ELSEVIER

Theoretical Computer Science 287 (2002) 3–38

Theoretical Computer Science

www.elsevier.com/locate/tcs

Topics in the theory of DNA computing

Martyn Amos[a,*], Gheorghe Păun[b,c], Grzegorz Rozenberg[d,e], Arto Salomaa[f]

[a] *Department of Computer Science, School of Biological Sciences, University of Liverpool, L69 7ZF, England, UK*

[b] *Institute of Mathematics of the Romanian Academy, P.O. Box 1-764, 70700 Bucureşti, Romania*

[c] *Rovira i Virgili University, Pl. Imperial Tarraco 1, 43005 Tarragona, Spain*

[d] *Leiden Institute for Advanced Computer Science, Leiden University, Niels Bohrweg 1, CAA 2333 Leiden, The Netherlands*

[e] *Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309, USA*

[f] *Turku Centre for Computer Science, Lemminkäisenkatu 14A, 20520 Turku, Finland*

Abstract

DNA computing, or, more generally, molecular computing, is an exciting fast developing interdisciplinary area. Research in this area concerns theory, experiments, and applications of DNA computing. In this paper, we demonstrate the theoretical developments by discussing a number of selected topics. We also give an introduction to the basic structure of DNA and the basic DNA processing tools.

Keywords: DNA molecules; Biomolecular tool box; DNA computing; Molecular computing; Turing universality; Recursively enumerable languages; Boolean circuits; Cryptography; Splicing systems

0. Introduction

DNA computing is a fast growing research area concerned with the use of DNA molecules for the implementation of computational processes. One of the bold goals of this area is to design of computers which will be based on DNA molecules (rather than on silicon), and which will in the future either replace or beneficially complement silicon based computers. Although constructing computers from molecules was already suggested by R. Feynman in 1961, and quite a number of theoretical ideas were published since then, it is generally acknowledged that the real beginning of this area is

* Corresponding author.

E-mail addresses: m.amos@csc.liv.ac.uk (M. Amos), gpaun@imar.ro, gp@astor.urv.es (G. Pabreve;un), rozenber@liacs.nl (G. Rozenberg), asalomaa@utu.fi (A. Salomaa).

0304-3975/02/$ - see front matter
PII: S0304-3975(02)00134-2

the publication of the seminal paper "Molecular Computations of Solutions to Combinatorial Problems" by Adleman [2]. The reason for this generally accepted acknowledgement is that Adleman was the first one to actually implement a computation in a wet lab—in more biological terms, Adleman performed the first "proof-of-principle" experiment for DNA computing.

Since then, DNA computing has evolved into an exciting multidisciplinary research area. Although some theoretical research on computing with DNA molecules was done before 1994, the Adleman paper has instigated much theoretical research on the nature of DNA computing. By today the theory of DNA computing is well developed and it is impossible to give its account in a journal paper of reasonable size. Therefore, and also in order to give some depth to this paper (rather than to have a paper discussing too many topics in a sketchy way), we have produced a paper consisting of a (small) number of sections with each section devoted to a specific topic. This is truly a personal choice, because each of the sections is such that at least one of the authors has worked on the topic covered in that section.

Independently of the future technological success of DNA computing, this area has led already to interesting new computing paradigms which certainly enriched our understanding of the nature of computation. Since understanding of what computation is about is a central goal of (theoretical) computer science, one may say that the area of DNA computing is already successful. It is the appreciation of these new computing paradigms that we want to convey to the reader in this paper.

The paper is organized as follows.

The first two sections discuss the basic structure of DNA and describes the basic techniques of molecular biology for DNA processing. These techniques constitute the basic tool box for the experiments in DNA computing.

The third section discusses the role of the twin-shuffle language in the theory of DNA computing. As a matter of fact the "old" result from computation theory giving the representation of Turing computations through the twin-shuffle language has predicted the rise of DNA computing: DNA molecules are the physical embodiment of the twin shuffle language. Therefore, the Turing-universality of various models of DNA computing is not surprising.

Section 4 deals with applications to cryptography. In particular, steganography, one-time pads and cryptoanalysis are discussed.

Section 5 describes how to implement Boolean circuits in DNA. Since Boolean circuits are an important model of parallel computation, this is an important connection between DNA computing and the classic (theory of) computation.

DNA computing is just a subarea of molecular computing, where one considers also the use of molecules other than DNA for the purpose of computing. Section 6 discusses the paradigm of forbidding–enforcing which emerges from the investigation of molecular reactions in the context of computing.

The most important paper on the theory of DNA computing from the period preceding the Adleman's paper is [29] where Head formulates the model of the processing of DNA molecules by the restriction enzymes. Splicing systems formulated in [29] became very successful in both DNA computing and formal language theory. Section 7 gives an account of main developments in the theory of splicing systems.

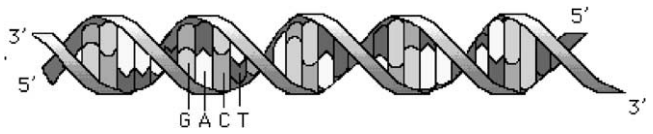

Fig. 1. Structure of double-stranded DNA.

1. The structure of DNA

DNA molecules (where DNA stands for *deoxyribonucleic acid*) [1,20,48,65] are polymers constructed from monomers called *nucleotides*. These have, for our purposes, a very simple structure, consisting of three components: sugar, phosphate and base. There are four distinct bases: adenine, guanine, cytosine and thymine, abbreviated A, G, C and T, respectively. Since nucleotides differ only in terms of their bases, we use the base abbreviations to identify them (i.e., we may introduce "G nucleotides").

Single-stranded DNA molecules are simply chains of nucleotides where two consecutive nucleotides are bound together by a strong covalent bond along a sugar-phosphate "backbone". Each single strand has, according to chemical convention, a $5'$ and a $3'$ end, thus any single strand has a natural orientation. This orientation (and, therefore, the notation used) is due to fact that one end of the single strand has a free (i.e., unattached to another nucleotide) $5'$ phosphate group, and the other has a free $3'$ deoxyribose hydroxyl group.

The most important feature of DNA is the Watson–Crick *complementarity* of bases. Bonding between single strands occurs by the pairwise attraction of bases; A bonds with T and G bonds with C. The pairs (A, T) and (G, C) are therefore known as *complementary* base pairs. The two pairs of bases form *hydrogen bonds* between each other, two bonds between A and T, and three between G and C (Fig. 2).

The classical double helix of DNA (Fig. 1) is formed when two separate strands bond. Two requirements must be met for this to occur; firstly, the strands must be complementary, and secondly, they must have opposite polarities (see Fig. 2 for an illustration).

2. Operations on DNA

All models of DNA computation apply a specific sequence of biological operations to a set of strands. These operations are all commonly used by molecular biologists. Note that some operations are specific to certain models of DNA computation.

2.1. Synthesis

Oligonucleotides may be synthesized to order by a machine the size of a microwave oven. The synthesizer is supplied with the four nucleotide bases in solution, which are

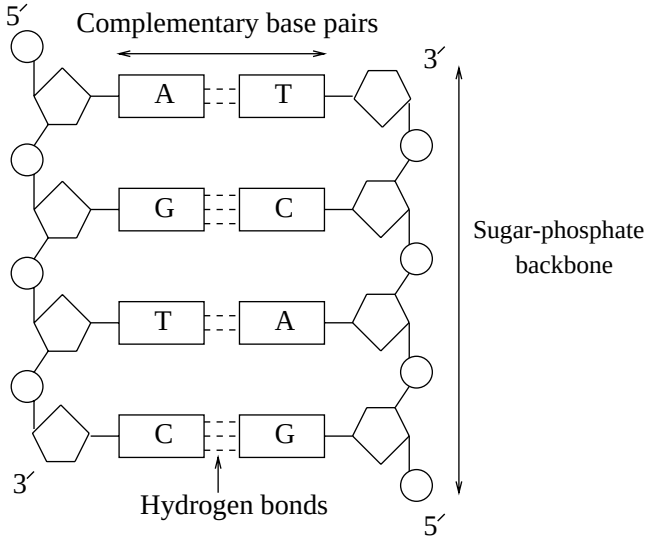

Fig. 2. Detailed structure of double-stranded DNA.

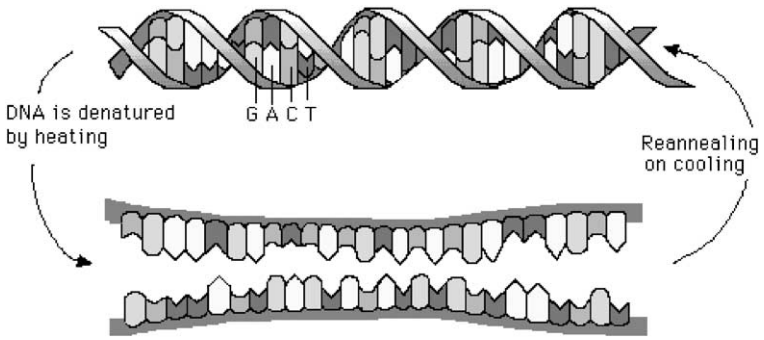

Fig. 3. DNA melting and annealing.

combined according to a sequence entered by the user. The instrument makes millions of copies of the required oligonucleotide and places them in solution in a small vial.

2.2. Denaturing, annealing and ligation

Double-stranded DNA may be dissolved into single strands (or *denatured*) by heating the solution to a temperature determined by the composition of the strand [10]. Heating breaks the hydrogen bonds between complementary strands (Fig. 3). Since the hydrogen bonds between strands are much weaker than the covalent bonds *within* strands, the strands remain undamaged by this process. Since a *G–C* pair is joined by

three hydrogen bonds, the temperature required to break it is slightly higher than that for an A–T pair, joined by only two hydrogen bonds. This factor must be taken into account when designing sequences to represent computational elements.

Annealing is the reverse of melting, whereby a solution of single strands is cooled, allowing complementary strands to bind together (Fig. 3).

In double-stranded DNA, if one of the single strands contains a discontinuity (i.e., one nucleotide is not bonded to its neighbour), then this may be repaired by DNA *ligase* [11].

2.3. Hybridisation separation

Separation by hybridisation is an operation often used in DNA computation, and involves the extraction from a test tube of any *single* strands containing a specific short sequence (e.g., extract all strands containing the sequence $TAGACT$). If we want to extract single strands containing the sequence x, we first create many copies of its complement. We attach to these oligonucleotides a biotin molecule [1] which bind in turn to a fixed matrix. If we pour the contents of the test tube over this matrix, strands containing x will anneal to the anchored complementary strands. Washing the matrix removes all strands that did not anneal, leaving only strands containing x. These may then be removed from the matrix.

2.4. Gel electrophoresis

Gel electrophoresis is an important technique for sorting DNA strands by size [11]. Electrophoresis is the movement of charged molecules in an electric field. Since DNA molecules carry negative charge, when placed in an electrical field they tend to migrate towards the positive pole. The rate of migration of a molecule in an *aqueous* solution depends on its shape and electrical charge. Since DNA molecules have the same charge per unit length, they all migrate at the same speed in an aqueous solution. However, if electrophoresis is carried out in a *gel* (usually made of agarose, polyacrylamide or a combination of the two) the migration rate of a molecule is also affected by its *size*. [2] This is due to the fact that the gel is a dense network of pores through which the molecules must travel. Smaller molecules therefore migrate faster through the gel, thus sorting them according to size.

A simplified representation of gel electrophoresis is depicted in Fig. 4. The DNA is placed in a well cut out of the gel, and a charge applied.

Once the gel has been run (usually overnight), it is necessary to visualize the results. This is achieved by staining the DNA with the fluorescent dye ethidium bromide and then viewing the gel under ultraviolet light. At this stage the gel is usually photographed for convenience.

One such photograph is depicted in Fig. 5. Gels are interpreted as follows; each *lane* (1–7 in our example) corresponds to one particular sample of DNA. We can

[1] This process is referred to as "biotinylation".

[2] Migration rate of a strand is inversely proportional to the logarithm of its molecular weight [52].

