Constructing DNA-based Parallel Adder with Basic Logic Operations in the Adleman-Lipton Model

Sientang Tsai

Department of Information Management, Southern Taiwan University of Science and Technology, Taiwan

*Corresponding author: tsai@mail.stust.edu.tw

Copyright © 2013 Horizon Research Publishing All rights reserved.

Abstract It is shown first by Adleman that deoxyribonucleic acid (DNA) strands could be employed towards calculating solution to an instance of The NP-complete Hamiltonian Path Problem (HPP). Lipton also demonstrated that Adleman's techniques could be used to solve the satisfiability (SAT) problem. In this paper, it is demonstrated how the DNA operations presented by Adleman and Lipton can be used to construct bio-molecular parallel adder with basic logic operations in the Adleman-Lipton model.

Keywords Parallelism, Adleman-Lipton Model, DNA-based Computing, NP-complete Problem

1. Introduction

Through advances in molecular biology [1], it is now possible to produce roughly 1018 DNA strands that fit in a test tube. Those 1018 DNA strands can also be applied to represent 1018 bits of information. Basic biological operation can be used to simultaneously operate 1018 bits of information. Or we can say that 1018 data processors can be executed in parallel.

The rest of the paper is organized as follows. Section 2, introduces DNA model of computation proposed by Adleman and his co-authors in detail. Section 3 we briefly introduce DNA algorithms based on the Adleman-Lipton model for constructing bio-molecular parallel adder with basic logic operations. The complexity of the proposed algorithm is also described. Conclusions are drawn in Section 4.

2. DNA Model of Computation

2.1. Abstract Descriptions of Biological Operations in DNA Model

In the last decade there have been revolutionary advances in the field of biomedical engineering particularly in

recombinant DNA and RNA manipulating. Due to the industrialization of the biotechnology field, laboratory techniques for recombinant DNA and RNA manipulation are becoming highly standardized. Basic principles about recombinant DNA can be found in [2]-[6]. In this subsection we describe eight biological operations that are useful for constructing bio-molecular parallel adder with basic logic operations. The method of generating DNA strains for the input operands of parallel N -bit adder is based on the proposed method in [7]-[8]. Basically it is a filtering model developed by L. Adleman (1996), which is memory-less in the sense that the strings themselves do not change during computation. Adleman extended the memory-less model to a memory-based filtering model by adding the so-called flip operation, which allows the implementation of n -bit registers used in this research.

A (test) tube is a set of molecules of DNA (a multi-set of finite strings over the alphabet $\{A, C, G, T\}$). Given a tube, one can perform the following operations:

1. *Extract*: Given a tube T and a short single strand of DNA, " s ", produce two tubes $+(T, s)$ and $-(T, s)$, where $+(T, s)$ is all of the molecules of DNA in T which contain the strand " s " as a sub-strand and $-(T, s)$ is all of the molecules of DNA in T which do not contain the short strand " s ".

2. *Merge*: Given tubes T_1 and T_2 , yield $\cup(T_1, T_2)$, where $\cup(T_1, T_2) = T_1 \cup T_2$. This operation is to pour two tubes into one, with no change of the individual strands.

3. *Amplify*: Given a tube T , the operation, Amplify (T, T_1, T_2) , will produce two new tubes T_1 and T_2 so that T_1 and T_2 totally a copy of T (T_1 and T_2 are identical) and T becomes an empty tube.

4. *Append*: Given a tube T and a short strand of DNA, " s ", and the operation will append the short strand, " s ", onto the end of every strand in the tube T . It is denoted by append (T, s) .

5. *Append-head*: Given a tube T and a short strand of DNA, " s ", and the operation will append the short strand, " s ", onto the head of every strand in the tube T . It is denoted by append-head (T, s) .

6. *Detect*: Given a tube T , say 'yes' if T includes at least one DNA molecule, and say 'no' if it contains none. It is

denoted by detect (T).

7. *Discard*: Given a tube T , the operation will discard the tube T . It is denoted by discard (T).

8. *Read*: Given a tube T , the operation is used to describe a single molecule, which is contained in the tube T . Even if T contains many different molecules each encoding a different set of bases, the operation can give an explicit description of exactly one of them. It is denoted by read (T).

2.2. Implementation of Biological operations in DNA Model

Experimental implementation of biological operations is very important for deciding whether any DNA-based algorithm to dealing with any problem can obtain the required answer(s) or not. The content of the following subsections cited from [6] is used to describe how biological operations used in the proposed algorithm are implemented.

1. Separate (T, s, T_1, T_2)

Affinity purification is used to extract any strands from T containing s . The implementation of the biological operation uses a probe sequence that is complementary to the target sequence being found for. Probes are fixed to a surface, and capture through annealing any strands consisting of the target sequence. Then, captured strands perhaps are separated from the rest of the population by placing them in a separate solution, which is heated to break the bonds between the probes and the target sequence. Hence, the probe applied is the complementary sequence of s . Retained strands are placed in one new tube T_1 and the remainder are placed in another new tube T_2 .

2. Merge ($\{T_i\}$)

The contents of tubes $\{T_i\}$ are simply merged by means of pouring. The number of tubes will generally be low, so this is considered to be a constant-time operation.

3. Discard (T)

The contents of T are discarded, and T is replaced by a new, empty tube.

4. Amplify ($T, \{T_i\}$)

The polymerase chain reaction (PCR) is employed, with its initial input that is tube T . This reaction is applied to massively amplify (possibly small) amounts of DNA strands that begin and end with specific primer sequences. Since every strand in tube T is delimited by these sequences, they are all copied by the reaction. Then, the result of the PCR is divided equally between the specified numbers of tubes (the number of PCR cycles may therefore be adjusted to ensure a constant DNA volume per tube, regardless of the number of tubes).

5. Concatenate (s_1, s_2)

Two strands (labeled S and T in Figure 1) may be concatenated by the following process: create a *linker* strand, which has a sequence that is the complement of S followed by the complement of T . The linker strand is affixed to a surface with a magnetic bead (Figure 1(a)). Then, strand S is added to the solution, and anneals with the

linker strand at the appropriate position (Figure 1(b)). Then, strand T is added to the solution, and this also anneals with the linker strand, at a position immediately adjacent to strand S (Figure 1(c)). Next, the ligase enzyme is added to the solution to seal the “nick” between S and T , forming a single strand which may be freed by heating the solution to break its bonds with the linker strand (Figure 1(d)).

