Computing with Molecules

Diana Rooß
Würzburg, Germany
diana@esme.mayn.de

Abstract

In 1994 Adleman published the description of a lab experiment, where he computed an instance of the Hamiltonian path problem with DNA in test tubes. He initiated a flood of further research on computing with molecular means in theoretical computer science. A great number of models have been introduced, and their computational power has been examined, with results on universality, complexity, efficient algorithms, and error resistance. The main results are presented in this survey.

1 Introduction

The pioneering achievement that every problem can be computed, providing that one can verbalize a method to solve it, belongs to the middle years of our century. Some years later the answer to the question "how to do it" was found. The necessary information was encoded in binary, and the code words were implemeted by a sequence of high and low voltage. This was the beginning of the phenomenal expansion of computers. A lot of innovations supported this breakthrough. But the idea to store and modify data by electronic means[1] has not changed in practice.

The description of a biological experiment caused a sensation among theoretical computer scientists in December 1994. A well-known scientist succeeded in computing the solution of an instance of a problem recognized to be hard, by means of biochemical manipulation of DNA (deoxyribonucleic acid): Adleman solved the Hamiltonian path problem for a graph with seven vertices and thirteen edges using a mixture of molecules.

Though in this computation the means of storage was new the result was not. For a long time it has been possible to search for Hamiltonian

[1] A few analog computers appeared but always stayed marginal.

paths in graphs of this size, and the usual method in this case is much faster than the new one by Adleman. However, complexity theoretical analysis of this experiment indicates that the input length and the computing time to solve the task are in good proportion. Adleman's procedure uses polynomial computing time, but so far there are only exponential time algorithms known to compute the Hamiltonian path problem on a sequential machine, and this does not rank as efficiently computable.

A Hamiltonian path is a sequence of edges in a graph, which touches every vertex exactly once. The Hamiltonian path problem is to decide, whether a graph has a Hamiltonian path or not. Concerning its structure, this problem is a typical representative of the important complexity class NP. Moreover, every other problem in NP can be solved with the same effort as the Hamiltonian path problem, however hard this will be. This property is referred to as NP-completeness. Adlemans method has initiated great interest, as one can hope to find out something new about all the problems in NP by analysing this special case.

The high degree of parallelism of the bio operations is the reason for the gain of time (considering larger inputs). The solution space is completely coded: each possible solution is represented by a DNA molecule. Subsequently some necessary conditions for a correct solution are checked step by step. In each of the steps all molecules are treated in parallel: those molecules that fit to the tested condition are seperated from those which do not.

A possible solution to the Hamiltonian path problem is a sequence of the graph's vertices. There are exponentially many sequences, but each of them has only polynomial length and therefore differs from other sequences at most at polynomially many positions. Consequently at most polynomially many conditions have to be tested. As all possible solutions are tested in parallel, their huge number is of no consequence when counting the time.

If we generalize Adleman's method we get a parallel model of computation. We view molecules as distributed data storage and molecular manipulations as parallel operations. In a test tube a soup of millions of molecules can be manipulated whereas silicon-based parallel computers are bounded to about ten thousand processors nowadays.

Such considerations encourage to ask, whether DNA-computers can break the "exponential border" and therefore problems that are not computable in a practical sense nowadays (as this would take thousands of years), could be solved in a sensible time. But simple means of calculation show that Adleman's procedure would need more molecules than the earth is able to give, for computing the Hamiltonian path problem for a graph with two hundred vertices [24, 38].

It is the task of theoretical computer science to find out the chances and the limits of any new computational model. The survey in hand first summarizes the article that initiated research on molecular computing. We give a condensed description of the experiment and Adleman's own estimation, hopes and doubts concerning this procedure (Section 2). "Is everything computable by DNA-computers?" We trace this question in Section 3. In fact there are different DNA models that have the power of an universal computer. In Section 4 we explain the first abstractions of Adleman's method towards a parallel computation model and their complexity theoretic analysis. Some more molecular operations and tests are mentioned which are expected to be realizable. There are many problems which can be solved with the machines of the first four sections only when taking great consumption of time and DNA resourses into the bargain. The efficient algorithms of the 5th Section among other things reduce the consumption of molecules considerably, compared with naive methods. In Section 6 we face the objection that molecular operations are probabilistic — so error analysis and strategies for error resistence are necessary. Finally (Section 7) we present a research area from the theory of formal languages: splicing systems model a biological prototype where DNA molecules are reorganized by the means of restriction and ligation. Many classes of languages can be characterized by different modifications of splicing systems.

Adleman's article caused a flood of further research work in various aspects and developments of molecular models. For that reason it is impossible to give a complete state of research in this presentation. Instead we select some papers which seem to be interesting to us and which afford an insight into the methodologies used when working on molecular computing. For the basics of theoretical computer science we recommend for example [27].

2 A Very Short Lesson in Biology

Just like in living creatures, in all the following computation models we use DNA as a storage medium for data. DNA manipulations are used to change this data, i. e. to compute something. The biological processes mentioned here have double meaning for computer scientists. On one hand they serve as a starting point for abstraction, on the other hand they help to estimate the implementability of further developed abstract "molecular operations" that will be presented later. So it seems necessary to summarize some basic biological features that will be important during this paper. For deeper insights they reader may consult for example [55].

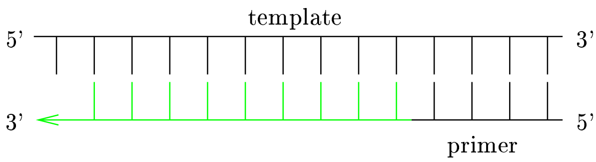
DNA is a linear-structured[2] molecule, built of four subunits called *nucleotides* that differ only by the attached bases adenine (A), guanine (G), thymine (T), or cytosine (C). Classically the letters A, G, T, and C will be used vicariously for the whole nucleotide. Arbitrary sequences of those nucleotides form *DNA strands*. Short DNA strands are referred to as *oligonucleotides*. DNA strands are oriented: a reaction between the 3' hydroxyl of one nucleotide and the 5' phosphate of the next, makes the nucleotides link together, so the DNA strands have a *5' end* and a *3' end*. In nature DNA does not exist as free single strands. The strands strive to pair with their *Watson-Crick complements* (named after the two biologists James Watson and Francis Crick who discovered the structure of DNA in 1953), to form a double helix structure which looks like a twisted ladder. The reason lies in the strong tendency of the bases A and T as well as G and C, to build hydrogene bonds. Therefore A and T (G and C, resp.) are called complements. Moreover, complementary strands are antiparallelly oriented. The following figure shows two complemetary oligonucleotides that attract each other and form a double strand.

```
5' ─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─ 3'
    T A T C G G A T C G G T A
    A T A G C C T A G C C A T
3' ─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─ 5'
```

One point that makes DNA interesting for computational questions is that we can make it behave in special ways. Here we introduce some operations that are every day work for biologists nowadays. *Annealing* is the name for the binding process of two complementary DNA strands, which happens by itself unter appropriate conditions like e. g. salt concentration and temperature (50–60°C). Conversely the separation of a double helix into single strands is called *melting* and can be reached by heating the DNA to 94°C.