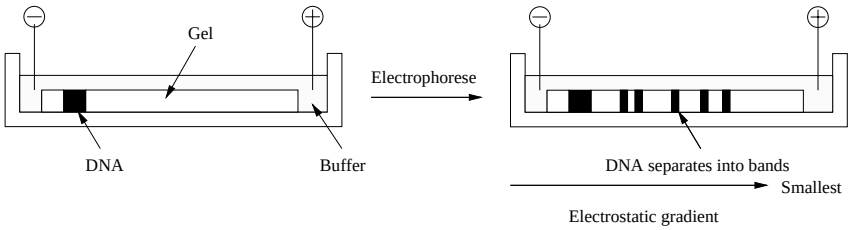

Fig. 4. Gel electrophoresis process.

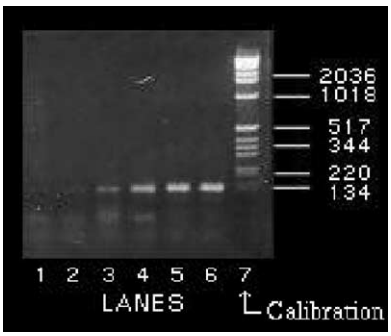

Fig. 5. Gel electrophoresis photograph.

therefore run several tubes on the same gel for the purposes of comparison. Lane 7 is known as the *marker lane*; this contains various DNA fragments of known length, for the purposes of calibration. DNA fragments of the same length cluster to form visible horizontal *bands*, the longest fragments forming bands at the top of the picture, and the shortest at the bottom. The brightness of a particular band depends on the amount of DNA of the corresponding length present in the sample. Larger concentrations of DNA absorb more dye, and therefore appear brighter. One advantage of this technique is its sensitivity—as little as 0.05 μg of DNA in one band can be detected as visible fluorescence.

The size of fragments at various bands is shown to the right of the marker lane, and is measured in *base pairs* (b.p.). In our example, the largest band resolvable by the gel is 2036 b.p. long, and the shortest 134 b.p. Moving right to left (tracks 6–1) is a series of PCR reactions which were set up with progressively diluted target DNA (134 b.p.) to establish the sensitivity of a reaction. The dilution of each tube is evident from the fading of the bands, which eventually disappear in lane 1.

2.5. Primer extension and PCR

The DNA *polymerases* perform several functions, including the repair and duplication of DNA. Given a short *primer* oligonucleotide, p, in the presence of nucleotide triphosphates, the polymerase extends p (always in the $5' \to 3'$ direction) if and only

Fig. 6. Primer extension by polymerase.

if p is bound to a longer *template* oligonucleotide, t. For example, in Fig. 6(a), p is the oligonucleotide TCA which is bound to t, $ATAGAGTT$. In the presence of the polymerase, p is extended by a complementary strand of bases to the $3'$ end of t (Fig. 6(b)).

Another useful method of manipulating DNA is the *Polymerase Chain Reaction*, or PCR [36,37]. PCR is a process that quickly amplifies the amount of a specific molecule of DNA in a given solution using primer extension by polymerase. Each cycle of the reaction doubles the quantity of this molecule, giving an exponential growth in the number of strands. In order to target specific molecules we need to know their "start" and "end" sections. A common problem in DNA computation is how to read-out the final solution to a problem encoded as a DNA strand, as the laboratory steps carried out may result in a very dilute solution. PCR solves this "needle in a haystack" problem: if a sought after molecule is present in the solution, then it will be hugely (exponentially) multiplied so that the volume of the solution will "visibly" grow—this solves then the detection problem.

2.6. Restriction enzymes

Restriction endonucleases [68, p. 33] (often referred to as *restriction enzymes*) recognize a specific sequence of DNA, known as a *restriction site*. Any *double-stranded* DNA that contains the restriction site within its sequence is cut by the enzyme at that point.[3] For example, the double-stranded DNA in Fig. 7(a) is cut by restriction enzyme *Sau3*AI, which recognizes the restriction site $GATC$. The resulting DNA is depicted in Fig. 7(b). The cleavage here generates "sticky" (cohesive) ends. Such ends are important in DNA manipulation, because they allow to catenate DNA molecules if they have complementary sticky ends.

3. Déjà vu: why universality is not surprising

So far quite many and diverse theoretical models have been proposed for DNA-based computing. The early ones are discussed in [48]. While the models have been based

[3] In reality, only certain enzymes cut specifically at the restriction site, but we take this factor into account when selecting an enzyme.

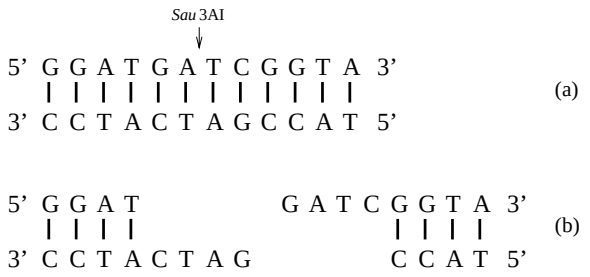

Fig. 7. Double-stranded DNA being cut by *Sau*3AI.

on different ideas and principles, Watson–Crick complementarity is somehow present in a computation or derivation step in a great majority of the models. This is natural, in view of the central role of complementarity in DNA operations. A typical model of DNA computing consists of augmenting a computational aspect of complementarity with some input–output format.

A property shared by most of the models is that they produce all recursively enumerable sets, that is, are universal in the sense of Turing machines. This property seems to be completely independent, for instance, of a model being grammatical or a machine model. Complementarity, augmented with adequate input–output facilities, seems to guarantee universality.

Why is this not surprising? This is something we have already seen before in theoretical computer science. We will now establish a link with certain fairly old results from computability theory, with the purpose of showing that complementarity is, in fact, a source of universality. Complementarity is such a powerful tool because it brings, in a certain sense, the universal *twin-shuffle language* to the computing scene. We are now ready for the formal details.

Consider the *DNA-alphabet* $\Sigma_{DNA}=\{A,G,T,C\}$, as well as the letter-to-letter morphism $h_W:\Sigma^*_{DNA}\to\Sigma^*_{DNA}$ defined by

$$h_W(A)=T,\quad h_W(G)=C,\quad h_W(T)=A,\quad h_W(C)=G.$$

The morphism h_W will be called the *Watson–Crick morphism*. Clearly, its square is the identity. Words over Σ_{DNA} can be viewed as single strands. Two single strands x and y are complementary (and, thus, subject to bondage) if $x=h_W(y)$ or, equivalently, $y=h_W(x)$. The morphism h_W is denoted also by an upper bar: $h_W(x)=\bar{x}$. Thus, in this notation, the double bar will be the identity: $\bar{\bar{x}}=x$. Moreover, we will view the DNA-alphabet as an extended binary alphabet $\{0,1,\bar{0},\bar{1}\}$, with the conventions:

$$A=0,\qquad G=1,\qquad T=\bar{0},\qquad C=\bar{1}.$$

(Observe that this agrees with the bar notation for the Watson–Crick morphism.)

A generalization of the DNA-alphabet and our extended binary alphabet is the *DNA-like alphabet*

$$\Sigma_n=\{a_1,\ldots,a_n,\bar{a_1},\ldots,\bar{a_n}\},\quad n\geqslant 2.$$

The letters in the unordered pairs $\{a_i, \bar{a}_i\}$, $1 \leqslant i \leqslant n$, are called *complementary*. Again, the morphism h_W mapping each letter to its complementary one is called the *Watson–Crick morphism* and also denoted by a bar.

3.1. The twin-shuffle language

The *twin-shuffle language TS* consists of all words over the alphabet $\{0, 1, \bar{0}, \bar{1}\}$, obtained in the following fashion. Take an arbitrary word w over $\{0, 1\}$, its complementary word $\bar{w}$, and shuffle the two in an arbitrary way. (Here we are using the customary language-theoretic shuffle operation, analogous to the shuffling of two decks of cards. The order of letters in the two words remains unchanged but the two words can be put to any order with respect to each other, including the orders $w\bar{w}$ and $\bar{w}w$.) For instance, the word $0\bar{0}0\bar{0}1\bar{1}0\bar{0}$ is in *TS*, whereas the word $0\bar{0}0\bar{0}1\bar{1}0\bar{1}0\bar{1}1$ is not. All words in *TS* contain equally many barred and nonbarred letters but, of course, this condition is not sufficient for a word to be in *TS*.

The *generalized twin-shuffle language* TS_n over the DNA-like alphabet is defined exactly as *TS* except that now w ranges over the words over the alphabet $\{a_1, \ldots, a_n\}$. We consider also the *reverse twin-shuffle language RTS*, defined as *TS*, except that now the words w and $\bar{w}^R$ are shuffled, where x^R denotes the reverse (also called the mirror image) of x.

We now come to the *universality* of the twin-shuffle language *TS*. The universality is due to the following *basic representation result* for recursively enumerable languages: for every recursively enumerable language L, a gsm-mapping g such that $L = g(TS)$ can be effectively constructed. (Here "gsm" refers to "generalized sequential machine", a device obtained by providing a finite automaton with outputs, see [58].) This result was established in [24]. For various proofs and the history of this result, the reader is referred to [58].

The basic representation result shows why *TS* is universal: it remains the same for all languages. Only the mapping g (that can be viewed to constitute the input–output format) has to be specified differently according to each particular L, in other words, according to the needs of each particular "task". The result is also highly invariant, which shows its fundamental character. It remains valid also if *RTS* is taken instead of *TS*. This is important because, in some theoretical models and certainly in nature, DNA double strands are read according to their orientation, which leads to words in *RTS*.

A further analysis of the mapping g leads to various strengthenings of the basic representation result. Strengthenings are needed for various reasons. The following, referred to as the *modified representation result*, is particularly suitable for various models of DNA computing: every recursively enumerable language L can be represented in the form

$$L = p(TS_n \cap R),$$

for some $n \geqslant 2$, regular language R, and projection p. (By a *projection* we mean a morphism mapping some letters into themselves, in this case the letters of L, and

erasing all the other letters.) Again, the items R, p, n are effectively constructable, provided L is effectively given.

We refer to [58] for a proof of the modified representation result. The modified version is stronger than the basic one, because it tells us that we may restrict the attention to a particular kind of gsm-mappings, namely, the composition of three mappings resulting from the operations $p, \cap R$, and the transition from TS to TS_n. A projection means simply erasing something from the output. The intersection with a regular language can be implemented with a finite automaton, and the third mapping means modifying the input. Altogether the modified version fits very well to machine models of DNA computing. A case study can be found in [57]. A variety of examples is given in [48].

The representation results presented above exhibit the universality of the twin-shuffle language TS. On the other hand, the interconnection between TS and Watson–Crick complementarity is rather obvious and will be discussed below. The universality of the Watson–Crick complementarity was first pointed out in [56] and elaborated further in [60,57].

3.2. Interconnection between TS and complementarity

The interconnection between the language TS and Watson–Crick complementarity can be presented in various ways, depending on the method of reading double strands as single strands. We now discuss two such methods. Instead of the DNA-alphabet $\{A, G, T, C\}$, we use the extended binary alphabet $\{0, 1, \bar{0}, \bar{1}\}$ in the way described above. Thus (disregarding orientation) the DNA double strands Z are of the form

$$\begin{array}{l} x_1\ x_2\ \ldots\ x_n, \\ \overline{x_1}\ \overline{x_2}\ \ldots\ \overline{x_n}, \end{array}$$

where each x_i is a letter of the extended binary alphabet, and double bars can be ignored in the way described above. We will first construct a single strand (or a word) from the double strand Z by the *up–down* method, taking letters alternately from upper and lower strands, beginning from the upper strand. The result is

$$UD(Z) = x_1\overline{x_1}x_2\overline{x_2}\ldots x_n\overline{x_n}.$$

The word $UD(Z)$ is always in TS. Indeed, words of the form $UD(Z)$ constitute the regular subset $\{0\bar{0}, \bar{0}0, 1\bar{1}, \bar{1}1\}^*$ of TS.

Secondly, construct from Z a single strand $FO(Z)$ *following the orientation*: read the upper strand from left to right and catenate the result by reading the lower strand from right to left. Thus,

$$FO(Z) = x_1x_2\ldots x_n\overline{x_n}\ldots\overline{x_2x_1}.$$

Clearly, $FO(Z)$ is in the language RTS but all words of RTS are not obtained from double strands in this fashion.

