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@mbox.cesnet.cz
Mathematical Institute, AV ČR
Žitná 25
115 67 Praha 1
Czech Republic

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.

1 Introduction

In the very short history of DNA (deoxyribonucleic acid) based computing there have already been a number of exciting results. First was Adleman's [1] beautiful insight that biological experiments could solve the Directed Hamiltonian Path problem (DHP). Then Lipton [12] 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 boolean formulas.

Since then there has been a number of papers on DNA computation. Most of these results are 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.

*Supported in part by NSF CCR–9304718.

†Partially supported by grants A119107 and A1019602 of AV ČR. Part of this work was done at Institute of Computer Science, Hebrew University, Jerusalem, Israel; partially supported by Golda Meir Postgraduate Fellowship.

The goal of this paper is twofold.

First, we significantly generalize the previous results on simulating classical computations using DNA.

We show how to compute efficiently satisfying assignments for general *boolean circuits* with arbitrary binary gates. This solves one of the main open problems from [12], where only the case of contact networks is studied. We also show that it is possible to use DNA to do *approximate counting* of satisfying assignments, which means that DNA can be used to do computations that go beyond NP. (Section 4.1)

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 4.2)

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 4.3)

Our second goal is to present a standard framework 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)

In this paper we 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. For the sake of completeness we note that several researchers have already begun to take steps to make DNA algorithms more noise tolerant [8, 10, 3].

2 Comparison of several DNA computing results

Table 1 describes the results on simulation of various classical computational models using DNA computation.

Each result is "rated" in two ways: how many *biological steps* does it take and how many *DNA strands* does it use? 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.

By the number of DNA strand we understand the number of distinct strands that may appear in the same test tube during the course of the algorithm (e.g. our circuit satisfiability algorithm uses 2^n different strands in each test tube, but during the course of the algorithm these strands are modified by appending the intermediate results). This approximately corresponds to the volume of the test tube. To be more precise, we should also include the length of the strands; however this turns out to be much less relevant, as the length of the strands is usually linear in the size of the problem, while the number of strands is exponential. For the purposes of evaluating the "practicality" of DNA algorithms we assume that 10^{21} is an upper bound on the number of DNA strands that are available to an algorithm. It will be useful to keep in mind that $10^{21} \approx 2^{70}$.

By the number of steps we simply mean the total number of biological operations during the algorithm. We do not distinguish between the different operations here, even though the time needed for each of them may be very different. We discuss the different operations later in this section. We also do not account for the possibility of executing some of the operations in parallel, which would add additional dimension to the classification; we should note however that this is important in the mentioned DES result [6].

A further description of each entry is given below. 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.

Problem	Bio Steps	Strands
(1) Directed Hamiltonian path	$O(n)$	$n!$
(2) Contact network satisfiability	$O(s)$	2^n
(3) Circuit satisfiability	$O(s)$	2^n
(4) MAX-Circuit-Satisfiability	$O(s)$	2^n
(5) Regular-Circuit-Satisfiability	$O(s)$	2^n
(6) 1-tape NTM	$O(t)$	2^N
(6a) Circuit satisfiability via (6)	$\Theta(s^2)$	2^n
(7) Cellular Automata	1	$t \cdot S$
(8) PSPACE	$O(S)$	2^{2S}
(9) Polynomial Hierarchy	$O(s)$	2^n

Table 1: Main results. **New results** are bold. We use s to denote the size of the computation (circuit, contact network, etc.) being simulated. The various parameters in the table are explained in more detail in the text.

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

(1) This 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 via reductions. However, the difficulty with using this method for general NP problems is that it uses too many strands of DNA to be practical for large scale problems. For graphs with n vertices, and hence for the problems that reduce to them, the algorithm could require up to $n!$ different strands. For DHP it is, however, efficient, as there are actually $n!$ potential solution. Accordingly, encoding the problem via boolean circuits and using our new algorithms leaves us with $2^{\Theta(n \log n)}$ strands, as we would encode each node of the path by $\log n$ bits.

(2) This is the result of Lipton [12] 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 network (resp. formula). The key improvement in this method is that it works for a more general class of problems, while the number of strands is only 2^n where n is the number of variables.

(3): We show that using DNA we can efficiently find satisfying assignments for general boolean circuits with fan-in two (i.e. with arbitrary binary gates), which significantly improves the result from [12]. The fact that circuits are very efficient for a wide variety of problems and they are easy to design, unlike efficient Turing machines or contact networks, makes this result applicable for practical problems. In Table 1 the number of gates in the circuit is denoted by s and the number of input variables is denoted by n.

