Comments on Theoretical Biology, 7: 177–198, 2002

0894-8550/02 $12.00 + .00
DOI: 10.1080/08948550290022123

DNA Computing: Models and Implementations

Mark J. Daley and Lila Kari
Department of Computer Science,
University of Western Ontario, London, Ontario, Canada

As the fabrication of integrated circuits continues to take place on increasingly smaller scales, we grow closer to several fundamental limitations on electronic computers. For many classes of problems, computing devices based on biochemical reactions present an attractive alternative to conventional computing paradigms. We present here a survey of the theory and implementation of biologically and biochemically based computers.

Keywords: DNA computing, biocomputing, molecular computing

The search for new methods of computing is something that has engaged humankind for as long as history has been recorded. From the abacus and Napier's bones (1658) to the modern electronic supercomputer, people have continuously sought new ways to automate the task of performing computations.

In recent decades the word "computer" has become synonymous with an electronic computing machine due to the overwhelming success of this particular paradigm. This, however, is only part of the story. Indeed, as we build faster and faster electronic computers, we are beginning to reach physical limits, beyond which our current technology cannot venture. There

This research was partially funded by National Science and Engineering Research Council of Canada grant R2824A01 to Lila Kari.

Address correspondence to Lila Kari, Department of Computer Science, University of Western Ontario, London, Ontario, N6A 5B7, Canada. E-mail: lila@csd.uwo.ca

are fundamental limits to how fast an electron can travel through a conductor (Lloyd, 2001).

Having recognized that the current electronic technology must reach a plateau at some point, computer science is beginning to expand to include the study of nontraditional methods of computation. These new paradigms may replace electronic computing, but it is more likely they will complement it by exploiting their unique advantages.

The basic foundations for biomolecular computing were laid out by Bennett and Landauer in their 1985 paper "The Fundamental Physical Limits of Computation" (Bennett & Landauer, 1985).

The first paper to explicitly use DNA as a computational medium was published by Leonard Adleman in 1994 (Adleman, 1994). In this paper, Adleman gave a description of how a combination of DNA, the correct enzymes, and appropriate laboratory protocols can solve an instance of the Traveling Salesman Problem (TSP). What was particularly attractive about this paper was that, in addition to providing the theoretical framework for such a computation, it also provided concrete results from a complete experimental implementation of the proposed protocol. Consequently, it is Adleman's experiment that is generally viewed as the landmark for the genesis of the field of biomolecular computing.

This single DNA computing experiment sparked the interest of a number of researchers in both computer science and molecular biology, and soon there were several competing models of DNA computing. Adleman's choice to solve a problem that is known to be NP-complete put exceedingly high expectations on DNA computing and resulted in some constructive criticism in the form a brief complexity analysis by Hartmanis (1995). In this paper, Hartmanis shows that using Adleman's protocol for solving a 200-node instance of the TSP would require 24 Earth masses of DNA. While this is clearly not feasible, subsequent work in Suyama et al. (1997) showed that modifications of the protocol lead to realistically solving much larger TSP instances. The objection that we raise to Hartmanis's critique is that he evaluated the efficiency of an experiment that was intended as a proof of concept, and an astonishing one for that matter. Similar criticisms of early transistor-based electronic computers would have led us to believe that they too would never become a viable technology.

In this article we introduce and provide a high-level view of a number of unique models and implementations of DNA computing. The results considered have been chosen to reflect a representative sample of key research areas within the field.

The article begins with two introductory sections on basic molecular biology and computer science. We then continue with an exposition of splicing systems and other theoretical models. The fifth section is devoted to more practical, implementation-based approaches. The sixth section provides a brief overview of the new field of *in vivo* computation, and the final section presents concluding remarks.

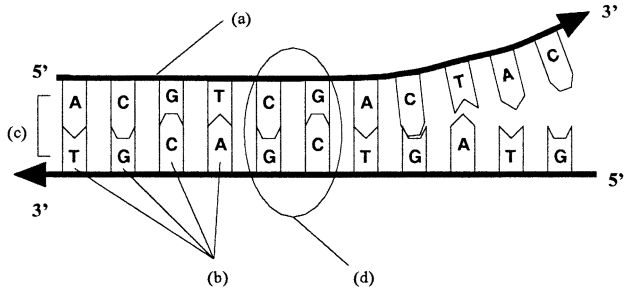

FIGURE 1 A DNA double strand. From Kari et al. (2001).

BASIC MOLECULAR BIOLOGY FOR DNA COMPUTING

DNA computing is based on the idea that molecular biology processes can be used to perform arithmetic and logic operations on information encoded as DNA strands.

A DNA single strand consists of four different types of units called *bases* (Figure 1b) strung together by an oriented *backbone* (Figure 1a) like beads on a wire. The bases are adenine (A), guanine (G), cytosine (C), and thymine (T), and A can chemically bind to an opposing T on another single strand (Figure 1c), while C can similarly bind to G (Figure 1d). Bases that can thus bind are called *Watson/Crick (W/C) complementary*, and two DNA single strands with opposite orientation and with W/C-complementary bases at each position can bind to each other to form a DNA double strand (Figure 1) in a process called *base pairing*.

To encode information using DNA, one can choose an encoding scheme mapping the original alphabet onto strings over {A, C, G, T}, and proceed to synthesize the obtained information-encoding strings as DNA single strands. A computation will consists of a succession of *bio-operations* (Kari, 1997), such as cutting and pasting DNA strands, separating DNA sequences by length, extracting DNA sequences containing a given pattern, or making copies of DNA strands. The DNA strands representing the output of the computation can then be read out and decoded.

Herein lie a wealth of problems to be explored, stemming from the fact that encoding information in DNA differs from encoding it electronically, and bio-operations differ from electronic computer operations. A fundamental difference arises from the fact that in electronic computing data interaction is fully controlled, while in a test-tube DNA computer, free-floating data-encoding DNA single strands can bind because of W/C complementarity. Another difference is that in DNA computing, a bio-operation usually consumes both operands. This implies that if one of the operands is either involved in an illegal binding or has been consumed by a

previous bio-operation, it is unavailable for the desired computation. Yet another difference is that while in electronic computing a bit is a single individual element, in DNA experiments each submicroscopic DNA molecule is usually present in millions of identical copies. The bio-operations operate in a massively parallel fashion on all identical strands. However, this process is governed by the laws of chemistry and thermodynamics, and the output obeys statistical laws.

