On The Computational Power of DNA

Dan Boneh
dabo@cs.princeton.edu

Christopher Dunworth
ctd@cs.princeton.edu

Richard J. Lipton*
rjl@cs.princeton.edu

Department of Computer Science
Princeton University
Princeton, NJ 08544

Jiří Sgall†
sgallj@earn.cvut.cz
Hebrew University
Jerusalem, Israel

Abstract

We show how DNA based computers can be used to solve the satisfiability problem for boolean circuits. Furthermore, we show how DNA computers can solve optimization problems directly without first solving several decision problems. Our methods also enable random sampling of satisfying assignments. Finally we suggest a procedure for evaluating functions in the polynomial hierarchy.

1 Introduction

In the very short history of DNA (deoxyribonucleic acid) based computing there have already been a number of exciting results. It all started with Adleman's [1] beautiful insight that showed that biological experiments could solve the Directed Hamiltonian Path problem (DHP). Then, Lipton [8] showed how to use DNA to solve more general problems, namely to find satisfying assignments for arbitrary (directed) contact networks, which includes the important case of arbitrary formulas.

Since then there has been a series of papers on DNA computation. Each of these subsequent results is of the following form: Given enough strands of DNA and certain biological operations; one can simulate some classic model of computation efficiently. Some compare to formulas, some to circuits, others to 1-tape Nondeterministic Turing Machines.

The goal of this paper is threefold.

(1) First, we show how to compute efficiently satisfying assignments for general *boolean circuits*. This solves one of the main open problems from [8], where only the case of contact networks is studied. This improvement is important, for example, it is the basis of the recent result that

*Supported in part by NSF CCR–9304718.

†on leave from Mathematical Institute AV ČR, Žitná 25, 115 67 Praha 1, Czech Republic

the Data Encryption Standard (DES) could be broken in 3-4 months of DNA experiments [3]. (Section 4.1)

We actually prove more. We show how to find satisfying assignments not just for circuits, but a for more powerful model: *boolean circuits with ROMs*. This generalization is critical to the above DES result. A circuit with ROMs is a circuit that has the additional ability to do table lookups. The size of such a generalized circuit is the number of gates plus the number of bits in the table. Clearly, the power of such circuits is polynomially related to the usual model. However, this change is important for practical computations. That is the reason that ROMs are used in real circuits. (Section 4.2)

We also show how to solve directly NP-hard *optimization problems* like MAX-Clique or MAX-Circuit-Satisfiability (given a circuit, find a satisfying assignment in which the largest number of variables are set to true). (Section 4.3)

As a last improvement, we show how the computation can be done more efficiently if we know that all the satisfying assignments fall in some simple subset, like a regular language. (Section 4.4)

(2) Second, we show that DNA may even be able to handle computations that go beyond NP. We show that it is possible to use DNA to do approximate counting of satisfying assignments. Actually, it can even do more: it can compute any predicate from the polynomial hierarchy. Note, this is weaker than results that show that DNA can simulate PSPACE. The advantage of our result is a practical one: we use vastly fewer strands of DNA. (Sections 4.1 and 4.5)

(3) Our last goal is to present a uniform model and survey the many results and claims about the computational power of DNA. This is important if we are to eventually be able to understand the power of DNA based computers. (Sections 2 and 3)

2 The Results

Table 1 describes the state of the world as we understand it. Solving the satisfiability problem for circuits means to decide if there is a satisfying assignment of a boolean circuit presented to us, i.e. to decide if it is possible to set the inputs of the circuit so that the circuit computes 1; similarly for other devices (formula, contact network). A further description of each entry is given below.

Each result is "rated" in two ways: how many *biological steps* does it take and how many *DNA strands* does it use? Some methods are theoretical in that they use too many strands. For example, Adleman's original method for DHP is not efficient: it uses $n!$ strands. The same is true for the results on PSPACE: they use at least 2^{2n} strands.

Rating algorithms on these attributes is not new. This already occurs in the area of parallel algorithms. A parallel algorithm must be fast, i.e. take few parallel steps. However, it also must use relatively few parallel processors. Thus, an algorithm that takes $O(\log n)$ steps but uses n^4 processors is not practical. Since strands of DNA are used as parallel processors it is natural to rate them in this dual manner.