Many enzymes operate on DNA, but here we need only some of them. *Restriction endonuclease enzymes* are able to specifically find (short) patterns in DNA double strands and cut them at these positions, cleaving the strands unevenly (called *sticky-ended*). *Ligase enzymes* ligate free ends of DNA strands. *Polymerase enzymes* elongate sticky-ended DNA strands from the free 3' end of the shorter strand (*primer*) by building a chain according to the Watson-Crick complement of the longer strand (*template*). The figure illustrates the process of elongation.

[2] There is also circular DNA found in bacteria, but that is not relevant here.



The whole procedure of melting templates, annealing primers and elongating DNA is called *polymerase chain reaction (PCR)*. It allows the amplification of wanted DNA fragments by using special primers. Often double-stranded DNA is to be filtered out. For that another enzyme, *nuclease* reduces single-stranded DNA down to nucleotides. Those can be seperated from the mixture by gel electrophoresis (see below), and only double strands remain. Both specially synthesised oligonucleotides and the mentioned ezymes are commercially available, also in large quantities.

There are procedures that can be used to filter out special DNA molecules from a mixture. For example gel is used to separate DNA by length, in a process called *gel electrophoresis*. DNA is negatively loaded. So if we put the DNA in a gel chamber to which an electric field is applied, the electric force will pull the strands towards the positive pole, the shorter the strands the faster they move. As DNA normally is invisible we use a kind of gel that colours the strands. After a while some fluorescent bands according to the different lengths of the strands appear in the gel. The DNA fragments can be separated, and the DNA can be eluated from the gel. Moreover, a given length can be detected when markers known to have that length are passed through other lanes of the gel. Another powerful and generally applicable procedure is *affinity purification* where all strands including a special pattern can be extracted from a mixture. This technique takes advantage of the affinity of single strands to their Watson-Crick complements. First an amount of the Watson-Crick complements of the pattern we look for is attached to magnetic beads in a column. A single-stranded mixture can be purified by passing it through this column. On their way through, all the strands that include the pattern bind to their complement strands and thus are retained. Unbound strands pass through without being retained.

From a more abstract point of view computer scientists can think of what is going on here as something very common to them: DNA strands are like strings over the alphabet $\{A, G, T, C\}$ with a string complementation operation defined on them. Test tubes do the job of set variables and hold sets of such strings. The biological lab procedures can be seen as operations on such set variables. Finally, to inspect the effect of string operations we have some tests on set variables.

There are different qualities of set operations and tests. Some of them are very easy to define and to implement. Perhaps the simplest of all operations is the *union* of set variables which can be implemented by pouring test tubes together. Moreover, all the enzyme based processes introduced above can easily be abstracted to set operations. But note that the biological processes have a probabilistic character. Gel electrophoresis can be used to realize an *emptiness test* on set variables (if there are no strands left, no fluorescent bands will appear). The *membership test* that decides whether a special string is in a set variable can be implemented as a combination of affinity purification (to select all strands containing the given one as a substrand) and gel electrophoresis (to seperate only the strands with the right length).

It can be a bit more tricky to implement other operations or tests that are quite common for classical string manipulation. E. g. the realization of a text substitution operation is described in detail on page 9 of this paper.

At least there are some desirable operations (from the computer scientist's point of view) for which implementation is unclear. Examples are *intersection* of set variables or a *subset test* (if true, one set variable holds a subset of another).

3 Adleman's Experiment

In a well-noticed article from December 1994 Adleman describes a completely new process for solving a combinatorial problem that he performed for an examplary input in lab [1]. He takes the NP-complete Hamiltonian path problem as representative of an important complexity class which is supposed to have exponential worst case time complexity.

Let G be a graph with n vertices $v_1, \ldots, v_n$, where vertices v_{in} and v_{out} are marked. An edge from v_i to v_j is abbreviated by $e_{i,j}$. G is called to have a *Hamiltonian path from* v_{in} *to* v_{out} if there is a path of edges starting with v_{in} and ending with v_{out} that contains every vertex of G exactly once. The *Hamiltonian path problem (HPP)* is the set of tuples (G, v_{in}, v_{out}) where G has a Hamiltonian path from v_{in} to v_{out}.

Adleman uses the following (nondeterministic) algorithm to solve the HPP for an input (G, v_{in}, v_{out}), where G has n vertices.

Step 1 Generate random paths in G.

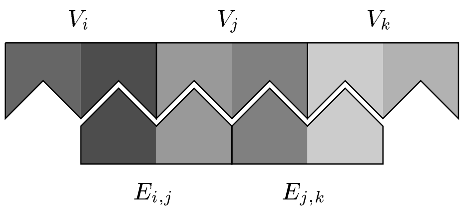
Step 2 Extract all paths beginning with v_{in} and ending with v_{out}.

Step 3 Extract all paths with length exactly n.

Step 4 Extract all paths that contain every vertex at least once.

Step 5 Accept if there are any paths left, otherwise reject.

Those steps are realized as molecular computation phases. Vertices and edges of G are coded by single-stranded DNA oligonucleotides of equal length. For all i and j let V_i denote the coding of v_i and $E_{i,j}$ the coding of $e_{i,j}$. Their Watson-Crick complements are marked by $\overline{V_i}$ and $\overline{E_{i,j}}$. Each V_i is randomly chosen while $E_{i,j}$ is composed of the second half of $\overline{V_i}$ and the first half of $\overline{V_j}$. Only the codings of edges from v_{in} and to v_{out} do not consider half but all of $\overline{V_{in}}$ and $\overline{V_{out}}$. If there are enough coding fragments in a mixture, annealing and sticky-end ligation build DNA double strands that represent random paths in G (Step 1). Supposed G has edges $e_{i,j}$ and $e_{j,k}$, the figure shows a piece of such a path.



For Step 2, PCR with the primers V_{in} and $\overline{V_{out}}$ is used to amplify the strands with the correct beginning and end, and makes the number of other path codings neglectable. In order to get path codings of length n the strands are separated in gel and PCR-amplified for several times to enhance its purity (Step 3). For Step 4 we melt the DNA, and one after another check each vertex for appearance in a path with the help of affinity purification. If a path of length n contains every vertex at least once, then it contains it exactly once and is a path we look for. To get the result in Step 5, again we use PCR-amplification and gel electrophoresis for testing whether there is any strand left or not.

The execution of the experiment took seven days of lab work. But Adleman remarks that optimizing the algorithms and the molecular implementation of the operations can reduce the time substantially. However, the time used for longer inputs grows only linearly with the number of vertices. On the other hand the number of different paths grows exponentially with the number of vertices — and so does the amount of DNA. Adleman himself poses some questions. He consideres that the proceeding has probabilistic properties. That means it has to be examined how many *equal* strands must be initialized, to reach high probability for the existence of at least one of the required codings in every step. All together careful error analysis and the examination of the realizability of the method is necessary [2, 31].

