STUDENT ARTICLE

9

APPLIED MATHEMATICS CORNER

DNA Computation and Algorithm Design

Shrenik Shah[†]
Harvard University ’09
Cambridge, MA 02138
sshah@fas.harvard.edu

9.1 Introduction

DNA computing was developed by a team run by Leonard Adleman in a 1994 experiment [Ad]. Since then, scientists have produced a number of developments in this area, both theoretical and practical. Adelman’s interest in DNA computing arose out of an effort to harness the power of the massive parallelism present in biological systems; theoretical computer science has developed parallel algorithms that efficiently speed up deterministic computation.

The key benefit of DNA computation is **parallelism**—a large number of processes may be run simultaneously. According to [KGY], just a liter of weak DNA solution can hold 10^{19} bits of information, which can encode the states of 10^{18} processors. All of these processors can be acted on simultaneously by the primitive operations of DNA computation. Of course, the structure of this information requires innovative approaches to details that are taken for granted when working with parallel computer grids. When encoding a problem as DNA, it becomes difficult to control the computation and extract the output.

DNA computation has been demonstrated in increasingly more powerful trials. In 2000, Yoshida and Suyama demonstrated a protocol for solving instances of 3-SATISFIABILITY, a very important computational problem [YS]. The instance they solved was rather simple, however—a human could work it out without a computer. A team led by Adleman, who originally developed DNA computing, managed to solve a 20-variable instance of this problem in 2002 [BCJ$^+$]. This difficult instance is beyond human capacity, though this is still just a couple seconds’ work for a modern computer.

The status of DNA computation today is still tentative. DNA computation has not yet exceeded the power of modern computers. There are several issues, including the expense of the equipment necessary and the error rate inherent to biological processes, that may prove to cripple the feasability and utility of DNA computation in the long run.

9.2 The DNA Computation Model

To put DNA computation on a concrete framework, the article [Ka] by Lila Kari breaks the process into smaller steps that can be regarded as primitive operations for the DNA computer.

1. **Synthesizing**: This stage involves creating a single DNA strand consisting of polynomially many base pairs.

2. **Mixing**: This step involves taking the contents of two test tubes and mixing them together in a third. This step may seem frivolous, but becomes important in the theoretical framework.

[†]Shrenik Shah, Harvard ’09, is a senior mathematics concentrator with a secondary field in English and is also pursuing a concurrent masters’ degree in Computer Science. He was a founding member and Articles Editor of The HCMR. His interests lie in algebraic number theory and complexity theory.

3. **Annealing**: This process, also known as **hybridization**, involves lowering the temperature of the solution so that two DNA sequences to attach together in a double helix.

4. **Amplifying**: A reaction known as the Polymerase Chain Reaction (PCR) allows one to produce a copy of a DNA strand. In [Ka], Kari observes that exponentially many strands can be produced by repeatedly performing this operation in parallel.

5. **Separating**: In this step, gel electrophoresis is used to determine the length of a DNA strand.

6. **Extracting**: A step called **affinity purification** allows one to find strands that match a given substring of DNA.

7. **Cutting**: Certain enzymes allow one to cut DNA at sites with particular patterns of bases.

8. **Ligating**: This is the opposite of cutting; certain enzymes provide for the ability to connect DNA strands with certain endings.

9. **Substituting**: This fairly complex operation allows for insertions, deletions, or substitutions of sequences of base pairs in a DNA strand.

10. **Detecting and Reading**: Once a DNA sequence is present in solution, this stage involves determining the sequence of base pairs that compose that strand of DNA in order.

These processes are standard laboratory procedures used by biologists in performing genetic analyses. It is fortunate that these operations are completely parallelizable.

9.2.1 Cutting, Ligating, and Substitution

The processes of **cutting**, **ligating** and **substitution** are frequently used together to patch together DNA sequences. The concatenation of two sequences, an operation that is very frequently used in DNA computation, is actually a sequence of a ligation, a cut, a ligation, a cut, and one simple final step, as described in [KGY]. Both cutting and ligating use the same enzymes that organisms use for the maintenance of their own DNA. For example, cutting uses a restriction endonuclease ([Ka]). The substitution operation is even more complex and requires more steps. From the perspective of algorithm design, one should regard cutting, ligating, and substitution as the main tools for "string manipulation" on DNA.