Figure 1. Concatenation process: (a) Linker strand affixed to surface. (b) S anneals to linker strand. (c) T anneals to linker strand, adjacent to S . (d) S and T ligated to form a single strand, which is then freed by heating the solution.

6. Append-head (U, s)

The implementation of the *concatenate* () operation denoted above may easily be applied to append a specific sequence, s , to the head of each strand in a tube U . The sequence s corresponds, in this case, to the strand S defined in Figure 1, and strand T in Figure 1 corresponds to the beginning sequence of every strand in the tube. In this case, only the beginning sequence of every strand anneals to the linker strand. Clearly, then, after a series of *append-head*() operations has been completed on a strand, its sequence will be made up of a number of sequences representing bit-strings.

7. Detect (T)

The tube T is run through a gel electrophoresis process, which is generally used to sort DNA strands on length. Any DNA present in T shows up as a visible band in the gel; if DNA strands of the appropriate length are present, the operation returns *true*. If there are no visible bands corresponding to DNA of the correct length, then the operation returns *false*. The length criterion is applied to ensure that DNA fragments present do not cause a false positive result. If the DNA in the band corresponding to the contents of T is required in a subsequent processing step, the band may be excised from the gel by cutting and then soaked to remove the strands for further use.

3. DNA Algorithms for Constructing Basic Logic Operations

3.1. The Construction of Parallel One-bit XOR Operation

The Exclusive-OR (XOR) operation of a bit for two Boolean variables X and Y generates an output of 1 if both X and Y have different values and 0 if they are the same. The $\oplus$ symbol represents the XOR operation. The four possible combinations for the XOR operation of a bit with two Boolean variables X and Y are as follows: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, and $1 \oplus 1 = 0$.

A truth table is usually used with logic operation to represent all possible combinations of inputs and the corresponding outputs. The truth table for the XOR operation is shown in Table 1.

Assume that two one-bit binary numbers X_k and Y_k , for $1 \leq k \leq n$, are used to represent the first input and the second input for the XOR operation of a bit respectively. Also suppose that $S_{k,m}$, for $1 \leq k \leq n$ and $1 \leq m \leq 2$, is applied to represent the output for the one-bit XOR operation. From [7]-[8], for every bit X_k representing the k th element for $1 \leq k \leq n$, two distinct 15-base value sequences are designed. One represents the value “0” for X_k and the other represents the value “1” for X_k . For the sake of convenience, assume that X_k^1 denotes that the value of X_k is 1 and X_k^0 denotes that the value of X_k is 0. Similarly, suppose that Y_k^1 denotes that the value of Y_k is 1 and Y_k^0 denotes that the value of Y_k is 0. Assume that $S_{k,m}^1$ denotes that the value of $S_{k,m}$ is 1 and $S_{k,m}^0$ denotes that the value of $S_{k,m}$ is 0. The following procedure is proposed to perform the parallel one-bit XOR operation.

Table 1. The truth table for the XOR operation of a bit with two Boolean variables X and Y

Input		output
X	Y	$S \leftarrow X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

Procedure ParalleOneBitXOR(T_0, X_k, Y_k, k, m)

- (1) $T_1 = + (T_0, X_k^1)$ and $T_2 = - (T_0, X_k^1)$
- (2) $T_3 = + (T_1, Y_k^1)$ and $T_4 = - (T_1, Y_k^1)$
- (3) $T_5 = + (T_2, Y_k^1)$ and $T_6 = - (T_2, Y_k^1)$
- (4) If (Detect (T_3)) = “yes” then
 - (4a) Append-head ($T_3, S_{k,m}^0$)
 EndIf
- (5) If (Detect (T_4)) = “yes” then
 - (5a) Append-head ($T_4, S_{k,m}^1$)
 EndIf
- (6) If (Detect (T_5)) = “yes” then
 - (6a) Append-head ($T_5, S_{k,m}^1$)

EndIf

- (7) If (Detect (T_6)) = “yes” then
 - (7a) Append-head ($T_6, S_{k,m}^0$)

EndIf

- (8) $T_0 = \cup (T_3, T_4, T_5, T_6)$

EndProcedure

Lemma 1: The algorithm, ParalleOneBitXOR(T_0, X_k, Y_k, k, m), can be employed to perform the biological parallel one-bit XOR operation.

Proof: The algorithm, ParalleOneBitXOR(T_0, X_k, Y_k, k, m), is implemented by means of the *extract*, *detect*, *append-head* and *merge* operations. Steps from (1) to (3) employ the *extract* operations to yield different tubes consisting of different inputs (T_1 to T_6). This implies, T_1 includes all of the inputs that have $X_k = 1$, T_2 contains all of the inputs that have $X_k = 0$, T_3 includes those inputs that have $X_k = 1$ and $Y_k = 1$, T_4 contains those inputs that have $X_k = 1$ and $Y_k = 0$, T_5 consists of those inputs that have $X_k = 0$ and $Y_k = 1$, and finally, T_6 includes those that have $X_k = 0$ and $Y_k = 0$. After having performed Step (1) through Step (3), four different inputs for the XOR operation of a bit as shown in Table 1 were poured into tubes T_3 through T_6 , respectively.

From Step (4) through (7), those steps are, respectively, applied to examine whether contains any input for tubes T_3 , T_4 , T_5 , and T_6 or not. If any a “yes” is returned from those steps, then the corresponding *append-head* operations will be done. Because tubes T_3 , T_4 , T_5 and T_6 , subsequently, contains the input for the fourth row, the third row, the second row and the first row in Table 1, $S_{k,m}^0$ from Step (4) and Step (7) is appended onto the head of every input in tubes T_3 and T_6 and $S_{k,m}^1$ from Step (5) and Step (6) is, subsequently, appended onto the head of every input in tubes T_4 and T_5 . After performing Step (4) through (7), four different outputs to the XOR operation of a bit as shown in Table 1 are appended into tubes T_3 through T_6 . Finally, the execution of Step (8) uses the *merge* operation to pour tubes T_3 through T_6 into tube T_0 . Tube T_0 contains the result finishing the XOR operation of a bit as shown in Table 1.

From ParalleOneBitXOR(T_0, X_k, Y_k, k, m), it takes three *extract* operations, four *detect* operations, one *merge* operation, one *append-head* operation and four test tubes to perform the function of one-bit XOR operation.