Our classification of algorithms is twofold. First, we classify them into three categories according to how the volume changes during the computation. In *Decreasing Volume Algorithms* the number of strands in a test tube decreases as the algorithm executes, *Constant Volume Algorithms* maintain the number of strands constant throughout the computation, and *Mixed Algorithms* are those fitting in neither of the previous classes. Second (not shown in Table 1), we say that an algorithm is

Problem	Bio Steps	Strands
Decreasing Volume Algorithms		
(1) Directed Hamiltonian path	$O(n)$	$n!$
(2) Any NP problem via (1)	$O(n^c)$	2^{n^c}
(3) CNF-formula satisfiability	$O(s)$	2^n
(4) MAX-CNF-Satisfiability	$O(s)$	2^n
Constant Volume Algorithms		
(5) Circuit satisfiability	$O(s)$	2^n
(6) Circuit with ROMs satisfiability	$O(s)$	2^n
(7) MAX-Circuit-Satisfiability	$O(s)$	2^n
(8) 1-tape NTM	$O(t)$	2^N
(9) Circuits via (8)	$\Omega(s^2)$	2^n
Mixed Algorithms		
(10) Contact network satisfiability	$O(s)$	2^n
(11) PSPACE	$O(s)$	2^{2s}
(12) Polynomial Hierarchy	$O(s)$	2^n

Table 1: Main results. **New results** are bold. We use s to denote the size of the computation (formula, circuit, etc.) being simulated.

Uniform if the following condition holds in every test tube throughout the computation: any two different strands have the same number of copies in the test tube. We classify the algorithms as constant volume and/or uniform even if the balance can be destroyed temporarily but is restored after a constant number of biological steps (typically during some complex operation).

All our algorithms are uniform, of the previous work the results (10) and (11) (contact networks and PSPACE) use non-uniform algorithms.

All the results, old and new, assume that the biological operations are *perfect*. It is important to know what the ultimate limits are without errors – if in the error-free case DNA cannot do exciting things, then there is no hope in the realistic case. However, in practice the operations are not prefect and errors must be addressed. The classification according to the volume and uniformity turns out to be very important in the context of resistance to errors – decreasing volume and uniform algorithms are significantly better than others in this respect, whereas mixed volume and nonuniform algorithms are hard to deal with; see [4, 5] for a further discussion of this. Let us point out that all our algorithms are very good in this respect, since they are all uniform and only the polynomial hierarchy result uses a mixed volume algorithm.

Here are some more specific comments about the results from Table 1:

(1) and (2): The first is the famous result of Adleman that shows that Directed Hamiltonian Path problem can be solved by a DNA based computer [1]. His method also implies the same for any NP problem. However, the difficulty with these results is that they use too many strands of DNA to be practical for large scale problems. Actually, this method is better than $n!$. If the degree of each vertex is c, then it requires at most c^n strands. This is still too many strands: for problems that are difficult for conventional machines, the number of strands needed is much too large.

(3) and (10): These are the results of Lipton that SAT (satisfiability for formulas in conjunctive normal form) and more generally contact network satisfiability (which includes general boolean formula satisfiability) can be done in time linear in the size of the formula (network). The key improvement in this method is that the number of strands needed is 2^n where n is the number of variables.

(5) and (6): These results show that DNA can find satisfying assignments for circuits or even generalized circuits, improving (3) and (10). The generalized circuits can access ROM, a "read-only-memory" or table. The ability to do such table lookups does not change the power of the model. It is, however, important since constants matter. Since DNA steps are slow reducing the number of steps by even a factor of 2 is critical. The use of tables plays a major role in our result of breaking DES in months, for example.

(4) and (7): It is possible to extend results (3) and (6) to handle the case of corresponding *optimization problems*: solving MAX-CNF-Satisfiability means finding a satisfying assignment for a CNF formula where the largest number of variables is set to true. MAX-Circuit-Satisfiability means the same for a general boolean circuit. This is easy to do via binary search using the results (3) and (6). However, the point is that we can avoid any slow down at all. This is a recurrent theme throughout our work: *Constants Matter!* In DNA based computers since the number of strands is limited and the steps are very slow one must be very careful to avoid certain "standard" tricks. If these tricks increase strands or steps greatly they may make a result totally impractical.

(8) and (9): The first result shows that DNA can simulate a 1-tape nondeterministic Turing machine. Here t means the time and N the number of nondeterministic bits used by the T. Since N can be much higher than n, the number of strands used may be too high. The latter result points out another reason that 1-tape NTM's are mainly of theoretical interest. 1-tape NTM's cannot in general do better than $\Omega(s^2)$ for simulation of circuits as there are many quadratic lower bounds on the time required for 1-tape TM's to do even simple tasks. We feel that, therefore, this result is not of great practical importance. Yet it continues to be discovered, see for example Beaver [2], Papadimitriou [9], Rothemund [11], Smith and Schweitzer [12].

(11): This is the result of Beaver [2] and Papadimitriou [9] that it is possible to simulate PSPACE with DNA operations. This, too, is mainly a theoretical result. The problem is that to do a PSPACE computation that takes at most space s, it requires $2^{2s} \geq 2^{2^n}$ strands of DNA, since the key doubling step used is too inefficient.