Differences like the ones just mentioned point to the fact that a new approach should be employed when analyzing and processing DNA-encoded information. These differences are starting to be tapped into by research in DNA computing, as will become apparent in the examples presented in this article. For more molecular biology terminology and notions the reader is referred to Kari (1997), Watson et al. (1987), and Calladin and Drew (1999).

BASIC COMPUTER SCIENCE

Most of the existing models of DNA computing have their formal basis in the theory of computing. Theoretical computer science is a broad area, which, at its highest level, encompasses the abstract study of the process of computation. We present here a brief introduction to those areas necessary to understanding the material in the sequel; for a more thorough introduction the reader is referred to Hopcroft et al. (2001).

The automata theory branch of computer science deals with abstract machines that take an arbitrary string of symbols as an input and either produce an output string, or give a "Yes/No" answer to the question "Is the input string accepted by this machine?" By varying the way in which these machines are allowed to operate (for example: giving them access to data structures for use as a "scratchpad"), we can vary their computational power.

Computational power is measured in terms of what types of functions a given machine can perform. The most general abstract model of computing, and the one with the greatest computational power, is known as the Turing machine. The Turing machine was proposed by Alan Turing as an abstract device capable of computing any computable function. Thus, to prove that a new model of computing achieves the highest level of computational power, it suffices to prove that it is equivalent to a Turing machine. This is the approach taken in most of the models reviewed in this article.

A Turing machine consists of two parts: a finite-state control (which we can think of as the program) and a data tape (which we think of as the storage). The data tape can be thought of as an infinite series of cells, each capable of holding a single symbol. The data tape also contains a read/write head, which is over a particular cell at any given time in the computation.

The finite state control consists of a set of states, represented as nodes in a graph, with a transition function that allows moving between states. A transition function is usually represented as an edge in the state graph.

A transition between two states s and s', labeled by r, w, m, can be interpreted as follows: To move from state s to state s', if the head of the Turing machine currently is over a cell containing the symbol r, this symbol must then be overwritten with the symbol w, and the head is then moved in the direction m, where m is either "Left", "Right", or "Don't move." If the symbol r is not present under the current position of the head, this transition from state s to state s' cannot happen. A word is accepted by a Turing machine if there exists a sequence of state transitions that starts with the "start state" and the given word on the tape and ends in a final state. The start state and final state are given in the description of the Turing machine. The language accepted by a Turing machine is the set of all words accepted by the Turing machine.

In addition to measuring computational power in terms of the functions an abstract machine can compute, we can also measure it in terms of what languages an abstract machine (such as a Turing machine) can accept. A language is a set of strings, and a string is an ordered set of symbols chosen from a given finite alphabet set. Depending on the form of the strings in a given language, we can place the language in one of several classes of formal languages.

Languages are customarily classified according to the Chomsky hierarchy of languages (Hopcroff et al., 2001). Inside the Chomsky hierarchy, each class of languages represents strictly more computational power than the previous class. The hierarchy consists of regular languages (REG) at the bottom, followed by context-free languages (CF), context-sensitive languages (CS), and ending with the recursively enumerable (RE) languages at the top. Any machine that can accept, or generate, recursively enumerable languages is equivalent to a Turing machine. Following this informal description of computational models, we now give several definitions and notations used in this article.

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

Since languages are sets, we may define the set-theoretic operations of union, intersection, difference, and complement in the usual fashion.

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

A *generative grammar* is an ordered quadruple

$$G = (N, T, S, P) \tag{1}$$

where N and T are disjoint alphabets, $S \in N$, and P is a finite set of ordered pairs (u, v) such that u, v are words over $N \cup T$ and u contains at least one letter of N. The elements of N are called *nonterminals* and those of T *terminals*; S is called the axiom. Elements (u, v) of P are called rewriting rules and are written $u \rightarrow v$. If $x = x_1 u x_2, y = x_1 v x_2$, and $u \rightarrow v \in P$, then we write $x \Rightarrow y$ and say that x derives y in the grammar G. The reflexive and transitive closure of the derivation relation $\Rightarrow$ is denoted by $\Rightarrow^*$. The language generated by G is

$$L(G) = \{w \in T^* | S \Rightarrow^* w\}. \tag{2}$$

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

Grammars are classified by imposing restrictions on the forms of productions. A grammar is called *type-0* if no restriction (zero restrictions) is applied to the rewriting rules, and is called *regular* if each rule of P is of the form $A \rightarrow aB, A \rightarrow a, A, B, \in N, a \in T$. The family of finite languages will be denoted by FIN, the family or languages generated by regular grammars by REG, and the family of languages generated by type-0 grammars by $\mathcal{L}_0$. $\mathcal{L}_0$ coincides with the family of languages accepted by Turing machines, and therefore a generative device that generates any language in $\mathcal{L}_0$ has the maximum computational power of a Turing machine. Such a generative device, the *splicing system*, is described in the next section.

SPLICING SYSTEMS

As described earlier, a DNA strand can be likened to a string over a four-letter alphabet. Consequently, a natural way to model DNA computation is within the framework of formal language theory, which deals with letters and strings of letters. Perhaps this is one of the reasons why the splicing systems introduced by Tom Head in 1987 as formal-language-based computational systems using splicing (an abstraction of DNA recombination) as their sole computational operation preceded the advent of DNA computing by 7 years.

The DNA recombination (a combination of cutting of DNA strands by restriction enzymes and crosswise pasting the obtained pieces by DNA ligases) that led to the abstract operation of splicing is illustrated by the following example from Head (1987). The reader can consult Head (1987, 1992), Paun and Salomaa (1996), and Paun et al. (1998), as well as Head et al. (1996) for details.