3.2. The Construction of Parallel One-bit AND Operation

The AND operation of a bit for two Boolean variables X and Y generates a result of 1 if both X and Y are 1. However, if either X or Y or both, are zero, then the result is 0. The dot $\cdot$ and $\wedge$ symbol are both used to represent the AND operation. The four possible combinations for the AND operation of a bit with two Boolean variables X and Y are as follows: $0 \wedge 0 = 0$, $0 \wedge 1 = 0$, $1 \wedge 0 = 0$, and $1 \wedge 1 = 1$.

A truth table is usually applied with logic operation to represent all possible combinations of inputs and the

corresponding outputs. The truth table for the AND operation is shown in Table 2.

Table 2. The truth table for the AND operation of a bit with two Boolean variables X and Y .

Input		output
X	Y	$TMP \leftarrow X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

Assume that two one-bit binary numbers X_k and Y_k , for $1 \leq k \leq n$, are employed to represent the first input and the second input for the AND operation of a bit respectively. Also assume that a one-bit binary number $TMP_{k,m}$, for $1 \leq k \leq n$ and $1 \leq m \leq 3$, is applied to represent the output for the AND operation of a bit. From [7]-[8], for every bit X_k representing the k th element for $1 \leq k \leq n$, two distinct 15-base value sequences are designed. One represents the value “0” for X_k and the other represents the value “1” for X_k . For the sake of convenience, suppose that X_k^1 denotes that the value of X_k is 1 and X_k^0 denotes that the value of X_k is 0. Similarly, assume that Y_k^1 denotes that the value of Y_k is 1 and Y_k^0 denotes that the value of Y_k is 0. Suppose that $TMP_{k,m}^1$ denotes that the value of $TMP_{k,m}$ is 1 and $TMP_{k,m}^0$ denotes that the value of $TMP_{k,m}$ is 0. The following procedure is offered to perform the parallel AND operation of a bit.

Procedure ParallelOneBitAND(T_0, X_k, Y_k, k, m)

- (1) $T_1 = + (T_0, X_k^1)$ and $T_2 = - (T_0, X_k^1)$
 - (2) $T_3 = + (T_1, Y_k^1)$ and $T_4 = - (T_1, Y_k^1)$
 - (3) $T_5 = + (T_2, Y_k^1)$ and $T_6 = - (T_2, Y_k^1)$
 - (4) If (Detect (T_3) = “yes”) then
 - (4a) Append-head($T_3, TMP_{k,m}^1$)
 EndIf
 - (5) If (Detect (T_4) = “yes”) then
 - (5a) Append-head($T_4, TMP_{k,m}^0$)
 EndIf
 - (6) If (Detect (T_5) = “yes”) then
 - (6a) Append-head($T_5, TMP_{k,m}^0$)
 EndIf
 - (7) If (Detect (T_6) = “yes”) then
 - (7a) Append-head($T_6, TMP_{k,m}^0$)
 EndIf
 - (8) $T_0 = \cup(T_3, T_4, T_5, T_6)$
- EndProcedure

Lemma 2: The algorithm, ParallelOneBitAND(T_0, X_k, Y_k, k, m), can be used to perform the biological parallel AND operation of a bit.

Proof: The algorithm, ParallelOneBitAND(T_0, X_k, Y_k, k, m), is implemented via the *extract*, *detect*, *append-head* and *merge* operations. Steps from (1) to (3) employ the *extract* operations to form some different test tubes including different inputs (T_1 to T_6). That is, T_1 includes all of the

inputs that have $X_k = 1$, T_2 includes all of the inputs that have $X_k = 0$, T_3 includes those that have $X_k = 1$ and $Y_k = 1$, T_4 includes those that have $X_k = 1$ and $Y_k = 0$, T_5 includes those that have $X_k = 0$ and $Y_k = 1$, and finally, T_6 includes those that have $X_k = 0$ and $Y_k = 0$. Having finished execution of Step (1) through (3), it implies that four different inputs for the AND operation of a bit as shown in Table 2 were poured into tubes T_3 through T_6 , respectively.

Steps from (4) to (7) are, respectively, used to test whether contains any input for tubes T_3 , T_4 , T_5 , and T_6 or not. If any a “yes” is returned for those steps, then the corresponding *append-head* operations will be done.

Next, Steps (4a), (5a), (6a) and (7a) use the *append-head* operations to append $TMP_{k,m}^1$ and $TMP_{k,m}^0$ onto the head of every input in the corresponding tubes. After performing Step (4a) through (7a), four different outputs to the AND operation of a bit as shown in Table 2 are appended into tubes T_3 through T_6 . Finally, the execution of Step (8) applies the *merge* operation to pour tubes T_3 through T_6 into tube T_0 . Tube T_0 contains the result performing the AND operation of a bit as shown in Table 2.

From ParallelOneBitAND(T_0, X_k, Y_k, k, m), it takes three *extract* operations, four *detect* operations, one *merge* operation, one *append-head* operation and four test tubes to perform the function of one-bit AND operation.

3.3. The Construction of Parallel One-bit OR Operation

The OR operation of a bit for two Boolean variables X and Y produces a result of 1 if X or Y , or both, are 1. However, if both X and Y being equal to zero, then the result is 0. A plus sign $+$ (logical sum) or $\vee$ symbol is normally applied to represent OR operation. The four possible combinations for the OR operation of a bit with two Boolean variables X and Y are as follows: $0 \vee 0 = 0$, $0 \vee 1 = 1$, $1 \vee 0 = 1$, and $1 \vee 1 = 1$.

A truth table is usually employed with logic operation to represent all possible combinations of inputs and the corresponding outputs. The truth table for the OR operation is shown in Table 3

Table 3. The truth table for the OR operation of a bit with two Boolean variables X and Y .

Input		output
X	Y	$Z \leftarrow X \vee Y$
0	0	0
0	1	1
1	0	0
1	1	1

Assume that two one-bit binary numbers X_k and Y_k , for $1 \leq k \leq n$, represent the first input and the second input for the OR operation of a bit respectively. Also assume that a one-bit binary number, $X_{k,m}$, for $1 \leq k \leq n$ and for $1 \leq m \leq 2$,

is used to represent the output for the OR operation of a bit. From [7]-[8], for every bit X_k representing the k th element for $1 \leq k \leq n$, two distinct 15-base value sequences are designed. One represents the value “0” for X_k and the other represents the value “1” for X_k . For the sake of convenience, suppose that X_k^1 denotes that the value of X_k is 1 and X_k^0 denotes that the value of X_k is 0. Similarly, assume that Y_k^1 denotes that the value of Y_k is 1 and Y_k^0 denotes that the value of Y_k is 0. Suppose that $Z_{k,m}^1$ denotes that the value of $Z_{k,m}$ is 1 and $Z_{k,m}^0$ denotes that the value of $Z_{k,m}$ is 0. The following procedure is proposed to perform the parallel OR operation of a bit.

