Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Parallel Computing with DNA

Semesterproject

Jan Schulze

schulzej@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Sebastian Brandt, Jochen Seidel
Prof. Dr. Roger Wattenhofer

19.06.2015

Abstract

In this thesis we present a DNA strand displacement binary division algorithm for a dividend a and a divisor b based on the Domain Replacement Model introduced in [1] that uses $\mathcal{O}(log(a) \cdot log(b) - log(b)^2))$ rules. Further we introduce two adapted divisibility algorithms. In the first one we reduce the number of steps for the calculation in exchange to using more rules and in the second one we use another representation of the binary numbers so that the algorithm uses $\mathcal{O}(b^2)$ rules independent of the size of the dividend.

We discuss two algorithms to solve the 3SAT problem with l clauses and a variables. The algorithms uses $\mathcal{O}(a^l)$ respectively $\mathcal{O}(l!)$ rules.

For pattern matching of large strings we present an algorithm to estimate the occurrences of patterns in a string that uses a constant amount of rules. At last we introduce a new model that improves the Domain Replacement Model. With this new model we are able to design algorithms to create palindromes and strands of $a^n b^n$ while using a constant amount of rules.

Contents

Chapter 1

Introduction

We present a binary division algorithm based on the long division. To analyse the algorithms, we use the rule complexity, which state how many rule strands an algorithm uses, and the speed complexity, which states how many steps an algorithm needs to give back a positive result. The Long Division Algorithm has a rule complexity of $\mathcal{O}(m \cdot n - n^2))$ for a $m+1$ bit long dividend a and a $n+1$ bit long divisor b and a speed complexity of $\mathcal{O}(m-n) \cdot n)$.

We adapt the Long Division Algorithm to compute more steps simultaneously to reduce the speed complexity. With the Parallel Long Division Algorithm we get a speed complexity of $\mathcal{O}(\frac{m-n}{k} \cdot n)$ with a speed up factor $k \leq \frac{m}{n}$. The speed up is achieved at the cost of rule complexity which is with $\mathcal{O}(2^{\frac{m-n}{k}} k^{\frac{m-n}{k}} \cdot n)$ larger than in the basic Long Division Algorithm.

We then introduce a new way to represent a binary number with a single DNA strand. We present an algorithm that checks divisibility with the new representation. The algorithm that uses this representation reaches a rule complexity of $\mathcal{O}(b^2)$ that is independent of the dividend. The speed complexity is $\mathcal{O}(log(a) - log(b))$.

In the next part we suggest two algorithms to solve the 3-Satisfiability problem with rule complexities of $\mathcal{O}(a^l)$ and $\mathcal{O}(l!)$ with l being the number of clauses and a being the number of variables.

In the third part of this thesis we discuss some basic pattern matching algorithm with the Domain Replacement Model and present how to estimate number of occurrences of a pattern in large DNA strands.

At last we introduce a new model that improves the Domain Replacement Model. With this new model we are able to design algorithms to create palindromes and strands of $a^n b^n$ with constant rule complexity.

1.1 Related Work

The Domain Replacement Model which we mainly use in this work to design the algorithms, was introduced in [1]. It uses strand displacement rules for domain replacement. Further it uses the more powerful collapsing and composition rules, which allow the concatenation of strands and the replacement of several strands by another. In the last part of this thesis we extend the Domain Replacement Model to be able to design more powerful algorithms.

In [2] a division algorithm for DNA computing is introduced. It bases on the Newton Method whereas we build up our algorithm from the long division. For dividing a m bit numbers it needs $\mathcal{O}(log(m))$ steps and uses $\mathcal{O}(m^2)$ DNA strands.

In [3] the satisfiability problem was solved for 20 variables and 24 clauses, but using a gel based DNA computer in which the DNA molecules are moved through different modules whereas we use in the Domain Replacement Model only DNA strands in a single soup.

[4] and [5] give a deep insight and analysis in the aspects of branch migration and hairpin-loops which has only been a side note in this thesis and is not necessary to understand this thesis. The important aspects of those papers are the following which inspired the newly introduced Insertion Model. Branch migration is the process in which a bound DNA strand is replaced by another. Hairpin-loops are loops of DNA that arise if some domains of a DNA strand bind to other domains of itself.

Chapter 2

Model Section

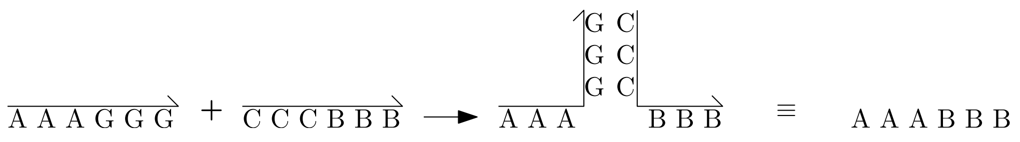
In this thesis we use the Domain Replacement DNA model introduced in [1] which we summarize in the following. A DNA strand consists of a sequence of the nucleotides Adenine (A), Guanine (G), Cytosine (C) and Thymine (T). The extremities of a strand are different, one is called 3' end and the other 5' end. The chemical and biological aspects of the extremities are not further important for this model, except that they give the strands a direction from the 5' towards the 3' end. The direction is represented by an arrow over each strand.

$$\overrightarrow{\mathrm{A\ C\ G\ A}}$$

Nucleotides can bind together if they are Watson-Crick complementary. A and T as well as G and C are WK complementary. Complementary sequences are also WK complementary if they have opposite directions

$$\begin{array}{c}\overrightarrow{\mathrm{A\ A\ G\ C\ T\ A\ C}}\\ \underleftarrow{\mathrm{T\ T\ C\ G\ A\ T\ G}}\end{array}$$

It is not necessary that whole strands bind together. Also parts of a strand can bind which is called domain replacement.



In the example above the GGG part of the the first strand is replaced by the BBB of the second strand. The resulting strand has the same properties of the strand $AAABBB$. Therefore $AAABBB$ is called the effective effective strand of the strand above.

To implement a computation with DNA, we mix the DNA strands together in a soup, e.g. a solution. The strands then react with each other and we can observe the resulting strands.

Each strand is in the form of $+X*$, where $+,*$ are domains used as delimiters that are not part of the alphabet $\sum$ of X. We assume that domain replacements can only occur if a delimiter is part of this replacement.