Consider the following two double strands of DNA:

$$\begin{array}{ll} 5'CCCCCTCGACCCCC3' & 5'AAAAAGCGCAAAAA3' \\ 3'GGGGGAGCTGGGGG5' & 3'TTTTTCGCGTTTTT5' \end{array} \tag{3}$$

and two restriction enzymes (*Taq*I and *Sci*NI), whose recognition sites are

$$\begin{array}{ll} 5'TCGA3' & 5'GCGC3' \\ 3'AGCT5' & 3'CGCG5' \end{array} \tag{4}$$

respectively. The effect of the enzymes on the two given DNA strands is the cutting, by each enzyme, of the strand containing its restriction site as a subsequence. As a result, four new DNA strands are produced:

$$\begin{array}{llll} 5'CCCCCT & CGACCCCC3' & 5'AAAAAG & CGCAAAAA3' \\ 3'GGGGGAGC & TGGGGG5' & 3'TTTTTCGC & GTTTTT5' \end{array} \tag{5}$$

Note that the sticky end of the first strand is complementary to the sticky end of the last one, while the same thing happens with the second and third strands. The DNA strands with compatible ends can now be ligated by DNA ligase, the result being

$$\begin{array}{ll} 5'CCCCCTCGCAAAAA3' & 5'AAAAAGCGACCCCC3' \\ 3'GGGGGAGCGTTTTT5' & 3'TTTTTCGCTGGGGG5' \end{array} \tag{6}$$

The DNA recombination operation exemplified above has been formalized in Head (1987) and modified in Gatterdam (1994) as follows. Given an alphabet X and two strings x and y over X, the splicing of x and y according to the splicing rule r consists of two steps: (1) cut x and y at certain positions determined by the splicing rule r, and (2) paste the resulting prefix of x with the suffix of y, respectively the prefix of y with the suffix of x. Using the formalism introduced in Paun (1996a), a splicing rule r over X is a word of the form $\alpha_1\#\beta_1\$\alpha_2\#\beta_2$, where $\alpha_1, \beta_1, \alpha_2, \beta_2$ are strings over X and £, \$ are markers not belonging to X.

We say that z and w are obtained by splicing x and y according to the splicing rule $r = \alpha_1\#\beta_1\$\alpha_2\#\beta_2$ (Figure 2), and we write

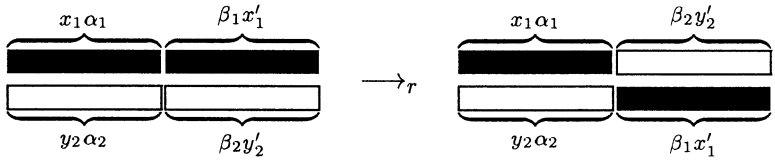

FIGURE 2 Splicing $x = x_1\alpha_1\beta_1 x_1'$ and $y = y_2\alpha_2\beta_2 y_2'$ according to the rule $r : \alpha_1\#\beta_1\ \$\alpha_2\#\beta_2$.

$$(x, y) \rightarrow (z, w) \tag{7}$$

if and only if

$$\begin{array}{l} x = x_1\alpha_1\beta_1x_1' \\ y = y_2\alpha_2\beta_2y_2' \end{array} \quad \text{and} \quad \begin{array}{l} z = x_1\alpha_1\beta_2y_2' \\ w = y_2\alpha_2\beta_1x_1' \end{array} \tag{8}$$

for some $x_1, x_1', y_2, y_2' \in X^*$.

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

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

One can define and classify families of splicing languages based on the types of language families the axiom set, respectively the set of rules belong to. For example, H (FIN, REG) denotes the family of languages generated by splicing systems where the set of axioms is finite and the set of rules is a regular language. Splicing systems have been extensively studied in the literature. For example, the generative power of different types of splicing systems has been studied in Csuhaj-Varju, Freund et al. (1996), Culik and Harju (1991), Freund et al. (1999), Gatterdam (1989), Head (1987), Paun (1995, 1996a, 1996b), and Paun et al. (1996). It has been proved in Freund et al. (1999) that splicing systems with multisets where both the set of axioms and the set of rules are finite generate the entire class of computable languages. (In a multiset of strings, each string has associated with it a multiplicity that decreases when the string is used as an operand in a splicing step and increases when the string occurs as a result of a splicing.) Decidability problems have been tackled in Deninnghoff and Gatterdam (1989). Moreover, other variations of the model have been considered: splicing systems with permitting/forbidding contexts in Csuhaj-Varju, Freund et al. (1996), linear and circular splicing systems in Head (1992), Pixton (1995) and Yokomori et al. (1997), splicing systems on graphs in Freund (1995), and distributed splicing systems in Csuhaj-Varju, Kari et al. (1996) and Dassow and Mitrana (1996). For a survey of the main results on splicing the reader is referred to Head et al. (1996), Paun and Salomaa (1996), and Paun et al. (1998).

An experimental implementation of a simple example of a wet splicing system has been shown in Laun and Reddy (1997) to generate in vitro the splicing language predicted by the corresponding theoretical splicing system.

POST-ADLEMAN FORMAL MODEL EXPLOSION

Direct Implementation

In many models of DNA computation, the authors have either devised or used a formal model equivalent to a Turing machine, rather than a direct implementation of a Turing machine. Rothemund (1996) proposes a model that directly implements a Turing machine with DNA and restriction endonucleases.

The fundamental idea behind this model is that the tape of the Turing machine is kept as a circular strand of DNA, while the rules of the Turing machine are implemented as transition oligonucleotides (oligo). When added to a population of "tapes" (circular DNA strands) along with specific enzymes, the appropriate transition oligo will become incorporated into the tape, thus simulating the action of a Turing machine. In the following we describe in detail the tape and then the transition rules.

The DNA strand encoding the Turing machine tape contains the current contents of the abstract Turing tape as well as information about the state of the Turing machine. In reality, then, rather than simply representing the Turing machine, our DNA strand becomes an *instantaneous description* of the Turing machine.

The instantaneous description is composed of three parts. The first consists of the symbols currently on the Turing tape. Each symbol in the input alphabet is assigned a unique DNA sequence to represent it. In addition, there are two *invariant* symbols L and R that are appended to the left and right sides of a symbol, respectively. The sequences used to represent the symbols must be chosen carefully so that each generates a unique sticky and when cut.

The second part of the instantaneous description is the head of the Turing machine represented by two adjacent symbols *Inv* and q_a; where q_a is a state of the Turing machine. These two symbols then act as restriction sites for enzymes, with the restriction site labeled by the state always pointing to the current symbol on the tape.

The third part of the instantaneous description is the current state of the Turing machine, which is actually encoded as the spacing between a recognition site for a restriction enzyme and the current symbol. The state is encoded in this fashion since by varying the number of bases between the recognition site and the current symbol, we can control exactly where the current symbol will be cut.

If the DNA sequences used for the symbols have been chosen carefully, as above, so that each symbol generates a unique sticky end, this property can be used to allow for the ligation of a new transition oligonucleotide into our DNA strand. This transition oligo encodes the new state, symbol, and direction of the Turing machine.

