CHAPTER IV. MOLECULAR COMPUTATION

B.3 Test tube programming language

1. Test Tube Programming Language (TTPL): These ideas can be extended to a Test Tube Programming Language (TTPL).

2. Developed in the mid 90s by Lipton and Adleman.

B.3.a Basic Operations

1. DNA algorithms operate on “test tubes,” which are multi-sets of strings over $\Sigma = \{A, C, T, G\}$.

2. There are four basic operations (all implementable):

3. 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:

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

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

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

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

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

6. Amplify: Given a test tube $t$, the amplify operation produces two copies of it: $t', t'' \leftarrow \text{amplify}(t)$.

7. Restricted model: Amplification is a problematic operation, which depends on the special properties of DNA and RNA. Also 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.
8. The following additional operations have been proposed:

9. **Length-separate:** Produces a test tube containing all the strands less than a specified length:

\[(t, \leq n) \overset{\text{def}}{=} \{s \in t \mid |s| \leq n\} \]

10. **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:

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

B.3.b **Examples**

1. **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
```

2. **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, \leq 140) //correct length
    for i=1 to 5 do
        t ← +(t, s[i]) //contain vertex i
    end for
    out ← detect(t) //any HP left?
end procedure
```
3. **SAT**: Programming Lipton's solution to SAT requires another primitive operation, which extracts all sequences for which the \( j \)th bit is \( a \in 2 \): \( E(t, j, a) \).

Recall that these are represented by the sequences \( x_j \) and \( x'_j \). Therefore:

\[
\begin{align*}
E(t, j, 1) &= +(t, x_j), \\
E(t, j, 0) &= +(t, x'_j).
\end{align*}
\]

4. **procedure** \([\text{out}] = \text{Sat}(t)\)

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