Using DNA to Solve NP-Complete Problems

Richard J. Lipton[†]
Princeton University
Princeton, NJ 08540
rjl@princeton.edu

Abstract: We show how to use DNA experiments to solve the famous "SAT" problem of Computer Science. This is a special case of a more general method that can solve NP-complete problems, first introduced in [3]. The advantage of these results is the huge parallelism inherent in DNA based computing. It has the potential to yield vast speedups over conventional electronic based computers for such search problems.

1. Introduction

In a recent breakthrough Adleman [1] showed how to use biological experiments to solve instances of the famous Hamiltonian Path Problem (HPP). Recall that this problem is: Given a set of "cities" and directed paths between them; Find a directed tour that starts at a given city, ends at a given city, and visits every other city exactly *once*.

This problem (HPP) is known to be NP-complete [2]. A computational problem is in NP provided it can be formulated as a "search" problem. Further, a problem is NP-complete provided, if it has an efficient solution, then so does *all* of NP. One of the major achievements of Computer Science in the last two decades is the understanding that many important computational search problems are not only in NP, but are NP-complete. Another major achievement is the growing evidence that *no* general efficient solution exists for any NP-complete problem.

Thus, Adleman's result that HPP can be solved by a DNA based biological experiment is exciting. However, it does *not* mean that all instances of NP problems can be solved in a *feasible* sense. Adleman solves the HPP in a totally brute force way: he designs a biological system that "tries" all possible tours of the given cities. The speed of any computer, biological or not, is determined by two factors: (i) how many parallel processes it has; (ii) how many steps each can perform per unit time. The exciting point about biology is that the first of these factors can be very large: recall that a

[†] Supported in part by NSF CCR-9304718. Draft of Jan. 19, 1995.

small amount of water contains about 10^{22} molecules. Thus, biological computations could potentially have vastly more parallelism than conventional ones.

The second of these factors is very much in the favor of conventional electronic computers: today a state of the art machine can easily do 100 million instructions per second (MIPS); on the other hand, a biological machine seems to be limited to just a small fraction of a biological experiment per second (BEPS). However, the advantage in parallelism is so huge that this advantage of MIPS over BEPS does not seem to be a problem.

Thus, the advantage of biological computers is their huge parallelism. However, even this advantage does not allow any instance of an NP problem to be feasibly solved. The difficulty is that even with 10^{22} parallel computers one cannot try all tours for a problem with 100 cities. The brute force algorithm is simply too inefficient.

The good news is that biological computers *can* solve any HPP of say 70 or less edges. However, a practical issue is that there does not seem to be a great need to solve such HPP's. It appears possible to routinely solve much larger HPP's on conventional machines

One might be tempted to conclude that this means that biological computations are only a curious footnote to the history of computing. This is incorrect: We have shown that it is possible to use biological computations to vastly speed up many important computations [3]. In particular, we can extend the method of Adleman in an essential way that allows biological computers to potentially radically change the way that we do all computations *not* just HPP's.

We will show how to solve another famous NP-complete problem, the so called SAT problem. In [3] we show how to solve essentially any problem from NP directly. The goal here is to present the full details of the results first sketched in [3].

2. SAT

In this section we will define SAT. It is a simple search problem that was one of the first NP-complete problems. Let us present it by giving an example: Consider the following formula:

$$F = (x \vee y) \wedge (\bar{x} \vee \bar{y}).$$

The variables x, y are *boolean*: they are allowed to range only over two values $0, 1$. Usually, one thinks of 0 as "false" and 1 as "true". Then, $\vee$ is the "logical or" operation and $\wedge$ is the "logical and" operation.

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

Thus, $x \vee y$ is 0 only if both x and y are 0; $x \wedge y$ is 1 only if both x and y are 1. Also, $\bar{x}$ denotes the "negation" of x, i.e. $\bar{x}$ is 0 if x is 1 and is 1 if x is 0.

Let us return to the formula F, i.e.

$$(x \vee y) \wedge (\bar{x} \vee \bar{y}).$$

The SAT problem is to find boolean values for x and y that make the formula F true, i.e. make it equal to 1. In this example, $x = 0$ and $y = 1$ works as does $x = 1$ and $y = 0$, while $x = 0$ and $y = 0$ does not.

The formula F consists of two *clauses*: the first is $x \vee y$ and the second is $\bar{x} \vee \bar{y}$. A clause is a formula that is of the form $v_1 \vee \ldots v_k$ where each v_i is a variable or its negation.

