Solving Intractable Problems with DNA Computing*

Richard Beigel and Bin Fu
Lehigh University

Abstract

We survey the theoretical use of DNA computing to solve intractable problems. We also discuss the relationship between problems in DNA computing and questions in complexity theory.

1. Introduction

Adleman's pioneering experiment [1] opened the possibility that moderately large instances of NP-complete problems might be solved via techniques from molecular biology. Since then numerous papers have explored more efficient molecular algorithms for particular problems in NP [27, 10, 3, 30, 8, 20, 21, 18], molecular solutions to PSPACE-complete problems [7, 37], and fault tolerant molecular algorithms [12, 25]. Other papers have examined the relationships between molecular complexity classes and classical complexity classes [38, 19]. We will survey some of these advances in this paper. For previous surveys in DNA computing, see [24, 36, 34, 32].

2. Biological Background

DNA is the storage medium for genetic information. It is composed of units called nucleotides, distinguished by the chemical group (base) attached to them. The four bases are adenine, guanine, cytosine, and thymine, abbreviated as A, G, C, and T. Single nucleotides are linked end-to-end to form DNA strands. Each DNA strand has two chemically distinguishable ends, called the $5'$ end and the $3'$ end respectively. Thus a DNA strand can be considered oriented; by convention a DNA strand is denoted by its sequence of bases (A, G, C, T), starting at the $5'$ end and finishing at the $3'$ end. Conversely, DNA strands can be used to encode strings over the alphabet $\{\mathrm{A}, \mathrm{G}, \mathrm{C}, \mathrm{T}\}$ or any other alphabet.

Mutual attraction, called hydrogen bonding, exists between As and Ts and between Gs and Cs. Hence, A/T and G/C are called complementary base pairs. For two complementary single strands, hydrogen bonding will only occur if the pair of complementary strands are oriented in an anti-parallel fashion. That is, as one strand runs in the $5' \to 3'$ direction, its complement runs in the $3' \to 5'$ direction. Two complementary single strands can pair up and twist around each other, forming the famous double helix structure discovered by Watson and Crick.

3. DNA Manipulations

Laboratory techniques for recombinant DNA and RNA manipulation are becoming highly standardized. Basic principles about recombinant DNA can be found in [48, 49]. In this section we informally describe some biological operations that form the basis for some of the computational operations we will formally define in Section 5. Some of our descriptions are similar to those in [10, 24, 35].

For a string x we denote by x the single DNA strand whose sequence is x and whose orientation is $5' \to 3'$, as x is read from left to right. $\overline{x}$ denotes the strand complementary to x. Finally, $\updownarrow x$ denotes the double strand that results when x and $\overline{x}$ anneal. For example if $x = 5'\mathrm{AGGCT}3'$, then $\overline{x} = 3'\mathrm{TCCGA}5'$ and $\updownarrow x$ is the double strand $\begin{matrix} 5'\mathrm{AGGCT}3' \\ 3'\mathrm{TCCGA}5' \end{matrix}$.

3.1. Merge

This is to pour the solution of two test tubes into a third one to achieve union. It can be performed by re-hydrating the tube contents and then combining the fluids together into a new tube by pouring.

3.2. Melt

This is to break double DNA strands into its single-stranded complementary components by heating the solution.

*Supported in part by the National Science Foundation under grants CCR-9796317 and CCR-9700417.
Address: Dept. of Electrical Engineering and Computer Science, 19 Memorial Dr W Ste 2, Bethlehem PA 18015-3084. Email: {beigel bif3}@eecs.lehigh.edu. Web: http://www.eecs.lehigh.edu/~{beigel bif3}.

3.3. Separation by Length

Using gel electrophoresis, we can approximately count the number of DNA sequences of each length. The molecules are placed at the top of a porous gel, to which an electric field is applied, drawing them to bottom. Larger molecules travel more slowly. After some time, the molecules spread into distinct bands according to size.

3.4. Separation by Subsequence

This is to extract all strands containing the sequence x. We use the method of biotin–avidin purification as described in [1]. We first create many copies of $\overline{x}$. To these oligos we attach biotin molecules, which are in turn anchored to an avidin bead matrix. We then melt the double strands in our test tube and pour them over this matrix. Those strings that contain x anneal to the $\overline{x}$ oligos anchored to the matrix. We wash away all strands that did not anneal, leaving behind only those strands that contain x, which can then retrieved from the matrix.

3.5. Amplification (PCR)

This is to make copies of DNA strands by using the polymerase chain reaction (PCR). If we have the duplex $\updownarrow xyz$, we first melt it to form xyz and $\overline{xyz}$. To this solution we add the primer oligos $\overline{z}$ and x, which anneal to form the partial duplexes $\begin{array}{c} xyz \\ \overline{z} \end{array}$ and $\begin{array}{c} x \\ \hline \overline{xyz} \end{array}$. DNA polymerase can then elongate the primer to create full duplexes of the form $\updownarrow xyz$. Now we have two copies of our original strand. Thus, if we can guarantee that the primer sequences that we use occur on the ends of every strand, and only on the ends, then we can use PCR to duplicate every strand in the test tube.

3.6. Append

This is to elongate every strand in a test tube T by tacking another short strand z onto the end. Suppose $z = D_1 D_2 \cdots D_k$, where each $D_i \in \{\mathrm{A}, \mathrm{G}, \mathrm{C}, \mathrm{T}\}$. A solution containing D_1 is poured into T, and each sequence d reacts with D_1 to form dD_1. After washing the excess D_1 solution away, one could have D_1 from the chain dD_1 coupled with D_2 to form dD_1D_2, and so on. Each character is appended in the $3'$ end (It will be not appended at $5'$-end). Each time only one character is appended. This can be guaranteed by the biological techniques that let the new character be in protected state. The protected state can be changed into unprotected state so that another character can be appended.

3.7. Anneal

This is to bond two single-stranded partially complementary DNA sequences by cooling the solution. $\alpha\sigma$ and $\overline{\sigma}\beta$ will be joined at $\sigma, \overline{\sigma}$ when two strands meet in the cooled solution.

3.8. Detect

This is to test if there is at least one DNA sequence in the solution of a test tube. It is usually done by amplification (see [41]).

4. Adleman's Algorithm for Hamiltonian Path

In this section we give a brief description of Adleman's [1] algorithm the Hamiltonian path problem. Similar descriptions can be also found in [45, 24]. A Hamiltonian path from s to t is a path that goes through each vertex exactly once. The input is a graph with vertices V and edges E. The output is yes if there is a Hamiltonian path from s to t; no, otherwise. A high-level description of Adleman's algorithm is given in Figure 1.

We now proceed to Adleman's biological implementation of his algorithm.

1. Choose at random n distinct single-stranded 20-character DNA sequences, and to each vertex v associate sequence S_v. For each such sequence obtain its reverse complement $\overline{S}_v$. Generate many copies of each $\overline{S}_v$ in test tube T_1.

