Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Parallel Computing with DNA

Semester Thesis

Nicolas Mattia

nmattia@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

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

January 16, 2015

Abstract

Over the past years, crafting artificially designed DNA strands has increasingly become a cheaper and easier process. This paper takes direct benefit from this and researches new ways of designing DNA-based algorithms, solely using carefully designed DNA strands; no enzymes are needed, making the implementation of such DNA algorithms a much simpler process. To those means, a new representational model was developed that is simple, predictable, and that should present good practical results. Divisibility and primality testing algorithms as well as a square root computing algorithm were designed, using the aforementioned model. Two different process models were also developed, used to assess the complexity of the DNA algorithms in terms of some defined complexity characteristics. During the course of this research, a simulation tool for the process models was also programmed.

Contents

Chapter 1

Introduction

DNA strands consist of nucleotides (or bases), linked together in a specific order. There are four types of nucleotides: Adenine, Guanine, Cytosine and Thymine (or simply A, G, C and T). At an atomic level, DNA strands have two types of extremities: the 3' end on one side, and the 5' end on the other side. The details and reasons why the extremities are called that way will not be of interest here, only that a direction can be assigned to a DNA strand, and that by convention DNA strands are written down from the 5' end towards the 3' end. Because they are directed, the strands $ATCT$ and $TCTA$ are *not* equivalent, and will be treated differently in the rest of this paper. When displayed, an arrow above the strand will indicate the direction from the 5' end to the 3' end, as in Fig 1.1.

An A can *bind* (via hydrogen bonds) with a T on a different strand, and similarly a G can bind with a C. This is the process that gives DNA its stability, and its helicoidal structure. Pairs of nucleotides which are able to bind in this manner are called *Watson-Crick complementary* (or simply *WK complementary*). A and T are WK complementary, and G and C are WK complementary. Complementary sequences of nucleotides can bind as well, given that those sequences have opposite directions (the directions are given by the strand that they are a part of). For instance, the strands $ATCT$ and $AGAT$ would bind completely (whereas $ATCT$ and $TAGA$ would not, given the convention on the direction).

Sequences of nucleotides can be conveniently abstracted by domains. Such a domain will be represented by a single character displayed in `teletype` font.

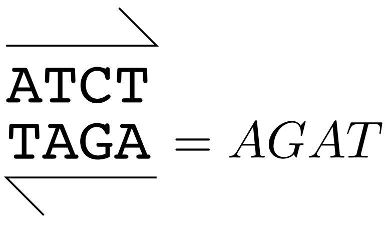

Figure 1.1: Example of two complementary strands. The arrowhead shows the 3' end.

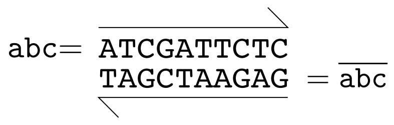

Figure 1.2: Example domain representation.

For instance:

$$ATCGATTCTC \equiv \texttt{abc} \ ,$$

where

$$ATCG \equiv \texttt{a}, ATT \equiv \texttt{b}, CTC \equiv \texttt{c} \ .$$

Note that the domains need not be of equal lengths, and that many different representations can be used for the same DNA strand. Two domains $\texttt{g}$ and $\texttt{h}$ can also bind, given that one is the *WK* complementary sequence of the other. Complementary domains will be represented as overlined, so $\texttt{g} = \bar{\texttt{h}} =$ WK-complement($\texttt{h}$). For the above:

$$CGAT \equiv \bar{\texttt{a}}, AAT \equiv \bar{\texttt{b}}, GAG \equiv \bar{\texttt{c}}, \ \overline{\texttt{abc}} \equiv \bar{\texttt{c}}\bar{\texttt{b}}\bar{\texttt{a}} \ .$$

Note that the last equality follows from the order of the complementary domains, as seen in Fig 1.2.

Two strands need not be perfect matches in order to bind. For instance, the strands $s_1 = ATCGATTCTC$ and $s_2 = ATAAATCGAT$ would still bind on s_1's first seven characters and s_2's last seven characters. This is a *partial* binding. It is assumed that bindings will always cover the longest number of domains possible; in the previous example, a binding over four, five, or six domains would only be temporary, until the seventh domain binds as well. In this paper, the process of binding over each domain successively will not be considered, and only the final binding will be taken into account.

1.1 Soup Setting and Domain Replacement

DNA computations, at a hardware level, do not bear much resemblance to typical electronic, silicon-based computations. In the case of this paper, a DNA computation takes place in a *soup*, i.e. a container filled with solution, in which DNA strands are floating around and binding with each other. In this sort of DNA computing, designing algorithms is equivalent to finding the right DNA strand types such that, when strands representing an input to the problem are added, a desired output is produced. Production is done through different steps of *strand binding*. The DNA strands that are designed are equivalent to a processor (with memory) and its circuits in a computer, and the input strand types are equivalent to an input signal.

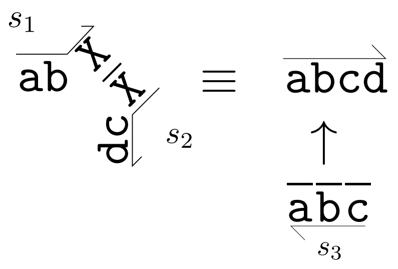

Figure 1.3: Example of a domain replacement.

Output reading is not straightforward. If an algorithm is successfully designed, and an output strand successfully created in the soup, how does one read it? Fortunately, chemical compounds can be crafted to react with a given strand type, and emit light. It will be assumed that outputs can be read (or at least detected).

Simple replacement When two strands s_1 =`abX` and s_2 =$\bar{\texttt{X}}$`cd` meet and bind partially with their `X` and $\bar{\texttt{X}}$ domains, respectively, only the `a`,`b`,`d` and `c` domains will be available for further bindings. Considering that DNA strands are not completely rigid, the above bound strands could therefore bend and bind with a third strand $s_3 = \overline{\texttt{abc}}$, as shown in Fig 1.3. After the first bond, s_1 and s_2 will form a new *effective* strand `abcd`, which can then bind again with s_3, yielding another *effective* strand containing a single domain, `d`. Such an effective strand would be a stack of actual DNA strands, which have bound together, effectively having only a few free domains able to bind. In theory, the process of stacking DNA strands can be repeated as many times as needed (in practice, this effective strand carries more and more inactive strands, which can prohibit freedom of movement). From now on, all strands will be considered as effective ones.

The process explained above shows something interesting. The strand s_2 can be used to create a new strand similar to s_1, except that the latter's rightmost domains have been replaced by that of the former. As an example, if three strands $s_0 =$ `mcX` are released in a solution containing the strands $s_1 = \bar{\texttt{X}}$`nuggets`, $s_2 = \bar{\texttt{X}}$`flurry`, $s_3 = \bar{\texttt{X}}$`kenzie`, given enough time, the following strands could be observed:

`mcnuggets`

`mcflurry`

`mckenzie`

1.2 A Simple Algorithm

This can be applied to solve simple problems, like deciding if a number is divisible by another one or not. To check that a natural number n is not divisible by another natural number m, one could apply the following algorithm, and verify that the remainder r is never equal to zero:

```
$r \leftarrow n$
**while** $r \geq 0$ **do**
    $r \leftarrow r - m$
**end while**
```

In unary representation, the number five can be written as

$$aaaaa$$

where each a is a unit, all summing up to five. To verify (using the technique above) that five is not divisible by two, two strands are created:

$$s_i = \mathtt{aaaaax}$$

$$s_r = \overline{\mathtt{aax}}\mathtt{x} = \bar{\mathtt{x}}\bar{\mathtt{a}}\bar{\mathtt{a}}\mathtt{x}\ .$$

Note that the strand s_i contains five $\mathtt{a}$ domains, and the strand s_r two $\bar{\mathtt{a}}$ domains. A large number of strands s_i and s_r are mixed in the soup. The last three domains of type s_i strands and the first three domains of type s_r strands will bind, forming a new (effective) strand

$$s_t = \mathtt{aaax}\ .$$

This new strand of type s_t (and all the strands of type s_t) will also be able to bind with strands of type s_r, forming a new strand

$$s'_t = \mathtt{ax}\ .$$

The strand s_i represents the number n (five in this case), s_r the subtraction by m (two in this case), and the strands s_t and s'_t represent the variable r at different steps of the algorithm.[1]

This process can be extended to any natural numbers n and m, as follows ($\mathtt{b}^\mathtt{k}$ represents the domain $\mathtt{b}$ repeated k times):

- Represent the number n with a strand $s_i = \mathtt{a}^\mathtt{n}\mathtt{x}$.

