Making DNA Computers Error Resistant

Dan Boneh

Richard J. Lipton

Department of Computer Science

Princeton University

DNA (deoxyribonucleic acid) based computing is days old. Yet 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 [10] 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 [4,5,15,3,12,13,11]. Each is 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.

All results assume that the biological operations are *perfect*. Clearly, this is *not* the case and errors must be addressed. However, it is important to know what the ultimate limits are without errors. Clearly, if in the error-free case DNA cannot do exciting things, then there is no hope for the realistic case.

However, DNA operations are *not* error free. Our main result is that it is possible to allow a large class of DNA computations to survive fairly high error rates. This class of DNA computations includes Adleman's original algorithm as well as Lipton's formula SAT algorithm. It, however, does not include the circuit method of [5] and [4]. We present a new algorithm for circuit satisfiability which can be made error resistant using the techniques of this paper.

All DNA algorithms use strands of DNA to encode many parallel computations. They naturally fall into three classes: Algorithms are *Decreasing Volume* if the number of strands decreases as the algorithm executes; they are *Constant Volume* if all the strands remain throughout the computation. A *Mixed Algorithm* is one that fits into neither class. Adleman's and Lipton's are decreasing volume. Note, we use "volume" to denote number of strands.

Adleman's and Lipton's algorithms are even more special. In these algorithms each strand is either *good* or *bad*. A good strand encodes a solution to the problem; a bad strand does not. The goal of the algorithms is simple: to keep the good strands around

and to eliminate the bad strands. If they succeed, then the final test tube will only contain good strands. The answer can then be found by sequencing one of the remaining strands.

The difficulty with this simple description is that errors can take place. An error may have terrible consequences. If a good strand is damaged or is lost, then the algorithm will fail. If on the other hand, bad strands are not removed, there may be many bad strands left at the end of the algorithm. This also could cause the algorithm to fail.

Errors come from two sources. First, every operation can cause an error. The principal operation in these algorithms is *extraction* [1,10]. This operation removes from a test tube those strands that contain some given pattern. However, extractions are not perfect. They usually only remove about 95% of the strands that match the pattern. In additional, strands that do not match will sometimes be removed anyway: rates are typically 1 part in 10^6. Another source of errors are due to [9,14,15] the fact that DNA has a "half-life": it decays at a finite rate. If the DNA algorithm takes months, for example, there is a probability that the good strands will simply dissolve.

The reason that errors in operations are so critical is that we envision algorithms that do thousands of steps. If each extraction removes 99% of the good strands, then the probability that a good strand is there at the end of a 1,000 step computation is tiny, i.e. is 0.00004. Thus, even at this high extraction rate the algorithm essentially will always fail. This does not even take into account the extraction of bad strands or the half-life problem.

Our main result is the following: *We can show how to map Adleman's and Lipton's algorithms into a new algorithms that are constant volume. These new algorithms are highly resistant to errors.* Moreover, these new algorithms run in almost the same number of steps. Thus, the time penalty for making the algorithms error resistant is small.

However, there is a "space" penalty for making the algorithms constant volume. The number of test tubes remains the same throughout the computation. Thus, the same amount of equipment is needed at the beginning of the computation as later on. This may not be much of a penalty. If one only wishes to perform one computation, then the original algorithm would have more and more equipment sit idle as the volume decreases.

Main Results

We now turn to showing how to make DNA algorithms more error resistant. In order to make the analysis clear we will assume that the algorithm operates as follows: (The more general case in [10] is handled in the same way.) Initially, there are 2^n strands of

DNA. Just one of these strands is good; all the rest are bad. Clearly, the case of one good strand is worst case.

The algorithm consists of s extraction steps. The good strand always matches the pattern of the extraction. However, a bad strand may or may not match the pattern. A *type I* error occurs when a strand is not correctly extracted: let p be the probability that this happens. A *type II* error occurs when a strand should not be extracted but is anyway: let q be the probability that this happens. Typical values of p are .95 and q are 10^{-6}.

Note, just because a strand is bad does *not* mean that at each step it will not match the pattern. The good strand always matches the pattern; the bad strands may or may not. Indeed if x is a bad strand let $M(x)$ be the number of extractions for which it does *not* match the pattern. Clearly, $M(x)$ is at least 1.