(12): This result shows how to simulate the polynomial time hierarchy. Note, it is much stronger than (11) since it uses only 2^n strands of DNA; the uniformity of our algorithm is a big advantage as well.

Next we compare the results based on the operations that they use. In our model of DNA computation, as introduced by Lipton in [8] and described in more detail in Section 3, a computation is just a sequence of test tubes. Each test tube contains many strands of DNA that encode certain computations. Each subsequent test tube is created from earlier ones by some biological operation from Table 2.

Table 3 summarizes the operations that are used by each of the previous results. Heat/Cool is used for each extract and some form of Amplify always needs to be used to prepare the initial tube and auxiliary tubes for some other operations – these occurrences are not included in the table below.

Operation	Comments
Extract	Extracts strands with given pattern
Length	Separate the strands by length
Pour	Pour two test tubes into one to do union
Heat/Cool	Used to break apart or put together
Amplify	PCR used to make copies
Cut	Restriction Enzymes used to cut DNA

Table 2: Allowed operations.

Problem	Extract	Length	Pour	Heat/Cool	Amplify	Cut
(1) Directed Hamiltonian path	yes	yes	no	no	no	no
(2) Any NP problem via (1)	yes	yes	no	no	no	no
(3) CNF-formula satisfiability	yes	no	yes	no	no	no
(4) MAX-CNF-Satisfiability	yes	yes	yes	no	no	no
(5) Circuit satisfiability	yes	no	yes	yes	no	no
(6) Circuit with ROMs satisfiability	yes	no	yes	yes	no	no
(7) MAX-Circuit-Satisfiability	yes	yes	yes	yes	no	no
(8) 1-tape NTM	yes	no	yes	yes	yes	no
(9) Circuit satisfiability via (8)	yes	no	yes	yes	yes	no
(10) Contact network satisfiability	yes	no	yes	no	yes	no
(11) PSPACE	yes	no	yes	yes	yes	yes
(12) Polynomial Hierarchy	yes	no	yes	yes	yes	yes

Table 3: Operations used in the results.

One of the exciting questions that is still open is what other operations are possible and how do the operations tradeoff among each other?

3 DNA Model of Computation

A DNA strand is essentially a sequence (polymer) of four types of nucleotides distinguished by one of four bases they contain; the bases are denoted A, C, G, T. The two ends of the strand are distinct and are conventionally denoted as 3' end and 5' end. Two strands of DNA can form (under appropriate conditions) a double strand, if the respective bases are Watson-Crick complements of each other – A matches T and C matches G; also 3' end matches 5' end. (For more discussion of the relevant biological background and the model see e.g. [3].)

We use a simple notation to explain the various operations to be performed on DNA. Given a string x over the alphabet $\{A, C, G, T\}$ we denote by $\uparrow x$ the single stranded DNA which is made up of the letters of x oriented from the 5' end to the 3' end (the first letter of x is on the 5' end). We denote by $\downarrow x$ the Watson-Crick complement of the strand $\uparrow x$. When $\downarrow x$ and $\uparrow x$ anneal to each

other they form a double strand which we denote by $\updownarrow x$.

Example:

$\uparrow$`ACCTGC` represents the single stranded DNA molecule `5'-ACCTGC-3'`.

$\downarrow$`ACCTGC` represents the single stranded DNA molecule `3'-TGGACG-5'`.

$\updownarrow$`ACCTGC` represents the double stranded DNA molecule $\begin{smallmatrix}\texttt{5'-ACCTGC-3'}\\ \texttt{3'-TGGACG-5'}\end{smallmatrix}$.

3.1 Biological Operations

Our fundamental model of computation is to apply a sequence of operations to a set of strands in a test tube. The operations that we make use of are derived from the following experiments commonly used in molecular biology today [1].

It is possible to dissolve the double strands into single strands by heating the solution. This process is referred to as *melting*. The reverse process when the complementary strands anneal is performed by cooling the solution.

Using restriction enzymes, it is possible to cut the strands at some distinctive marker.

Using a gelling technique called gel-electrophoresis [1] it is possible to separate the DNA strands by length.

It is possible to detect if there is a DNA strand in a test tube and to sequence a given strand (i.e., to "read" the sequence of bases of the strand).

Some more difficult experiments are described below.

3.1.1 Extract

We need the ability to extract from a test tube all strands that contain any specific short nucleotide sequence. To accomplish this we use the method of biotin-avidin affinity purification as described in [1]. This technique works in the following way. If we want to extract all strands containing the sequence $\uparrow x$, then we first create many copies of its complementary oligo (a short DNA strand), namely $\downarrow x$. To these oligos we attach a biotin molecule, which are in turn anchored to an avidin bead matrix. If we then melt the double strands in our test tube and pour them over this matrix, those single strands that contain $\uparrow x$ will anneal to the $\downarrow x$ oligos anchored to the matrix. A simple wash procedure will whisk away all strands that did not anneal, leaving behind only those strands that contain $\uparrow x$, which can then be retrieved from the matrix. We refer to this operation as an *extract* using beads of type $\downarrow x$.

