B Filtering models

Filtering models, an important class of DNA algorithms, operate by filtering out of the solution molecules that are not part of the desired result. A chemical solution can be treated mathematically as a finite bag or multi-set of molecules, and filtering operations can be treated as operations to produce multi-sets from multi-sets. Typically, for a problem of size $n$, strings of size $O(n)$ are required. The chemical solution should contain enough strings to include many copies all possible answers. Therefore, for an exponential problem, we will have $O(k^n)$ strings. Filtering is essentially a brute-force method of solving problems.

B.1 Adleman: HPP

B.1.a Review of HPP

Leonard Adleman’s solution of the Hamiltonian Path Problem was the first successful demonstration of DNA computing. The Hamiltonian Path Problem (HPP) is to determine, for a given directed graph $G = (V,E)$ and two of its vertices $v_{in}, v_{out} \in V$, whether there is a Hamiltonian path from $v_{in}$ to $v_{out}$, that is, a path that goes through each vertex exactly once. HPP is an NP-complete problem, but we will see that for Adleman’s algorithm the number of algorithm steps is linear in problem size.

Adleman (the “A,” by the way, of “RSA.”) gave a laboratory demonstration of the procedure in 1994 for $n = 7$, which is a very small instance of HPP. (We will use this instance, shown in Fig. IV.7, as an example.) Later his group applied similar techniques to solving a 20-variable 3-SAT problem, which has more than a million potential solutions (see p. 216).\footnote{https://en.wikipedia.org/wiki/Adleman,_Leonard (accessed 2012-11-04)}

B.1.b Problem Representation

The heart of Adleman’s algorithm is a clever way to encode candidate paths in DNA. Vertices are represented by single-stranded 20mers, that is, sequences of 20nt (nucleotides). They were generated at random and assigned to the vertices, but with the restriction that none of them were too similar or complementary. Each vertex code can be considered a catenation of two 10mers: $v_i = a_i b_i$ (i.e., $a_i$ is the 5’ 10mer and $b_i$ is the 3’ 10mer). Edges are also
5.4 Adleman’s Implementation

Adleman utilized the incredible storage capacity of DNA to implement a brute-force algorithm for the directed Hamiltonian Path Problem (HPP). Recall that the HPP involves finding a path through a graph that visits each vertex exactly once. The instance of the HPP that Adleman solved is depicted in Fig. 5.2, with the unique Hamiltonian Path (HP) highlighted by a dashed line.

![Figure IV.7: HPP solved by Adleman. The HP is indicated by the dotted edges. (source: Amos, Fig. 5.2)](image)

Adleman’s approach was simple:

1. Generate strands encoding random paths such that the Hamiltonian Path (HP) is represented with high probability. The quantities of DNA used far exceeded those necessary for the small graph under consideration, so it is likely that many strands encoding the HP were present.

2. Remove all strands that do not encode the HP.

3. Check that the remaining strands encode a solution to the HPP.

The individual steps were implemented as follows:

Stage 1: Each vertex and edge was assigned a distinct 20-mer sequence of DNA (Fig. 5.3a). This implies that strands encoding a HP were of length 140 b.p. Sequences representing edges act as ‘splints’ between strands representing their endpoints (Fig. 5.3b).

In formal terms, the sequence associated with an edge \(i \rightarrow j\) is the 3' 10-mer of the sequence representing \(v_i\) followed by the 5' 10-mer of the sequence representing \(v_j\). The oligonucleotides were then combined to form strands encoding random paths through the graph. An (illegal) example path (\(v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4\)) is depicted in Fig. 5.4.

Paths are represented by using complements of the vertex 20mers to stitch together the edge 20mers. (Of course, using the complements of the edges to stitch together the vertices works as well.) For example, a path 2 → 3 → 4 is represented:

\[
\begin{align*}
&\text{e}_2 \rightarrow 3 = b_2 a_3 \\
&\text{e}_3 \rightarrow 4 = b_3 a_4 \\
&\text{v}_3 \rightarrow \text{v}_2 \rightarrow \text{v}_4 \\
&\text{v}_1 \rightarrow \text{v}_2 \rightarrow \text{v}_3 \\
&\text{v}_2 \rightarrow \text{v}_3 \rightarrow \text{v}_4
\end{align*}
\]

The edges from \(v_{\text{in}}\) and to \(v_{\text{out}}\) have special representations as 30mers:

\[
e_{\text{in} \rightarrow j} = v_{\text{in}} a_j, \text{ where } v_j = a_j b_j,
\]

\[
e_{\text{i} \rightarrow \text{out}} = b_i v_{\text{out}}, \text{ where } v_i = a_i b_i.
\]

Note that the special representation of the initial and terminal edges results in blunt ends for complete paths.

