes the content of a

[21]For simplicity—both in [RW95] and in [RW96]—the alphabet $\Sigma = \{0, 1\}$ is used, because the results do not depend on the choice of $\Sigma$.

test tube, that is a set of strands of nucleotides. The computation terminates either with the DNA-Pascal instruction 'reject' or 'accept'. As we have introduced, a reduced set of Pascal instruction is used and new operations and tests are defined. The operations are the following:

**DNA-operations**[22]

**(UN) Union** $T := T_1 \cup T_2$

**(AS) Assignment** $T := T_1$

**(EX) Extraction** $T := EX(T_1, m, a)$
where $EX(T_1, m, a) = T_1 \cap (\{0,1\}^{m-1}a\{0,1\}^*)$, $a \in \{0,1\}$ and $m \geq 1$.[23]

**(IN) Initialization** $T := IN(m)$
where $IN(m) = \{0,1\}^m$ and $m \geq 0$.

**(EW) Empty Word** $T := \{\varepsilon\}$

**(RA) Right Append** $T := T_1 \cdot a$
where $T_1 \cdot a = \{za | z \in T_1\}$.

**(LA) Left Append** $T := a \cdot T_1$
where $a \cdot T_1 = \{az | z \in T_1\}$.

**(CO) Concatenation** $T := T_1 \cdot T_2$
where $T_1 \cdot T_2 = \{xy | \ x \in T_1, \ y \in T_2\}$.

**(RC) Right Cut** $T := T_1/$
where $T_1/ = \{z/ | z \in T_1\}$ and $za/ = z \ \forall a \in \{0,1\}$ and $\varepsilon/ = \varepsilon$.

**(LC) Left cut** $T := /T_1$
where $/T_1 = \{/z | z \in T_1\}$ and $/za = z \forall a \in \{0,1\}$ and $/\varepsilon = \varepsilon$.

**(IS) Intersection** $T := T_1 \cap T_2$.

[22] The operation set we describe here is a little different from the one suggested by the authors. In particular, we do not present some operations that appear in [RW96]—whose computational power appear interesting—for two reasons: first, it is not clear their feasibility in a lab. Second, because the interesting results about the computational power of these operations and tests will not be detailed in this report. The interested reader can consult [RW95] and [RW96].

[23] The authors of [RW95] and [RW96] call this operation *Bit Extraction*, to distinguish it from the more general *Word Extraction* (that they also define).

In order to define new conditions that can be verified on DNA-Pascal variables, the following tests are proposed:

**DNA-tests**

**(SU) Subset** $T_1 \subseteq T_2$.

**(EM) Detect** $T = \emptyset$.

**(ME) Membership** $x \in T$.

All of these operations and tests take one step to be executed, except the initialization ($IN$) and the concatenation ($CO$) that take linear time in the size of the longest word that appears in the test-tube at the end of the operation. If we assume that DNA computations begin with an empty test-tube, a polynomial number of steps can produce only words of polynomial length.

We will not give a formal syntactic description of DNA-Pascal. The *molecular programmer* could choose any reduced—sufficiently expressive—set of Pascal instructions. Moreover, a subset of DNA-operations and a subset of DNA-tests must be chosen to define the model of computation. The authors of DNA-Pascal suggested the notation

$$DNA - (\mathcal{O}, \mathcal{T}) - \mathcal{P}$$

to denote the class of problems that can be solved in polynomial time by DNA-Pascal programs that use the set $\mathcal{O}$ of DNA-operations and the set $\mathcal{T}$ of DNA-tests. For example, problems that can be solved using a polynomial number of steps by Lipton's model of DNA computing should be the class $DNA - (\{UN, EX, IN\}, \{EM\}) - \mathcal{P}$, since Lipton uses exactly these three DNA-operations (union, extraction and initialization) and the emptiness test (it is the biological operation *detect* of section 3.2). In section 7.2 we will show the DNA-Pascal description of Lipton's algorithm for the solution of the SAT problem. Finally, note that the annealing (and melting) operation are not mentioned in the DNA-operations, but they can be considered at disposal in any DNA computation.

## 7.2 Two DNA-Pascal programs

We described Lipton's solution of the SAT problem in section 3.2. As an example of DNA-Pascal program, we now propose a description of Lipton's algorithm (it appears both in [RW95] and in [RW96]. As known, also the $3 - SAT$ problem is an $\mathcal{NP}$-complete one. The solution we present refers exactly to this problem. It is formulated in the following way:

*3-SAT Problem*: Search—if it exists—an assignment of boolean values to $n$ variables $(x_0, x_1, \ldots, x_{n-1})$, such that the formula

$$F(x_0, x_1, \ldots, x_{n-1}) = \bigwedge_{i=1}^{k} (x_{\alpha[i]}^{a[i]} \vee x_{\beta[i]}^{b[i]} \vee x_{\gamma[i]}^{c[i]})$$

is valid. Where:

- $x_{\alpha[i]}$ indicates that we refer to the variable whose index is in the vector $\alpha \in \{0, \ldots, n-1\}^k$. It indicates—for each of the $k$ clauses—which one of the $n$ variables refers the first literal of the clause. In the same way, the vectors $\beta$ and $\gamma$ (they also are in $\{0, \ldots, n-1\}^k$) indicate—respectively—the identity of the second and third variable of the clause.
- $x_{\alpha[i]}^{a[i]}$ means that the value we want to be assumed by the variable $x_{\alpha[i]}$ is the $i^{th}$ element of a vector $a \in \{0,1\}^k$. To be more precise, if $a[i] = 0$ then the first variable of the $i^{th}$ clause (that is $x_{\alpha[i]}$), is denied, otherwise—if $a[i] = 1$—it is not. In a similar way, the two vectors $b, c \in \{0,1\}^k$ indicate the value to be assumed by—respectively—the variables $x_{\beta[i]}$ and $x_{\gamma[i]}$, for each $i = 1, \ldots, k$.