3.1.2 Polymerization via DNA Polymerase

Given a particular single strand of DNA, we may wish to create its Watson-Crick complementary strand. To do this we use the enzyme DNA polymerase. DNA polymerase will "read" the given strand, called the template strand, in the $3' \rightarrow 5'$ direction and build the complementary strand in the $5' \rightarrow 3'$ direction, one nucleotide at a time. In order to work, DNA polymerase actually requires that there be a short portion of the template that is double stranded, and it is onto the end of this short complementary piece, called the *primer*, that the enzyme will add the new nucleotides. For example, if we have some strand $\uparrow xyz$, DNA polymerase cannot create its complement. However,

if we add $\downarrow z$ to the solution and let it anneal to $\uparrow xyz$, we obtain $\uparrow xy \updownarrow z$, and DNA polymerase will be able to add nucleotides onto the free 3' end of z to create $\updownarrow xyz$. Note that because DNA polymerase only works in one direction, the partial duplex $\uparrow x \updownarrow y \uparrow z$ will yield $\updownarrow xy \uparrow z$ and not the full duplex $\updownarrow xyz$.

3.1.3 Amplification via PCR

At times we need to make copies of all the DNA strands in a test tube. This can be done with a straightforward application of the polymerase chain reaction (PCR). PCR is a process that uses DNA polymerase to make many copies of a DNA sequence. PCR works in the following way. If we have the duplex $\updownarrow xyz$, we first melt it to form $\uparrow xyz$ and $\downarrow xyz$. To this solution we will add the primer oligos $\downarrow z$ and $\uparrow x$, which anneal to form the partial duplexes $\uparrow xy \updownarrow z$ and $\updownarrow x \downarrow yz$. DNA polymerase can then elongate the primers to create full duplexes of the form $\updownarrow xyz$. Note that we now have *two* copies of our original strand. If we just repeat this process, we will again double the number of copies of the original strand in solution. Soon we will have four copies, then eight, then sixteen, and so on, until we have enough copies for our purposes. 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. We call this operation *amplify*.

3.2 Representing Binary Strings

DNA strands can be used to represent binary strings as was shown in [8]. Let $x = x_1 \ldots x_n$ be an n-bit binary string. The idea is to assign a unique sequence of 30 bases (a 30-mer) to each bit position and bit value. The DNA strand representing the binary string $x \in \{0,1\}^n$ is

$$\updownarrow S_0\, B_1(x_1)\, S_1\, B_2(x_2)\, S_2\, , \ldots , S_{n-1}\, B_n(x_n)\, S_n,$$

where

1. $B_i(0)$ is the 30-mer used to encode the fact that the i-th bit of x is 0. Similarly, $B_i(1)$ is the 30-mer used to encode the fact that the i-th bit is 1.
2. S_i is a 30-mer which used as a separator between consecutive bits ($0 \leq i \leq n$).

It is crucial that the strings $B_i(x)$, S_i, and their complements are distinct. In fact, it is desirable that no two of them contain a long common substring. This can be achieved either by using the words of some good code, or by choosing these words randomly (the length of 30 bases is sufficient to make the random construction very reliable).

From now on we will freely switch between a binary string and the DNA strand representing it.

To initialize our algorithms, we create a test tube of DNA strands representing all 2^n binary strings of length n. This is done by forming all paths in the graph of Figure 1 using the method of Adleman [1].

3.3 Operations on Binary Strings

The key feature of this representation is that it enables us to extract all strings whose i'th bit it 1 by extracting all strands containing the DNA strand $B_i(1)$, using the biological extraction described

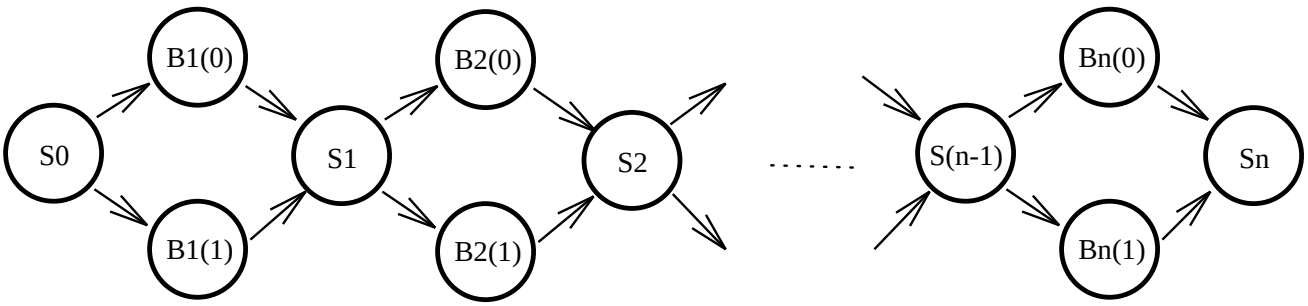

Figure 1: Initialization Graph

above. We call this operation extract.

