

Automated design and programming of a microfluidic DNA computer

MICHAEL S. LIVSTONE¹, RON WEISS² and LAURA F. LANDWEBER^{1,*}

¹*Department of Ecology and Evolutionary Biology, Princeton University, Princeton, NJ, 08544, USA (* Author for correspondence, E-mail: lfl@princeton.edu);* ²*Department of Electrical Engineering, Princeton University, Princeton, NJ, 08544, USA*

Abstract. Previously, we described ways to implement the functions AND and OR in a DNA computer consisting of microreactors with attached heating elements that control annealing of DNA. Based on these findings, we have devised a similar device that can solve a satisfiability problem in any form. The device occupies linear space and operates in quadratic time, while a previously described competing device is built in quadratic space and operates in quadratic time or greater. Reducing the number of reactors in a DNA computer reduces the loss of DNA through binding to the surfaces of the system.

Key words: algorithm, Boolean expression, DNA computing, microfluidics, NP-complete, satisfiability

1. Introduction

One of the primary goals in the field of DNA computing is to reduce the time and space necessary to perform a computation. A promising approach to this problem is to use automated miniaturized systems that reduce the role of the slow human operator and the amount of material needed to build a computer. We have been interested in using microfluidic (Unger et al., 2000; van Noort et al., 2004) or microelectrophoretic (Zangmeister and Tarlov, 2004) systems where microreactors sort potential answers, represented by DNA strands, based on whether the strands represent solutions to satisfiability (SAT) problems (Lipton, 1995). Previously (Livstone and Landweber, 2003), we described how to implement the functions AND (*) and OR (+) in small systems consisting of microreactors arranged in series where annealing, and therefore retention, of answer strands is controlled by heating elements mounted on the backs of each reactor. In

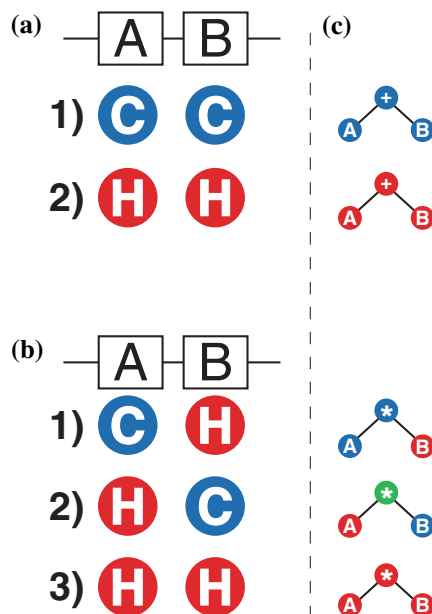


Figure 1. Protocols to solve sample SAT problems, showing hot (“H”) and cold (“C”) heater settings for each round of flow. (a) The expression “A OR B” (“A + B”). The microreactors contain oligonucleotides complementary to sequences representing TRUE values for the variables A and B. The DNA library flows through the reactors, left to right, with the heaters off; strands binding in either reactor must have sequences corresponding to either A or B being true. Bound strands are eluted and recovered by turning both heaters on. (b) Notably, the same apparatus can also implement “A AND B” (“A * B”). The library flows through the reactors with heater B on; binding is prohibited in reactor B, so only those strands which encode A = TRUE are retained. Then, heater A is turned on and heater B off, so that strands bound in reactor A are eluted and flow into reactor B, where they bind only if they encode B = TRUE. Finally, heater B is turned on, eluting the strands which bound in both reactors and satisfy “A * B.” (c) The same expressions and commands portrayed on binary trees. HEAT (red), COOL (blue), and ADVANCE (green) are applied to nodes containing the functions AND (*) or OR (+) and propagated downwards as shown. See text for details.

addition to being simple SAT problems, these two functions are the components from which many more complex functions can be built. Since both can be implemented in series (*Ibid.* and Figure 1), we suggested that more complex problems could also be solved in a device with all reactors arranged in series.

Here, we describe a device capable of solving a SAT problem and a program that automatically designs and programs it. Starting with a

problem, characterized by a Boolean expression in any form, the program generates two types of output: (1) A set of instructions for building the device, consisting of an ordered list of variables to be encoded by oligonucleotides in the reactors, and (2) A set of instructions for operating the device, consisting of instructions to turn the heaters on and off in the proper order. It can be roughly viewed as a compiler for DNA computers.