[1]The strands s'_t and s_r could further bind, producing strands of the form $\overline{\mathtt{ax}}$, which would not bear any meaning in regard to this algorithm (not even that of -1), and the problem of preventing them from being created will be addressed later. For now, they will be ignored, as will strands resulting from bonds not involving an $\mathtt{x}$ domain.

- Represent the *subtraction-by-m* steps with a strand $s_r = \overline{\mathtt{x}\mathtt{a}^{\mathtt{m}}}\mathtt{x}$.
- If at any time the *strand* `x` is observed, representing a remainder equal to zero, then n is divisible by m.
- If at no time the *strand* `x` is observed, n is not divisible by m.

This algorithm effectively calculates p, the modulo of n by m ($p = (n \bmod m)$). Then, from observing a modulo equal to zero, one can infer the divisibility of n by m. Note that if at any point p is observed, it can be inferred that the algorithm has reached a result; yet, in practice p is not known beforehand (that would make the algorithm trivial). In general, one could not know if bindings are still to occur, or if the m simply does not divide n.

1.3 Related Work

Since DNA was first used to solve a seven-city Hamiltonian path problem in [1], it remained an important topic of study in the field of biomolecular computation. The technique of strand displacement using toehold exchange was first fully described in [2], which allowed the design of simpler DNA circuits using DNA strands only (and thus disposing of restriction enzymes or deoxyribozymes which had been needed until then, as seen in [3]).

This led to the *seesaw gate* model in [4], allowing the design of arbitrary DNA circuits, and also led to the development of single DNA algorithms like the Approximate Majority in [5]. The goal of the model described in this paper is to be more scalable than the seesaw gate model, and have more applications than just the Approximate Majority algorithm. Over the past years, simulation tools have also been created to examine the behavior of DNA algorithms (e.g. in [6]), and were the catalyst for the research in this paper.

Chapter 2
Model

The technique of domain replacement presented in Section 1.1 can be used for more DNA algorithms than just the divisibility DNA algorithm. This chapter presents the Domain Replacement Model, a model that can be used to design DNA algorithms in a more general and abstract way. The last section focuses on problems that can arise and how to address them.

2.1 Domain Replacement Model

The Domain Replacement Model is a set of constraints on the strands which are present in the soup's initial state, before any binding has occurred. It ensures that any (effective) strand produced in the soup will be in the form of `<X>`, with $\texttt{X} \in \Sigma^*$, where `<`, `>` are domains used as delimiters, and `X` represent the data carried by the strands (e.g. the representation of the variable r in the divisibility DNA algorithm). The alphabet Σ contains the symbols used for those data. Such *symbols* are the model abstraction of domains, and are equivalent to domains.

Three strand types are allowed in the Domain Replacement Model:

- strands of the form `<XYZ>`, $\texttt{X}, \texttt{Y}, \texttt{Z} \in \Sigma^*$, which are referred to as *input strands*, and are represented underlined:

$$\underline{\texttt{<XYZ>}}\ .$$

- strands of the form $\overline{\texttt{Y>}}\texttt{Z>}$,[1] with $\texttt{Y}, \texttt{Z} \in \Sigma^*$, which are referred to as *rule strands*, and are represented as

$$\texttt{Y>} \rightarrow \texttt{Z>}\ .$$

- strands of the form $\overline{\texttt{<Y>}}\texttt{<Z>}$, $\texttt{Y}$, $\texttt{Z} \in \Sigma^*$, which are referred to as *output rules*, and are represented as

$$\texttt{<Y>} \rightarrow \texttt{<Z>}\ .$$

[1]Note that rule strands could also be formed to replace the left part of strands (instead of the right side as shown above). This would give such strands : $\texttt{<Y} \rightarrow \texttt{<Z}$.

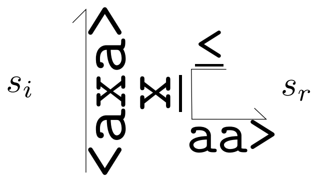

Figure 2.1: Three-legged strand.

Input strands are used to represent the input of a DNA algorithm, while the rule and output rule strands represent the operations performed on the input strands, and on any strand produced using those rules. These rule strands effectively replace a part of another strand by specific domains.

The alphabet Σ must also satisfy some conditions:

1. $\mathtt{a} \in \Sigma \iff \bar{\mathtt{a}} \in \bar{\Sigma}$
2. $\mathtt{<,>} \notin \Sigma, \bar{\Sigma}$
3. $\Sigma \cap \bar{\Sigma} = \emptyset$.

The condition (1) defines $\bar{\Sigma}$, the alphabet of the complement symbols, while (2) implies that delimiter symbols may not be used to represent data, but only at the extremities. Finally, the condition (3) avoids situations where non-delimiter symbols could bind.

The role of the delimiter domains `<` and `>` is to provide a *docking* location for the rule strands. For instance, the input strand $s_i = \underline{\ \mathtt{<axa>}\ }$ and the rule strand $s_r = \mathtt{x>} \to \mathtt{aa>} = \bar{\mathtt{>}}\bar{\mathtt{x}}\mathtt{aa>}$ can bind, which produces a strand that is not well-formed, as shown in Fig. 2.1. In practice this can be avoided by covering all the non-delimiter domains with single domain strands, prior to introducing the strands in the soup. This prevents any binding from occurring, except when two complementary delimiter domains are involved. It still lifts the covering single domain strands off one by one when the strands bind past the delimiter. For now it is assumed that binding can only occur if such a delimiter is involved.[2]

2.2 Composition Rules

The model presented thus far is very simple, and its results can easily be anticipated. Unfortunately, it is not very powerful; not enough even to allow aggregation. That is, at no point is it possible for a DNA algorithm to make such a decision:

[2]Note that two strands $s_1 = \mathtt{<x>}$ and $s_2 = \bar{\mathtt{>}}\bar{\mathtt{x}}\mathtt{>}$, for instance, can still bind on s_1's last symbol and s_2's first symbol. The produced strand is not well formed, but this should not cause any problems in practice.

If strands of both types s_A and s_B are observed, then produce strands of type s_C.

For instance, the presence of p (the modulo of n by m) in the divisibility algorithm presented earlier indicates that the algorithm has reached a final state. If that p is zero, then m divides n. If p is not equal to zero, m does not divide n. In theory, if it is possible to compute if any number m divides n, it should also be possible to find out if n is prime or not. This would be done by checking the divisibility of n by 2, 3, ... $\sqrt{n}$, yielding the results p_2, p_3, ..., $p_{\sqrt{n}}$. If no remainder p_2, p_3, ..., $p_{\sqrt{n}}$ is equal to zero, no number divides n, which implies it is prime[3].

The problem with the domain replacement model presented above is that it is never possible to check if two different strands have been derived; at least not on the DNA level. One could observe every strand one by one, and if strands of wanted types were observed, then one could give the algorithm (the soup) some new task, for instance by introducing a new strand. The rest of this section presents a way to introduce such a mechanism on the strand level, by only slightly broadening the model.