Procedure ParallelOneBitOR(T_0, X_k, Y_k, k, m)

(1) $T_1 = + (T_0, X_k^1)$ and $T_2 = - (T_0, X_k^1)$

(2) $T_3 = + (T_1, Y_k^1)$ and $T_4 = - (T_1, Y_k^1)$

(3) $T_5 = + (T_2, Y_k^1)$ and $T_6 = - (T_2, Y_k^1)$

(4) If (Detect (T_3) = “yes”) then

(4a) Append-head ($T_3, Z_{k,m}^1$)

EndIf

(5) If (Detect (T_4) = “yes”) then

(5a) Append-head ($T_4, Z_{k,m}^1$)

EndIf

(6) If (Detect (T_5) = “yes”) then

(6a) Append-head ($T_5, Z_{k,m}^1$)

EndIf

(7) If (Detect (T_6) = “yes”) then

(7a) Append-head ($T_6, Z_{k,m}^0$)

EndIf

(8) $T_0 = \cup(T_3, T_4, T_5, T_6)$

EndProcedure

Lemma 3: The algorithm, ParallelOneBitOR(T_0, X_k, Y_k, k, m), can be used to perform the biological parallel OR operation of a bit.

Proof: The algorithm, ParallelOneBitOR(T_0, X_k, Y_k, k, m), is implemented via the *extract*, *detect*, *append-head* and *merge* operations. Steps from (1) to (3) employ the *extract* operations to form some different test tubes including different inputs (T_1 to T_6). That is, T_1 includes all of the inputs that have $X_k = 1$, T_2 includes all of the inputs that have $X_k = 0$, T_3 includes those that have $X_k = 1$ and $Y_k = 1$, T_4 includes those that have $X_k = 1$ and $Y_k = 0$, T_5 includes those that have $X_k = 0$ and $Y_k = 1$, and finally, T_6 includes those that have $X_k = 0$ and $Y_k = 0$. Having performed Step (1) through (3), this implies that four different inputs for the OR operation of a bit as shown in Table 3 were poured into tubes T_3 through T_6 , respectively.

Steps from (4) to (7) are, respectively, used to test whether contains any input for tubes T_3 , T_4 , T_5 , and T_6 or not. If any a “yes” is returned for those steps, then the corresponding *append-head* operations will be done. Next, Steps (4a), (5a), (6a) and (7a) use the *append-head* operations to append $Z_{k,m}^1$ and $Z_{k,m}^0$ onto the head of every input in the corresponding tubes. After performing Step (4a) through (7a), we can find that four different outputs to the OR operation of a bit as shown in Table 3 are appended into tubes T_3 through T_6 . Finally, the execution of Step (8) applies the *merge* operation to pour tubes T_3 through T_6 into

tube T_0 . Tube T_0 contains the result performing the OR operation of a bit as shown in Table 3.

From ParallelOneBitOR(T_0, X_k, Y_k, k, m), it takes three *extract* operations, four *detect* operations, one *merge* operation, one *append-head* operation and four test tubes to perform the function of one-bit OR operation.

4. DNA Algorithms for Constructing Parallel Adder

4.1. The Construction of a Parallel One-bit Adder

A one-bit adder is a Boolean function that forms the arithmetic sum of three inputs. It includes three inputs and two outputs. Two of the input bits represent augend and addend respectively, denoted by A_k and B_k . The third input represents the carry, denoted by C_k , from the lower significant position. The first output gives the value of the least significant bit of the sum, designated by S_k , for augend, addend and carry. The second output gives the output carry, designated by C_{k+1} , becoming the input carry of the next one-bit adder. From the viewpoint of Boolean circuit implemented with logic gates to form an adder, we will find the Boolean functions specified in the Table 4 are given as follows:

$$S_k = A_k \oplus B_k \oplus C_k$$

and

$$C_{k+1} = (A_k \wedge B_k) \vee (B_k \wedge C_k) \vee (C_k \wedge A_k),$$

Where $1 \leq k \leq n$.

The truth table of the one-bit adder is shown below:

Table 4. The truth table of the one-bit adder

Input	output	
$A_k B_k C_k$	$S_k = A_k \oplus B_k \oplus C_k$	$C_{k+1} = (A_k \wedge B_k) \vee (B_k \wedge C_k) \vee (C_k \wedge A_k)$
0 0 0	0	0
0 0 1	1	0
0 1 0	1	0
0 1 1	0	1
1 0 0	1	0
1 0 1	0	1
1 1 0	0	1
1 1 1	1	1

For the reason of algorithmic design, suppose that three one-bit binary numbers, A_k , B_k and C_k represent the input of a one-bit adder where $1 \leq k \leq n$. Two one-bit binary number, $S_{k,m}$ and C_{k+1} is applied to represent the output of a one-bit adder. From [7]-[8], two *distinct* DNA sequences are designed to represent the value “0” and “1” for every

corresponding bit. For the sake of convenience in our representations, assume that A_k^1 contains the value of A_k to be 1 and A_k^0 contains the value of A_k to be 0. Also suppose that B_k^1 contains the value of B_k to be 1 and B_k^0 contains the value of B_k to be 0. Suppose that C_k^1 contains the value of C_k to be 1 and C_k^0 denotes the value of C_k to be 0. Similarly, suppose that $S_{k,m}^1$ defines the value of $S_{k,m}$ to be 1 and $S_{k,m}^0$ contains the value of $S_{k,m}$ to be 0. $C_{k+1,m}^1$ contains the value of $C_{k+1,m}$ to be 1 and $C_{k+1,m}^0$ denotes the value of $C_{k+1,m}$ to be 0. The following procedure is proposed to perform the Boolean function of a parallel one-bit adder.