9.2.2 Amplification and the Polymerase Chain Reaction

Cells in the body need to replicate their DNA on a regular basis, and use the enzyme **DNA polymerase** to do this. The **Polymerase Chain Reaction** (PCR) repeats this process of replication many times, using the new strands created by the replication to produce many new strands in parallel. Through this process, one can acheive exponential growth: one strand becomes two, which becomes four, and so on. This process, called amplification, thus rapidly produces a very large number of copies of the original DNA. It occurs in a solution containing chemicals from which base pairs are constructed, as DNA polymerase cannot create new strands from nothing. At the end of the amplification, then, the DNA must be separated out from this solution. The actual PCR requires a series of heating and cooling cycles, and due to its prevalence in biological research, the lengths and nature of these cycles have been carefully optimized [JK].

9.2.3 Separation and Gel Electrophoresis

Gel electrophoresis is a technique that separates a solution of different DNA strands by length. Detailed accounts of this procedure are found in [LBM$^+$] and [PRS]. A prepared solution of DNA strands having known lengths is used as a kind of "ruler" for comparison with the DNA placed in other wells, and an electric current is passed through a gel in which solutions of DNA strands are placed in wells on one end. The DNA travels slowly through the gel, moved by the electromotive force, with shorter strands traveling more quickly than the longer ones due to a negative charge on the phosphate group of every nucleotide. When the current stops, so does the DNA. In this way, the DNA is separated by size, which can be measured by comparison to the "ruler." In order to see where the strands ended, one stains the molecules with ethidium bromide, which flouresces under ultraviolet light. Alternatively, one can attach radioactive labels to the DNA and use radiation screening techniques.

9.2.4 Extraction and Affinity Purification

In order to find a known DNA sequence, one can use a complementary strand—called a **probe** in [PRS]—to "fish" for that sequence. A more complicated procedure can even allow one to search for two strands in different solutions that match each other, using a method described in [KGY]. The idea behind more general examples of extraction/affinity purification is to use the natural property that DNA binds to its complement to search for sequences identical to or similar to a known sequence. One can, for example, use this technique progressively, testing larger and larger strands to sequence the DNA, in a manner described in [PRS]. From the perspective of algorithm design, one should use these techniques to test for the existence of a string in a solution.

9.3 DNA Algorithms

In order to illustrate the ways in which the operations discussed in Section 9.2 can be combined to carry out a computation, we present an example, the Bounded Post Correspondence Problem (BPCP). Our focus will be to show two aspects of this process:

- How a problem framed in computational terms can be translated into statements about DNA strands, in a method that lends itself naturally to using those strands for computation. There are many important considerations here, including, for example, the use of complementarity to "search" for a particular sequence in a solution of DNA molecules.

- How the primitive operations fit together and can be used to obtain a much more powerful procedure than originally imaginable.

The algorithm discussed will address a very important computational problem. The Bounded Post Correspondence Problem is classified as $\mathbf{NP}$-complete. There are no known efficient algorithms to solve these problems on a standard computer. The parallelism in DNA computing will become very concrete in this example.

We first define a few terms that will be essential in what follows. We may use a letter such as u to stand for a sequence of base pairs, such as $GCCTA$. Given two sequences, u and v, the **concatenation** $u \cdot v$, usually written simply as uv, is just the concatenation of these two sequences of base pairs. For example, if $u = TA$ and $v = GC$, $uv = TAGC$. Concatenation is made possible using the primitive operation of ligating.

It will be important to encode numbers as well, since these are essential to most computations. For this, we use some kind of precursor sequence followed by the number in base 4, with, say, $A = 0$, $C = 1$, $G = 2$, $T = 3$. It is also possible to write the number in binary using just, say, A and T. The details of this encoding are unimportant in what follows; what is important is that each number has a unique, known encoding.

We next introduce the computational problem, translated into the language of DNA. A detailed account of this algorithm may be found in [KGY].