If $(u, v) \in E$, then build sequence S_{uv} by concatenating the 10-base suffix of S_u to the 10-base of S_v. Generate many copies of each S_{uv} sequence in test tube T_2.

Pour T_1 and T_2 into T_3. We assume that in T_3 any binding between sequences can freely take place. Those bindings form all of the possible paths.

For a path $u_1 u_2 \cdots u_k$, a 10-base suffix of one S_{u_1} will bind to the 10-base prefix of one $S_{u_1 u_2}$ because one is complementary to the other; at the same time the 10-base suffix of the same $S_{u_1 u_2}$ binds to the 10-base prefix of one S_{u_2} sequence whose 10-base suffix binds to the 10-base of one $S_{u_2 u_3}$, and so on (see Section 3.7).

1. generate random paths through the graph
2. delete all paths that do not begin with s
   delete all paths that do not end with t
3. delete all paths of length n or greater
4. for each vertex $v \in V$
   delete all paths that do not enter v
5. if any paths remain then return "yes" else return "no"

Figure 1: Adleman's Algorithm for Hamiltonian Path

2. Amplify the solution by the polymerase chain reaction (see Section 3.5). In the process, those molecules encoding paths that begin with s and ended with t will be amplified. The primers are S_u and $\overline{S}_v$.

3. Separate the double-stranded DNA having exactly $20n$ bases, which therefore represent paths with exactly n vertices. Apply the gel electrophoresis techniques of Section 3.3 to separate DNA sequences by length.

4. For each v in V, use affinity purification to eliminate all strands not containing S_v (see Section 3.4).

5. Amplify the solution by the polymerase chain reaction (see Section 3.5). Output "yes" if the solution contains at least one DNA sequence; no, otherwise.

5. Molecular Computation Models

The models we define were inspired by the work of [3, 38]. A molecular sequence is a string over an alphabet Σ (we can use any alphabet we like, encoding characters of Σ by finite sequences of base pairs). A test tube is a multiset of molecular sequences. We describe the allowable operations formally below. Where set notation is applied to multisets, multiplicities are respected. In the definitions, T_1, T_2, and T_3 denote distinct test tubes and c denotes a character.

The running time for a molecular algorithm is proportional to the number of operations on test tubes. An important complexity measure is the "solution volume size," *i.e.*, the maximum number of strings in a tube, counting multiplicities. Adleman [2] has speculated that molecular computation with a solution volume of size 2^{70} might be possible.

5.1. Encoding

Let $x = x_1 \cdots x_n$ be an n-bit binary string. Assign a unique sequence of 30 bases to each bit position and bit value. Sequence $B_i(0)$ encodes the ith bit of x as 0. Sequence $B_i(1)$ encodes the ith bit of x as 1. $x = x_1 \cdots x_n$ is represented by the following DNA sequence:

$$B_1(x_1)B_2(x_2)\cdots B_n(x_n).$$

We require that for any $\langle i,a\rangle \neq \langle j,b\rangle$, $B_i(a)$ and $B_j(b)$ have no long common subsequence.

5.2. Operations

We list some operations that we will study in the next three chapters. We will introduce some other operations in Chapters 6 and 7.

Sep(T_1, c, T_2, T_3)
  $T_2 :=$ the multiset of all strings in T_1 that contain the character c anywhere;
  $T_3 :=$ the multiset of all strings in T_1 that do not contain the character c
  $T_1 := \emptyset$.

Separate(T_1, c, i, T_2, T_3)
  $T_2 :=$ the multi-set of all strings in T_1 whose ith character is c;
  $T_3 :=$ the multi-set of all strings in T_1 whose ith character is not c or whose length is less than i;
  $T_1 := \emptyset$.

Append(T, c)
  $T := \{xc : x \in T\}$.

Merge(T_1, T_2, T_3)
  $T_3 := T_1 \cup T_2$;
  $T_1 := \emptyset$;
  $T_2 := \emptyset$.

Split(T_1, T_2, T_3)
  T_2 and T_3 are obtained by halving the multiplicities of all strings in T_1, which are assumed to be even;
  $T_1 := \emptyset$.

Amplify(T_1, T_2, T_3)
$T_2 := T_1$;
$T_3 := T_1$;
$T_1 := \emptyset$.

Test(T)
If T is not empty then "true" else "false".

Init(T, m)
$T := \{i : 1 \leq i \in m\}$.

Merge, Append, and Test can be implemented by the methods of Sections 3.1, 3.6, and 3.8 respectively. Sep can be implemented as "separation by sequence," which is described in Section 3.4. Note the similarity between Sep and Separate. Of the two, the Sep operation is easier to implement physically and is standard in molecular computation models. Note, however, that Separate can simulate Sep in polynomial time with no volume overhead. More important, it is well known that Sep can simulate Separate with no time or volume overhead if we use polynomial-size alphabets (to be specific, we can represent each binary string $b_1 \cdots b_n$ over the alphabet Σ by the sequence $\langle 1, b_1\rangle \cdots \langle n, b_n\rangle$ over the alphabet $\{1, \ldots, m\} \times \Sigma$ where m is a suitable bound on the length of any string generated during the computation). For convenience, our models will all contain the Separate operation, rather than Sep.

The molecular operation performed at each step is determined by the input string x. Take the 3-SAT problem for example. In the simplest molecular algorithm, the operations to be performed are determined by examining each clause one at a time. The operations corresponding to a particular clause separate out the strands that do not satisfy that clause. We consider only uniform algorithms, in which the sequence of DNA operations to be performed is a polynomial-time computable function (in the classical sense) of the input.

Definition 1.

- Let Q be a set of DNA operations. A Q-molecular algorithm is an algorithm that only permits operations in Q.
- The *Time* of a molecular algorithm is the number of operations.
- The *Volume* of a molecular algorithm is the number of DNA sequences.
- A molecular algorithm is $t(n)$-*time uniform* if there is a $t(n)$-time Turing machine T such that for any input, operations can be generated by T.
- A MOL* computation permits Separate, Append, and Merge operations. The operation sequences can be generated by a Turing machine in polynomial steps. MOL$^*(t(n), v(n))$ is the class of languages accepted by such a computation where the running time is $O(t(n))$.
- $\mathrm{MOL}(v(n)) = \bigcup_{k=1}^{\infty} \mathrm{MOL}^*(n^k + k, v(n))$.
- In the MOL-A$(s(n))$ model, T_0 is initialized to contain a single copy of the empty string, the Separate, Append, Merge, and Amplify operations are permitted, and the total number of strings in all tubes must be at most $s(n)$ at all times. All tubes other than T_0 are initialized to be empty. The molecular algorithm *accepts* if T_0 is nonempty after the last molecular operation is performed.

6. Lipton's Algorithm for SAT

Motivated by Adleman's pioneer work, Lipton designed a simple molecular algorithm for solving the SAT problem. His algorithms starts with all assignments in tube T_0. For each clause, it weeds out assignments that do not satisfy that clause. We illustrate it via the example in Figure 2, which is from [27].