Collapsing rule The first step towards the general model is the introduction of the *collapsing rule*. *Collapsing rule strands* are strands of the form $\overline{\texttt{X}}\texttt{><}\overline{\texttt{Y}}$, with $\texttt{X}, \texttt{Y} \in \Sigma^*$, and are represented as

$$\texttt{X>} \bowtie \texttt{<Y} .$$

As an example, two strands $s_1 =$ `<hellox>` and $s_2 =$ `<yworld>` do not bind when they are together in a soup. However, with the addition of a strand $s_3 = \texttt{>}\overline{\texttt{x}}\overline{\texttt{y}}\texttt{<} = \texttt{x>} \bowtie \texttt{<y}$, the following happens: s_3's first two symbols bind with s_1, while its last two symbols bind with s_2. The whole process produces a new strand $s_4 =$ `<helloworld>`. The strand s_3 effectively helped the strands s_1 and s_2 collapse together, forming a new strand s_4 comprised of s_1's beginning and s_2's ending.

Simple composition rule Now, this can be slightly extended to allow the *simple composition*. *Simple composition rules* represent in fact a collection of the four strand types

1. $$\texttt{<A>} \rightarrow \texttt{<C}_\texttt{L}\texttt{X}_\texttt{L}\texttt{>}$$

[3]It is not necessary to test against numbers greater than $\sqrt{n}$. For $d, m \neq 1$, if m divides n, then $d = \frac{n}{m}$. The existence of d implies the existence of m, and vice versa, and also means that n is divisible (and hence not prime). Finding either d or m is sufficient, and if $d \leq m$ the greatest d can get is $\sqrt{n}$ (since $md = n$).

2.
$$\texttt{<B>} \rightarrow \texttt{<X}_{\texttt{R}}\texttt{C}_{\texttt{R}}\texttt{>}$$

3.
$$\texttt{X}_{\texttt{L}}\texttt{>} \bowtie \texttt{<X}_{\texttt{R}}$$

4.
$$\texttt{<C}_{\texttt{L}}\texttt{C}_{\texttt{R}}\texttt{>} \rightarrow \texttt{<C>}$$

and are represented as
$$\texttt{<A>} + \texttt{<B>} \rightarrow \texttt{<C>}\ .$$

Example For instance, to get a rule of the form

$$\texttt{<one>} + \texttt{<two>} \rightarrow \texttt{<helloworld>}$$

it is sufficient to add the two following rules to the above soup:

$$\texttt{<one>} \rightarrow \texttt{<hellox>}$$

$$\texttt{<two>} \rightarrow \texttt{<yworld>}\ .$$

The presence of `<one>` allows the production of `<hellox>`, while the presence of `<two>` allows the productions of `<yworld>`. Using the collapsing rule, `<hellox>` and `<yworld>` produce `<helloworld>`.

Note that the *simple composition rule* construction could be simplified using complementary symbols:
$$\texttt{<A>} \rightarrow \texttt{<C}_{\texttt{L}}\texttt{X>}$$

$$\texttt{<B>} \rightarrow \texttt{<}\bar{\texttt{X}}\texttt{C}_{\texttt{R}}\texttt{>}$$

which effectively does the same as $\texttt{<A>} + \texttt{<B>} \rightarrow \texttt{<C>}$, using less rules. But using the $\bar{\texttt{X}}$ symbol does not fit in the model, so it is better avoided unless a smaller number of rules is really needed.

Composition The composition does not have to be restricted to two inputs and one output. Using a composition of different simple composition rules, one can have:
$$\texttt{<A}_1\texttt{>} + \texttt{<A}_2\texttt{>} \rightarrow \texttt{<B}_1\texttt{>}$$

$$\texttt{<B}_1\texttt{>} + \texttt{<A}_3\texttt{>} \rightarrow \texttt{<B}_2\texttt{>}$$

$$\texttt{<B}_2\texttt{>} + \texttt{<A}_4\texttt{>} \rightarrow \texttt{<B}_3\texttt{>}$$

$$\vdots$$

$$\texttt{<B}_{\texttt{N - 1}}\texttt{>} + \texttt{<A}_{\texttt{N}}\texttt{>} \rightarrow \texttt{<C>}$$

which leads to a cascade of compositions, stopping if one of the $\texttt{<A}_\texttt{i}\texttt{>}$ is missing. This cascade is equivalent to the following:

$$\texttt{<A}_1\texttt{>} + \texttt{<A}_2\texttt{>} + \ldots + \texttt{<A}_\texttt{N}\texttt{>} \rightarrow \texttt{<C>}\ .$$

This can be used to produce *(full) composition rules*, which require the strand types

$$\begin{gathered}
\texttt{<A}_1\texttt{>} + \texttt{<A}_2\texttt{>} + \ldots + \texttt{<A}_\texttt{N}\texttt{>} \rightarrow \texttt{<B>} \\
\texttt{<B>} \rightarrow \texttt{<C}_1\texttt{>} \\
\texttt{<B>} \rightarrow \texttt{<C}_2\texttt{>} \\
\vdots \\
\texttt{<B>} \rightarrow \texttt{<C}_\texttt{M}\texttt{>}
\end{gathered}$$

and are represented as

$$\texttt{<A}_1\texttt{>} + \texttt{<A}_2\texttt{>} + \ldots + \texttt{<A}_\texttt{N}\texttt{>} \rightarrow \texttt{<C}_1\texttt{>} + \texttt{<C}_2\texttt{>} + \ldots + \texttt{<C}_\texttt{M}\texttt{>}\ .$$

Example Consider the soup containing the two following rules:

$$r_1 : \texttt{<wood>} + \texttt{<tools>} \rightarrow \texttt{<chair>} + \texttt{<table>}$$

$$r_2 : \texttt{<wood>} + \texttt{<fire>} \rightarrow \texttt{<smoke>}$$

where r_1 is itself comprised of four rules, which together effectively work as a full composition rule:

$$\begin{gathered}
r_{1,1} = \texttt{<wood>} + \texttt{<tools>} \rightarrow \texttt{<B>} \\
r_{1,2} = \texttt{<wood>} + \texttt{<tools>} \rightarrow \texttt{<B>} \\
r_{1,3} = \texttt{<B>} \rightarrow \texttt{<chair>} \\
r_{1,4} = \texttt{<B>} \rightarrow \texttt{<table>}\ .
\end{gathered}$$

Note that the rules

$$r'_{1,1} = \texttt{<wood>} + \texttt{<tools>} \rightarrow \texttt{<chair>}$$

$$r'_{1,2} = \texttt{<wood>} + \texttt{<tools>} \rightarrow \texttt{<table>}\ .$$

have the same effect as r_1 as well, and can be used in place of $r_{1,1}$, $r_{1,2}$, $r_{1,3}$ and $r_{1,4}$ if having a smaller number of rules is more important than having rules of shorter length.

If now a large number of input strands of the types `<wood>` and `<tools>` were added to the soup containing the two rules r_1 and r_2, one could observe strands of types `<chair>` and `<table>` being produced over time. There will never be strands of type `<smoke>`, because no strand of type `<fire>` is present; such strands of type `<fire>` would be necessary for the rule r_2 to produce strands of type `<smoke>`.

2.3 Strand Design

Sometimes, the model as described above will not be enough to enforce well-formed DNA strands production throughout the DNA computation. The rest of this chapter will highlight a few subtleties one has to be aware of when designing a DNA algorithm.

