Branching in Biological Models of Computation

Bethany Andres-Beck
Smith University

Vera Bereg
University of British Columbia

Stephanie Lee
Columbia University

Michael Lindmark
University of Washington

Wojciech Makowiecki
AGH University of Science and Technology
Jagiellonian University

August 9, 2004

Abstract

In this paper, we will look at DNA computing, in particular at different ways that branching can be done in the sticker model (Roweis et. al. 1996) and the solutions to the NP complete problems. We will refer to control structures that are used in programming languages, looping and "if else" statements, as branching. There are five sections to this paper.

The first section will describe the process of DNA transcription then suggest a way that branching can be implemented into this process through the use of an OR gate. The section will continue to discuss the pros and cons of this idea.

The second section will look at a series of approaches to DNA computing and how branching is implemented or could be implemented in them.

The third section will briefly review state automata then go on to explaining a recent success in DNA computing ─ "The smart drug" experiment, its conclusions and implications for branching in DNA computing. It will look at a possible combination of the "smart" drug and the sticker models. In conclusion, pluses along with the minuses of this combination will be explored.

The fourth section will continue to look at expanding the DNA Sticker Based model of computation to include looping and if - then branching instructions without any outside intervention. Unfortunately in the process of adding these capabilities to the model, it gets complicated and very likely impractical. However the model is still valuable from a theoretical standpoint as an example of the computational power of DNA.

The last section of the paper will give an overview of some basic concepts which are the key to understanding why it is so important to find new ways of building computers, define what it means for a problem to be NP-com(plete), then describe the SAT problem which belongs to NP-com class of problems. Further this section will present the Cook's theorem which proofs SAT problem to be an example of NP-com problem and mention some examples of NP-com problems. The common features between branching and NP-com problems will be pointed out along with a few examples of computations on DNA computer which are presently possible. In the end, the weak points of computing NP-com problems on DNA computers will be briefly discussed. At the end of the paper some common conclusions will be stated.

Introduction

In this paper, we would like to take a look at DNA computing, in particular at different ways that branching can be done in the sticker model (Roweis et. al. 1996) and the solutions to the NP complete problems. We will refer to control structures that are used in programming languages, looping and "if else" statements, as branching. There are five sections to this paper.

There are many ways in which branching can be done. We have listed and described a number of them: transcriptional logic branching, sticker model branching, "smart" drug model branching, CPU branching incorporated into the sticker model, and branching in the solutions to the SAT problems.

Branching can be viewed as anything that supports different path taking, sort of a fork in the road where one can make a decision about which route to follow. Thus any of the control structures can be viewed as branching ones. An OR gate or a FOR loop that many of us who are familiar with programming know well, are control structures and thus have rudimentary branching. Branching exists in the decision trees and a path that a binary algorithm takes. Branching is when a doctor diagnoses a patient and decides on whether s/he is sick and what drug if any to prescribe. Branching is a world in its own. One can say that algorithm analysis and logic heavily relies on branching.

Another no less wonderful and magnificent world of its own is cellular biology and in particular, DNA. DNA is a code that the nature has came up with through trial and error way before any human being was capable of thinking of programming something, or even before *Homo Sapiens* exited.

The sections of this paper take you through all or most other above. We hope that you find this paper an interesting read.

Branching in DNA Transcription

By Stephanie Lee

Introduction

DNA has a lot of unique but simple properties which allow it to be easily programmable and also easy to use in experiments. In this part of the paper some of these properties will be explored and used to expand the capabilities of the physical limits of scientific experimentation by attempting to introduce branching to the process of DNA transcription.

DNA and RNA basics

DNA is structured so that each piece is a sequence of DNA nucleotides, each of which is composed of a deoxyribose sugar, a phosphate, and a nitrogenous base. There are four bases in DNA: adenine (A), thymine (T), guanine (G), and cytosine (C). What allows for such variation in DNA is the length of the strands and the sequence of these nucleotides. The sugar and phosphate are the same in every nucleotide and the sugar attaches to the phosphate in the next nucleotide in such a way that together, the sugars and phosphates of the nucleotides create what is known as a sugar-phosphate backbone. This uniformity allows for DNA to assemble with any sequence of DNA nucleotides to form a strand, and consequently, a DNA sequence is usually noted by the sequence of the bases. One important property of these bases is their specific base pairing – the fact that that each base is only complementary with one of the other bases. Adenine and guanine are both purines, meaning that they both have two organic rings. Thymine and cytosine are both pyrimidines, meaning they each have one organic ring. To keep the width of double stranded DNA consistent, each purine can only bind with a pyrimidine. Two purines would make the width too large and two pyrimidines would make the width too small. In addition, the structure of each base only allows it to form two or three hydrogen bonds. Adenine and theymine can only form two while guanine and cytosine can form three. Therefore, adenine, with two organic rings, can only bind with thymine, which has one organic ring. Similarly, guanine, with two organic rings, can only bind with cytosine, which has one organic ring. The simple but specific structure of DNA allows each base to only have one other base with which it can pair.

RNA has a very similar structure to DNA. One o f the main differences is that its sugar is ribose instead of deoxyrisbose. However, RNA is still composed of sequences of various nucleotides. Another difference between DNA and RNA is in the bases. RNA has the base uracil (U) instead of thymine (T). Conveniently, uracil also binds to adenine, so there is no difference in which bases are compatible.

The Process of DNA Transcription

DNA transcription is a process which results in the synthesis of RNA. There are three stages: initiation, elongation, and termination.

When a strand of DNA is aligned in the 5’ to 3’ direction, the beginning of the sequence, about the first 100 nucleotides on the 5’ end, is called the promoter sequence. When the promoter finds the complementary sequence on another strand of DNA going in the 3' to 5' direction, the complementary nucleotides at each position in the two strands will bind. With DNA transcription, this promoter sequence starts with what is called the "TATA box" because it is usually made entirely, or at least mostly, of thymine (T) and adenine (A) nucleotides, which means that the complement to the promoter, which would be found on the complementary DNA strand, is also made up entirely or mostly of thymine and adenine as well.