Theorem 2 (Lipton). *There is an $\{$Init,Separate,Merge,Test$\}$-molecular algorithm for* SAT *that runs in time $O(m)$ and volume 2^n, where n is the number of variables and m is the length of the formula.*

7. Rooß and Wagner's Characterization

Rooß and Wagner characterized the power of Lipton's polynomial-time molecular algorithms, which are allowed the operations Init, Separate, Merge, and Test. They proved that those algorithms recognize exactly the class P^{NP}.

Definition 3. Let DNA(Init, Sep, Merge, Test)-P be the class of languages that can be decided in polynomial-time with $\{$Init, Sep, Merge, Test$\}$-molecular algorithms.

MOL(4) Algorithm	T_0	T_1	T_2	T_3	T_4
Initialization	$00, 01, 10, 11$				
Separate$(T_0, 1, 1, T_1, T_2)$		$10, 11$	$00, 01$		
Separate$(T_2, 1, 2, T_2, T_3)$		$10, 11$	01	00	
Merge(T_1, T_2, T_2)			$10, 11, 01$	00	
Separate$(T_2, 0, 1, T_1, T_2)$		01	$10, 11$	00	
Separate$(T_2, 0, 2, T_2, T_4)$		01	10	00	11
Merge(T_1, T_2, T_0)	$01, 10$			00	11

Figure 2: Lipton's SAT algorithm. In this example, we find all satisfying assignments to the formula $(x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$.

If $L \in \mathrm{P}^{\mathrm{NP}}$, there is a polynomial time oracle Turing machine M such that M^{SAT} recognizes L. By Lipton's result, each query to SAT can be answered in polynomial time by molecular computation. Thus the entire membership test can be performed in polynomial time by molecular computation.

Conversely, Rooß and Wagner proved that DNA$(\{\mathrm{Init}, \mathrm{Sep}, \mathrm{Merge}, \mathrm{Test}\})$-P is contained in P^{NP}. Thus the two classes are equal:

Theorem 4 (Rooß, Wagner).

$$\mathrm{P}^{\mathrm{NP}} = \mathrm{DNA}(\{\mathrm{Init}, \mathrm{Sep}, \mathrm{Merge}, \mathrm{Test}\})\text{-P}.$$

8. Simulation of Circuits and Turing Machines

One method to develop DNA efficient algorithms is by simulating efficient classical algorithms. This method also implies relations among DNA complexity classes and classical complexity classes. In this section we will discuss how various researchers have simulated Boolean circuits and Turing machines.

8.1. Simulation of Boolean Circuits on Many Inputs in Parallel

Boneh, Dunworth and Lipton [11] showed how to simulate a Boolean circuit on a large collection of inputs, thus solving the satisfiability problem for Boolean circuits. The following lemma follows the exposition in [8]. To simulate the circuit on a single input, we initialize one strand to that input, and we simulate one gate at a time, working towards the output, appending the result each gate to the strand as we go. That process can be performed in parallel on a large collection of inputs.

Lemma 5.
Let π be a circuit with m gates. Given a tube T, a molecular algorithm using only the operations Separate, Append, and Merge, running in time $O(m)$, and using only four test tubes can create tubes T_0 and T_0 such that T_0 contains all strings z from tube T that satisfy $\pi(z) = 0$ and T_0 contains all strings z from tube T that satisfy $\pi(z) = 1$.

Proof: Let π's input gates be $g_1, \ldots, g_n$ and internal gates be $g_{n+1}, \ldots, g_m$ in topological order; in particular g_m is the output gate. We will use four tubes T, T_0, T_1, T_2. For each i, let g_i compute $f_i(g_{j(i)}, g_{k(i)})$ where $j(i) < i$, $k(i) < i$, and f_i is a binary function. We perform the following algorithm:

```
for i := n + 1 to m do
    Separate(T, 0, j(i), T1, T0)
    Separate(T1, 0, k(i), T, T2)
    Append(T, fi(0, 0))
    Append(T2, fi(0, 1))
    Merge(T, T2, T1)
    Separate(T0, 0, k(i), T, T2)
    Append(T, fi(1, 0))
    Append(T2, fi(1, 1))
    Merge(T, T2, T0)
    Merge(T1, T0, T)
Separate(T, 1, m, T1, T0)
```

At completion, T_1 contains all strings that satisfy π and T_0 contains all strings that do not satisfy π. ∎

8.2. Faster Simulation of Boolean Circuits on One Input

Ogihara and Ray [31] designed an algorithm to simulate deterministic boolean circuit (with one input), which consists of unbounded fan-in OR gates, 2-fanin AND gates and $2n$ input gates $x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n$. They called such a circuit as semi-unbounded fan-in circuit. They showed that for a semi-unbounded fan-in circuit with depth d, size m, and maximum fan-out F, it can be simulated by DNA computing in $d \log F + O(d)$ steps.

In their algorithm, the circuit is simulated level by level. For each gate g_i, choose a DNA strand δ_i with length L. For the input level (the 0th level), pour δ_i if gate g_i evaluates to 1.

Suppose we have simulated the i-th level such that for every gate g_k in i-th level, δ_k is in the solution of tube T if g_k evaluates to 1.

For an OR level, amplify T so that each DNA strand has at least Fp copies (where p is a parameter). For each gate g_j at the $(i+1)$st level, pour δ_j into the solution. If g_i is an input to gate g_j, then a linker is poured. (For $g_i = uv$ and $g_j = xy$, a linker is $\overline{vx}$. In their implementation, they take $|u| = |v|$ and $|x| = |y|$.) If g_i has an input with value 1, the linker can link δ_j to δ_i. Separate all of the DNA strand of length $2L$. Cleave the length $2L$ strands by restriction enzyme. Thus, Remove all of the δ_i and get δ_j.

For an AND level, amplify T so that each DNA strand has at least Fp copies. For each gate g_j at the $(i+1)$st level, pour δ_j into the solution. If gate g_j has inputs δ_{i_1} and δ_{i_2}, pour linker for $\delta_{i_1}, \delta_{i_2}$ and δ_j. Separate all of the DNA strands with length $3L$. Remove δ_{i_1} and δ_{i_2} to get δ_j.

8.3. Simulation of Turing Machines

Beaver [6] and Rothemund [39] showed how to simulate a time-bounded Turing machine (with many inputs). In the following we give a brief description about Beaver's simulation.

The configuration of the Turing machine is encoded as a DNA sequence. The configuration of a Turing machine is determined by the symbols on its tape, its state and head position. If the symbol sequence is $a_{1 \ldots} a_m$, the state is q and the head position is i. The configuration can be characterized by $a_{1 \ldots} a_{i-1} q a_i \cdots a_m$. This sequence can be encoded as a DNA strand

$$(1, a_1) \cdots E(i-1, a_{i-1}) E(i, q) E(i, a_i) \cdots E(m, a_m).$$

In every step, a Turing machine can move its head at most one cell and can change at most one symbol on the tape. The next configuration is almost the same as the current, except that $a_{i-1} q a_i$ is replaced by another three bits. Thus the problem of Turing machine reduces to how to do character substitution in DNA strands.