A way for obtaining all strings in *TS* by scanning the nucleotides of DNA molecules is based on the encoding suggested below:

	upper strand	lower strand
A, T	0	$\bar{0}$
C, G	1	$\bar{1}$

In other words, both nucleotides A and T are identified with 0, without a bar when appearing in the upper strand and barred when appearing in the lower strand; the nucleotides C, G are identified with 1 in the upper strand and with $\bar{1}$ in the lower strand. Given a DNA (double-stranded) molecule, by reading nondeterministically the two strands from left to right, one nucleotide at a time, we get strings in *TS*.

Conversely, we can obtain *all* strings in *TS* if we consider *all* molecules (complete double stranded sequences) and *all* possibilities to read them as specified above. The same result is obtained if we use molecules containing in the upper strand only nucleotides in any of the pairs

$$(A, C),\ (A, G),\ (T, C),\ (T, G).$$

If we read the two strands of a molecule in opposite directions, then we get all strings in *RTS*.

Consider now the reverse problem of constructing a double strand from a word in *TS* or *RTS*. We discuss here only the case of *TS*. Let y be a nonempty word in *TS*. Necessarily, y is of even length, $|y| = 2m$. Moreover, the scattered subword y' (resp. y'') of y consisting of nonbarred (resp. barred) letters is of length m. For $1 \leqslant i \leqslant m$, we denote by y'_i (resp. y''_i) the ith letter of y' (resp. y''). Because y is in *TS*, the unordered pair (y'_i, y''_i) equals either $(0, \bar{0})$ or $(1, \bar{1})$. When we speak of y'_i or y''_i, we have these particular occurrences in mind. The occurrences may lie far apart in y. However, one of them is always to the left of the other. The left occurrence is referred to as the *up-occurrence at position i*, the right occurrence is similarly referred to as the *down-occurrence at position i*.

Consider now the double strand of length m, where for $1 \leqslant i \leqslant m$, the ith letter in the upper (resp. lower) strand is the up- (resp. down-) occurrence at position i in y. This double strand is called the *left parse* of y and denoted $LP(y)$. Clearly, $LP(y)$ satisfies the complementarity requirement for DNA double strands. Observe that LP is not injective, for instance,

$$LP(\bar{1}\bar{0}10) = LP(\bar{1}1\bar{0}0).$$

On the other hand, for all double strands Z, we have $LP(UD(Z)) = Z$. The equation $UD(LP(y)) = y$ is valid if y belongs to the aforementioned subset $\{0\bar{0}, \bar{0}0, 1\bar{1}, \bar{1}1\}^*$ of *TS*.

We have shown how to go from words in *TS* to DNA double strands, and vice versa. Our observations can be summarized as follows.

For any nonempty word y in the twin-shuffle language *TS*, $LP(y)$ is a unique DNA double strand. For any DNA double strand Z, $UD(Z)$ is a unique word in the subset

$\{0\bar{0}, \bar{0}0, 1\bar{1}, \bar{1}1\}^*$ of *TS*. When restricted to this subset, *LP* is the inverse of *UD*. For any DNA double strand Z, $FO(Z)$ is a unique word in the reverse twin-shuffle language *RTS*.

The strength of the representation results (such as the basic and modified result presented above) is shown also by their invariance. We have seen how both *TS* and *RTS* can be used as a basis. Similarly, the universality results are not affected if one assumes that one of the strands in the double strands contains only purines (nonbarred letters).

Watson–Crick complementarity is a phenomenon provided for us "for free" by nature. When bondage takes place (under ideal conditions) between two single strands, we know that the bases opposite each other are complementary. This information is "free"; there is no need to check it in any way. At a first glance, it might seem that not much information is obtained: one just reads the same information twice when investigating a double strand. However, conclusions can be made from the *history* of a double strand, from the knowledge of how it came into being. The conclusions in Adleman's experiment are made in this way. If we know how information was encoded on the DNA strands subjected to bondage, we may learn much from the fact that bondage has actually taken place.

4. DNA computing and cryptography

We begin with listing some general notions. Our over all term for activities dealing with secret writing is *cryptography*. It includes both the activities of legal users of the system ("good guys"), as well as eavesdroppers ("bad guys"). The basic set-up consists of a message being sent through an insecure channel, where it may be intercepted by an eavesdropper. The basic goal of the eavesdropper is to violate the secrecy of the communication and benefit from the secret information. There might be also more sophisticated goals. The eavesdropper might alter the message, thus confounding the legal receiver with a corrupted message. In this fashion the eavesdropper also deceives the receiver about the identity of the sender.

The message m in its original form will be referred to as the *plaintext*. The sender *encrypts* the plaintext, thus obtaining the *cryptotext* or *ciphertext* c, in symbols, $E(m)=c$. After receiving c through the (insecure) channel, the receiver *decrypts* it: $D(c)=m$. The encryption and decryption methods, E and D, are referred to as *keys*. In *classical or symmetric cryptosystems*, the decryption key D can be easily computed from the encryption key E and, consequently, the latter should be kept secret. In *public-key or unsymmetric cryptosystems*, computing D from E is intractable and, thus, E can be publicized. In any case, the eavesdropper should not know D. *Cryptanalysis* refers to the activities of the eavesdropper. There are different set-ups for cryptanalysis. In the set-up *cryptotext only* the analysis has to be based only on some samples of cryptotext. In the set-up *known plaintext* the cryptanalyst knows in advance some pairs $(m, E(m))$. The set-up *chosen plaintext* differs in the respect that the cryptanalyst has been able to choose the plaintext m in the previously known pairs $(m, E(m))$. Details about these matters can be found in [59] or [62]. We present here

only (in somewhat modernized form) the three requirements for good cryptosystems proposed by Sir Francis Bacon.

(1) Given E and m, the computation of $E(m)$ is easy. Given D and c, the computation of $D(c)$ is easy.
(2) Without knowing D, it is impossible to find m from c.
(3) The cryptotext should be without suspicion: look innocent.

The requirements become very clear also in Bacon's own words [39, p. 123]:

> The virtues of them, whereby they are to be preferred, are three: that they be not laborious to write and read; that they be impossible to decipher; and, in some cases, that they be without suspicion.

The frustration of Bacon becomes visible later on (p. 529) in the same reference:

> But such is the rawness and unskilfulness of secretaries and clerks in the courts of kings, that the greatest matters are commonly trusted to weak and futile ciphers.

We will now discuss briefly some areas, where DNA-based methods might be used to improve "weak and futile ciphers".

4.1. DNA and steganography

The art of *steganography* (hiding a message) is very old. The very existence of a secret is concealed. Perhaps the oldest example is told by Herodotos, about Histaios who allowed the shaving of the head of a slave, writing the message thereon and waiting the hair to grow again, after which the messenger was ready to travel. Other tricks used include invisible inks, small pin punctures on certain characters, pencil marks on typewritten characters and minute differences between handwritten characters. Solomaa [59] describes a method used by Richelieu: grilles which cover most of the message except for a few characters. In this way sinister orders can be concealed in an innocent-looking love letter.

One can argue that steganography is not actually encryption, since plaintext is not encrypted but only disguised within other media. In the literature, see [62], steganographic methods are generally considered to have low security and, indeed, they have many times been broken in practice. On the other hand, the simplicity of steganographic methods is very appealing.

Several techniques are presented in [27] for applying steganography in the context of DNA computing. One method consists of taking one or more input DNA strands (considered to be the plaintext) and appending to them one or more *secret key strands*. The latter are preferably constructed randomly. The resulting *tagged plaintext strands* are then hidden by mixing them with many additional *distracter strands*. The latter may again be chosen from a random assembly.

If the secret key strands are known, the entire solution of DNA strands (ciphertext) can be *decrypted* by some of the known recombinant DNA separation methods. For instance, the plaintext message strands may be separated out by hybridization with the complementary strands of the secret key strands. The separation steps may be combined with amplification steps.

As regards *cryptanalysis*, the security of the above system depends entirely on the fact that the enemy (eavesdropper) is either unaware of the existence of the message

in the medium of transmission, or cannot distinguish the tagged plaintext strands. (This is always the state of affairs in steganography.) Thus, it is of crucial importance that the tagged plaintext strands are indistinguishable from the distracter strands. We may assume that secret key strands are indistinguishable from distracter strands, since both come from a random source. However, if the plaintext comes from a natural source such as a natural language or natural DNA and is not initially compressed, then the tagged plaintext strands can be distinguished from the distracter strands, the latter coming from a random source. Estimates about the probabilities involved are given in [27]. However, the security of the original DNA steganography system may be enhanced. First, the construction of the distracter strands can be improved to *mimic the plaintext source distribution*. This means that the distracter strands are not chosen randomly but from a source, where it is difficult to distinguish probability distributions from those of the plaintext source. Secondly, one may recode the plaintext using a suitable compression algorithm. In this case resulting distributions of the recoded plaintext will approximate universal distributions.

4.2. One-time pads

Perfect secrecy means, briefly stated, that the cryptotext does not give away any information whatsoever to the cryptanalyst. The cryptanalyst may or may not intercept the cryptotext: he/she has exactly the same knowledge in both cases. The cryptotext gives away no information about the plaintext. Such a situation is achieved by *one-time pads*. The plaintext is of bounded length, say a sequence of at most 20 bits. The key is a sequence of 20 bits. It is used both for encryption and decryption and communicated to the receiver via some secure channel. Take the key 11010100001100010010. A plaintext, say 010001101011, is encrypted using bitwise addition with the bits of the key, starting from the beginning of the key. Thus, the cryptotext is 100100101000. This as such gives no information to the cryptanalyst because any specific bit of the cryptotext might come directly from the plaintext or might have been changed by the key. It is essential that the key is used only once, as also the name indicates. A previous plaintext together with the corresponding cryptotext give away a prefix of the key. Also a set of previous cryptotexts, with plaintexts remaining unknown, give away some information. Legal decryption is obvious: bitwise addition of the cryptotext to a prefix of the key.

The obvious disadvantage of one-time pads is the difficult key management. The key, at least as long as the plaintext, has to be communicated separately via some secure channel. In some sense, nothing has been accomplished: the difficulties in secret communication have only been transferred to a different level. If one-time pads in the form of DNA strands are used, then the situation is facilitated by the very compact nature of DNA in solution.

In fact, one-time pad methods using DNA have been suggested in [27]. First, a large one-time pad in the form of a DNA strand is assembled. More specifically, it is randomly assembled from short oligonucleotide sequences, and then isolated and cloned. These one-time pads are constructed in secret, and shared in advance by the sender and receiver of the secret message. Thus, the one-time pad has to be initially

communicated between the sender and receiver. The communication is facilitated by the large information storage capacity of DNA.

Two methods are proposed in [27], whereby a large number of short message sequences can be encrypted, in such a way that the original plaintext message cannot be determined from the resulting DNA. The bitwise computations are accomplished via biomolecular techniques. The techniques used in the two methods consist of substitutions, where each message sequence is encoded by an associated matching with the corresponding sections of the one-time pad or, alternatively, direct bitwise additions. The decryption is accomplished similarly. For instance, in the former method (substitution), one test tube of short DNA strands (the plaintext messages) is converted into another set of entirely different strands (the cryptotexts) in a random yet deterministic and reversible way. The original plaintext strands are removed in the process.

4.3. Cryptanalysis of DES

Many of the celebrated computationally intractable problems can be solved by an exhaustive search through all possible solutions. However, the insurmountable difficulty lies in the fact that such a search is too vast to be carried out using present technology. On the other hand, the density of information stored in DNA strands and the ease of constructing many copies of them might render such exhaustive searches possible. A typical example is *cryptanalysis*: all possible keys can be tried out simultaneously. Moreover, cryptographic tasks in general seem to be very suitable for DNA computing, since error rates much greater than those normally required of electronic computers will suffice. If a cipher can be broken in 95% of the cases, the threat is already adequately serious.

Data encryption standard, *DES* has been the most widely used cryptosystem. (See [59] for a detailed description of the system.) The cryptanalysis presented in [3] suggests that an attack might be mounted on a table-top machine, based on DNA computing but using also robotic parts. Approximately one gram of DNA would be needed. Quite importantly, the attack is likely to succeed even in the presence of a large number of errors. Thus, even if some of the DNA operations are error prone, the attack might succeed with a reasonable probability.