Subsequently, certain proteins called transcription factors bind to the promoter, including one which recognizes the "TATA box." After the proper transcription factors are bound to the promoter, RNA polymerase, the key enzyme in DNA transcription, binds as well. Once the RNA polymerase binds, it unwinds the two strands of DNA, and using the one in the 3' to 5' direction as the template strand, gets the RNA nucleotides and base pairs them to the template strand. About 10 to 20 DNA bases are exposed at a time, and the DNA transcription process occurs at a rate of about 60 nucleotides per second.[5]

Thus the expression is turned on, meaning the following DNA is read and expressed as output, in the form of RNA. The specific output is based on the specific sequence of DNA nucleotides. This elongation phase continues as the RNA polymerase continues along the DNA until it reaches the terminator.

When the RNA polymerase encounters the terminator, the termination phase begins. The transcription of the specific DNA sequence – that is the RNA sequence AAUAAA – functions as the actual termination signal, prompting the enzyme to cut the RNA 10-35 nucleotides later. A temporary double helix hairpin loop in the RNA forms and eventually the stress this causes in the RNA polymerase enzyme results in a cut in the RNA strand and the termination of the DNA transcription process.

The rate of DNA transcription is controlled by the concentrations of the transcription factors and RNA polymerase, allowing for easy regulation of the expression of output. The concentration of a specific output is directly related to the concentration of the corresponding transcription factors since they are necessary for the instruction resulting in that output to be read. In addition, oftentimes, the concentration of a specific output is also related to the repression of the output, allowing for self-regulation. Once the proper concentration of the output is reached, there is enough of that instruction's repressor to prevent that output from being promoted, and the instruction is essentially "turned off."

Programming DNA Transcription

The sequence of the nucleotides in the RNA strand resulting from DNA transcription is directly related to the DNA sequence which is read. Thus, the input can easily be programmed to perform a specific output. With modern technology, it is fairly easy to control the sequence of nucleotides in DNA, which therefore determines the sequence in both the complementary strand of DNA and the output strand of RNA.

Definition of Branching

Branching allows for conditional if-else statements to be used. If the condition is true, then one instruction is executed, but if the condition is false, then a different instruction is executed. The program then continues, regardless of which instruction was followed. The pseudo code looks like this:

```
{
        if (condition)
                {output 1};
        else
                {output 2};
}
continuation …
```

The continuation is important because it makes sure that the program does not die after the if-else statement in both cases – the case that the statement is true (the if case) and the case that the statement is not true (the else case).

Implementing Branching in DNA Transcription

Branching can be implemented in DNA transcription using an OR gate,[8] which would allow for the continuation whether or not the condition is found to be true. If the condition is true, a specific transcription factor, *1a*, is present, and will bind to the promoter region of the strands of DNA for which the condition is true. RNA polymerase will then bind to the promoter and the output indicating that the condition is true will subsequently be expressed. But the DNA sequence launching the construction of the transcription factor for the second instruction, *2*, would also be executed. The promoter regions for those strands where the condition is false will have an affinity for a different transcription factor, *1b*. This will result in a different output, indicating that the condition is false, but the same DNA sequence ordering the construction of *2* would also be present. The second instruction will be read if the condition is true and if it is false, allowing for the essential continuation.

Pros and Cons

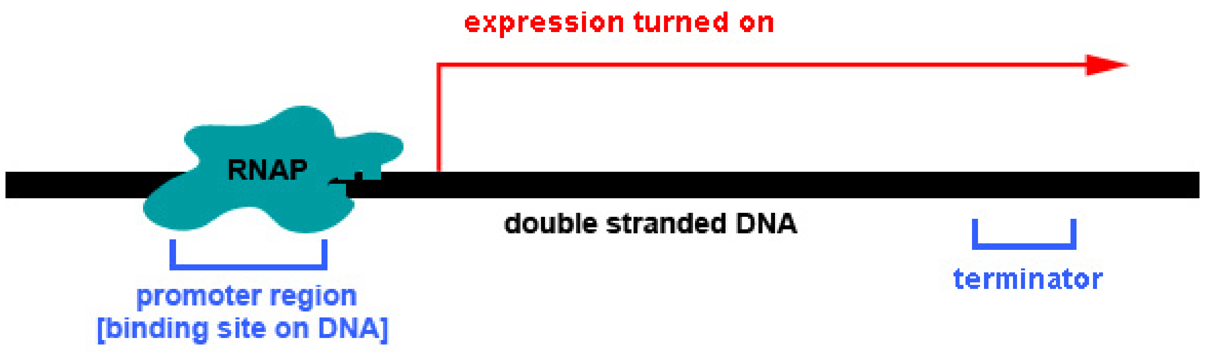
One of the pros of this process is that it is easy to program both the output and the speed. The output is controlled by the DNA sequences, which can easily be customized with today's technology, and the speed can be controlled by the concentrations of the transcription factors and the RNA polymerase. It is also easy to track the progress by taking a sample and approximating the concentrations of each of the outputs. Another important benefit of this process is that it does not require assistance at every step [from a robot], which some DNA experiments such as the sticker model do. Since the basic if-else statement has been explain, the idea can also be expanded to implement if-else if statements as well.

One of the drawbacks of this process is that it does not allow for parallelism. In addition, it is somewhat inefficient and in reality would probably be fairly slow.