Given this information, each rule in our transition table is encoded with four transition oligos, one for each of the above-mentioned cases.

A computation step of the Turing machine consists of six distinct steps:

1. Cut the current symbol with two restriction enzymes, *state* and *invariant*. The state enzyme will create a sticky end unique to the current state and input symbol. The invariant enzyme, appropriately enough, cuts an L or R invariant symbol to create a uniform other end. The result is that our DNA tape is no longer circular, but a single strand with one unique end and one uniform end.
2. We mix all of the transition oligos (remember that there are four for each transition rule) with the homogenous population of DNA Turing tapes. At this point, the unique sticky end created in Step 1 is bound to the correct transition oligo (which will have a complementary sticky end) by DNA ligase. Since the other end of the transition oligo will not be unique, it is protected by a small sequence *cap*.
3. Now that the unique end of the transition oligo has been successfully ligated to the DNA tape, the cap on the other end is removed.
4. The removal of cap in the preceding step creates a sticky end, which is now ligated into the DNA tape returning it to a circular DNA strand.
5. A symbol-excision restriction enzyme is now applied in order to cut out the previous symbol from the tape.
6. The enzyme in Step 5 leaves two matching ends so the DNA is again recircularized and left with the state restriction site pointing to the current symbol.

After Step 6, a polymerase chain reaction (PCR) is run on a small sample from the computation using the *Halt* sequence as a primer. In this manner, only halted tapes would be amplified and would thus be easily detectable.

An obvious attractive feature of this model is that it directly implements a Turing machine as the model of computation. This is clearly advantageous over implementations that use new or less frequently used models of computation.

Moreover, from the very beginning, Rothemund indicates the exact reactions that will implement each of the operations. Rather than relying on fictitious or yet-to-be-discovered biochemistries, everything in his paper is implementable using currently available restriction enzymes and laboratory techniques. This being said, this model is not practical for large-scale implementation. The most difficult aspects are the prohibitively high cost of Class IIS restriction enzymes and the relatively sparse selection available.

Insertion/Deletion Systems

Two of the most fundamental biological operations on DNA are that of cutting out (or deleting) a segment of DNA, and inserting a DNA sequence into existing DNA. The existence of these operations in nature makes the formal model of insertions and deletions given in Kari (1991) a natural candidate for DNA computing.

Given words u and v, the insertion of v into u consists of all words that can be obtained by inserting v in an arbitrary position of u:

$$u \leftarrow v = \{u_1 v u_2 | u = u_1 u_2, u_1, u_2 \in X^*\} \tag{9}$$

In the context of biomolecular computing, the insertion operation is too nondeterministic to model the type of insertions occurring in DNA strands. Consequently, a modified version, *contextual insertion*, was defined in Kari and Thierrin (1996) to capture the fact that insertions of DNA strands are context-sensitive. Given a pair of words $(x, y) \in X^*$, called a context, the (x,y)-contextual insertion of v into u is defined as

$$u \overset{(x,y)}{\leftarrow} v = \{u_1 x v y u_2 | u_1, u_2 \in X^*, u = u_1 x y u_2\} \tag{10}$$

An insertion deletion system $ID = (X, T, I, D, \omega)$ is composed of an alphabet X, a set of terminal symbols $T \subseteq X$, insertion $[I \subseteq (X^*)^3]$ and deletion $[D \subseteq (X^*)^3]$ rules, and an axiom $\omega \in X^*$. Insertion rules are given in the form $(c_1, x, c_2)_I$, which means that the word x can be inserted in the current word at a point where the context $c_1 c_2$ exists. Similarly, a deletion rule $(c_1, d, c_2)_D$ indicates that if the substring $c_1 d c_2$ exists in the current word, then d can be deleted.

It is proven in Kari and Thierrin (1996) that linear insertion/deletion systems have expressive power equivalent to a Turing machine.

Although an insertion/deletion system that operates on purely linear strands of DNA is capable of universal computation, it does not capture the important biological reality of plasmids. A plasmid is a strand of DNA that is circular, rather than linear. In order to model plasmids as well as linear strands, the insertion/deletion model was extended in Daley et al. (1999) to include circular strands of DNA.

Daley et al. (1999) also contains the results of a laboratory implementation of a small example of a circular insertion/deletion system.

Both linear and circular insertion/deletion provide computing models that are not only universal, but based on existing fundamental biological operations. The most positive aspect of this is that implementing these models in a laboratory (and perhaps in vivo in the future) is possible since the techniques involved are well understood and easily repeatable. One

drawback, pointed out in J. Khodor (personal communication), is that there is no obvious way to implement a deletion rule of the form (c_1, x, c_2) for a specific x. Indeed, the current system will delete anything flanked by the sequences c_1 and c_2.

Equality Checking

The equality checking model originally proposed by Yokomori and Kobayashi (1997) presents an interesting alternative to current models. The model focuses around the idea that an equality machine could be effectively implemented in the DNA biochemical domain. Essentially, computation is achieved by the equality machine keeping two output tapes during the course of reading an input tape and simply checking whether or not the two output tapes are equal at the end of the computation.

An *equality* machine consists of four major components: finite control, a one-way read-only input tape, and two read-write output tapes.

In Yokomori and Kobayashi (1997) this is formalized by representing an equality machine M as $M = (Q, \Sigma, \Delta, \delta, q_0, F)$ where Q is a finite set of states, Σ the input alphabet, Δ the output alphabet, δ a set of transition relations, q_0 the initial state, and F the set of final states $(q_0 \in Q, F \subseteq Q)$. The finite control is represented as a transition function, mapping states and inputs of read-only tape onto new states and associated outputs on the two output tapes. A transition is of the form $(p, a) \rightarrow (q, u, i)$, where p and q are states, a is an input symbol, u is an output symbol, and i refers to output tape 1 or 2.

According to Yokomori and Kobayashi (1997) their research into a DNA implementation of this model was motivated by the result that the family of languages accepted by nondeterministic equality machines equals the set of recursively enumerable languages (Engelfriet & Rozenberg, 1980).

The operations presented in this particular model are classified into three distinct categories: *basic* operations, *test set* operations, and *set* operations. There is only one basic operation:

- AM (*Amplify*), which, given a test tube T, produces two new test tubes, T, T' with exactly the same contents as T.

There are two test set operations:

- EM (*emptiness test*), which, contrary to what one might expect, returns a "Yes" if there *is* a string in the given test tube and "No" if the given test tube contains no strings.
- EQ (*equivalence test*), which takes a single test tube and returns "Yes" if the test tube contains at least one complete double-stranded string and "No" otherwise.