In general a SAT problem consists of a boolean formula of the form $C_1 \wedge \ldots \wedge C_m$ where each C_i is a clause. The question is, then, to find values for the variables so that the whole formula has the value 1. Note, this is the same as finding values for the variables that make each clause have the value 1. The current best method essentially tries all 2^n choices for the n variables.

3. A Simple Model Of DNA

Before we explain how to use DNA to solve SAT, we must first present our model of how DNA behaves. It is a simple idealized model. It ignores many complex known effects; however, it is an excellent first order approximation [5].

For us DNA strands are just sequences $\alpha_1, \ldots, \alpha_k$ over the alphabet $\{A, C, G, T\}$. Double strands of DNA (dsDNA) consists of two DNA sequences $\alpha_1, \ldots, \alpha_k$ and

$\beta_1, \ldots, \beta_k$. Further, the two sequences must satify the famous Watson-Crick complementary condition: for each $i = 1, \ldots, k$, α_k and β_k must be complements, i.e. $A \leftrightarrow T$ or $C \leftrightarrow G$. Note, that complementary sequences anneal in an antiparallel fashion where $5'$ and $3'$ refer to the chemically distinct ends of the DNA strands.

$$\begin{array}{ccccccc} \alpha_1 & - & \alpha_2 & - & \alpha_3 & \ldots \\ \updownarrow & & \updownarrow & & \updownarrow & \\ \beta_1 & - & \beta_2 & - & \beta_3 & \ldots \end{array}$$

There are a number of simple operations that can be performed on test tubes that contain DNA strands. First, it is possible to create large numbers of copies of any short single strand. Here "short" is definitely at least 20, which is all that we will require. Second, it is possible to create dsDNA from complementary single strands by allowing them to anneal.

Third, given a test tube of DNA we can extract out of the test tube those sequences that contain some consecutive pattern. Thus, assume that the pattern is $\delta_1, \ldots, \delta_l$ where each δ_i is in $\{A, C, G, T\}$. Then, a DNA strand $\alpha_1, \ldots, \alpha_k$ will be removed only if for some i,

$$\delta_1 = \alpha_i, \ldots, \delta_k = \alpha_{i+k-1}.$$

We call this operation *extract*.

Fourth, we assume that given a test tube, we can perform a *detect* operation. This operation simply determines whether or not there are any DNA strands at all in the test tube. Finally, the last operation we call *amplify*. This operation replicates all the DNA strands in the test tube. (This operation is used in [3] but is not needed for the case of SAT.)

3. Using DNA to Solve SAT

The next issue we must address is what is the model of biological computation? In this regard we follow [3] closely. In our computations we will always start with one fixed test tube: it is the same for all computations. Clearly, this is an advantage over the method in [1]. The set of DNA in this test tube corresponds to the following simple graph G_n. The test tube is formed in the same way that [1] forms the test tube of all paths to find the Hamiltonian Path. The graph G_n is as follows: It has nodes $a_1, x_1, x'_1, a_2, x_2, x'_2, \ldots, a_{n+1}$. Its edges are as follows: there is an edge from a_k to both x_k and x'_k; also there is an edge from x_k to a_{k+1} and from x'_k to a_{k+1}. Then, the paths

of length $n+1$ that start at a_1 and end at a_{n+1} are assumed to be in the initial test tube.

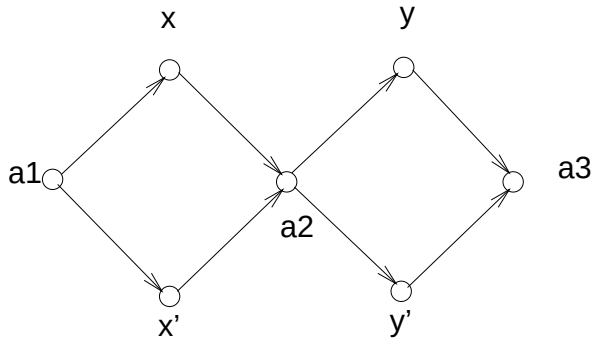

Figure 1: The graph for 2-bit numbers.

The point of this graph is that all paths that start at a_1 and end at a_{n+1} encode a binary n-bit number. For example, the path $a_1 x' a_2 y a_3$ encodes the binary number 01 in the obvious way. At each stage a path has exactly two choices: if it takes the vertex with an unprimed label it will encode a 1; if it takes the vertex with a primed label, it will encode a 0.