The device itself is built in linear space, where the number of reactors is equal to the number of literals in the Boolean expression, and operates in quadratic time. In contrast, a previously reported DNA computer (Braich et al., 2002), where correct DNA answer strands are retrieved using a gel electrophoresis apparatus, solves 3-SAT problems in linear time. However, since conversion of a general SAT problem to an equivalent 3-SAT problem generates an expression whose length is at least quadratic with respect to the length of the original expression (Garey and Johnson, 1979), it would require at least quadratic space to build such a device to solve a general SAT problem. We presume that a smaller device will be easier and less expensive to manufacture and, having fewer components, will have higher accuracy.

2. The system

The DNA computer we previously described (Livstone and Landweber, 2003) has two main components: (1) A combinatorially-assembled library of DNA strands representing possible solutions to a Boolean expression (for a detailed description, see (Faulhammer et al., 2000; Braich et al., 2002)), and (2) A microfluidic (or microelectrophoretic – we will use the terms interchangeably) system, consisting of a set of microreactors arranged in series and connected by channels, that is used to separate the library into populations of strands that either do or do not represent correct solutions to the expression. Each reactor contains a large set of identical single-stranded oligonucleotides complementary to a sequence encoding a single Boolean variable (“literal”) and, mounted on the back, a heater (or some other device) that can control annealing. The following section describes the operation of the system, but the reader is referred to (Livstone and Landweber, 2003) for a detailed discussion of its theoretical limitations.

As the DNA library flows through a reactor, those strands containing the sequence whose complement is immobilized in the reactor will be captured (if the heater is off), and the remaining strands will pass through. Strands can also be released from a reactor by turning its heater on. A complete system is operated as follows. First, the library is allowed to flow through the system, and strands bind only in those reactors where the heaters are off; unbound strands flow through and are lost. Flow is stopped, and one or more heaters are turned on or off. Strands are released from those reactors where the heater is on, and the flow is turned back on. This process is repeated until the entire Boolean expression has been implemented and only strands representing correct solutions to the expression are left in the system; they are eluted by turning on all heaters. For examples, see figure 1.

2.1. *Implementing NOT*

There is one additional technical issue that needs to be addressed: how to deal with expressions containing the function NOT (\sim).

All Boolean expressions can be written as the composite of three elementary functions: AND, OR, and NOT. AND and OR are implemented by the sorting systems shown in Figure 1. The expression " $\sim A$ " can be implemented in a single microreactor by retaining those strands with the sequence representing "A is FALSE." In contrast, however, an electronic NOT gate would transform $A = \text{TRUE}$ to $A = \text{FALSE}$. This is not possible using a sorting system, which can only retain a molecule or allow it to pass through, but cannot transform one sequence into another. Therefore, more complicated systems such as " $\sim(A + B)$ " cannot be implemented directly, but must first be transformed algebraically using the DeMorgan Laws $\sim(A * B) = \sim A + \sim B$ and $\sim(A + B) = \sim A * \sim B$ (Kuntzmann, 1967). The DeMorgan Laws must be applied repeatedly until no NOT symbols precede parentheses, but are found only in conjunction with single variables.

(We note as a technical point that either the set {AND, NOT} or the set {OR, NOT} is sufficient *mathematically* to compose all Boolean functions. For example, all OR's can be replaced with AND's because " $A + B$ " can be rewritten as " $\sim(\sim A * \sim B)$." However, the latter expression contains a NOT symbol preceding parentheses, which cannot be implemented in a sorting system. Therefore, we retain all three functions AND, OR, and NOT.)

3. Automated design and programming

We have written a program that reads a SAT problem and uses the DeMorgan Laws to transform it into an equivalent form that can be solved in a sorting system. Next, it designs the DNA computer to solve the problem by making a list of variables that are to be encoded by the oligonucleotides in each reactor. Finally, it generates a list of operating instructions for turning the heaters on and off. These three goals are all accomplished by transforming the expression into a binary tree, which is then processed to parse the expression and generate the design and the instructions. The program requires only linear time and space.