2.3.1 Phantom Rule

The phantom rule is a rule effectively acting in a DNA algorithm, but that wasn't part of the design. For instance, the DNA algorithm

$$r_1 = \texttt{YX>} \to \texttt{Z>}$$

$$r_2 = \texttt{Z>} \to \texttt{YW>}$$

with input

$$s_i = \underline{\texttt{<WX>}}$$

will produce strands of type

$$s_o = \texttt{<WW>}\ ,$$

even though none of the rule strands should be able to bind with the input s_i. To see why s_o is produced, the rules r_1 and r_2 are first rewritten as

$$r_1 = \texttt{YX>} \to \texttt{Z>} = \bar{\texttt{>}}\bar{\texttt{X}}\bar{\texttt{Y}}\texttt{Z>}$$

$$r_2 = \texttt{Z>} \to \texttt{YW>} = \bar{\texttt{>}}\bar{\texttt{Z}}\texttt{YW>}$$

where it is now clear that r_1 and r_2 can bind over r_1's last three domains and r_2's first three domains, forming the strand

$$r_p = \bar{\texttt{>}}\bar{\texttt{X}}\texttt{W>} = \texttt{X>} \to \texttt{W>}$$

which is the phantom rule.

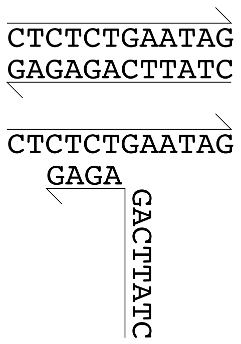

Figure 2.2: A shift at the nucleotide level.

2.3.2 Nucleotide-scale Shift

Depending on which nucleotides are chosen for encoding the domains $\mathtt{X}, \mathtt{Y}, \ldots \in \Sigma^*$, shifted bindings can occur, where symbols do not start at the same index on both strands. For instance, if the encoding

$$\mathtt{x} = CTGA \implies \bar{\mathtt{x}} = TCAG$$

$$\mathtt{<} = CTCT \implies \bar{\mathtt{<}} = AGAG$$

$$\mathtt{>} = ATAG \implies \bar{\mathtt{>}} = CTAT$$

is chosen and if the strands $s_1 = \mathtt{<x>}$ and $s_2 = \overline{s_1} = \bar{\mathtt{>}}\bar{\mathtt{x}}\bar{\mathtt{<}}$ are introduced together in a soup, then two bindings can occur. The first binding, from the first to the twelfth nucleotide on strand s_1 and from the twelfth nucleotide to the first on strand s_2, is the expected binding, producing an empty strand. The second binding is from the third to the tenth nucleotide on strand s_1 and from the tenth to the first nucleotide on strand s_2, and will produce two malformed strands.

This happens because the binding is shifted, as shown in Fig 2.2. This shift can lead to unexpected behaviors of the DNA algorithm. It can be avoided by verifying that nucleotide encodings never overlap for any domain. For instance, the encoding for every domain could begin with a starting sequence of repeated As (and ending with a sequence of Ts for the complementary domains), signifying the beginning of a new domain.

Chapter 3

DNA Algorithms

When designing DNA-algorithms, one of the goals is to keep the number of rules involved low. There are several reasons to that:

- the more rules involved (and actually the more strand types present in the soup), the more different by-products will be produced, potentially binding together and leading to false positives. If the strand concentration is too high, it also prohibits the overall freedom of movement, and slows down the process.
- the production of n strands of different types is much more difficult than the production of n strands of the same type.
- if the number of rule strand types is of the order of the number of inputs the algorithm can be executed on, the algorithm could simply be replaced by another trivial one, where every input is mapped to its expected output using *output rules*. The results would need to be computed in advance, but complexity in terms of the number of strand types will be the same.

The latter reason gives a condition on the number of strand types involved in a DNA-algorithm: for an algorithm involving N_s initial strand types and accepting N_i different inputs, the following should hold:

$$N_s \in o(N_i)\ .$$

To represent the number of inputs that can be expressed, N_i is defined as follows:

- In case of inputs expressed in a unary form, $N_i = n$, where n is both the length of the input (without delimiters) and the number being expressed.
- In case of binary inputs (as will be the case in 3.3), $N_i = 2^p > n$, where n is the number being expressed, and p is the highest power of two needed to express n.

3.1 Domain Replacement Model Algorithms

3.1.1 Divisibility

The divisibility algorithm encountered earlier satisfies the condition $N_s \in o(N_i)$ since it has a constant number of rules:

Divisibility Algorithm 1

$\underline{\texttt{<a}^{\texttt{n}}\texttt{>}}$

$\texttt{a}^{\texttt{m}}\texttt{>} \rightarrow \texttt{>}$

$\texttt{<>} \rightarrow \texttt{<'divisible'>}$

There is yet one flaw with this algorithm. The only output that can be observed is `divisible`. By only checking if the remainder is zero, the algorithm lacks the ability to assess that n is *not* divisible by m. This is easily fixed by adding output rules that state that n is not divisible by m if the remainder is greater than zero; that is, for m not diving n, it holds that $r \in \{1, 2, ..., m - 1\}$, where r is the remainder $r = (n \mod m)$. That way, one can know that the algorithm has reached a result, either if `divisible` or `not divisible` is observed.

Divisibility Algorithm 2

$\underline{\texttt{<a}^{\texttt{n}}\texttt{>}}$

$\texttt{a}^{\texttt{m}}\texttt{>} \rightarrow \texttt{>}$

$\texttt{<>} \rightarrow \texttt{<'divisible'>}$

$\forall i = 1, 2, 3, ..., m - 1 :$

$\texttt{<a}^{\texttt{i}}\texttt{>} \rightarrow \texttt{<'not divisible'>}$

With one *subtraction-by-m* rule, one output rule if the remainder is zero and $m - 1$ output rules for a non-zero remainder, the number of rules is equal to $1 + 1 + m - 1 = m + 1$. Since m is fixed for a given algorithm, it holds that $N_s \in o(N_i)$.

3.1.2 Primality

A prime number is a number that is only divisible by itself and by one; it then makes sense to investigate how the primality of a number can be determined using the divisibility technique used above. For the primality of n, the divisibility has to be checked for every number from two to the square root of n. To do so, the strands are first marked[1] using the rule `a>` $\rightarrow$ `ab^m>`, for different $m = 2, 3, ..., \lfloor\sqrt{n}\rfloor$.[2] Those are the possible smallest dividers. Marking is necessary in order to be able to always subtract the same m.

Primality Algorithm 1

$$\underline{\texttt{<a}^{\texttt{n}}\texttt{>}}$$

$\forall m = 2, 3, ..., \lfloor\sqrt{n}\rfloor$:

$$\texttt{a>} \rightarrow \texttt{ab}^{\texttt{m}}\texttt{>}$$

$$\texttt{a}^{\texttt{m}}\texttt{b}^{\texttt{m}}\texttt{>} \rightarrow \texttt{b}^{\texttt{m}}\texttt{>}$$

$$\texttt{<b}^{\texttt{m}}\texttt{>} \rightarrow \texttt{<'not prime'>}$$

When a large number of strands of the input strand type $\underline{\texttt{<a}^{\texttt{n}}\texttt{>}}$ is released into the soup, some will be marked with $m = 2$, some with $m = 3$, etc. The rest of the algorithm will run independently for each strand, in parallel of the other ones. The strands $\texttt{a}^{\texttt{m}}\texttt{b}^{\texttt{m}}\texttt{>} \rightarrow \texttt{b}^{\texttt{m}}\texttt{>}$ are the *subtraction-by-m* steps, with the only difference to the divisibility algorithm being that they keep track of the marker $\texttt{b}^{\texttt{m}}$.

For each possible divider m, three rules are added, so the number of rules is equal to $3(\sqrt{n} - 1)$. It thus holds that $N_s \in o(N_i)$. As with the first divisibility algorithm, a result can only be observed if some m divides n, in this case meaning that n is not prime. The next section will show how to alleviate this issue.