The Domain Replacement Model consists of three strand types: Input strands, rule strands and output strands. Input strands are the strands that carry the information of the input. They are represented underlined, e.g. $\underline{+X*}$. Rule strands do not depend on the input and implement the steps of the algorithm itself. Note that because domain replacements can only take place if a delimiter is involved, a delimiter has to be part of the rule strands, e.g. the rule $A* \rightarrow B*$, which is another way of writing the strand $\bar{*}\bar{A}B*$. Output rules take complete strands and replace them by a strand, e.g. the rule $+A* \rightarrow +B*$ which is implemented by the strand $\overline{+A*}+B*$.

The alphabet Σ for the domains has to hold the following conditions, (taken from [1], page 7)

$$\begin{aligned} &1.\ a \in \Sigma \Leftrightarrow \bar{a} \in \bar{\Sigma} \\ &2. \qquad +, * \notin \Sigma\bar{\Sigma} \\ &3. \qquad \Sigma \cap \bar{\Sigma} = \emptyset \end{aligned}$$

Another rule in the Domain Replacement Model is the collapsing rule. It merges two strands e.g. $+AX*$ and $+YB*$ by using rules like $\overline{X*}\ \overline{+Y}$. This collapsing rule can be also written as $X * \bowtie +Y$.

We can use the collapsing rule with an output rule to create a composition rule in the form of $+A* + +B* \rightarrow +C*$ by collapsing the strands $+A*$ and $+B*$ with the collapsing rule strand $\overline{A*}AB\overline{+B}$. Then we use the output rule $+AB* \rightarrow +C*$.

Chapter 3

Binary Division

In this chapter we demonstrate how to use DNA in the Domain Replacement Model to divide two binary numbers and to check divisibility. In [2] a division algorithm is proposed that bases on Newton's Method. For dividing a m bit numbers it needs $\mathcal{O}(log(m))$ steps and uses $\mathcal{O}(m^2)$ DNA strands.

We present an algorithm based on the long division and analyse its performance. To measure the performance of an algorithm we use two main criteria . The speed complexity states how many steps an algorithm needs to return a result. The rule complexity states how many different rule strands are necessary for an algorithm to work. Based on the results of the analysis we present an algorithm with an improved Speed Complexity and an algorithm that uses a different representation of the binary numbers.

Let a be a $m+1$ bit number and b a $n+1$ bit number, both in two's complement and positive. The numbers are in two's complement because it is possible that they become negative during the algorithm.

Every bit is represented by a single strand (like in [1]) and such a bit strand looks like this: $+0*$ or $+1*$ where $+$ is the starting, $*$ is the ending delimiter. For every different number and every different bit of the number we use a unique delimiter to be able to distinguish the bits. Each delimiter has two labels. One superscript to allocate a bit to a number and one subscript to determine the position of the bit in the number. The delimiter of the n^{th} bit of number a e.g. looks like $*^a_n$.

3.1 Long Division Algorithm

This division algorithm is a straightforward adaptation of the long division to the binary DNA model. The algorithm can divide a number a by a number b if a has at most $m+1$ bits and b at most $n+1$ bits. The result is the number r. There is only one difference of the algorithm to the long division. The difference is that in the long division a subtraction only takes place if the result of the subtraction is a positive number, whereas in our algorithm the subtraction takes place every time and afterwards the algorithm selects the right values to continue with. If the result of the subtraction is positive it continues the calculation with the result of the subtraction, if it is negative it continues with the values before.

The algorithm consists of three basic steps that are repeated until the algorithm finishes. First we subtract the divisor b bitwise from the first $n+1$ bits of a. Then we check if the resulting number is positive. If the resulting number is positive, the algorithm continues with the bit values of the result of the subtraction, if the resulting number is negative, the algorithm selects the initial bit values of a from before the subtraction took place. The algorithm repeats these three steps until $a < b$ then it finishes. If and only if $a = 0$ a is divisible by b.

Let α_i and β_i be the bit values of either 0 or 1 of a and b. As described in the introduction of this chapter the input strands look like:

$$\underline{+0*^a_m},\underline{+\alpha_{m-1}*^a_{m-1}},\ldots,\underline{+\alpha_0*^a_0}$$
$$\underline{+0*^b_n},\underline{+\beta_{n-1}*^b_{n-1}},\ldots,\underline{+\beta_0*^b_0}$$

Every time we make a calculation in which we change a, we cannot save the result with the same delimiters as a because then we would mix up the bits from a before and a after the calculation. We have to use a new variable with new delimiters. Let a be treated as a^0. After each subtraction from an a_i the interim result is saved in the variable $a^{t(i+1)}$ (t for temporary). Then the algorithm decides whether to continue with either a^i or $a^{t(i+1)}$ (depending whether $a^{t(i+1)}$ is negative or positive). The variable the algorithm continues with is then saved in a^{i+1}.

The formal rules for the first bitwise subtraction are the following, where the delimiter $*^{C_A}_{i+1}$ marks a carry bit (a detailed explanation of these kind of subtraction rules can be found in [1].). Again we use α_i and β_i as the bit values 0 or 1 for a and b. γ_i and α^t_i are bit values of the carry C_A and a^t1 and depend on the strands on the left side of the rules so that the subtraction is correct.

$$+\alpha_{m-n+i}*^a_{m-n+i} + +\beta_i*^b_i + +\gamma_i*^{C_A}_i \rightarrow +\alpha^t_{m-n+i}*^{a^{t1}}_{m-n+i} + +\gamma_{i+1}*^{C_A}_{i+1} \ \forall i = 0,\ldots,n$$

If the value of the most significant bit (MSB) of the result of the subtraction is equal to zero, the algorithm continues with the interim result a^{t1}. If the MSB is equal to one, the result of the subtraction is negative and the algorithm therefore continues with the values of a. The result of the division is saved in the variable r in the following way. Every time when the algorithm decides if it continues with a^i or $a^{t(i+1)}$ the value of the i^{th} bit of r is set to 0 or 1.

The formal rules for selecting the values for a^1 and saving the result is:

$$\begin{array}{ccccccc}
+0*^{a^{t1}}_{m} & + & +0*^{a^{t1}}_{m-n+i} & \rightarrow & +0*^{a^1}_{m-n+i} & + & +1*^{r}_{m-n} \\
+1*^{a^{t1}}_{m} & + & +0*^{a}_{m-n+i} & \rightarrow & +0*^{a^1}_{m-n+i} & + & +0*^{r}_{m-n} \\
+0*^{a^{t1}}_{m} & + & +1*^{a^{t1}}_{m-n+i} & \rightarrow & +1*^{a^1}_{m-n+i} & + & +1*^{r}_{m-n} \\
+1*^{a^{t1}}_{m} & + & +1*^{a}_{m-n+i} & \rightarrow & +1*^{a^1}_{m-n+i} & + & +0*^{r}_{m-n} \\
\end{array}$$
$$\forall i = 0, \ldots, n-1$$

Because the next subtraction takes place at the bits $a_{m-1} \ldots a_{m-n-1}$ the current MSB is not needed for the next step. Therefore $a^i + 1$ is always one bit shorter than a^i.

