Chapter 11

DNA Computation

In 1994, Len Adleman showed that DNA molecules in solution could be used to perform certain types of computation. In particular he showed, and actually performed the chemical experiment, that an instance of a Hamiltonian Path problem (HPP) for a graph with 7 nodes could be solved by this method. The interest was that, because the general problem of finding Hamiltonian paths is NP complete, then maybe, by using the massive parallelism inherent in chemical processes, that this might be an approach to obtaining efficient solutions to NP hard problems. It thus encouraged a number of researchers, Lipton in particular, to see how these ideas could be improved and several articles appeared in the lay press because of the novelty.

We begin by explaining the original experiment and they show how things have progressed.

11.1 DNA computer solution to HPP

Adleman showed how to solve the hamiltonian graph problem for the 7 node graph in Figure 11.1 — that is to show there is a path from node 1 to 7 which visits every other node once only. The basic idea of the algorithm is quite simple and essentially relies on the massively parallel capabilities of DNA computers.

11.1.1 Basic Algorithm

Adleman’s algorithm is designed following the strategy, which many other DNA algorithms follow:

1. create in a test tube the whole solution space by letting DNA strands represent potential solutions,

2. by pattern matching of DNA strands extract real solutions in another test tube, and then
3. test whether the new test tube contains a DNA strand at all.

For the particular case of HPP, for a graph $G$ with $n$ vertices and source 1 and sink $n$, the more detailed non-deterministic algorithm is:

1. $S_0 =$ all random paths from source to sink,
2. $S_1 =$ keep only paths with $n$ vertices,
3. for $i = 2, \ldots, n-1$, $S_i =$ set of all paths in $S_{i-1}$ that pass through $i$
4. $G$ has a Hamiltonian path iff $S_{n-1} \neq \emptyset$.

To see how to implement this with DNA we first need some information about manipulation of DNA molecules.

### 11.1.2 DNA Details

DNA molecules are built up from a double helix of two very long strands of the bases. The bases are adenine (A), thymine (T), cytosine (C) with guanine (G). Strict base-pairing rules are adhered to: adenine will pair only with thymine (an A–T pair) and cytosine with guanine (a C–G pair). The two strands are complementary using this association and of course this is how the molecules can reproduce. Strands have directionality one end called the 5’-end and the other 3’-end. Below we see a strand with its Watson–Crick complement:

$5’-$CATGCCGATATCTGACC$-3’$

$3’-$GTACGGCTATAGACTGG$-5’$
DNA strands can be synthesized to create any specific pattern. These generated strands can be at most \(10^4\)-base long. In addition there are other operations such as: connecting, cutting, selecting, measuring length and creating complements. Such operations can be used to access information stored in DNA strands and so can be used for computation. There is a problem in that these operations can be error prone. The operations are highly parallel and can be carried out concurrently on many distinct DNA strands.

We suppose the graph has \(n\) nodes with source 1 and sink \(n\). To represent paths in the graph, we first of all choose \(n\) random 20-base sequences \(p_1, \ldots, p_n\) so that they are all easily distinguishable. We denote each of these sequences as a concatenation of two 10-base subsequences \(l_i, r_i\), so \(p_i = l_i r_i\). For example we might have

\[
p_i = \text{CATGCCGATATCTGACCTCT}
\]

\[
l_i = \text{CATGCCGATA}
\]

\[
r_i = \text{TCTGACCTCT}
\]

in the above the strands go from 5' to 3'.

For a sequence \(p_i\) we denote the complement by \(\hat{p}_i = \hat{l}_i \hat{r}_i\), so

\[
\hat{p}_i = \text{GTACGGCTATAGACTGGAGA}
\]

\[
\hat{l}_i = \text{GTACGGCTAT}
\]

\[
\hat{r}_i = \text{AGACTGGAGA}
\]

where \(\hat{p}_i\) is from 3' to 5'.

For each edge \(<i, j>\) in the graph we construct the sequence \(p_{<i,j>}\) defined as \(r_i l_j\) unless \(i = 1\) or \(j = n\) when the whole sequence is included. For example if we have edges \(<1, 2>\) and \(<2, 3>\) then these sequences are

\[
p_1 = \text{CATGCCGATATCTGACCTCT}
\]

\[
p_2 = \text{AATCCGGTAATCTGACCTCT}
\]

\[
p_3 = \text{AGTACCTATCTGACCTCT}
\]

\[
p_{<1,2>} = \text{CATGCCGATATCTGACCTCTAATCCGGGTA}
\]

\[
p_{<2,3>} = \text{ACTGATCTGTATACCTATG}
\]