Let $ABCDE$ be a DNA strand. We would like to replace C by F. We create $\overline{BFD}$ and let it anneal $ABCDE$ bonded between B and $\overline{B}$, D and $\overline{D}$. Adding $\overline{E}$ as the start point and polymerase the molecules. $\overline{A}$ is created and link to A. The double chain is created that is the combination of $ABCDE$ and $\overline{ABFDE}$. Change double strands into single strands and remove $ABCDE$. We get $\overline{ABFDE}$ left. One more PCR, we can get $ABFDE$.

The simulation can be adapted to nondeterministic Turing machine. A path of a nondeterministic Turing machine can be encoded as a $0,1$-sequence. The computation of nondeterministic Turing machine is deterministic at a fixed path.

Reif [35] studied how to use DNA operations to simulate parallel computation. In his DNA computing model, it permits operations Merge, Separate and Test. In addition, it also permits the operation that provides for the combination of all pairs of DNA sequences with subsequences that have a complementary match. Let $E(\alpha)$ be the encoding of α and let $E(\alpha, \beta)$ be the coding of tuple (α, β). PA-Match is defined as

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

A CREW machine is a parallel machine with a large shared memory and allows concurrent reads and exclusive writes. Given a CREW machine with time bound D, M memory cells, and P processors, it can be simulated by his model with $O(D+s)$ PA-Match operations and $O(s \log s + D)$ other operations, where $s = O(\log PM)$.

8.4. Bounded Nondeterminism

NP computation with a limited amount of nondeterminism was introduced in [26] and studied further in [14, 16, 33, 13, 22, 42, 23, 9].

Definition 6.

- An NPinit$(s(n))$ Turing machine is an NP TM that nondeterministically chooses a number between 1 and $s(n)$, then proceeds deterministically.
- An NPpaths$(s(n))$ Turing machine is an NP TM with at most $s(n)$ paths on any length-n input.
- NPinit$(s(n))$ is the class of languages accepted by NPinit$(s(n))$ machines.
- NPpaths$(s(n))$ is the class of languages accepted by NPpaths$(s(n))$ machines.

Definition 7.

- For $n > 0$, define $\mathrm{pad}_n(x) : \{0,1\}^{\leq n} \to \{0,1\}^{2n}$ by
$$\mathrm{pad}_n(x_1 x_2 \cdots x_k) = 0x_1 0x_2 \cdots 0x_k 1^{2n-2k}.$$
- A family of circuits $\{C_n\}_{n=1}^{\infty}$ is P-uniform if there is a polynomial time computable function f such that $f(0^n) = C_n$ for all n.
- A function f is P-uniform $s(n)$-size computable if there is a P-uniform family of circuits $\{C_n\}_{n=1}^{\infty}$ such that (1) each C_n is of size $\leq s(n)$; and (2) $C_n(\mathrm{pad}_n(x)) = f(x)$ for all x with length $\leq n$.
- $L \in \mathrm{NPinit}(s(n), v(n))$ if there is a P-uniform family of circuits $\{C_n\}_{n=1}^{\infty}$ such that C_n is of size $\leq v(n)$ and $x \in L$ iff $C_{|x|}(x, i) = 1$ for some $1 \leq i \leq v(n)$.

It is easy to see that $\mathrm{NPinit}(v(n)) = \bigcup_{k=1}^{\infty} \mathrm{NPinit}(n^k + k, v(n))$.

We show how to simulate bounded nondeterministic computation via bounded-volume molecular computation. Results of this type appear in [7, 38, 39, 46], but they assume models of molecular computation with more powerful operations, such as Amplify, that may be harder to implement in practice.

Theorem 8 (Beigel, Fu). $\mathrm{NPinit}(s(n), v(n)) \subseteq \mathrm{MOL}^*(s(n), v(n))$.

Proof: Let $L \in \mathrm{NPinit}(s(n), v(n))$ via a P-uniform family of circuits $\{C_n\}_{n=1}^{\infty}$ of size $s(n)$ such that each C_n is of size $\leq s(n)$, and $x \in L$ iff $C_{|x|}(x, i) = 1$ for some $1 \leq i \leq v(|x|)$. For each x, fix circuit $C'_x(i) = C_{|x|}(x, i)$. Applying Lemma 5, we get a molecular algorithm to determine if there is a $1 \leq i \leq v(|x|)$ with $C'_x(i) = 1$, and the algorithm uses no more than $s(|x|)$ operations. Since $\{C_n\}_{n=1}^{\infty}$ is P-uniform, the molecular operations sequences can be also computed in polynomial time. ▮

Corollary 9 (Beigel, Fu). $\mathrm{NPinit}(v(n)) \subseteq \mathrm{MOL}(v(n))$.

Furthermore, Fu and Beigel [19] proved

- $\mathrm{MOL}'(s(n)) = \mathrm{MOL}(s(n)) = \mathrm{NPinit}(s(n))$
- $\mathrm{MOL\text{-}A}(s(n)) = \mathrm{NPpaths}(s(n))$

Molecular computation suggests an interesting open problem in complexity theory:

$$\mathrm{NPinit}(s(n)) \stackrel{?}{=} \mathrm{NPpaths}(s(n))$$

In [19], we constructed an oracle that separates those classes.

8.4.1. Breaking DES

Boneh, Dunworth, and Lipton [10] proposed to crack the DES encryption scheme via DNA computing. Because DES uses 56-bit keys, they can encode each possible key as a DNA strand and then try all of them in parallel.

9. Volume-Efficient Algorithms

Although DNA computing has the advantage of huge parallelism, its power is limited by the volume of DNA that can be manipulated in the lab. Assuming a maximum volume of 2^{70}, a brute-force 2^n-volume algorithm like Lipton's solves only instances of size 70 or less. An $n!$ algorithm like Adleman's solves only instances of size 22 or less. We have to decrease volume complexity in order to solve large problem instances. In this section we survey research on volume-efficient solutions solving NP-complete problems.

9.1. Bach et al's Approach

Bach, Condon, Glaser, and Tanguay [3] made the first efforts to find nonexhaustive molecular algorithms for NP-complete problems.

The *3-coloring* problem is to determine if the vertices of a graph can be colored red, green, and blue so that no adjacent nodes have the same color. If exhaustive search is used, 3^n cases would be considered in determining if a graph is 3-colorable. Bach et al consider the following nonexhaustive algorithm.

If we are given the set H of all nodes in a graph G that have one particular color, then we just need to determine whether $G - H$ can be colored with the other two colors, i.e, whether $G - H$ is bipartite, which can be determined in $O(n)$ time. In fact, it can be determined by a circuit of size $O(n^2)$.

If a graph is 3-colorable, there is a color, say red, such that the number of nodes colored red is no more than $n/3$. The total number of subsets with size $n/3$ is

$$\sum_{j=0}^{n/3} \binom{n}{j} \leq 1.89^n.$$

Their molecular algorithm generates $\sum_{j=0}^{n/3} \binom{n}{j}$ DNA sequences, which encode every set H consisting of at most $n/3$ vertices. For each sequence, the algorithm checks whether the graph $G - H$ is bipartite. Since we know how to simulate a circuit, this can be done in n^2 steps.