Finally, there is a collection of set operations:

Name	Syntax	Result
UN (union)	$T = T_1 \cup T_2$	$T_1 \cup T_2$
LC (left cut)	$T = a/T_1$	$\{u \mid au \in T_1\}$
LA (left add)	$T = a \cdot T_1$	$\{a\} \cdot T_1$
EX (extract)	$T = \mathrm{Ex}(T_1, \omega)$	$T_1 \cap \Sigma^*\{\omega\}\Sigma^*$
RE (replace)	$T = \mathrm{Re}(T_1, u, v)$	$\{xvy \mid xuy \in T_1\}$

From a strictly mathematical point of view, the DNA equality checking model offers an attractive theoretical basis for computation. The number of atomic operations required is not exceedingly high and the model itself is simple enough (recall that Alan Turing's goal in the creation of the Turing machine was to design the simplest possible model of computation). The feasibility of this DNA implementation of the model, however, is still untested.

GETTING WET: DNA COMPUTING IMPLEMENTATIONS

The majority of models for DNA computing in the current literature are primarily theoretical. Most have been designed by computer scientists and mathematicians with the goal of proving that universal computation can be achieved by DNA computing. Although this is clearly a critical property for any successful model of DNA computing, it is not necessarily the best starting point for practical DNA computing applications. In this section we examine models inspired by biological techniques that lend themselves much more readily to implementation using existing methods and technologies.

Whiplash PCR, proposed in Kiga et al. (1997) and Sakamoto et al. (2000), is based on the classical model of a state machine. The state machine consists of a single-stranded DNA molecule with the subsequence at the 3′ end of the molecule being considered the current state of the machine. Since the molecule is single-stranded, it will naturally attempt to bond with another complementary strand of DNA to form a stable double-stranded molecule. In the absence of other molecules, and if a self-complementary subsequence exists, the single strand will form a bond with itself, creating what is known as a hairpin structure (Figure 3).

This allows one to encode the state machine's transition table in the molecule itself in the form **stop–newstate–oldstate**. This creates a situation where if the 3′ head of the DNA molecule (the current state) is complementary to the sequence representing **oldstate** then the two will anneal, forming a new molecule. At this point, the enzyme DNA polymerase will attempt to extend the old state toward the new state.

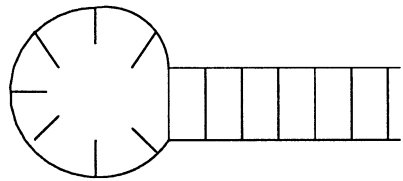

FIGURE 3 Hairpin structure.

Although it is possible to perform single-molecule computations with such a system, its real power lies in the ability to have many such molecular state machines operating in parallel. Eric Winfree (1998) presents a modification of this system that is capable of solving some NP-complete problems in $O(1)$ biosteps (although, of course, the space requirements make solving large instances impractical) using parallelization.

In contrast to whiplash PCR, which is solution-based, the model of *surface-based* DNA computing emerged, which is dependent upon the DNA strands being affixed to a physical surface. Liu et al. (2000) present a method for solving the satisfiability problem (SAT) that involves working with single-stranded DNA affixed to a gold surface.

The first step involves generating the entire solution set for the given problem (with respect to the chosen encoding) and then attaching each potential solution DNA single strand to the glass surface. We work here with the example given in Liu et al. (2000). Each base in a given single strand represents a single bit, A and C for 0, G and T for 1.

The algorithm for generating the solution is then implemented as an iterated sequence composed of a combination of operations: *mark* (i, b): mark all strings in which bit i has value b; *mark* $[(i_1, b_1), (i_2, b_2), \ldots, (i_k, b_k)]$: extension of the *mark* operation to multiple bits; *destroy—marked*: removed all marked strings; *destroy—unmarked*: removes all unmarked strings; *unmark*: converts all marked strings into unmarked strings; *test—if—empty*: determine if all strings have been destroyed.

Marking is accomplished by synthesizing the set of DNA strands that are complementary to the strings in which bit i has value b and allowing these strands to anneal to the surface-affixed strands. The destroy-marked and destroy-unmarked operations are easily accomplished by washing the surface with a solution containing enzymes known as exonucleases that will specifically destroy either single-stranded or double-stranded DNA. Unmark can be accomplished by applying distilled water to the surface. The low salinity of the water will destabilize the double-stranded DNA and cause it to separate into single strands again. Finally, the test for emptiness is implemented by removing the DNA from the surface, amplifying it with PCR, and checking for a product.

Liu et al. (2000) presented experimental results from an implementation of a solution to a four variable, four-clause, 3-SAT problem using surface-based computing and concluded that the methods they have presented lend themselves reasonably well both to scaling to larger problems and, potentially, to automation. The largest DNA computation to date using a solution-based approach, presented in Braich et al. (2002), involved finding the solution to a 20-variable instance of the 3-SAT problem.

Of necessity, formal models of DNA computing often reduce the actual structure of the DNA molecule to an approximate abstraction in order to remove details that are deemed irrelevant to the computational task at hand. One of the dangers of this approach is oversimplifying the situation and thus missing a key feature, as is the case with the three-dimensional structure of DNA itself that could be used in computing. Recognizing the importance of the 3D structure, such models were put forth by Winfree (1998b; Winfree, Yang et al. 1999), which gave rise to the DNA self-assembly approach to computation.

The general idea is to design DNA structures called tiles that are capable of self-assembly, thus resulting in one- or two-dimensional macrostructures when allowed to react in solution. By carefully choosing the construction of the tiles, the assembly process becomes a single-pot computation. Examples of implementations of the tiling approach include solving SAT (Lagoudakis and LaBean 1999), performing XOR an addition operations (Winfree, LaBean et al., 1999), generating the output languages of finite-visit Turing machines (Rozenberg et al., 2001), and Horn clause computation (Hagiya et al., 2001). Details on experimental design of DNA tiles can be found in Winfree et al. (1998).

The results we have seen so far have one common factor tying them together: They all use DNA as their primary information medium. In Lipton et al. (1999) the possibility of using another information-rich biomolecule, RNA, is explored.

Traditionally, RNA has been avoided in biomolecular computing research due to the intrinsic difficulty in working with the molecule. RNA is far less stable than DNA and this problem is exacerbated by the fact that the enzyme RNA-ase (which digests RNA) is present on the skin of every human being. Working with RNA requires stricter lab protocols and more experience than is required to perform simple DNA experiments.