To improve readability the following rule is used in this script. The rule makes all bits of the interim result have the same delimiter. The rule is not necessary for the algorithm itself to work.

$$\begin{array}{ccl}
+0*^{a}_{i} & \rightarrow & +0*^{a^1}_{i} \; \forall i = 0, \ldots, m-n-1 \\
+1*^{a}_{i} & \rightarrow & +1*^{a^1}_{i} \; \forall i = 0, \ldots, m-n-1
\end{array}$$

These rules give back the variable a^1 with the length m. The steps have to be implemented for all interim results a^j so that they give back the according results a^{j+1}, $\forall j = 1, \ldots, (m-n)$. If all bits of the variable a^{m-n} equal zero a is divisible by b.

The formal positive output rule for the divisibility is:

$$+0*^{a^{(m-n)}}_{n-1} \; + \; \ldots \; + \; +0*^{a^{(m-n)}}_{0} \quad \rightarrow \quad +valid*$$

The result is with ρ_i as the bit values or r

$$+\rho_{m-n}*^{r}_{m-n} \cdots +\rho_0*^{r}_{0}$$

If necessary the following rules can be used for a negative output if a is not divisible by b:

$$+1*^{a(m-n)}_{i} \quad \rightarrow \quad +invalid* \qquad \forall i = 0, \ldots, n-1$$

This algorithm only works if $b_{n-1} = 1$. Otherwise it is possible that b can be subtracted on the same position of a twice without resulting in a negative number. The algorithm above can not subtract twice at the same position therefore the algorithm has to be adapted to this condition. There are several ways to circumvent a complicated adaptation of the algorithm. One is to extend the input number a by the bits a_{m+n} to a_{m+1} which are initially zero. Then the values of a and b are simultaneously multiplied by two (left shifted by one bit) until $b_{n-1} = 1$.

Example

As an example we divide the number $a = 01001$ by the number $b = 011$. The input strands are therefore:

$$\underline{+0*^{a}_{4}},\quad \underline{+1*^{a}_{3}},\quad \underline{+0*^{a}_{2}},\quad \underline{+0*^{a}_{1}},\quad \underline{+1*^{a}_{0}},\qquad \underline{+0*^{b}_{2}},\quad \underline{+1*^{b}_{1}},\quad \underline{+1*^{b}_{0}}$$

The subtraction takes place at the first three bits of a. To subtract the bits of a and b we add $-b = 101$ to the bits of a. In the first step we subtract b from the first three bits of a: 010.

$$\begin{array}{lcl}
+0*^{a}_{2} + \quad +1*^{b}_{0} & \rightarrow & +1*^{a^{t1}}_{2} + +0*^{C_A}_{1} \\
+1*^{a}_{3} + +1*^{b}_{1} + +0*^{C_A}_{1} & \rightarrow & +1*^{a^{t1}}_{3} + +0*^{C_A}_{2} \\
+0*^{a}_{4} + +0*^{b}_{2} + +1*^{C_A}_{2} & \rightarrow & +1*^{a^{t1}}_{4} + +0*^{C_A}_{3}
\end{array}$$

Because the result of the subtraction is 111 is negative the algorithm selects the bits of a to continue the calculation.

$$\begin{array}{lcl}
+1*^{a^{t1}}_{4} + +1*^{a}_{3} & \rightarrow & +1*^{a^{1}}_{3} + +0*^{r}_{2} \\
+1*^{a^{t1}}_{4} + +0*^{a}_{2} & \rightarrow & +0*^{a^{1}}_{2} + +0*^{r}_{2}
\end{array}$$

The following rules to change the delimiters for the readability are executed independently of the subtraction.

$$\begin{array}{lcl}
+0*^{a}_{1} & \rightarrow & +0*^{a^{1}}_{1} \\
+1*^{a}_{0} & \rightarrow & +1*^{a^{1}}_{0}
\end{array}$$

At this point we have $+0*^{r}_{2}$ as a part of the result and the interim variable a^1 as:

$$+1*^{a^{1}}_{3} \quad +0*^{a^{1}}_{2} \quad +0*^{a^{1}}_{1} \quad +1*^{a^{1}}_{0}$$

In the next step we get the subtraction $100 + 101 = 001$ which gives us the result bit $+1*^{r}_{1}$. Here the result of the subtraction is positive therefore the rules

$$\begin{array}{lcl}
+0*^{a^{t2}}_{3} + +0*^{at2}_{2} & \rightarrow & +0*^{a^{2}}_{2} + +1*^{r}_{1} \\
+0*^{a^{t2}}_{3} + +1*^{a}_{1} & \rightarrow & +1*^{a^{2}}_{1} + +1*^{r}_{1}
\end{array}$$

are executed. The variable a^2 is

$$+0*^{a^{2}}_{2} \quad +1*^{a^{2}}_{1} \quad +1*^{a^{2}}_{0}$$

And we get $+1*_0^r$ and a^3 with $011 + 101 = 000$:

$$+0*_1^{a^3} \quad +0*_0^{a^3}$$

We get a positive result for the divisibility with the output rule:

$$+0*_1^{a^3} + +0*_0^{a^3} \rightarrow +valid*$$

The result of the division is 011

$$+0*_2^r, +1*_1^r, +1*_0^r$$

Rule and Speed Complexity Analysis

For a complete division with the Long Division Algorithm, the three steps presented above are executed $(m - n)$ times. Because the interim results all use different delimiters the algorithm uses $(m - n)$ different subtraction rules and $(m - n)$ different rules for selecting the bits to continue with. "Each subtraction rule consists of $32n$ strands (eight composition rules each consisting of four rules)" [1] and the rules for selecting the bit values to continue consists of $16(n-1)$ strands. With the output rule that uses n strands the algorithm uses a total amount of rule $(m - n) \cdot (32n + 16(n - 1)) + n$ strands.