Now if there is any edges \(<i, j>\) and \(<j, k>\) then we can join the sequences \(p_{<i,j>}\) and \(p_{<j,k>}\) together by using \(\hat{p}_j\) as a splint. This is illustrated below for edges \(<1, 2>\) and \(<2, 3>\):

\[
P_{<1,2>}
P_{<2,3>}
\]

\[
\text{CATGCCGATATCTGACCTCTAATCCGGGTA-ACTGATCTGTATACCTATG}
\]

\[
\text{TTAGCCCAT-TGACCTAGACA}
\]

We can now describe the implementation:
• Build up the strands representing potential paths in the graph using the technique indicated above. So we mix up in test tube $T_0$ $p_{<1,i>}$ for $i$ connected to 1 and $p_{<j,n>}$ for $j$ connected. Then for each $i = 2, \ldots, n - 1$ and for each edge $< i, j >$ mix $p_{<i,j>}$ and $\hat{p}_i$ with $T_0$. This will result in $T_0$ containing representations for paths from 1 to $n$ plus many other strands.

• For step 2 we remove all strands which do not start at 1 and terminate at $n$. This can be achieved using $\hat{p}_1$ and $\hat{p}_n$ as primers which can attach to the two ends. This results in $T_1$.

• For step 3 we eliminate all strands from $T_1$ which are not of length $20 \cdot n$.

• For $i = 2, 3, \ldots, n - 1$ collect from $T_{i-1}$ all strands containing $p_i$ forming $T_i$. (This is done by annealing to $\hat{p}_i$).

• Test if $T_{n-1}$ contains any strands. If so then there is a Hamiltonian path (and the strand encodes it) otherwise there is no such path.

Adleman managed to solve the HPP for the graph in Figure 11.1 using this technique. It took about seven days to conduct the experiment.

It is clear that the same techniques could be applied to other NP-complete problems. Lipton showed how to solve the Satisfiability problem (see next section). There have also been ideas on using DNA computers to break encryption algorithms such as DES.

With present methods it seems the largest size problems are of size up to about 60 (as the number of strands in the test tube grows exponentially). Clearly, such instances can be easily taken care of by conventional computers. In order to clarify potentiality of DNA computing for solving difficult problems, an urgent task for a computer scientist would be to discover algorithms that can cope with the exponential growth and can reduce the size of search space, thereby extending the bound on tractable input size, but researchers are looking at alternative (e.g. using other types of chemicals such as RNA).

11.2 Another Example — 3SAT

We will introduce notation for the basic operations identified by Adleman and Lipton.

$\text{append}(A, \sigma)$: For each single DNA strand in the test tube $A$, append pattern $\sigma$.

$B \leftarrow \text{extract}(A; \sigma)$: Extract from the test tube $A$ all the single DNA strands having pattern $\sigma$ and put the extracted strands into another test tube $B$.

$\text{detect}(A)$: Test whether $A$ contains a single DNA strand. The result is either true or false. Adleman suggests that this has to be done only at the end of computation, since it can destroy the DNA strands.

$\text{polymer}(A)$: Polymerizations. Create a complementary strand of each DNA strand in $A$. 

96
\[ B \leftarrow \text{mix}(A_1, \ldots, A_m) : \text{Mix test tubes } A_1, \ldots, A_m \text{ and name the mixture } B. \]

We will show how to solve the 3SAT problem which is known to be NP–complete. This is the problem of determining whether a set of clauses, with \( n \) variables, \( \{C_1, \ldots, C_m\} \) is simultaneously satisfiable where each clause is a disjunction of three literals. So each clause \( C \) is of the form \( l_1 \lor l_2 \lor l_3 \) where the \( l_i \)'s are literals \( (l_i = x_j \text{ or } l_i = \neg x_j) \). In other words we want an assignment to the variables \( \{x_1, \ldots, x_k\} \) of true or false so that each \( C_i \) is true.

We now outline Lipton’s solution.

We define strands \( t_i \) and \( f_i \) for \( i = 1, \ldots, k \) which are all different and of the same length \( L \). The idea is to represent assignments for the variables as a sequences of length \( L \cdot k \) of the \( t_i \) and \( f_i \) where for each \( i \) only one of \( t_i \) or \( f_i \) is present. It is also assumed that

1. for all \( i \) \( t_i \) and \( f_i \) are complementary,

2. for \( \sigma \) from \( t_i \) and \( \tau \) from \( f_i \), they disagree with each other at least at a half of the positions and no nonempty prefix of \( \sigma \) is a suffix of \( \tau \).