Following [1] we encode this graph into a test tube of DNA as follows. First, we assign to each vertex of the graph a random pattern from $\{A, C, G, T\}$ of length l. The length of $l = 20$ used in [1] should also suffice here. We think of this "name" of the vertex as having two parts: we denote the first half by p_i and the second half by q_i. Thus, $p_i q_i$ is the name associated with the i^{th} vertex. Then, fill a test tube with the following kinds of DNA strands:

(*1*) For each vertex put many copies of a $5' \rightarrow 3'$ DNA sequence of the form $p_i q_i$ into the test tube.

(*2*) For each edge from $i \rightarrow j$ place many copies of a $3' \rightarrow 5'$ DNA sequence of the form $\hat{q}_i \hat{p}_j$. Here $\hat{x}$ denotes the sequence that is the Watson-Crick complement to x.

(*3*) Add a $3' \rightarrow 5'$ sequence of length $l/2$, complementary to the first half of the initial vertex, to the test tube. Similarly, add a $3' \rightarrow 5'$ sequence complementary to the last half of the final vertex, to the test tube, i.e. add $\hat{p_1}$ and $\hat{q_n}$.

The key is the following: Every legal path in G_n corresponds to a correctly matched sequence of vertices and edges. In order to see this consider any path in the graph.

It naturally consists of a sequence that alternates "vertex, edge, vertex, edge, ... ". Suppose that $v \to u$ is an edge. Then, a path that passes through v and then u fits together like "bricks":

$$\begin{array}{lccccccccc} & & & v & & & & u & & \\ (5' \to 3') & - & - & - & & & - & - & - & \\ (3' \to 5') & & & - & - & - & & & & \\ & & & v & \to & u & & & & \end{array}$$

The top $5' \to 3'$ part consists of a series of "vertices". The bottom $3' \to 5'$ part consists of a series of "edges". Note, the vertex v is coded as $p_v q_v$ while the edge is $\hat{q_v}\hat{p_u}$. Thus, the end of the vertex and the beginning of the edge can anneal since they are Watson-Crick complements. In the same way the end of the edge and the beginning of the next vertex can also anneal.

Moreover, if l is large enough, then with high probability there will be no inadvertent paths formed. Thus, after the DNA is allowed to anneal it will contain all the paths through the graph, i.e., it will encode all the n-bit sequences possible.

We only perform extracts on the DNA sequences from our graph G_n. For these we use $E(t, i, a)$ to denote all the sequences in test tube t so that the i^{th} bit is equal to a for a in $\{0, 1\}$. We, of course, do this by performing one extract operation that checks for the sequence that corresponds to the *name* of x_i if $a = 1$ and the *name* of x_i' if $a = 0$. Since the length of these names is long enough it is likely that this sequence does not occur by accident somewhere else in the piece of DNA. In some of our constructions we use the remainder, i.e. the strands that do not match the given pattern.

In the following we assume first that extraction operates perfectly, i.e. that *all* sequences are extracted. Later on we will address the issue that extract does not work perfectly.

Let us return once again to the simple example

$$(x \vee y) \wedge (\bar{x} \vee \bar{y}).$$

Before we prove our general result it may be helpful to see how we would handle this example. We construct a series of test tubes. The first one t_0 is just the test tube of all two bit sequences. Then, operate as follows:

(*1*) Let t_1 be the test tube that corresponds to $E(t_0, 1, 1)$. Let the remainder be t_1' and let t_2 be $E(t_1', 2, 1)$. Then, pour t_1 and t_2 together to form t_3.

(*2*) Let t_4 be the test tube that corresponds to $E(t_3, 1, 0)$. Let the remainder be t_4'. Let t_5 be $E(t_4', 2, 0)$. Again pour t_4 and t_5 together to form t_6.

(*3*) Check to see if there is any DNA in the last test tube t_6.

In order to understand how this works consider the following table:

test tube	values present
t_0	00, 01, 10, 11
t_1	10, 11
t_1'	00, 01
t_2	01
t_3	01, 10, 11
t_4	01
t_4'	10, 11
t_5	10
t_6	01, 10

Note, that t_3 consists of all those sequences that satisfy the first clause: 01, 10, 11. In the same way t_6 consists of all those *from* t_3 that satisfy the second clause: 01, 10. The latter are exactly the correct answers to the given SAT problem.