With $m = log(a)$ and $n = log(b)$ the rule complexity is

$$\mathcal{O}(m \cdot n - n^2) = \mathcal{O}(log(a) \cdot log(b) - log(b)^2)$$

To get all a_i $\forall i = 1, \ldots, (m - n)$, $4n$ rules per bit for the subtraction and $4(n - 1)$ rules for selecting the right bits have to be executed. The steps needed by the algorithm to come to a result therefore is $(m-n)\cdot(4n+4(n-1))$. Which results in a speed complexity of

$$\mathcal{O}((m - n) \cdot n) = \mathcal{O}((log(a) - log(b)) \cdot log(b))$$

3.2 A Parallel Long Division Algorithm

The here presented algorithm is therefore a slightly varied version of the Long Division algorithm that has an improved speed complexity in a trade-off to a larger rule complexity. In the Long Division Algorithm for every position of a, the algorithm decides once if b is subtracted at this position or not. To improve the speed complexity in the Parallel Long Division Algorithm, we can subtract at k positions at once. Let p_i $\forall i = 1, \ldots, m + 1$ be all positions of a. We now divide the set P of all p_i in disjoint subsets P_j $(P_j \cap P_l = \emptyset, \forall j \neq l)$ of size k.

In every step of the algorithm we take one subset P_j. Then at every $p_i \in P_j$ we do two different calculations simultaneously. In one calculation the algorithm subtracts b at position p_i of a and in the other it does not. For all k p_i this leads to a total of 2^k different combinations and therefore interim results.

If a subtraction leads to a negative carry, the carry is saved as a 1 at its bit position in a new number c. For every interim result a different c exists that has an according delimiter. In the beginning every bit of c is zero. The number a is divisible by b if and only if at the end of the algorithm there exists an a that is equal to its corresponding c.

Example

We show the principle of this algorithm with the division of $a = 001001$ by $b = 011$. We define $P_1 = \{1, 3\}$ and $P_2 = \{0, 2\}$. In the first step the algorithm computes the following. We select P_1 so $p_1 = 1$ and $p_2 = 3$. At position $p_1 = 1$ of a the bits 00 are chosen and extended by the sign bit 0. Then $-b = 101$ is added: $000 + 101 = 101$. The new bits at position $p_1 = 1$ are now 01 and the in c a 1 is added at the 4^{th} bit.

Simultaneously the algorithm does the same at position $p_2 = 3$: $001 + 101 = 110$. We now have $2^2 = 4$ possible combinations with the respective saved carries. We distinguish the interim variables with a superscript which includes at which position a subtraction has taken place.

No subtraction:

$$a^{0x0x} = 001001, \quad c^{0x0x} = 000000$$

Subtraction at position $p_1 = 1$:

$$a^{0x1x} = 001\mathit{01}1, \quad c^{0x1x} = 001000$$

Subtraction at position $p_2 = 3$:

$$a^{1x0x} = 0\mathit{10}001, \quad c^{1x0x} = 100000$$

Subtraction at position $p_1 = 1$ and $p_2 = 3$:

$$a^{1x1x} = 0\mathit{1001}1, \quad c^{1x1x} = 101000$$

The algorithm now does the same for P_2. This results, e.g. in

$$a^{0000} = 001001, \quad c^{0000} = 000000$$

and

$$a^{0011} = 0010\mathit{00}, \quad c^{0011} = 001000$$

We see that 0011 is the result of the division because $a^{0011} = c^{0011}$.

Implementation

First we explain the implementation for the case that the selected positions p_i for the subtraction are at least n (not $n+1$ because of two's complement) bits away of each other. In this case no bit is affected by two subtractions at once. A subtraction at position p_i is implemented the following way. We take the bits $a_{p_i+n} \dots a_{p_i}$ and add a 0 as MSB. This is because of the two's complement the $n+1^{th}$ bit is the sign bit:

$$0\, a_{j+n} \dots a_j$$

Then we subtract b from the bits. If the result is negative ($MSB = 1$), we save the negative carry as a 1 in the corresponding bit in c of the corresponding path. The interim result are the last n bits of the result (without sign bit). If the result is positive, this is the same value as the result. If the result was negative, the value of the interim result equals the negative result added by 2^{n+1}.

The rule strands are equivalent to the Long Division Algorithm only that in the Parallel Long Division Algorithm more unique delimiters for the different interim variables are used. Because for every delimiter other rule strands are used more rules have to be implemented too.

If the some positions p_i and p_{i+1} for the subtractions are less than n bits away of each other, some bits are affected by two subtractions at once. In this case we cannot do the subtractions all at once but have to compute the overlapping subtractions consecutively. Because of this, we cannot improve the speed complexity any further with this algorithm than by $k \leq \lceil m \rceil$.

Complexity and Speed Analysis

Without overlapping subtractions the following complexities hold. Again for every subtraction $32n$ strands are used. Because for every unique delimiter a different subtraction rule has to be used $\sum_{i=0}^{\frac{m-n}{k}} (2k)^i = \frac{2^{\frac{k+m-n}{k}} k^{\frac{k+m-n}{k}} - 1}{2k-1}$ subtraction rules are used for k parallel subtraction positions.

$$\mathcal{O}(2^{\frac{m-n}{k}} k^{\frac{m-n}{k}} \cdot n)$$

For each subtraction the algorithm again needs $4n$ steps, therefore the number of steps we need in one path to reach a result is $\frac{m-n}{k} \cdot 4n$. This results in a speed complexity of

$$\mathcal{O}(\frac{m-n}{k} \cdot n)$$

We get an expected speedup of k in terms of speed complexity. But the rule complexity increases compared to the Long Division Algorithm.

3.3 Single Strand Division Algorithm

As we have seen before the representation of the binary numbers by single bit strands leads to high rule complexities. Therefore we introduce a single strand representation of a binary number and a Single Strand Division Algorithm that uses this representation. The advantage of the single strand approach is that there is no need of special delimiters to determine the belonging of a bit. The biggest disadvantage is that the further away a bit is from a delimiter the more complicated it gets to access the bit. The basic idea of the algorithm is again the long division. The algorithm uses single rules to subtract b from the last a bits until a equals zero or no more subtraction is possible.

Single Strand Binary Representation

To represent the bits in one strand a special binary representation is used. The bits are all on one strand and are separated by a separating character $|$. Every bit can have the following values " " (corresponding to 0), "1", "-1" or "$1-1$". In the beginning all strands consist only of either " " or "1", e.g.

$$1||1|1||||1 = 10110001$$

A number does not have a unique representation e.g.

$$1_{10} = 1|-1 = 1-1|1 = |1$$

The input strands are the dividend a in the special binary representation and the divisor b in normal binary representation. For each subtraction with all possible divisors b and the last $n = l(b)$ possible bits of a, we have rule strands which are compositions rules and directly return us the right result, e.g. $10*^a$ ++$01*^b \rightarrow 01*^a$. If $b > a$ a "-1" is added as a negative carry one digit left of a_n. At every position we can only subtract the divisor once. If the last bit of a is a zero, then the algorithm can delete the last bit. Therefore for some strands of a, the algorithm subtracts b at the last position and for others it does not.

