Speeding Up Computations via Molecular Biology

Richard J. Lipton[†]

Princeton University

Princeton, NJ 08540

rjl@princeton.edu

Abstract: We show how to extend the recent result of Adleman [1] to use biological experiments to directly solve any NP problem. We, then, show how to use this method to speedup a large class of important 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). Since this problem is known to be NP-complete it follows that biology can be used to solve any problem from NP. Recall that all problems in NP can be reduced to any NP-complete one.

However, this 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 small amount of water contains about 10^{23} molecules. Thus, biological computations could potentially have vastly more parallelism that 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 an 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. We will return to this point later on.

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.

[†] Supported in part by NSF CCR-9304718. Draft of Dec. 9, 1994.

The difficulty is that even with 10^{23} 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 isue 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.

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 will show that it is possible to use biological computations to vastly speed up important computations such as factoring. 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.

Our main first point is that we can extend Adleman [1] to show that we can build a biological computer that can solve any NP problem *directly*. This is important since our biological machines will be limited in the amount of parallelism that they can perform. Thus, solving a SAT problem on 70 variables directly is fundamentally better than using the reduction from SAT to HPP. If one used the standard reduction, then the best SAT problem one could solve in this way would be tiny. Our biological machines will also have some other technical advantages over the original method of [1].

Consider the following computational problem: Given a boolean formula

$$F(x_1, \ldots, x_n)$$

of size s; Determine if there is an $x_1, \ldots, x_n$ so that $F(x_1, \ldots, x_n)$ is satisfied. Call this problem the (n, s)-*Formula Satisfaction Problem* (FSP). Our main result is the following:

Theorem 2: *Any (n, n^c) FSP can be solved with at most $O(n^c)$ biological steps.*

Note, the original result from [1] only works for the special case of HPP. There the formulas F are of a very simple form. The main advantage of our result is that the formula F can be arbitrary.

The second point is that we will show how to use our biological machines as *subroutines* to solve important problems such as factoring. The general idea is the following: suppose that we can solve any FSP of the form $(n, n^{O(1)})$ in constant time. Can we use this oracle to speedup computations?

2. Biological Computations

In this section we will give the exact model of biological computing. We will then show how to prove our main theorem about solving instances of FSP.

The key issue is what is the model of biological computation? In this regard we follow [1] closely. We assume in particular the following simple model. The fundamental concept of a biological computation is that of a set of DNA strands. Since this are usually kept in a test tube will say that a *test tube* is just a collection of pieces of DNA. Thus, if t is a test tube, from the point of view of a computer scientist, it is just a finite multi-set of strings from $\{A, C, G, T\}$. (A multi-set is a set that allows repeated copies of a string.)

The first question is what sets can we start from? The second question is what operations can we perform on test tubes? 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 Hamitonian Path (see also Appendix). 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.

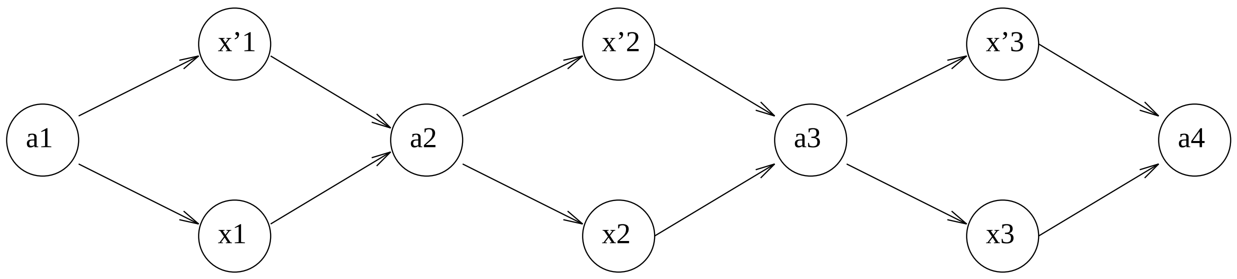

Figure 1: The graph for 3-bit numbers.

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 $a_1x'_1a_2x_2a_3x_3a_4$ encodes the binary number 011 in the obvious way.

We need to be able to perform a small collection of operations on test tubes:

(*1*) *Extract.* In this operation we can extract out of a test tube t all those sequences that contain some consecutive subsequence.

(*2*) *Detect.* In this operation we can determine if there is *any* DNA sequence in the test tube at all.

(*3*) *Amplify.* In this operation we can replicate all the sequences from the test tube.

We only operate on the DNA sequences from our graph G_n. For these we use $E(t,i,a)$ to denote all the sequences in t so that the i^{th} bit is equa l to a for $a \in \{0,1\}$. We, of course, do this by perfoming one extract operation that checks for the sequence that corresponds to x_i if $a=1$ and x_i' if $a=0$. In the following we assume first that extraction operates prefectly, i.e. that *all* sequences are extracted. Later on we will address the issue that extract does not work prefectly.

Before we prove our main Theorem lets prove the following simple theorem:

Theorem 1: *Any SAT problem on n variables and n^c clauses can be solved with at most $O(n^c)$ extract steps and one detect step.*

Proof: 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 $C_1(x)=C_2(x)=\ldots=C_k(x)=1$. 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 $E(t_k,j,0)$. 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. ∎
Our next result is the following:

Theorem 2: *Any (n,n^c) FSP can be solved with at most $O(s)$ extract steps and one detect step.*
Here an FSP is just a CSP where the circuit is restricted to be a formula.