Procedure ParallelOneBitAdder(T_0, A_k, B_k, C_k, k, m)

ParallelOneBitXOR($T_0, A_k, B_k, k, 1$)

ParallelOneBitXOR($T_0, S_{k,1}, C_k, k, 2$)

ParallelOneBitAND($T_0, A_k, B_k, k, 1$)

ParallelOneBitAND($T_0, C_k, B_k, k, 2$)

ParallelOneBitAND($T_0, A_k, C_k, k, 3$)

ParallelOneBitOR($T_0, TMP_{k,1}, TMP_{k,2}, k, 1$)

ParallelOneBitOR($T_0, Z_{k,2}, TMP_{k,3}, k, 2$)

$T_1 = + (T_0, Z_{k,2}^1)$ and $T_2 = - (T_0, Z_{k,2}^1)$

(9) If (Detect (T_1)) = "yes" then

(9a) Append-head (T_1, C_{k+1}^1)

EndIf

(10) If (Detect (T_2)) = "yes" then

(10a) Append-head (T_2, C_{k+1}^0)

EndIf

(11) $T_0 = \cup (T_1, T_2)$

EndProcedure

Lemma 4: The algorithm, ParallelOneBitAdder (T_0, A_k, B_k, C_k, k, m), can be applied to perform the Boolean function of a parallel one-bit adder.

Proof: The algorithm, ParallelOneBitAdder(T_0, A_k, B_k, C_k, k, m) is implemented by performing procedure ParallelOneBitXOR(T_0, X_k, Y_k, k, m) and ParallelOneBitOR(T_0, X_k, Y_k, k, m) two times, where $1 \leq m \leq 2$, and ParallelOneBitAND(T_0, X_k, Y_k, k, m) three times, where $1 \leq m \leq 3$. On the execution of Step (1), the algorithm, ParallelOneBitXOR($T_0, A_k, B_k, k, 1$), is employed to perform the function of one-bit XOR operation for input operand A_k and B_k . and tube T_0 contains the result of $A_k \oplus B_k$. Next, the execution of Step (2) performs ParallelOneBitXOR($T_0, S_{k,1}, C_k, k, 2$). Tube T_0 includes the strands encoding the result of $A_k \oplus B_k \oplus C_k$, which is the sum of three input bits A_k, B_k and previous carry C_k . Having performed Step (3) through Step (5), it performs ParallelOneBitAND(T_0, X_k, Y_k, k, m) sequentially and tube T_0 consists of the result of one-bit AND operation, $A_k \wedge B_k, B_k \wedge C_k$, and $C_k \wedge A_k$ in each step respectively. When Step (6) is executed, it performs ParallelOneBitOR($T_0, TMP_{k,1}, TMP_{k,2}, k, 1$). This algorithm performs the function of one-bit OR operation for input operand $TMP_{k,1}$ and $TMP_{k,2}$ and tube T_0 contains the result of $(A_k \wedge B_k) \vee (B_k \wedge C_k)$. When Step (7), ParallelOneBitOR($T_0, Z_{k,1}, TMP_{k,3}, k, 2$), is finished, tube T_0 contains the result of $(A_k \wedge B_k) \vee (B_k \wedge C_k) \vee (C_k \wedge A_k)$. Sequentially we first separate the possible value of carry $Z_{k,2}$ into two test tubes T_1 and T_2 by *extract* operation in Step (8). In Step (9) and Step (10), if the detection result of tube

T_1 or tube T_2 is yes, then we append the value "1" in tube T_1 or "0" in tube T_2 onto the head of corresponding strands at position C_{k+1} . Thus the value of $Z_{k,2}$ is transferred to C_{k+1} by pouring the DNA from tube T_1 and T_2 into T_0 in Step (11).

From ParallelOneBitAdder (T_0, A_k, B_k, C_k, k, m), it takes 22 *extract* operations, 30 *detect* operations, eight *append-head* and *merge* operations and four test tubes to perform the logical operations XOR, AND, and OR for two corresponding input bits. Two output bits of a one-bit adder encode the sum and carry to the addition of three input bits. A value sequence for every output bit contains 15 base pairs. Therefore the length of a DNA strand, encoding two output bits has 30 base pairs, consisting of the concatenation of one value sequence for each output bit.

4.2. The Construction of a Parallel N-bit Adder

4.2.1. Generation of DNA Strands for the Input Operands of Parallel N-bit Adder

Now let's add some binary numbers. Assume that there are three input operands of a parallel N-bit adder. Two of input operands, denoted by A and B , represent augend and addend respectively. The third input operand, C , represents the carry from the previous lower significant position. Suppose that $A_n A_{n-1} \dots A_2 A_1$ is an n -bit binary number, which represents an unsigned integer of augend A . Bit A_n and A_1 are the most significant bit and the least significant bit respectively. The range of the value to unsigned integer of n bits is from 0 to $2^n - 1$. For every bit $A_k, 1 \leq k \leq n$, from [7]-[8], two distinct 15-base value sequences are designed. One represents the value "0" for A_k and the other represents the value "1" for A_k . For sake of convenience in our representation, assume that A_k^1 denotes the value of A_k to be 1 and A_k^0 denotes the value of A_k to be 0. For simplicity, we assume that $B_n B_{n-1} \dots B_2 B_1$ is an n -bit binary number, which represents an unsigned constant integer of addend B . The following procedure is used to generate 2^n possible unsigned integer values A and constant value B .

Procedure Init (T_0, n)

(1) Append (T_1, A_n^1)

(2) Append (T_2, A_n^0)

(3) $T_0 = \cup (T_1, T_2)$

(4) For $k = n - 1$ downto 1

(4a) Amplify (T_0, T_1, T_2)

(4b) Append (T_1, A_k^1)

(4c) Append (T_2, A_k^0)

(4d) $T_0 = \cup (T_1, T_2)$

End For

(5) For $k = n$ downto 1

(5a) Append (T_0, B_k)

EndFor

EndProcedure

For the purpose of demonstrating simply and clearly, we assume that the first input integer number, denoted by A , could be 000, 001, 010, 011, 100, 101, 110, and 111, and the second input integer, denoted by B , is 2 to be added. They are $A = (A_3 A_2 A_1)$ and $B = (B_3 B_2 B_1) = 010$ when

expressed in a 3-bit binary numbers. Also we suppose that the third input C_1 , represented the previous carry, is 0.