We next assume that the computation decreases its volume at a uniform rate. Thus, we will assume from now on that every step decreases the volume at the same rate. Thus, since there are s steps and 2^n strands, the volume goes down by a factor of 2 every s/n steps. If $s = 900$ and $n = 60$, then every 15 steps the volume of DNA halves.

In the case that each step decreases the volume at the same rate, our analysis is simple. However, our method can easily be adjusted for a rate of decrease that is only approximately constant. Our method does not work in the pathological case where the volume goes down in some very complex way. We believe that it may be possible to randomize the order of extractions to "average" out the rate. Finally, such pathological cases may not arise in real problems.

Our modification to the basic algorithms is twofold: First, *every s/n steps double the amount of DNA.* We do this by a PCR operation [6]. There are several points: In practice one might wait until the volume decreased by a larger factor. We will do the analysis in this simple case; the generalization to bigger steps is quite easy. Second, the additional cost in time is small. Even if the PCR step is slow, it only happens every s/n steps. In our example, this is every 15 steps. Thus, the additional required time is a small percent of the total time.

Our second modification is to the final detect step. Before we assumed that ideally only one strand remained in the final test tube. Thus, to determine the answer we only had to sequence that strand. Now there may be both good and bad strands in the final test tube. We, therefore, replace this simple detect step by the following: Remove a strand from the test tube and sequence it. Then, check whether or not it is a solution. If it is, the algorithm is has found a solution. If it is not, then try another strand. After some fixed number of these steps, say m, stop and report failure. Note, it is easy to computationally check whether or not a sequence does indeed encode a solution to the problem being solved.

The modified algorithm can now fail in two different ways. First, there might be no good strands in the final test tube. Second, there might be good strands in the

final test tube, but there might be so many bad strands that the probability that the final step gets a solution is small. This leads naturally to the following: Let P_s be the *survival probability* i.e. the probability that some good strand is in the final test tube; also let P_r be the *selection probability* i.e. the probability that at least δ of the strands in the final test tube are good.

The probabilities P_s and P_r together control the probability that our algorithm succeeds. More precisely, we can use them to easily compute an upper bound on the probability that our algorithm fails. The probability of failure is at most

$$1 - P_s + P_s(1 - P_r) + P_s P_r(1 - \delta)^m$$

where recall that we do m repeated detects and δ is the amount of good strands in the final test tube. Thus, the probability that our method works is at least

$$P_s P_r(1 - \delta)^m.$$

The key is to bound these two probabilities, P_s and P_r. Let us first consider P_s. It should be clear that without the PCR step the probability that the good strand survives is p^s: recall that s extractions are performed and each succeeds with probability p. Now, every s/n steps the surviving strands are doubled in number. This applies equally to the good and the bad strands. Therefore, if a good strand survives s/n steps, it creates 2 new good strands. (Note, we are assuming that PCR step is error-free. However, any error there can be handled by decreasing p.)

This is an example of a *branching process* [7]. Branching processes are famous for modelling nuclear reactions and spread of diseases. In our case we are modelling whether or not the single initial good strand or its descendants survive many extractions. An example of such a branching process can be seen in Figure 1. Let r be the probability that a single good strand survives to the next PCR step, i.e. $r = p^{s/n}$. If a strand survives, then it becomes 2 good ones. Thus, the chances have greatly increased that some good strand will survive all the future extractions. Following Feller [7] the key quantity is the extinction probability ζ; for this branching process it is the smallest positive solution to the equation

$$1 - r + r\zeta^2 = \zeta.$$

A simple computation shows that the value of ζ is $\frac{1}{r} - 1$. The probability, P_s, that some good strand survives is $1 - \zeta$. Therefore,

$$P_s = 1 - \zeta = 2 - \frac{1}{r} = 2 - p^{-s/n}$$

Table 1 contains some example results. The probability that the original algorithms without PCR improvement succeed is essentially 0 in all cases. The table shows the probability that *some* good strands survive the entire series of steps.