3.1. Reading the expression

A SAT problem is characterized by a Boolean expression that begins as a string of characters and may contain six types of “tokens,” representing AND (*), OR (+), NOT (~), left and right parentheses, and the variables (literals) themselves. A literal may consist of one or more characters – such as “A,” “x₀,” or “first_state” – but no spaces. Certain tokens may not begin or end an expression, and certain tokens may not follow other specific tokens (Table 1). Furthermore, all parentheses must be balanced; i.e. the number of left and right parentheses must be equal, and at no point in the expression may the number of right parentheses exceed the number of left parentheses. The expression is divided into tokens and checked to confirm that it is legal. This converts the expression to an array whose length

Table 1. Permitted (+) or forbidden (–) positions of tokens within a Boolean expression

Token	Begin	Followed by...					End	
		Literal	()	AND	OR		NOT
Literal	+	–	–	+	+	+	–	+
(+	+	+	–	–	–	+	–
)	–	–	–	+	+	+	–	+
AND	–	+	+	–	–	–	+	–
OR	–	+	+	–	–	–	+	–
NOT	+	+	+	–	–	–	+	–

(measured in number of elements) is less than or equal to the length of the expression (measured in characters).

3.2. *Making the tree*

The order of operations for Boolean expressions is: (1) Parentheses, (2) NOT, (3) AND, and (4) OR. The tree is constructed in this order, from the bottom up, as follows.

The array is scanned from the left until the first right parenthesis is found, then backwards from that point until the first left parenthesis, thereby identifying the first complete parenthetical for further processing. If no parentheses exist, the entire expression is processed.

Next, the parenthetical or expression is scanned from the left for NOTs. When one is found, the next token may be either a literal ("A") or another NOT. If it is "A," the slice "(~, A)" of the token array is replaced with a reference (equivalent to a pointer in C) to that slice. The reference acts like a literal in all further iterations of the token array and reduces the size of the token array by one, an important feature of the algorithm which helps to keep it linear. If the NOT is followed by another NOT, both are removed as a double negative. The process is repeated until all NOTs are replaced/removed.

The resultant array is scanned from the left for ANDs. When an AND is found, the two tokens preceding and following it must either be literals or references; the slice consisting of all three tokens is replaced with a reference, and the process is repeated until all ANDs are gone. The same process is then repeated for ORs. The expression now consists of a single reference. If it began as a parenthetical, the surrounding parentheses are removed and the main expression is processed for further parentheticals; otherwise, the process is complete and the entire expression has been converted to a linked list of short arrays, a binary tree.

To this point, the expression has been read seven times, once each to identify tokens, perform a grammar check, and scan for right and left parentheses, NOTs, ANDs, and ORs. Each time the expression either remained the same length or got shorter, so, for an expression initially consisting of n characters, the total number of tokens read is at most $7n$, and the time required is $O(n)$. Each internal node is now either a NOT, AND, or OR, and each external node ("leaf") is a literal. Parentheses have been eliminated, so the total number of nodes

is less than or equal to the length of the token array, and the tree occupies $O(n)$ memory.

3.3. *Pushing down NOTs*

Internal nodes containing NOT symbols cannot be implemented by a sorting system; they are equivalent to a NOT preceding a parenthesis. The NOTs are removed by pushing them down to the leaves of the tree (Figure 2). There are four possible operations: one to eliminate double NOTs, one to combine NOTs with literals, and two to implement the DeMorgan Laws. The operations are applied from the root of the tree and propagated downward until the only NOTs remaining are in conjunction with literals.

The program traverses the tree in order as follows. Starting at the root node, if a node has two daughters, traverse the left subtree completely, then the right. If the node has only one daughter, traverse it. If the node has no daughters, it is a leaf, so record the literal (or negated literal) in the leaf. The leaves are visited in the same order in which they appear in the original Boolean expression, so this procedure generates the list of literals to be encoded in the microreactors, in order. The same order is used to generate the heater instructions below.

The operations to implement the DeMorgan Laws cause a temporary increase in the number of NOT nodes, but they are all removed by the end of the process. Assume that the tree initially has n nodes and ends with n' nodes. Every node is visited only once, so the process takes $O(n)$ time, and $n' \leq n$. The maximum number of nodes that can exist at any given time is less than $2n$, so the process requires $O(n)$ memory.

At this point, every internal node contains a “*” or a “+” and has two daughters, and every leaf contains a literal (or a negated literal, which is equivalent). All other symbols from the original expression have been eliminated.