Problem. *Suppose one is given two collections of DNA strands, $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$, where each u_i and v_i may be an arbitrary sequence of base pairs, as well as an integer $K \leq n$. Do there exist integers $i_1, \ldots, i_\ell$, such that $1 \leq i_j \leq n$ for $j = 1, \ldots, \ell$, and $\ell < K$, such that the concatenation $u_{i_1} \ldots u_{i_\ell}$ is the same as the concatenation $v_{i_1} \ldots v_{i_\ell}$?*

Note that the i_j may include repetitions, and that u_i and v_i are not required to have the same length.

Example 1. Let $u = (AGT, AGA, TAG, GAG)$, $v = (AG, AA, TTAG, AGGG)$, and $K = 4$. Then $u_1 \cdot u_3 = AGT \cdot TAG = AGTTAG$, which is the same as $v_1 \cdot v_3 = AG \cdot TTAG = AGTTAG$. Thus the answer is "yes."

Example 2. Let $u = (AAGTATAG, GATATCC, AGTA, CCAA)$ and $v = (AA, TA, GTA, A)$. Then every single strand of u is longer than every strand of v, so it is impossible that for some choices of i_j, $u_{i_1} \dots u_{i_\ell}$ can match $v_{i_1} \dots v_{i_\ell}$, since the first of these is a longer sequence.

The algorithm to solve this problem is as follows:

1. *Synthesizing Needed Strands*: One needs to produce DNA strands for each u_i and v_i, as well as strands encoding the integers i. For reasons that will become clear momentarily, one should choose an additional sequence we will denote $\#$ that is easily recognizable and not a substring of some concatenation of the u_i and v_i. The sequence $\#$ should also be made into DNA strands. The $\#$ strands will act as markers in strings of base pairs to separate information. Such a strand is called a **bridge** by [KGY]. The $\#$ stands should be placed into two test tubes we denote by U and V. Finally, set a counter k to be 1.

2. *Creating a Solution of Concatenations*: The following routine will be repeated, incrementing k each time:

   We pour the contents of U into n test tubes $U_1, \dots, U_n$, and similarly, V into test tubes $V_1, \dots, V_n$. Recall that this operation is called **mixing**, and is one of the primitive operations in Section 9.2. Then, for each test tube U_i, we prepend u_i to the beginning of every DNA strand inside, and append i to the end of every DNA strand inside. The same is done for each test tube V_i. Finally, we mix the contents of the U_i back into U and the contents of the V_i back into V.

   When $k = 1$, the result is a number of strands of the form $u_{i_1}\#i_1$ in U, where $1 \leq i_1 \leq n$, and similarly strands of the form $v_{i_1}\#i_1$ in V. After repeating this process for k steps, one obtains strands of the form $u_{i_1} \dots u_{i_\ell}\#i_\ell \dots i_1$, where $1 \leq i_j \leq n$ for $j = 1, \dots, \ell$, and similarly for V. Note that we have listed the indices $i_1, \dots, i_\ell$ in increasing order, but u_{i_ℓ} is in fact the first strand prepended to $\#$.

3. *Checking for Matching Strings*: After each run of the routine in Step 2, before repeating the routine, one checks for matching strings in U and V. This can be done by first using affinity purification. If a match is found, the algorithm outputs "yes" and halts. Otherwise, the value of k is incremented. If $k = K$, the algorithm outputs "no" and halts.

9.4 Algorithm Design

The algorithm in Section 9.3 suggests some general principles regarding DNA algorithm design. We discuss the relationship of the BPCP to a class of problems called $\mathbf{NP}$, and then propose a general approach to algorithm design for problems within this class.

9.4.1 Remarks on NP Computation

The class $\mathbf{P}$ contains, roughly, the computational problems a computer can solve when restricted to a polynomial number of time steps, where the polynomial is viewed as a function of the input size. For example, multiplication is such a computation, and the standard multiplication technique constitutes a polynomial-time multiplication algorithm.