Example

We divide $a = 100100_{bin} = 1|||1||$ by $b = 11$. In the first step the algorithm can either subtract b from the last two bits of a or delete a zero at the end of a. Subtracting b from a leads to the following result $1|||1-1||1$. The algorithm cannot continue with this number because it can only subtract a number at a certain position once and because the last bit is not a zero the algorithm cannot delete the last bit. Let us take a look now on the path that leads to $a = 0$. After deleting twice the last zero the algorithm subtracts b from a: $1|||1-b = 1|-1|1|$. Now again the algorithm deletes a zero and we obtain $1|-1|1$. The algorithm pushes the negative carry one bit to the left: $1|-1|1 = 1-1|1|1 = |1|1$. The next subtraction with b results in zero therefore a is divisible by b.

A special case occurs if the last bit of b is zero, e.g. if we divide $a = 10_{bin} = 1|$ by $b = 10$. For some a the algorithm deletes zero resulting in $a = 1$. The algorithm cannot continue with this number. But because simultaneously the algorithm also subtracts b from a with other strands, the algorithm gives back the correct result that a is divisible by b.

Implementation

The rule strands look like this:

$$\begin{aligned}
|||\cdots|* \ + \ {+0}\cdots0*_b &\ \rightarrow\ |||\cdots|*_{subt}\\
|||\cdots|1* \ + \ {+0}\cdots0*_b &\ \rightarrow\ |||\cdots|1*_{subt}\\
&\ \ \vdots\\
|||\cdots|* \ + \ {+0}\cdots01*_b &\ \rightarrow\ -1|||\cdots|*_{subt}\\
&\ \ \vdots\\
1|1|1|1\cdots|1* \ + \ {+1}\cdots1*_b &\ \rightarrow\ |||\cdots|*_{subt}
\end{aligned}$$

With the subtraction rules b is subtracted from a and we add a negative carry if necessary ("-1") to the $n+1^{th}$ bit from the right. The delimiter $*_{subt}$ prevents that the algorithm subtracts twice at the same position. For any possible $a_n \dots a_0$ where -1 occurs in a_n the following rules ensure that the last n bits only consist of "" and "1" so that the subtraction rules can always be used. Only one negative carry can exist at once :

$$\begin{aligned}
|-1|||\cdots* &\ \rightarrow\ -1|1|||\cdots*\\
|1-1|||\cdots* &\ \rightarrow\ ||||\cdots*
\end{aligned}$$

.

The following rules delete a zero at the end of a:

$$\begin{aligned}
|* &\ \rightarrow\ *\\
|*_{subt} &\ \rightarrow\ *
\end{aligned}$$

The output rule is

$$\texttt{+}* \quad \rightarrow \quad 'divisible'.$$

Rule and Speed Complexity Analysis

An advantage of the Single Strand Division Algorithm is that its rule complexity is independent of the dividend a. For the subtraction the algorithm uses $2^n \cdot 2^n$ rules. In addition the algorithm use 2^n rules shift the carry. All together this leads to a rule complexity of

$$\mathcal{O}(2^{2n}) = \mathcal{O}(b^2)$$

To reach a positive result the algorithm has to subtract b at most $(m-n)$ times. After the subtraction the algorithm has to delete a zero at the end and in the worst case shift the negative carry one bit to the left, which are in total three steps. The maximum number of steps used by the algorithm to reach a positive result is therefore $(m-n) \cdot 3$. Giving a speed complexity of

$$\mathcal{O}(log(a) - log(b))$$

This algorithm is useful if b is small compared to a because then the rule complexity of the Single Strand Algorithm $\mathcal{O}(b^2)$ is smaller than the complexity of the Long Division Algorithm $\mathcal{O}(log(a) \cdot log(b) - log(b)^2)$.

Chapter 4

3SAT

The satisfiability problem copes with the question whether a certain boolean equation (e.g. $x_1 \cap (x_2 \cup x_3)$) can be fulfilled or not. If the equation can be fulfilled, it is called satisfiable. The 3-Satisfiability restricts the problem to the equations that are represented in the conjunctive normal form with at most three variables per clause (e.g. $(x_1 \cup x_2 \cup x_3) \cap (x_1 \cup x_4 \cup \neg x2) \cap \dots$). The 3SAT problem has been approached several times with DNA computing e.g. in [3] where the problem was faced with a DNA Computer. We present two possible algorithms using the Domain Replacement Model that compute if an equation is satisfiable.

4.1 One of Each Clause Algorithm

The idea of the algorithm is to pick one variable per clause that the algorithm sets to true. The algorithm does this for all clauses. In the end the algorithm checks if there are not contradictions. If there are no contradictions the equation is satisfiable.

The input of the algorithm are the single variables encoded with delimiters that are both labelled according to its clauses, e.g. a clause $(x_1 \cup x_2 \cup x_3)$ yields in $+_1 x_1 *_1$, $+_1 x_2 *_1$ and $+_1 x_3 *_1$. Either we create the input strands directly like this or if it is not possible or preferred, then splitting the clauses into its variables can also be implemented with rule strands. (e.g. $+(x_1 \cup x_2 \cup x_3)*_1 \rightarrow +_1 x_1 *_1 + +_1 x_2 *_1 + +_1 x_3 *_1$)

We implement the selection of one variable for every bracket the following way. For all succeeding clauses a collapsing rule exists that joins the content of both strands. E.g. , the collapsing rule for clause one and two looks the following: $\bar{*}_1 \bar{+}_2$. With the variable strands $+arg1*_1$ and $+arg2*_2$ the collapsing of the two strands looks like:

$$+_1 arg1 *_1 + \bar{*}_1 \bar{+}_2 + +_2 arg2 *_2 \rightarrow +_1 arg1\ arg2 *_2$$

This results in a long string of variables (e.g. $x_1 x_1 x_4 x_6 x_1 x_2 \neg x_4 \dots$). To check if the string contains contradictions, all output strands for all possible strands with a length l (number of clauses) that contain no contradictions have to be created. With the number of different variables a this results in a rule complexity of $\mathcal{O}(a^l)$.

4.2 Clause Choosing Algorithm

The second algorithm decides for each variable whether it is true or not. Then it randomly selects clauses that are true. In the end the algorithm checks whether the selection contains all clauses. If it contains all clauses the equation is satisfiable.