When the procedure $\text{Init}(T_0, n)$ is called from Step (1) of Algorithm 1 in Subsection 4.2.2, it is applied to initiate the possible combinations of binary number. Tube T_0 is an empty tube and regarded as an input tube for $\text{Init}(T_0, n)$, and the number of bits encoding an unsigned integer, n , is regarded as the second parameter for $\text{Init}(T_0, n)$. Suppose that the value of n is three. After the execution of Step (1) and Step (2) are performed, tube $T_1 = \{A_3^1\}$ and tube $T_2 = \{A_3^0\}$. Then, after the execution of Step (3) is finished, the *merge* operation makes tube $T_0 = \{A_3^1, A_3^0\}$, tube $T_1 = \emptyset$, and tube $T_2 = \emptyset$.

Because the value of q is three, Step (4a), (4b), (4c) and (4d) will be executed two times. After the first execution of Step (4a) is done, tube $T_0 = \emptyset$, tube $T_1 = \{A_3^1, A_3^0\}$, and tube $T_2 = \{A_3^1, A_3^0\}$. Next, after the execution of Step (4b) and Step (4c) are performed, tube $T_1 = \{A_3^1 A_2^1, A_3^0 A_2^1\}$ and tube $T_2 = \{A_3^1 A_2^0, A_3^0 A_2^0\}$. Then, after the first execution of Step (4d) is finished, the *merge* operation makes tube $T_0 = \{A_3^1 A_2^1, A_3^0 A_2^1, A_3^1 A_2^0, A_3^0 A_2^0\}$, tube $T_1 = \emptyset$ and tube $T_2 = \emptyset$. Finally, after each execution of Step (4) and Step (5) are done, the contents of tube T_0 are shown in Table 5. Lemma 5 is used to demonstrate the correctness of algorithm $\text{Init}(T_0, n)$.

Lemma 5: The algorithm, $\text{Init}(T_0, n)$, is used to generate the DNA strands for 2^n possible unsigned integer values A and constant value B .

Proof: The algorithm, $\text{Init}(T_0, q)$, is implemented via *amplify*, *append* and *merge* operations. Step (1) and Step (2) are subsequently applied to append DNA sequences, respectively, representing the value “1” and “0” for A_n , onto the end of every strand in tube T_1 and tube T_2 . This means that subsets containing the n th element to be 1 appear in tube T_1 and subsets containing the n th element to be 0 appear in tube T_2 . Next, Step (3) is used to pour tube T_1 and T_2 into tube T_0 . This indicates that DNA strands in tube T_0 include DNA sequences of $A_n = 1$ and $A_n = 0$.

Each time Step (4a) is performed, it uses *amplify* operation to copy the contents of tube T_0 into two new tubes, T_1 and T_2 , which are copies of T_0 . Tube T_0 becomes empty. Step (4b) and Step (4c) are used to subsequently append DNA sequences, respectively representing the value “1” and “0” for A_k , onto the end of every strand in tube T_1 and tube T_2 . This implies that strands containing the k th element to be 1 appear in tube T_1 and strands containing the k th element to be 0 appear in tube T_2 . Next, Step (4d) is used to pour tube T_1 and T_2 into tube T_0 . This indicates that DNA strands in tube T_0 include DNA sequences of $A_k = 1$ and $A_k = 0$. After repeating execution of Step (4a) through Step (4d), it finally produces tube T_0 that consists of 2^n DNA sequences representing 2^n different unsigned integer values A . After *append* operation in Step (5a) is also performed n times, the results are binary string $B_n B_{n-1} \dots B_2 B_1$ is appended at the end of every DNA sequence in tube T_0 .

From $\text{Init}(T_0, n)$, it takes $(n - 1)$ *amplify* operations, $3 \times n$ *append* operations, n *merge* operations and three test tubes

to generate the DNA strands for two input operands A and B . A value sequence for every bit contains 15 bases. Therefore, the length of a DNA strand, encoding two unsigned integer value of n bits, is $15 \times (3 \times n)$ bases pair consisting of the concatenation of one value sequence for each bit.

Table 5. The result generated by $\text{Init}(T_0, n)$.

Tube	The result generated by $\text{Init}(T_0, n)$
T_0	$\{A_3^1 A_2^1 A_1^1 B_3^0 B_2^1 B_1^0, A_3^1 A_2^1 A_1^0 B_3^0 B_2^1 B_1^0, A_3^1 A_2^0 A_1^1 B_3^0 B_2^1 B_1^0, A_3^1 A_2^0 A_1^0 B_3^0 B_2^1 B_1^0, A_3^0 A_2^1 A_1^1 B_3^0 B_2^1 B_1^0, A_3^0 A_2^1 A_1^0 B_3^0 B_2^1 B_1^0, A_3^0 A_2^0 A_1^1 B_3^0 B_2^1 B_1^0, A_3^0 A_2^0 A_1^0 B_3^0 B_2^1 B_1^0\}$

4.2.2. DNA Algorithms for Constructing Bio-molecular Parallel Adder with Basic Logic Operations

The parallel one-bit adder introduced in Section 4.1 figures out the arithmetic sum of two bits and a previous carry. Similarly, a binary parallel n -bit adder is also perform the arithmetic sum for the two input operands of n -bit and the input carry by means of performing this one-bit adder n times. The following algorithm is proposed to perform the arithmetic sum for a parallel n -bit adder.

Algorithm 1. ParallelAdder (T_0, A, B, C, n, m)

(1) Init (T_0, n)

(2) Append (T_0, C_1^0)

(3) For $k=1$ to n

(3a) ParallelOneBitAdder (T_0, A_k, B_k, C_k, k, m)

EndFor

EndAlgorithm

When the algorithm ParallelAdder (T_0, A, B, C, n, m) is performed, the execution of Step (1), $\text{Init}(T_0, n)$, in the Algorithm 1 is demonstrated in previous section. The argument A, B and C represent, respectively, the augend, addend, and carry in the parallel n -bit adder. The value of n is three. Tube T_0 with the result shown in Table 5 is regarded as an input tube of this algorithm. Step (2) uses *append* operation to initialize the previous carry to be zero. When Step (3) is executed, it performs Step (3a), the algorithm ParallelOneBitAdder (T_0, A_k, B_k, C_k, k, m) three times to compute the sum of A_k, B_k , and C_k for $1 \leq k \leq 3$. After the loop in Step (3) iterates execution for three times, the result of tube T_0 is shown in Table 6. The boldfaces inside the table represent the sum and carry of corresponding input operands A and B . **Theorem 1** is applied to prove the correctness of algorithm ParallelAdder (T_0, A, B, C, n, m).