![Figure 11.2: Truth Assignment Graph](image)

A truth assignment corresponds to a path (from source to sink) in the above graph, Figure 11.2. Here is the DNA algorithm:

**Step 1:** Create, in a test tube, many copies of \( t_i \) and \( f_i \) for all \( i = 1, \ldots, k \). Connect these strands together and eliminate strands of length greater than \( L \cdot k \). Extract strands matching one of \( t_i \) and \( f_i \) for all \( i \). This is done by executing sequentially for all \( i \); \( 1 \leq i \leq k \), \( A \leftarrow \text{extract}(T, t_i) \); \( A' \leftarrow \text{extract}(T, f_i) \); \( T \leftarrow \text{mix}(A, A') \). The property (2) above guarantees that the extracted strands are of length exactly \( L \cdot k \), and thus, extracted strands represent full assignments.

**Step 2:** Sequentially for each clause \( l_1 \lor l_2 \lor l_3 \), extract from \( T \) the strands that satisfy one of the three literals. More precisely, execute \( A_1 \leftarrow \text{extract}(T, \bar{l}_1) \); \( A_2 \leftarrow \text{extract}(T, \bar{l}_2) \); \( A_3 \leftarrow \text{extract}(T, \bar{l}_3) \); \( T \leftarrow \text{mix}(A_1, A_2, A_3) \) in this order.
Step 3: Execute \texttt{detect}(T). If the result is positive, the formula is satisfiable. Otherwise, it is not satisfiable.

In the above \( \bar{l} \) is \( t_i \) if \( l = x_i \) and \( f_i \) if \( l = \neg x_i \). At the first application of stage2, \( T \) contains just the strings satisfying \( C_1 \). The next application \( T \) contains strings satisfying \( C_1 \land C_2 \) and so on until finally \( T \) contains only strings (if any) satisfying \( C_1 \land C_2 \land \cdots \land C_m \) and so stage 3 checks if there are any such strings present.

The number of necessary biochemical operations is \( k - 1 \) for connecting patterns, 1 for extracting length \( \leq L \cdot k \) strands, \( 3k \) for extracting strands that represent assignments, \( 4m \) for Step 2, and 1 for Step 3.

Thus, we have:

\textbf{Theorem 1} \textit{Lipton’s method runs in time} \( 4m + 4k + 1 \) \textit{and uses space} \( 2^k \).

We finally note that the chemical reactions are not free from error. They are affected by temperature, ionic strength of the solvent, and the base sequence of the interacting strands. At this moment, it is not perfectly clear in what accuracy these operations can be performed. Reliability of the above operations has to be rigorously tested.

11.3 \hspace{1em} \textbf{Advantages and Disadvantages}

11.3.1 \hspace{1em} \textbf{Advantages}

\textbf{Parallelism}

DNA computers are massively parallel — \( 10^{14} \) MIPS. Conventional computers \( 10^9 \) MIPS. The chemical reactions take place in parallel.

\textbf{Lightweight}

DNA computer have tremendous potential computing power per gram.

\textbf{Low Power}

DNA computers use no power supply. The only power necessary is to keep DNA from denaturing. DNA computers could be a billion times as energy efficient as conventional computers.

\textbf{Solves Complex Problems}

DNA computers have the potential to solve intractable problems such as NP-complete problems. However, whether current technology can scale up to tackle large examples needs investigation.
11.3.2 Disadvantages

Solves only Combinatorial Problems

The type of problems which have been tackled by DNA computers have all been basically combinatorial in nature.

Procedure Slow

The current approaches take weeks to perform the experiments. It needs to be shown that problems can be solved which are larger than those that can be tackled by conventional computers to justify the time taken.

Scalability

One of the main weaknesses of Adleman’s procedure is that the number of necessary single strands, codes of vertices or of edges, is of the order of \( n! \), where \( n \) is the number of vertices in the graph. This imposes drastic limitations on the size of the problems which can be solved in this manner. J.Hartmanis showed that in order to handle in this manner graphs with 200 nodes, graphs with a size of practical importance, easily handled by conventional computers, we need \( 3 \cdot 10^{25} \) Kg of DNA — more than the weight of the Earth! Adleman’s original procedure is elegant, copes in a nice way with the errors, but it cannot be scaled-up.

Several improvements of Adleman’s procedure have been proposed. The most promising approach is to use an evolutionary computing strategy. Instead of constructing a complete data pool from which the solutions are then filtered out, only good candidates are produced, which saves a lot of the needed DNA.

Reliability

Some of the basic operations on DNA can be error prone and DNA can deteriorate or mutate. Work is being undertaken to deal with such errors.