Another simple operation is to discard variables $x_i, i > m$ by cutting the strings at the marker S_m and extracting the parts containing the marker S_0; we use this to compute functions in polynomial hierarchy.

We need two new operations beyond [8].

3.3.1 Append

The first new operation is *append*, we use it to compute circuit satisfiability and functions in polynomial hierarchy. This operation takes a test tube T of DNA strands representing n bit strings (i.e., ending by S_n) and adds the string representing the bit 0 (resp. 1) to all strands. There are several ways of carrying out this operation.

We propose two possible procedures. The first one uses the EcoRI restriction enzyme. This enzyme recognizes the sequence `GAATTC`. Suppose the separator S_n ends with this sequence, $S_n = S$ `GAATTC`. Then the strands in the test tube are $\updownarrow y_1 \ldots y_n\ S$ `GAATTC`, for some $y_1, \ldots, y_n$. When the EcoRI enzyme is allowed to act in the test tube the strands are cut at the restriction site and become $\updownarrow y_1 \ldots y_n\ S$ `G` $\uparrow$ `AATT`.

The EcoRI enzyme leaves a sticky end. To append the sequence $B_{n+1}(0)\ S_{n+1}$ to all strands we put many copies of the single stranded DNA molecule $\downarrow$ `AATTA` $B_{n+1}(0)\ S_{n+1}$ in the solution. These strands will anneal to the sticky end of strands in the solution. A polymerase enzyme can then be used to complete the strands to full double stranded DNA. The resulting strands in the solution are $\updownarrow y_1 \ldots y_n\ S$ `GAATTA` $B_{n+1}(0)\ S_{n+1}$.

The separator S_n has now changed to S `GAATTA`. Note that the string adjacent to S is now `GAATTA` which is not a recognition site for the EcoRI enzyme. This means that the enzyme will not cut any separator S_i other than the one at the end of the strand.

The second procedure for *append* uses only extracts and heat/cool, not restriction enzymes. On the other hand it needs more steps, and hence it might be slower.

1. Prepare a test tube T' containing many copies of the DNA strand $\downarrow S_n\ B_{n+1}(0)\ S_{n+1}$. (Or prepare in advance a tube with corresponding double strands and extract the lower strands now.)

2. Heat the contents of T (to melt all the double stranded DNA strands to single stranded ones)

and extract the strands containing $\uparrow S_n$ (i.e., all the upper strands).

3. Pour together the result of the previous step and T', and let the solution cool down to allow the strands to anneal. The resulting strands have the form $\uparrow y_1 \ldots y_n \updownarrow S_n \downarrow B_{n+1}(0)\ S_{n+1}$.

4. Use a polymerase enzyme to complete these strands to double stranded DNA molecules of the form $\updownarrow y_1 \ldots y_n\ S_n\ B_{n+1}(0)\ S_{n+1}$.

5. Extract all strands that contain S_0 (thus no superfluous unattached strands from T' are left in the solution).

3.3.2 Set-extract

The second new operation is a generalized extract, which we call *set-extract*; this operation is used only for computing functions in the polynomial hierarchy. It is not used by the other results in this paper.

The set-extract operation takes two test tubes T and T' and produces a test tube containing all strands from T that do not appear in T'. For implementation we need to assume that if a strand appears in T', it has more copies in T' than in T; as we shall see, it is always possible to guarantee this condition in our case. We propose the following procedure:

1. Heat T' to get single strands and extract all the strands containing $\uparrow S_0$. Heat T and extract all the single strands containing $\downarrow S_0$. (This keeps one single strand of each double strand DNA and guarantees that only pairs of one strand from T and one strand from T' can anneal later.)

2. Pour together the extracted strands from T and T' and cool the solution to let the strands anneal.

3. Extract all strands containing $\uparrow S_0$. (Note that the strands that are both in T and T' have formed double strands and will therefore not stick to the avidin bead matrix. This means that the only strands that will be extracted are those in T which are not in T'.)

4. Complete the extracted strands to full double strand DNA molecules using a polymerase enzyme.

In step 2, it is possible that a strand from T does not anneal to a strand from T' even if it can. Therefore in practice this procedure may have to be repeated several times to make sure that the resulting solution contains no strands from T'.