Conclusion

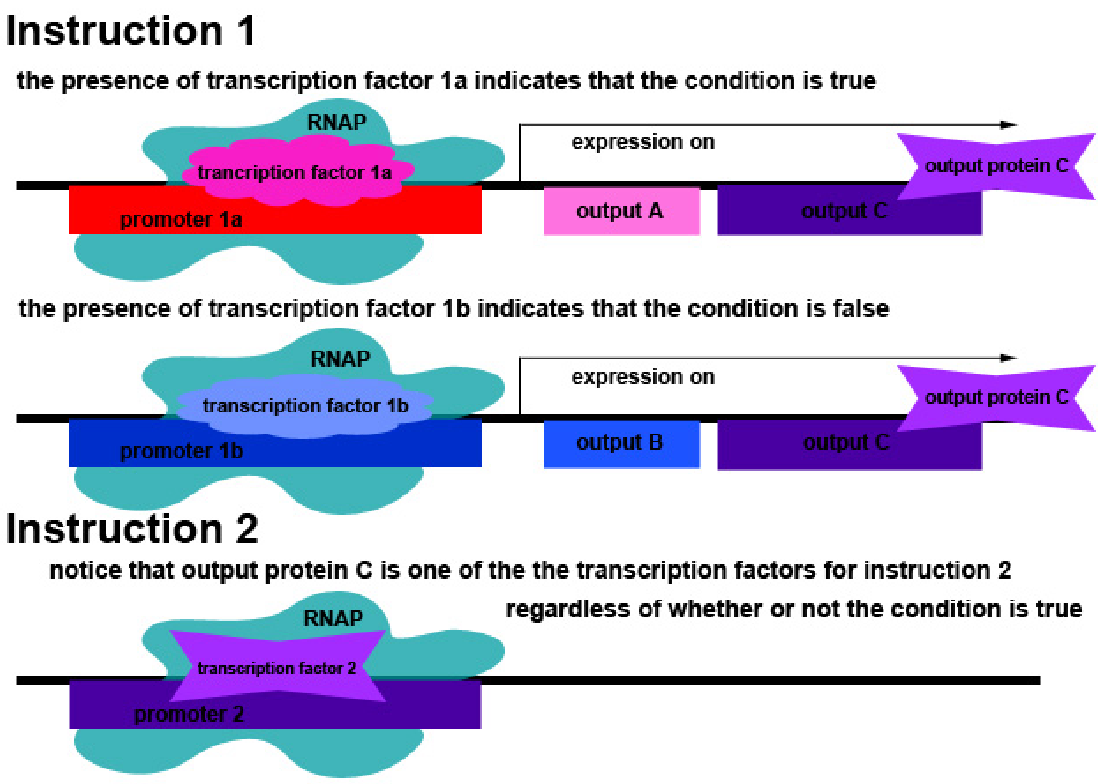
The properties of DNA allow it to be used in a lot of different applications, and this section of the paper has explored a new one of these – branching in the process of DNA transcription. The rest of this paper will explore further into the new possibilities DNA presents, including looking further into the possibility branching with DNA.

Figure 1.



This figure gives a visual representation of the basic setup of DNA transcription. After the proper transcription factors have bound to the promoter, the enzyme RNA polymerase (RNAP) binds to the promoter region of the DNA as well. The expression is then turned on and RNAP proceeds until it reaches the terminator, shortly after which the process terminates.

Figure 2.


Instruction 1
the presence of transcription factor 1a indicates that the condition is true
RNAP
trancription factor 1a
promoter 1a
expression on
output A
output C
output protein C
the presence of transcription factor 1b indicates that the condition is false
RNAP
transcription factor 1b
promoter 1b
expression on
output B
output C
output protein C
Instruction 2
notice that output protein C is one of the the transcription factors for instruction 2
regardless of whether or not the condition is true
RNAP
transcription factor 2
promoter 2

This figure demonstrates the OR gate which allows for the implementation of branching in DNA transcription. If the condition is true for a sample of DNA, then a specific transcription factor will be present, and a specific programmable output will be expressed, as well as the transcription factor for the next instruction, allowing for continuation. If the condition is false, a different transcription factor will be present and a different programmable output will be expressed in addition to the transcription factor for the next instruction. Since the transcription factor for the next instruction is present regardless of whether or not the condition is true, continuation is possible.

Branching using existing instructions in the Sticker model of DNA computation

Bethany Andres-Beck

The sticker model (Roweis et. al. 1996) performs computation by manipulating two types of single stranded DNA: memory strands and complementary stickers. The stickers will adhere to exactly one position on the memory strand, representing one bit of data. (See Figure 1.)

Many memory strands are acted on in parallel, which provides the power of DNA computation. A single vial can have one of four operations performed on it. Set adds a complementary sticker for a bit, setting it to one, clear eliminates a complementary sticker, setting that bit to zero, separate divides the strands into two vials based on a condition, and combine recombines tow vials that had been separated. (See Figure 2.)

Using these existing commands, branching is already possible. We take branch instructions to be in the form "IF…THEN…ELSE", where IF is the condition, THEN is to be performed on those strands where the condition is true and ELSE on those where it is false. To perform a branch operation based on a condition separated the DNA by that condition. On the vial for which the condition is true, perform the THEN instructions. On the other perform the ELSE instructions. Then recombine them. To test for more complex conditions, multiple separates may be performed and the strands that fail combined into an ELSE vial. Since an instruction set can always be empty, this covers all branch instructions.

To perform looping, we need the ability to test for a condition without disrupting the DNA. One way of doing this is to use fluorescent markers. These are single stranded complementary markers which will adhere if the bit they complement to is zero and won't if it is one. After adding them careful measures must be taken to assure that non-adhered fluorescents remain.

These have several advantages. First, they can be detected by a robotic assistant, eliminating the need for human interaction or lengthy testing procedures to establish the presence of strands that fulfill a condition. Second, they are reversible. Clearing the bit they are complementary to will clear them. Third, acceptance conditions are variable, and a level of fluorescents that constitutes a "true" can be set according to need and the level of experimental error. Finally, the robotic assistant can detect different types of fluorescence, allowing for multiple markers to be present at once and enabling nested looping.

We will take loops to be in the form "LOOP WHILE…END LOOP". When it reaches the END LOOP instruction, the robotic assistant performs a separate based of the loop condition, adds a fluorescence marker to the tube that would fulfill that condition. It must be a positive test; that is, it must test for the presences of the marker not the absence, since the absence could also indicate an absence of strands and would lead to operations being performed on an empty vial. This can easily be accomplished in cases where the condition is a positive by performing a clear operation on a bit and adding a marker complementary to that bit. If the WHILE condition is true, the robotic assistant goes through the loop instructions again with that vial. If it is false, it recombines the vials and continues on to the remainder of the instruction set.