For all that, Adleman rates the potential of his DNA model to be promising. While modern supercomputers perform 10^{12} operations per second, he estimates 10^{20} operations per second to be realistic for molecular manipulations. Similar impressive views concern the consumption of energy and the capacity of memory: supercomputers need one joule for 10^9 operations, whereas the same energy is sufficient to perform $2 \cdot 10^{19}$ ligation operations. (For his energy calculations Adleman uses among others [56].) On a videotape every bit needs 10^{12} cubic nanometres storage; DNA stores information with a density of one bit per cubic nanometre (also see [5]).

A converse debate followed Adleman's publication (see e. g. [33]). On one hand the new ideas were picked up enthusiastically [5, 36], on the other hand critical voices came up immediately being mostly sceptic about the huge consumption of DNA that was required to solve hard problems [22, 54]. Adleman got down to many striking problems with his model in [2]. But the description of Kaplan, Cecchi, and Libchaber [28] who reperformed Adleman's experiment in lab but did not get clear results, throws doubts on the smooth realizability of even simple algorithms on small inputs.

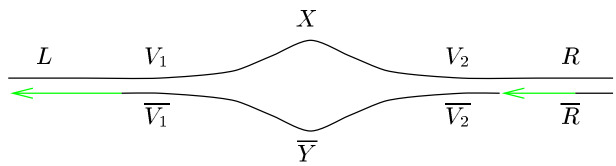
It depends on future research whether the prototype of a reliable biocomputer or even industrial mass production will be reality. However, Adleman got an avalanche underway.

4 Universal Computers

The term *universal computer* originates from recursion theory and describes a machine that is able to solve any computable problem. As an input pair it obtains a program which implements an algorithm for solving a computable problem, and a parameter for that program. There are many equal models which fit for a universal computer. Probably the *Turing machine* is the most important one. The commonly accepted *Thesis of Church* says that any intuitively computable problem is computable by a Turing machine. According to these standards every new computational model is going to be examined whether it is an universal computer or not.

A Turing machine M is described by a tuple $(Q, \Sigma, \delta, s, h)$ where Q is a set of states, Σ is a finite alphabet, $\delta : (Q \times \Sigma) \mapsto (Q \times \Sigma \times \{L, R\})$ a transition function (the last component describes the movement of the head), $s \in Q$ the start state, and $h \in Q$ the halting state. Without loss of generality we assume M to work on one one-sided tape, the input $a_1, \ldots, a_n$ to be found on the first n tape squares, and if M accepts the symbol '1' to be written on the first square and '0' otherwise. Each Turing computation can be represented by a sequence of configurations. Each configuration $C = (x_1 \ldots x_{k-1} q x_k x_{k+1} \ldots x_m)$ encodes a single step in the computation, where $x_1 \dots x_m$ represents the significant contents of the tape, q is the state and k is the position of the head. Note that successor configurations differ at maximally three neighboring positions — thanks to the local action of Turing machine transitions. M accepts an input, if there is a sequence of successor configurations that begins with the start configuration $C_0 = (sa_1 \dots a_n)$ and ends with a configuration $C_{acc} = (h1 \dots)$.

Beaver describes a molecular universal computer in [9]. For that he first shows how to implement a molecular *contextsensitive text substitution*: assume V_1, V_2, X, Y to be oligonucleotides and L, R to be single DNA strands. The aim of the operation is to substitute V_1XV_2 by V_1YV_2 in LV_1XV_2R. The first step is annealing of $\overline{V_1YV_2}$ and a fragment of $\overline{R}$ (working as a primer) and elongating it with polymerase. The figure shows the resulting double strand.



Next the strand is melted and the $\overline{LV_1YV_2R}$ part is elongated with a fragment of L as a primer. Nuclease destroys the remaining single stranded LV_1XV_2R, the resulting nucleotides are filtered out and only double strands remain.

Assumed we have such an (idealized) contextsensitive substitution operation on $\{A, G, T, C\}^*$ we can think of simulating a Turing machine. The idea is to follow the sequence of configurations, always keeping the actual configuration in a test tube. Keep in mind that all that is necessary to transduce one configuration to the next is the ability to replace a constant-length substring at a specified position — thanks to the local action of Turing machine transitions. Consequently a step of computation is simulated by substituting the significant area of a configuration according to the transition function. We need a suitable encoding function (unequivocal coding having no significant substrings in common) mapping pairs of states and positions (symbols and positions, resp.) to DNA, namely $e : (Q \cup \Sigma) \times \mathbb{N} \mapsto \{A, G, T, C\}^*$. Such an encoding can be designed deterministically or generated randomly. e allows us to encode the configurations: $C = (x_1 \dots x_{k-1} q x_k \dots x_m)$ is encoded by $C^e = e(x_1, 1) \dots e(x_{k-1}, k-1) e(q, k) e(x_k, k) \dots e(x_m, m)$. We start the simulation by initializing a test tube with C_0^e. Performing a transition $\delta(q, a) = (q', a', L)$ means to substitute the significant area $e(x_{k-1}, k-1)e(q, k)e(a, k)$ in a configuration strand by $e(q', k-1)e(x_{k-1}, k-1)e(a', k)$ (right moves are simulated analogously). When a halting configuration is reached in this way, we only have to test (via affinity purification) whether it starts with $e(h,1)e(1,1)$ and accept, or otherwise reject. This is the simulation of a deterministic Turing machine. Nondeterministic ones can be simulated directly also, by keeping all configurations that can be reached in the same number of steps in parallel. In that case affinity purification is used to select the configurations to which a transition is applicable before performing it.

From [48] it follows that without time bounds Lipton's model can already process any computable problem. In [51, 58] other universal molecular computers are introduced. An interesting approach is the definition of CNA (computational nucleic acid), modelled after RNA (ribonucleic acid) translation by Kurtz, Mahaney, Royer, and Simon [30]. Their model bases on modified copying of single molecules that serve as general purpose computers.

5 Complexity

The first to take up Adleman's method was Lipton [34, 36]. He made some idealized assumptions and gets a deterministic parallel computation model. By that he built an abstraction of the molecular computation in Adleman's experiment. The main results of those papers are a concrete DNA algorithm for the 3-SAT problem and the proof that any problem in NP can be solved in polynomial time using his model.