4 New Results

4.1 Circuit Satisfiability and Approximate Counting

Theorem 4.1 *The circuit satisfiability problem can be solved with 2^n strands and $O(s)$ biological steps, where n is the number of variables and s is the size of the circuit.*

Proof Sketch We start as in Lipton [8] with a test tube of DNA strands that code all 2^n possible input bit sequences $x_1 \ldots x_n$. Inductively we will produce a test tube that contains DNA strands of the form $x_1 \ldots x_n\ y_1 \ldots y_k$ where $y_1, \ldots, y_k$ are the values of the first k gates of the circuits.

We first show how to add the next gate. Suppose the gate is $y_i \vee y_j$; the same method works for all the other cases. We use extraction to form four test tubes: $T_{00}, T_{01}, T_{10}, T_{11}$ where T_{uv} contains all the strands that have $y_i = u$ and $y_j = v$. Now use the append operation to add 0 to all the strands in the first three test tubes, and append 1 to all the strands in the last test tube. Finally we pour all the test tubes together.

Let $\mathcal{C}$ be some circuit. We run through the process described above for all the gates, and finish with a test tube which contains DNA strands representing binary strings of the form $x_1 \ldots x_n\ I\ y$ where $x_1 \ldots x_n$ is an input to the circuit, I is a string of bits representing intermediate values of gates in $\mathcal{C}$, and y is $\mathcal{C}(x_1, \ldots, x_n)$. We now extract all strands that have $y = 1$ and obtain a set of strands which correspond to satisfying assignments. In particular we can test if there is any strand, which solves the circuit satisfiability problem. □

In fact the above procedure gives something much stronger than just a satisfying assignment. Throughout the procedure we maintain the fact that the relative frequency of the DNA strands corresponding to each of 2^n possible assignments to $(x_1, \ldots, x_n)$ is the same. The consequences of this are twofold.

First, it follows that the frequency of every satisfying assignment in the final test tube is the same. We can take a random DNA from the test tube and read from it a satisfying assignment. This enables us to perform random sampling on the set of satisfying assignments. Once we can do random sampling of satisfying assignments, it is well-known we can also approximately count their number [6]. While for some problems approximate counting is known to be in random polynomial time (mostly using rapidly mixing Markov chains, see e.g. [7] and references therein), in general it is only known to be in $\Sigma_2 \cap \Pi_2$. Hence this is stronger than just finding a satisfying assignment, which is in NP.

The second consequence is that our method is uniform and constant volume. This means that this method is potentially more feasible and error resistant than the previous ones. If we are allowed to pour together arbitrary test tubes, the frequencies of different assignments can differ, which could cause problems for implementation of the contact networks algorithm from [8] (however note that the algorithm for CNF from the same paper is uniform and even decreasing volume). For further discussion of this topic see [5].

4.2 Circuits with ROMs

Next we show how to implement any function $f : \{0,1\}^m \to \{0,1\}$ in very few steps when m is small. In many cases this will be more efficient than implementing the function using binary gates. The implementation is very similar to performing a table lookup to find the value of the function. We therefore regard this improvement as an implementation of a ROM. Whenever a large circuit has to be evaluated, one may replace parts of it by a ROM and thus save on the overall number of biological steps it takes to evaluate the circuit.

Theorem 4.2 *Let $\mathcal{C}$ be some circuit containing ROMs. Let s be the size of $\mathcal{C}$ and n be the number of inputs. It is possible to decide the satisfiability of C using 2^n strands and $O(s)$ biological steps.*

Proof We only need to extend the proof of Theorem 4.1 to show how to evaluate ROMs. As before, inductively we will produce a test tube T that contains DNA strands of the form $x_1 \ldots x_n y_1 \ldots y_k$ where $y_1, \ldots, y_k$ are the values of the first k gates of the circuit. We now wish to append to all strands in T the value of $f(z_1, \ldots, z_m)$ for some $z_i \in \{x_1, \ldots, x_n, y_1, \ldots, y_k\}$. This is done by separating T into T_0 and T_1 according to the value of $f(z_1, \ldots, z_m)$.

To separate T into T_0 and T_1 quickly we need to assume that the variables $z_1, \ldots, z_m$ are physically adjacent on the DNA strands in T (if the variables are not adjacent, we can copy them to the end with $O(m)$ biological steps, using append). For small m, e.g. $m \leq 6$, it is possible to extract all strands in T which satisfy $f(z_1, \ldots, z_m) = 1$ (resp. $f(z_1, \ldots, z_m) = 0$) in one step. This biological procedure is explained in detail in [3].

Once we obtain the solutions T_0 and T_1 we may append i to all strands in T_i for $i = 0, 1$. We then recombine T_0, T_1 into one test tube T'. This proves the inductive statement of Theorem 4.1 for a ROM and therefore completes the proof. □

4.3 Optimization problems

Our next result shows how to directly solve any optimization problem that involves finding the largest (or smallest) set that satisfies a certain property in P. This includes optimization problems such as MAX-Clique, MIN-Set-Cover, Shortest-Vector in a linear code, and others.