The cryptanalysis presented in [3] uses the *sticker model* for DNA computing. (See [48] for details of the sticker model and its history.) The model uses two basic kinds of DNA molecules, referred to as *memory strands* and *stickers*. A memory strand is n bases in length and contains k nonoverlapping substrands, each of which is r bases long. Thus, we must have $n \geqslant rk$. During the course of a computation, each substrand is identified with exactly one bit position. The substrands should be significantly different from one another. Each sticker is r bases long and complementary to one of the k substrands in the memory strand. A specific substrand of a memory strand is either *on* or *off*, depending on whether or not a sticker is annealed to it. A *memory complex* is a general term used for memory strands, where the substrands are on or off. Memory complexes represent binary numbers, where a substrand being on (resp. off) represents the bit 1 (resp. 0). A (k,l) *library*, $1 \leqslant l \leqslant k$, consists of memory complexes with k substrands, the last $k-l$ of which are off, whereas the first l substrands are on or off

in all possible ways. Thus, the represented binary sequences are of the form $w0^{k-l}$, where w is an arbitrary binary sequence of length l.

A *test tube* in the sticker model is a multiset of memory complexes. A *computation* consists of a sequence of four operations *merge*, *separate*, *set*, *clear*. In the operation *merge* two test tubes are combined into one. Given a test tube T and an integer i, $1 \leqslant i \leqslant k$, the operation *separate* produces two test tubes $+(T,i)$ and $-(T,i)$, where the former (resp. the latter) consists of all the memory complexes in the original T, where the ith substrand is on (resp. off). Given T and i, the operation *set* (resp. *clear*) produces a new test tube, where the ith substrand of each memory complex in T is turned on (resp. off).

The cryptosystem DES (in its basic version) translates plaintext blocks 64 bits in length into 64-bit cryptotext blocks under the control of a 56-bit key. (In fact, also the key is given as a 64-bit word, but 8 bits are determined by the others and used only for error detection in key distribution and storage.) The same key is used both for encryption and decryption: DES is a symmetric system. The cryptanalytic attack presented in [3] is based on the set-up *known plaintext*. Thus, the analyst knows a pair $(m,E(m))$ consisting of the plaintext and the corresponding cryptotext. The analysis is based on an exhaustive search through all 2^{56} keys.

The initial test tube will be a (579,56) library. The first 56 bits represent all possible keys. Another region of 64 bits will, after the computation, show the cryptotext corresponding to the plaintext m. The remaining substrands are needed to store intermediate results during the computation. In [3] the substrands in the memory complexes are oligonucleotides of length 20. The known pair $(m,E(m))$ is not represented in the memory complexes. It always remains the same; each of the keys works on this particular plaintext m, and the resulting cryptotext is compared with $E(m)$. Thus, the whole cryptanalysis consists of the following three steps.

(1) Construct the initial (579,56) library, representing all possible 2^{56} keys.
(2) On each memory complex, compute the cryptotext obtained by encrypting the known plaintext m by the key represented by the memory complex.
(3) Select the memory complex whose cryptotext matches the known cryptotext $E(m)$, and read its key.

The main part of the work is step 2. The machine implementing the algorithm can be envisioned as a parallel robotic work station. It consists of a rack of tubes, some robotics, as well as a microprocessor that controls the robotics. The robotics perform any of the four operations discussed above in connection with the sticker model. Moreover, the robotics are capable of performing the operations in a parallel sense, for instance, they can merge the DNA from 64 data tubes into one data tube, or separate the DNA from 32 data tubes, by using 32 specific operator tubes, into two or more data tubes. The reader is referred to [3] or [48] for details, as well as estimates about the success rate under various assumptions concerning the reliability of the DNA operations. We conclude this section by explaining the construction in step 1, the creation of the initial library. How can one obtain all the possible 2^{56} keys? The technique is also of general interest in DNA computing.

We begin with approximately 2^{56} identical memory strands (single strands) of the correct length, and divide them equally into two tubes T_1 and T_2. Large amounts of

each of the 56 stickers are added to T_1, so that in the ligation reaction all of the 56 appropriate substrands in T_1 are turned on. The unused stickers are washed away from T_1, after which T_1 and T_2 are merged into one tube T. Finally, T is heated and cooled, to randomly reanneal the stickers. Roughly 63% of the keys will be represented after this process. (The figure comes from the Poisson distribution.) If we begin with three times the necessary amount of DNA, the percentage is increased to 95%.

5. Boolean circuits

Boolean circuits are an important Turing-equivalent model of parallel computation (see [21,28]). The question of whether we can implement these at the molecular level using DNA is therefore of great interest. In this section we describe one such implementation.

An n-input *bounded fan-in* Boolean circuit may be viewed as a directed, acyclic graph, S, with two types of node: n *input* nodes with in-degree (i.e., input lines) zero, and *gate* nodes with maximum in-degree two. Each input node is associated with a unique Boolean variable x_i from the input set $X_n = (x_1, x_2, \ldots, x_n)$. Each gate node, g_i is associated with some Boolean function $f_i \in \Omega$. We refer to Ω as the circuit *basis*. A *complete* basis is a set of functions that are able to express all possible Boolean functions. It is well known that the NAND function provides a complete basis by itself, but for the moment we consider the common basis, according to which $\Omega = \{\wedge, \vee, \neg\}$. In addition, S has some unique *output* node, s, with out-degree zero. An example of a Boolean circuit for the three-input *majority function* is depicted in Fig. 8.

The two standard complexity measures for Boolean circuits are *size* and *depth*: the size, m, of a circuit S is the number of gates in S; its depth, d, is the number of gates in the *longest* directed path connecting an input vertex to an output gate. The circuit depicted in Fig. 8 has size 8 and depth 3.

5.1. DNA-based Boolean circuits

We now describe a DNA-based implementation of Boolean circuits, first described in [5] (and subsequently in [6]). Since it is well-known [21,28,66] that the NAND function provides a complete basis by itself, we restrict the model to the simulation of such gates. In fact, the realisation in DNA of this basis provides a far less complicated simulation than using other complete bases. It is interesting to observe that the fact that NAND offers the most suitable basis for Boolean network simulation within DNA computation continues the traditional use of this basis as a fundamental component within new technologies. Thus, from the work of Sheffer [63] that established the completeness of NAND with respect to propositional logic, through classical gate-level design techniques [28], and, continuing, in the present day, with VLSI technologies both in nMOS [35], and CMOS [67, pp. 9,10].

The simulation takes place in three distinct phases:

(1) Set-up.
(2) Level simulation.

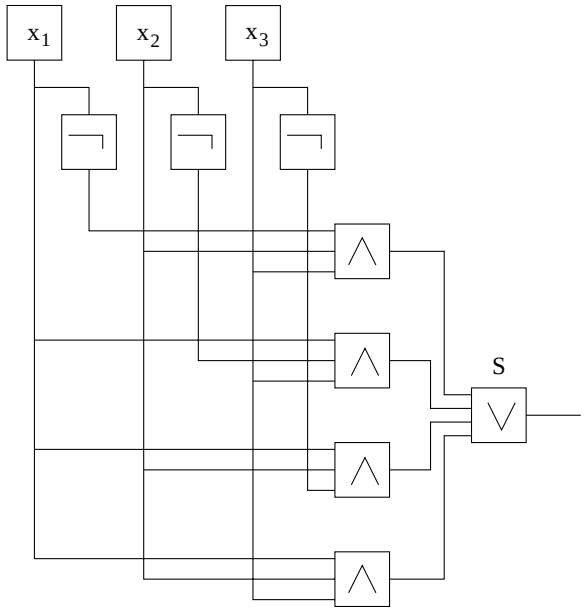

Fig. 8. Boolean circuit for the three-input majority function.

(3) Final read-out of output gates.
We now describe each phase in detail.

5.2. Set-up

In what follows we use the term *tube* to denote a set of strings over some alphabet σ. We denote the jth gate at level k by g_k^j. We first create a tube, T_0, containing unique strings of length l, each of which corresponds to only those input gates that have the value 1. We then create, for each level $1 \leqslant k < D(S)$, a tube T_k containing unique strings of length $3l$ representing each gate at level k. We also create a tube S_k, containing strings corresponding to the complement of positions $2l-5$ to $2l+5$ for each g_k^j. We assume that if sequence x and its complement are present in the same tube, the string containing sequence x is in some way "marked".

We then create tube $T_{D(S)}$, containing unique strings representing the output gates $\langle t_1, \ldots, t_m \rangle$. These strings representing gates at level $1 \leqslant k < D(S)$ are of the form $x_k^j y_k^j z_k^j$. If gate g_k^j takes its input from gates g_{k-1}^m and g_{k-1}^n, then the sequence representing x_k^j is the complement of the sequence representing z_{k-1}^m, and y_j^k is the complement of the sequence representing z_{k-1}^n. The presence of z_k^j therefore signifies that g_k^j has an output value of 1.

The strings in tube $T_{D(S)}$ are similar, but the length of the sequence $z_{D(S)}^j$ is in some way proportional to j. Thus, the length of each string in $T_{D(S)}$ is linked to the index of the output gate it represents.

5.3. Level simulation

We now describe how levels $1 \leqslant k < D(S)$ are simulated. We create the set union of tubes T_{k-1} and T_k. Strings representing gates which take either of their inputs from a gate with an output value of 1 are "marked", due to their complementary nature. We then remove from T_k all strings that have been marked twice (i.e., those representing gates with both inputs equal to one). We then split the remaining strings after section y_k^j, retaining the sequences representing z_k^j. This subset then forms the input to tube T_{k+1}.

5.4. Final read-out of output gates

At level $D(S)$ we create the set union of tubes $T_{D(S)-1}$ and $T_{D(S)}$ as described above. We then, as before, remove from this set all strings that have been marked twice. By checking the length of each string in this set we are therefore able to say which output gate has the value 1, and which has the value zero by the presence or absence of a string representing the gate in question.

5.5. Physical implementation

We now describe how the abstract model detailed in the previous paragraphs may be implemented in the laboratory using standard bio-molecular manipulation techniques. The implementation is similar to that of the *parallel filtering model*, described in [4,8]. We first describe the design of strands representing the input gates X_n. For each X_n that has the value 1 we synthesize a unique strand of length l. We now describe the design of strands representing gates at level 1. We have already synthesized a unique strand to represent each g_k^j at the set-up stage. Each strand is comprised of three components of length l, representing the gate's inputs and output. Positions 0 to l represent the first input, positions $l+1$ to $2l$ represent the second input, and positions $2l+1$ to $3l$ represent the gate's output. Positions $l-3$ to $l+3$ and positions $2l-3$ to $2l+3$ correspond to the restriction site $CACGTG$. This site is recognized and cleaved exactly at its mid-point by the restriction enzyme *Pml*I, leaving blunt ends. Due to the inclusion of these restriction sites, positions 0 to 2, $l+1$ to $l+3$ and $2l+1$ to $2l+3$ correspond to the sequence GTG, and positions $l-3$ to l, $2l-3$ to $2l$ and $3l-3$ to $3l$ correspond to the sequence CAC. The design of the other sub-sequences is described in Section 5.2. A graphical depiction of the structure of each gate strand is shown in Fig. 9.

The simulation proceeds as follows for levels $1 \leqslant k < D(S)$.

(1) At k pour into T_k the strands in tube T_{k-1}. These anneal to the gate strands at the appropriate position.
(2) Add ligase to T_k in order to seal any "nicks".
(3) Add to T_k the restriction enzyme *Pml*I. Because of the strand design, the enzyme cleaves only those strands that have *both* input strands annealed to them. This is due to the fact that the *first* restriction site $CACGTG$ is only made fully double-stranded if both of these strands have annealed correctly. This process is depicted in Fig. 10.

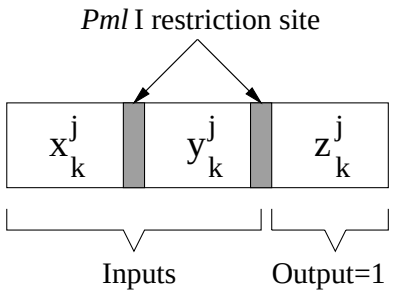

Fig. 9. Structure of a gate strand.