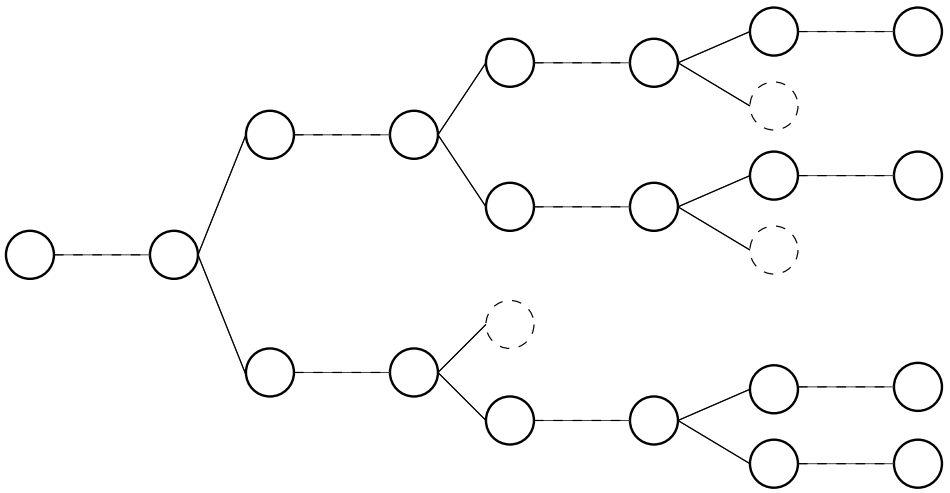

Figure 1: An example branching process. A good strand is replicated every few steps. Even though some of its descendants die (dashed discs), its prevalence increases exponentially.

It remains to show that P_r is large, i.e. that it is likely that the ratio of good strands to bad strands is not too small. For this argument we can *ignore* the PCR modification. The reason is simple: we are interested only in the *ratio* of the number of good to bad strands. Since PCR doubles both the good and bad strands equally their ratio is unchanged. Thus, in this case we can assume that there is exactly 1 good strand.

Recall that a bad strand x does not match the pattern of an extract for $M(x)$ extractions. It is therefore the case that a bad x will survive only if $M(x)$ type II errors occur. Thus, the expected number of bad strands that survive is at most

$$\sum_x q^{M(x)}$$

where the sum is over all bad strands since q is the probability that a type II error occurs. (The actual sum is smaller since bad strands still might be subject to type I errors.)

In order to simply this sum define M_k as the number of bad strands x that have $M(x) = k$. Then, the above is equal to

$$M_1 q + M_2 q^2 + M_3 q^3 + \ldots$$

where the missing terms are insignificant. This follows since $q = 10^{-6}$ and there are at most 2^{70} strands.

The critical point is that $M_1 q + M_2 q^2 + M_3 q^3$ determines how long the final detect step takes. Let this quantity be l. Then, the detect step will likely require order l steps. Clearly, we want this to be as small as possible. However, with out any assumption it *is* possible that l is large. Perhaps $M(x) = 1$ for most bad strands.

There are three points. First, in practical problems it seems that it would be rare for many bad strands to satisfy all but a few of the constraints. Typically one would expect that a bad strand might have $M(x)$ fairly large. Clearly, in this case l would be small. Note, even if $M_1 = 10^8$ and $M_2 = 10^{14}$ and $M_3 = 10^{20}$, the value of l is only 300.

Second, we can restructure the computation so that l is made smaller. For example, if l is large we can just repeat the whole computation and reduce l. If one repeats extractions twice, then all values of $M(x)$ are doubled. This is quite expensive and it would be better to avoid this. However, it is interesting that there is a general method to reduce l. Note, we do not have to do this repeating until we fail at the detect step.

Finally, it may be possible to do a more interesting transform on problems that will reduce the number of "almost" solutions, i.e. will reduce l. This seems to be related to the famous Probabilistic Checkable Proofs of [2].

p	probability good survives
.97	.42
.98	.65
.99	.84

Table 1: $s = 900$ **and** $n = 60$

Error Resistant Circuit Satisfaction

The techniques presented above transformed a decreasing volume algorithm into a constant volume algorithm. By doing so we made sure that the prevalence of good strands increases in the test tube. However, this technique can not be applied to an algorithm which is constant volume to begin with. The reason is that the amount of material used by the algorithm would double after every PCR step. Within a few steps the amount of material would be to large to handle.

The algorithm for solving the circuit satisfiability problem described in [5] is a constant volume algorithm. For this reason, it can not be made error resistant using the technique described above. To solve this problem we describe a new decreasing

volume algorithm for solving circuit satisfaction. The results of the previous section can be directly applied to this algorithm.

Given a circuit C with n inputs we present an algorithm for finding an assignment to the variables which makes the circuit C output a 1. We assume that the circuit is made of gates which are either "and" or "or". The circuit should contain no negation gates. This can usually be achieved by pushing negations down to the variables using De-Morgan's laws.