In general, we define the MAX-Circuit-Satisfiability problem to be the problem of finding the maximal Hamming weight (number of 1's) of a satisfying assignment to a given circuit. We now show how to directly solve this optimization problem.

The results of the previous section already imply these results in an inefficient manner. For instance, the ability to solve the satisfiability problem enables to test whether a clique of size k exists in the graph. To find the largest clique we can perform a binary search on the values of k. The problem with this approach is that it requires us to run a long bio-experiment several times. Even if we do the binary search on the tube of all satisfying assignments obtained as in the previous section, we need additional $n \log n$ biological steps. Thus the direct solution below is significantly more efficient.

Theorem 4.3 *MAX-Circuit-Satisfiability can be solved with 2^n strands and $O(s)$ biological steps, where n is the number of variables and s is the size of the circuit.*

Proof We modify the model of coding assignments by DNA strand introduced in Section 3 as follows. We encode $B_i(0)$ and S_i by 20-mers (DNA strands of length 20) and $B_i(1)$ by 30-mers. Thus the length of a strand representing a binary string with k ones in it is

$$20(n+1) + 30k + 20(n-k) = 20(2n+1) + 10k.$$

To solve an instance of MAX-Circuit-Satisfiability problem we create a tube of DNA strands representing all 2^n assignments. This is done by forming all paths in the graph of Figure 1 using the method of Adleman [1]. The length of the DNA strand representing an assignment with m 1's is $20(2n+1) + 10m$.

We apply the algorithm from Theorem 4.1 to obtain a tube containing DNA strands representing all the satisfying assignments. (For MAX-CNF-Satisfiability we use Lipton's algorithm from [8].)

We now use the gelling technique described in [1] to separate the DNA strands by length. The longest DNA strand corresponds to a maximal satisfying assignment. □

The simplicity of this algorithm is perhaps best seen on the example of the MAX-Clique problem. Given a graph $G = (V, E)$ on n vertices, we encode the 2^n sets of vertices similarly as in Theorem 4.3. The length of the DNA strand representing the set $S \subseteq V$ is $20(2n + 1) + 10|S|$. Now we filter out all DNA strands that represent sets that are not cliques in G. This can easily be done by looping over all non-edges $e = (u, v)$ of G and throwing away those sets that contain both u and v. At the end of the process we separate the DNA strands by length. The longest DNA strand corresponds to a maximal clique in G. This process requires only $n^2 - |E|$ biological steps.

4.4 Using State Automata

As was explained in the introduction, molecular computers can be thought of as vastly parallel machines. However, each step of a molecular machine takes a long time, e.g. several hours. It is thus crucial to try and save as much as possible on the number of steps it takes to solve a given problem.

Towards this goal we show how one can reduce the number of biological steps by replacing parts of the circuit by a state automaton. The idea is to replace the initialization graph of Figure 1 with a more complicated graph. For example, instead of generating all string in $\{0, 1\}^n$ we can generate only strings $\bar{x} \in \{0, 1\}^n$ such that $x_1 \oplus \cdots \oplus x_n = 0$. This can be used to reduce the size of the circuit being evaluated since the circuit need not test the parity of the input. Even though we use a more complicated version of the graph in Figure 1, the number of biological steps it takes to form all paths in the graph is unaffected. We only increase the number of different initial strands proportionally to the size of the initialization graph, which is a very small penalty.

A typical application is the following problem: Given a graph $G = (V, E)$ with $|V|$ odd, find a 3-coloring of G with an even number of red vertices. The method of [8] can be used to solve this problem in the following way: first create a set of DNA strands representing all strings in $\{R, G, B\}^{|V|}$. This can be done using a graph similar to the one shown in Figure 1. Then filter out all strands that represent illegal colorings of G. Finally, filter out all strands representing colorings with an odd number of red vertices. This last step takes $O(|V|)$ steps. Had the initial set of DNA strands only represented strings with an even number of red vertices this last filtering step would be unnecessary. This can be done by modifying the graph of Figure 1 to only generate strings in $\{R, G, B\}^{|V|}$ with an even number of R's. Simply at each level of the graph split the nodes S_i into two nodes S_i', S_i'' that keep track of the parity of the number of R's so far.

This technique can be generalized to more complicated graphs. Clearly the initial soup can be made to contain strings in any regular language. Let A be an automata with k states. The idea is to split each separator node S_i into k nodes $S_i^{(1)}, \ldots, S_i^{(k)}$. These nodes will keep track of the state of the automata.