Therefore, for the \(n = 7\) problems, candidate solutions were 140bp in length: There are \(n - 1\) edges, but the first and last edges are 30mers. Hence \(2 \times 30 + (n - 3) \times 20 = 140\). Ligation is used to remove the nicks in the backbone.
B.1.c  Adleman’s Algorithm

algorithm Adlemen:

Step 1 (generation of all paths): Generate multiple representations of all possible paths through the graph. This is done by combining the oligos for the edges with the oligos for the complements of the vertices in a single ligation reaction.

Step 2: Amplify the concentration of paths beginning with \( v_{in} \) and ending with \( v_{out} \). This is done by PCR using \( v_{in} \) and \( \overline{v}_{out} \) as primers. Remember that denaturation separates the sense and antisense strands. PCR extends the sense strand in the 3' direction from \( v_{in} \), and extends the antisense strand in the 3' direction from \( \overline{v}_{out} \). At the end of this step we have paths of all sorts from \( v_{in} \) to \( v_{out} \).

Step 3: Only paths with the correct length are retained; for \( n = 7 \) this is 140bp. This operation is accomplished by gel electrophoresis. The band corresponding to 140bp is determined by comparison with a marker lane, and the DNA is extracted from this band and amplified by PCR. Gel electrophoresis and PCR are repeated to get a sufficient quantity of the DNA. We now have paths from \( v_{in} \) to \( v_{out} \), but they might not be Hamiltonian.

Step 4 (affinity purification): Select for paths that contain all the vertices (and are thus necessarily Hamiltonian). This is done by first selecting all those paths that contain \( v_1 \), and then, of those, all that contain \( v_2 \), and so forth. To select for \( v_i \), first heat the solution to separate the strands. Then add the \( \overline{v}_i \) bound to a magnetic bead. Rehybridize (so the beads are bound to strands containing \( v_i \)), and use a magnet to extract all the paths containing \( v_i \). Repeat this process for each vertex.

Step 5 If there any paths left, they are Hamiltonian. Therefore amplify them by PCR and inspect the result by gel electrophoresis to see if there are any
strands of the correct length. If there are, then there is a Hamiltonian path; if there aren’t, then there is not. If desired, the precise HP can be determined by a graduated PCR procedure: Run $n - 1$ parallel PCR reactions. In the $i$th lane, $v_{in}$ is the left primer and $v_i$ is the right primer. This will produce bands with lengths $40, 60, 80, 100, 120, \text{ and } 140$ bp. The lane that has a band at $40$ corresponds to the first vertex after $v_{in}$ in the path, the lane with a band at $60$ corresponds to the next vertex, etc. This final readout process depends on there being only one Hamiltonian path, and it is error-prone due to its dependence on PCR.

\[
\]

B.1.d DISCUSSION

Adleman’s algorithm is linear in the number of nodes, since the only iteration is Step 4, which is repeated for each vertex. Step 5 is also linear if the path is read out. Thanks to the massive parallelism of molecular computation, it solves this NP-complete problem in linear time. Adleman’s experiment took about a week, but with a more automated approach it could be done in a few hours. On the other hand, the PCR process cannot be significantly shortened.

In addition to time, we need to consider the molecular resources required. The number of different oligos required is proportional to $n$, but the number of strands is much larger, since there must be multiples instances of each
possible path. If $d$ is the average degree of the graph, then there are about $d^n$ possible paths (exponential in $n$). For example, if $d = 10$ and $n = 80$, then the required $10^{80}$ DNA molecules is more than the estimated number of atoms in the universe. Hartmanis calculated that for $n = 200$ the weight of the DNA would exceed the weight of the earth. So this brute-force approach is still defeated by exponential explosion. Lipton (1995) estimates that Adleman’s algorithm is feasible for $n \leq 70$, based on an upper limit of $10^{21} \approx 2^{70}$ DNA strands (Boneh, Dunworth, Lipton & Sgall, 1996), but this is also feasible on conventional computers.