This method does have some problems. First, it relies on intervention from outside the model by the robotic assistant. This complicates the model, adds additional room for error and is less desirable than a fully contained model. The second is the variable levels of fluorescence. Experimental error is likely to mean there will always be some fluorescence present, but there may be times we wish to loop if there is a single strand present that fulfils the condition. We sacrifice that ability when we use this method of loops.

Finally, both the branching and the looping described here are slow. They involve separate operations, which are difficult, although not impossible to perform accurately (Roweis et. al. 1996). This is a particular disadvantage of the sticker model, but branching and looping operations increase the complexity and use more of these slower operations. However, this jump in complexity also allows for easier mapping of conventional algorithms on to the massively parallel structure of DNA.

Figure 1

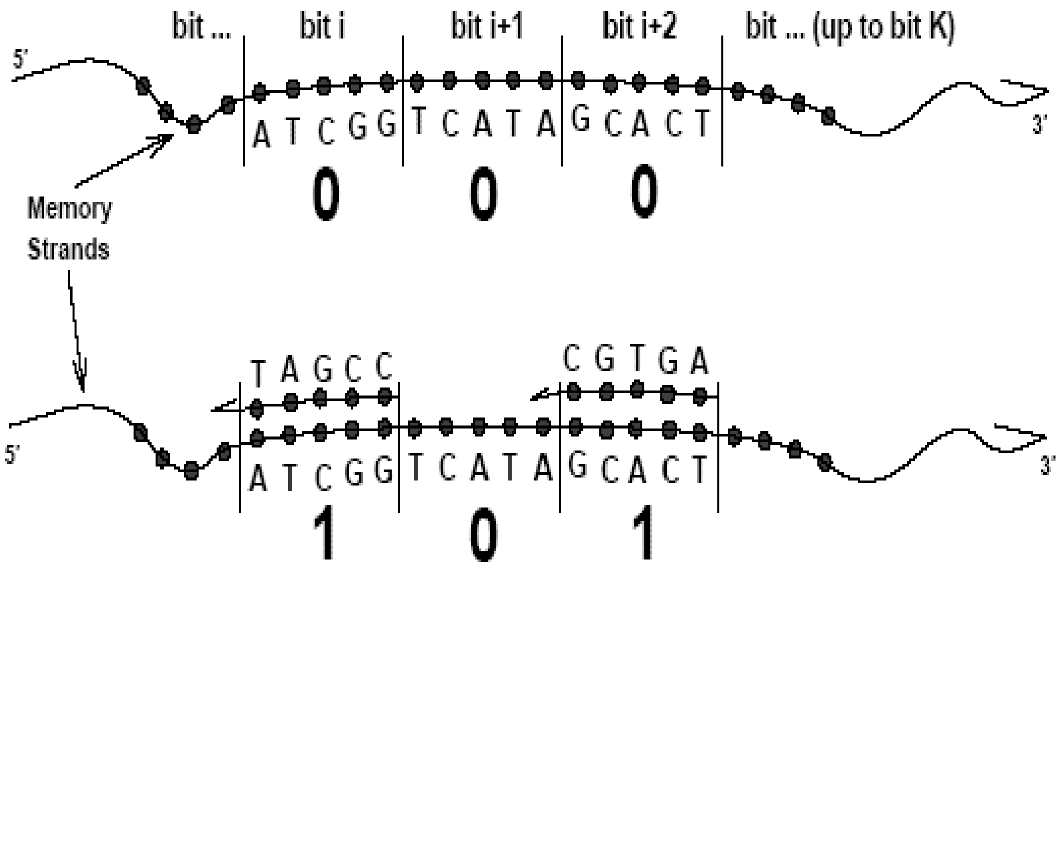
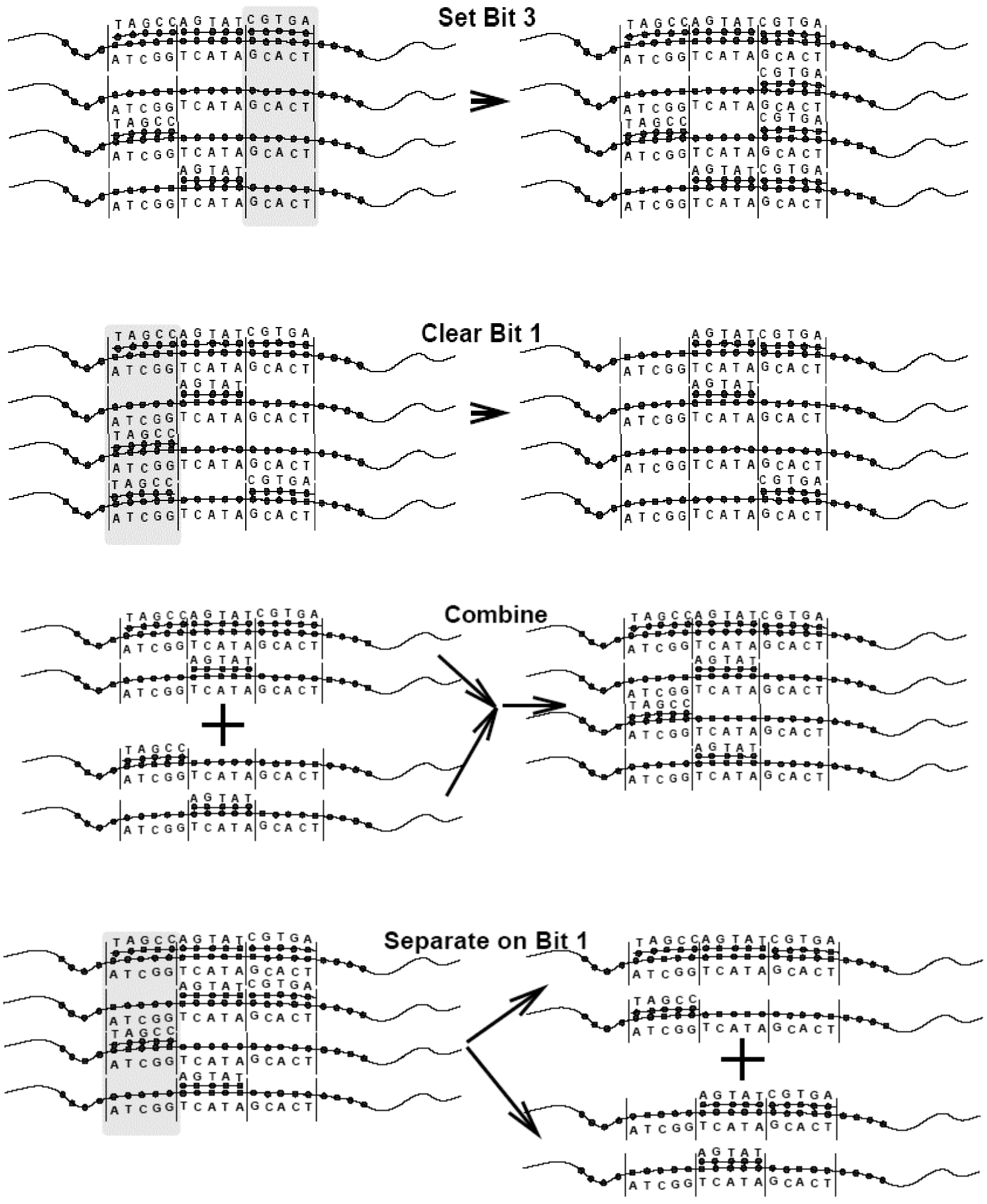