We now turn to the general case. *Any SAT problem on n variables and m clauses can be solved with at most order m extract steps and one detect step.* By "order m" we mean that the number of steps is linear in m. Note, this assumes, as usual, that each clause consists of a fixed number of variables or their negations.

Let $C_1, \ldots, C_m$ be the clauses. We will construct a series of test tubes $t_0, \ldots, t_m$ so that t_k is the set of n-bit numbers x so that $C_1(x) = C_2(x) = \ldots = C_k(x) = 1$. Note, $C_i(x)$ is the value of the clause C_i on the setting of the variables to x. For t_0 use the set t_{all} of all possible n-bit numbers. Let t_k be constructed; we will show how to construct t_{k+1}. Let C_{k+1} be the clause

$$v_1 \vee \ldots \vee v_l$$

where each v_i is a literal or a complement of a literal. For each literal v_i operate as follows: If v_i is equal to x_j, then form $E(t_k, j, 1)$; if it is equal to $\bar{x}_j$, then form $E(t_k, j, 0)$. Note, as in the example, we perform each extraction and then use the reminder for the next one. Put all these together by pouring to form t_{k+1}. Then, do one detect operation on t_m to decide whether or not the clauses are satisfiable or not.

In the above we have assumed that the operations are *perfect*, i.e. that the operations are performed with no error. This definitely needs to be studied. One immediate comment is that the assumption that extract gets *all* of the sequences is *not* needed. If the original test tube has many copies of the desired sequence, then we only need that there is some reasonable probability that it is correctly extracted to make everything work properly.

4. Using DNA to Solve Generalized SAT

In this section we show how to use the methods developed here to solve a generalization of SAT. This generalization includes most "natural" examples of NP problems.

The key is to generalize the class of boolean formulas that we consider. Recall, a SAT problem corresponds to a formula of the restricted form:

$$C_1 \wedge \ldots \wedge C_m$$

where each C_i is a clause. A natural generalization of this is to consider problems that correspond to *any* boolean formula. Thus, we now allow formulas to be unrestricted: they can use the operations of negation, logical "or", logical "and" without any restrictions. More precisely, formulas are defined by the following recursive definition:

(*1*) Any variable x is a formula;

(*2*) If F is a formula, then so is $\overline{F}$;

(*3*) If F_1 and F_2 are formulas, then so are $F_1 \vee F_2$ and $F_1 \wedge F_2$.

The *size* of a formula is measured by the number of operations used to build the formula. The *SAT Problem for Formulas* is: Given a formula F; Find an assignment of boolean values to the variables so that the formula is true, i.e. is equal to 1. Since this problem includes normal SAT, it clearly is still NP-complete.

Our main result is: *We can solve the SAT problem for formulas in a number of DNA experiments that are linear in the size of the formula.*

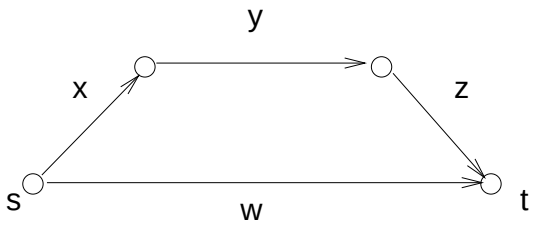

Figure 2: A Simple Contact Network.

We now turn to proving this last statement. The key is to actually prove more. We show how to solve not just any formula but any *Contact Network* [4]. A contact network is a directed graph with a single special source s and a single special sink t. Each edge is labeled with either x or $\bar{x}$ where x is some variable.

Given any assignment of values to the variables we say that an edge is *connected* provided the edge's formula evaluates to 1. Thus, if the edge is labeled with $\bar{x}$ then it

is connected only if $x = 0$. Thus, the network in Figure 2 is equal to 1 only if $w = 1$ or $x = y = z = 1$.

The *SAT Problem for Contact Networks* is to determine whether or not there is an assignment of values to the variables so that there is a directed connected path from s to t. Note, if two edges have the same label, then one is connected if and only if the other is. Put another way, all values of x or $\bar{x}$ are consistent.

Our result follows from the next two simple claims:

(*1*) Given any formula of size S there is a contact network of size linear in S so that the same set of assignments satisfy the formula as the network.