Theorem 10 (Bach et al). *There is an $n1.89^n$ volume, $O(n^2 + m^2)$ time molecular algorithm for the 3-coloring problem.*

They also designed a molecular algorithm for the independent set problem.

Theorem 11 (Bach et al). *There is a 1.51^n volume, $O(n^2 m^2)$ time molecular algorithm for the independent set problem.*

9.2. Ogihara's Approach

Ogihara [30] designed a breadth-first molecular algorithm for 3-SAT, based on Monien and Speckenmeyer's classical algorithm [29]. The 3-SAT problem is to determine whether a set of clauses, each consisting of at most three literals, is satisfiable.

For clarity, we will first present the 3-SAT algorithm of Monien and Speckenmeyer. Let $f|_\ell$ denote the formula obtained from f by replacing the literal ℓ by true and the literal $\overline{\ell}$ by false. A k-clause is a disjunction of exactly k literals.

The last case in the recursion is ostensibly the worst, yielding two subproblems of size $n-1$, but it only occurs on the first call or immediately after eliminating a single variable, which yields a single subproblem of size $n-1$; unrolling the recursion, we see that the last case gives two subproblems of size $n-2$. The worst case is the second to the last, which yields subproblems of size $n-1$ and $n-2$. Thus the number of leaves in the self-reduction is at most $2f(n)$ where $f(n)$ is given by the recurrence $f(n) = f(n-1) + f(n-2)$; in particular $2f(n) \leq 1.62^n$ for almost all n.

Based on Monien and Speckenmeyer's Algorithm, Ogihara [30] designed a molecular implementation that uses the Amplify, Merge, Append, and Separate operations. His implementation is essentially a breadth first search. DNA strands encode the partial assignments. At the beginning of the computation, tube T contains only empty assignments. The computation proceeds in n identical stages, which we describe below, each taking $O(\max(m^2, n))$ steps.

At the beginning of each stage, T is divided into $3n$ tubes $\mathrm{pos}_1, \ldots, \mathrm{pos}_n$, $\mathrm{neg}_1, \ldots, \mathrm{neg}_n$, $\mathrm{two}_1, \ldots, \mathrm{two}_n$, where pos_i (resp. neg_i) consists all strands to be extended with t_i (resp. f_i) and two_i consists of those to be extended both ways. This partition can be achieved via Separate and Merge operations. The partial solutions in tube pos_i or neg_i can be extended via the Separate, Merge, and Append operations. The contents of tube two_i are duplicated via the Amplify operation; then its partial solutions can be extended separately. At the end of each stage, all tubes are Merged into tube T.

Theorem 12 (Ogihara). *There is a {Separate,Amplify,Merge,Append}-molecular algorithm for 3SAT that runs in $O(n \max(m^2, n))$ time and uses 1.62^n volume, where n is the number of variables and m is the number of clauses.*

9.3. Implementing Recursion with Bounded Nondeterminism

In [8], Beigel and Fu took systematic study for transforming classical algorithms into molecular algorithms. They derived a general result that indicates a large class of recursive algorithms can be transformed into molecular algorithms.

Schnorr [43], Meyer and Paterson [28] introduced self-reduction. Recursive algorithms for NP problems often take the form of d-self-reductions ("d" for disjunctive), which is a special case of self-reduction and were defined by Selman [44].

Definition 13. A partial order $\prec$ is $\langle d(n), b(n) \rangle$-well-founded if

- $y_d \prec \cdots \prec y_1 \Rightarrow d \leq d(|y_1|)$; and
- $y_d \prec \cdots \prec y_1 \Rightarrow |y_d| \leq b(|y_1|)$

For technical simplicity we will consider only languages L containing the empty string, Λ.

Definition 14.
- A $\langle d(n), b(n), s(n), w(n) \rangle$-d-self-reduction for a language L consists of a P-uniform $s(n)$-size computable function $h(x) = \{x_1, \ldots, x_m\}$ and

```
function 3SAT(f)
if f is the empty set of clauses then return true
else if f contains an empty clause then return false
else if some variable v appears only in positive literals then return 3SAT(f|_v)
else if some variable v appears only in negative literals then return 3SAT(f|_{\overline{v}})
else if f contains a clause C consisting of a single literal ℓ then return 3SAT(f|_ℓ)
else if f contains a clause C consisting of two literals ℓ_1, ℓ_2 then
     return 3SAT(f|_{ℓ_1}) ∨ 3SAT(f|_{\overline{ℓ_1}}|_{ℓ_2})
else
     let v be the first variable to appear in f
     return 3SAT(f|_v) ∨ 3SAT(f|_{\overline{v}})
```

Figure 3: Monien and Speckenmeyer's 3-SAT algorithm

a $\langle d(n), b(n)\rangle$-well-founded partial order $\prec$ on problem instances such that

- Λ is the only minimal element under $\prec$
- for all $x \neq \Lambda$, $x \in L \iff h(x) \cap L \neq \emptyset$
- for all x, $x_i \in h(x) \Rightarrow x_i \prec x$.
- $||h(y)|| \leq w(|x|)$ for any node y in H_x, where H_x is the least set such that $x \in H_x$ and $(y \in H_x \Rightarrow h(y) \subseteq H_x)$.

• A d-self-reduction is a $\langle d(n), b(n), s(n), w(n)\rangle$-d-self-reduction for some polynomials $d(n), b(n), s(n), w(n)$.

Definition 15. Let $\langle h, \prec\rangle$ be a $\langle d(n), b(n), s(n), w(n)\rangle$-d-self-reduction and let x be a problem instance.

- $T_{h,\prec}(x)$ is the unordered rooted tree that satisfies the following rules: (1) the root is x; (2) for each y, the set of children of y is $h(y)$.
- $|T_{h,\prec}(x)|$ is the number of leaves in $T_{h,\prec}(x)$.

If $\langle h, \prec\rangle$ is a d-self-reduction for L, then the corresponding recursive algorithm for L runs in time $|T_{h,\prec}(x)|$. The analysis of such an algorithm usually provides a bound on $|T_{h,\prec}(x)|$ that is suitable for use in constructing a molecular algorithm for L. We formalize this below:

Definition 16.
• A language L belongs to $\text{REC}^*(d(n), b(n), s(n), w(n), T(x))$ if there exists a $\langle d(n), b(n), s(n), w(n)\rangle$-d-self-reduction $\langle h, \prec\rangle$ for L such that for all x

(1) $|T_{h,\prec}(x)| \leq T(x)$, and

(2) $T(x) \geq \sum_{x_i \in h(x)} T(x_i)$.

• $\text{REC}(T(x)) = \bigcup_{k=1}^{\infty} \text{REC}^*(n^k + k, n^k + k, n^k + k, n^k + k, T(x))$.