Figure 2



Set Bit 3
TAGCC AGTAT CGTGA
ATCGG TCATA GCACT
Clear Bit 1
Combine
Separate on Bit 1

Incorporating branching from the “Smart” drug[6] model with the Sticker model

by Vera Bespamiatnykh (Bereg)

Introduction

The basic idea of this section is to use the “if else” statements from the “smart” drug model with stickers instead of drugs. This way some of the robotic operations from the sticker model may be eliminated and it might even be possible to consider a model where no robotic intervention is necessary. This is a very simple and yet powerful idea, which in the end might require less than or equal to the original sticker model computation time per a given problem. Thus the objective is to get rid of as many mechanical operations proposed in the sticker model as possible, while trying to ensure that the advantages of the sticker model are preserved.

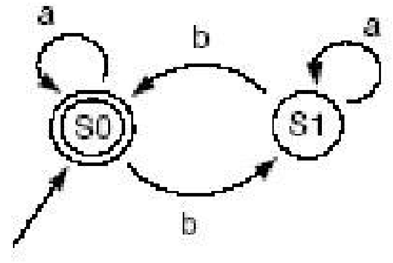
In the heart of the “smart” drug model is the finite state automaton, a concept very familiar to a computer scientist. Automata are devices that convert information from one form into another according to a definite procedure. Thus they have states, one of which is marked as an initial state, some terminating/accepting states and rules that have a starting and a destination states, plus an action that occurs when this rule is being executed.

Example 1:

$$\begin{tikzcd} S0 \arrow[r, "b"] & S1 \end{tikzcd}$$

This example illustrates a rule, which can be interpreted as “when we go from state zero to state one, append letter ‘b’ to our string of characters (which is initially zero)”. Here is a graph representation of an automaton from which this rule could be inferred:

Example 2:



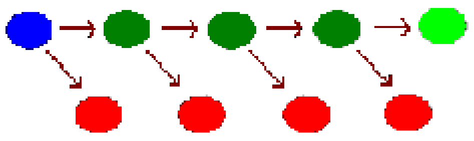
This is a machine that accepts strings over an alphabet {a,b} that are in its language L. Language L, in this case, has a special feature, all its words can contain only a zero or an even number of b’s. At any given moment when machine is running, it can be in one of its finite number of internal states. The computation terminates on processing the last string (input) symbol. An automaton accepts a string if it terminates in an accepting final state.

The programmable and autonomous computing machine that Benenson et all did used biomolecules as its hardware and software[4]. In particular, the hardware consisted of a restriction nuclease and ligase, whereas the software and input were encoded by the double-stranded DNA (ds DNA). The programming part amounted to choosing appropriate software molecules. Once the solutions with these components were mixed the automaton started to processes the input molecule via a sequence of restriction, hybridazation and ligation cycles. In the end, a detectable output molecule that encoded the automaton's final state, and thus the computation result was produced.

This was the concept that Benson and his colleges have tested out in the laboratory with florescent light as output of the computation. However, later on they have thought of an even more interesting idea: "local diagnosis and drug administration", that is the diagnosis of sickness and drug administration (or its suppression) was transferred into the cell. Thus, they were still using an automaton, but its computation was viewed as a series of steps in a diagnosis of a sickness (prostate cancer), while its output was an action of administrating the drug or a drug suppressor.

In the "smart" drug model[3], hardware and software were used just like in the previous automata experiment. First a FolkI cleaves the input molecule and exposes the sticky ends. Then transition molecules (double stranded DNA) try to bind to the input, if they can, then a hybridization occurs and a ligation follows. The FolkI binding and cleaving off yet another piece off the input molecule forms the next configuration. Thus, there is a series of "if-then" statements that gets evaluated. If they all evaluate to "true", then the drug or an anti-drug will be released (depending on what the evaluation is supposed to signify, is the cell cancerous, or not?). Note that it is possible for the evaluation to get stuck at some intermediate step and never reach the release of the drug/anti-drug.

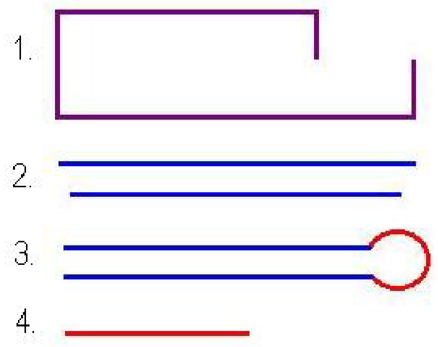
The following is a representation of the states that the final state automata had to go through in the "smart" drug experiment:



Ellipses represent states. The blue is the initial state. The green states represent "true" states, the red states represent the "false" states (evaluation of the "if" statement). The light green state is where the drug or the drug repressor is being released.