3.2 Using the General Model

To allow the primality algorithm to give a result also for numbers that are prime (and not only for those that are not prime), all the possible remainders from the divisibility algorithm need again to be taken into account. Since a non-zero remainder means that n is not divisible by m, primality exists only if *all* the remainders are different from zero. To infer primality the composition rule is used, producing the result `'prime'` only if all the strands `'!m|n'` are present.[3]

[1]Here, the marker used is the symbol `b` repeated m times. Since for a given m the marker will never change, any symbol (or symbol sequence) unique with respect to m can be used too. This produces shorter strands, to the expense of using more different symbols.

[2]The floor function of the square root is used, since it holds that $m \leq \sqrt{n}$, as mentioned in Section 2.2.

[3]Here the notation `!m|n` is used for m *does not divide* n.

Primality Algorithm 2

<ins>`<a^n>`</ins>

$\forall m = 2, 3, ..., \lfloor\sqrt{n}\rfloor$:

$$\texttt{a>} \rightarrow \texttt{ab}^{\texttt{m}}\texttt{>}$$

$$\texttt{a}^{\texttt{m}}\texttt{b}^{\texttt{m}}\texttt{>} \rightarrow \texttt{b}^{\texttt{m}}\texttt{>}$$

$$\texttt{<b}^{\texttt{m}}\texttt{>} \rightarrow \texttt{<'not prime'>}$$

$\forall i = 1, 2, ..., m - 1$:

$$\texttt{<a}^{\texttt{i}}\texttt{b}^{\texttt{m}}\texttt{>} \rightarrow \texttt{<'!m|n'>}$$

Finally :

$$\texttt{<'!2|n'>} + \texttt{<'!3|n'>} + ... + \texttt{<'!sqrt(n)|n'>} \rightarrow \texttt{<'prime'>}$$

Here, counting the number of strands is a bit more difficult than for the previous DNA algorithms. As for the previous DNA algorithm, three rules of the form `X>` $\rightarrow$ `Y>` are added for each possible divider m; additionally, for each m, $m - 1$ rules of this form are also added. This already leads to a total number of rules of

$$N_s' = (3 + 1) + (3 + 2) + ... + (3 + \sqrt{n} - 1) = 3\sqrt{n} + \sum_{m=1}^{\sqrt{n}} m - 1$$
$$= 3\sqrt{n} + \sum_{i=0}^{\sqrt{n}-1} i = 3\sqrt{n} + \frac{(\sqrt{n} - 1)\sqrt{n}}{2}$$

where $i = m - 1$.

Moreover, the last composition rule is itself comprised of multiple number of simple rules, namely $\sqrt{n} - 2$. This means that the total number of strands in the soup is

$$N_s = 3\sqrt{n} + \frac{(\sqrt{n} - 1)\sqrt{n}}{2} + \sqrt{n} - 2 = \frac{1}{2}(n + 7\sqrt{n} - 4)$$

which unfortunately is not in $o(N_i)$, and thus could be replaced by a DNA algorithm with less rules, which could map any input to its expected result. Still, this would require computing all the results before coding them into the DNA algorithm, which is not necessary in the *Primality Algorithm 2*.

3.2.1 Square Root

For the two previous DNA algorithms to work for input numbers up to n, the value of $\sqrt{n}$ needs to be known in advance. This is one of the reasons why it is interesting to be able to compute a square root with a DNA algorithm. The algorithm

```
$x \leftarrow n$
$k \leftarrow 1$
**while** $x \neq k$ **do**
    $x \leftarrow x - 2k$
    $k \leftarrow k + 1$
**end while**
**return** $k$
```

computes the square root of a perfect square n. To see that, one can use the following relation

$$1 + 2 + 3 + ... + q = \sum_{k=1}^{q} k = \frac{q(q+1)}{2} = \sigma$$

which can be rewritten as

$$2\sigma = q^2 + q \iff q^2 = \Big(\sum_{k=1}^{q} 2k\Big) - q = \Big(\sum_{k=1}^{q-1} 2k\Big) + q$$

and finally

$$\sqrt{n} = n - \Big(\sum_{k=1}^{\sqrt{n}-1} 2k\Big), \text{ with } n = q^2 \ .$$

The algorithm returns a result if at any point $x = k$, that is if at any point

$$x = n - \Big(\sum_{\tilde{k}=1}^{k-1} 2\tilde{k}\Big) = k$$

which is true only if $k = \sqrt{n}$. The DNA algorithm is implemented as follows:

Unary square root

$\underline{\texttt{<a}^{\texttt{n}}\texttt{>}}$

$\texttt{a>} \rightarrow \texttt{ab>}$

$\forall i = 1, 2, ..., \sqrt{n} - 1 :$

$\texttt{a}^{\texttt{2i}}\texttt{b}^{\texttt{i}}\texttt{>} \rightarrow \texttt{b}^{\texttt{i+1}}\texttt{>}$

$\texttt{<a}^{\texttt{i}}\texttt{b}^{\texttt{i}}\texttt{>} \rightarrow \texttt{<'sqrt(n) = i'>}$

where strands are equivalent to $\texttt{<a}^{\texttt{x}}\texttt{b}^{\texttt{k}}\texttt{>}$. The number of strands in the soup is $N_s = 2(\sqrt{n}-1)+1 \in o(N_i)$. Thus, with rules for a fixed n, such a DNA algorithm can be used to compute the square root of any perfect square $\tilde{n} \leq n$.

As with *Primality Algorithm 1*, there are numbers for which the algorithm does not produce any output, namely for non-perfect squares. Producing outputs for those numbers too would require (as in the primality DNA algorithms) adding many new rules, and might leave the condition $N_s \in o(N_i)$ unfulfilled. To alleviate this issue, the next section introduces binary inputs, which can express more different numbers for input strand types of a given length.

3.3 Binary Inputs

So far, the input was limited to a unique strand type; this is not a necessity. As seen in Section 2.2, it is possible to implement aggregation with rules of the type `<A> + <B>`→`<C>`. This, among others, means that the input need not be unique, and that different input strand types can be processed separately, still giving a common result.

One of the possibilities this gives rise to is an easy, modular binary representation of the input number; modular because the different powers of the represented number are not tied together. The design is easy: since each number n can be expressed in base two as $(a_p a_{p-1} ... a_1 a_0)_b$, with $n = a_p 2^p + a_{p-1} 2^{p-1} + ... + a_1 2^1 + a_0 2^0$, each bit a_i can be represented using a specific strand type (the subscript b stands for "written in binary form") of the form $\texttt{<a}_\texttt{i}\texttt{>}_\texttt{i}$. The power of two to which the bit belongs is encoded with a special right delimiter symbol, but it could also be added as its own extra symbol between the two delimiters.

For instance, the number $6 = 110_b$ is represented as:

$$6 \equiv \left\{ \underline{\texttt{<1>}_2},\ \underline{\texttt{<1>}_1},\ \underline{\texttt{<0>}_0} \right\} \ ,$$

where $\texttt{<1>}_2$, $\texttt{<1>}_1$ and $\texttt{<0>}_0$ are three different strand types.