Nevertheless, Adleman’s algorithm illustrates the massive parallelism of molecular computation. Step 1 (generation of all possible paths) took about an hour for $n = 7$. Adleman estimates that about $10^{14}$ ligation operations were performed, and that it could be scaled up to $10^{20}$ operations. Therefore, speeds of about $10^{15}$ to $10^{16}$ ops/sec (1–10 peta-operations/s) should be achievable, which is, digital supercomputer range. Adleman also estimates that $2 \times 10^{19}$ ligation operations were performed per joule of energy. Contemporary supercomputers perform only $10^9$ operations per joule, so molecular computation is $10^{10}$ more energy-efficient. It is near the thermodynamic limit of $34 \times 10^{19}$ operations per joule. Recall (Ch. II, Sec. B.1) $kT \ln 2 \approx 3 \times 10^{-9}$ pJ $= 3 \times 10^{-21}$ J, so there can be about $3.3 \times 10^{20}$ bit changes/J.\footnote{DNA is of course space efficient. One bit of information occupies about 1 cubic nm, whereas contemporary disks store a bit in about $10^{10}$ cubic nm. That is, DNA is a $10^{10}$ improvement in density.}

A more pervasive problem is the inherent error in the filtering processes (due to incorrect hybridization). Some strands we don’t want, get through; and some that we do want, don’t. With many filtering stages the errors accumulate to the extent that the algorithms fail. There are some approaches to error-resistant DNA computing, but this is an open problem.
B. FILTERING MODELS

213

Figure IV.9: Graph $G_2$ for Lipton’s algorithm (with two variables, $x$ and $y$). [source: Lipton (1995)]

B.2 Lipton: SAT

In this section we will discuss DNA solution of another classic NP-complete problem, *Boolean satisfiability*, in fact the first problem proved to be NP-complete.\(^6\)

B.2.a Review of SAT problem

In the Boolean satisfiability problem (called “SAT”), we are given a Boolean expression of $n$ variables. The problem is to determine if the expression is *satisfiable*, that is, if there is an assignment of Boolean values to the variables that makes the expression true.

Without loss of generality, we can restrict our attention to expressions in *conjunctive normal form*, for every Boolean expression can be put into this form. That is, the expression is a conjunction of *clauses*, each of which is a disjunction of either positive or negated variables, such as this:

$$(x_1 \lor x'_2 \lor x'_3) \land (x_3 \lor x'_5 \lor x_6) \land (x_3 \lor x'_6 \lor x_4) \land (x_4 \lor x_5 \lor x_6),$$

For convenience we use primes for negation, for example, $x'_2 = \neg x_2$. In the above example, we have $n = 6$ variables $m = 4$ clauses. The (possibly negated) variables are called *literals*.

B.2.b Data representation

To apply DNA computation, we have to find a way to represent potential solutions to the problem as DNA strands. Potential solutions to SAT are

n-bit binary strings, which can be thought of as paths through a particular graph $G_n$ (see Fig. IV.9). For vertices $a_k, x_k, x'_k, k = 1, \ldots, n$, and $a_{n+1}$, there are edges from $a_k$ to $x_k$ and $x'_k$, and from $x_k$ and $x'_k$ to $a_{k+1}$. Binary strings are represented by paths from $a_1$ to $a_{n+1}$. A path that goes through $x_k$ encodes the assignment $x_k = 1$ and a path through $x'_k$ encodes $x_k = 0$. The DNA encoding of these paths is essentially the same as in Adleman’s algorithm.

B.2.c Lipton’s Algorithm

algorithm Lipton:

Input: Suppose we have an instance (formula) to be solved: $I = C_1 \land C_2 \land \cdots \land C_m$. The algorithm will use a series of “test tubes” (reaction vessels) $T_0, T_1, \ldots, T_m$ and $T_1^i, T_2^i, \ldots, T_m^i, T_m^i$, for $i = 0, \ldots, n$.

Step 1 (initialization): Create in a test tube $T_0$ a library of all possible $n$-bit binary strings, encoded as above as paths through the graph.