Inspired by Lipton's abstraction Rooß and Wagner picked up his ideas and constructed an abstract model called *DNA-Pascal*, in order to show the exact computational power of Lipton's model. Adleman's test tubes filled with DNA are mapped to set variables containing words over the alphabet $\{0,1\}$. Let T, T_1, and T_2 be such set variables. Concrete DNA manipulations are transformed into operations on these set variables. In detail, *initialisation* fills a set variable T with $\{0,1\}^n$ ($T := \mathrm{In}(n)$). The *union* of two set variables T_1 and T_2 contains all the words that were in either of the both before ($T := T_1 \cup T_2$). *Extraction* filters out all words of a set variable T_1 that have a special pattern; for convenience we introduce both, *bit extraction* (looks for a special bit a at a special position i, $T := \mathrm{Bx}(T_1, a, i)$) and *subword extraction* (looks for a special subword w anywhere in a word, $T := \mathrm{Sx}(T_1, w)$) which can be shown to have the same computational power [49]. Adleman's DNA test is the origin of the *emptiness test* that checks, whether there is any word left in a set variable T ($T = \emptyset$). These set operations and tests are supplemented by "plain" Pascal operations that can do what Adleman did implicitely as a laboratory assistant (like counting, conditional deciding and performing loops). To the time complexity of DNA-Pascal programs: to avoid the production of very long strands in a few steps the idea of the logarithmic cost criterion is adapted. The execution time of an operation "$T := \ldots$" is estimated as the length of the longest strand in T after the execution of this operation. A polynomial time bound on this model deliveres a complexity class that we call DNA-P.[3] The class DNA-P describes exactly Lipton's model.

P is the class of problems that can be deterministically computed in polynomial time. The P problems are estimated to be efficiently computable.

NP is a very important complexity class that contains many practically relevant problems, namely those whose solution space is representable as a search tree of polynomial depth but exponential width. Each path is coding one possible result for the problem. Other than algorithms combing all the tree are unknown to solve *NP-complete problems* (the most difficult problems in NP, see [20]). Nondeterministic algorithms compute along every path in the tree in parallel, each of those computation paths checking one result. This computation needs only polynomial time, but the exponentially many different computation paths are not allowed to communicate. The whole computation accepts if there exists a computation path that accepts.

The class $\mathrm{P}^{\mathrm{NP}} (= \Delta_2^p)$ is defined as follows: a deterministic polynomial time computation is underlying. It has the additional ability to ask questions to an *NP oracle*. This is something like an unlimited database that holds an NP problem and that needs only one step to deliver an answer whether a questioned word is in the problem or not. By reason of the transducability of any NP-complete problem into any other in polynomial time, the concrete choice of the oracle problem makes no difference. Moreover, because the oracle answers are "yes" or "no" it makes no difference whether an NP or an coNP oracle is concerned. In contrast to NP computations, P^{NP} computations have the possibility to use the results of NP questions for computing.

The class Σ_2^p is an extension of NP (exactly $\mathrm{NP}^{\mathrm{coNP}}$). Again the solution space can be arranged as a tree, but two-levelled this time. A Σ_2^p computation accepts if there is a computation path in the first level such that all its successor paths in the second level accept. You can think of it as an $\exists\forall$-acceptance.

[3] In the original paper the class is called DNA({IN,UN,EX},{EM})-P to distiguish it from other classes observed. IN, UN, and EX denote the allowed set operations and EM the set test.

In analogy to Δ_2^p, Δ_3^p is defined as a basic P computation with a Σ_2^p oracle.

All the classes described above clearly build a hierarchy, but they could not yet be shown to be different, although this is commonly believed. They (probably) are far below PSPACE (PSPACE computation trees have an unbounded succession of $\exists$ and $\forall$ levels), but besides P they seem to be more difficult than what is estimated to be efficiently computable.

Now we are ready to exactly classify DNA-P.

Theorem. DNA-P $= \Delta_2^p$.

Proof. In the first part of the proof it will be shown in analogy to Lipton's method [35] that any NP problem can be solved by a DNA-P computation. The second part gives the simulation of an arbitrary DNA-P program by a Δ_2^p computation.

[$\supseteq$]: The following DNA-Pascal program shows that HPP $\in$ DNA-P. Given a graph G with vertices $v_1, \ldots, v_n$, v_{in} and v_{out}. For all i let the vertex v_i be block coded by $V_i(1) \ldots V_i(\log n)$. The step numbers in the following algorithm correspond with the numbers in the description of Adleman's algorithm (see page 6 of this paper).

begin
  $T := \mathrm{In}(n \log n)$; (Step 1 + Step 3)
  for $i := 1$ to $\log n$ do (Step 2)
    $T := \mathrm{Bx}(T, V_{in}(i), i)$;
  for $i := 1$ to $\log n$ do
    $T := \mathrm{Bx}(T, V_{out}(i), n \log n - (\log n - i))$;
  for $i := 1$ to n do (Step 4)
    $T := \mathrm{Sx}(T, V_i(1) \ldots V_i(\log n))$;
  if $T = \emptyset$ then reject else accept (Step 5)
end

Step 1 + Step 3 Create all possible paths in G, i.e. initialize the set variable T with $\{0,1\}^{n \log n}$. (The factor n stands for the number of vertices and the factor $\log n$ for the length of the vertex coding.)

Step 2 Extract those paths from T that begin with v_{in} and end with v_{out}.

Step 4 Extract one after the other all paths from T that contain $v_1, \ldots, v_n$.

Step 5 Test whether T is empty — if so reject, otherwise accept.

Now we can conclude $\Delta_2^p = \mathrm{P}^{\mathrm{NP}} \overset{(a)}{=} \mathrm{P}^{\mathrm{HPP}} \overset{(b)}{\subseteq} \mathrm{P}^{\mathrm{DNA-P}} \overset{(c)}{=}$ DNA-P, where (a) holds because HPP is NP-complete, (b) because HPP $\in$ DNA-P, and

(c) because you can already do the basic P computation in the DNA-P computation.

[$\subseteq$]: The idea of this part of the proof bases on the fact that the set variables in a DNA-Pascal program have an influence on the result of the computation only via the emptiness test. So we show how to simulate an emptiness test in any arbitrary polynomially time-bounded DNA-Pascal program P. This time we only allow bit extraction because this operation is at least as strong as subword extraction [49]. Suppose the set variables $T_1, \ldots, T_k$ to be used in P.

Inspect the following problem R_P: given the words x, $a_1 \ldots a_m$, and z, and the integers i and t. Is $z \in T_i$ at step t of program P on input x, if the m tests before were answered by $a_1, \ldots, a_m$?

We prove that R_P is in P. Let s_j ($j = 1, \ldots, k$) be integer variables with the intention to keep whether z is in T_j ($s_j = 1$) or not ($s_j = 0$). All the s_j are initialized with 0 (because in the beginning of the program all set variables are empty). Now we follow P step by step, replacing the set instructions by Pascal instructions without set variables ($z(l)$ means the lth position in z):

set instruction	replaced by Pascal instruction
$T_j := \mathrm{In}(n)$	if $\lvert z\rvert = n$ then $s_j := 1$ else $s_j := 0$
$T_j := T_u \cup T_v$	$s_j := \max(s_u, s_v)$
$T_j := \mathrm{Bx}(T_u, l, a)$	if $z(l) = a$ then $s_j := s_u$ else $s_j := 0$

When tests appear they are replaced by the given answers. After having simulated t steps in this way, the answer to our problem is the current value of s_i. So $R_P \in \mathrm{P}$.