(4): We extend the results (2) and (3) to handle the case of corresponding *optimization problems*: solving MAX-Circuit-Satisfiability means finding a satisfying assignment for a boolean circuit here the largest number of variables is set to true. This is easy to do via binary search using the results (2) and (3). 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.

(5): The results (3) and (4) can be made more efficient by combining them with finite state machines. Let $L = L_1 \cap L_2$ be a set of n bit binary strings where L_1 can be recognized by a circuit with s gates and L_2 can be recognized by a state automaton with k states. We show that a DNA solution representing all strings in L can be constructed using $O(kn + s)$ synthesized oligos and $O(s)$ extraction steps. This generalization increases the efficiency of the algorithms presented in (3) and (4) since one can reduce the size of a circuit by implementing part of it as an automaton.

(6) and (6a): 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 Turing machine. The latter result points out the reason that 1-tape NTM's are mainly of theoretical interest. While 1-tape NTM's can do boolean circuit satisfiability in $O(s^2)$ time, it is in general impossible to do it in better time than $\Omega(s^2)$ 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. The result was discovered by several authors using various significantly different constructions. See for example Beaver [5], Papadimitriou [13], Rothemund [16], Smith and Schweitzer [18], Rooß and Wagner [15].

(7): This is a construction due to Winfree [22] which shows how complicated DNA patterns can be used to simulate cellular automata. The number of nucleotides used by the DNA pattern is proportional to the product of the space used by the automata (S) and the number of generations for which it is run (t). The attractive feature of this model is that computations are done in vitro. Unlike other models, no intervention of a lab technician is required.

(8): This is the result of Beaver [5], Reif [14] and Papadimitriou [13] that it is possible to simulate PSPACE with DNA operations; S denotes the space needed. This, too, is mainly a theoretical result. The problem is that these methods use biological operations that are likely to be impossible to implement in practice. Very roughly speaking, these operations expect that strands of DNA will anneal with their exact counterparts, in parallel for all different strands in the test rube. It seems to be the case that for this to be feasible in constant time, the number of copies of each strand needs to be of the same order of magnitude as the number of distinct strands. Using 2^{70} as the bound on the number of available strands it follows that one can use these methods to run algorithms which require at most about 35 bits of space. Such algorithms can be easily simulated on conventional machines. Hence, any problem that can be solved in DNA using this technique could also be solved on a conventional machine.

(9): This result shows how to simulate the polynomial time hierarchy; n now refers to the total number of all quantified variables. Unfortunately, like (8) it requires the manipulation of DNA in ways which is unlikely to work in practice. It appeared in the previous version of this paper [7], but we omit it in the current version because of our doubts about its practical relevance.

Next we compare the results based on the operations that they use. In our model of DNA computation, as introduced by Lipton in [12] 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. We describe the operations in more detail in the next Section. Now we classify them according to Table 2.

Operation	Meaning
Extract	Extract strands with given substring
Length	Separate the strands by length
Pour	Pour two test tubes into one, with no change of the individual strands
Amplify	PCR used to make copies of strands or selected subregions
Anneal	Represents all the operations that combine a test tube of single stranded DNA with other prepared strands and let them anneal together to form double strands
Cut	Apply a restriction enzyme to cut strands in test tube
Join	Represents the annealing steps combining two test tubes that are unlikely to be possible in practice (cf. (8) and (9) above)

Table 2: Basic types of operations used in current algorithms.

Table 3 summarizes the operations that are used by each of the previous results. In addition, the test tubes are converted from double strands to single strands and back by heating and PCR; also some form of Amplify and Anneal always need to be used to prepare the initial tube and auxiliary tubes for some other operations – these occurrences are not included in the table below. Similarly, we do not include the Amplify steps needed for the final test of presence of DNA.

Some results can be obtained using different set of operations. Thus, the algorithms in this paper can either use annealing, or can be implemented using restriction enzymes. Similarly, different variants of (6) use different building blocks.

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

Typically the results claim the ability to solve some NP-hard problems in polynomial number of biological steps. At this point it is important to stress what does it mean in practice. All the techniques that are used so far can be simulated on classical parallel machines with the number of processors proportional to the number of strands. Accordingly, the needed number of strands is exponential in the size of the problem. Due to the physical limitations the number of strands is limited, and hence this only means that DNA can help to solve instances of corresponding size. Again, this is the reason why we have to be careful to make the methods as efficient as possible.