The first algorithm that will be derived is a simple addition, on which the other DNA algorithms will be based. Adding two numbers $a+b=r$ in their binary forms is done just the way two numbers are added in decimal form: same powers are added together, and if they exceed the base (10 for decimal), a so-called *carry* is added to the next power. Each possible addition resulting in a power of r can be expressed with a rule

$$\texttt{<a}_\texttt{i}\texttt{>}^{\texttt{a}}_{\texttt{i}} + \texttt{<b}_\texttt{i}\texttt{>}^{\texttt{a}}_{\texttt{i}} + \texttt{<c}_\texttt{i-1}\texttt{>}^{\texttt{c}}_{\texttt{i-1}} \rightarrow \texttt{<r}_\texttt{i}\texttt{>}^{\texttt{r}}_{\texttt{i}} + \texttt{<c}_\texttt{i}\texttt{>}^{\texttt{c}}_{\texttt{i}}$$

where the outputs r_i and c_i are given below, giving eight composition rules in total per power (though only four for $i=0$, because c_{-1} is null), as seen in Table 3.1.[4]

[4]Though each power needs eight composition rules, each of those composition rules itself is

a_i	b_i	c_{i-1}	r_i	c_i	a_i	b_i	c_{i-1}	r_i	c_i
0	0	0	0	0	1	1	0	0	1
1	0	0	1	0	1	0	1	0	1
0	1	0	1	0	0	1	1	0	1
0	0	1	1	0	1	1	1	1	1

Table 3.1: Relations between operands, carry and result bits for an addition.

a_i	b_i	c_{i-1}	r_i	c_i	a_i	b_i	c_{i-1}	r_i	c_i
0	0	0	0	0	1	1	0	0	0
1	0	0	1	0	1	0	1	0	0
0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1

Table 3.2: Relations between operands, carry and result bits for a subtraction.

Note that the right delimiter symbol bears both the power and the variable to which the bit belongs (to a, b, c or r). This need not be done using a special delimiter symbol like here; it could also be achieved with an extra symbol within the delimiters. The subtraction $r = a - b$ can be achieved too, with the relations given in Table 3.2.

Example The input for the addition $r = a + b, a = 2, b = 1$ is represented as follows:

$$a = 2 \equiv \left\{\underline{\texttt{<0>}^{\texttt{a}}_{\texttt{2}}},\ \underline{\texttt{<1>}^{\texttt{a}}_{\texttt{1}}},\ \underline{\texttt{<0>}^{\texttt{a}}_{\texttt{0}}}\right\} \ ,$$
$$b = 1 \equiv \left\{\underline{\texttt{<0>}^{\texttt{b}}_{\texttt{2}}},\ \underline{\texttt{<0>}^{\texttt{b}}_{\texttt{1}}},\ \underline{\texttt{<1>}^{\texttt{b}}_{\texttt{0}}}\right\} \ ,$$

The first rule that is going to trigger is the rule

$$\texttt{<0>}^{\texttt{a}}_{\texttt{0}} + \texttt{<1>}^{\texttt{a}}_{\texttt{0}} \to \texttt{<1>}^{\texttt{r}}_{\texttt{0}} + \texttt{<0>}^{\texttt{c}}_{\texttt{0}}$$

producing the least significant bit of the result and a null carry. Here the previous power's carry c_{i-1} is omitted, since there is no bit with power $i = -1$.

Now, the presence of a strand of type $\texttt{<c}_{\texttt{0}}\texttt{>}^{\texttt{c}}_{\texttt{0}}$ will trigger the rule computing the result's bit of power 2^1. Since $a_1 = 1, b_1 = 0, c_0 = 0$ the rule triggered is

$$\texttt{<1>}^{\texttt{a}}_{\texttt{1}} + \texttt{<0>}^{\texttt{b}}_{\texttt{1}} + \texttt{<0>}^{\texttt{c}}_{\texttt{0}} \to \texttt{<1>}^{\texttt{r}}_{\texttt{1}} + \texttt{<0>}^{\texttt{c}}_{\texttt{1}}$$

Since there are no strands of the form $\texttt{<a}_{\texttt{2}}\texttt{>}^{\texttt{a}}_{\texttt{2}}$ or $\texttt{<b}_{\texttt{2}}\texttt{>}^{\texttt{b}}_{\texttt{2}}$, no other rule will trigger, and the algorithm has reached the result

comprised of four simple rules, i.e. an addition adds 32 strands to the soup.

$$\{\texttt{<1>}^{\texttt{r}}_{1},\ \texttt{<1>}^{\texttt{r}}_{0}\} \equiv 3 = r\ .$$

This result could then be used for further calculations, since it has the same form as a or b above.

3.3.1 Square Root in Binary

To use the technique described above to compute a square root, this time with DNA binary inputs, different x_k's have to be used for each value the variable x holds.

Binary Square Root

$$\underline{\texttt{<n}_{\texttt{p}}\texttt{>}},\ \underline{\texttt{<n}_{\texttt{p-1}}\texttt{>}}, ..., \underline{\texttt{<n}_{\texttt{0}}\texttt{>}}$$

$\forall k = 1, 2, ..., \sqrt{n}$:

$$\texttt{x}_{\texttt{k+1}} = \texttt{x}_{\texttt{k}} - 2k$$

$$\texttt{<k}_{\texttt{p}}\texttt{>}^{\texttt{x}_{\texttt{k}}}_{\texttt{p}} + \texttt{<k}_{\texttt{p-1}}\texttt{>}^{\texttt{x}_{\texttt{k}}}_{\texttt{p-1}} + ... + \texttt{<k}_{\texttt{0}}\texttt{>}^{\texttt{x}_{\texttt{k}}}_{\texttt{0}} \rightarrow \texttt{<'sqrt is k'>}$$

Here, p is the highest power of 2 needed to express the input number n. As mentioned above, each subtraction adds 32 strands per power of 2, giving

$$N_s = \sqrt{n}(p-1)32 + p$$

where an extra p strands come from the composition rule. From the definition of N_i for binary inputs it follows that

$$N_s < \sqrt{N_i}(\log_2 N_i - 1)32 + \log_2 N_i \in o(N_i) \implies N_s \in o(N_i)\ .$$

Chapter 4

Analysis

Two different process models are analyzed, each dealing with different aspects of an algorithm's complexity. The first process model, M1, only takes into account the presence or absence of strand types in the soup at a given time. The second model, M2, also takes into account the number of strands of each strand type that are present in the soup.

4.1 M1

The process model M1 assumes that the soup configuration moves forward in time in discrete steps, from state to state. A state S_k is a set of all the strand types present in the soup at step k. It is also assumed that if two strands s_x and s_y are present at step k in the state S_k and can bind to form a new strand s_z, all three strands s_x, s_y and s_z will be present at in the set S_{k+1}.

A transition represents a possible binding between two strand types. A transition is a transformation that takes two strands and produces a third (different) one. Transitions $t_a, t_b, ...$ are arcs between two states S_k and S_{k+1}. The set of all transitions between states S_k and S_{k+1} is T_k. An example of states and transitions is represented in Fig. 4.1 and discussed at the end of the section.

$$S_1 \xrightarrow{T_1} S_2 \xrightarrow{T_2} ... \xrightarrow{T_{N-1}} S_N$$

The *diversity at step k* is defined as

$$d_k = |S_k|$$

and represents the number of different strand types present in the soup at step k. The *width at step k* is defined as

$$w_k = |T_k|$$

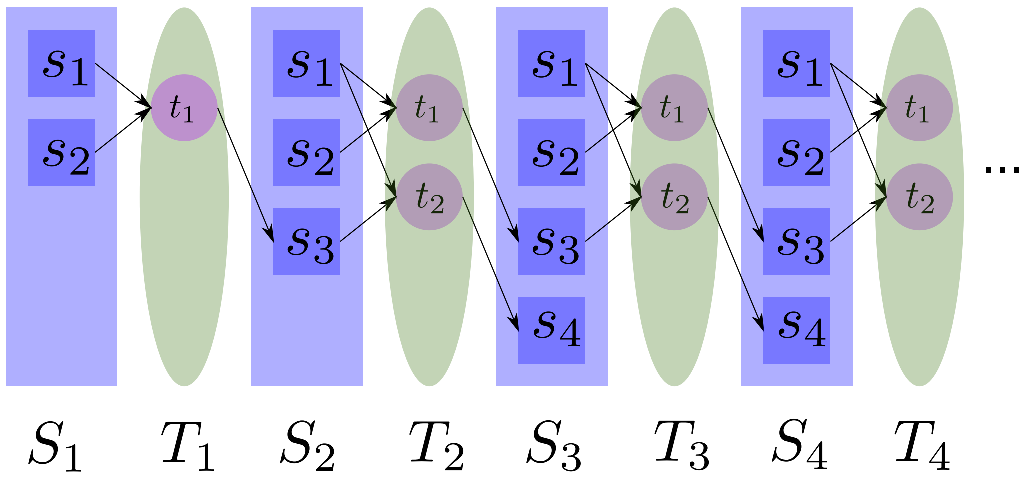

Figure 4.1: A representation of the M1 process model. The number of strands s_i within each state S_k is d_k, the *diversity* at step k. The number of transitions t_i in T_k is w_k, the *width* at step k.