Proof: Critical to the proof is the function $TT(f,t)$ defined to be

$$\{x \in t | f(x)=1\}.$$

Note, f is a boolean function on n bits and t is a set of n bit numbers. Let F be the given formula and let t_{all} be the set of all n bit numbers. Then, we want to construct $TT(F,t_{all})$. Then, one detect operation is enough to find out if there are any x that make F equal to 1. As usual we assume that all internal operations are either $\wedge$ or $\vee$.

Let $C(F)$ denote the cost in steps needed to construct the set $TT(F,t)$ and at the same time also construct $t - TT(F,t)$. We plan to prove by induction on the formula F that $C(F)$ is linear in the size of the formula F.

We now describe how to compute TT. There are four cases:

(*1*) $TT(x_i,t)$. In this case it is equal to $E(t,i,1)$.

(*2*) $TT(\bar{x}_i,t)$. In this case it is equal to $E(t,i,0)$.

(3) $TT(F_1 \vee F_2, t)$. In this case form $t_1 = TT(F_1, t)$ and $t_2 = t - TT(F_1, t)$. Then, inductively form $t_3 = TT(F_2, t_2)$ and $t_4 = t_2 - t_3$. The result is then $t_1 \cup t_3$ and the remainder is t_4.

(4) $TT(F_1 \wedge F_2, t)$. In this case first inductively form $t_1 = TT(F_1, t)$ and $t_2 = t - t_1$. Then, inductively form $t_3 = TT(F_2, t_1)$ and $t_4 = t_1 - t_3$. The result is t_3 and the remainder is $t_2 \cup t_4$.

Note, we do the unions by just pouring the two test tubes together. Clearly, it is easy to prove that all the sets are computed correctly. Moreover, the number of steps to get the formula F is linear in its size. ∎

If one allows amplify steps, then there is a more direct proof of the last theorem. In the induction we only need to operate as follows: Steps (1) and (2) stay the same. For step (3) we just do the "or" by $TT(F_1, t) \cup TT(F_2, t)$. This requires that we copy t, i.e. requires an amplify step. Step (4) is now just $TT(F_1, TT(F_2, t))$.

It may be interesting to note that our method is fundamentally different from usual methods. Logical or's are done by using union's: of course this is not new. However, we do logical and's by a new method. Essentially, a logical and operation is performed by "functional composition". The logical and of F_1 and F_2 is performed by first applying the operator F_1 and then applying F_2.

In all the above theorem's we have assumed that the operations are *prefect*, 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 probablity that it is correctly extracted to make the theorems work.

3. Using CSP to Speedup Computations

As stated earlier the key question is suppose that one can solve any $(n, n^{O(1)})$ CSP in constant time. How can one use this ability to speedup computations?
Consider any search problem that operates as follows:

```
for each x=1,...,M
    if p(x) then
        return x
```

where $p(x)$ is some predicate and $M = 2^m$. Assume that p takes polynomial time and logarithmic depth. Then, the time to do this search is clearly $O(n^c 2^m)$ for some constant c. We claim that the ability to do any $(n, n^{O(1)})$ FSP in constant time allows us to do the above search problem in time

$$O(n^c 2^{m-n}).$$

The proof of this is trivial: just generate $m - n$ of the bits of each x. Then, set up a FSP that is true precisely in the case that some x has the given $m - n$ bits and satisfies the test p. This easily can be set up as a $(n, n^{O(1)})$ FSP. Essentially, we use our oracle to do 2^n tests of p at one time.

Therefore, we can speedup by 2^n any search problem. An important point is that the speedup is actually a bit more complex. Let's compute the exact speedup in the above method. The conventional machine does 2^m tests of p. The biological method uses 2^{m-n} FSP's. Each of these can be done in $O(s)$ biological steps where s is the circuit complexity of p. Thus, the speedup is

$$\frac{2^m s/\alpha}{2^{m-n} s/\beta}$$

where α is the MIPS of the conventional computer and β is the BEPS of the biological computer and α. Clearly, the speedup is $2^n \beta/\alpha$: for reasonable values of n and β and α this is over a trillion!

The critical open question that we cannot yet completely answer is the following: Can we speedup any computation not just the important class of search problems? In any event, it appears that we should attempt to classify those problems which have speedups using FSP's. We expect that there will be progress on this in the near future.

We can, however, say something about some problems that are not exactly search problems. Consider the problem of factoring integers. The inner loop of many fast methods is the search to find $y = x^2 \bmod N$ so that y factors completely over a given "factor basis" (see [2] for details). It is clear that we can speedup the naiive search for such any y by a factor of 2^n. The question however is can we use our methods to speedup the current more sophisticated methods?

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 for his help.

References

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

[2] H. Cohen, A Course in Computational Algebraic Number Theory, Springer-Verlag, 1993.

Appendix

In this appendix we show how to use Adleman's method to form the initial test tube for our graph G_n. The key to his method is to assign to each vertex a 3' DNA sequence and to also assign another 5' DNA sequence to each edge. Each vertex in G_n gets a sequence of 5' DNA that is of the form $v_i u_i$ where each part is l in length and is randomly selected. The value of l is large enough to avoid accidental matches. Each edge from $i \rightarrow j$ gets a 3' DNA that equal to $\hat{u}_i \hat{v}_j$. Here $\hat{x}$ denotes the sequence that is the Watson-Crick complement to x. The initial vertex also adds a 5' sequence that corresponds to its first half; the final vertex also adds its last half. The key is the following: First, every legal path in G_n corresponds to a correctly mathched sequence of vertices and edges. Second, if l is order $\log(n)$, then there will be no other paths. Actually, the latter depends on the random choices, but the chance that it is false can be made as small as one wishes.