It depends on the particular problem whether DNA computations have a good chance to compare favorably with classical problems or not. For example, for the MAX-Clique problem, there

Problem	Extract	Length	Pour	Amplify	Anneal	Cut	Join
(1) Directed Hamiltonian path	yes	yes	no	yes	no	no	no
(2a) CNF-formula satisfiability	yes	no	yes	no	no	no	no
(2b) Contact network satisfiability	yes	no	yes	yes	no	no	no
(3) Circuit satisfiability	yes	no	yes	no	yes	no	no
(4) MAX-Circuit-Satisfiability	yes	yes	yes	no	yes	no	no
(5) Regular-Circuit-Satisfiability	yes	no	yes	no	yes	no	no
(6) 1-tape NTM	yes	no	yes	yes	yes	yes	no
(7) Cellular Automata	no	no	no	no	yes	no	no
(8) PSPACE	yes	no	yes	yes	yes	no	yes
(9) Polynomial Hierarchy	yes	no	yes	yes	yes	no	yes

Table 3: Operations used in the results.

are algorithms achieving running time of about $2^{n/3}$ [20, 17], and these will be preferable to the approach presented here which needs 2^n strands of DNA. The methods presented in this paper can be combined with results of [4] to produce more efficient DNA algorithms for solving the MAX-Clique problem. They show that 1.51^n DNA strands are sufficient for solving the MAX-Clique problem in a graph with n vertices. In general, DNA algorithms work for any problem, and hence may be favorable for problems where no algorithms significantly faster than 2^n are known. An interesting example of an application where the use of DNA may be favorable to classical algorithms is the method for breaking DES [6].

To make these results practically applicable, it would be necessary to perform large-scale experiments to verify whether it is possible to perform the needed operations on such a scale as needed here, and give estimates of how long these operations will take. In this paper, we are not trying to answer these questions, rather, our goal is to motivate such experiments.

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. [6].)

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]. Here we present an idealized model which assumes that all the operations are error-free.

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. Usually we use double strands of DNA to store the information since the single strands are fragile. We convert them to single strands by heating as needed for other operations.

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. They are converted to double strands by PCR (see below), if needed. 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 $3' \to 5'$ direction and build the complementary strand in the $5' \to 3'$ 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 $\upharpoonright xyz$, DNA polymerase cannot create its complement. However, if we add $\downharpoonleft z$ to the solution and let it anneal to $\upharpoonright xyz$, we obtain $\upharpoonright xy \updownarrow z$, 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 $\upharpoonright x \updownarrow y \upharpoonright z$ will yield $\updownarrow xy \upharpoonright z$ 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 $\upharpoonright xyz$ and $\downharpoonleft xyz$. To this solution we will add the primer oligos $\downharpoonleft z$ and $\upharpoonright x$, which anneal to form the partial duplexes $\upharpoonright xy \updownarrow z$ and $\updownarrow x \downharpoonleft yz$. 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*.

3.1.4 Append

Sometimes we will need to elongate every strand in a test tube by tacking another short strand onto the end. If every strand in the tube is of the form $\updownarrow Xy$, where X is arbitrary and y is fixed, then this elongation can be accomplished in the following manner. We first perform an extract on $\upharpoonright y$, which will give us all the "top" strands of every pair—that is, we get every $\upharpoonright Xy$ and we discard every $\downharpoonleft Xy$. Then we introduce many copies of the single strand $\downharpoonleft yz$ into the solution, and allow these to anneal with the $\upharpoonright Xy$ strands. This results in partial duplexes of the form $\upharpoonright X \updownarrow y \downharpoonleft z$. We can then use DNA polymerase to fill in the rest of the duplex, giving us the full duplex strands $\updownarrow Xyz$. This is exactly what we wanted: every strand has been elongated by the addition of another short strand, in this case $\updownarrow z$. We call this operation *append*. We note that there are alternate methods for implementing append using restriction enzymes.

3.2 Representing Binary Strings

DNA strands can be used to represent binary strings as was shown in [12]. 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 used as a separator between consecutive bits ($0 \le i \le n$).

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. Our suggestion of using 30-mers should be regarded as an estimate. Adleman, in his original experiment, used 20-mers. It is an open research problem for experimental biologists to determine the appropriate oligo length to be used in DNA computations. 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].