The class $\mathbf{NP}$ consists of problems whose answers are easy to check, but for which it may be difficult to come up with an answer. For example, one such problem is that of determining, given a list of linear inequalities in variables x_i, whether there exists a solution in integers to all of these inequalities. It is very easy to check that a proposed solution satisfies the inequalities, but may be very difficult to determine whether one exists. A working solution is called **witness** to the solvability of the problem. Some problems, including the Bounded Post Correspondence Problem, are **complete**, and it is known that if one $\mathbf{NP}$-complete problem can be solved in polynomial time, then *every* problem in $\mathbf{NP}$ also has a polynomial-time algorithm.

The most important characterization of the class $\mathbf{NP}$ for our purposes is as follows: Any problem in $\mathbf{NP}$ may be solved by testing all strings of some bounded length n (a set that is exponential in size) using a single polynomial-time algorithm in hopes of finding a solution. In fact, a general result from complexity theory shows that this algorithm can be completely parallelized. In the example provided in Section 9.3, the algorithm produced a search space in the first two stages of the algorithm, and tested them in the third.

Another elegant algorithm to solve an $\mathbf{NP}$-complete problem, SATISFIABILITY, is presented in [BDLS]. Thus, assuming that the computational problem lies in $\mathbf{NP}$, we may divide the computation into the two processes of creating a test space and testing each strand in this space.

9.4.2 Creating a Test Space

In the computation above, all of the new strands at each stage are prepared at the same time, so the growth in the total number of strands is exponential in the number of steps. Given a sufficient quantity of DNA at the outset, one could potentially solve very large instances of the Bounded Post Correspondence Problem.

In general, it is relatively easy to generate all strings of some length by using a concatenation technique repeatedly. However, it can be fruitful to create a limited search space, all of whose members represent "likely" witnesses to the search problem. In the algorithm presented in Section 9.3, this restriction was to search for strings obtained by concatenating members from one of the two collections u and v.

9.4.3 Searching the Test Space

One next needs a technique to determine the result of the computation. Since we now have a space of test strings, we need to check, for each one, whether it is a witness for the problem. If, for example, one simply needed to check for the presence of a particular string, one might use try to match the solution with the complement of a desired strand—this is the extraction technique discussed in Section 9.2.4. One could also use a more complicated method, described in [KGY], in order to check whether there are identical strands in a pair of given solutions.

More generally, one can construct a parallelizable test for whether a witness is correct. The general procedure is somewhat technical, so we merely sketch the idea. If the top node of the circuit is an `OR` gate, then (inductively) one creates a test space S where the strings in S each are complementary strings, forming a "test" for the conditions leading into the `OR` gate. One also creates a second test space T of all strings of some length, and any string that binds to an element of S upon mixing is acceptable. If the node is a `NOT` gate, where one is trying to take the complement of a set S of strings, one can create a test space T of all strings of some length, and any string that fails to bind to an element of S is considered acceptable. In this case, the resulting strings that are left constitute the complement of the desired set, but one can just permute the alphabet to continue.

9.4.4 Procedural Efficiency

The algorithm in Section 9.3 is particularly efficient because it requires few steps. In general situations, one might benefit by recasting a computational problem in a manner amenable to easy searching of the test space, because in practice this process could become excessively complicated.

9.5 Limitations

DNA computation, as the above discussion reveals, has an enormous potential to speed up important computations by a factor of 10,000 or more. However, there are limitations to DNA computation that, while not crippling, suggest that it may not be very useful in practice.

9.5.1 Sequence length

It is very difficult to synthesize sequences much longer than 100–200 base pairs, so either DNA computation must restrict itself to using short strands, or better synthesis technology must be developed. Moreover, longer strands are less easily distinguishable by gel electrophoresis, more prone to error in amplification, and more likely to denature under slight stresses. This is a limitation that has an analogy in memory limitation with computers, but is far more severe. These errors can be crippling to many computing applications, so this could potentially threaten the ability of DNA computation to be useful for certain computational problems even if they are parallelizable. That said, for medium-sized problems, meaning problems that would take a grid of computers on the order of a month to 10 years to solve, DNA computing appears to be a very promising approach.