Theorem 1: The algorithm, ParallelAdder (T_0, A, B, C, n, m), can be applied to perform the Boolean function for a binary parallel adder of n bits.

Proof: The algorithm, ParallelAdder (T_0, A, B, C, n, m), is implemented via the *extract*, *amplify*, *append*, and *merge* operations. On the execution of Step (1), the algorithm, $\text{Init}(T_0, n)$, is applied to generate the DNA strands for two input operands A and B , represented by binary numbers $A_n A_{n-1} \dots A_2 A_1$ and $B_n B_{n-1} \dots B_2 B_1$ respectively, and Tube T_0 contains the DNA strands corresponding to these two encoded operands. Next the execution of Step (2) is mainly

employed to set the initial value of previous carry C_1 to be zero. When the first execution of Step (2), it uses the *append* operation to append 15-base DNA sequences representing C_1^0 onto the end of strand in T_0 . This is to say that the first bit of the input carry C is set to zero. Thus tube T_0 contains the strands encoding two operands A and B with the zero previous carry C_1

Table 6. The results generated by ParallelAdder (T_0, A, B, C, n, m) in Algorithm 1.

Tube	The result generated by ParallelAdder (T_0, A, B, C, n, m) in Algorithm 1
T_0	$\{C_4^1 Z_{3,2}^1 Z_{3,1}^1 TMP_{3,3}^1 TMP_{3,2}^0 TMP_{3,1}^0 S_{3,2}^0 S_{3,1}^1 C_3^1$ $Z_{2,2}^1 Z_{2,1}^1 TMP_{2,3}^0 TMP_{2,2}^0 TMP_{2,1}^1 S_{2,2}^0 S_{2,1}^0 C_2^0 Z_{1,2}^0$ $Z_{1,1}^0 TMP_{1,3}^0 TMP_{1,2}^0 TMP_{1,1}^0 S_{1,2}^1 S_{1,1}^1 A_3^1 A_2^1 A_1^1 B_3^0$ $B_2^1 B_1^0 C_1^0,$ $C_4^1 Z_{3,2}^1 Z_{3,1}^0 TMP_{3,3}^1 TMP_{3,2}^0 TMP_{3,1}^0 S_{3,2}^0 S_{3,1}^1 C_3^1$ $Z_{2,2}^1 Z_{2,1}^1 TMP_{2,3}^0 TMP_{2,2}^0 TMP_{2,1}^1 S_{2,2}^0 S_{2,1}^1 C_2^0 Z_{1,2}^0$ $Z_{1,1}^0 TMP_{1,3}^0 TMP_{1,2}^0 TMP_{1,1}^0 S_{1,2}^1 S_{1,1}^1 A_3^1 A_2^1 A_1^1 B_3^0$ $B_2^1 B_1^0 C_1^0,$ $C_4^0 Z_{3,2}^0 Z_{3,1}^0 TMP_{3,3}^0 TMP_{3,2}^0 TMP_{3,1}^0 S_{3,2}^1 S_{3,1}^1 C_3^0$ $Z_{2,2}^0 Z_{2,1}^0 TMP_{2,3}^0 TMP_{2,2}^0 TMP_{2,1}^0 S_{2,2}^1 S_{2,1}^1 C_2^0 Z_{1,2}^0$ $Z_{1,1}^0 TMP_{1,3}^0 TMP_{1,2}^0 TMP_{1,1}^0 S_{1,2}^1 S_{1,1}^1 A_3^0 A_2^0 A_1^0 B_3^0$ $B_2^1 B_1^0 C_1^0,$ $C_4^0 Z_{3,2}^0 Z_{3,1}^0 TMP_{3,3}^0 TMP_{3,2}^0 TMP_{3,1}^0 S_{3,2}^1 S_{3,1}^1 C_3^0$ $Z_{2,2}^0 Z_{2,1}^0 TMP_{2,3}^0 TMP_{2,2}^0 TMP_{2,1}^0 S_{2,2}^1 S_{2,1}^1 C_2^0 Z_{1,2}^0$ $Z_{1,1}^0 TMP_{1,3}^0 TMP_{1,2}^0 TMP_{1,1}^0 S_{1,2}^1 S_{1,1}^1 A_3^0 A_2^0 A_1^0 B_3^0$ $B_2^1 B_1^0 C_1^0,$ $C_4^0 Z_{3,2}^0 Z_{3,1}^0 TMP_{3,3}^0 TMP_{3,2}^0 TMP_{3,1}^0 S_{3,2}^1 S_{3,1}^1 C_3^1$ $Z_{2,2}^1 Z_{2,1}^1 TMP_{2,3}^0 TMP_{2,2}^0 TMP_{2,1}^1 S_{2,2}^0 S_{2,1}^1 C_2^0 Z_{1,2}^0$ $Z_{1,1}^0 TMP_{1,3}^0 TMP_{1,2}^0 TMP_{1,1}^0 S_{1,2}^1 S_{1,1}^1 A_3^1 A_2^1 A_1^1 B_3^0$ $B_2^1 B_1^0 C_1^0,$ $C_4^0 Z_{3,2}^0 Z_{3,1}^0 TMP_{3,3}^0 TMP_{3,2}^0 TMP_{3,1}^0 S_{3,2}^1 S_{3,1}^1 C_3^0$ $Z_{2,2}^0 Z_{2,1}^0 TMP_{2,3}^0 TMP_{2,2}^0 TMP_{2,1}^0 S_{2,2}^1 S_{2,1}^1 C_2^0 Z_{1,2}^0$ $Z_{1,1}^0 TMP_{1,3}^0 TMP_{1,2}^0 TMP_{1,1}^0 S_{1,2}^1 S_{1,1}^1 A_3^0 A_2^0 A_1^0 B_3^0$ $B_2^1 B_1^0 C_1^0,$ $C_4^0 Z_{3,2}^0 Z_{3,1}^0 TMP_{3,3}^0 TMP_{3,2}^0 TMP_{3,1}^0 S_{3,2}^1 S_{3,1}^1 C_3^0$ $Z_{2,2}^0 Z_{2,1}^0 TMP_{2,3}^0 TMP_{2,2}^0 TMP_{2,1}^0 S_{2,2}^1 S_{2,1}^1 C_2^0 Z_{1,2}^0$ $Z_{1,1}^0 TMP_{1,3}^0 TMP_{1,2}^0 TMP_{1,1}^0 S_{1,2}^1 S_{1,1}^1 A_3^0 A_2^0 A_1^0 B_3^0$ $B_2^1 B_1^0 C_1^0\}$