The design of this automaton incorporates the idea of molecular Turing machines. Therefore programs that are being run on the conventional computers are in theory computable by the drug model. The "smart" drug model has "if-else" statements and thus branching is possible. Another thing that captures the imagination is that there was no need for any robotic intervention, and the computation was done in parallel. The timing was reasonable and this has already been tested out in the laboratory (as opposed to the sticker model). Unfortunately, for the "smart" drug model it has been implemented in vitro and has not been tried in vivo yet. However this should not affect the sticker model as it was meant to be used in vitro anyway. It is true that the sticker model uses no costly-to-manufacture enzymes, however, in the “drug” model they are only used as hardware and thus are reusable.

Here is how the two models can be combined: we use stickers and anti-stickers (ie those that peel a sticker off the memory strand) instead of the drug generating single stranded molecules. Therefore, in the test tube we would have our memory strand from the sticker model and the final state automata from the drug model with its input, hardware and software. Thus we need to manufacture the appropriate single stranded DNA, which will be used as the input that would fold on itself. It should have its ends complimentary to one another, so that they would be able to bind easily leaving the sticker/anti-sticker in a hairpin loop (input molecule), and put in the “rules” (or software) that we would like to use in this computation.



1. FolkI enzyme that cleaves the DNA molecule exposing the sticky ends (hardware)

2. Transition molecules (software)

3. Input molecule (with a hairpin in red—sticker/anti-sticker)

4. Output (sticker/anti-sticker)

By manipulating the software and the input molecule’s code, I believe it is possible to model the behavior of random setting/unsetting of bits on given memory strands in the sticker model. It could be true that the separation/set bit/clear bit operations could be eliminated all together. This would mean that there would be no need for any intervention and the sticker model would become more feasible and robust.

It is easy to see how an SISD model could be done with the combination of sticker and “smart” drug models. In order to be able to do a MIMD, we need to answer the following question: How do we ensure that each sequence of “if-then” statements operates on its specific memory strand? Or better yet, do we need to ensure this? It is conceivable that we might not need to ensure this at all, since we are trying to model random setting of bits.

In conclusion, it is true that there are other things that one needs to consider before finalizing this model and trying it out in the laboratory. For example: Would the environment that the “smart” drug model requires be suitable for the sticker model? Would the time taken by the robotic assistant to carry out the procedure, plus the time that it takes the chemical changes to take place (ie stickers binding to memory strand, and being peeled off by an anti-sticker) be equivalent to the time it takes to carry out the “if-else” statements in the vile/tube (ie the binding of FolkI, it cleaving off a piece of the code, until the sticker/anti-sticker is released). It is conceivable that the time taken by the original sticker model to carry out the procedure would be larger than that of the combination of the two, as long as the sequence of the “if-then” statements is not too long (as when a lot of binding of the FolkI and it cleaving the code is required it might take a while).

Conclusion

The length of the sequence allowed in the hairpin part of the input to the “smart” drug model, however, could be up to 21 bases. This is the length or a sticker that he sticker model proposed (around 20 nucleotides). Therefore this part seems to fit. Once the other concerns are taken care of, I believe that this branching model could be quite powerful.

Further Expanding the Sticker-Based Model

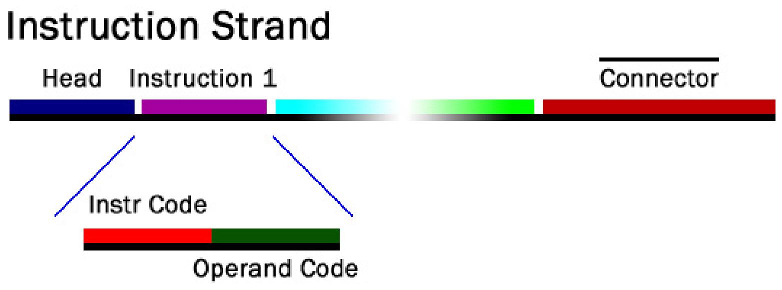
By Michael Lindmark

Introduction

In this section we continue to look at expanding the DNA Sticker Based model of computation to include looping and if - then branching instructions without any outside intervention. Unfortunately in the process of adding these capabilities to the model, it gets complicated and very likely impractical. However the model is still valuable from a theoretical standpoint as an example of the computational power of DNA.

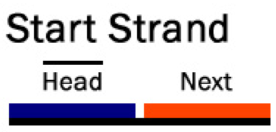
Changes to the standard model

This extension makes a few changes to the strands used in the basic model. To add branching we need to have selective operations, operations that are only applied to certain data. This model accomplishes this by borrowing a few ideas from computers. In effect, it introduces a program that is written in DNA and uses a DNA version of a program counter which keeps track of which instruction the program is currently running. To implement this in DNA we introduce an instruction strand, a couple types of helper strands to run the program counter and a number of solid-bound DNA chambers like those used for the separate operation in the original model.



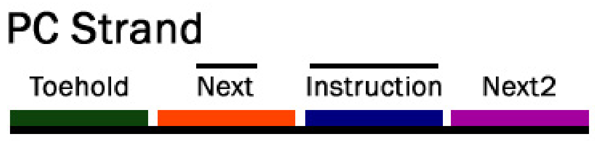
The instruction strands consist of a head region followed by the string of instructions and end with the data connector. The head region provides the start location of the program. Each instruction is divided into two pieces, the instruction code and the operand code. The instruction code specifies which operation is to be performed: *set*, *clear*, *if*, *end-if*, *loop-if-not*, and *exit*. Notice that the separate and combine operations from the original model are no longer part of the operation set. For all operations the operand code specifies on which bit the operation is performed.

The DNA program counter is implemented by four types of strands working together. The first of these is the start strand which enables the program to start by pointing to the first instruction. It is simple, consisting of a sequence complementary to the head marker and the next sequence, which is the pointer signifying that the following instruction should be performed next.