To implement this algorithm, the input strands do not contain the variables but the number of the clauses. The delimiters are labelled with the according variable, e.g. $+_{x_1}1*_{x_1}$ or $+_{x_1}10*_{x_1}$ implicate that in bracket 1 and 10 the variable x_1 occurs. Again we use collapsing rules to stick the strands together. The algorithm uses collapsing rules for succeeding variables, e.g. the collapsing rules for the variables x_1 and x_2 are

$$\overline{*_{x_1}}\,\overline{+_{x_2}},\quad \overline{*_{\neg x_1}}\,\overline{+_{x_2}},\quad \overline{*_{x_1}}\,\overline{+_{\neg x_2}},\quad \overline{*_{\neg x_1}}\,\overline{+_{\neg x_2}}$$

but also for the same variables

$$\overline{*_{x_1}}\,\overline{+_{x_1}},\quad \overline{*_{\neg x_1}}\,\overline{+_{\neg x_1}}$$

This results in a long string of numbers (e.g. '1"17"5"23' ...). To check if the string contains every clause number at least once all output strands for strands with a length l (number of clauses) containing every clause number exactly once are created. This leads to an amount of rules in the order of $\mathcal{O}(l!)$.

Chapter 5

Pattern Matching

In this chapter we analyse the search for certain patterns in a string. We have a string s, a known pattern p that may appear several times in s and the elements ε_i in the known alphabet Σ.

5.1 Matching a Single Pattern in a String

There are two ways for matching p in s. Either by deleting elements from the input strand containing s until the strand only contains p or by adding elements to a strand containing p until it matches s.

An Example Cutting Algorithm

The algorithm is a very basic example of how the string s can be cut to find the pattern p. It is not the most efficient way to do so but it points out the idea. Randomly either the first or last element of s is removed. If $s = p$ the output rule gives a positive feedback.

The Input strands contain s:

$$\underline{+s*}$$

The rule strands randomly delete the first or last element of s:

$$+\varepsilon \rightarrow +$$

$$\varepsilon * \rightarrow *$$

$$\forall \varepsilon \in \Sigma$$

The output rule gives a positive result for a strand matching p.

$$+p* \rightarrow' matched'$$

An Example Algorithm for Extending p

The example algorithm for extending `p` is similar to the cutting algorithm. Instead of deleting elements in `s`, the algorithm adds randomly elements at the front or back of `p`. If such an extended `p` equals `s` the output rule gives a positive feedback.

The Input strands are the searched pattern `p`:

$$\underline{+p*}$$

The rule strands randomly add elements to the front or back of `p`:

$$+ \rightarrow +\varepsilon$$
$$* \rightarrow \varepsilon*$$
$$\forall\varepsilon \in \Sigma$$

Output rule:

$$+s* \rightarrow' matched'$$

If `s` is not well known (e.g. `s` is a completely unknown DNA strand that we want to analyse further or `s` is so long that it takes a non negligible effort to build its complement) the output rule cannot be created like above because $\bar{s}$ is not known. Instead we can create the output rule $+\bar{s}* \rightarrow' matched'$ by adding a $'matched'$ and the complementary domains of the delimiters to `s`. The output rule then looks like $\bar{+}s\bar{*}+'matched'*$. Instead of the input and rule strands introduced above, we have to use the complement of the input and rule strands.

Additional Approaches

If we want to determine the position of the pattern, the algorithm has to count the deleted or added elements. If two different patterns p_1 and p_2 both have to be found, this can be implemented with an adapted version of the extension algorithm by adding elements between, in front and after the patterns, e.g. $X_1p_1X_2p_2X_3$ and $X_1p_2X_2p_1X_3$ where X_i is a random sequence of elements of the alphabet.

5.2 Estimating the Number of Occurrences of a Pattern in a String

In this part we want to find out how often `p` occurs in `s`. For few occurrences of `p` in `s` we can implement this the following way. An algorithm that can find a maximum of a patterns `p` in `s` is an adapted extension algorithm where $\forall i \in [1, \ldots, a]$, i patterns are chosen and extended with random elements, between, in front and after the patterns, to match the string. The output rules of the algorithm then return positive results for all $i \leq n$ where n is the searched amount of patterns `p` in `s`. The biggest i is the result. A downside of this algorithm is that the number of rules increases the more patterns we want to find. In the next section we present a rule efficient way to estimate the number n of occurrences of `p` in `s` with a constant amount of rule strands needed.

Example

We demonstrate the algorithm with a maximum number $a = 3$ of rules that can be found with this algorithm and with the number of occurrences $n = 2$ of `p` in `s`. Let X_i be a random sequence of elements of the alphabet Σ. Then `s` has the form of

$$X_1pX_2pX_3$$

For $i = 1$ we get strands in the form of X_1pX_2 which result in a positive output.

For $i = 2$ we get strands in the form of $X_1pX_2pX_3$ which results also in a positive output.

For $i = 3$ we get the strands in the form of $X_1pX_2pX_3pX_4$ which does not result in a positive output. Therefore $n = 2$ patterns `p` exist in the string `s`.

For every i, the algorithm uses the number of elements e in Σ times i new rules. If we want to count a patterns the algorithms uses $\Sigma_{i=1}^{a} i \cdot e = \frac{1}{2} \cdot n \cdot (n+1)$.

5.3 Estimation of Occurrences of a Pattern `p` in a String `s`

We use a cutting pattern matching algorithm that searches for `p` in `s`. A pattern `p` is found only with a certain probability p_0 because some of the patterns are deleted by the algorithm. p_0 can be influenced by the concentration ratio between the rule strands and the output strands in the cutting algorithm. If there are many output strands compared to rule strands, then it is more probable that a positive output occurs, thus p_0 gets bigger.

If a `p` is found in `s`, we give an output, e.g. by fluorescing or by a DNA strand. At most one output can be created per string `s` because the other occurrences of `p` are deleted by the algorithm. Independent of how we count the matches exactly, we assume that we get for every match a "+1". We assume that we cannot observe all counters but only a sample.