The $3-SAT$ problem is solved by the following algorithm, written in DNA-Pascal:

**begin**
$T_0 := IN(n)$;
**for** $i := 1$ **to** $k$ **do**
    **begin**
    $T_1 := EX(T_0, \alpha[i], a[i])$;
    $T_2 := EX(T_0, \beta[i], b[i])$;
    $T_3 := EX(T_0, \gamma[i], c[i])$;
    $T_0 := T_1 \cup T_2$;
    $T_0 := T_0 \cup T_3$
    **end**;
**if** $T_0 = \emptyset$ **then** reject **else** accept
**end**

Brief explanation: the initialization operation results in the test-tube $T_0$ that contains all the (encodings of the) sequences of $\{0,1\}^n$. They represent all the $2^n$ possible assignments of values to the $n$ boolean variables

$x_0, x_1, \ldots, x_{n-1}$. The *for* instruction operates on each one of the $k$ clauses, performing the following operations:

1. The first $EX$ extracts from $T_0$ the sequences that assigned the right value ($a[i]$) to the $\alpha[i]^{th}$ variable. We remember that Lipton's algorithm is so that—at each one of the $k$ steps of the *for*—the test-tube $T_0$ contains the subset of $\{0,1\}^n$ that corresponds to the assignments that satisfy all the first $i-1$ clauses. Consequently—after the execution of this operation—in $T_1$ appear all the (encodings of the) assignments that satisfy the first $i-1$ clauses and the first literal of the $i^{th}$ clause. In the same way, the other two $EX$ put in $T_2$ and $T_3$—respectively—the assignments that satisfy (the first $i-1$ clauses and) the second and the third literal of the clause $C_i$.

2. The two unions realize the operation $T_0 := T_1 \cup T_2 \cup T_3$. That is $T_0$ contains all the assignments that satisfy the first $i$ clauses.

At the end of the *for*, $T_0$ contains the assignments that satisfy the entire formula, if they exist. Thus, the final detect operation simply checks if $T_0$ is empty or not. The result is given consequently ('yes' if $T_0$ contains a solution and 'no' otherwise).

It is easy to see that this algorithm's time complexity is linear in $n+k$, because the $IN$ operation costs $O(n)$ steps and the remainder of the algorithm executes $5k+1$ operations which take constant time.

Using the same sets $\mathcal{O} = \{UN, EX, IN\}$ and $\mathcal{T} = \{EM\}$ of operations and tests, it is possible to simulate the behavior of a Turing Machine. The algorithm we are going to show appears in [RW95].

Let $M = (S, \Sigma, \delta, s_0, s_1)$ be a Turing Machine whose executing time is polynomially bounded in the size $n$ of the input. As in the usual notation, $S$ is the set of states, $\Sigma$ is the input alphabet (we suppose—without loss of generality—it is $\Sigma = \{0,1\}$), $s_0$ is the start state, $s_1$ is the final state and $\delta$ is the state transition function, that is:

$$\delta \,:\, S \times \Sigma \longrightarrow S \times \Sigma \times \{-1, 0, 1\}$$

where the last component describes the move of the head. We suppose that—when the computation begins—the first $n$ positions of the tape contain the input. Finally, we suppose again that $M$ accepts the input if—when it is in the final state—the symbol '1' is on the first tape position. Otherwise, a '0' symbol means that the input is rejected.

Before going on, we show the way in [RW95] is suggested to represent the tape of the Turing Machine. Let $i, n$ be natural numbers and let $x \in \{0,1\}^*$. We indicate with $bin(i)$ the binary representation of $i$ and with $bin_n(i)$ the binary representation whose length is exactly $n$ (if necessary, it can be achieved adding zeros). Let $x_i$ be the $i^{th}$ bit of $x$ (obviously—given $i, n$ and $j$—the value $bin_n(i)_j$ can be easily calculated). In order to simulate $M$ by a DNA-Pascal program, the information of the tape is stored in a set variable $B$. Since the words of a set variable in DNA-Pascal are defined on the alphabet $\{0,1\}$, we can use the standard encoding $c : \Sigma \rightarrow \{0,1\}^m$ to codify each element of $\Sigma$ into a binary code, where $m = \log|\Sigma|$. The content $b_i$ of the $i^{th}$ tape position is represented by the presence of the word $bin_l(i)c(b_i)$ in $B$. This happens for the tape positions from 1 to—say—$p(n)$, because $M$ uses not more that $p(n)$ tape positions (let $l = |bin(p(n))|$). Thus, each word in $B$ has length $l + m$. The following DNA-Pascal program initializes $B$, filling it with words that encode the input $b_1 \dots b_n$:

**begin**
    $T := IN(l+m)$;
    **for** $i := 1$ **to** $p(n)$ **do**
        **begin**
            $T' := T$;
            **for** $j := 1$ **to** $l$ **do**
                $T' := EX(T', j, bin_l(i)_j)$;
            **for** $j := 1$ **to** $m$ **do**
                $T' := EX(T', l+j, c(b_i)_j)$;
            $B := B \cup T'$
        **end**
**end**

The $IN$ operation puts in a test-tube $T$ all the strands of length $l + m$. The *for* instruction selects—among them—those that begin with $bin_l(i)$ for each $i = 1 \dots p(n)$ (the first of the two internal *for*) and that end with the correct $c(b_i)$ (the second internal *for*). The strands that survive this selection are moved in $B$, so that $B$ contains exactly the $p(n)$ words of the desired form.

In order to manipulate $M$'s head, it is useful to define a counter $h$ that stores the current head position. Obviously, it is initialized with the value '1'. Moreover, we need a variable $s$—initialized with $s_0$—that stores the current state. Now we can see how a state transition is realized: let's suppose—for example—that $M$ is in the state $s$, its head points to the symbol $d$ and the transition

$$(z, d) \longrightarrow (z', d', k).$$

must be performed. The program of figure 6 realizes it. The first *for* realizes the search of the right transition to be executed. When it is found, the algorithm verifies that $M$ is in the right state. After that—using the temporary variable $T$—the following operations are executed:

- A first *for* separates the words that begin with $bin_l(h)$. They are moved in $T$, while all the others end in $R$.

- A second *for*—among the words that begin with $bin_l(h)$—keeps in $T$ only those that contain the encoding of $d$. Again, the others end in $R$.

- If $T$ is not empty (that is, if there are words that verify the condition to perform the transition), the computations continues giving $s$ the new state value (that is $s'$) and building the new word $bin_l(h)c(d')$ that must be added into $B$. Finally, the variable $h$ is opportunely modified.

When the computation stops (this is detected because $s = s_1$, the final state), the algorithm could load the value stored in the first tape position, to read the result. That is, if the word that begins with $bin_l(1)$ ends with $c(1)$, then the input is accepted. Otherwise (that word ends with $c(0)$), the input is rejected.

# 8 Other models

## 8.1 The Sticker Model

In most of the algorithms we have seen until now, binary data are encoded in the following way: the presence of a certain strand in the test-tube means that a particular bit is set to a particular value. Thus, it was necessary to associate two DNA strands to each boolean variable. The central idea of the *Sticker Model*—suggested in [RWB+96]—is to store binary information in another—more efficient—way, that avoids using enzymes (that seem to be too expensive and—above all—they have a short time of life), and allows to have a random access memory.

Two different kinds of single DNA strands are employed: *memory strands* and *stickers*. Memory strands are—more or less—long strands that represent

```
begin
for (z, d) → (z', d', k) ∈ δ do
    begin
    if s = z then
            begin
            T := B;
            for i := 1 to l do
                begin
                R := R ∪ EX(T, i, 1 − bin_l(h)_i);
                T := EX(T, i, bin_l(h)_i)
                end
            for i := 1 to m do
                begin
                R := R ∪ EX(T, l + i, 1 − c(d)_i);
                T := EX(T, l + i, c(d)_i)
                end
            B := R;
            if T ≠ ∅ then
               begin
               s := z';
               T := IN(l + m);
               for i := 1 to l do T := EX(T, i, bin_l(h)_i);
               for i := 1 to m do T := EX(T, l + i, c(d')_i);
               B := B ∪ T;
               h := h + k
               end
            end
    end
end
```

Figure 6: DNA-Pascal algorithm that simulates a Turing Machine.

a binary string. They are $n$ bases long sequences, that are the concatenation of $k$ regions. Each one of these regions is $m$ bases long and refers to a bit position. The stickers are $m$ bases long DNA strands that are the Watson-Crick complementary of the regions of the memory strands. If a memory strand has a region (associated to the bit $b$) that is single stranded, it means that $b = 0$. Instead, if the sticker is bonded with the region, it means that $b = 1$. A memory strand that is partially double stranded (because its regions are linked with some stickers) is said to be a *complex*. A complex represents a string of $k$ bits. The information density—comparable with those of the other models—is of $(1/m)$ bits/base. The advantages of this way of storing binary information lie in the efficiency in manipulating data. The operations that are realizable on the complexes (and that compose the sticker model) are: *union* of test-tubes containing complexes, *extraction* of complexes that have a certain bit set to '1' (or '0'), *set* of a bit (all the complexes in a test-tube are annealed with the sticker of that bit, setting its value to '1'), *reset* of a bit (with the obvious meaning), and—finally—the *clear* operation that removes a bit in every complex in the tube. A computation is a sequence of these operations that begins with an initial test-tube (*mother tube* in the paper) and ends with a final one that is the output. Obviously, in [RWB+96], it is described the biological implementation of each one of the proposed operations. The paper is also complete of an example of a *sticker based* solution of a $\mathcal{NP}$-complete problem, that shows the model's flexibility for implementing general algorithm. Moreover, an interesting treatment of the problem of errors is addressed (opportune design of sequences that achieves sticker specificity and avoids complexes to anneal with themselves), also suggesting a reducing error *Refinery Model*. Finally, a possible realization of a *sticker machine* is proposed. It is shown how—with the actual biotechnology—it is possible to construct a sticker-based molecular computer. The idea is to design different kinds of tubes—*data tubes* and *operator tubes*—and opportune combinations of them that implement the operations of the model, using membranes, connectors that drive the computation, including heating and cooling the molecules, if necessary.