(*2*) Given any contact network of size S, in order S DNA experiments we can solve the SAT problem for the network.

Clearly, these two claims will prove our assertion about formulas.

The first claim is classic [4]. However, for completeness we will supply the argument. In the following say that two formulas are *equivalent* if they always give the same value for any assignment to the variables. Any formula can be placed into a normal form based on the famous identities that are called DeMorgan's Laws:

$$\overline{x \vee y} = \bar{x} \wedge \bar{y}$$

and

$$\overline{x \wedge y} = \bar{x} \vee \bar{y}.$$

Using these it is easy to see any formula is equivalent to one where all the negations are on variables. We assume that our formulas are so restructured. Then, we will build a contact network that simulates the formula inductively. If the formula is a variable or its negation, then there clearly there is a single edge contact network that is equivalent. For example, the formula $\bar{x}$ is just equivalent to the network with an edge from s to t with the label $\bar{x}$.

The general case the formula is equal to either $E \vee F$ or $E \wedge F$. Assume that G is the network for E and that H is the one for F. Then, the network for $E \vee F$ is constructed by placing G and H in parallel: see Figure 3. Clearly, there is a connected path from s to t provided there is *either* a path from s to t through G or through H. The network for $E \wedge F$ is constructed by placing them in series: see Figure 4. In this case, there is a connected path from s to t provided there is one *both* through G and H.

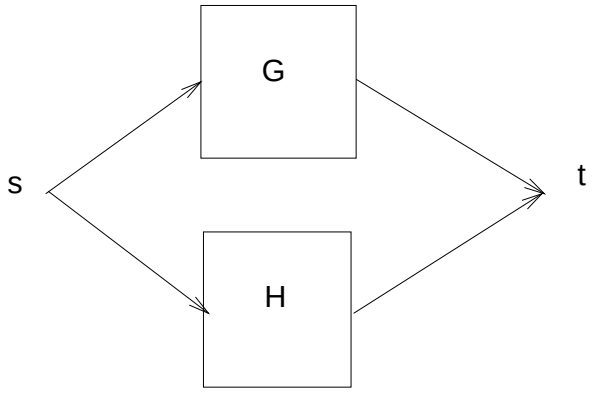

Figure 3: The "or" case.

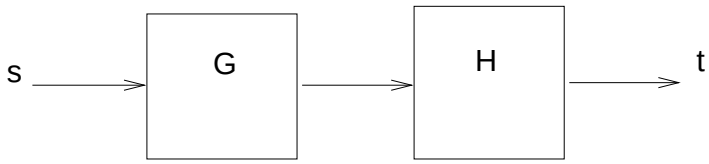

Figure 4: The "and" case.

Second, we will argue how to use DNA experiments to solve the SAT problem for any contact network. This is quite simple. We plan to associate a test tube P_v with each node v in the contact network. The test tube will encode in the usual way the set of assignments to the variables that connect s to v. The test tube P_t associated with t is the "answer". The key is the following: Suppose that $v \rightarrow u$ is an edge with the label x (respectively $\bar{x}\}$ and that P_v is already constructed. Then, construct P_u simply by doing the extraction $E(P_v, x, 1)$ (respectively $E(P_v, x, 0)$). Note, if several edges leave a vertex v, then use an amplify step to get multiple copies of the test tube P_v. Also if several edges enter a vertex v, then pour the resulting test tubes together to form P_v.

5. Conclusions

In summary, we have shown how to do any NP search problem defined by a formula. In an earlier version of [3] it was believed that this could be proved for any *circuit*,

but this seems to be open. Also open is how to exactly handle errors in extractions. Finally, the main open question is, of course: Can one actually build DNA computers based on the methods described here?

Acknowledgements: I would like to thank David Dobkin for a number of: helpful conversations about this work. I would also like to thank Len Adleman for taking the time to explain his wonderful construction. I would also like to thank Dan Boneh and Chris Dunworth for their help.

References

[1] L. Adleman, Molecular Computation of Solutions to Combinatorial Problems. Science, vol. 266, Nov. 11, 1994.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co. 1979.

[3] R. J. Lipton, Speeding Up Computations via Molecular Biology, unpublished manuscript Dec. 9, 1994.

[4] C. Shannon, The Synthesis of Two-Terminal Switching Circuits. Bell System Technical Journal, 28, 59-98.

[5] R. R. Sinden, *DNA Structure and Function*, Academic Press, 1994.