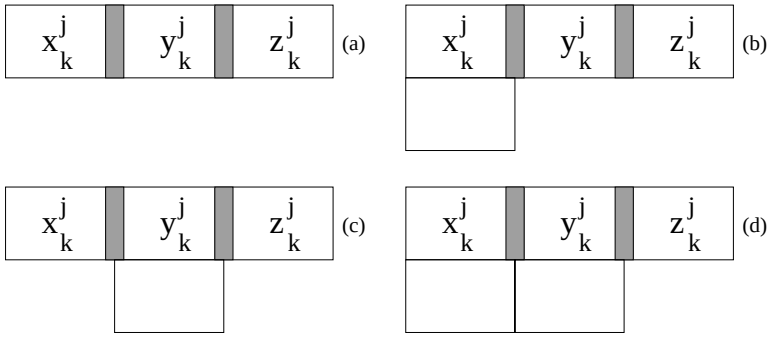

Fig. 10. (a) Both inputs= 0; (b) first input= 1; (c) second input= 1; (d) both inputs= 1.

(4) Denature the strands and run T_k through a gel, retaining only those strands of length $3l$. This may be achieved in a single step by using a denaturing gel [61].
(5) Add tube S_k to tube T_k. The strands in tube S_k anneal to the *second* restriction site embedded within each retained gate strand.
(6) Add enzyme *Pml*I to tube T_k, which "snips" off the z_k^j section (i.e., the output section) of each strand representing a retained gate.
(7) Denature and run T_k through another gel, this time retaining only strands of length l. This tube, T_k of retained strands forms the input to T_{k+1}. We now proceed to the simulation of level $k+1$.

At level $D(S)$ we carry out steps 1–7, as described above. However, at steps 4 and 7 we retain all strands of length $\geqslant 3l$. We are now ready to implement the final read-out phase. This involves a simple interpretation of the final gel visualisation. Since we know the unique length u_j of each $z_{D(S)}^j$ section of the strand for each output gate t_j, the presence or absence of a strand of length $u_j + 2l$ in the gel signifies that t_j has the value one or zero, respectively.

5.6. Analysis

We first analyse the model described above in terms of the feasibility of its biological implementation. During the course of the simulation, we use the following operations: primer annealing, ligation, restriction, and denaturing gel electrophoresis. As well as avoiding the need for PCR, the proposed implementation minimizes the degree of physical manipulation of tubes of DNA. This, in turn, minimizes potential problems such as strand shear and material loss due to strands sticking to the surface of tubes.

We now evaluate the model by describing how Batcher sorting networks [9] may be implemented within it. Batcher networks sort n inputs in $O(\log^2 n)$ stages. In [66], Wegener showed that if $n = 2^k$ then the number of *comparison* modules is $0.5n(\log n)(\log n - 1) + 2n - 2$. The circuit *depth* (again expressed in terms of the number of comparison modules) is $0.5(\log n)(\log n + 1)$. A comparison module has two (Boolean) inputs, x, y, and two outputs $MIN(x, y)$ (which is just $x\ AND\ y$); $MAX(x, y)$ (which is just $x\ OR\ y$).

Using NAND we can build a comparison module with five NAND gates and having depth 2 (the module is levelled, so since the Batcher network is levelled with respect to comparison modules, the whole realisation in NAND gates will be levelled). The NAND gate realisation is

$$\begin{aligned} &2 \text{ gates, depth } 2: && MIN(x, y) = NAND(NAND(x, y), NAND(x, y)),\\ &3 \text{ gates, depth } 2: && MAX(x, y) = NAND(NAND(x, x), NAND(y, y)). \end{aligned}$$

If we assume that $n = 2^k$, this gives the total size (in terms of number of gates) as $2.5(\log n)(\log n - 1) + 10n - 10$ and depth (in gates) as $(\log n)(\log n + 1)$.

Within the context of the *strong model* from [7] (the main feature of which being the restriction that pour operations are performed in a linear fashion rather than in parallel) an n-input Batcher network can therefore be simulated using $K(2.5(\log n)(\log n - 1) + 10n - 10)$ volume in $7(\log n)(\log n + 1)$ time, where K is a constant representing the number of copies of a single strand required to give reasonable guarantees of correct operation. The coefficient of 7 in the time figure represents the number of separate stages in a single level simulation.

Since Roweis et al. [54] claim that their *sticker model* is feasible using 2^{56} distinct strands, one may postulate that, in principle, the DNA implementation described above is technically feasible for input sizes that could not be physically realized *in silico* using existing fabrication techniques.

We end this section by noting that the first DNA-based simulation of Boolean circuits is described in [38], but this requires different resources to the implementation described here.

6. Forbidding–enforcing systems

In this section we discuss a model of molecular computing that is based on two kinds of "boundary conditions": *forbidding* and *enforcing*. Forbidding conditions require that

a conflicting group of molecules may not be present in a molecular system, as otherwise the system will "die" (e.g., will lose its functionality). An enforcing condition requires that if a certain group of molecules is present in a system, then eventually other molecules will be present in the system—in this way an enforcing condition models a molecular reaction. Thus the evolution of a system is determined by the enforcing conditions, but it is constraint by the forbidding conditions. *Forbidding–enforcing systems* (*fe systems*) may be considered in the context of various formalizations of the notion of molecule, but in this section we investigate them in the framework of strings—i.e., molecules are represented by strings.

Although we deal with (evolution of) strings, the model of forbidding–enforcing systems that we propose is *not* a grammatical model. It is based on the two types of boundary conditions rather than on rewriting by productions. *Forbidding conditions* are given as a family of forbidders, where each forbidder is a group of string patterns which *cannot occur together* in the system. *Enforcing conditions* are given as a family of enforcers, where each enforcer says that if a certain group of strings is present in the system, then some other strings will eventually be present in the system.

Then in a *forbidding–enforcing system*, *fe system* for short, which is specified by a set of forbidding conditions $\mathscr{F}$ and a set of enforcing conditions $\mathscr{E}$, the evolution of the system proceeds according to the molecular reactions specified by $\mathscr{E}$, *but* it is constrained by $\mathscr{F}$: the evolution cannot lead to any group of patterns specified by a forbidder from $\mathscr{F}$. In this way an fe system specifies a (possibly infinite) *family of languages* with each language obeying both $\mathscr{F}$ and $\mathscr{E}$. This is in sharp contrast to grammars considered in formal language theory, where each grammar specifies *one language*.

6.1. Forbidding sets

We move now to formalize our key notions of forbidding and enforcing conditions.

A *forbidding set* $\mathscr{F}$ (over an alphabet Σ) is a family of finite nonempty subsets of Σ^+; each element of a forbidding set is called a *forbidder*. Note that a forbidding set may be infinite—we only require that each forbidder is finite.

Consider now a forbidding set $\mathscr{F}$ and a language K. The way that $\mathscr{F}$ restricts K is defined as follows (for a language K, we use $sub(K)$ to denote the set of subwords of all words from K).

(1) Let $F \in \mathscr{F}$. We say that K is *consistent with* F, written K *con* F, iff $F \not\subseteq sub(K)$.
(2) We say that K is *consistent with* $\mathscr{F}$, written K *con* $\mathscr{F}$, iff K *con* F for each $F \in \mathscr{F}$.

Thus a forbidding set $\mathscr{F}$ (over Σ) defines the family of all languages (over Σ) that are consistent with $\mathscr{F}$—this family is denoted by $\mathscr{L}_\Sigma(\mathscr{F})$; we also write $\mathscr{L}(\mathscr{F})$ whenever Σ is understood from the context of considerations.

We say then that forbidding sets $\mathscr{F}_1, \mathscr{F}_2$ are *equivalent*, written $\mathscr{F}_1 \sim \mathscr{F}_2$, iff $\mathscr{L}(\mathscr{F}_1) = \mathscr{L}(\mathscr{F}_2)$.

We give now three basic properties of the consistency relation. We say that a sequence of languages $\sigma = K_0, K_1, \ldots$ is *ascending* if $K_i \subseteq K_{i+1}$ for all i, if σ is infinite, and $K_i \subseteq K_{i+1}$ for all $i < m$, if σ is finite and K_m is the last element of σ. Also $\bigcup \sigma$ denotes

the union of all languages in σ, and $|\sigma|$ denotes the *length* of σ: if $\sigma = K_0, K_1, \ldots, K_m$ for some $m \geqslant 0$, then $|\sigma| = m + 1$, and if σ is infinite, then $|\sigma|$ equals the cardinality of the set of natural numbers.

Theorem 6.1. *Let $\mathscr{F}$ be a forbidding set, and let K be a language such that K con $\mathscr{F}$.*
(1) *sub(K) con $\mathscr{F}$.*
(2) *If $K' \subseteq K$, then K' con $\mathscr{F}$.*
(3) *If $\sigma = K_1, K_2, \ldots$ is an ascending sequence of languages such that, for each $1 \leqslant i \leqslant |\sigma|$, K_i con $\mathscr{F}$, then $\bigcup \sigma$ con $\mathscr{F}$.*

As an example, consider $\Sigma = \{a, b\}$, and $\mathscr{F} = \{\{ab, ba\}, \{aa, bb\}\}$. Then for $K \subseteq \Sigma^+$, K *con* $\mathscr{F}$ iff $K \subseteq K_i$ for some $i \in \{1, 2, 3, 4\}$, where $K_1 = \{a\}\{b\}^* \cup \{b\}^+$, $K_2 = \{a\}^*\{b\} \cup \{a\}^+$, $K_3 = \{b\}\{a\}^* \cup \{a\}^+$, and $K_4 = \{b\}^*\{a\} \cup \{b\}^+$.

Enforcing conditions are formalized through the notion of enforcing set.

An *enforcing set* $\mathscr{E}$ (over an alphabet Σ) is a family of ordered pairs (X, Y) such that all $X, Y \subseteq \Sigma^+$, X, Y are finite, and each $Y \neq \emptyset$; each element of an enforcing set is called an *enforcer*.

Consider now an enforcing set $\mathscr{E}$ and a language K. The way that $\mathscr{E}$ restricts K is defined as follows.
(1) Let $E = (X, Y) \in \mathscr{E}$. We say that E is *applicable* to K (or that E is *K-applicable*), written E *app* K, iff $X \subseteq K$. Then we say that K *satisfies* E, written K *sat* E, iff E *app* K implies $Y \cap K \neq \emptyset$. If E *app* K but $Y \cap K = \emptyset$, then E is a *K-violator*.
(2) We say that K satisfies $\mathscr{E}$, written K *sat* $\mathscr{E}$, iff K *sat* E for each $E \in \mathscr{E}$.

Thus an enforcing set $\mathscr{E}$ (over Σ) defines the family of all languages (over Σ) that satisfy $\mathscr{E}$—this family is denoted by $\mathscr{L}_\Sigma(\mathscr{E})$; we also write $\mathscr{L}(\mathscr{E})$ whenever Σ is understood from the context of considerations. We say that enforcing sets $\mathscr{E}_1$ and $\mathscr{E}_2$ are equivalent, written $\mathscr{E}_1 \sim \mathscr{E}_2$, iff $\mathscr{L}(\mathscr{E}_1) = \mathscr{L}(\mathscr{E}_2)$.

As an example, consider $\Sigma = \{a, b\}$ and $\mathscr{E} = \{(X, Y) \mid X = \{u\}, Y = \{uu\}$ with $u \in \Sigma^+\}$. Then for $K \subseteq \Sigma^+$, if K *sat* $\mathscr{E}$, then K is closed under the square operation: for each $u \in K$ also $uu \in K$.

The intuition behind an enforcing set $\mathscr{E}$ is that if a molecular system satisfies $\mathscr{E}$ and some molecules are in the system, then eventually some other molecules will be present in the system. This evolving through enforcing is formalized in our string framework as follows.

For an enforcing set $\mathscr{E}$ and languages K_1, K_2 we say that K_2 is an *$\mathscr{E}$-extension* of K_1, written $K_1 \vdash_{\mathscr{E}} K_2$, iff, for each $(X, Y) \in \mathscr{E}$, $X \subseteq K_1$ implies $K_2 \cap Y \neq \emptyset$. Note that it follows directly from this definition that
(1) if K *sat* $\mathscr{E}$, then $K \vdash_{\mathscr{E}} K$, and
(2) if $K_1 \vdash_{\mathscr{E}} K_2$, $K_2 \vdash_{\mathscr{E}} K_3, K_1 \subseteq K_2$ and $K_2 \subseteq K_3$, then $K_1 \vdash_{\mathscr{E}} K_3$.