In section 6.2, we described the proposal of breaking DES using DNA computing. The sticker model has been conceived after that proposal and—in [ARRW96]—some authors have suggested to use the sticker model to improve the DES attack. The interested reader can find the details in the paper. We just remark here the result of the analysis: a sticker machine (of a dimension that allows to keep it on a desktop) could implement a plaintext-ciphertext attack using only a gram of DNA molecules, even if several errors occur.

## 8.2 The PAM Model

In [Rei95], Reif suggested his Parallel Associative Memory Model, one of the first proposal in order to define operations on DNA strands that can be implemented with the aim of calculating with DNA. This proposal was also the first one that suggested how to compute with DNA implementing something different from a brute force approach to a problem. In fact, the PAM Model is a formalism that expresses the *molecular parallelism.* The operation that characterizes the PAM Model is the *PA-Match* operation. Let $E(\alpha)$ be the encoding of $\alpha$ and let $E(\alpha, \beta)$ be the one of the pair $(\alpha, \beta)$. The PA-Match operation is so defined:

$$E(\alpha, \beta) \bowtie E(\beta', \gamma) = \begin{cases} E(\alpha, \gamma) & \text{if } \beta = \beta' \\ \perp & \text{otherwise} \end{cases}$$

The operation is applied to entire test-tubes with the following meaning:

$$T \bowtie T' = \{x \bowtie x' | x \in T, x' \in T'\}$$

where $x$ and $x'$ are DNA molecules. In addition to this special operation, the PAM Model uses the well known operations of union, extraction and detect. To evaluate an algorithm, Reif suggests to consider the number of required steps (as a time measure), the maximum number of strands that appear at a time in the test-tube, and the length of the longest strand in the test-tube.

Using $O(t)$ PA-Match operations and $O(s \log s)$ operation (that are not the PA-Match), Reif shows how to:

- Simulate the behavior of a nondeterministic Turing Machine with space bound $s$ and time bound $2^{O(s)}$.
- Simulate a CREW PRAM with $M$ memory cells, $P$ processors and time bound $D$. Where $s = O(\log(PM))$ and $t = D + s$ (see section 6.1).
- Find an assignment of values to $n$ boolean variables that satisfies a circuit constructable in $s$ space with unbounded fan-out and depth $D$. Where—again—$t = D + s$.
- Do List and Tree Contraction for $L$ sized inputs constructable in $s$ space. Here $t = \log L$, each of the used tubes contains $O(B + L/\log L)$ aggregates and it is assumed that there is a precomputed tube of $B$ aggregates.

The details of these implementations can be found in the paper ([Rei95]). We here just remark another aspect that characterizes Reif's work. The PAM Model is a high level one. Thus, it allows to express algorithm that should be implementable at a lower level. In the case of DNA computing, the lowest level involves molecules. Reif formalized this aspect defining another model, the *Recombinant DNA Model* (RDNA). It is shown how the operations of the PAM Model are implemented by those of the RDNA Model, that is a lower level one. It is an abstraction of the recombinant DNA techniques—available in a molecular biology lab—that are at the bottom of any DNA computation. Finally, it is showed how—with a low slowdown—the probability of errors can be reduced.

## 8.3 Splicing Systems

In the previous sections, we often have referred to DNA strands as strings over the alphabet $\Sigma_{DNA} = \{A, C, G, T\}$. Splicing Systems have been conceived as a generative formalism with the aim of analyzing the generative capacity of the recombinant behavior of DNA molecules, in terms of formal languages. The splicing systems were introduced in [Hea87] ten years ago, that is when nobody spoke about DNA computing. In fact, only in 1995—thus, after Adleman's paper—Splicing Systems have been suggested to represent DNA computations and their computational properties, by various authors. The content of their papers is the object of this section ([PS96], [Kar95], [FKP95], [CVFKP96], [CVKP96], [FMZ97]).

The central operation of the model is the *Splicing operation*: given an alphabet $\Sigma$ and two strings $x, y \in \Sigma^*$, it is defined the splicing of $x$ and $y$, as indicated by the rule $r$. Formally, a splicing rule $r$ defined on the alphabet $\Sigma$ is a word of the form $\alpha_1\#\beta_1\$\alpha_2\#\beta_2$, where $\alpha_1, \beta_1, \alpha_2, \beta_2 \in \Sigma^*$, while $\#$ and $\$$ are special symbols that are not in $\Sigma$. If we have $x = x_1\alpha_1\beta_1x_1'$, $y = y_2\alpha_2\beta_2y_2'$ and $r = \alpha_1\#\beta_1\$\alpha_2\#\beta_2$, we write

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

to indicate that the strings $z$ and $w$ are obtained from $x$ and $y$ applying the splicing rule $r$, if and only if $z = x_1\alpha_1\beta_2y_2'$ and $w = y_2\alpha_2\beta_1x_1'$, where $x_1, x_1', y_2, y_2' \in \Sigma^*$. In few words, the operation concatenates an opportune prefix of $x$ with an opportune suffix of $y$ and, viceversa, the prefix of $y$ with the suffix of $x$.