9.5.2 Error rate

Enzymes inherently have their own error rate, and papers such as [KGY] make special modifications to their algorithms to provide error-correction, usually using additional enzymes that serve this purpose in actual organisms. It may become the case that certain potential applications of DNA computation, which cannot tolerate even a small percentage chance of error, will fail for this reason. On the other hand, the technique of repetition and taking a majority vote of the result can always shrink a constant-sized error rate to become exponentially small, albeit at the cost of additional material use. Since the computations can be done in parallel, the repetition might not involve too substantial an increase in computing time. Researchers have even found certain error-resistant strands that could be used, for example, as the bridge in the algorithm described above. ([Ba] is a patent on some such strands.)

9.5.3 Resources

Working with DNA is very expensive, and certain procedures that seem simple in theory require a great amount of time and care to perform without errors. For a computation with $n = 100$, one would need to purchase at least 300 strands at a cost of $7,500 and wait for a lab to produce the strands. If a problem required running several such algorithms, the costs could very quickly become unmanagable. It may be that only fully automated processes such as those discussed in [RWB$^+$] will be able to cost-effectively carry out practical DNA computing algorithms.

9.6 Conclusions

DNA computation, while still in its infancy, could potentially be a new source of computing power. The largest existing parallel grids contain less than a million computers, while a single liter of DNA solution can hold the states of 10^{18} simple processors. If we could eliminate the major contributors to the cost of DNA contribution, both in time and money, by coming up with easily automated mechanisms for each type of primitive operation, then DNA computation could potentially outrun these grids, particularly on computations involving exponential-sized search spaces.

This automation of the procedures for DNA manipulation necessary for DNA computing could also help biology and chemistry researchers, who often face fairly repetitive work, a large amount of time and energy. If DNA computation ever becomes economically viable, the investments that could pour in from companies who specialize in engineering could make some great developments along the lines of mechanization as well, which could then be used for biological and chemical research.

Neither previous work on algorithms nor on parallel computation harness the full power of DNA computation, since the primitive operations for these models are so different. Algorithm design involves an interesting new set of tradeoffs between space, time, and money. The last of these considerations does not usually enter into standard algorithm design, but is frequently important when working with DNA.

References

[Ad] Leonard M. Adleman: Molecular computation of solutions to combinatorial problems, *Sci.* **266** (1994), 1021–1024.

[Ba] E. B. Baum: DNA sequences useful for computation, US Patent 6,124,444 (2000).

[BCJ$^+$] R. S. Braich, N. Chelyapov, C. Johnson, P. W. K. Rothemund, and L. Adleman: Solution of a 20-Variable 3-SAT Problem on a DNA Computer, *Sci.* **296** #5567 (2002), 499.

[BDLS] Dan Boneh, Christopher Dunworth, Richard J. Lipton, and Jiri Sgall: On the computational power of DNA, *DAMATH: Disc. App. Math. & Comb. Op. Res. & Comp. Sci.* **71** (1996).

[JK] N. Jonoska and S. A. Karl: Ligation experiments in computing with DNA, *IEEE International Conference on Evol. Comp.* (1997), 261–266.

[Ka] Lila Kari: DNA computing: arrival of biological mathematics, *The Math. Intelligencer* **19** #2 (1997), 9–22.

[KGY] Lila Kari, Greg Gloor, and Sheng Yu: Using DNA to solve the Bounded Post Correspondence Problem, *Theor. Comp. Sci.* **231** #2 (2000), 193–203.

[LBM$^+$] H. Lodish, A. Berk, P. Matsudaira, C. A. Kaiser, M. Krieger, M. P. Scott, S. L. Zipursky, and J. Darnell: *Molecular Cell Biology*, 5th ed. New York: W. H. Freeman and Co. 2003.

[PRS] G. Paun, G. Rozenberg, and A. Salomaa: *DNA Computing: New Computing Paradigms*. Berlin: Springer 1998.

[RWB$^+$] S. T. Roweis, E. Winfree, R. Burgoyne, N. V. Chelyapov, M. F. Goodman, P. W. K. Rothemund, and L. M. Adleman: A Sticker-Based Model for DNA Computation, *J. of Comp. Bio.* **5** #4 (1998), 615–630.

[YS] H. Yoshida and A. Suyamaf: Solution to 3-SAT by Breadth First Search, *DIMACS Workshop DNA Based Computers* #5 Massachusetts Institute of Technology (2000).