The basic property of the extension relation is given by the following result.

Theorem 6.2. *Let $\mathscr{E}$ be an enforcing set and let $\sigma = K_1, K_2, \ldots$ be an infinite ascending sequence of languages. If, for each $i \geqslant 1$ we have $K_i \vdash_{\mathscr{E}} K_{i+1}$, then $\bigcup \sigma$ sat $\mathscr{E}$.*

6.2. Finiteness conditions

The issue of finiteness is always important from both the theoretical and the "real world" applications point of view. In this section we will consider this issue for enforcing sets.

We will use the following notation. For a finite language Z, $\mathscr{E}(Z) = \{(X, Y) \in \mathscr{E} \mid X = Z\}$, and if $\mathscr{E}(Z) \neq \emptyset$, then we say that Z is *relevant for* $\mathscr{E}$.

We are ready now to formulate two notions of finiteness for enforcing sets.

(1) An enforcing set $\mathscr{E}$ is *finitary*, iff, for each finite language Z, $\mathscr{E}(Z)$ is finite.
(2) An enforcing set $\mathscr{E}$ is *weakly finitary*, iff, for each finite language K_1 there exists a finite language K_2 such that $K_1 \vdash_{\mathscr{E}} K_2$.

Being finitary is a *syntactic* property, quite convenient if we have to either specify or analyze $\mathscr{E}(Z)$ for some Z relevant for $\mathscr{E}$. Being "weakly finitary" is a *semantic* property—it says that if $\mathscr{E}$ is weakly finitary, then each finite set can evolve (according to $\mathscr{E}$) into a finite set. The basic relationship between these properties is given by the following result.

Theorem 6.3. (1) *Every finitary enforcing set $\mathscr{E}$ is weakly finitary.*
(2) *There exist weakly finitary enforcing sets that are not finitary.*

It turns out that, up to equivalence $\sim$, one can consider only finitary enforcing sets, as expressed by the following result.

Theorem 6.4. *For every enforcing set $\mathscr{E}$ there exists a finitary enforcing set $\mathscr{E}'$ such that $\mathscr{E} \sim \mathscr{E}'$.*

6.3. Forbidding–enforcing systems

We combine now forbidding and enforcing, and define forbidding–enforcing systems.

A *forbidding–enforcing system*, *fe system* for short, is a 3-tuple $\Gamma = (\Sigma, \mathscr{F}, \mathscr{E})$, where Σ is an alphabet, $\mathscr{F}$ is a forbidding set over Σ, and $\mathscr{E}$ is an enforcing set over Σ.

The language family of an fe system is defined by combining the evolution by enforcing and the "protection" by forbidding.

A language $K \subseteq \Sigma^*$ *obeys* an fe system $\Gamma = (\Sigma, \mathscr{F}, \mathscr{E})$, written K *obs* Γ, iff K *con* $\mathscr{F}$ and K *sat* $\mathscr{E}$. Then $\mathscr{L}(\Gamma)$ denotes the family of all languages obeying Γ—it is referred to as an *fe family*.

The finitary restriction on enforcing carries over to fe systems as follows: an fe system $\Gamma = (\Sigma, \mathscr{F}, \mathscr{E})$ is *finitary* if $\mathscr{E}$ is finitary.

We refer the reader to [25,64] for examples of (1) using fe systems for the description of the structure of DNA molecules, and (2) for examples of translations of computational problems, such as the Hamiltonian Path Problem and Satisfiability Problem, into fe systems.

Forbidding–enforcing systems are highly combinatorial—both the forbidding and the enforcing set may be of arbitrary cardinality, they may be also infinite. This implies that the analysis of fe systems may be very involved—one has to evaluate the effect

of all enforcers that have to be applied in such a way that they do not violate the constraints set up to forbidders. Therefore an important research topic is to look for a structure behind all the computations (evolutions) in an fe system. It turns out that such a structure exists for finitary fe systems: all computations and their effects (languages) can be elegantly represented by trees.

We consider here directed node-labeled trees with no order between direct descendants of the same node. Also, a *complete* path begins always in the root and if it is finite, then it ends in a leaf. We associate a tree with an fe system as follows.

We are especially interested in complete Γ-trees.

Let $\Gamma = (\Sigma, \mathscr{F}, \mathscr{E})$ be an fe system. A Γ-tree τ is a *complete* Γ-tree iff, for every $K \in \mathscr{L}(\Gamma)$, we have:

(1) If K is finite, then K is a node label in τ.

(2) If K is infinite, then there exists a complete path π in τ such that $K = \bigcup_{v \in \pi} lab_\tau(v)$.

The following result gives a tree representation of all computations and languages of a *finitary* fe system.

Theorem 6.5. *For each finitary fe system Γ there exists a complete Γ-tree.*

The restriction to finitary fe systems is very essential—the above theorem does not hold for arbitrary fe systems! This supports our view that one should not consider arbitrary fe systems, but rather restrict oneself to finitary fe systems, where computations "happen gradually". They evolve rather than deliver in one step a whole infinite language.

On the other hand, because every enforcing set is equivalent to a finitary enforcing set (Theorem 6.5) complete Γ-trees represent all families of fe languages. This is expressed by the following result.

Theorem 6.6. *For each fe family $\mathscr{K}$ there exists a finitary branching tree τ with nodes labeled by finite languages such that a language $K \in \mathscr{K}$ iff there exists an infinite complete path π such that K is the union of all languages that label the nodes on π.*

This section is based on [25]—the fe systems were introduced in this paper. Then the basic theory of fe systems was further developed in [22,23,64]; a good account on the theory of fe systems is given in [64].

7. DNA computing by splicing: H systems

In this section we discuss one of the most investigated models of DNA computing, called *splicing systems* or H systems. They were introduced by Head (that's why "H" stands for) in 1987 [29], thus seven years *before* Adleman's experiment. The original motivation behind H systems was to model the way that restriction enzymes process DNA molecules. More specifically, *splicing* of two DNA molecules is achieved by two

operations: cutting the molecules by restriction enzymes and pasting together molecules obtained in this way, providing that they have matching sticky ends (see Section 2.6).

For example, consider the following two (double stranded) DNA molecules:

$$5' - \text{CCCCCTCGACCCCC} - 3'$$
$$3' - \text{GGGGGAGCTGGGGG} - 5'$$

and

$$5' - \text{AAAAAGCGCAAAAA} - 3'$$
$$3' - \text{TTTTTCGCGTTTTT} - 5'$$

and the restriction enzymes *Taq*I and *Sci*NI, for which the recognition sites are

$$\begin{array}{l} T\,\big|\,C\ G\ A \\ A\ G\ C\,\big|\,T \end{array} \quad \text{and} \quad \begin{array}{l} G\,\big|\,C\ G\ C \\ C\ G\ C\,\big|\,G \end{array}$$

respectively (we have also indicated the cuts that these enzymes make within their recognition sites).

The restriction enzymes *Taq*I and *Sci*NI will cut the above two molecules producing the following four molecules:

$$\begin{array}{l} 5' - \text{CCCCCT} \quad \text{CGACCCCC} - 3' \\ 3' - \text{GGGGGAGC}, \quad \text{TGGGGG} - 5' \\ 5' - \text{AAAAAG} \quad \text{CGCAAAAA} - 3' \\ 3' - \text{TTTTTCGC}, \quad \text{GTTTTT} - 5' \end{array}$$

Because the first of these molecules has a sticky end complementary to the sticky end of the second and the fourth molecule, and the third of these molecules also has a sticky end complementary to the sticky end of the second and the fourth molecule, the annealing of sticky ends followed by ligation will either reproduce the two original molecules, or the following two new molecules will be formed:

$$\begin{array}{l} 5' - \text{CCCCCTCGCAAAAA} - 3' \\ 3' - \text{GGGGGAGCGTTTTT} - 5' \end{array}$$

$$\begin{array}{l} 5' - \text{AAAAAGCGACCCCC} - 3' \\ 3' - \text{TTTTTCGCTGGGGG} - 5' \end{array}$$

7.1. The formal splicing operation

The abstraction of the bio-operation sketched above leads to the mathematical operation of splicing through the following reasoning steps (for more detailed discussion see [29,48]):

- Due to the Watson–Crick complementarity, each strand of a double stranded DNA molecule uniquely identifies the other strand—thus one can consider single strands, and consequently strings denoting them.

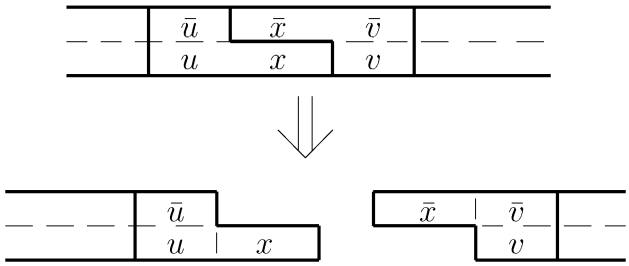

Fig. 11. Cutting by a restriction enzyme.

- Given such a string, the site that is recognized by a restriction enzyme is identified by a triplet of strings (u,x,v), where u is "the left context", v is "the right context", and x is the sticky end. This is illustrated in Fig. 11; the strings $\bar{u},\bar{x},\bar{v}$ are the Watson–Crick complements of u,x,v, respectively.
- The splicing of two strings $w_1u_1xv_1w_2$ and $z_1u_2xv_2z_2$ (with the restriction sites (u_1,x,v_1) and (u_2,x,v_2) as identified) produces the strings $w_1u_1xv_2z_2$ and $z_1u_2xv_1w_2$. The same result is obtained if we identify restriction sites by the pairs (u_1x,v_1) and (u_2x,v_2), cut the strings as indicated by these pairs, and then recombine the so-obtained "halves" of the original strings.
- In a general set-up, one can consider an arbitrary alphabet rather than the specific four letter alphabet of nucleotides.
- Finally, from a mathematical point of view, it suffices to consider only one of the two strings obtained by such splicing, because the other string is obtained by a "mirror" operation (using the symmetric "splicing rule").

In this way, we arrive at the following general and elegant formal splicing operation.

Consider an alphabet V and two special symbols, # and \$ which are not in V. A *splicing rule* (over V) is a string of the form $r=u_1\#u_2\$u_3\#u_4$, where $u_1,u_2,u_3,u_4\in V^*$. Given such a rule r, and $x,y,z\in V^*$, we write

$$(x,y)\vdash_r z \text{ iff } x=x_1u_1u_2x_2,\ y=y_1u_3u_4y_2,\ z=x_1u_1u_4y_2,$$
$$\text{for some } x_1,x_2,y_1,y_2\in V^*.$$

We say that x,y are *spliced* at the *sites* u_1u_2 and u_3u_4, respectively, yielding z. We may omit the specification of r, and write $\vdash$ instead of $\vdash_r$, whenever r is understood from the context of consideration.

The splicing operation on strings is extended to languages as follows.

A pair $\sigma=(V,R)$, where V is an alphabet, and $R\subseteq V^*\#V^*\$V^*\#V^*$ is a set of splicing rules, is called *an H scheme*. Note that R can be infinite, and moreover *it is itself a set of strings*, hence a language. Thus, we can consider its complexity, e.g., its place in the Chomsky hierarchy, or in any other classification of languages. In general, if $R\in FL$, for a given family FL of languages, then we say that the H scheme σ is *of the FL type.*

Now, for a given H scheme $\sigma = (V, R)$ and a language $L \subseteq V^*$, we define

$$\sigma(L) = \{z \in V^* \mid (x, y) \vdash_r z, \text{ for some } x, y \in L, \ r \in R\},$$
$$\sigma^0(L) = L,$$
$$\sigma^{i+1}(L) = \sigma^i(L) \cup \sigma(\sigma^i(L)), \quad \text{for } i \geqslant 0, \text{ and}$$
$$\sigma^*(L) = \bigcup_{i \geqslant 0} \sigma^i(L).$$

Thus, $\sigma(L)$ is the language obtained by a one-step splicing of any pair of strings from L with respect to any rule from R. Then, $\sigma^*(L)$ is the closure of L under the splicing (using the rules) in σ, i.e., the smallest language which contains L, and is closed under the splicing using the rules from σ.