Step 2 (clause satisfaction): For each clause $C_k$, $k = 1, \ldots, m$: we will extract from $T_{k-1}$ only those strings that satisfy $C_k$, and put them in $T_k$. (These successive filtrations in effect do an AND operation.) The goal is that the DNA in $T_k$ satisfies the first $k$ clauses of the formula. That is, $\forall x \in T_k \forall 1 \leq j \leq k : C_j(x) = 1$. Here are the details.

For $k = 0, \ldots, m - 1$ do the following steps:

Precondition: The strings in $T_k$ satisfy clauses $C_1, \ldots, C_k$.

Let $\ell = |C_{k+1}|$ (the number of literals in $C_{k+1}$), and suppose $C_{k+1}$ has the form $v_1 \lor \cdots \lor v_\ell$, where the $v_i$ are literals (positive or negative variables). Our goal is to find all strings that satisfy at least one of these literals. To
accomplish this we will use an *extraction operation* $E(T, i, a)$ that extracts from test tube $T$ all (or most) of the strings whose $i$th bit is $a$.

Let $T_0 = T_k$. Do the following for literals $i = 1, \ldots, \ell$.

**Positive literal:** Suppose $v_i = x_j$ (some positive literal). Let $T^i_k = E(T_{i-1}^i, j, 1)$ and let $a = 1$ (used below). These are the paths that satisfy this positive literal, since they have 1 in position $j$.

**Negative literal:** Suppose $v_i = x'_j$ (some negative literal). Let $T^i_k = E(T_{i-1}^i, j, 0)$ and let $a = 0$. These are the paths that satisfy this negative literal, since they have 0 in position $j$.

In either case, $T^i_k$ are the strings that satisfy literal $i$ of the clause. Let $T^i_k = E(T_{i-1}^i, j, \neg a)$ be the remaining strings (which do not satisfy this literal). Continue the process above until all the literals in the clause are processed. At the end, for each $i = 1, \ldots, \ell$, $T^i_k$ will contain the strings that satisfy literal $i$ of clause $k$.

Combine $T^1_k, \ldots, T^\ell_k$ into $T_{k+1}$. (Combining the test tubes effectively does OR.) These will be the strings that satisfy at least one of the literals in clause $k + 1$.

**Postcondition:** The strings in $T_{k+1}$ satisfy clauses $C_1, \ldots, C_{k+1}$.

Continue the above for $k = 1, \ldots, m$.

**Step 3 (detection):** At this point, the strings in $T_m$ (if any) are those that satisfy $C_1, \ldots, C_m$, so do a *detect* operation (for example, with PCR and gel electrophoresis) to see if there are any strings left. If there are, the formula is satisfiable; if there aren’t, then it is not.

If the number of literals per clause is fixed (as in the 3-SAT problem), then performance is linear in $m$. The main problem with this algorithm is the
effect of errors, but imperfections in extraction are not fatal, so long as there are enough copies of the desired sequence. In 2002, Adelman’s group solved a 20-variable 3-SAT problem with 24 clauses, finding the unique satisfying string. In this case the number of possible solutions is $2^{20} \approx 10^6$. Since the degree of the specialized graph used for this problem is small, the number of possible paths is not excessive (as it might be in the Hamiltonian Path Problem). They stated, “This computational problem may be the largest yet solved by nonelectronic means,” and they conjectured that their method might be extended to 30 variables.

---

B. FILTERING MODELS

B.3 Test tube programming language

Filtering algorithms use a small set of basic DNA operations, which can be extended to a Test Tube Programming Language (TTPL), such as was developed in the mid 90s by Lipton and Adleman.

B.3.a Basic Operations

DNA algorithms operate on “test tubes,” which are multi-sets of strings over \( \Sigma = \{A, C, T, G\} \). There are four basic operations (all implementable):

**Extract (or separate):** There are two complementary *extraction* (or *separation*) operations. Given a test tube \( t \) and a string \( w \), \((t, w)\) returns all strings in \( t \) that have \( w \) as a subsequence:

\[
(t, w) \overset{\text{def}}{=} \{ s \in t \mid \exists u, v \in \Sigma^* : s = uwv \}.
\]

Likewise, \((- (t, w))\) returns a test tube of all the remaining strings:

\[
-(t, w) \overset{\text{def}}{=} t - +(t, w) \quad \text{(multi-set difference)}.
\]