This operation is suggested as the fundamental tool for the definition of a generative mechanism, that is a *Splicing System*. Informally, given a set of strings (axioms) and a set of splicing rules, the language that is generated consists in strings obtained by iterative applications of the rules on the strings, beginning with the set of the axioms. Before giving a formal definition, we introduce few concepts: let $\mathbf{N}$ be the set of the natural numbers. A *multiset* on $\Sigma^*$ is a function

$$M : \Sigma^* \rightarrow \mathbf{N} \cup \{\infty\}$$

where, $\forall w \in \Sigma^*$, $M(w)$ represents the number of the occurrences of $w$ in the considered system. $M(w) = \infty$ means that there is not a limitation to the number of copies of $w$. Finally, let $supp(M) = \{w \in \Sigma^* | M(w) \neq 0\}$ be the *support* of $M$.

A *Splicing System* is a tuple $\gamma = (\Sigma, T, A, R)$, where $\Sigma$ is an alphabet, $T \subseteq \Sigma$ is the terminal alphabet, $A$ is the set of the axioms and it is a multiset on $\Sigma^*$, and $R \subseteq \Sigma^* \# \Sigma^* \$ \Sigma^* \# \Sigma^*$ is the set of the splicing rules.[24]

A splicing system $\gamma$ defines a binary relation $\Longrightarrow_\gamma$ on the family of multisets on $\Sigma^*$, in the following way: given two multisets $M$ and $M'$, we have $M \Longrightarrow_\gamma M'$ if and only if:
$\exists\ x, y \in supp(M)$ and $\exists\ z, w \in supp(M')$ such that:

1. $M(x) \geq 1$, $M(y) \geq 1$ if $x \neq y$ ($M(x) \geq 2$ if $x = y$);
2. $(x, y) \longrightarrow_r (z, w)$ as defined by the splicing rule $r \in R$;
3. $M'(x) = M(x) - 1$, $M'(y) = M(y) - 1$ if $x \neq y$ ($M'(x) = M(x) - 2$ if $x = y$);
4. $M'(z) = M(z) + 1$, $M'(w) = M(w) + 1$ if $z \neq w$ ($M'(z) = M(z) + 2$ if $z = w$).

It is easy to imagine that a biological implementation of the splicing operation is possible. For example, it could be possible to associate a restriction enzyme to a splicing rule. In this way, the operation would be performed by the enzyme itself, if it were put in the test-tube that contains the DNA strands associated with the (multiset of the) strings of the splicing system.

The *Language generated by a Splicing Systems* is

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

[24]In literature, splicing systems are also known as *H System*. In particular those we have defined are indicated as *mH Systems*, remarking that the set of the axioms is a multiset.

where $A$ is the set of the axioms and $\Longrightarrow_{\gamma}^{*}$ is the reflexive and transitive closure of the relation $\Longrightarrow_{\gamma}$.[25]

Let $F_1$ and $F_2$ be two families of languages. We denote with

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

the family of the languages that can be generated by extended splicing systems whose axiom set is a language of $F_1$ type, and the one of the rules is of $F_2$ type. In this case, the splicing system is of $(F_1, F_2)$ *type*. In the same way, we use the notation $EH(mF_1, F_2)$ to refer to the family of the languages generated by an extended H system whose axioms set is a multiset.

Significant results about the computational power of splicing systems have been obtained. We cite some of them. Let $FIN, REG$ and $RE$ indicate—respectively—finite, regular and recursive enumerable languages:

- $EH(FIN, FIN) = EH(F, FIN) = REG$
  $\forall F$ such that $FIN \subseteq F \subseteq RE$. This first results appear discouraging, because in a practical (biological) implementation only finite sets of axioms and rules can be employed, and they do not seem to be sufficient to generate recursive enumerable languages. The following results go in the direction of solving this problem.

- $EH(mF_1, F_2) = RE$
  $\forall F_1, F_2$ such that $FIN \subseteq F_1 \subseteq RE$ and $FIN \subseteq F_2 \subseteq RE$. Thus, a finite set of axioms and a finite set of rules are sufficient to generate recursive enumerable languages if the axioms set is a multiset. That is—if we keep track of the multiplicities of strings—the computational power of the splicing systems equals the computational power of the Turing Machines.

- There is another way to reach such a computational power. In fact, recursive enumerable languages can be generated by splicing systems with finite sets of rules and axioms, using another kind of control mechanism: checking the presence of certain substrings in the strings to be spliced. In order to formalize this concept, a new type of splicing rule is defined. A rule of the form $\alpha_1\#\beta_1\$\alpha_2\#\beta_2C_1C_2$ is called a *rule with permitting contexts*. An extended H system whose rules set is of this

[25] In literature, if only terminal strings (that is—as in our case—strings that are in $T^*$) are accepted in the language, the splicing systems (or H systems) are called *extended*. These systems are also denoted with *EH systems*.

type is a *H system with permitting contexts*. The notation $EH(F_1, cF_2)$ indicates the family of the languages that can be generated by this kind of splicing systems.[26]

$$EH(F_1, cF_2) = RE$$

$\forall F_1, F_2$ such that $FIN \subseteq F_1 \subseteq RE$ and $FIN \subseteq F_2 \subseteq RE$.