Let us consider a simple example. Let L be the singleton language $\{abba\}$ over the alphabet $V = \{a, b\}$, and let R be the set of two splicing rules:

$$\{r_1 : a\#b\$b\#ba, r_2 : b\#a\$b\#ba\}.$$

By using the first rule, we get

$$(a|bba, ab|ba) \vdash_{r_1} aba.$$

(By vertical bars we have indicated the place where the two strings are cut.) By splicing the strings $aba, abba$ using rule r_1 we get the same string aba. However, by using iteratively the second rule, we can obtain strings of an increasing length:

$$(ab|a, ab|ba) \vdash_{r_2} abba,$$
$$(abb|a, ab|ba) \vdash_{r_2} abbba$$

and, in general,

$$(ab^n|a, ab|ba) \vdash_{r_2} ab^{n+1}a, \quad \text{for all } n \geqslant 1.$$

Therefore, for the splicing scheme $\sigma = (V, R)$ and L, we have

$$\sigma(L) = \{aba, abbba\},$$
$$\sigma^0(L) = \{abba\},$$
$$\sigma^1(L) = \{aba, abba, abbba\}, \text{ and}$$
$$\sigma^i(L) = \{ab^n a \mid 1 \leqslant n \leqslant i + 2\}, \quad \text{for } i \geqslant 1.$$

Consequently,

$$\sigma^*(L) = \{ab^n a \mid n \geqslant 1\}.$$

7.2. H systems

We are now ready to define a "computing system" based on splicing. We consider it in the general form, as introduced in [47].

Table 1
The generative power of extended H systems

	FIN	*REG*	*LIN*	*CF*	*CS*	*RE*
FIN	*REG*	*RE*	*RE*	*RE*	*RE*	*RE*
REG	*REG*	*RE*	*RE*	*RE*	*RE*	*RE*
LIN	*LIN, CF*	*RE*	*RE*	*RE*	*RE*	*RE*
CF	*CF*	*RE*	*RE*	*RE*	*RE*	*RE*
CS	*RE*	*RE*	*RE*	*RE*	*RE*	*RE*
RE	*RE*	*RE*	*RE*	*RE*	*RE*	*RE*

An *extended H system* is a quadruple $\gamma = (V, T, A, R)$, where V is an alphabet, $T \subseteq V$, $A \subseteq V^*$, and $R \subseteq V^*\#V^*\$V^*\#V^*$, where $\#, \$$ are special symbols not in V.

We call V the *alphabet* of γ, T its *terminal* alphabet, A its set of *axioms*, and R its set of splicing rules.

The *language generated* by γ is defined by $L(\gamma) = \sigma^*(A) \cap T^*$, where $\sigma = (V, R)$ is the *underlying H scheme* of γ.

For two families of languages, FL_1, FL_2, we denote by $EH(FL_1, FL_2)$ the family of languages $L(\gamma)$ generated by extended H systems $\gamma = (V, T, A, R)$ such that $A \in FL_1$ and $R \in FL_2$.

Two fundamental results on H systems are:

Lemma 7.1 (The Regularity Lemma). $EH(REG, FIN) = REG$.

Lemma 7.2 (The Basic Universality Lemma). $EH(FIN, REG) = RE$.

Thus, by Lemma 7.1, using finite sets of splicing rules starting from regular languages, we get regular languages only. If we allow to use regular rather than only finite sets of splicing rules, then, by Lemma 7.2, we get the "maximal gain": all recursively enumerable languages.

The inclusion $\subseteq$ from Lemma 7.1 was first proved in [17]; a direct finite automata proof is given in [49] (see also [48]); Lemma 7.2 was proved in [44]. A result much more general than Lemma 7.1 is given in [50]: each full AFL is closed under iterated splicing with respect to a finite H scheme.

A systematic consideration of all pairs (FL_1, FL_2) taken from the Chomsky hierarchy yields the results from Table 1, where the family $EH(FL_1, FL_2)$ is placed at the intersection of the row labeled with FL_1 with the column labeled with FL_2, except for the case of $FL_1 = LIN$, where we give the pair of families LIN, CF, because $LIN \subset EH(LIN, FIN) \subseteq CF$.

Thus, the only class which does not yield a family within the Chomsky hierarchy is $EH(LIN, FIN)$. Also, note that the family of context-free languages has no nontrivial characterization in terms of H systems, and the family CS of context-sensitive languages does not appear at all in the table.

7.3. H systems with controlled splicing

From a computational point of view, Lemmas 7.1 and 7.2 are quite "frustrating": finite H systems compute only at the level of finite automata, while the computational universality of Lemma 7.2 is obtained by using an *infinite* set of splicing rules. Fortunately, the proof of the Basic Universality Lemma indicates a number of ways of overcoming this drawback.

The proof of Lemma 7.2 goes as follows. Starting from a type-0 Chomsky grammar G, one constructs an equivalent extended H system γ, whose sentential forms are circularly permuted versions of the sentential forms of G, and the simulation of the rules of G takes place within suffixes of the sentential forms of γ (the circular permutation ensures that each derivation step in G can be simulated in this way). Very crucial for this "rotate-and-simulate" procedure are the first and the last symbols of each sentential form, which in fact are markers, holding the information about the current stage of the simulation. That is, we can ignore (almost completely) the strings we splice as long as we know their first and last symbols, and the splicing sites. In other words, it is sufficient to have a finite number of splicing rules, and to associate with each rule certain "promoters", which are symbols whose presence allows the splicing of a given string.

This observation leads to *extended H systems with permitting contexts*, whose rules have associated finite sets of symbols such that a rule is applicable only to strings which contain the associated symbols. A formal definition can be found in [15], where these systems were introduced, and in a series of subsequent papers; [48] is a good comprehensive reference. We also refer the reader to [48] for other classes of H systems with controlled splicing. There are about a dozen such systems, in general imitating the types of controls known from the "classic" regulated rewriting area in formal language theory (see, e.g., [19]). In particular, the following controls were investigated: *forbidding contexts* (symbols are associated with rules and a string cannot be spliced if it contains such a symbol), *target languages* (the splicing of two strings is allowed only if the resulting string belongs to a given regular language which is associated with the rule or associated with the whole set of rules; in the former case we say that we have *local* targets, and in the latter case we have a *global* target), *programmed control* (a *next* mapping is given on the set of rules, which indicates the sequencing of rules), *evolving sets of rules* (at each step, a different set of rules is produced, by point mutation rules which act on the splicing rules themselves), *double splicing* (the strings resulting from the splicing of two strings are immediately spliced again by any available rule—note that in this case the splicing produces two output strings, both strings obtained by recombination). *In all these cases one gets characterizations of recursively enumerable languages.*

7.4. Distributed H systems

One way to increase the power of H systems with finite sets of axioms and rules is to organize their computations in a distributed way. This idea comes from grammar systems (see [14]), but it is also natural in the context of DNA computing from both

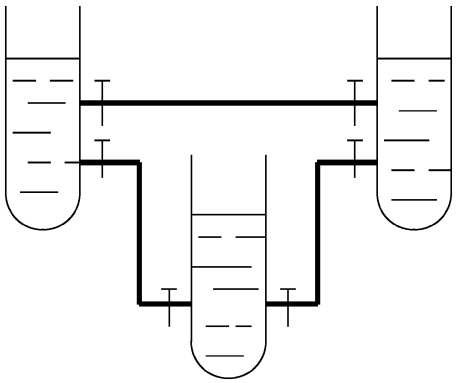

Fig. 12. A test tube system.

the mathematical point of view (one gets indeed the universality in this way) and the experimental point of view (the biochemical process is distributed over several test tubes which together implement specific computations).

Up to now, five grammar-system-like models were considered for H systems. The most investigated class of distributed H systems are the so-called *test tube systems*, introduced in [16]: several H systems ("tubes"), having their own axioms and splicing rules, working simultaneously, and redistributing among themselves the results of splicing. Such a redistribution takes place all the time (i.e., after each derivation step), but it is a subject of the *filtering* restriction: there is a filtering alphabet associated with each component, and each component admits only those strings (produced by the other components) that are over its filtering alphabet. After each derivation step, the only strings that remain within this component are those strings that cannot be redistributed to other components. To define the language of a test tube system one defines the terminal alphabet of the system, and designates one of the test tubes as the language defining tube. Then, the language of the system is the set of all strings over the terminal alphabet that reside in this tube at any moment during the string generation process. Fig. 12 illustrates the structure of a test tube system.

We denote by TTH_n the family of languages generated by test tube systems with at most n components, $n \geqslant 1$.

A characterization of RE using test tube systems was obtained in [16], however without providing a bound on the number of components. Such bounds were given later in several papers (see [69,26,40]); the currently best result is given in [51]: $TTH_3 = RE$. It is an *open problem* whether or not this result can be strengthened. (We *conjecture* that $TTH_2 \subset CF$.)

Another well-investigated class is that of *time-varying H systems*. It can be viewed as a sequential counterpart of the test tube systems: at different moments we use different sets of splicing rules; the transition from one set of rules to another is specified by a control cycle. Thus, this model corresponds both to periodically time-varying

Table 2
The generative power of distributed H systems

Type of systems	Components	Rules per component
Test tube systems	3	1
Time-varying H systems	1	3
Two-level H systems	3	?
Cooperating distributed H systems	3	3
Splicing grammar systems	2	2

grammars in regulated rewriting area and to controlled tabled Lindenmayer systems (see, e.g., [19,55]).

We denote by TVH_n, $n \geqslant 1$, the family of languages generated by time-varying distributed H systems of degree at most n.

It was proved in [43] (where time varying distributed systems were introduced) that the family of languages generated by time varying systems is equal to RE. Moreover, it has been proved there that $RE = TVH_7$—this result was improved in [53,31,32], and finally it has been proved in [33] that $TVH_1 = RE$. One component suffices, because in a single transition step of a computation the only strings that survive are those that are produced by splicing in this step—the "old strings" are filtered out.

As mentioned above, five types of distributed H systems were investigated in the literature. Two of them were discussed above, and the other three are the *two-level H systems* [41,42], *sequentially cooperating distributed H systems* [34], and *splicing parallel grammar systems* [18]. Also for these systems, characterizations of recursively enumerable languages are obtained, and they use a small number of components.

As distributed splicing systems can be considered also the membrane systems with string-objects which evolve by splicing operations—see [45], as well as [46].

Also relevant for determining the complexity of distributed H systems is the number of rules per component of a system—one would like to have distributed H systems with components having the number of rules as small as possible. The number of components and the number of rules per component needed for generating the recursively enumerable languages by various classes of distributed H systems is given in Table 2 (proofs for most of these results are given in [48]).

All characterizations of RE discussed in this section can be considered as theoretical proofs of the possibility to devise DNA "computers" based on splicing which are *programmable*: the corresponding classes of H systems have universality properties, that is, fixed universal devices exist which can simulate any given device as soon as a "code" of the simulated device is given (as an axiom) to the universal one.

8. Discussion

As we have indicated already, the topics discussed in this paper cover only a small part of developments in the theory of DNA computing. However, other pa-

pers on molecular computing in this special issue cover the topics of strand design, complexity analysis and membrane systems. These papers together with our paper should give then the reader a good insight into research in (the theory of) molecular computing.

The best source of information on developments in (also the theory of) DNA computing are the Proceedings of the Annual International Workshop on DNA-Based Computers. From the 6th Workshop on, the proceedings appear in the *Lecture Notes in Computer Science* (LNCS) by Springer (the Proceedings of the 6th Workshop [13] appeared as vol. 2054). The proceedings of the first five workshops (which started in 1995), with the exception of the 4th Workshop, were published by the American Mathematical Society in the DIMACS Series in Discrete Mathematics and Theoretical Computer Science (vols. 27, 48, 52, and 54). The Proceedings of the 4th Workshop [30] appeared as an issue of the journal *BioSystems*.

Also, [48] is a book on DNA computing focused on theory—it covers a lot of early models. Some of the models not covered in [48], most notably the self-assembly and the membrane systems, are discussed in [12] which is devoted to the theory of molecular and quantum computing.

Acknowledgements

The second and the third author gratefully acknowledge the support of the ESPRIT Working Group APPLIGRAPH.

References

[1] R.L.P. Adams, J.T. Knowler, D.P. Leader, The Biochemistry of the Nucleic Acids, 10th Edition, Chapman & Hall, London, 1986.