3.4. *Generating instructions*

Having parsed the tree and generated the design of the DNA computer, the program now generates instructions to operate the heaters

←

Figure 2. Removing incompatible NOTs and generating instructions for building and operating a DNA computer to solve the SAT problem $\sim((A + \sim B) * C) * (D * E)$. (a) The expression is converted to a binary tree, which is processed (left to right) using four operations: applying the DeMorgan laws to AND (red) and OR (orange) subtrees, removing double NOTs (blue) and combining NOTs with literals (green). The resulting tree has NOTs only in conjunction with literals in the leaves of the tree. (b) Generating instructions to build and operate the DNA computer. First (left), the command COOL is applied to the root node and recursively passed down through the tree as described in figure 1. Next (second through fourth panels), the command ADVANCE is applied to each of the *'s in order; only *changes* are passed down or recorded. (c) The resultant commands as they are recorded by the program. Each group of commands is separated by a slash (/). The first group lists the oligos that are to be incorporated into the reactors. The second group shows the initial heater settings, hot (H) or cold (C). The remaining groups show the heaters, numbered 0 through 4, to be toggled on or off during that round of operation of the computer. (d) A schematic of the computer, showing the reactors arranged in series with the oligos and heater settings from (c). Not shown: retained strands are eluted by turning on all heaters.

node, pass COOL to both daughters, (2) If COOL is applied to a “*” node, pass COOL to the left daughter node and HEAT to the right, (3) If any node receives the command HEAT, pass HEAT to both daughters, and (4) Whenever a leaf receives a command, record the command as the instruction for the corresponding heater.

Let n be the number of internal nodes of the tree. During this process, every branch is traversed once during a function call, and every leaf receives one command. Therefore, there are $2n$ function calls and $n+1$ instructions generated, and the process requires $O(n)$ time and memory.

3.4.2. Advance

The purpose of ADVANCE is to move a set of strands downstream from one (or more) reactor(s) in order to implement AND (Figure 1). ADVANCE is applied to each * node in order (as described above) and passes HEAT to the left daughter node (or subtree) and COOL to the right.

The total number of instructions for all heaters during this phase is mn , where m is the number of *'s in the tree (i.e., the number of times ADVANCE is invoked) and n is the number of literals (each with a heater requiring an instruction). Since $m \leq n-1$, this procedure would require $O(n^2)$, or quadratic, memory, but, in order to reduce its complexity, we modified ADVANCE so that only *changes* are recorded.

Specifically, HEAT and COOL are applied to a node only if doing so would change the last command applied to that node. This eliminates both redundant function calls within the tree and duplicate commands passed to leaves, making ADVANCE $O(n)$ instead of $O(n^2)$. Proof is as follows.

Let T be a tree with root R , a leaf L , a path P from R to L , and n internal nodes. Then T has $n+1$ leaves and $2n$ branches, and there are three possible cases:

1. P contains no $*$'s. Then L receives the command COOL during SETUP and does not change during ADVANCE.
2. L lies in the left subtree of every $*$ along P . Then L receives COOL during SETUP, and all subsequent commands to L come from applying ADVANCE to $*$'s on P . Since L lies in the left subtree of every $*$ on P , they are all HEAT.
3. L lies in the right subtree of at least one $*$. Let Q be the $*$ on P closest to L such that L lies in the right subtree of Q . Then L receives HEAT during SETUP, and continues to receive HEAT until Q receives ADVANCE and passes COOL to its right subtree. The right subtree is now a tree with one of the properties listed in case 1 or 2, and it has effectively just received the command SETUP. All further commands passed to L , if any, will be HEAT.

The maximum number of state changes per leaf (two) occurs when L lies in the left subtree of one $*$ and the right subtree of another. Therefore, the total number of state changes for the heaters cannot exceed two per leaf, or $2(n+1)$. Furthermore, the total number of calls of HEAT and COOL is equal to the number of branches traversed during the protocol. Branches are traversed only when new instructions are passed to them, so, by a similar argument, each branch is traversed at most twice, for a maximum of $4n$ calls, plus at most n calls of ADVANCE. Therefore, both the number of state changes and the number of function calls are $O(n)$, so ADVANCE needs only linear memory to store its output and linear time to execute.

3.4.3. Elute

SETUP and ADVANCE cause all incorrect solutions to pass through the microreactor system, leaving only correct solutions bound to the reactors. To elute these strands, turn all heaters on.

4. Discussion

We have devised a microelectrophoretic DNA computer to solve satisfiability problems and have written a program to design and program it automatically. The program, which acts rather like a compiler, operates in linear time and space. The computer itself (see figure 2 for an example) has n reactors, where n is the number of literals in the corresponding Boolean expression, and is therefore built in $O(n)$, or linear, space. It requires $m+2$ electrophoresis steps, where m is the number of *'s in the final binary tree (one step each for SETUP and ELUTE, and m steps for ADVANCE). Each step requires an amount of time proportional to the length of the system, which is $O(n)$ (see above). Therefore, electrophoresis requires $O((m+2)n)$, or $O(n^2)$ (since $m \leq n-1$), time, which is quadratic.

