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 $\Sigma = \{\text{A, C, T, G}\}$. 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:

$$+(t, w) \stackrel{\text{def}}{=} \{s \in t \mid \exists u, v \in \Sigma^* : s = uwv\}.$$

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

$$-(t, w) \stackrel{\text{def}}{=} t \; - \; +(t, w) \quad \text{(multi-set difference)}.$$

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

$$\cup(t_1, t_2, \ldots, t_n) \stackrel{\text{def}}{=} t_1 \cup t_2 \cup \cdots \cup t_n.$$

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

$$\text{detect}(t) \stackrel{\text{def}}{=} \begin{cases} \textbf{true}, & \text{if } t \neq \emptyset \\ \textbf{false}, & \text{otherwise} \end{cases}.$$

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:

$$(t, \leq n) \stackrel{\text{def}}{=} \{s \in t \mid |s| \leq n\}.$$

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:

$$\begin{aligned} B(t,w) &\stackrel{\text{def}}{=} \{s \in t \mid \exists v \in \Sigma^* : s = wv\}, \\ E(t,w) &\stackrel{\text{def}}{=} \{s \in t \mid \exists u \in \Sigma^* : s = uw\}. \end{aligned}$$

B.3.b Examples

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

```
procedure [out] = AllC(t, A, T, G)
  t ← –(t, A)
  t ← –(t, T)
  t ← –(t, G)
  out ← detect (t)
end procedure
```

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

```
procedure [out] = HPP(t, vin, vout)
  t ← B(t, vin)                  //begin with vin
  t ← E(t, vout)                 //end with vout
  t ← (t, ≤ 140)                 //correct length
  for i=1 to 5 do                //except vin and vout
    t ← +(t, s[i])               //contains vertex i
  end for
  out ← detect(t)                //any HP left?
end procedure
```

SAT: Programming Lipton's solution to Sat requires another primitive operation, which extracts all sequences for which the jth bit is $a \in \mathbf{2}$: $E(t, j, a)$. Recall that these are represented by the sequences x_j and x'_j. Therefore:

$$\begin{aligned} E(t,j,1) &= +(t, x_j), \\ E(t,j,0) &= +(t, x'_j). \end{aligned}$$

```
procedure [out] = SAT(t)
  for k = 1 to m do                  // for each clause
    for i = 1 to n do                // for each literal
      if C[k][i] = x_j               // i-th literal in clause k
        then t[i] ← E(t,j,1)
        else t[i] ← E(t,j,0)
      end if
    end for
    t ← merge(t[1], t[2], ..., t[n]) // solutions for clauses 1..k
  end for
  out ← detect(t)
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 algorthms, 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."[8] 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 $\overline{S}$ 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

[8] Amos, p. 50.

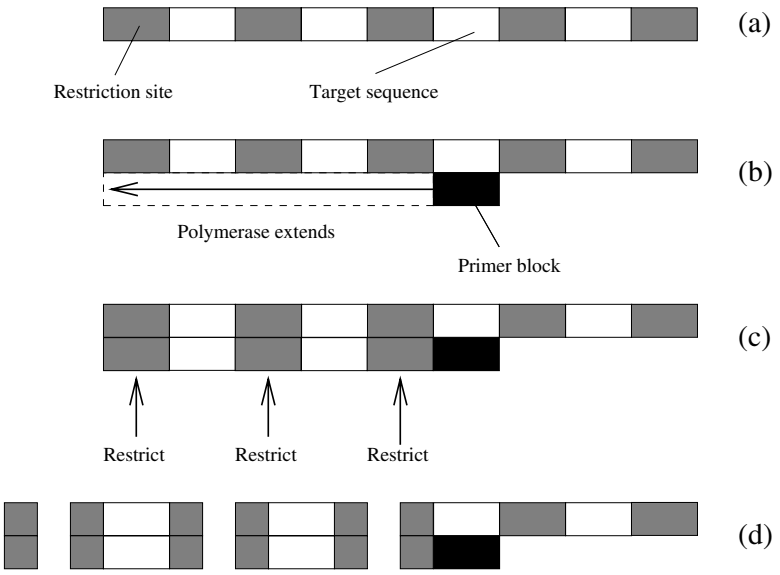

Figure IV.10: *Remove* operation implemented by *mark* and *destroy*. [source: Amos]

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 $p_1 i_1 p_2 i_2 \cdots p_n i_n$ 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."[9]

[9] 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$(U_i, \{p_j i_j \neq p_j i, p_k i\})$
  // U_i contains i in jth position and no other is
  union$(\{U_1, U_2, \ldots, U_n\}, U)$
end for
$P_n \leftarrow U$

In the preceding, remove$(U_i, \{p_j i_j \neq p_j i, p_k i\})$ means to remove from U_i all strings that have a p_j value not equal to i and all strings containing $p_k i$ for any $k > j$. For example, if $i = 2$ and $j = n - 1$, this remove operation translates to remove$(U_2, \{p_{n-1}1, p_{n-1}3, p_{n-1}4, \ldots, p_{n-1}n, p_n 2\})$. 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:

$$\overbrace{p_1 i_1 p_2 i_2 \cdots p_j i_j}^{\alpha} \underbrace{p_{j+1} i_{j+1} \cdots \ p_n i_n}_{\beta}$$

where α represents a permutation of j integers from $1, \ldots, n$, and none of these integers $i_1, \ldots, i_j$ are in β.

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