and represents the number of different transitions from state S_k to state S_{k+1}. The diversity (or maximum diversity) is defined as

$$d = \max_k d_k$$

and represents the number of different strand types that can exist in the algorithm. Similarly, the width (or maximum width) is defined as

$$w = \max_k w_k$$

and represents the number of different bindings between any two strand types. The diversity is a measure of how many different types of strands are present and could potentially bind together; a high diversity could indicate more chances for unplanned bindings to occur and interfere with the execution of the DNA algorithm. The width is a measure of the parallelism of the process; a small width could indicate a bottleneck in the execution of the DNA algorithm.

The set S is the set comprised of all the strand types that can exist in any state:

$$S = S_1 \cup S_2 \cup ... = \{s_1, s_2, ..., s_d\}$$

as the set A is the set comprising all the pairs of strand types that can bind together:

$$A \subseteq S \times S \; .$$

The set T is the set comprised of all the transitions from any state S_k to state S_{k+1}:

$$T = T_1 \cup T_2 \cup ... = \{t_1, t_2, ..., t_w\}$$

and it thus follows that

$$t(s_a, s_b) = t_i \in T \iff (s_a, s_b) \in A$$

where t is defined as the relation between strand types and transitions:

$$t : A \to T$$

and represents which (if any) transition exists for two given strands. A transition $t_i = t(s_a, s_b)$ is in T_k if and only if the strand types s_a and s_b are present in S_k, that is

$$t_i \in T_k \iff s_a, s_b \in S_i$$

and produces strands of type $s_c = \Gamma(s_a, s_b) = \Gamma(s_b, s_a) \in S_{k+1}$. The function Γ defines the strand production for a transition:

$$\Gamma : A \to S$$

and it follows from the model description that only one transition can exist for two given strand types, and hence the symmetry in the parameters. This also means that to any transition from S_k to S_{k+1} must correspond a unique pair of strand types, so the number of transitions from S_k to S_{k+1} cannot be greater than the number of combinations of strand types of S_k:

$$w_k \leq \frac{d_k(d_k - 1)}{2} \ .$$

A few noteworthy considerations:

- At each state S_i , it is only known what strand types are present, not how many strands of each type are present in the soup.
- If a strand type exists in state S_i, it follows that it will exist in $S_{i+1}, S_{i+2}, \ldots$.

The algorithm could reach a point where no new strand type can be produced; that is, no binding is possible between any two strands produced during the last transitions. If defined, k_{eq} is the step at which the soup reaches that *equilibrium*, defined as

$$k_{eq} = \min k \text{ s.t. } S_k = S_{k+1} \ .$$

Note that because of how the states $S_k, S_{k+1}, \ldots$ are defined it follows that

$$S_{k_{eq}} = S_{k_{eq}+1} = \ldots = S_{k_{eq}+n}, \forall n \geq 0 \ .$$

From the definition of the states S_k, it follows that $S_k \supseteq S_{k-1} \supseteq \ldots \supseteq S_1$ and it then holds that

$$w_{k+1} \geq w_k$$

$$d_{k+1} \geq d_k$$

and

$$w = w_{k_{eq}}$$

$$d = d_{k_{eq}}$$

and from above

$$w \leq \frac{d(d-1)}{2} \ .$$

Example Strand types are designed such that s_1 can bind with s_2 to produce a strand of type s_3, and s_1 can also bind with s_3, to produce s_4. Only two transitions are involved:

$$t_1 = t(s_1, s_2)$$

$$t_2 = t(s_1, s_3)$$

and

$$s_3 = \Gamma(s_1, s_2)$$

$$s_4 = \Gamma(s_1, s_3) \ .$$

This is the DNA algorithm represented in Fig. 4.1, with two strand types present initially: s_1 and s_2. Thus the first state is $S_1 = \{s_1, s_2\}$, implying $d_1 = 2$, and it holds that $T_1 = \{t_1\}$, which gives $w_1 = 1$.

Since t_1 produces strands of type s_3, the transition t_2 is possible from S_2, producing a new strand type s_4 for the next state, S_3. Also, because t_1 and t_2 are the only transitions (and thus forming T), the algorithm has reached an equilibrium, with

$$k_{eq} = 3$$

$$d = 4$$

$$w = 2$$

$$S = \{s_1, s_2, s_3, s_4\}$$

$$T = \{t_1, t_2\} \ .$$

4.2 M2

The process model M2 represents the steps in the execution of a DNA algorithm as states x_k, where x_k is defined by its strand type levels:

$$x_k = x(n_1, n_2, ..., n_d)$$

where n_i is the number of strands of type s_i. The DNA algorithm starts in state $x_0 = x(n_1^{(0)}, n_2^{(0)}, .., n_d^{(0)})$ where $n_i^{(0)}$ is the initial number of strands of type s_i. The set X is the set of all states:

$$X = \{x_1, x_2, ..., x_N\}$$

where N is the total number of states. The jumps between two states are modeled using Poisson processes, where $q_{i,j}$ is the rate from state x_i to state x_j. A jump can only occur if it mirrors a transition, and thus

$$q_{i,j} \neq 0 \iff q_{i,j} = f(n_a(x_i), n_b(x_i)) \iff \exists! t_k = t(s_a, s_b)$$

with

$$\begin{aligned} n_a(x_j) &= n_a(x_i) - 1 \\ n_b(x_j) &= n_b(x_i) - 1 \\ n_c(x_j) &= n_c(x_i) + 1 \end{aligned}$$

where

$$s_c = \Gamma(s_a, s_b)\ .$$

It is assumed that, if the rate is not equal to zero, it then only depends on the number of strands present that are of the consumed types.

4.3 Divisibility Algorithm Analysis

The rest of this chapter analyses the first divisibility algorithm using the two models introduced. The first analysis uses M1, and these results will serve as a base for the M2 analysis.