A previously described device (Braich et al., 2002) can solve 3-SAT problems, a subset of SAT problems that is also NP-complete, in linear time, but it is restricted to solving 3-SAT problems. Conversion of an arbitrary SAT problem to the equivalent 3-SAT form requires $O(P(ij))$ terms, where i and j are the number of variables and clauses in the original expression and $P(ij)$ is a polynomial in ij (Garey and Johnson, 1979). Such a polynomial is, by definition, an expression of order $(ij)^k$, where $k \geq 1$, so it would be necessary to build a device with $O(ij)$ reactors – that is, a quadratic number of reactors – or greater. Since every reactor must be traversed, and the amount of time required to traverse the system is proportional to the number of reactors, the Braich system actually requires $\geq O(ij)$ time, i.e. quadratic time, to solve a general SAT problem. In contrast, the system we describe here also requires quadratic time, but only linear space.

One feature that remains constant in both this system and the Braich system is that they both perform the same basic function: sorting through a combinatorially-assembled DNA library that initially represents all possible solutions to a SAT problem. Such a library contains $O(2^n)$, or an exponential, number of strands, where n is the number of literals.

In any complex system, the likelihood of failure increases with the number of components. In a DNA computer, failure may take several forms, including manufacturing defects. Another factor that may affect the efficacy of a DNA computer is inappropriate binding of strands to the walls or materials of the system. Consider the following estimate. Let p represent the fraction of DNA applied to a reactor

that sticks to the walls nonspecifically, say 5%. Then the yield of DNA from that reactor will be reduced to $1 - p$, or 95%. If p is constant for all q reactors of a system, then overall yield will be $(1 - p)^q$, which decreases exponentially if q is $O(n)$, but much more quickly if q is $O(n^2)$, so smaller systems have higher yield. Furthermore, the inappropriately bound material, which is not held in place by base-pairing, is not responsive to the heaters and may therefore be dislodged at any point in the protocol, contaminating the downstream pools of retained strands with strands that do not represent correct solutions to the SAT problem. This effect will be greater in systems with $O(n^2)$ reactors than those with $O(n)$ reactors. Therefore, a smaller device will have higher yield and accuracy than a larger device that solves the same SAT problem.

We have designed a DNA computer that can solve SAT problems in a linear number of reactors. It is predicted to have higher yield and accuracy than a comparable device with a quadratic number of reactors. Furthermore, because it accepts SAT problems in any form, it allows the user to rewrite a problem in an equivalent form that may have fewer terms, thereby reducing the number of reactors needed. This added flexibility should further decrease the error rate and loss by adsorption.

Acknowledgements

This work was supported by NSF awards 9875184 and 0121405 and DARPA award F30602-01-2-0560.

References

- Braich RS, Chelyapov N, Johnson C, Rothmund PW and Adleman L (2002) Solution of a 20-variable 3-SAT problem on a DNA computer. *Science* 296: 499–502
- Faulhammer D, Cukras AR, Lipton RJ and Landweber LF (2000) Molecular computation: RNA solutions to chess problems. *Proceedings of the National Academic of Science (USA)* 97: 1385–1389
- Garey MR and Johnson DS (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York
- Kuntzmann J (1967) *Fundamental Boolean Algebra*. Blackie, London, Glasgow
- Lipton RJ (1995) DNA solution of hard computational problems *Science* 268: 542–545
- Livstone MS and Landweber LF (2003) Mathematical considerations in the design of microreactor-based DNA computers. In: Chen J and Reif J (eds) 9th International

- Workshop on DNA-Based Computers, DNA9, pp. 180–189. Springer-Verlag, Madison, Wisconsin, USA
- Unger MA, Chou HP, Thorsen T, Scherer A and Quake SR (2000) Monolithic microfabricated valves and pumps by multilayer soft lithography. *Science* 288: 113–1166
- van Noort D, Tang Z and Landweber LF (2004) Fully Controllable Microfluidics for Molecular Computers. *J. Association Laboratory Automation (JALA)* 9(5): 285–290
- Zangmeister RA and Tarlov MJ (2004) DNA displacement assay integrated into microfluidic channels. *Analytical Chemistry* 76(13): 3655–3659