Now we are ready to describe the simulation of P by an ordinary Pascal program P' using a coNP oracle, which is a Δ_2^p computation then. Note, that since P is polynomially time-bounded the words in the set variables are polynomially length-bounded. Thus an emptiness test $T_i = \emptyset$ in step $t+1$ in the computation of P on x can be replaced by the query

$$\neg\exists z(z \text{ polynomially length-bounded and } (x, a_1 \ldots a_m, z, i, t) \in R_P)$$

assuming that there have been exactly m test before with the results $a_1, \ldots, a_m$. This is clearly a question to a coNP oracle. Since P' can collect the answers to all oracle queries already made, the answer word $a_1 \ldots a_m$ for the current query is known. □

Moreover, Rooß and Wagner expand DNA-Pascal with other molecular operations and tests [48, 49, 50]. For the exact definition of the operations refer to the original articles, because the computational power depends only on a few properties that are presented here. In detail, we can categorize the operations as follows.

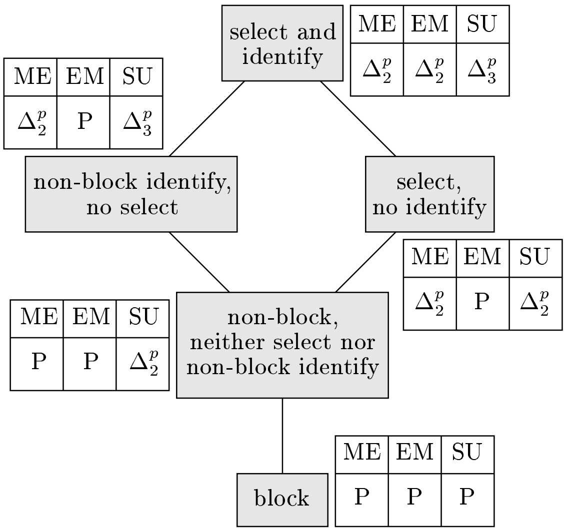
- Operations have the *block property* if they preserve the fact that all strands of a certain length are contained in a test tube, or none of them;
- Operations have the *select property* if they can exclude certain strands from a test tube (and probably modify the remaining);
- Operations have the *identify property* if they can identify different strands (i. e. by identifying or cutting certain symbols or substrings).

We call the operations with the block property *block operations*, those with the select property *select operations* and those with the identify property *identify operations*. A *non-block operation* does not have block property, an operation with the identify and the block property is called *simple identify operation*.

Lipton uses the *emptiness test (EM)* that is true, if there are no strands in a test tube in question. Additionally we examine another two DNA tests: the *membership test (ME)* is true, if there is a certain strand in the test tube in question; the *subset test (SU)* is true, if the contents of a certain test tube is as a subset in the test tube in question.

To construct an algorithm for the model, on the one hand the set of commands of classical Pascal, on the other hand a set of molecular operations and tests are allowed. Some variants of this DNA-Pascal are inspected, they differ in the set of valid molecular operations and tests. If we limit the resulting programming languages to polynomial computing time, they define different problem classes that can be characterized by classical complexity classes. The pure Pascal commands do not influence the magnitude of those classes because polynomially time-bounded Pascal exactly characterizes the class P.

We can summarize the results of [50] in the following figure. The grey vertices of the graph are combinations of operation properties. A table is attached to each vertex. It shows the computational power of polynomially time-bounded DNA-Pascal using operations with the properties in the vertices and different tests.



If only block operations are used no molecular test can increase the power of polynomially time-bounded DNA-Pascal above P. Non-block operations without select property and at most simple identify operations lift polynomially time-bounded DNA-Pascal to the magnitude of Δ_2^p.

The select property seems to turn the scale for the power of EM: with select operations and EM (or SU) polynomially time-bounded DNA-Pascal is as powerful as Δ_2^p. (Lipton's model can be classified at this point.) Without select operation, EM does not have an influence on the power of the programming language.

A non-block identify operation lifts the power to Δ_2^p if ME is allowed — with SU the power is Δ_3^p. We get the same effect if only simple identify operations but additionally select operations are valid.

There are other operations (e. g. intersection of set variables) which raise the power of Lipton's model even to PSPACE. If the strands are logarithmically length-bounded then DNA-Pascal with Lipton's operations and test is exactly characterized by the class L [49].

Beaver [9] proposes the direct simulation of a Turing machine based on molecular substitution (see page 9 of this paper) to show that DNA computations can solve all PSPACE problems.

6 DNA Algorithms

Adleman's and Lipton's naive algorithms for the Hamiltonian path and the 3-SAT problem, are valuable contributions to the analysis of the computational power of those machines. But it appears that only small inputs are realistic, and for these input sizes we know classical algorithms that get the solution much faster than the mentioned molecular algorithms. Crucial are the vast solution spaces of exponential order that we get, when performing exhaustive search. Indeed, we can not expect to reduce the solution space for such problems to polynomial size even with optimal algorithms because this would imply the equality of P and NP [49]. Nevertheless, better algorithms can raise the number of the actually computable instances considerably.

Bach, Condon, Glaser, and Tanguay [4] present clever algorithms that use Adleman's operations and test. For example they reduce the DNA consumption from an order of 3^n (following Adleman in [2]) to 2^n while computing the NP-complete 3-Coloring problem. A graph is called to be *3-colorable* if its vertices can be painted in three colors such that every two vertices connected by an edge, have different colors. The *3-Coloring problem* is the set of graphs that are 3-colorable. Assume the three colors to be red, green and blue. In addition to the operations of Lipton's model we need another DNA operation: *append* appends a given symbol to every strand in a test tube. The algorithm works as follows. Let $G = (V, E)$ be the input graph with $V = \{v_1, \ldots, v_n\}$, $e_{i,j}$ is an edge from v_i to v_j.

Step 1 In a preprocessing step in classical Pascal we compute a spanning tree $G' = (V, E')$ of G. We assume the prefix order of the vertices in G' to be the same as their index order (otherwise rename G). Moreover, we fix the color of the root v_1 to be red.

Step 2 Now we construct the solution space, starting with a test tube T containing only one strand representing red. Successively we construct in T all valid colorings of G': one after another for every vertex v_i we separate T into three different tubes T_r, T_g, and T_b, resp.: it depends on the color of the parent node of v_i being red, green or blue, resp. As v_i cannot be painted in the same color as its parent node we have only two choices to elongate the strands in T_r, T_g, and T_b. So e. g. we construct the tubes T_{rg} and T_{rb}, resp., by appending the codings of green or blue, resp., to T_r. The tubes T_g and T_b are treated in the same way. The six resulting tubes are united to be the new T.

Step 3 To check whether there are valid colorings for G coded in T, we separate the solutions where v_i and v_j are differently colored, with the help of the edges $e_{i,j}$ that are not in E'. In the end we test if there are any strands left in T and accept, or reject otherwise.