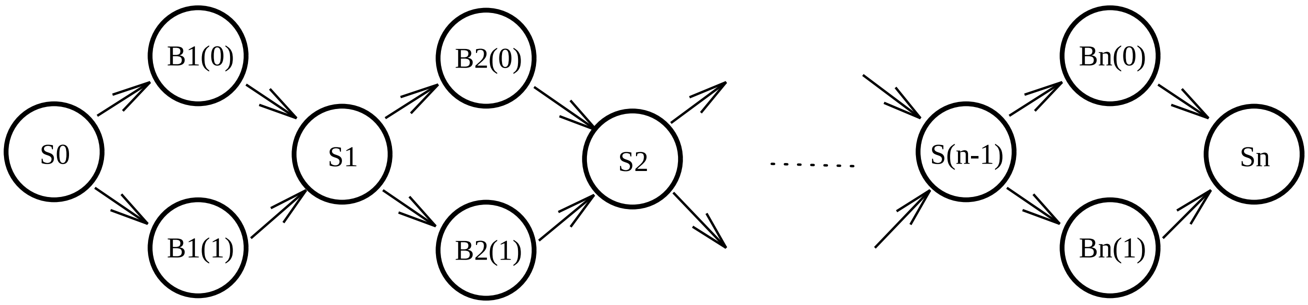

Figure 1: Initialization Graph

The key feature of this representation is that it enables us to extract all strings whose i'th bit has value $j \in \{0, 1\}$ by extracting all strands containing the DNA strand $B_i(j)$, using the biological extraction described above.

4 New Results

4.1 Circuit Satisfiability and Approximate Counting

Theorem 4.1 *The circuit satisfiability problem for general boolean circuits with fan-in two 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 (the number of gates).*

Proof We start as in Lipton [12] 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 the given boolean circuit with fan-in two. 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 $x_1 \dots x_n \ I \ y$ where $x_1 \dots x_n$ is an input to the circuit, I is a string of bits representing intermediate values of gates in $\mathcal{C}$, and y is $\mathcal{C}(x_1, \dots, x_n)$. 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. $\square$

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 $(x_1, \dots, x_n)$ is the same.

It follows that the frequency of every satisfying assignment in the final test tube is the same. Using a method described by Adleman [2] one can pick a random DNA strand from the test tube and read from it a satisfying assignment (this can also be done using cloning techniques). Hence one can perform random sampling on the set of satisfying assignments. It is well known that the ability to uniformly sample the set of satisfying assignments enables one to approximately count their number using a polynomial number of samples [21]. Note that we do not need to repeat the whole procedure to obtain another sample—rather we can take many samples from the single last test tube, amplifying it first if necessary. While for some problems approximate counting is known to be in random polynomial time (mostly using rapidly mixing Markov chains, see e.g. [11] and references therein), in general it is only known to be in $\Sigma_2 \cap \Pi_2$ (see [19]). Hence this result is stronger than just finding a satisfying assignment, which is in NP.

4.2 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 boolean circuit of fan-in two. We now show how to directly solve this optimization problem.

The results of the previous sections 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 4.2 *MAX-Circuit-Satisfiability for boolean circuits with fan-in two 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 [12].)

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 4.2. 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. We note that recently [4] showed that these techniques combined with a more clever combinatorial algorithm can be used to solve MAX-Clique and 3-Coloring more efficiently, i.e. using less than 2^n strands for a graph with n vertices. They present an algorithm requiring 1.51^n DNA strands for the MAX-Clique problem and 1.89^n strands for the 3-Coloring problem.

4.3 Regular-Circuit-Satisfiability: 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 [12] 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.

The above method can be generalized to arbitrary automata. Let A be a non-deterministic automaton accepting binary inputs where Q is the set of states of A. Let $\delta : Q \times \{0,1\} \to 2^Q$ be the transition map of the automaton (i.e. the automaton moves from state q on input t to all states in the set $\delta(q,s)$). The size of the transition map δ, denoted by $|\delta|$, is defined to be the number of triplets $(q_1, q_2, t) \in Q \times Q \times \{0,1\}$ such that $q_2 \in \delta(q_1, t)$. For instance, a deterministic automaton always satisfies $|\delta| = 2|Q|$. Notice that always $|\delta| < 2|Q|^2$. We obtain the following theorem:

Theorem 4.3 *Let $L \subseteq \{0,1\}^n$ be a set of strings recognized by a non-deterministic automata with a transition map of size $|\delta|$. Then by synthesizing $n|\delta| + n + 1$ oligos one can construct a solution of DNA strands containing all strings in L using only hybridization and ligation reactions, and two extractions.*

Proof Let A be an automata with k states recognizing the language L. The idea is to split each separator nodes S_i of Figure 1 into k nodes $S_i(1), \ldots, S_i(k)$. These nodes will keep track of the state of the automata.