[2] L.M. Adleman, Molecular computation of solutions to combinatorial problems, Science 226 (1994) 1021–1024.

[3] L.M. Adleman, P.W.K. Rothemund, S. Roweiss, E. Winfree, On applying molecular computations to the data encryption standard, in: E. Baum, D. Boneh, P. Kaplan, R. Lipton, J. Reif, N. Seeman (Eds.), DNA Based Computers, Proc. 2nd Annual Meeting, Princeton, 1996, pp. 28–48.

[4] M. Amos, DNA computation, Ph.D. thesis, Department of Computer Science, University of Warwick, UK, September 1997.

[5] M. Amos, P.E. Dunne, DNA simulation of Boolean circuits, Technical Report CTAG-97009, Department of Computer Science, University of Liverpool, December 1997.

[6] M. Amos, P.E. Dunne, A. Gibbons, DNA simulation of Boolean circuits, in: J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, R. Riolo (Eds.), Genetic Programming, Proc. 3rd Annual Conference, Morgan Kaufmann, Los Altos, CA, 1998, pp. 679–683.

[7] M. Amos, A. Gibbons, P.E. Dunne, The complexity and viability of DNA computations, in: Lundh, Olsson, Narayanan (Eds.), Proc. 1st International Conference on Bio-Computing and Emergent Computation, University of Skövde, Sweden, World Scientific, Singapore, 1997, pp. 165–173.

[8] M. Amos, S. Wilson, D.A. Hodgson, G. Owenson, A. Gibbons, Practical implementation of DNA computations, in: C.S. Calude, J. Casti, M.J. Dinneen (Eds.), Unconventional Models of Computation, Springer, Singapore, 1998, pp. 1–18.

[9] K.E. Batcher, Sorting networks and their applications, Proc. American Federation of Information Processing Societies 1968 Spring Joint Computer Conference, Vol. 32, Thompson Book Co, Washington, DC, 1968, pp. 307–314.
[10] K.J. Breslauer, R. Frank, H. Blocker, L.A. Marky, Predicting DNA duplex stability from the base sequence, Proc. Natl. Acad. Sci. (1986) 3746–3750.
[11] T.A. Brown, Genetics: A Molecular Approach, Chapman & Hall, London, 1993.
[12] C. Calude, Gh. Păun, Computing with Cells and Atoms, Taylor & Francis, London, 2000.
[13] A. Condon, G. Rozenberg (Eds.), DNA Computing, Proc. 6th Internat. Workshop on DNA-Based Computers, DNA 2000, Leiden, The Netherlands, Lecture Notes in Computer Science, Vol. 2054, Springer, Berlin, 2000.
[14] E. Csuhaj-Varju, J. Dassow, J. Kelemen, Gh. Păun, Grammar Systems. A Grammatical Approach to Distribution and Cooperation, Gordon and Breach, London, 1994.
[15] E. Csuhaj-Varju, L. Freund, L. Kari, Gh. Păun, DNA computing based on splicing: universality results, in: L. Hunter, T.E. Klein (Eds.), Proc. 1st Annu. Pacific Symp. on Biocomputing, Hawaii, January 1996, World Scientific Publishing, Singapore, 1996, pp. 179–190.
[16] E. Csuhaj-Varju, L. Kari, Gh. Păun, Test tube distributed systems based on splicing, Comput. Artif. Intell. 15 (2–3) (1996) 211–231.
[17] K. Culik II, T. Harju, Splicing semigroups of dominoes and DNA, Discrete Appl. Math. 31 (1991) 261–277.
[18] J. Dassow, V. Mitrana, Splicing grammar systems, Comput. Artif. Intell. 15 (2–3) (1996) 109–122.
[19] J. Dassow, Gh. Păun, Regulated Rewriting in Formal Language Theory, Springer, Berlin, 1989.
[20] K. Drlica, Understanding DNA and Gene Cloning. A Guide for the CURIOUS, Wiley, New York, 1992.
[21] P.E. Dunne, The Complexity of Boolean Networks, Academic Press, New York, 1988.
[22] A. Ehrenfeucht, H.J. Hoogeboom, G. Rozenberg, N. van Vugt, Forbidding and enforcing, in: E. Winfree, D.K. Gifford (Eds.), DNA Based Computers V, DIMACS Series in Discrete Mathematics, Vol. 54 (1999 proceedings), Massachusetts Institute of Technology, USA, 2000.
[23] A. Ehrenfeucht, H.J. Hoogeboom, G. Rozenberg, N. van Vugt, Sequences of languages in forbidding–enforcing families, Soft Comput. 5 (2) (2001) 121–125.
[24] J. Engelfriet, G. Rozenberg, Fixed-point languages, equality languages, and representations of recursively enumerable languages, J. Assoc. Comput. Mach. 27 (1980) 499–518.
[25] A. Ehrenfeucht, G. Rozenberg, Forbidding-enforcing systems, Theoret. Comput. Sci., to appear.
[26] C. Ferretti, G. Mauri, C. Zandron, Nine test tubes generate any RE language, Theoret. Comput. Sci. 231 (2) (2000) 171–180.
[27] A. Gehani, T.H. LaBean, J.H. Reif, DNA-based cryptography, manuscript, 2000.
[28] M.A. Harrison, Introduction to Switching and Automata Theory, McGraw-Hill, New York, 1965.
[29] T. Head, Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors, Bull. Math. Biol. 49 (1987) 737–759.
[30] L. Kari, H. Rubin, D.H. Wood (Eds.), Proc. 4th Internat. Meeting on DNA-Based Computers, Pennsylvania University, June 1998, BioSystems 52 (1999).
[31] M. Margenstern, Y. Rogozhin, A universal time-varying distributed H systems of degree 2, Proc. 4th Internat. Meeting on DNA Based Computers, Philadelphia, June 1998.
[32] M. Margenstern, Y. Rogozhin, Time-varying distributed H-systems of degree 2 generate all RE languages, in: C. Martin-Vide, V. Mitrana (Eds.), Where Mathematics, Computer Science, Linguistics, and Biology Meet, Kluwer, Dordrecht, Boston, London, 2001, pp. 399–407.
[33] M. Margenstern, Y. Rogozhin, Time-varying distributed H systems of degree 1 generate all recursively enumerable languages, in: M. Ito, Gh. Păun, S. Yu (Eds.), Words, Semigroups, and Transductions, World Scientific Publishing, Singapore, 2001, pp. 329–340.
[34] C. Martin-Vide, Gh. Păun, Cooperating distributed splicing systems, J. Automata, Languages, Combin. 4 (1) (1999) 3–16.
[35] C. Mead, L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980.
[36] K.B. Mullis, The unusual origin of the polymerase chain reaction, Scientific American 262 (1990) 36–43.
[37] K.B. Mullis, F. Ferré, R.A. Gibbs (Eds.), The Polymerase Chain Reaction, Birkhauser, Basel, 1994.

[38] M. Ogihara, A. Ray, Simulating Boolean circuits on a DNA computer, Technical Report 631, University of Rochester, August 1996.
[39] R. Old, S. Primrose, Principles of Gene Manipulation: An Introduction to Genetic Engineering, 5th ed., Blackwell Scientific Publications, Oxford, 1994.
[40] Gh. Păun, Regular extended H systems are computationally universal, J. Automata, Languages, Combin. 1 (1) (1996) 27–36.
[41] Gh. Păun, DNA computing, distributed splicing systems, in: J. Mycielski, G. Rozenberg, A. Salomaa (Eds.), Structures in Logic and Computer Science. A Selection of Essays in Honor of A. Ehrenfeucht, Lecture Notes in Computer Science, Vol. 1261, Springer, Berlin, 1997, pp. 353–370.
[42] Gh. Păun, Two-level distributed H systems, in: S. Bozapalidis (Ed.), Proc. Developments in Language Theory Conf., Vol. III,Aristotle University of Thessaloniki, Thessaloniki, 1997, pp. 309–327.
[43] Gh. Păun, Distributed architectures in DNA computing based on splicing: limiting the size of components, in: C.S. Calude, J. Casti, M.J. Dinneen (Eds.), Unconventional Models of Computation, Springer, Singapore, 1998, pp. 323–335.
[44] Gh. Păun, On time-varying H systems, Bull. EATCS 67 (1999) 157–164.
[45] Gh. Păun, DNA computing based on splicing: universality results, Theoret. Comput. Sci. 231 (2) (2000) 275–296.
[46] Gh. Păun, Computing with membranes, J. Comput. System Sci. 61 (1) (2000) 108–143.
[47] Gh. Păun, G. Rozenberg, A guide to membrane computing, Theoret. Comput. Sci., this volume.
[48] Gh. Păun, G. Rozenberg, A. Salomaa, Computing by splicing, Theoret. Comput. Sci. 168 (2) (1996) 321–336.
[49] Gh. Păun, G. Rozenberg, A. Salomaa, DNA Computing: New Computing Paradigms, Springer, Berlin, 1998.
[50] D. Pixton, Regularity of splicing languages, Discrete Appl. Math. 69 (1996) 101–124.
[51] D. Pixton, Splicing in abstract families of languages, Theoret. Comput. Sci. 234 (1–2) (2000) 135–166.
[52] L. Priese, Y. Rogozhin, M. Margenstern, Finite H systems with 3 tubes are not predictable, in: R.B. Altman, A.K. Dunker, L. Hunter, T.E. Klein (Eds.), Pacific Symp. on Biocomputing, World Scientific, Singapore, 1998, pp. 547–558.
[53] J.M. Robertson (Ed.), The Philosophical Works of Francis Bacon, George Rutledge and Sons, London, 1905.
[54] S. Roweis, E. Winfree, R. Burgoyne, N. Chelyapov, M. Goodman, P. Rothemund, L. Adleman, A sticker based architecture for DNA computation, Proc. 2nd Annual Meeting on DNA Based Computers, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, June 1996.
[55] G. Rozenberg, A. Salomaa, The Mathematical Theory of L Systems, Academic Press, New York, 1980.
[56] G. Rozenberg, A. Salomaa, Watson–Crick complementarity universal computations and genetic engineering, Computer Science Technical Report 28, Leiden University, 1996.
[57] G. Rozenberg, A. Salomaa, DNA Computing: New Ideas and Paradigms, Lecture Notes Computer Science, Vol. 1644, Springer, Berlin, 1999, pp. 106–118.
[58] A. Salomaa, Jewels of Formal Language Theory, Computer Science Press, Rockville, MD, 1981.
[59] A. Salomaa, Public-Key Cryptography, Springer, Berlin, 1996.
[60] A. Salomaa, Turing, Watson–Crick and Lindenmayer. Aspects of DNA complementarity, in: C.S. Calude, J. Casti, M.J. Dinneen (Eds.), Unconventional Models of Computation, Springer, Singapore, 1998, pp. 94–107.
[61] J. Sambrook, E.F. Fritsch, T. Maniatis, Molecular Cloning: A Laboratory Manual, 2nd ed., Cold Spring Harbor Press, New York, 1989.
[62] B. Schneier, Applied Cryptography, Wiley, New York, 1996.
[63] H.M. Sheffer, A set of five independent postulates for Boolean algebras, with application to logical constants, Trans. Amer. Math. Soc. 14 (1913) 481–488.
[64] N. van Vugt, Models of molecular computing, Ph.D. Thesis, Leiden Institute of Advanced Computer Science, Leiden University, 2002.
[65] J.D. Watson, M. Gilman, J. Witkowski, M. Zoller, Recombinant DNA, 2nd ed., Scientific American Books, 1992.
[66] I. Wegener, The Complexity of Boolean Functions, Wiley/Teubner, New York/Stuttgart, 1987.

[67] N.E. Weste, K. Eshragan, Principles of CMOS VLSI Design, Addison-Wesley, Reading, MA, 1993.
[68] J. Williams, A. Ceccarelli, N. Spurr, Genetic Engineering, Bios Scientific Publishers, Oxford, UK, 1993.
[69] C. Zandron, C. Ferretti, G. Mauri, A reduced distributed splicing system for RE languages, in: Gh. Păun, A. Salomaa (Eds.), New Trends in Formal Languages: Control, Cooperation, Combinatorics, Lecture Notes in Computer Science, Vol. 1218, Springer, Berlin, 1997, pp. 319–329.