Toehold DNA Languages are Regular [Extended Abstract][*]

Sebastian Brandt, Nicolas Mattia, Jochen Seidel, and Roger Wattenhofer

ETH Zurich, Switzerland
{brandts, nmattia, seidelj, wattenhofer}@ethz.ch

Abstract. We explore a method of designing algorithms using two types of DNA strands, namely rule strands (rules) and input strands. Rules are fixed in advance, and their task is to bind with the input strands in order to produce an output. We present algorithms for divisibility and primality testing as well as for square root computation. We measure the complexity of our algorithms in terms of the necessary rule strands. Our three algorithms utilize a super-constant amount of complex rules.
Can one solve interesting problems using only few—or at least simple—rule strands? Our main result proves that restricting oneself to a constant number of rule strands is equivalent to deciding regular languages. More precisely, we show that an algorithm (possibly using infinitely many rule strands of arbitrary length) can merely decide regular languages if the structure of the rules themselves is simple, i.e., if the rule strands constitute a regular language.

1 Introduction

DNA is sometimes considered as an alternative to orthodox silicon-based technologies for computing. But how powerful can a DNA-based computer be? In this paper, we analyze the computational expressiveness of toehold DNA computing, c.f. [15,20]. In the toehold method, 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. Designing a DNA algorithm is equivalent to designing a set of DNA strands (the *rule strands*) such that, when strands representing an input (*input strands*) are added, a desired output, e.g., an indicator for YES or NO, is produced. An "execution" of a DNA algorithm corresponds to multiple steps of *strand binding*, where initially only rule and input strands bind, incrementally forming larger and larger molecules that become available for binding. To ensure that bindings occur in the desired manner, special care must be taken when designing the DNA strands. The details are explained in Section 2, where we also define how DNA algorithms formally operate.

We present DNA algorithms for square root computing, and primality testing (based on a simple divisibility testing method) in Section 3. These algorithms are expressed using a super-constant amount of rules. Moreover, the rules are rather complex in the sense that the description of each strand relies on the ability to

[*] In this extended abstract, due to space issues, all proofs are deferred to the appendix.

count, and on the knowledge of an upper bound for the input. It is therefore natural to ask whether this complexity is necessary. Our main contribution is to answer this question affirmatively. Specifically, in Section 4 we show that if the set of rule strands describing some algorithm $\mathcal{A}$ form a regular language (i.e., are not complex), then $\mathcal{A}$ decides some regular language. We further establish that a constant number of rules is sufficient to express any DNA algorithm deciding a regular language.

Related Work. Utilizing DNA for the purpose of computation was first explored by Adleman [2], who solved a seven-city instance of the Hamiltonian path problem, and Lipton [10], who suggested a method to solve the SATISFIABILITY problem. After these first algorithmic usages of DNA, the idea to use the computational capacity to control devices on a molecular level emerged (see, e.g., [5]). These early techniques rely on enzymes to perform the desired task.

Our studies are motivated by the rise of the toehold exchange method [15], which is a way to perform DNA computations without relying on enzymes. The benefit of this method is that synthesizing the DNA strands needed for it (see, e.g., [19], in particular the supplementary material) is a simpler process than producing the building blocks for enzyme-based DNA computations. Strand displacement using toehold exchange is described thoroughly in [20] and simulated in [9]. The toehold technique enables the design of DNA circuits using only DNA strands, thus disposing of the necessity of other molecules, like enzymes. An essential building block in toehold-based computation is the *seesaw gate* [14], which allows the design of arbitrary DNA circuits. The seesaw gate was used to, e.g., compute approximate majority [6] with a protocol analyzed in [3]. Following this line of work, we utilize toeholds to initiate strand binding.

It is well understood that DNA molecules can form complex structures (see, e.g., [12,16] for an overview). For instance, Winfree [18] investigated how DNA strands can bind to form linear duplex strands, or duplex "tree" strands using junctions, which connect more than one strand in one point. He found that the linear strands correspond to a transition sequence in a finite automaton, whereas the trees correspond to a derivation tree of a context-free grammar. Note that these two structures do not return an output in the classical sense—the only "output" is a DNA molecule in which every base is bound. In contrast to that, in our work, we add input to the strand binding process, and are interested in an output, e.g., in the form of a YES or NO answer.

The techniques used in Adleman's construction inspired the study of so-called sticker systems [7], which are a generalized version of binding processes where the binding relation is not necessarily symmetric. Other studied language operations that are motivated by DNA interactions encompass the superposition [4], the PA-matching [8], and the hairpin [11,13] operator. This line of work examines the effect of these operations on a language's classification within the Chomsky hierarchy. In our studies, we also utilize methods from formal language theory in order to describe the strand binding process, which ultimately allows us to derive a lower bound on the complexity of DNA algorithms for non-regular languages.

2 Model

DNA Basics. DNA strands consist of nucleotides (or bases), linked together in a specific order. There are four types of nucleotides: Adenine, Cytosine, Guanine and Thymine (or simply A, C, G, and T). At an atomic level, DNA strands have two types of extremities: a so-called 5' end on one side, and a 3' end on the other side. The two extremities assign a *direction* to a DNA strand, and throughout this paper we orient nucleotide sequences of DNA strands from the 5' end towards the 3' end. When displayed, an arrow indicates the direction from the 5' end to the 3' end, as in Fig. 1a.

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*). That is, A and T are WK-complementary, and G and C are WK-complementary. Sequences of nucleotides can bind as well, given that those sequences have opposite directions and are complementary. For instance, the strands ATCG and CGAT from Fig. 1a can bind completely, whereas the two strands ATCG and TAGC (reversed CGAT) cannot.