More precisely, let $Q = \{1, 2, \ldots, k\}$ be the states of the automaton A where 1 is the initial state and k is the accepting state (since we know the length of the string, it is sufficient to have one accepting state). Let $\delta : Q \times \{0,1\} \to 2^Q$ be the transition map of the automaton. We assume that 30-mers $S_i(q)$ and $B_i(t)$ have already been agreed upon as discussed in Section 3.2. Note that the separators $S_i(q)$, $1 \le i < n$, are doubled in the strands we use; We synthesize the following oligos:

1. For each $i = 1, \ldots, n-1$ and state $q \in Q$ synthesize the Watson-Crick complement of the doubled separator oligo $S_i(q)S_i(q)$; we also synthesize the Watson-Crick complements of $S_0(1)$ and $S_n(k)$.

2. For each $i = 1, \ldots, n$ and triplet $(q_1, q_2, t) \in Q \times Q \times \{0,1\}$ such that $q_2 \in \delta(q_1, t)$ synthesize the oligo $S_{i-1}(q_1) \cdot B_i(t) \cdot S_i(q_2)$.

Overall, $n|\delta| + n + 1$ oligos were synthesized. When these oligos are mixed together and are allowed to anneal to one another we obtain the set of all computations of the automata. We then apply a ligation enzyme to ligate the oligos into a DNA sequence. Finally we extract all strands containing the oligo $S_0(1)$ (representing the initial state) and the oligo $S_n(k)$ (representing the final state). The resulting strands are exactly those strings accepted by the automaton. Alternatively, this final step can be done by applying PCR with $S_0(1)$ and $S_n(k)$ as primers. Since only the strings accepted by the automata are amplified at an exponential rate, all other strings are diluted to an undetectable level. $\square$

Figure 2 shows the strands that are formed when the standard two state automata for creating even binary strings (i.e. $x_1 \oplus \ldots \oplus x_n = 0$) is used. As was discussed above, the method described in Theorem 4.3 can be used to simplify the Circuit-Satisfiability problem by replacing parts of the circuit by an automaton. We obtain the following result which we refer to as Regular-Circuit-Satisfiability.

Theorem 4.4 *Let $L = L_1 \cap L_2 \subseteq \{0,1\}^n$ be a set of strings where L_1 is recognized by a non-deterministic automaton with transition map of size $|\delta|$ and L_2 can be recognized by a circuit with s gates. Then using $O(|\delta|n + s)$ synthesized oligos and $O(s)$ biological steps one can construct a solution of DNA strands representing all strings in L.*