The key new molecular primitive used is "intersection". Let T_1 and T_2 be two test tubes each representing some set of binary strings. The intersection of these two test tubes is the set of strands that appear in both T_1 and T_2. There are many ways to implement the intersection primitive. Here we propose one possible method: Append a fixed string α to one end of all strands in T_1. Similarly append a string β to all strands in T_2. The append procedure is describes in [4]. Pour T_1 and T_2 into one container and melt the hydrogen bonds by heating the solution. Then cool the solution and allow the strands to reanneal. Strands that appear in both T_1 and T_2 will form double stranded DNA with a sticky end of α on one side and a sticky end of β on the other side. A polymerase enzyme can now be used to complete these strands to full double stranded DNA molecules. Now extract from the resulting solution all strands that contain both the strings α and β. The only extracted strands will be the ones that appear in both T_1 and T_2. This procedure may have to be repeated several times to make sure that all the strands in the intersection have been extracted.

The algorithm for circuit satisfiability is as follows: As in [10], create a solution of DNA strands representing all 2^n bit strings of length n. Using this solution create two solution for each variable x_i: one containing all strings where $x_i = 0$ and one containing all strings where $x_i = 1$. The gates in the circuit can now be evaluated one by one in the following way: an "or" gate is is evaluated by pouring the two input solutions into one container. An "and" gate is evaluated by taking the intersection of its two input solutions. When all the gates have been evaluated, the resulting solution contains all strands that satisfy the circuit. Clearly the running time of this algorithm is proportional to the number of gates in the circuit.

Suppose the circuit C has only one satisfying assignment. Then this is a decreasing volume algorithm in the sense that the higher the gate is in the circuit, the less strands there are at the output of the gate. At the output of the top most gate there is only one strand, namely the satisfying assignment.

As far as the error analysis is concerned, "or" gates may be ignored since they are unlikely to produce any errors. To analyze the errors introduced by "and" gates we can use the methods of the previous section. Let L be some path in the circuit from an input to the output. As before, we must make the assumption that along this path the volume of strands decreases uniformly as the "and" gates are evaluated. Let l be the length of the path L then every l/n "and" gates the volume of strands will halve.

By doubling the number of strands every l/n steps using PCR this algorithm becomes a constant volume algorithm. The same argument used in the previous section directly applies to analyze the survival and selection probabilities.

Conclusions

We showed that the formula satisfaction algorithm of [10] can be made error resistant by doubling the amount of material every several steps. This doubling step turns this algorithm into a constant volume algorithm. The analysis showed that the error parameters of the extract operation are sufficiently small to make the formula satisfaction algorithm work in the presence of errors. Finally, we presented a new circuit satisfaction algorithm which can be made error resistant using the same method.

References

[1] L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems", Science 266:1021-1024 (Nov. 11) 1994.

[2] S. Arora, "Probabilistic Checkable Proofs", Ph.D. thesis, U.C. Berekely, 1994.

[3] D. Beaver, "A Universal Molecular Computer", Penn State University Tech Report CSE-95-001.

[4] D. Boneh, C. Dunworth, R. Lipton, "Breaking DES Using a Molecular Computer", Princeton CS Tech-Report number 489.

[5] D. Boneh, C. Dunworth, R. Lipton, J. Sgall, "On the Computational Power of DNA", unpublished manuscript.

[6] H. Erlich, "PCR Technology: Principled and Applications for DNA Amlification", Stockton Press.

[7] W. Feller, An Introduction to Probability Theory and Its Application, Volume I., John Wiley and Sons 1957.

[8] M. R. Jerrum, L. G. Valiant, V. V. Vazirani, "Random generation of combinatorial structures from a uniform distribution", Theor. Comp. Sci 43:169-188, 1986.

[9] Lindahl, Tomas, Instability and decay of the primary structure of DNA, Nature 362 (Apr 1993) 709-715

[10] R. Lipton, "Using DNA to solve NP-Complete Problems", Science, to appear.

[11] C. Papadimitrou, private communications.

[12] D. Rooβ and Klaus W. Wagner, "On the power of Bio-Computers", unpublished manuscript.

[13] Rothemund, private communication.

[14] Saul, R.L., Ames, B.N., Background levels of DNA damage in the population, Basic Life Sciences 38 (1986) 529-535.

[15] W. Smith, "DNA Computers in Vitro and Vivo", unpublished manuscript.

[16] Supported in part by NSF CCR-9304718.