It is easy to see that the algorithm acts with as solution space of only $O(2^n)$, and nevertheless, it does not consume more than polynomial time.

Nature has a method to comb huge solution spaces: repeated selection cycles on much smaller subspaces. Dynamical programming picks up this idea to reduce the number of required DNA strands [7, 54]. Ogihara utilizes the Monien-Speckenmeyer algorithm [39] to construct a DNA algorithm for the NP-complete problem 3-SAT that increases the maximal input length from 70 (specified by Lipton) to 120 [40].

Not only the decision of sets with DNA computation is of interest. If molecular models want to compete with silicon-based models, the computation of functions must be possible. Within that, the addition plays an important part; a corresponding DNA algorithm is presented in [23]. Molecular matrix multiplication is the theme in [42] and a quite early paper by Beaver deals with a factorisation algorithm [8]. The #P-complete problem of permanent calculation in a matrix is solved in [57, 32] by a DNA algorithm.

Boolean circuits are one of the extensively studied parallel computation models. Their processors are gates that execute the boolean functions "not", "and" and "or". The complexity of those circuits are given by the *fan-in* and *fan-out* (the maximal number of inputs and outputs, resp. of a gate), the *size* (the number of gates) and the *depth* (the longest way through the circuit). Often the fan-in of and-gates is bounded to two whereas the fan-in of or-gates is arbitrary; such circuits are called *semi-unbounded*. The simulation of boolean circuits by DNA-computers is among others examined in [35, 11, 29]. Ogihara and Ray [41] specify a molecular algorithm that simulates semi-unbounded boolean circuits of depth t and size g in time $\mathrm{O}(t \log f)$ and DNA consumption $\mathrm{O}(gf)$, where f is the fan-out of a circuit. In particular this makes it possible to run the simulation of the important circuit class NC^1 in a time that is linear to the depth of the circuits.

DNA molecules in solution have an alternative in DNA molecules that are fixed to a surface. Liu et al. [37] take in that biochemical method and develop a molecular computation model to efficiently simulate circuits [13] (the number of parallel operations is proportional to the size of the circuit).

Another possible application of molecular computers is cryptography. In [10] it is outlined how to break the data encription standard (DES) with Adleman's model extended with some more operations. DES codes information with a 56 bit key. For an input pair of cipher text and plain text, a classical algorithm had to sequentially test 2^{56} possible keys, what takes some ten thousand years assuming a performance of hundred thousand operations per second. Boneh, Dunworth and Lipton show how to compute the key in about four month of lab work. The *sticker model* by Roweis, Winfree et al. is a computation model that combines DNA manipulations and a random access memory [52]. With one gram of DNA and in spite of many error possibilities, DES can be broken by the sticker model in reasonable time [3].

7 Error Resistance

Most of the models mentioned above base on idealized assumptions, in particular deterministic working methods of molecular operations and tests. Indeed the analysis of the biochemical process as well as the repetition of Adleman's experiment that could not deliver definite results [28], let the realization of those assumptions seem to be difficult. The upper bounds of molecular computation models with perfect operations and tests, show limits that surely can not be broken by realistic implementation. Nevertheless, a profound error analysis and strategies for error resistance help to improve the classification of these models and give statements according to their realizability.

The implementation of the extraction operation by affinity purification like e. g. in Adleman's experiment throws doubts on the finite result of such a molecular computation. Assume affinity purification to extract the proper strands with a probability of 0.95. If we perform seven extractions as Adleman did (see Step 4 on page 6), then the probability that exactly those strands are left which meet the extraction criterion, is only $0.95^7 \approx 0.7$. We can easily compute that after one hundred extraction steps it would be nothing but 0.006. Instead of that Amos, Gibbons, and Hodgeson implement the extraction operation with the means of restriction enzymes that destroy all strands containing a certain pattern. Although restriction enzymes work with a small non-zero error rate, too, they reach a distinctly lower error probability.

The reliability of the extraction operation can also be increased with a proceeding by Karp, Kenyon, and Waarts [29]: they apply PCR to a series of test tubes instead of only one. Boneh and Lipton [12] make their 3-SAT algorithm error resistant by duplicating the number of strands periodically.

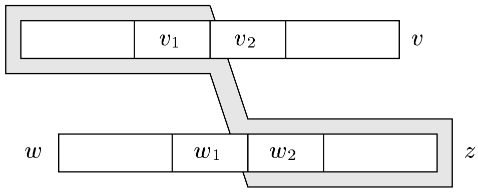
Among the first to be engaged in the error analysis of Adleman's and Lipton's operations are Bach, Condon, Glaser, and Tanguay [4]. They present a probabilistic model for the duplication of test tubes. Indeed duplicating is a process of splitting test tubes. Lipton assumes each amount of each strand in an old tube to be divided equally to the new tubes. In the probabilistic model each possible partition of the old test tube is equally likely. This distribution delivers a redundancy factor r, dependent of a given error parameter ϵ. We then have to multiply the number of strands we deal with by r, in order to get all the strands we desire after the duplication (or series of duplications) with probability $1-\epsilon$. Further amplification can be avoided this way, and with that another source of error.

In [6] Baum is concerned with the coding of strings in DNA strands. It appears that not only the weight of the DNA that is used for computation gives a natural border for the input size, but also the number of strings that have to be coded. Two crucial problems appear with random codings as they are suggested in [1, 9, 34]: common subsequences arise that make extraction operations unreliable, and complementary subsequences in one molecule take the risk of lumping, when these fragments happen to bind. Assume oligonucleotides of length k firstly to be sufficient long to serve as an "anchor" in the process of affinity purification, and secondly to be not long enough to cause the mentioned problems accidentially. The exact value of such k is to be found out in practical experiments. Baum presents a coding which guarantees that no common subsequences and no complementary subsequences of length k appear, and he shows that the maximum number of strings that can be coded this way is $3 \cdot 4^{k-2} + 5 \cdot 4^{k-4} - 2 \cdot 4^{\frac{k}{2}-1}$, $k \geq 5$. So, if $k = 5$ the coding of $\{0,1\}^7$ (or 208 strings) and if $k = 9$ the coding of $\{0,1\}^{12}$ (or 54,144 strings) is possible.

8 Splicing Systems

Finally we should not miss to consider a related disposition from the theory of formal languages. Already in 1987 an article of Head [25] leads this concurrency off. He constructs a language theoretical model after a biological example: a language is given by a test tube filled with DNA and a number of valid operations (modelling restriction endonucleasis and sticky end ligation) working on it.

Head calls the formalization of his idea a *splicing system*, consisting of an initial set of strings on a final alphabet and a rule set of four tuples $(v_1, v_2; w_1, w_2)$ of strings over that alphabet. The rules are applied on two strings v and w at a time. If v contains the pattern v_1v_2 and w the pattern w_1w_2, a new string z is produced, beginning like v including v_1 and ending like w from w_2 on (see the following figure).



If the patterns v_1v_2 or w_1w_2 happen to appear more than once in a string then the processing chooses nondeterministically: the strings are cut and the fragments recombined in any possible way.