Let h be the number of occurrences of `p` in `s`, that we want to estimate. Then $p_1 = 1 - (1 - p_0)^h$ is the probability that one pattern in one string is found. To get h we therefore need p_1. We know n that is the number of strings `s` in the soup and that is also the maximum amount of "+1"s that we can count (because at most one pattern `p` can be matched per string `s`). We define A as a random variable that gives back the number of matched `p`s. It is binomial because for every string `s` we have the probability p_1 that one pattern is found and $(1 - p_1)$ that none is found. The probability mass function of A is:

$$P(A = a|p_1, n) = \binom{n}{a} \cdot p_1^a \cdot (1 - p_1)^{n-a}$$

We assume that we have $y \gg 1$ counters. The probability that an output (e.g. a "+1") is counted by a certain counter is $p_2 = \frac{1}{y}$. We define the random process X that gives back the number of counted outputs of one counter. It is also binomial with the probability mass function:

$$P(X = x|p_2, a) = \binom{a}{x} \cdot p_2^x \cdot (1 - p_2)^{a-x}$$

Let c be the number of observed counters. We get c random processes X_i and the respective observed random variables x_i, $\forall i = 1, \ldots, c$. Because we assume that n and therefore a are very large and p_2 is very small the x_i can be treated as being independent.

We get the following conditional probability

$$P(X_i = x_i|P_1 = p_1) = \sum_{a=0}^{n} P(X_i = x_i|A = a) \cdot P(A = a|P_1 = p_1)$$

with

$$P(X_i = x_i|A = a) = 0 \quad \forall x > a$$

resulting in

$$P(X_i = x_i|P_1 = p_1) = \sum_{a=x_i}^{n} \binom{a}{x_i} \cdot p_2^{x_i} \cdot (1 - p_2)^{a-x_i} \cdot \binom{n}{a} \cdot p_1^a \cdot (1 - p_1)^{n-a}$$

Because the x_i can be treated as independent we get for the joint probability mass function:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_c = x_c|P_1 = p_1) =$$
$$\prod_{j=1}^{c} \sum_{a=x}^{n} (\binom{n}{a} \cdot p_1^a \cdot (1 - p_1)^{n-a}) \cdot (\binom{a}{x_j} \cdot p_2^{x_j} \cdot (1 - p_2)^{a-x_j})$$

In this formula the only unknown variable is p_1. We can now insert the other variables and calculate the most likelihood estimator of p_1 and its reliability. Then we can solve the equation $p_1 = 1 - (1 - p_0)^h$ for h which is the estimation of the occurances of `p` in the string `s`.

Chapter 6

The Inclusion Model

In this chapter we suggest and discuss a varied model for DNA computation. With the model we are able to create more powerful languages than with the previously used model. It will allow us to create algorithms to include domains into strands while using a constant amount of rules.

6.1 Inclusion into a Strand with the Domain Replacement Model

Accessing a position in a strand gets more difficult the further away the position is from the ends of the strand. Given an alphabet Σ, we have to create the following rules if we want to replace a character a by a character b at the third position from the right

$$\bar{*}\bar{\varepsilon}_1\bar{\varepsilon}_2\bar{a}b\varepsilon_2\varepsilon_1*$$
$$\forall(\varepsilon_1,\varepsilon_2)\epsilon\Sigma\times\Sigma$$

The further away the position is from the delimiters and the bigger the alphabet of the language is, the more rule strands are needed. Implementing a replacement like above but on the i^{th} position with n elements of the alphabet yields in n^i necessary rule strands for only a single action. In the following we present a model that allows us to create algorithms for inclusion of strands into other strands while using a constant number of rules, independent of the position where the inclusion takes place and of the number of elements in the alphabet.

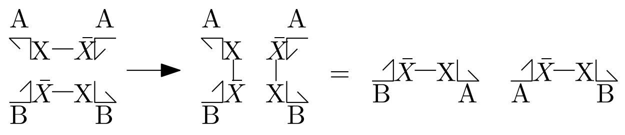
6.2 The Inclusion Model

The new model bases on the Domain Replacement Model but we make some assumptions to extend the model. We assume that there exist domains X to which complementary domains $\bar{X}$ can bind at all positions of the strand. We represent such a binding by a "−", e.g.

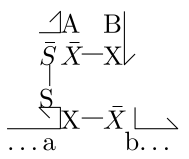
$$\mathrm{X}{-}\bar{X}$$

Next we assume that a binding can dissolve, but only if instantly a new binding with one of the dissolved domains is formed. In practice this has reasons regarding the energy of the molecules. Because bound strands have a lower energy state than unbound strands and are therefore more stable, the bindings will not dissolve without creating a new binding. Otherwise they would have to take an amount of energy permanently from their environment which we assume does not happen in our model.

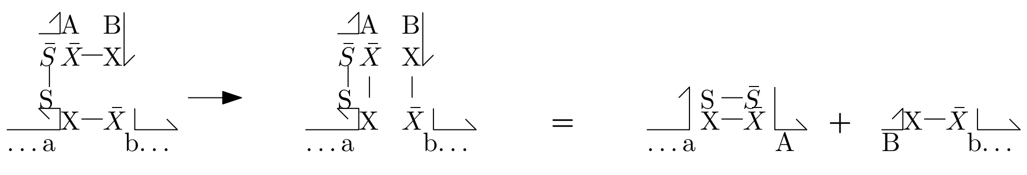
We assume that it is unlikely for a binding to dissolve. Therefore we assume that the probability that two bindings dissolve at the same moment close enough together to swap the bindings like this



is so small that we neglect this case in our model. Only if we keep the bindings together, e.g. like this



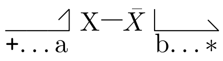
a swap of the bindings is possible



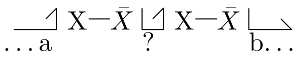
The swap of the binding of the first X next to S is a branch migration and is treated in detail in e.g. [4]. We assume that there do not exist unbound X or $\bar{X}$ in the soup, therefore a binding between an X and $\bar{X}$ can only dissolve during branch migration as shown above.

6.3 Developing an Inclusion Algorithm

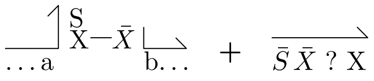
To implement an inclusion the domains should not be all on the same single strand otherwise we face the same problem as shown in 6.1 with a non constant amount of rules to reach any position. If we want to include a domain between a domain a and a domain b in the strand $+\ldots ab\cdots*$, we use the following bound strands instead



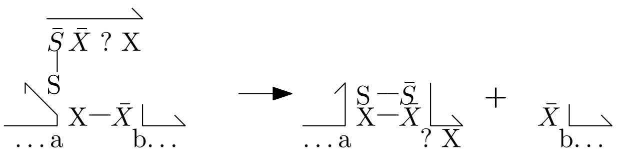
which has the effective strand $+\ldots ab\cdots*$. If we manage to dissolve the binding, then a strand $\bar{X}?X$ can be included



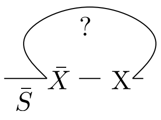
There are two main problems in this step. How are the strands separated and how is ensured that both strands bind to the same inclusion strand. For separating the a- and the b- strand we add a start-symbol S and its complement to the strands:



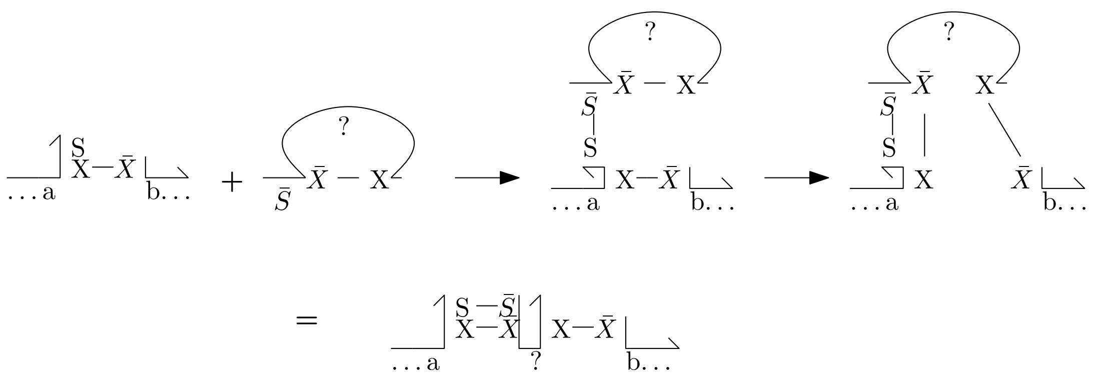
The starting symbols can now bind together and then allow branch migration with the X and its complement to take place.



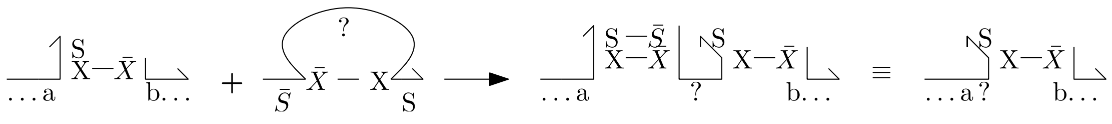
The strand $\bar{X}b\ldots$ does not necessarily bind to the $S\bar{X}?X$ strand from the same reaction in this case. The reaction above that does not fit our model because there are unbound X and $\bar{X}$ in the strand $\bar{S}\bar{X}?X$ and in the strand $\bar{X}b\ldots$. To make the strands fit into our model, we create a loop strand similar to hairpin loops covered in [5].



With this loop strand we can do the reaction to separate a and b from above without violating our model assumptions:

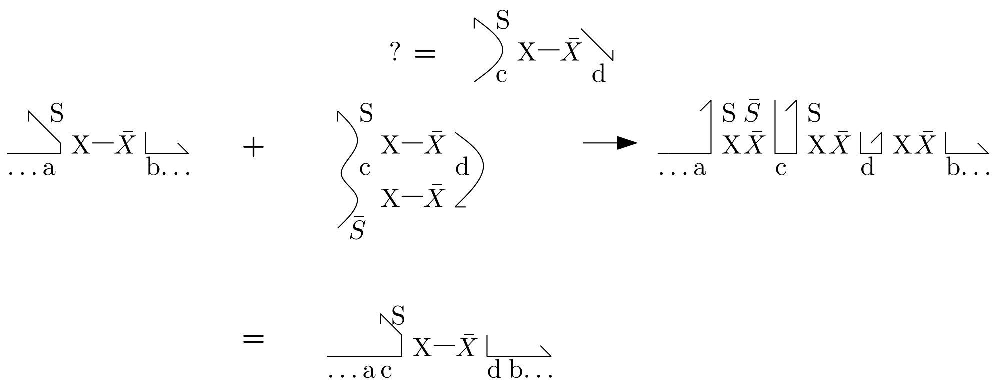


With these strands we can realize one inclusion. To be able to always be able to include a strand at this position we add a starting domain to the ?-strand: $\bar{S}\bar{X}?XS$

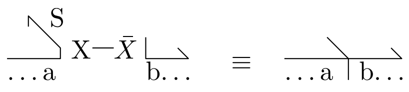


The strand on the right of the reaction is nearly the same strand as on the left but with a ? domain included between a and b. Because we did not restrict the domain ? in any way, ? can be any domain as long as it fits our model. The strands on the left and on the right of the equation sign have the same properties therefore an equality sign is used to express the effective strand on the right.

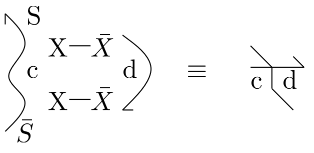
An even more powerful approach is to create the new inclusion in the ? part of the included strand



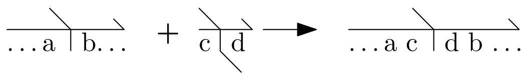
We denote a strand in which such an inclusion can take place as



and the strand that we can include as

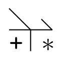


In inclusion then looks the following

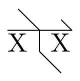


6.3.1 Creating Palindromes

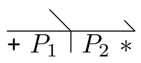
An algorithm that creates palindromes uses the following rules. We start with the strands:



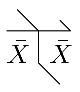
and as rule strands where X can be every element in the alphabet of possible domains.



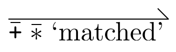
With these rules we can create strands that are palindromes. If we want to check if an input strand is a palindrome. We can either use input strands of the form:



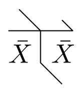
with the rules



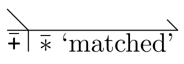
and the output rule



Another way to check if a string P is a palindrome, is to use $+P*$ as input strand and then create palindromes on the complementary alphabet of the string we want to check.



and then create an output rule with it



Strings in the form of $a^n b^n$ can be created in a similar way. Also the generation of rule strands out of other rules strands gets easier with this.

Bibliography

[1] Mattia, N.: Parallel Computing with DNA (January 2015)

[2] Fukagawa, H., Fujiwara, A.: Procedures for Multiplication and Division in DNA Computing

[3] Ravinderjit S. Braich1, Nickolas Chelyapov1, C.J.P.W.K.R.L.A.: Solution of a 20-Variable 3-SAT Problem on a DNA Computer. Science **296**(5567) (2002) 499–502

[4] Panyutin, I.G., Hsieh, P.: The kinetics of Spontaneous DNA Branch Migration. Proceedings of the National Academy of Sciences **91**(6) (1994) 2021–2025

[5] Bonnet, G., Krichevsky, O., Libchaber, A.: Kinetics of Conformational Fluctuations in DNA Hairpin-Loops. Proceedings of the National Academy of Sciences **95**(15) (1998) 8602–8606