Landweber (1999) designed an RNA-computing experiment to solve the "Knights Problem" on a three-by-three chessboard. This problem consists of finding all the ways of placing knights on the chessboard such that no knight may be placed on a square that can be attacked by another knight. The experiment generated 43 solutions, which were then verified by hand, resulting in 42 of the solutions being declared correct.

Recently, work has been started on possibly using peptide-antibody interactions for computing (Balan et al., 2001), and progress has been made

also in the direction of constructing a programmable DNA-based device (Benenson et al., 2001).

BORROWING FROM NATURE: IN VIVO COMPUTING

All of the preceding methods have been in vitro attempts to utilize molecular biological techniques for the purpose of computing, but are these optimal solutions? The two models presented next are instead derived from natural biological phenomena that have evolved over billions of years of selection. It is quite possible, and indeed likely, that nature may have developed better solutions to the problems we face.

Bloom and Bancroft (1999) present a practical framework for biomolecular computation using liposomes. A liposome is a biological membrane composed of a lipid bilayer and used in vivo for the separation of material within a cell. The structure of the membrane can be seen in Figure 4.

Through standard lab techniques, such membranes can be artificially constructed and operated upon in order to allow the segmentation of a reaction on a microscopic scale. Bloom and Bancroft (1999) exposit six basic operations that can be performed on liposomes.

The first operation is initialization, which consists of constructing a liposome containing the desired contents. A number of implementation possibilities are then given for a merge/union operation, which would allow two liposomes to aggregate into a single liposome, combining the contents of the originals. A number of utility operations are also proposed: A filter operation, capable of removing small solutes from a liposome while retaining large ones; an operation for separating a population of liposomes based on size; an operation for detecting the occurrence (or lack thereof) of a merge/union event; and finally, the ability to lyse (break) all membranes releasing the contents for analysis. Potential applications of this model in DNA self-assembly (tiling) and other models were suggested by the authors, though no implementation using these techniques has yet been attempted. However, it seems that this construct is ideally suited to a method of

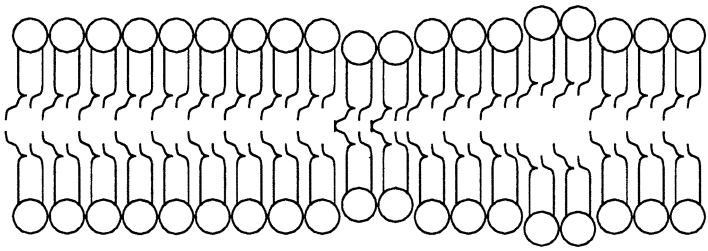

FIGURE 4 Lipid bilayer.

computation based on membranes. Interestingly enough, a theoretical model of computing with membranes was independently proposed by Paun (2000). This model is being thoroughly studied by computer scientists and has had extensions (Martin-Vide et al., 2001; Paun, 2001) and electronic simulations (Baranda et al., 2001) proposed.

Supercell systems (or P-systems) are based on the idea of a hierarchy of membranes, each containing various rules and substrates for those rules to act upon. A set of standard rewriting rules (with the exception of an additional reserved symbol δ, which causes the current membrane to lyse) is given for each membrane, along with any substrates to which the rules can be applied. The rules are then applied, in parallel, until a state is reached where either the membrane is lysed or no further application is possible. In order to avoid ambiguity in cases where two or more rules may be applicable simultaneously, an explicit priority ordering of rules is allowed. When a rule results in a δ being written, the membrane containing the δ is lysed and all rules associated with this membrane are lost. The substrates, however, are now present in the parent membrane and are subject to the rules present here.

One of the most exciting side effects to arise from the current interest in the field of DNA computing is the rapid growth in the number of groups attempting to model cellular processes as computation. In some cases, the same formal models that have been used to describe in vitro DNA computing systems can be modified to provide a model of actual in vivo genetics. We briefly look here at a formal mathematical model proposed by Landweber and Kari (1999) for the in vivo gene rearrangement process found in hypotrichous ciliates. Recently, Petre et al. (2001) have proposed an alternative model for the same process.

In many organisms, the genes encoding specific proteins are linear DNA segments contained in the organism's genome. The beginning and end of the gene are marked with special marker sequences so that the cellular transcription mechanism knows where to begin and end transcribing a given gene. In certain species of ciliates, however, the encoding of genes is not nearly as straightforward. In fact, the DNA representing the genes is actually broken up into smaller subsections. These subsections are moreover scrambled in nonlinear order. Thus, for these ciliates to properly transcribe their DNA they must have some method of first reassembling the scrambled genes into the correct order.

The DNA bio-operation that apparently accomplishes the unscrambling is illustrated in Figure 5. Landweber and Kari showed that the *contextual guided recombination systems*, which are generative mechanisms based on the bio-operations in Figure 5, have universal computational power (i.e., they are equivalent in power to a Turing machine). This result points to the possibility of solving man-made computational problems by using the ciliate system, and even to the notion of a "programmable cell."

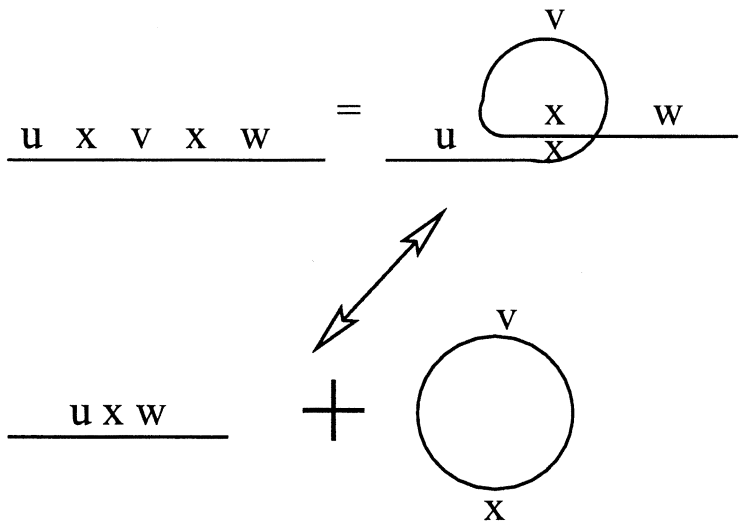

FIGURE 5 Recombination operations. A linear DNA strand *uxvxw* in the presence of direct repeats *x* will undergo recombination, resulting in a circular DNA strand $^{\bullet}vx$ and a shorter DNA strand *uxw*. The operation can happen also in the opposite direction: A circular DNA strand can be inserted into a linear one, provided they share a common sequence *x*.