The proofs of the previous results can be found in literature. They prove that splicing systems are computationally complete. That is, every effective procedure can be carried out by a splicing system. A more formal (in the sense that a formal notion of effective procedure is given) demonstration of this assertion is given in [Kar95], where it is shown that the actions of any computable function can be implemented by a splicing system. These results would not be interesting if a molecular computer based on splicing could not exists. In fact, we know that everything is Turing-computable can be computed by this model of DNA computation too, but this does not say anything about the existence of a *programmable DNA computer*. To this end, we need a *universal splicing system*: a system that is able to simulate the behavior of a given splicing system $\gamma$. Such a system exists and it is described both in [Kar95] and [FKP95]. It is denoted with $\gamma_U$ and it receives a description of the system $\gamma$ (the one that must be simulated) in its set of axioms. This result allows to think about universal and programmable DNA computers based on the splicing operation.

In order to define mechanism—based on the splicing operation—able to represent DNA computations, a new system has been defined ([CVKP96] and [CVFKP96]): the *Test Tube Distribuited System*. The components are test-tubes that work as splicing systems and communicate among themselves. Again, the idea is taken from the formal language theory, because the scheme is similar to that of parallel communicating grammar systems. Formally, a *Test-Tube system of degree n* ($n \geq 1$) is

$$\Gamma = (V, (A_1, R_1, V_1), \ldots, (A_n, R_n, V_n))$$

where $V$ is the alphabet and each $(A_i, R_i, V_i)$ is a component (a test-tube), with its set of axioms $A_i$, its set of rules $R_i$, and its *selector* $V_i \subseteq V$. A *configuration* of a Test-tube system is an $n$-tuple $(L_1, \ldots, L_n)$ where each $L_i \subseteq V^*$ is the content of the $i^{th}$ test-tube. A *computation* is a sequence of configurations, where the splicing phases generate the new strings in each

[26] In a similar way, rules (and H systems) with *forbidding* contexts have been defined. The results that follow are valid for this class of H systems too.

component, and the communication steps redistribuite them to the test-tubes according to the selectors $V_1, \ldots, V_n$. More formally, a computation of length $k \geq 0$ is

$$C = \{(L_1^{(t)}, \ldots, L_n^{(t)}) \mid 0 \leq t \leq k\}$$

such that

1. $(L_1^{(0)}, \ldots, L_n^{(0)}) = (A_1, \ldots, A_n)$,
2. $(L_1^{(t)}, \ldots, L_n^{(t)}) \Longrightarrow_\Gamma (L_1^{(t+1)}, \ldots, L_n^{(t+1)})$, for each $t$ such that $0 \leq t \leq k-1$.

where we assume that the relation $\Longrightarrow_\Gamma$ defines the conditions and the operations that allows to perform—with respect to $\Gamma$—a step of the distribuited computation (see [CVKP96] for details). One of the $n$ components is designed as the *master* and the result of the system computations is defined as the union of its contents in all possible computation. Assuming that the first test-tube is the master, the *language generated by* $\Gamma$ is

$$L(\Gamma) = \{w \in V^* \mid w \in L_1^{(t)} \text{ for some } (A_1, \ldots, A_n) \Longrightarrow_\Gamma^* (L_1^{(t)}, \ldots, L_n^{(t)}),\ t \geq 0\},$$

where $\Longrightarrow_\Gamma^*$ is the reflexive and transitive closure of the relation $\Longrightarrow_\Gamma$.

Using a notation that is analogous to the one used for H systems, we indicate the type of a test-tube distribuited system. Let $F_1$ and $F_2$ be two families of languages. We denote with $TT_n(F_1, F_2)$ the family of languages $L(\Gamma)$, for $\Gamma = (V, (A_1, R_1, V_1), \ldots, (A_m, R_m, V_m))$, $m \leq n$, $A_i \in F_1$, $R_i \in F_2$, for each $i$ such that $1 \leq i \leq m$. In this case we say that $\Gamma$ is of $(F_1, F_2)$ *type*. If $n$ is not specified, it is replaced by $*$, that is

$$TT^*(F_1, F_2) = \bigcup_{n \geq 1} TT_n(F_1, F_2).$$

The final result is that also these systems are computationally complete: test-tube distribuited systems whose tube have a finite content at the beginning and that use finite sets of splicing rules can generate recursive enumerable languages:

$$TT^*(FIN, FIN) = TT^*(F_1, F_2) = RE$$

$\forall F_1, F_2$ such that $FIN \subseteq F_1 \subseteq RE$ and $FIN \subseteq F_2 \subseteq RE$. Finally, a *universal TT system* is conceived, similarly to the universal splicing system. A first result ([CVKP96]) has been that—for any given alphabet $T$—there are universal TT systems of degree $card(T) + 8$ and of $(FIN, FIN)$ type. This result has recently been improved: any recursively enumerable language can be generated using a test-tube distribuited system with nine test-tubes. Here, the number of required test-tube does not depend on the cardinality of the used alphabet ([FMZ97]), that is:

$$TT^9(FIN, FIN) = TT^*(FIN, FIN) = TT^*(F_1, F_2) = RE$$

for all families $F_1, F_2$ such that $REG \subseteq F_1 \subseteq RE$ and $REG \subseteq F_2 \subseteq RE$.