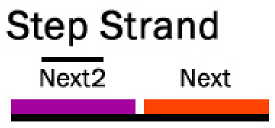
The next type of strand performs the primary function of the counter; incrementing. The pc strands contain a *toehold* region, the complement of the *next* marker, the complement of one instruction, and the *next2* marker. The *toehold* is used by the anti-pc strands, which are complete complements of pc strands, for removing pc strands from the instruction strands.

The *next* marker ensures that only the current instruction is covered by the complement *instruction* region. The *next2* marker signifies that the next instruction is ready, but will not allow another pc strand to bind, ensuring that the pointer is only incremented by one

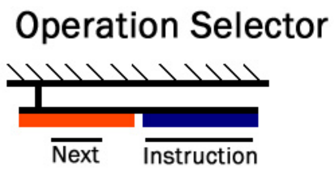


instruction regardless of the number of pc strands around.

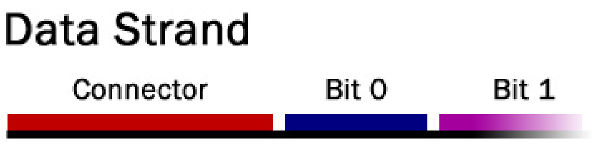
The last type of strand, the step strand, performs the simple operation of converting a *next2* marker to a *next* marker. It is part of the system that prevents more than one instruction from being covered every cycle.



The key part of the operation selectivity that this model requires is achieved by using solid-bound operation selectors. The selectors work exactly like the separation tubes in the original model, except instead of being complements of the bit tags they complement the *next* marker and the *instruction* marker that it is selecting for.



The only change that needs to be made to the data strand is the addition of the



instruction-data connector at one end.

The execution cycle

To run a program on the extended model the instructions are first translated into an instruction strand. This strand is duplicated and then added to a mixture of randomly set data strands. Once connected the start strands are added to initialize the program counter, and then the strands enter the compute cycle. All of the strands pass through a series of operation selector chambers, each of which bind a specific *instruction* marker and the next tag from the pc strands. After all of the strands are selected into different chambers they are sealed away and the selected operation is performed. The series of operation chambers include those for the operations *set bit* 1, 2 ... n, *clear bit* 1, 2 ... n, *if bit* 1, 2 ... n, *loop-if-not bit* 1, 2 … n, and *exit*. After all of the individual operations have been performed the strands are released from the selectors and collected. All of the various pc strands are then added to the collected strands, so that one strand binds to each instruction strand. Then the rest of the pc strands are removed and step strands are added, preparing the instructions for the next cycle. This two step process is required to ensure that the counter can only increment by one instruction every cycle. After removal of the step strands the cycle is repeated. The computation ends when an exit instruction is reached. The strands that are sorted into the exit selector are removed from the system and the answer of the finished computation is ready. One could easily extend the one exit instruction into an exit-true and an exit-false and then only the exit-true strands would need to be read. Once the computation is completed all of the strands can then be sorted, separated, and all strands except the instruction strands can be reused in another computation.

New extension if - then branching

Branching is achieved by slightly altering the design of the *if* operation selectors and directing the selected strands into a separate sub cycle. The *if* selector chambers contain specially weakened selectors that are not strong enough to bind an instruction strand without further help. That help comes from other solid-bound DNA that complements the operand bit. So instruction-data strands in which *if bit one* is the next instruction and bit one is clear are the only strands that bind in the *if bit one* selector. All of these selected strands enter a sub-cycle to skip all instructions until the next *end-if bit one* instruction. This is done by adding an excess of all of the pc and step strands except for the pc strand that corresponds to *end-if one* operation. The strands are then selected by the *end-if* selector chamber to guarantee that the counter has incremented enough. Those strands which fail to bind remain in the sub-cycle while the bound ones are returned to the main cycle. The extra *end-if* operation is required to allow for nested branching by providing different return points based on the if condition.

New extension looping

For looping the model takes advantage of branching help instead of trying to address instructions individually. The *loop-if-not* instructions are selected in the same way *if* selectors work, requiring a clear operand bit in addition to the *next* and *instruction* tags. The selected strands are then cleared of pc strands by adding all of the anti-pc strands. The *toehold* region allows the anti-pc strands to pull off the pc strands and leave the instruction strand with only the start strand bound. This idea of looping could be more intuitively called start-over-if-not. With a little extra branching help traditional looping can be attained. The following pseudo code demonstrates.

```
if (stage1) {
        …
        if (NOT done) {
                loop;
        }
        …
        stage1 = false;
}
```

Complexity

The extensions to the model and the added requirement of no outside intervention while the computation is running force this model to be much more complicated than the Sticker Based Computing Model on which it was based. Assuming that n is the number of data bits the complexity in terms of unique strands is $16n + 7 = O(n)$. The number of chambers as currently set up is $5n + C = O(n)$. A majority of the complexity is in finding the correct design and binding lengths to get energy differences between the various strands correct.

Problems and issues

This model suffers from the same difficulties as the original Sticker Model. There is a tradeoff between the error rates and the compute cycle speed. The time consuming selecting process and lost strands are the major drawbacks. Possible solutions may include using lots of duplicate strands, synthetic DNA backbones or use of metal nano-crystals.

Exploring branching in SAT problems

by Wojciech Makowiecki

Introduction:

The most important question in Computer Science still lacks the answers.
Does P equal NP ? Nobody knows, though most experts in theory of computation suppose it does not. If this is the case and we can not find better algorithms for some extremely crucial problems, we need to find a different approach. The speed of processors doubles twice in 18 months, but it is not enough when we take into consideration NP-complete problems. Even 1000 times faster computers will not let us solve big enough instances of problems that could be useful in practice. As Adleman has shown first in 1994 and then in 2002*(1), it is possible to solve complex problems using a DNA computer.

Definition: NP-complete problem

NP ("non-deterministic polynomial-time") is the set of decision problems solvable in polynomial time on a non-deterministic Turing machine.

NP-com is the complexity class of decision problems for which answers can be checked for correctness, given a certificate by an algorithm whose run time is polynomial in the size of the input (that is, it is NP) and no other NP problem is more than a polynomial factor harder.