SUMMARY AND CONCLUSIONS

In this article we have presented a brief survey of several research areas in the field of DNA computing. We presented a number of formal models as well as implementations of biomolecular computing, where biomolecules and biochemical reactions are used to perform specific computations. The advantage of this mode of computation over more traditional methods (e.g., electronic computing) is the promise of massive parallelism that would be unachievable in silicon. Clearly, biomolecular computing is not an appropriate solution to every computational problem, given the amount of overhead involved in setting up a reaction protocol. However, as laboratory techniques mature, DNA computing may become an attractive alternative to electronic computers for difficult problems that would benefit from a parallel approach.

In addition to the in vitro approach, more interest is now being placed on in vivo computing. The ability to "program" a living cell offers not only interesting computational possibilities, but also perhaps a new tool for medical and biological sciences. If one could accurately utilize the genetic mechanisms of a cell for accomplishing man-made tasks, it is theoretically possible that a collective of such cells could be used as

biological nanomachines to carry out many of the desirable medical functions described in the current nanotechnology literature (Freitas, 2001).

In an era when the term *biotechnology* is used loosely by many, DNA computing offers the possibility of true biotechnology that will allow large, difficult computations to be performed in reasonable time; possibly the ability to utilize cellular function at the genetic level; new tools and methods for analyzing the overwhelming amount of genetic information that global efforts in gene sequencing and analysis are producing; and most significantly, a formal mathematical framework for understanding the underlying mechanisms of cellular genetics.

REFERENCES

Adleman, L. 1994. Molecular computation of solutions to combinatorial problems. *Science* 266:1021–1024.

Balan, M. S., K. Krithivasan, and Sivasubramanyam. 2001. *DNA Computing* (*DNA-Based Computers 7*), *LCNS 2340* (Lecture Notes in Computer Science), N. Jonoska, N. C. Seeman, Eds., pp. 290–299. Berlin: Springer-Verlag.

Baranda, A. V., F. Arroyo, J. Castellanos, and R. Gonzalo. 2001. *DNA Computing* (*DNA-Based Computers 7), LCNS 2340* (Lecture Notes in Computer Science), N. Jonoska, N. C. Seeman Eds., pp. 350–359. Berlin: Springer, Verlag.

Benenson, Y., T. Paz-Elizur, R. Adar, Z. Livneh, E. Keinan, and E. Shapiro. 2001. Programmable and autonomous computing machine made out of biomolecules. *Nature* 414:430–434.

Bennett, C. H., and R. Landauer. 1985. The fundamental physical limits of computation. *Sci. Am.* 253(1):48–56 (intl. ed. 38–46).

Bloom, B., and C. Bancroft. 1999. Liposome-mediated biomolecular computation. In *DNA Based Computers* V, eds. E. Winfree and D. Gifford, pp. 39–48. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 54. Providence Rhode Island: American Mathematical Society Press.

Braich, R. S., N. Chelyapov, C. Johnson, P. W. K. Rothemund, and L. Adleman. 2002. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296:499–502.

Calladin, C. R., and H. R. Drew. 1999. *Understanding DNA: The Molecule and How It Works.* San Diego: Academic Press.

Csuhaj-Varju, E., R. Freund, L. Kari, and G. Paun. 1996. DNA computing based on splicing: Universality results. In *First Annual Pacific Symposium on Biocomputing*, eds. L. Hunter and T. E. Klein, pp. 179–190. Singapore: World Scientific.

Csuhaj-Varju, E., L. Kari, and G. Paun. 1996. Test tube distributed systems based on splicing. *Comput. AI* 15:211–232.

Culik, K., and T. Harju. 1991. Splicing semigroups of dominoes and DNA. *Discrete Appl. Math.* 31:261–277.

Daley, M., L. Kari, R. Siromoney, and G. Gloor. 1999. Circular contextual insertion/deletion with applications to biomolecular computation. *Proc. 6th Int. Symp. String Processing and Information Retrieval*, Cancun Mexico, 47–54.

Dassow, J., and V. Mitrana. 1996. Splicing grammar systems. *Comput. AI* 15:109–122.

Deninnghoff, K. L., and R. W. Gatterdam. 1989. On the undecidability of splicing systems. *Int. J. Comput. Math.* 27:133–145.

Engelfriet, J., and G. Rozenberg. 1980. Fixed point languages, equality languages, and representation of recursively enumerable languages. *J. ACM* 27:499–518.

Freitas, R. A., Jr. 2001. *Nanomedicine*, vol. I. Georgetown, VA: Landes Bioscience.

Freund, R. 1995. Splicing systems on graphs. *Proc. Intelligence in Neural and Biological Systems*, 189–194.

Freund, R., L. Kari, and G. Paun. 1999. DNA computing based on splicing: The existence of universal computers. *Theory Comput. Syst.* 32:69–112.

Gatterdam, R. 1989. Splicing systems and regularity. *Int. J. Comput. Math.* 31:63–67.

Gatterdam, R. 1994. DNA and twist free splicing systems. In *Words, Languages and Combinatorics II*, eds. M. Ito and H. Jürgensen, pp. 170–178. Singapore: World Scientific.

Hagiya, M., H. Uejima, and S. Kobayashi. 2001. Horn clause computation by self assembly of DNA molecules. *Proc. Seventh Int. Workshop on DNA Based Computers*, Tampa, 63–74.

Hartmanis, J. 1995. On the weight of computations. *Bull. Eur. Assoc. Theoret. Comput. Sci.* 55:136–138.

Head, T. 1987. Formal language theory and DNA: An analysis of the generative capacity of recombinant behaviors. *Bull. Math. Biol.* 49:737–759.

Head, T. 1992. Splicing schemes and DNA. In: *Lindenmayer Systems—Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pp. 371–383. Berlin: Springer-Verlag.

Head, T., G. Paun, and D. Pixton. 1996. Language theory and genetics. Generative mechanisms suggested by DNA recombination. In *Handbook of Formal Languages*, vol. 2, eds. G. Rozenberg and A. Salomaa, pp. 295–360. Berlin: Springer-Verlag.