Lest conditions (1) and (2) above seem restrictive, we argue that they are quite natural. We consider the typical analysis of a recursive algorithm. One introduces a function T and proves by induction on x that $|T_{h,\prec}(x)| \leq T(x)$, which is (1). The inductive hypothesis is that $|T_{h,\prec}(w)| \leq T(w)$ if $w \prec x$. Inspection of the algorithm yields

$$|T_{h,\prec}(x)| = \sum_{x_i \in h(x)} T_{h,\prec}(x_i) \leq \sum_{x_i \in h(x)} T(x_i),$$

by the inductive hypothesis. The last step in the induction consists of showing that T satisfies $\sum_{x_i \in h(x)} T(x_i) \leq T(x)$, which is (2).

The function T above depends on problem instances rather than their size because the analysis of the algorithm may depend on two or more parameters. We will need an analogous variant of NPinit().

Definition 17. $\text{NPinit}'(s(n), V(x))$ consists of languages L recognized by a P-uniform family of circuits $\{C_n(x, i)\}_{n=1}^{\infty}$ such that each C_n is of size $\leq s(n)$, and $x \in L$ iff $C_{|x|}(x, i) = 1$ for some $1 \leq i \leq V(x)$.

Clearly, if $V(x) \leq v(|x|)$ then $\text{NPinit}'(s(n), V(x)) \subseteq \text{NPinit}(s(n), v(n))$.

Theorem 18 (Beigel, Fu).
If $T(x)$ is P*-uniform $s'(n)$-size computable, then* $\mathrm{REC}^*(d(n), b(n), s(n), w(n), T(x)) \subseteq \mathrm{NPinit}'(t'(n), T(x))$, *where* $t'(n) = O(d(n)(s(b(n)) + w(n)s'(b(n))))$.

Corollary 19. $\mathrm{REC}(T(x)) \subseteq \mathrm{NPinit}(T(x))$.

This result indicates that many complicated recursive algorithms, including Monien and Speckenmeyer's Algorithm, for NP-complete problem can be implemented via molecular computation.

Theorem 20 (Beigel, Fu). *There is a {Separate,Init,Merge,Append}-molecular algorithm for 3*SAT *that runs in $O(nm)$ time and uses 1.62^n volume, where n is the number of variables and m is the number of clauses.*

10. Algorithms for #P-complete Problems

Definition 21.

- A function $f : \Sigma^* \longrightarrow \mathsf{N}$ is in the class #P if there is a polynomial-time computable predicate $p(x, y)$ and a polynomial $q(n)$ such that $f(x) = |\{y : y \in \Sigma^{q(n)}$ and $p(x, y) = 1\}|$.
- Language $L \in$ PP if there is a polynomial-time nondeterministic Turing machine M such that $x \in L$ iff the number of accepting paths of $M(x)$ is more than the number of rejecting paths of $M(x)$.

#SAT is the function such that #SAT(f) is the number of assignments satisfying boolean formula f. #Hamiltonian is the function such that #Hamiltonian(G, s, t) is the number of Hamiltonian paths from s to t in graph G. Both #SAT and #Hamiltonian are #P-complete.

Toda [47] showed that polynomial time hierarchy is polynomial time Turing reducible to #SAT. Thus, #P is widely believed far above NP.

Computing #P-hard problem with DNA computing was studied by Fu, Beigel and Zhou [21]. A #P language is polynomial-time Turing reducible to a language in PP. Let's consider how to deal with a language in PP with DNA computing. For a language L in PP and a string x, there is a polynomial size circuit $C(i)$ such that $x \in L$ iff $|\{i : i \in \Sigma^{p(|x|)}$ and $C(i) = 1| > |\{i : i \in \Sigma^{p(|x|)}$ and $C(i) = 0|$.

Suppose tube T contains all sequences in $A = \Sigma^{p(|x|)}$. We use the method in Section 8.1 to get two tubes T_1 and T_2 such that T_1 contains all sequences A with $C(i) = 1$ and T_1 contains all sequences A with $C(i) = 0$. Clearly, $x \in L$ iff T_1 contains more sequences that T_2.

We use the following method to compare the two tubes. The idea can be characterized from the following simple example. There are two groups of people. One group consists a large number of men and the other group consists a large number of women. We need to determine which group has more people.

We merge the two groups. Let each man find a woman randomly as his wife. Separate all of those couples. We can determine if the number of men is greater than women from those persons left.

We use the same idea to compare the number of DNA sequences in two tubes T_1, T_2. Append a DNA sequence s to T_1 and the reverse of its complement $\overline{s}^\wedge$ to T_2. Merge the two tubes. Sequences of T_1 will bind to sequences of T_2 via s and $\overline{s}^\wedge$ for the mutual attraction between them. This looks like a man finding a woman as his wife randomly. Separate all double sequences. We know which tube has more sequences from those sequences left.

Bax [5] gave an $O(2^n)$-time algorithm for computing #Hamiltonian, where n is the number of vertices in the graph. Let $G = (V, E)$ be a graph and let s and t be two nodes in G. Let A denote the adjacency matrix of G, *i.e.*,

$$A(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

For $S \subseteq V$, let W_S be the number of paths of length $n - 1$ that do not enter any node in S and let A_S be the matrix such that

$$A_S(u, v) = \begin{cases} A(u, v) & \text{if } u \notin S \text{ and } v \notin S \\ 0 & \text{otherwise.} \end{cases}$$

Then $W_S = A_S^{n-1}(s, t)$. By the principal of inclusion and exclusion, the number of Hamiltonian paths from s to t is equal to $\sum_{S \subseteq V} (-1)^{|S|} W_S$.

In [21], the above formula is evaluated by computing a polynomial number of #P functions; each time only 2^n DNA strands are used. Thus they obtained an $n^{O(1)}2^n$ volume molecular algorithm for computing #Hamiltonian.

11. Algorithms for PSPACE-complete Problems

Rooß and Wagner [37], and Beaver [7] showed how to solve a PSPACE-hard problem with DNA computing in a polynomial number steps. The following is Rooß and Wagner's algorithm for testing membership in QBF, which is the set of all true formulas of the form $\exists x_1 \forall x_2 \cdots Q x_n H(x_1, \ldots, x_n) = 1)$, where Q is $\exists$ if n is odd, and $\forall$ otherwise.

We define the operation $\text{Rightcut}(T) = \{x : xa \in T\}$. Its biological implementation is similar to Ogihara and Ray's Cleave operation (see Section 8.2). The intersection of two tubes is the set of DNA strands that are in both tubes. Its biological implementation is similar to Reif's PA-Match (see Section 8.3). Rooß and Wagner's algorithm is described below:

$T := \{0,1\}^n$;
for $i := n$ downto 1 do
    Separate$(T, i, 0, T_0, T_1)$;
    Rightcut(T_0);
    Rightcut(T_1);
    if i is even then $T := T_1 \cap T_2$
    else Merge(T_0, T_1, T);
Test(T);

Intersection operation is impractical. As the number of DNA strands in a test tube is large, the chance for a strand to meet another strand which is the same as it is very small. By the same reason, PA-Match in section 8.3 is also impractical.