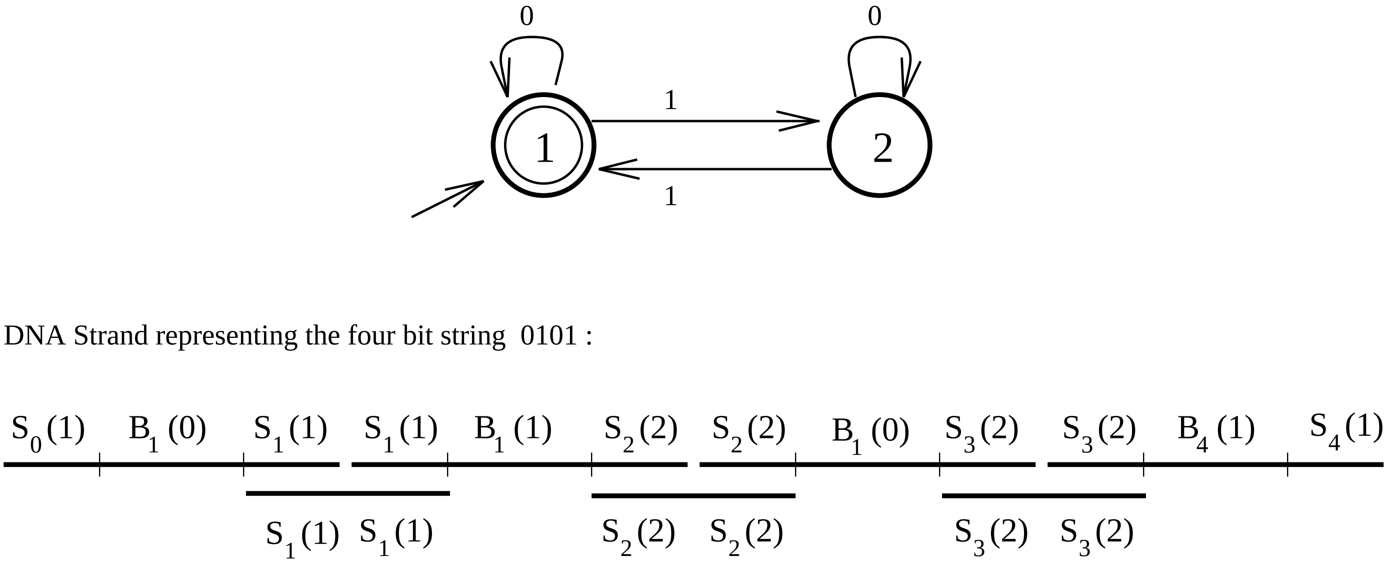

Figure 2: Strands formed by parity automaton

Proof The theorem is an immediate consequence of Theorems 4.3 and 4.1. □

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.

It worth pointing out that the order of the inputs to a circuit does not matter (e.g. the variable x_1 can be input as the second wire and vice versa). However, reordering the inputs to an automata can greatly simplify things. For instance, the language 0^n1^n can recognized by an n-state automata. However if we reorder the inputs to obtain the language $(01)^n$ then an automaton with two states suffices. Therefore, when applying Theorem 4.4 one should choose a clever ordering of the inputs so as to minimize the number of the states in the appropriate automata.

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 4.2 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.

5 Conclusions

The main result of this paper is that DNA based computers can be used to solve the satisfiability problem for boolean circuits. The algorithm presented is considerably more efficient than simulating a NTM using DNA as was suggested by [5, 15, 16, 18]. Furthermore we showed how to improve the performance of the algorithm by using state automata. 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.

There are still many issues to be considered. Foremost is the issue of errors. DNA molecules are known to be fragile, they break easily. Steps towards coping with errors were taken in [8, 9, 10, 3]. 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.

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. In [9] the algorithms are classified in two ways. First, the algorithms are classified 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, an algorithm is said to be *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. 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 [8, 9] for a discussion of this topic. All our algorithms are very good in this respect, since they are all uniform and constant volume.

References

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

[2] L. Adleman. On constructing a molecular computer. In proceedings of the first DIMACS workshop on DNA computing.

[3] M. Amos, A. Gibbons and D Hodgson. Error-resistant Implementation of DNA Computations. Unpublished manuscript.

[4] E. Bach, A. Condon, E. Glaser and C. Tanguay. DNA Models and Algorithms for NP-Complete Problems. Proceedings of the 11th Annual IEEE Conference on Computational Complexity, 1996.

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

[6] D. Boneh, C. Dunworth, and R. J. Lipton. Breaking DES using a molecular computer. Technical Report CS-TR-489-95, Princeton University, 1995. Also in Proceedings of first DIMACS workshop on DNA computing.

[7] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. On the computational power of DNA. Technical Report CS-TR-499-95, Princeton University, 1995.

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

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

[10] R. Karp, C. Kenyon, O. Waarts. Error-resilient DNA computations. In Proceedings of SODA 1996, pp. 458–467.

[11] 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.

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

[13] C. Papadimitriou. Private communications.

[14] J. Reif. Parallel Molecular Computation. In proceedings of 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95, 1995, pp. 213–223.

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

[16] Rothemund. A DNA and restriction enzyme implementation of Turing machines. Available at `http://www.ugcs.caltech.edu/ pwkr/oett.html`.

[17] J. Robson. Algorithms for maximum independent sets. J.of Algorithms, 7, 1986, pp. 425–440.

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

[19] L. Stockmeyer. On Approximating Algorithms for #P. SIAM J. on Computing, 14, 1985, pp. 849–861.

[20] R. Tarjan and A. Trojanowsky. Finding a maximum independent set. SIAM J. on Computing 6, 1977, pp. 537–546.

[21] M. Jerrum, L. Valiant and U. Vazirani. Random generation of combinatorial structures from a uniform distribution. Theoretical Computer Science, Vol. 43, 1986, pp. 169–188.

[22] E. Winfree. On the Computational Power of DNA Annealing and Ligation. Available at `http://dope.caltech.edu/winfree/DNA.html`.