Hopcroft, J., J. Ullman, and R. Motwani. 2001. *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Reading, MA: Addison–Wesley.

Kari, L. 1997. DNA computing: Arrival of biological mathematics. *Math. Intelligencer* 19(2):9–22.

Kari, L. 1991. *On Insertions and Deletions in Formal Languages*. Ph.D. thesis, University of Turku, Finland.

Kari, L., S. Konstantinidis, E. Losseva, and G. Wozniak. 2001. *Sticky-Free and Overhang-Free DNA Languages*, submitted.

Kari, L., and G. Thierrin. 1996. Contextual insertions/deletions and computability. *Inform. Comput.* 131:47–61.

Kiga, D., K. Sakamoto, M. Hagiya, M. Arita, and S. Yokoyama. 1997. Towards parallel evaluation and learning of Boolean μ-formulas with molecules. In *DNA Based Computers III*, eds. H. Rubin and D. H. Wood. pp. 57–72. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 48. Providence: American Mathematical Society Press.

Lagoudakis, M., and T. LaBean. 1999. 2D DNA self-assembly for satisfyability. In *DNA Based Computers V*, eds. E. Winfree and D. Gifford, 141–154. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 54. American Mathematical Society Press.

Landweber, L., and L. Kari. 1999. Universal molecular computation in ciliates. In *Evolution as Computation*. Eds. L. F. Landweber, E. Winfree. Berlin: Springer-Verlag.

Laun, E., and K. J. Reddy. 1997. Wet splicing systems. In *DNA Based Computers III*, eds. H. Rubin and D. H. Wood, pp. 73–84. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 48. Providence: American Mathematical Society Press.

Lipton, R., D. Faulhammer, A. Cukras, and L. Landweber. 1999. When the knight falls: On constructing an RNA computer. In *DNA Based Computers V*, eds. E. Winfree and D. Gifford, 1–8. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 54. American Mathematical Society Press.

Liu, Q., L. Wang, A. G. Frutos, A. E. Condon, R. M. Corn, and L. M. Smith. 2000. DNA computing on surfaces. *Nature* 403:175–179.

Lloyd, S. 2001. Ultimate physical limits to computation. *Nature* 406:1047–1054.

Martin-Vide, C., M. Margenstern, and G. Paun. 2001. Computing with membranes: Variants with an enhanced membrane handling. *Proc. Seventh Int. Workshop on DNA Based Computers*, Tampa, 53–62.

Paun, A. 2001. On P-systems with global rules. *Proc. Seventh Int. Workshop on DNA Based Computers*, Tampa, 43–52.

Paun, G. 1995. On the power of the splicing operation. *Int. J. Comput. Math.* 59:27–35.

Paun, G. 1996a. On the splicing operation. *Discrete Appl. Math.* 70:57–79.

Paun, G. 1996b. Regular extended H systems are computationally universal. *J. Automata Languages Combinatorics* 1(1):27–36.

Paun, G. 2000. Computing with membranes. *J. Comput. Syst. Sci.* 61(1):108–143.

Paun, G., G. Rozenberg, and A. Salomaa. 1996. Computing by splicing. *Theor. Comput. Sci.* 168(2):321–336.

Paun, G., G. Rozenberg, and A. Salomaa. 1998. *DNA Computing—New Computing Paradigms.* Berlin: Springer-Verlag.

Paun, G., and A. Salomaa. 1996. DNA computing based on the splicing operation. *Math. Jpn.* 43(3):607–632.

Petre, I., A. Ehrenfeucht, T. Harju, and G. Rozenberg. 2001. Patterns of micronuclear genes in cilliates. *Proc. Seventh Int. Workshop on DNA Based Computers*, Tampa, 33–42.

Pixton, D. 1995. Linear and circular splicing systems. *Proc. Intelligence Neural Biol. Syst.* 181–188.

Rothemund, P. 1996. A DNA and restriction enzyme implementation of Turing machines. In *DNA Based Computers*, eds. R. Lipton and E. Baum, pp. 75–120. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 27. Providence: American Mathematical Society Press.

Rozenberg, G., E. Winfree, and T. Eng., 2001. String tile models for DNA computing by self-assembly. In *DNA Computing*, eds. A. Condon and G. Rozenberg, pp. 63–88. Lecture Notes in Computer Science 2054. Berlin: Springer-Verlag.

Sakamoto, K., H. Gouzu, K. Komiya, D. Kiga, S. Yokoyama, T. Yokomori, and M. Hagiya. 2000. Molecular computation by DNA hairpin formation. *Science* 288:1223–1226.

Suyama, A., M. Arita, and M. Hagiya. 1997. A heuristic approach for Hamiltonian path problem with molecules. *Proc. Second Annu. Conf. Genetic Programming*, eds. J. R. Koza et al., 457–462.

Watson, J., N. Hopkins, J. W. Roberts, J. A. Steitz, and A. Weiner, 1987. *Molecular Biology of the Gene*, 4th ed. Menlo Park, CA: Benjamin Cummings.

Winfree, E. 1998a. Whiplash PCR for O(1) computing. *Proc. Fourth Int. Meet. DNA-Based Computers*, University of Pennsylvania, pp. 175–188.

Winfree, E. 1998b. *Algorithmic Self-Assembly of DNA*. Ph.D. thesis, California Institute of Technology.

Winfree, E., T. LaBean, and J. Reif. 1999. Experimental progress in computation by self-assembly of DNA tilings. In *DNA Based Computers V*, eds. E. Winfree and D. Gifford, pp. 123–140. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 54. American Mathematical Society Press.

Winfree, E., F. Liu, L. A. Wenzler, and N. C. Seeman. 1998. Design and self-assembly of two-dimensional DNA crystals. *Nature* 394:539–544.

Winfree, E., X. Yang, and N. Seeman. 1999. Universal self-assembly of DNA: Some theory and experiments. In *DNA Based Computers II*, 191–21. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 44. Providence: American Mathematical Society Press.

Yokomori, T. and S. Kobayashi. 1997. DNA-EC: A model of DNA-computing based on equality checking. In *DNA Based Computers III*, eds. H Rubin and D. H. Wood, pp. 347–360. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 48. Providence: American Mathematical Society Press.

Yokomori, T., S. Kobayashi, and C. Ferretti. 1997. On the power of circular splicing systems and DNA computability. *IEEE Int. Conf. Evolutionary Computing*, Indianapolis, 219–224.