12. Fault Tolerance

Many kinds of errors can occur in DNA computing. Even a straightforward operation like Merge may have errors, because DNA strands remaining on the walls of tubes may be lost. Experimental evidence shows that PCR may result in complex structures involving several DNA strands as well as incorrect ligation. In this section, we will discuss two methods that deal with misclassification errors in the Separate operation.

12.1. Boneh and Lipton's Method

Boneh and Lipton noted that Adleman's Hamiltonian path algorithm and Lipton's SAT algorithm start with a space of potential solutions that they winnow down until it contains only actual solutions. They showed [12] how to combat computational errors in such algorithms.

A strand is called *good* if it encodes a solution for the input problem instance. Boneh and Lipton considered two error probabilities: the probability P_s that some good strand is in the final test tube, and the probability P_r that at least δ of the strands in the final tube are good. In both Adleman and Lipton's algorithms, as well as many other molecular algorithms for solving NP-complete problems, the volume goes down as many DNA strands are thrown away. Such algorithms are called volume decreasing. They suggested to transform a volume-decreasing computation into a constant-volume computation. They augment the volume-decreasing computation with some Amplify operations. This decreases the chance of losing DNA strands. At the end, they repeatedly select strands from the final tube to test if they are solutions. If a large fraction of DNA strands in the final tube are in fact solutions, their method succeeds with high probability. The probability that they fail to find a solution (when one exists) is at most $(1 - \text{P}_s) + \text{P}_s(1 - \text{P}_r) + \text{P}_s\text{P}_r(1 - \delta)^m$, where m is the number of strands they select from the final tube.

A method for determining P_s and P_r is described in [12]. For the operation $\text{Sep}(T, s, T_1, T_2)$, let p be the probability that a strand containing the subsequence s is incorrectly placed into T_2, and let q be the probability that a strand without subsequence s is incorrectly placed into to T_1. In practice, we typically have $p \leq 0.05$ and $q \leq 10^{-6}$. For a bad strand x, let $M(x)$ be the number of separations for which it does not match sequences (In many molecular algorithms for solving NP-complete problems, good strands in T always match s and go to T_0 in Separate(T, s, T_0, T_1) operations).

Assume there are initially 2^n DNA strands in the beginning, the algorithm runs for s steps, and the volume decreases by a factor 2 every s/n steps. After every s/n steps, the DNA solution is amplified. If a good strand survives after s/n steps, then it becomes 2 good strands, which increase the chance that some good strands survive in future. Using branch process theory [17], the extinction probability (for good strands) is the root x in the equation $1 - r + rx^2 = x$, where $r = p^{s/n}$. Therefore, $\text{P}_s = 1 - x = 2 - p^{-s/n}$.

They showed P_r is large by giving some evidence that the ratio of good strands to bad strands is not too small. Assume there is at least one good strand. Let M_k be the number of bad strands that have $M(x) = k$. The expected number of bad strands that survive is $e = M_1 q + M_2 q^2 + M_3 q^3 + \cdots$. They claimed that $l = M_1 q + M_2 q^2 + M_3 q^3$ determines how long the final detect step takes. The detect step may require $O(l)$ steps. In the worst that $M(x) = 1$ for all bad strands x. That makes M_1 large, thus l is large. For many practical problems, $M(x)$ is usually large, thus l is small. By repeating the Separate operation, we can make $M(x)$ large.

12.2. Karp et al's Method

Karp, Kenyon and Waarts [25] show how to make the Separate operation more robust via repetition. For simplicity, assume a two-character alphabet. Now without loss of generality any call to Separate has the form Separate$(T, 0, i, T_0, T_1)$, which puts all strings from T whose ith character is 0 into T_0 and all whose ith character is 1 into T_1. Assume that the Separate operation misclassifies each strand with probability ϵ (Karp et al actually considered a more general problem where T_0 and T_1 have different misclassification probabilities). Here is their algorithm:

```
count_T := 0;
H := {T};
for j := 1 to 4D do
      for each T' ∈ H do
            Separate(T', 0, i, T'_0, T'_1)
            count_{T'_1} := count_{T'} + 1
            count_{T'_0} := count_{T'} − 1
      Partition {T'_0, T'_1 : T' ∈ H} into
       H_1, ..., H_m such that all tubes in the
       same H_i have the same count c_i.
      for i := 1 to m do
            T''_i := ⋃_{T' ∈ H_i} T'
            count_{T''_i} := c_i;
      H := {T''_1, ..., T''_m};
T_1 := ⋃_{T' ∈ H, count_{T'} > 0} T';
T_0 := ⋃_{T' ∈ H, count_{T'} ≤ 0} T';
```

Fix a position i and call a strand a 1-*strand* if it has a 1 in position i, a 0-*strand* otherwise. A 1-strand is classified correctly with probability at least $1 - \epsilon$. By Chernoff's bound, the probability that the count of the tube that contains the 1-strand is not positive is $O(\epsilon^D) = O(\delta)$.

For the general case, let ϵ be the probability that a 1-strand is misclassified, and let r be the probability that a 0-strand is misclassified. The so-called "δ-goal" is to partition all DNA strands from tube T into two tubes T_0 and T_1 such that each 1-strand has probability at most δ of ending up in T_0, each 0-strand has probability at most δ of ending up in T_1. For the general case, Karp et al proved:

Theorem 22 (Karp, Kenyon, Waarts). *The number of Separate operations required for the δ-goal is $\Theta(\log_\epsilon \delta \log_r \delta)$.*

13. Associative Memory

Baum [4] proposed building an associative memory that is much larger than conventional storage media. Up to 2^{70} records of the form (key,data) can be stored, each one represented by a single DNA strand. Records can be looked up using the Separate operation.

14. Other Models

Roweis et al [40] proposed a sticker model that behaves like a random memory. There are two kinds of DNA strands in the sticker model: memory strands and sticker strands. Each memory strand is divided into k regions, each consisting of m bases. A sticker is the (Watson–Crick) complement of a memory region. A memory region bound to a sticker encodes a 1. A memory region not bound to a sticker encodes a 0. They described a biological method to implement operations like Merge, Separate, and Set (which turns on a specified bit in every strand in a tube).

Condon et al [15] proposed another model based on surface chemistry. They also proved that their model can simulate circuits as in Section 8.1. Therefore many of the algorithms we described can be implemented in their model. They estimate that the maximum volume possible in their model is 10^{12}.

Acknowledgments

We are grateful to Mitsunori Ogihara and William Gasarch for their comments on earlier drafts.

References

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

[2] L. Adleman. On constructing a molecular computer. In *Proceedings of the 1st DIMACS Workshop on DNA Based Comput-*

ers (1995), pages 1–21. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[3] E. Bach, A. Condon, E. Glaser, and C. Tanguay. DNA models and algorithms for NP-complete problems. In *Proceedings of the 11th Annual Conference on Structure in Complexity Theory*, pages 290–299, 1996.