**Merge:** The merge operation combines several test tubes into one test tube:

\[
\bigcup(t_1, t_2, \ldots, t_n) \overset{\text{def}}{=} t_1 \cup t_2 \cup \cdots \cup t_n.
\]

**Detect:** The detect operation determines if any DNA strings remain in a test tube:

\[
detect(t) \overset{\text{def}}{=} \begin{cases} 
\text{true,} & \text{if } t \neq \emptyset \\
\text{false,} & \text{otherwise} 
\end{cases}.
\]

**Amplify:** Given a test tube \( t \), the amplify operation produces two copies of it: \( t', t'' \leftarrow \text{amplify}(t) \). Amplification is a problematic operation, which depends on the special properties of DNA and RNA, and it may be error prone. Therefore it is useful to consider a *restricted model* of DNA computing that avoids or minimizes the use of amplification.

The following additional operations have been proposed:

**Length-separate:** This operation produces a test tube containing all the strands less than a specified length:

\[
(t, \leq n) \overset{\text{def}}{=} \{ s \in t \mid |s| \leq n \}.
\]
Position-separate: There are two position-separation operations, one that selects for strings that begin with a given sequence, and one for sequences that end with it:

\[
B(t, w) \overset{\text{def}}{=} \{ s \in t \mid \exists v \in \Sigma^* : s = wv \}, \\
E(t, w) \overset{\text{def}}{=} \{ s \in t \mid \exists u \in \Sigma^* : s = uw \}.
\]

B.3.b Examples

AllC: The following example algorithm detects if there are any sequences that contain only C:

\[
\text{procedure } [\text{out}] = \text{AllC}(t, A, T, G) \\
\text{t } \leftarrow -(t, A) \\
\text{t } \leftarrow -(t, T) \\
\text{t } \leftarrow -(t, G) \\
\text{out } \leftarrow \text{detect} (t) \\
\text{end procedure}
\]

HPP: Adelman’s solution of the HPP can be expressed in TTPL:

\[
\text{procedure } [\text{out}] = \text{HPP}(t, \text{vin}, \text{vout}) \\
\text{t } \leftarrow B(t, \text{vin}) \quad /\!\!/ \text{begin with vin} \\
\text{t } \leftarrow E(t, \text{vout}) \quad /\!\!/ \text{end with vout} \\
\text{t } \leftarrow (t, \leq 140) \quad /\!\!/ \text{correct length} \\
\text{for } i=1 \text{ to } 5 \text{ do} \quad /\!\!/ \text{except vin and vout} \\
\text{t } \leftarrow +(t, s[i]) \quad /\!\!/ \text{contains vertex i} \\
\text{end for} \\
\text{out } \leftarrow \text{detect}(t) \quad /\!\!/ \text{any HP left?} \\
\text{end procedure}
\]

SAT: Programming Lipton’s solution to SAT requires another primitive operation, which extracts all sequences for which the jth bit is \( a \in \{0,1\} \): \( E(t, j, a) \). Recall that these are represented by the sequences \( x_j \) and \( x'_j \). Therefore:

\[
E(t, j, 1) = +(t, x_j), \\
E(t, j, 0) = +(t, x'_j).
\]
procedure \( \text{out} = \text{SAT}(t) \)

\[
\begin{align*}
\text{for } & k = 1 \text{ to } m \text{ do } \quad \text{// for each clause} \\
\text{for } & i = 1 \text{ to } n \text{ do } \quad \text{// for each literal} \\
& \text{if } C[k][i] = x_j \quad \text{// i-th literal in clause } k \\
& \quad \text{then } t[i] \leftarrow E(t,j,1) \\
& \quad \text{else } t[i] \leftarrow E(t,j,0) \\
& \text{end if} \\
& \text{end for} \\
& t \leftarrow \text{merge}(t[1], t[2], \ldots, t[n]) \quad \text{// solutions for clauses } 1..k \\
& \text{end for} \\
& \text{out } \leftarrow \text{detect}(t) \\
\text{end procedure}
\]
B.4 Parallel filtering model

The parallel filtering model (PFM) was developed in the mid 90s by Martyn Amos and colleagues to be a means of describing DNA algorithms for any NP problem (as opposed to Ableson’s and Lipton’s algorithms, which are specialized to particular problems). “Our choice is determined by what we know can be effectively implemented by very precise and complete chemical reactions within the DNA implementation.” All PFM algorithms begin with a multi-set of all candidate solutions. The PFM differs from other DNA computation models in that removed strings are discarded and cannot be used in further operations. Therefore this is a “mark and destroy” approach to DNA computation.