We can extend this technique beyond regular languages by using state automata with a non-constant (but relatively small) number of states. An example would be to find a 3-coloring with equal number of red, blue and green vertices for a graph with $n = 3m$ nodes. The state machine which keeps count of the red and blue vertices has $O(n^2)$ states, hence the initialization graph has $O(n^3)$ vertices. As stated above, the size of the initialization graph does not influence the number of biological steps needed for the initialization.

In another variation of this technique, we can use the separation of DNA strands of various lengths. For example, suppose we want to find a 3-coloring with at most 10 red vertices. Similarly as in Theorem 4.3 we encode red vertices by shorter strands than the other ones and after forming all the sequences we use the length separation to extract the sequences we want. This time we use a single extra biological step, with no penalty in the size of the graph at all.

4.5 Polynomial Hierarchy

For computing functions in polynomial hierarchy we also need the operations cut and set-extract described in Section 3.

Theorem 4.4 *Let f be a function in kth level of the polynomial hierarchy, where the polynomial predicate in its definition has n variables (including the ones bound by the quantifiers) and can be computed by a circuit of size s. Then f can be computed with 2^n strands and $O(k+s)$ biological steps.*

Proof Sketch Suppose we have a predicate $Q = \forall y \exists z P(x, y, z)$, where the total length of x, y, and z is n, and P can be evaluated by a circuit of size s. We will show how to obtain a test tube containing DNA strands coding exactly all the boolean assignments to x for which the predicate Q is true.

Using the algorithm from Theorem 4.1, we obtain a test tube T containing all the satisfying assignments (x, y, z, g) for $P(x, y, z)$, where g are the values of the gates of the circuit computing P. Using the cut (and extract), we cut each DNA into parts for (x, y) and (z, g), and retain only the (x, y) parts. This gives us a test tube T' containing all the assignments (x, y) for which $\exists z P(x, y, z)$ is true. Now we take a tube u containing all (x, y) assignments and using set-extract we remove all assignments in T'. This gives T'' with all the assignments (x, y) for which $\neg \exists z P(x, y, z)$ is true. Using cut again, we retain only x parts of the strands, thus getting T''' with all the assignments x such that $\exists y \neg \exists z P(x, y, z)$. Finally, we use a test tube of all assignments x and set-extract T''' from it to get all the assignments such that $Q = \neg \exists y \neg \exists z P(x, y, z)$ is true.

The case of more quantifiers is similar – we need one cut and one set-extract for each quantifier.

When discussing implementation of set-extract, we have assumed that a frequency of any assignment in the extracted tube is larger than in the tube from which we extract. The tube from which we extract always has a uniform distribution on all assignments. The extracted tube is created from a tube with uniform distribution (on a larger set of assignments, since there are more variables) by discarding a segment of each strand; this can cause the frequency of some assignment to increase but it will never go down. Since there is only a small constant number of these steps, we can adjust the volume of the tube from which we extract (using amplification), to satisfy the condition. □

5 Conclusions

The main result of this paper is that DNA based computers can be used to solve the satisfiability problem for circuits. The algorithm presented is considerably more efficient than simulating a NTM using DNA as was suggested by [2, 10, 11, 12]. Furthermore we showed how to improve the performance of the algorithm by using state automata and ROMs. For optimization problems such

as MAX-Clique we showed a technique for solving the problem directly without first solving several decision problems. We also showed that the algorithm can be extended to perform approximate counting and even compute functions from polynomial hierarchy efficiently.

There are still many issues to be considered. Foremost is the issue of errors. DNA molecules are known to be fragile; they dissolve in water and easily break. Steps towards coping with errors were taken in [4, 5]. Let us point out that our algorithms seem to be more feasible and resistant to certain kind of errors than most of the previous ones, since they are uniform. It is essential to obtain a better understanding of the type of errors which may occur and to come up with further techniques for error recovery.

References

[1] L. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, Nov. 11, 1994.

[2] D. Beaver. A universal molecular computer. Technical Report CSE-95-001, Penn State University, 1995.

[3] D. Boneh, C. Dunworth, and R. J. Lipton. Breaking DES using a molecular computer. Technical Report CS-TR-489-95, Princeton University, 1995.

[4] D. Boneh and R. J. Lipton. Making DNA computers error resistant. Technical Report CS-TR-491-95, Princeton University, 1995.

[5] D. Boneh, R. J. Lipton, and J. Sgall. Error resistant and uniform DNA computers. in preparation.

[6] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Comput. Sci.*, 43:169–188, 1986.

[7] R. Kannan. Markov chains and polynomial time algorithms. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 656–671. IEEE, 1994.

[8] R. J. Lipton. Using DNA to solve NP-complete problems. *Science*, 268:542–545, Apr. 28, 1995.

[9] C. Papadimitriou. private communications.

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

[11] Rothemund. private communication.

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