# Conclusions

All the results reported in this report are theoretical, except for Adleman's experiment that has been realized in laboratory. The eventuality of a practical demonstration of DNA computing (i.e., the construction of a real programmable and general molecular computer) is left to the actual feasibility of the models that have been suggested. An answer is yet to know.

# Acknowledgements

I would like to thank Linda Pagli, Claudio Ferretti, Paolo Ferragina, Margherita Maffei, Mario Costa and Francesca Serragoni for conversations. A particular gratitude to Prof. Fabrizio Luccio, my guru. Finally, thanks to all the authors that appear in the references.

# References

[Adl94] Leonard Adleman. Molecular Computation of Solutions to Combinatorial Problems. *Science*, 266:1021–1024, November 1994.

[Adl95] Leonard Adleman. On Constructing a Molecular Computer. Technical Report TR 79-387, Computer Science Department, University of Southern California, USA, January 1995.

[Ame95] J. Amenyo. Biocomputers: architecture, engineering and technology. ftp://ftp.ans.net/pub/misc/biocomputer.txt, 1995.

[Ame96a] J. Amenyo. Personal impressions about the 2nd annual workshop on DNA computing. ftp://ftp.ans.net/pub/misc/jta/DNAComp2rept.txt, June 1996.

[Ame96b] J. Amenyo. Mesoscopic Computer Engineering: automating DNA based computing via traditional practices of parallel computer architecture design. In *Proceedings of the Second Annual Meeting on DNA based computer, Princeton University, USA*, June 1996.

[ARRW96] Leonard Adleman, Paul Rothemund, Sam Roweis, and Erik Winfree. On Applying Molecular Computation To The Data Encryption Standard. In *Proceedings of the Second Annual Meeting on DNA based computer, Princeton University, USA*, June 1996.

[Bas95] Thomas Bass. Gene Genie. *Wired*, 1995.

[Bau95] Eric Baum. Building an Associative Memory Vastly Larger Than the Brain. *Science*, 268:583–585, April 1995.

[Bau96] Eric Baum. DNA Sequences Useful for Computation. Technical report, NEC Research Institute, Princeton, USA, June 1996.

[BDL95] Dan Boneh, Cristopher Dunworth, and Richard Lipton. Breaking DES Using a Molecular Momputer. Technical Report CS-TR-489-95, Department of Computer Science, Princeton University, USA, 1995.

[BDLS95] Dan Boneh, Cristopher Dunworth, Richard Lipton, and Jiri Sgall. On The Computational Power of DNA. Technical Report TR-499-95, Princeton University, USA, October 1995.

[Bea94] Donald Beaver. Factoring: the DNA Solution. In Lecture Notes in Computer Science Springer-Verlag, editor, *Advances in Cryptology - Asiacrypt 1994*, volume 917, pages 419–423, 1994.

[Bea95] Donald Beaver. Molecular Computing. Technical Report TR 95-001, Penn State University, USA, January 1995.

[Bea96] Donald Beaver, 1996. Submitted to 28th annual ACM Symposium on Theory of Computing 1996.

[BL95] Dan Boneh and Richard Lipton. Batching DNA Computations. Technical report, Princeton University, USA, 1995.

[BL96] Dan Boneh and Richard Lipton. Making DNA Computers Error Eesistant. In *Proceedings of the Second Annual Meeting on DNA based computer, Princeton University, USA*, June 1996.

[Bun95] Barry Bunow, 1995. Letter to 'Science', April 1995.

[CB88] Helena Curtis and N.Sue Barnes. *Introduzione alla biologia.* Zanichelli, 1988.

[Cri70] Francis Crick. Central Dogma of Molecular Biology. *Nature*, 227:561–563, August 1970.

[CVFKP96] Erzsebet Csuhaj-Varju, Rudolf Freund, Lila Kari, and Gheorghe Paun. DNA Computing Based on Splicing: Universality Results. In Lawrence Hunter and Teri Klein, editors, *Proceedings of the 1996 Pacific Symposium.* World Scientific Publishing Co., Singapore, 1996.

[CVKP96] Erzsebet Csuhaj-Varju, Lila Kari, and Gheorghe Paun. Test Tube Distribuited Systems Based on Splicing. *Computers and AI*, 15, 1996.

[FKP95] Rudolf Freund, Lila Kari, and Gheorghe Paun. DNA Computing Based on Splicing: The Existence of Universal Computers. Technical Report 185-2/FR-2/95, Institute for Computer Languages, Technical University Wien, Austria, 1995.

[FMZ97] Claudio Ferretti, Giancarlo Mauri, and Zandron. Nine Test Tubes Generate any RE Language, 1997. http://mercurio.sm.dsi.unimi.it/ ferretti/papers/testo9.ps.

[GFB+96] J. Gray, T. Frutos, A. Berman, A. Condon, M. Lagally, L. Smith, and R. Corn. Reducing Errors in DNA Computing by Appropriate Word Design. Draft, 1996.

[Gif94] David Gifford. On the Path to Computation with DNA. *Science*, 266:993–994, November 1994.