4.3.1 M1 Analysis

The output rule `<>` $\rightarrow$ `<'divisible'>` can only be used when a strand of type `<>` is present in the soup. Thus, the output rule will be omitted in this analysis, which will focus on the apparition of the strand `<>`; the result strand `<'divisible'>` is then only one step ahead. In order to refer easily to the strand types, the ones representing a number (like n, $n-m$, $n-2m$) will be referenced as

$$s_{i+1} = \texttt{<a}^{\texttt{(n-im)}}\texttt{>}$$

and the unique rule will be represented as

$$s_r = \texttt{a}^{\texttt{m}}\texttt{>} \to \texttt{>} \ .$$

It is clear that at the beginning only two strands are present. The input s_1 and the rule s_r:

$$S_1 = \{s_1, s_r\} \ .$$

The only possible transition is t_1, consuming one input strand and a rule strand:

$$T_1 = \{t_1\}, t_1 = \Gamma(s_1, s_r), s_2 = T(s_1, s_r)$$

with s_2 representing the number $n-m$ as mentioned above. The second state is then simply

$$S_2 = \{s_1, s_2, s_r\} \ .$$

Now, not only s_1 can bind with the rule strand s_r, but so can s_2[1], giving a new transition:

$$T_2 = \{t_1, t_2\} \ .$$

Since every strand s_k binds in the same way with s_r, it generally gives:

$$S_k = \{s_1, s_2, ..., s_k, s_r\} \implies d_k = k+1$$

$$T_k = \{t_1, t_2, ..., t_k\} \implies w_k = k$$

$$t_k = \Gamma(s_k, s_r), \ s_{k+1} = t(s_k, s_r)$$

and this as long as s_k is defined, that is as long as $n - km \geq 0$. Thus it holds that

$$k_{eq} = \max\{k : n - km \geq 0\} = \left\lfloor \frac{n}{m} \right\rfloor \implies d = \left\lfloor \frac{n}{m} \right\rfloor + 1, \ w = \left\lfloor \frac{n}{m} \right\rfloor \ .$$

4.3.2 M2 Analysis

The M1 analysis does not infer that result strands will ever be produced. The M2 analysis partially answers that question with the notion of *hitting time*.

A hitting time (or first hit time) τ_j^B is the time it takes for the process to go from state x_j to any state $x \in B$. Thus the time between the strands' introduction into the soup (the initial state) to the first production of a result strand is

$$\tau = \tau_0^C$$

with

$$C = \{x : n_{k_{eq}}(x) > 0\}$$

$$x_0 = x(n_1^{(0)}, 0, ..., 0, n_r^{(0)}) \ .$$

[1]As long as $n - m \geq 0$.

Probability of producing a result strand The divisibility DNA algorithm does not necessarily reach a result in the M2 model. It could happen that all the rule strands of type s_r are consumed for the production of strands of type s_1, or for the production of any other type $s_i \neq s_{k_{eq}}$, where $s_{k_{eq}}$ is a result strand, as shown in the M1 analysis. The absence of result strands $s_{k_{eq}}$ when all the strands of type s_r have been consumed results in an infinite hitting time τ.

As shown in [7, p. 112], it holds that

$$\begin{aligned} h_i^C &= 1 \text{ for } i \in C \\ \sum_{j \in X} q_{i,j} h_j^C &= 0 \text{ for } i \notin C \end{aligned}$$

where

$$h_j^C = P(\tau_j^C < \infty) \ .$$

Also, since

$$q_{i,j} \neq 0 \iff q_{i,j} = f(n_k(x_i), n_r x_i)$$

with

$$\begin{aligned} n_k(x_j) &= n_k(x_i) - 1 \\ n_r(x_j) &= n_r(x_i) - 1 \\ n_{k+1}(x_j) &= n_{k+1}(x_i) + 1 \end{aligned}$$

the rates can also be rewritten as

$$q_{i,j} = g(n_k(x_i)) = \sigma(n_k(x_i))\sigma(n_r(x_i))f(n_k(x_i), n_r(x_i))$$

with $\sigma(n_k) = 1$ for $n_k > 1$ and $\sigma(n_k) = 0$ otherwise.

Generally, using $h(n) = h(n(x_i)) = h_i^C$, with $n(x_i) = (n_1(x_i), n_2(x_i), ..., n_d(x_i))$, it holds that

$$\sum_{j \in X} q_{i,j} h_j^C = \sum_{i=1}^{k_{eq}} g(n_k) h(\tilde{n}^{(k)}) - h(n) \sum_{k=1}^{k_{eq}} g(n_k) = 0$$

$$\iff h(n) = \frac{\sum_{k=1}^{k_{eq}} g(n_k) h(\tilde{n}^{(k)})}{\sum_{k=1}^{k_{eq}} g(n_k)}$$

with

$$\tilde{n}^{(k)} = (n_1, n_2, ..., n_{k-1}, n_k - 1, n_{k+1} + 1, n_{k+2}, ..., n_{k_{eq}}, n_r - 1) \ .$$

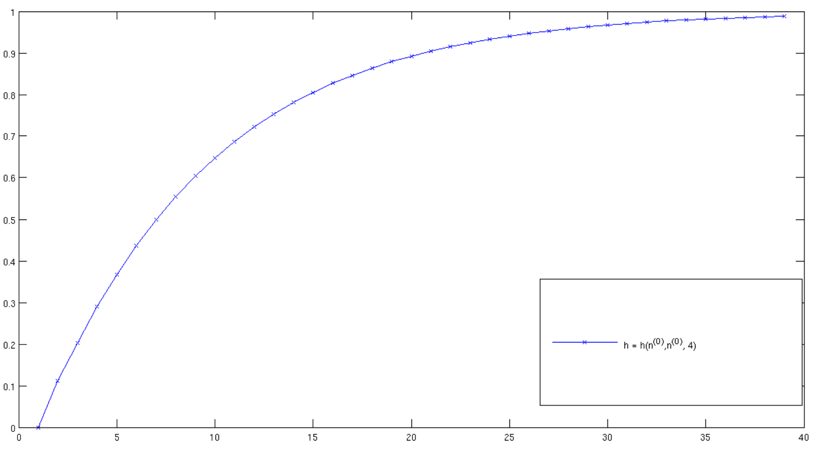

Figure 4.2: Representation of the probability $P(\tau < \infty)$, where τ is the time it takes the divisibility DNA algorithm to produce a result strand. In this case, $f(n_i, n_r) = n_i n_r$ and $k_{eq} = 4$.

For the divisibility DNA algorithm, the probability of producing a result strand is then $h(n^{(0)})$, with $n^{(0)} = (n_1^{(0)}, 0, ..., 0, n_r^{(0)})$. Its expression is recursive and non-linear, and thus an explicit form

$$h = h(n_1^{(0)}, n_r^{(0)}, k_{eq}) ,$$

if it exists, is not straightforward. Fig. 4.2 shows its evolution for $k_{eq} = 4$, $f(n_i, n_r) = n_i n_r$, and $n_1^{(0)} = n_r^{(0)}$ ranging from 1 to 40. It shows that for $n_1^{(0)} = n_r^{(0)} = 28$ the probability of producing a strand of type $s_{k_{eq}}$ is already above 95%. Note that multiplying $f(n_i, n_r)$ by a constant would not affect $h(n)$. Other functions for computing rates $(f(n_i, n_r))$ could be simulated using the same setting by just modifying the initial strand levels, and generally need not be linear in n_i or n_r.

4.4 Simulator

During the course of this research, a simple simulator was programmed. Rules can be specified, and the DNA algorithm can be simulated for a given input using either the model M1 or the model M2. Data like d, w, k_{eq} can then be gathered and analyzed further.

Bibliography

[1] Adleman, L.M.: Molecular computation of solutions to combinatorial problems. In: Science Magazine. (November 2009)

[2] Zhang, D.Y., Winfree, E.: Control of dna strand displacement kinetics using toehold exchance. In: J A C S Articles. (June 2009)

[3] Adleman, L.M.: A deoxyribozyme-based molecular automaton. In: Nat. Biotechnol. (September 2003)

[4] Qian, L., Winfree, E.: Scaling up digital circuit computation with dna strand displacement cascades. In: Science Magazine. (June 2011)

[5] Cardelli, L., Csikász-Nagy, A.: The cell cycle switch computes approximate majority. In: Scientific Reports. (September 2012)

[6] Lakin, M., Phillips, A.: Modelling, simulating and verifying turing-powerful strand displacement systems. In: International Conference on DNA Computing and Molecular Programming. (September 2011)

[7] Norris, J.: Markov Chains. Cambridge University Press (September 2004)