The *language generated by a splicing system* is the set of all strings that can be produced by a repeated application of the rules on the initial set. In the theory of formal languages two classes of languages have a great importance: the *class of regular languages* that is easy to decide, and the *class of recursively enumerable languages (r. e.)* that not only contains the decidable but also the semi-decidable languages. Every recursively enumerable language can be generated by a *type-0 grammar*, i. e. a four-tuple of a finite alphabet, a set of terminal symbols and a set of initial words over the alphabet, and a set of unrestricted rules. Type-0 grammars have the same computational power as Turing machines.

Splicing systems have extensively been investigated. The generative power of splicing systems is examined among others in [25, 21, 43, 45, 46]. Culik and Harju show that a splicing system with regular initial set and finite set of rules generates again only regular sets [16]. But if we allow a regular set of rules, then the set of generated languages is already r. e. [44].

When considering extended splicing systems, the same holds even for finite initial sets and regular rule sets [45]. The definition of extended splicing systems is quite grammar like: they are given by a finite alphabet where special terminal symbols are marked, a set of initial words over the alphabet, and a set of rules. The proof bases on simulating an arbitrary type-0 grammar by a suitable extended splicing system with a finite set of initial words and a regular set of splicing rules. The splicing rules have threefold characters: first they encode the type-0 rules, next they guaranty that these rules are applied only at "right" positions in the strings, and last they destroy auxiliary symbols in the end of the generation phase to produce terminal words. Such a construction allows to produce any word of the given recursively enumerable language with the splicing system, and the other way round, any generatable word is in the language.

Denninghoff and Gatterdam introduce the *multiplicity of strings in sets* [18]: they model that the strings are consumed in the process of splicing by considering the appearence frequency of a string in a set. By this means we implicitly get the possibility to add, subtract and multiply integers. If the multiplicity of sets is taken into consideration, splicing systems with a finite set of rules can already generate r. e. Also the existence of a "universal computer" in form of a splicing system is proved: there are universal splicing systems that can simulate the working method of any arbitrary splicing system. Those universal splicing systems either have regular rule sets or finite rule sets either with multiplicity or with an adding mechanism [18, 14, 44]. That means in particular that splicing systems have the same power as Turing machines.

Nature uses lots of different non-linear molecular structures. Some of them like the circular shaped DNA which can be found in bacteria have been modelled as splicing systems. In [26, 47, 53, 59] such cyclic splicing systems are considered, and it turns out that in this case the result of Culik and Harju does not hold [53]. Freund looks at splicing systems on graphs in [19].

A combination of splicing systems with the meaning of Head (splicing operations are applied in test tubes) and DNA-computers in the sense of Adleman (new test tubes are filled by extraction from old ones) are examined in [15, 17]. Those *distributed splicing systems* with finite initial and rule sets already characterize r. e. On its base a universal computer can be constructed, too.

9 Summary and Final Remark

The kind of computation that is done in each of our body's cells every moment we live, was finally discovered by computer science. During the last two and a half years many different DNA models have been developed and analysed under several points of view. It turned out to be quite fruitful that many researchers with different interests have picked up that subject. Thereby complexity, stability and realizabilty played an important part. Nevertheless, a lot of questions remain open.

- Under idealized assumptions for molecular operations and tests a lot of research was done.
  - Universal DNA-computers were constructed.
  - Different molecular computation models with the power of the simplest boolean circuits up to the power of PSPACE were defined.
  - Tricky algorithms were designed.

— Encription systems were broken.

Can we build models that take the probabilistic character of the biological processes into account? What is their computational power and their algorithmic potential?

- An enormous number of computation models were introduced, different in the level of abstraction and in functionality. What does a comparison of these models deliver? Is there perhaps one basic model that can serve as *the* theoretical model for DNA-computation (with a significance similar to Turing machine or PRAM)?
- First steps in the field of error resistance were done.

  — The reliability of some molecular manipulations was examined and error-resilient operations were suggested.

  — Unequivocal data encodings were presented.

  What are the error probabilities of molecular mechanisms and which strategies help to reduce these errors? Up to which borders can such errors be reduced in principle?
- DNA manipulations were intensively examined and a model for RNA manipulation was introduced. What other molecular structures and shapes can be manipulated and utilized for computation?
- Molecular operations on test tubes were assumed to be "completely" executed in all known computation models. What effect has the partially execution, i. e. what happens if the biological process is intentionally broken off (by turning off the heat, ...)?

The subject is very recent and fast developing, so it is hard to give a perspective. Though it is not quite clear now, how exactly molecular computers can be constructed and what features they will have, it is worth investing intellectual power in the question of such machines' abilities in principle. The hope is to do important preparatory work, similar to the theory of computation in the 30's, that was founded already before real computers could be built.

Research on DNA-computing is still in the beginning. The results up to now show, that DNA-computers will not turn our insights of efficient computation upside down. Nevertheless, they are powerful instruments for the implementation of parallel processes.

The future of molecular machines could be in a combination with classical computers. Such systems should profit by the specific suitability of the components: well parallelizable tasks should be computed with DNA, whereas inherently sequential jobs should be done by silicon-based chips. It is still too early to expect highly efficient realizations, but an unremitting search of biological possibilities for quick and inexpensive manipulation of molecules is in process.

Acknowledgements

Special thanks are due to many people, without whom this article could not have been written. Ramanujam and German Rooß carefully read the manuscript and improved it with their engaged suggestions. Diana Raffelsbauer helped me getting inside the technical aspects of DNA manipulations. Klaus Wagner, the head of the Lehrstuhl für Theoretische Informatik at the University of Würzburg, allowed me to use the infrastructure of the institute: to surf through the net and to use the library in order to collect all the material summarized here. And last, but not least there is the hard work of so many computer scientists, who made this research area progress and become interesting to the public.

References

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

[2] L. M. Adleman. On constructing a molecular computer. In E. B. Baum and R. J. Lipton, editors, *DNA based computers.* American Mathematical Society, 1996. Number 27 in DIMACS: Series in Discrete Mathematics and Theoretical Computer Science.

[3] L. M. Adleman, P. W. K. Rothemund, S. Roweis, and E. Winfree. On applying molecular computation to the data encryption standard. In *Proceedings of the Second Annual Meeting on DNA based Computers*, June 1996.

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

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

[6] E. B. Baum. DNA sequences useful for computation. Technical report, NEC Research Institute, 1996.

[7] E. B. Baum and D. Boneh. Running dynamic programming algorithms on a DNA computer. In *Proceedings of the Second Annual Meeting on DNA based Computers*, June 1996.

[8] D. Beaver. Computing with DNA. *Journal of Computational Biology*, 2(1):1–8, 1995.

[9] D. Beaver. Molecular computing. Technical report, Penn State University, January 1995.

[10] D. Boneh, C. Dunworth, and R. J. Lipton. Breaking DES using a molecular computer. Technical report, Princeton University, 1996.