A decision problem *C* is NP-complete if it is in NP, and if every other problem in NP is reducible to it.

Description of SAT problem

The **Boolean satisfiability problem (SAT)** is a decision problem.
An instance of the problem is defined by a Boolean expression written using only AND, OR, NOT variables, and parentheses.
The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true?

Example of SAT problem:

$E = (x_1 \text{ or } {\sim}x_2 \text{ or } {\sim}x_3) \text{ and } (x_1 \text{ or } x_2 \text{ or } x_4)$

$x_{1,\dots}$	- variables
~	- indicates “not”
${\sim}x_{2,\dots}$	- negations of variables
or, and	- boolean operators

Cook’s theorem

The language SAT of satisfiable Boolean formulas is NP-complete.

Definition. polynomial-time Turing reduction (Cook reduction)

Formally:

A polynomial-time Turing reduction or Cook reduction of a decision problem *L* to a decision problem *M* is an oracle machine that has an oracle for *M* and can decide *L* in polynomial time.

More intuitively:

If such a reduction exists, than every algorithm for *M* immediately yields an algorithm for *L*, with only a modest (i.e. polynomial) slow-down.

Some important examples of NP-com problems:

- Boolean satisfiability problem (SAT)
- Traveling salesman problem (TSP)

- Hamiltonian cycle problem
- Subgraph isomorphism problem
- Vertex cover problem
- Independent Set problem
- Fifteen puzzle

Branching and NP-com problems

It has not been proved yet but is most probable that no general efficient solution exists for any NP-complete problem.

Consider the SAT problem. There is no known algorithm that is faster than the exponential one. Each boolean variable can be assigned either “true” or “false”. If we have 1 variable we have only 2 possibilities. When we have 20 variables number of possible assignments grows exponentially to 2^{20}=1,048,576. One can check whether the first variable is “true” or “flase” and in each case an appropriate action is done. So it reflects the “if condition”.

In classical approach the computer checks every possible combination one by one. The idea behind DNA computer is different. It works in parallel.

It does not try every single possibility but tries criteria one by one, eliminating all false solutions that do not satisfy the criteria. It starts by the first criterion, deletes all solutions which do not suit the condition, then checks solutions whether or not they satisfy the next criterion and if they do not it deletes them as well, and so on until all criteria are checked.

It is so powerful because every “if” acts on several variables at the same time.

We can imagine every variable in the problem as a node in the tree and two values can take (either “true” or “false”) as branches. The Classical computation would rely on searching the tree using certain algorithm (like depth-first search). In contrast DNA computation will depend on checking few nodes and cutting out the appropriate branches, not checking nodes one by one.

At the end of the procedure, only the solution(s) will survive. The last thing one needs to do is to read off the solution.

Few different models that are being used are as follows:

- Slightly modified Sticker model [2]
- Surface based approach to DNA computation [11]
- Fluorescent DNA computing (“molecular beacon”) [14]

Conclusions

Even though DNA computers heavily utilize parallelism, they cannot solve any instances of NP-complete problem. They do not seem to be scalable enough to solve problems of practical importance. A theoretical forecast expanse for solving 50 variable SAT problem is high, as producing DNA sequences needed for this approach is expensive. In addition, it may be hard to design enough unique DNA strands, to encode all solutions that will not interfere with each other. The other important issue is the problem with data correction as error rate might get higher with attempts to solve larger problems. There exist however some strategies for handling error correction [7,9]. As many researchers including the inventor of DNA computing professor Len Adleman believe the future of DNA computing might be something else other than SAT problem solving, it is still interesting and very young field of study.

Conclusion

In this paper, we have tried to list and describe different ways that branching in DNA computing can be thought of and realized as. There are limitations and barriers that each one of the described here models has, however, they all have their advantages, benefits and purposes.

In conclusion, we would like to thank you for sitting down and taking a minute to read this. We would also like to thank California Institute of Technology, and everyone who organized and participated in the Computing Beyond Silicon Valley Summer School for the wonderful opportunity to learn and to expand our knowledge far beyond our reach.

References

1. Adleman, Leonard. "Molecular computation of solutions to combinatorial problems." Science. 226 (1994): 1021--1024.
2. Adelman, Leonard, et al. “Solutions of 20-Variable 3-SAT Problem on a DNA Computer.” Science. 296(2002): 499--502.
3. Benenson, Yaakov, et al. “An autonomous molecular computer for logical control of gene expression.” Nature. 429(2004): 423—428.
4. Benenson, Yaakov, et al. “Programmable and autonomous computing machine made of biomolecules.” Nature. 414(2001): 430—434.
5. Campbell, Neil A., Jane B. Reece, and Lawrence G. Mitchell. Biology, 5th edition. Menlo Park: Benjamin Cummings, 1999.
6. Condon, Anne. “Automata make antisense.” Nature. News and Views 2004, 429.6990(2004): 351—352.
7. Karp, R, et al. “Error-resilient DNA computation”. Random Structure Algorithms. 15(1999): 450-466.
8. Lauria, Mario, Kaustubh Bhalerao, Muthu M. Pugalanthiran, and Bo Yuan. "Building blocks of a biochemical CPU based on DNA transcription logic." 3rd Workshop on Non-Silicon Computation (NSC-3), Munich, June 2004.
9. Lipton, C. et al. “DNA Based Computers II” DIMACS. 44(1999): 163—170.
10. Lipton, R. “Using DNA to Solve NP-Complete Problems.” Science. 268(1995): 542--545.
11. Liu, Q, et al. “DNA computing on surfaces.” Nature. 403(2000): 175.
12. Lloyd et al. “DNA computing on a chip.” Nature. 403(2000): 143--144.
13. Roweis, Sam, et al. “A Sticker Model for DNA Computation.” Journal of Computational Biology 5.4 (1998): 615--629
14. Tan, W, et al. “Molecular beacons for DNA biosensors with micrometer to submicrometer dimensions.” Analitical Biochem. 283.1(2000):56--63.