[4] E. Baum. Building an associative memory vastly larger than the brain. Technical report, NEC, 1996.

[5] E. T. Bax. Inclusinon and exclusion algorithm for the hamiltonian path problem. *Information Processing Letters*, 47:203–207, 1993.

[6] D. Beaver. Molecular computation. Technical report, Penn State University, 1995.

[7] D. Beaver. A universal molecular computer. In *Proceedings of the 1st DIMACS Workshop on DNA Based Computers (1995)*. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[8] R. Beigel and B. Fu. Molecular computing, bounded nondeterminism, and efficient recursion. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming*, pages 816–826, 1997. LNCS 1256.

[9] R. Beigel and J. Goldsmith. Downward separation fails catastrophically for limited nondeterminism classes. In *Proceedings of the 9th Annual Conference on Structure in Complexity Theory*, pages 134–138, 1994.

[10] D. Boneh, C. Dunworth, and R. J. Lipton. Breaking DES using a molecular computer. In *Proceedings of the 1st DIMACS Workshop on DNA Based Computers (1995)*. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[11] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. On the computational power of DNA. *Discrete Applied Math.*, 71:79–94, 1996.

[12] D. Boneh and R. Lipton. Making DNA computer error resistent. In *Proceedings of Second Annual DIMACS Workshop on DNA Computation*, 1996.

[13] J. F. Buss and J. Goldsmith. Nondeterminism within P. *SICOMP*, 22:560–572, 1993.

[14] J. D. C. Alvarez and J. Torán. Complexity classes with complete problems between P and NP-complete. In *Foundations of Computation Theory*, pages 13–24. Springer-Verlag, 1989. LNCS 380.

[15] W. Cai, A. E. Condon, R. M. Corn, E. Glaser, Z. Fei, T. Frutos, Z. Guo, M. G. Lagally, Q. Liu, L. M. Smith, and A. Thiel. The power of surface-based DNA computation. Manuscript, 1996.

[16] J. Díaz and J. Torán. Classes of bounded nondeterminism. *MST*, 23:21–32, 1990.

[17] W. Feller. *An Introduction to Probability Theory and Its Application*, volume 1. John Wiley and Sons, 1957.

[18] B. Fu. *Volume Bounded Molecular Computation*. PhD thesis, Yale University, Dept. of Computer Science, New Haven, CT, 1997.

[19] B. Fu and R. Beigel. A comparison of resource-bounded molecular computation models. In *Proceedings of the 5th Israel Symposium on Theory of Computing and Systems*, pages 6–11, 1997.

[20] B. Fu and R. Beigel. Molecular approximation algorithm for np-optimization problems. In *Proceedings of Third Annual DIMACS Workshop on DNA Computation*, pages 93–101, June 1997.

[21] B. Fu, R. Beigel, and F. Zhou. An $O(n^2(\log^2 n)2^n)$ volume molecular algorithm for Hamiltonian path. Manuscript, 1997.

[22] J. Goldsmith, M. Levy, and M. Mundhenk. Limited nondeterminism. *SIGACT News*, pages 20–29, June 1996.

[23] L. Hemachandra and S. Jha. Defying upward and downward separation. *Inf. & Comp.*, 121:1–13, 1995.

[24] L. Kari. DNA computing: Arrival of biological mathematics. *The Mathematical Intelligencer*, 19(2):9–22, 1997.

[25] R. Karp, C. Kenyon, and O. Warts. Error-resilient DNA computations. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 458–467, 1996.

[26] C. M. R. Kintala and P. C. Fischer. Refining nondeterminism in relativized polynomial-time bounded computations. *SICOMP*, 9(1):46–53, Feb. 1980.

[27] R. Lipton. Using DNA to solve NP-complete problems. *Science*, 268:542–545, Apr. 1995.

[28] A. Meyer and M. Paterson. With what frequency are apparently intractable problems difficult? Technical Report MIT/LCS/TM-126, M. I. T., 1979.

[29] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Math.*, 10:287–295, 1985.

[30] M. Ogihara. Breadth first search 3SAT algorithms for DNA computers. Technical Report 629, University of Rochester, July 1996.

[31] M. Ogihara and A. Ray. Simulating boolean circuits on DNA computers. In *Proceedings of First International Conference on Computational Molecular Biology*, pages 326–331. ACM Press, 1997.

[32] M. Ogihara, A. Ray, and K. Smith. Biomolecular computing—a shape of computation to come. *SIGACT News*, 28(3):2–11, 1997.

[33] C. H. Papadimitriou and M. Yannakakis. On limited nondeterminism and the complexity of the V–C dimension. In *Proceedings of the 8th Annual Conference on Structure in Complexity Theory*, pages 12–18, 1993.

[34] N. Pisanti. A survey on DNA computing. Technical report, University Di Pisa, 1997.

[35] J. Reif. Parallel molecular computation. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 213–223, 1995.

[36] J. Reif. Paradigms for biomolecular computation. In *Unconventional Models of Computation*, pages 72–93. Springer, 1998.

[37] D. Roos and K. Wagner. On the power of bio-computers. Technical report, University of Wurzburg, Feb. 1995. ftp://haegar.informatik.uni-wuerzburg.de/pub/TRs/ro-wa95.ps.gz.

[38] D. Roos and K. Wagner. On the power of dna-computing. *Information and Computation*, 131:95–109, 1996.

[39] P. Rothemund. A DNA and restriction enzyme implementation of Turing machines. In *Proceedings of the 1st DIMACS Workshop on DNA Based Computers (1995)*, pages 75–119. American Mathematical Society, 1996. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 27.

[40] S. Roweis, E. Winfree, R. Burgoyne, N. V. Chelyapov, M. F.Goodman, P. W. Rothemund, and L. M. Adleman. A sticker based model for DNA computation. In *Proceedings of the Second Annual Meeting on DNA Based Computers*, 1996.

[41] J. Sambrook, E. Fritsch, and T. Maniatis. *Molecular Cloning.* Cold Spring Harbor, NY, 1989.

[42] L. Sanchis. Constructing language instances based on partial information. *International Jour. Found. Comp. Sci.*, 5(2):209–229, 1994.

[43] C. P. Schnorr. Optimal algorithms for self-reducible problems. In *Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming*, pages 322–337, 1976.

[44] A. L. Selman. Natural self-reducible sets. *SICOMP*, 17:989–996, 1988.

[45] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

[46] W. Smith and A. Schweitzer. DNA computers in vitro and vivo. Technical report, NEC, 1995.

[47] S. Toda. PP is as hard as the polynomial-time hierarchy. *SICOMP*, 20(5):865–877, 1991.

[48] J. Watson, M. Gilman, J. Witkowski, and M. Zoller. *Recombinant DNA (2nd edition).* Scientific American Books, W.H.Freeman and Co., 1992.

[49] J. Watson, N. Hoplins, J. Roberts, et al. *Molecular Biology of the Gene.* Benjamin/Cummmings, Menlo Park, CA, 1987.