[11] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. On the computational power of DNA. Technical report, Princeton University, 1995.

[12] D. Boneh and R. J. Lipton. Making DNA computers error resistant. Technical report, Princeton University, 1996.

[13] 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 surfaced-based DNA computation. Technical report, University of Wisconsin, July 1996.

[14] E. Csuhaj-Varjú, R. Freund, L. Kari, and G. Păun. DNA computation based on splicing: universality results. In *Proceedings of the First Annual Pacific Symposium on Biocomputing*, 1996.

[15] E. Csuhaj-Varjú, L. Kari, and G. Păun. Test tube distributed systems based on splicing. *Computers and AI*, 15(2–3):211–232, 1996.

[16] K. Culik II and T. Harju. Splicing semigroups of dominoes and DNA. *Discrete Applied Mathematics*, 31:261–277, 1991.

[17] J. Dassow and V. Mitrana. Splicing grammar systems. *Computers and AI*, 15(2–3), 1996.

[18] K. L. Denninghoff and R. W. Gatterdam. On the undecidability of splicing systems. *International Journal of Computer Mathematics*, 27:133–145, 1989.

[19] R. Freund. Splicing systems on graphs. In *Proceedings of Intelligence in Neural and Biological Systems*, pages 189–194, May 1995.

[20] M. Garey and D. Johnson. *Computers and intractability.* Freeman and C., San Francisco, 1978.

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

[22] D. K. Gifford. On the path to computation with DNA. *Science*, 266:993–994, November 1994.

[23] F. Guarnieri, M. Fliss, and C. Bancroft. Making DNA add. *Science*, 273:220–223, July 1996.

[24] J. Hartmanis. On the weight of computations. *Bulletin of th European Association for Theoretical Computer Science*, 55:136–138, February 1995.

[25] T. Head. Formal language theory and DNA: an analysis of the generative capacity of specific recombinat behaviours. *Bulletin of Mathematical Biology*, 49:737–759, 1987.

[26] T. Head. Splicing schemes and DNA. *Nanobiology*, 1:335–342, 1992.

[27] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.

[28] P. D. Kaplan, G. Cecchi, and A. Libchaber. Molecular computation: Adleman's experiment repeated. Technical report, NEC Research Institute, 1995.

[29] R. Karp, C. Kenyon, and O. Waarts. Error resilent DNA computation. Technical report, École Normale Supérieure de Lyon, September 1995.

[30] S. A. Kurtz, S. R. Mahaney, J. S. Royer, and J. Simon. Biological computing. to appear in L. A. Hemaspaandra and A. Selman, editors, Complexity Retrospective II.

[31] S. A. Kurtz, S. R. Mahaney, J. S. Royer, and J. Simon. Active transport in biological computing (preliminary version). In *Proceedings of the Second Annual Meeting on DNA based Computers*, June 1996.

[32] T. H. Leete, M. D. Schwartz, R. M. Williams, D. H. Wood, J. S. Salem, and H. Rubin. Massively parallel DNA computation: Expansion of symbolic determinants. In *Proceedings of the Second Annual Meeting on DNA based Computers*, 1996.

[33] M. Lineal, N. Lineal, Y.-M. D. Lo, K. F. C. Yiu, S. L. Wong, B. Bunow, and L. M. Adleman. Letters: On the potential of molecular computing. *Science*, 268:481–484, April 1995.

[34] R. J. Lipton. Speeding up computations via molecular biology. Technical report, Princeton University, 1994.

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

[36] R. J. Lipton. Using DNA to solve NP-complete problems. Technical report, Princeton University, 1995.

[37] Q. Liu, Z. Guo, A. E. Condon, R. M. Corn, M. G. Lagally, and L. M. Smith. A surface-based approach to DNA computation. In *Proceedings of the Second Annual Meeting on DNA based Computers*, June 1996.

[38] D. A. Mac Dónaill. On the scalability of molecular computational solutions to NP problems. *The Journal of Universal Computer Science*, 2(2):87–95, February 1996.

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

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

[41] M. Ogihara and A. Ray. Simulating boolean circuits on a DNA computer. Technical report, University of Rochester, August 1996.

[42] J. S. Oliver. Computation with DNA-matrix multiplication. In *Proceedings of the Second Annual Meeting on DNA based Computers*, June 1996.

[43] G. Păun. On the power of the splicing operation. *International Journal of Computer Mathematics*, 59:27–35, 1995.

[44] G. Păun. On the splicing operation. *Discrete Applied Mathematics*, 70(1):57–79, September 1996.

[45] G. Păun. Regular extended H systems are computationally universal. *Journal of Automata, Languages and Combinatorics*, 1(1):27–36, 1996.

[46] G. Păun, G. Rozenberg, and A. Salomaa. Computing by splicing. *Theoretical Computer Science*, 168(2):321–336, 1996.

[47] D. Pixton. Linear and circular splicing systems. In *Proceedings of Intelligence in Neural and Biological Systems*, pages 181–188, May 1995.

[48] D. Rooß and K. W. Wagner. On the power of bio-computers. Technical report, Universität Würzburg, February 1995.

[49] D. Rooß and K. W. Wagner. On the power of DNA-computers. Extended version of [48], 1995.

[50] D. Rooß and K. W. Wagner. On the power of DNA-computing. *Information and Computation*, 131(2):95–109, December 1996.

[51] P. Rothemund. A DNA and restriction enzyme implementation of Turing machines. In E. B. Baum and R. J. Lipton, editors, *DNA based computers.* American Mathematical Society, 1996. Number 27 in DIMACS: Series in Discrete Mathematics and Theoretical Computer Science.

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

[53] R. Siromoney, K. G. Subramanian, and V. R. Dare. Circular DNA and splicing systems. In *Proceedings of the 2nd International Conference on Parallel Image Analysis*, pages 260–273, 1992.

[54] W. P. C. Stemmer. The evolution of molecular computation. *Science*, 270:1510, December 1995.

[55] L. Stryer. *Biochemistry.* W. H. Freeman and Company, New York, fourth edition, 1995.

[56] J. D. Watson, N. H. Hopkins, J. W. Roberts, J. A. Steitz, and A. M. Weiner. *Molecular Biology of the Gene.* Benjamin/Cummings, Menlo Park, CA, third edition, 1987.

[57] R. M. Williams and D. H. Wood. Exascale computer algebra problems interconnect with molecular reactions and complexity theory. In *Proceedings of the Second Annual Meeting on DNA based Computers*, June 1996.

[58] E. Winfree. On the computational power of DNA annealing and ligation. In E. B. Baum and R. J. Lipton, editors, *DNA based computers.* American Mathematical Society, 1996. Number 27 in DIMACS: Series in Discrete Mathematics and Theoretical Computer Science.

[59] T. Yokomori, S. Kobayashi, and C. Feretti. On the power of circular splicing systems and DNA computability. Technical Report CSIM 95-01, University of Electro-Communications, Tokyo, July 1995.