B.4.a Basic operations

The basic operations are remove, union, copy, and select.

Remove: The operation remove($U, \{S_1, \ldots, S_n\}$) removes from $U$ any strings that contain any of the substrings $S_i$. Remove is implemented by two primitive operations, mark and destroy:

Mark: mark($U, S$) marks all strands that have $S$ as a substring. This is done by adding $S$ as a primer with polymerase to make it double-stranded.

Destroy: destroy($U$) removes all the marked sequences from $U$. This is done by adding a restriction enzyme that cuts up the double-stranded part. These fragments can be removed by gel electrophoresis, or left in the solution (since they won’t affect it). Restriction enzymes are much more reliable than other DNA operations, which is one advantage of the PFM approach.

Union: The operation union($\{U_1, \ldots, U_n\}, U$) combines in parallel the multi-sets $U_1, \ldots, U_n$ into $U$.

Copy: The operation copy($U, \{U_1, \ldots, U_n\}$) divides multi-set $U$ into $n$ equal multi-sets $U_1, \ldots, U_n$.

Select: The operation select($U$) returns a random element of $U$. If $U = \emptyset$, then it returns $\emptyset$.

Homogeneous DNA can be detected and sequenced by PCR, and nested PCR can be used in non-homogeneous cases (multiple solutions). All of these operations are assumed to be constant-time. Periodic amplification (especially after copy operations) may be necessary to ensure an adequate

---

8Amos, p. 50.
B. FILTERING MODELS

Figure IV.10: Remove operation implemented by mark and destroy. [source: Amos]

number of instances. Amos et al. have done a number of experiments to determine optimum reactions parameters and procedures.

B.4.b PERMUTATIONS

Amos et al. describe a PFM algorithm for generating all possible permutations of a set of integers.

algorithm Permutations:

Input: “The input set $U$ consists of all strings of the form $p_1i_1p_2i_2\cdots p_ni_n$ where, for all $j$, $p_j$ uniquely encodes ‘position $j$’ and each $i_j$ is in $\{1, 2, \ldots, n\}$. Thus each string consists of $n$ integers with (possibly) many occurrences of the same integer.”

---

Amos, p. 51.
Iteration:

for \( j = 1 \) to \( n - 1 \) do
  copy(\( U, \{U_1, U_2, \ldots, U_n\}\))
  for \( i = 1, 2, \ldots, n \) and all \( k > j \)
    in parallel do remove(\( U_i, \{p_ji_j \neq p_ji, p_ki\}\))
    // \( U_i \) contains \( i \) in \( j \)th position and no other is
    union({\( U_1, U_2, \ldots, U_n \), \( U \)})
  end for

\( P_n \leftarrow U \)

In the preceding, \( \text{remove}(U_i, \{p_ji_j \neq p_ji, p_ki\}) \) means to remove from \( U_i \) all strings that have a \( p_j \) value not equal to \( i \) and all strings containing \( p_ki \) for any \( k > j \). For example, if \( i = 2 \) and \( j = n - 1 \), this remove operation translates to remove(\( U_2, \{p_{n-1}1, p_{n-1}3, p_{n-1}4, \ldots, p_{n-1}n, p_n2\}\)). That is, it eliminates all strings except those with 2 in the \( n - 1 \) position, and eliminates those with 2 in the \( n \) position. At the end of iteration \( j \) we have:

\[
\begin{array}{c}
\alpha \\
p_{i_1}i_1p_{i_2}i_2 \cdots p_{i_j}i_j \cdots p_{i_{j+1}}i_{j+1} \cdots p_{i_n}i_n
\end{array}
\]

where \( \alpha \) represents a permutation of \( j \) integers from 1, \ldots, \( n \), and none of these integers \( i_1, \ldots, i_j \) are in \( \beta \).

Amos shows how to do a number of NP-complete problems, including 3-vertex-colorability, HPP, subgraph isomorphism, and maximum clique.