[GJ78] Michael Garey and David Johnson. *Computers and intractability.* Freeman and C., San Francisco, CA, 1978.

[Har95a] Juris Hartmanis. On the Computing Paradigm and Computational Complexity. In Springer-Verlag Lecture Notes in Computer Science vol.269, editor, *Proceedings of the 20th international symposium of Mathematical Foundation of Computer Science*, pages 82–92, 1995.

[Har95b] Juris Hartmanis. On the Weight of Computation. *Bullettin of the European Association for Theoretical Computer Science*, 55:136–138, February 1995.

[Hea87] Tom Head. Formal language theory and DNA: an analysis of the generative capacity of specific recumbinant behaviors. *Bullettin of Mathematical Biology*, 49(6):737–759, 1987.

[Hof90] Douglas Hofstadter. *Godel, Escher, Bach.* gli Adelphi editions, 1990. Chapters 6 and 16.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, 1979. chapters 7 and 8.

[Kar95] Lila Kari. DNA computing: the arrival of biological mathematics. http://www.csd.uwo.ca/ lila/amsn.ps, 1995.

[KKW95] Richard Karp, Claire Kenyon, and Orli Waarts. Error resilient DNA Computation. In *7th ACM-SIAM Symposium on Discrete Algorithms*, pages 458–467, 1995.

[KMRS96a] Stuart Kurtz, Stephen Mahaney, James Royer, and Janos Simon. Biological Computing. To be published, http://www.cs.uchicago.edu/ stuart/Research/bc.ps, 1996.

[KMRS96b] Stuart Kurtz, Stephen Mahaney, James Royer, and Janos Simon. Active Transport in Biological Computing. In *Proceedings of the Second Annual Meeting on DNA based computer, Princeton University, USA*, June 1996.

[Kol95] Gina Kolata. A vat of DNA may become the fast computer of the future. *New York Times*, November 1995.

[Leu95] Kristin Leutwyler. Calculating with DNA. *Scientific American*, pages 18–20, September 1995.

[Lip94] Richard Lipton. Speeeding Up Computation via Molecular Biology. http://www.cs.princeton.edu/ rjl/bio.ps, December 1994.

[Lip95a] Richard Lipton. Using DNA to solve SAT, 1995. http://www.cs.princeton.edu/ dabo/bio-comp/sat.ps.

[Lip95b] Richard Lipton. DNA Solution of Hard Computational Problems. *Science*, 268:542–545, April 1995.

[Lip95c] Richard Lipton. Using DNA to solve NP-complete problems. Technical report, Princeton University, USA, January 1995.

[LL95] Michal Linial and Nathan Linial. On the Potential of Molecular Computing, 1995. Letter to 'Science', April 1995.

[Luc84] Fabrizio Luccio. *La struttura degli algoritmi.* Boringhieri, 1984.

[LYW95] Lo, Yiu, and Wong, 1995. Letter to 'Science', April 1995.

[Pis96] Nadia Pisanti. *Sull'interpretazione del calcolo in sistemi di biologia molecolare.* University of Pisa, Italy, master thesis, 1996.

[Pol91] Giorgio Poli. *Microbiologia e immunologia.* UTET, second edition, 1991. Chapter 6.

[Poo96] Robert Pool. Forget silicon, try DNA. *New scientist*, 2038:26–31, July 1996.

[PS96] Gheorghe Paun and Arto Salomaa. DNA computing based on the splicing operation. *Mathematica Japonica*, 43(3):607–632, 1996.

[Rei95] John Reif. Parallel Molecular Computation. In *7th annual ACM Symposium on Parallel Algorithms and Architectures, Santa Barbara, USA*, 1995.

[Rot96] Paul Rothemund. A DNA and restriction enzyme implementation of Turing Machine. to be published in 'DNA based computer', American Mathematical Society, ISBN 0-821800518-5, 1996.

[RW95] Diana Rooßand Klaus Wagner. On the Power of DNA-Computers. Technical report, University of Wurzburg, Germany, 1995.

[RW96] Diana Rooßand Klaus Wagner. On the Power of DNA-Computing. *Information and computation*, 131:95–109, 1996.

[RWB+96] Sam Roweis, Erik Winfree, Richard Burgoyne, Nickolas V.Chelyapov, Myron F.Goodman, Paul W.K.Rothemund, and Leonard Adleman. A Sticker Based Architecture for DNA Computation. In *Proceedings of the Second Annual Meeting on DNA based computer, Princeton University, USA*, June 1996.

[Smi95] Warren Smith, 1995. An opinionated, but reasonably short, summary of the Mini DIMACS Workshop on DNA based computers, held at Princeton University on April 4 1995.

[Ste95] Willer Stemmer. The Evolution of Molecular Computation. *Science*, 270:1510–1510, December 1995.

[WHR+87] James Watson, Nancy Hopkins, Jeffrey Roberts, Joan Argetsinger Steitz, and Alan Weiner. *Molecular biology of the gene.* Benjamin/Cummings, third edition, 1987.

[Win95a] Eric Winfree. Complexity of Restricted and Unrestricted Models of Molecular Computation. Technical report, California Institute of Technology, USA, May 1995.

[Win95b] Eric Winfree. On the Computational Power of DNA Annealing and Ligation. Technical report, California Institute of Technology, USA, May 1995.