Step (3) is the main loop to perform the function of a parallel n -bit adder. On the first execution of Step (3a), it performs the algorithm, ParallelOneBitAdder(T_0, A_k, B_k, C_k, k, m), and uses three logic operations XOR, AND, and OR to perform the addition of three bits (two significant bits and a previous carry). Repeat execution of Step (3a) until the most significant bits of the augend and addend are processed. The tube T_0 contains the strands performing the arithmetic sum of the two input operands and the zero previous carry. After finishing the execution of loop, tube T_0 includes the strands, encoded as $C_{n+1} S_{n,2} S_{n-1,2} S_{n-2,2} \dots S_{2,2} S_{1,2}$, which represent the sum of two input binary numbers $A_n A_{n-1} \dots A_2 A_1$, $B_n B_{n-1} \dots B_2 B_1$ and carry $C_n C_{n-1} \dots C_2 C_1$.

From ParallelAdder (T_0, A, B, C, n, m), it takes $22 \times n$ *extract* operations, $30 \times n$ *detect* operations, $12 \times n + 1$ *append* operations, $10 \times n$ *merge* operations, $(n-1)$ *amplify* operations and four test tubes to compute the sum of two

operands and the previous carry. An n -bit binary number encodes the size of an operand. One bit encodes the carry of the sum. Therefore, $(n + 1)$ bits correspond to the sum of three inputs. A value sequence for every bit contains 15 bases. Therefore, the total length of DNA strands, encoding the final result of the sum for two operands A, B and the previous carry C_1 , is $15 \times (n + 1)$ base pairs consisting of the concatenation of one value sequence for each bit.

5. Conclusions

There are two important features of DNA, at least in principle, to support the Adleman's experiment for solving the Hamiltonian Path Problem (HPP): Watson-Crick complementarity and massive parallelism. Taking advantages of these features, many papers have presented for designing DNA procedures and algorithms to solve various NP-complete problems [9]-[15]. In this paper, based on biological operations in the Adleman-Lipton model, we propose an algorithm for constructing parallel adder with fundamental biological logic operations. The complexity of algorithm for the number of tubes, the number of DNA library strands, the longest length of DNA library strands, and the number of biological operations are $O(1)$, $O(2n)$, $O(15 \times n)$ and $O(n)$, respectively.

The present algorithm has several advantages lying in its massive parallelism as described below. First, operations needed in the algorithm are performed in a fully automated manner in lab level. It is essential not only for the speedup of computation but also for error-free computation. Second, the proposed procedures are realizable practically, because every procedure can be implemented and every DNA manipulation in the procedure is feasible in their lab. Third, proposed algorithm also shows that arithmetic computation can directly be performed with biological operations. This indicates the Boolean circuits are not needed to perform mathematical operations on a molecular computer. Fourth, the proposed algorithm takes a number of steps being polynomial in the computational complexity.

Although the future of biological computation theory and its potential for physical realization is unclear, we have many evidences for showing the ability of DNA computing to solve many simple problems. The authors hope that this paper to demonstrate that molecular computing is a technology worth pursuing.

Acknowledgements

The author would like to give many thanks to Dr. Amos who is the author of the 6th reference and the 16th reference for proposing valuable information about implementation of biological operations.

I am also deeply grateful to Dr. W.L. Chang for his kind and constructive suggestions to improve this research.

REFERENCES

- [1] R. R. Sinden. DNA Structure and Function. Academic Press, 1994.
- [2] J. Watson, M. Gilman, J. Witkowski, and M. Zoller, Recombinant DNA (2nd edition). Scientific American Books, W. H. Freeman and Co., 1992.
- [3] J. Watson , N. Hoplins, J. Roberts, and et. al. Molecular Biology of the Gene. Benjamin/Cummings Menlo Park CA., 1987.
- [4] G. M. Blackburn, and M. J Gait, Nucleic Acids in Chemistry and Biology. IRL Press, 1990.
- [5] F. Eckstein, Oligonucleotides and Analogues. Oxford University Press. , 1991.
- [6] M. Amos, Theoretical and Experimental DNA Computation. Springer, ISBN 3540657738, 2005.
- [7] R. S. Braich, J. Clifford, W.K. Paul Rothmund, D. Hwang , N. Chelyapov, and L. M. Adleman, Solution of a satisfiability problem on a gel-based DNA computer. Proceedings of the 6th International Conference on DNA Computation in the Springer-Verlag Lecture Notes in Computer Science series, 1-27, 2001.
- [8] L. M. Adleman, R. S Braich, J. Clifford, W.K. Paul Rothmund, D. Hwang, and N. Chelyapov, Solution of a 20-Variable 3-SAT Problem on a DNA Computer. Science, Vol. 296, Issue 5567, 499-502, 19 April, 2002 .
- [9] R. J. Lipton, DNA solution of hard computational problems, Science, 268, 542-545, 1995.
- [10] Q. Quyang, P.D. Kaplan, S. Liu, and S. Libchaber A., DNA solution of the maximal clique problem. Science, 278:446-449, 1997.
- [11] M. Amos, DNA Computation, Ph.D. Thesis, department of computer science, the University of Warwick, 1997.
- [12] W.L. Chang , M. Guo, and M. Ho , Towards solution of the set-splitting problem on gel-based DNA computing, Future Generation Computer Systems, Vol. 20, Issue 5, June 15, 875-885, 2004.
- [13] W. L. Chang, M. Ho, and M. Guo, Molecular solutions for the subset-sum problem on DNA-based supercomputing. BioSystems, Vol. 73, No. 2, 117-130, 2004.
- [14] W.L. Chang, M. Ho, and M. Guo, Fast parallel molecular algorithms for DNA-based computation: factoring integers", IEEE Transactions on Nanobioscience, Vol. 4, No. 2, 149-163, 2005.
- [15] W.L.Chang, Fast Parallel DNA-based Algorithms for Molecular Computation: the Set-Partition Problem. IEEE Transaction on NanoBioscience, Vol. 6, No. 1, March, 2007.
- [16] M. Amos. "DNA Computation", Ph.D. Thesis, department of computer science, the University of Warwick, 1997.