Notation. To ease readability, sequences of nucleotides are commonly grouped into so-called *domains*. A domain is represented by a single character displayed in `teletype` font. Two domains `g` and `h` are complementary to each other if the nucleotide sequence of `g` is the WK-complement of `h`'s sequence. Complementary domains will be represented as overlined, e.g., $\mathtt{g} = \bar{\mathtt{h}}$. Note that for a sequence `gh`, the WK-complementary $\overline{\mathtt{gh}}$ is $\bar{\mathtt{h}}\bar{\mathtt{g}}$. Please refer to Fig. 1b for an illustration.

For a set S of strands we denote by $\overline{S}$ the set containing all WK-complements of strands in S. For two strands σ, τ, the *(concatenated) strand* $\sigma\tau$ is the strand obtained from concatenating the nucleotide sequences of σ and τ. For two sets S, T of strands we write ST for the set $\{\sigma\tau : \sigma \in S, \tau \in T\}$. Similarly, when σ is a strand, we also write σS and $S\sigma$ for the sets $\{\sigma\}S$ and $S\{\sigma\}$, respectively. For positive integers i, we write S^i for the set SS^{i-1}, and by convention S^0 contains only the empty strand ε. We denote by S^* the set $\cup_{i\geq 0} S^i$. The notation for sets of strands naturally extends to sets of domains, which can be viewed as sets containing the corresponding strands.

(a) Two complementary DNA strands. The arrowhead shows the 3' end.

(b) The strand `abc` composed of the three domains $\mathtt{a} \equiv$ ATCG, $\mathtt{b} \equiv$ ATT, and $\mathtt{c} \equiv$ CTC binds with its complement $\overline{\mathtt{abc}}$.

Fig. 1

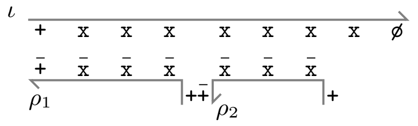

ι
+ x x x x x x x ∅
$\bar{+}$ $\bar{x}$ $\bar{x}$ $\bar{x}$ $\bar{x}$ $\bar{x}$ $\bar{x}$
ρ_1
$+\bar{+}$
ρ_2
+

Fig. 2: Checking whether 7 is divisible by 3 using DNA strands. The upper strand ι represents the input in unary. Both strands ρ_1 and ρ_2 of the form $\rho = \mathtt{+}\bar{\mathtt{x}}^3\bar{\mathtt{+}}$ bind to ι, first ρ_1 and then ρ_2. The unmatched domains $\mathtt{+x}\emptyset$ represent the remainder 1 of the division.

Example. Consider, as an example, the question whether some integer d is divisible by another integer q. To answer this question, let $\mathtt{+}$, $\emptyset$, and $\mathtt{x}$ be domains, and denote by $\iota = \mathtt{+x}^d\emptyset$ the strand composed of d repetitions of $\mathtt{x}$, delimited by $\mathtt{+}$ and $\emptyset$. We refer to ι as the *input strand* for our question.

The idea is to successively let ι bind with multiple copies of the strand $\rho = \mathtt{+}\bar{\mathtt{x}}^q\bar{\mathtt{+}}$. Note that ρ can only partially bind to ι. In particular, the first copy of ρ interacting with ι will bind with $\mathtt{+}$ and q $\mathtt{x}$ domains. Since DNA is not completely rigid, the unmatched $\mathtt{+}$ domain of ρ becomes available for binding. In the next step, a new copy of ρ binds with the $\mathtt{+}$ part from the previous ρ-strand and the next q $\mathtt{x}$ domains of the input strand. Each such step corresponds to subtracting q, and we end the process when the remainder of the division is left. An "output" can now be obtained by checking for all possible remainders of the division. In particular, if q divides d, then the above process results in the observable strand $\mathtt{+}\emptyset$. Please refer to Fig. 2 for an illustration.

Rule Strands and DNA Algorithms. Let $\mathcal{U}$ be a universe of domains so that $\mathcal{U} \cap \bar{\mathcal{U}} = \emptyset$, i.e., if some domain $\mathtt{x}$ is in $\mathcal{U}$, then $\bar{\mathtt{x}}$ is not. Let $\Sigma \dot{\cup} \Delta \dot{\cup} \Lambda$ be a partition of $\mathcal{U}$. We refer to Σ as the set of *input domains*, to $\Delta \cup \overline{\Delta}$ as the set of *delimiter domains* (*delimiters* for short), and to $\Psi = \Sigma \cup \overline{\Sigma} \cup \Lambda \cup \overline{\Lambda}$ as the set of *rule domains*. In the strand binding process, the delimiter domains will function as *toeholds* that initiate the binding. A strand $\rho \in \mathtt{d}_1\Psi^*\mathtt{d}_2$, where $\mathtt{d}_1$ and $\mathtt{d}_2$ are delimiters, is referred to as a *rule strand*. A collection $\mathcal{A}$ of rule strands is called a *DNA algorithm*. The input to a DNA algorithm is specified in the form of an *input strand*, which is a strand ι of the form $\mathtt{+}\Sigma^*\emptyset$, where $\mathtt{+}$ and $\emptyset$ are two fixed delimiters chosen from the set Δ.

A DNA algorithm is "executed" in a *soup*, i.e., a container filled with solution, in which the rule strands of some algorithm $\mathcal{A}$ are floating around. The execution is initiated by adding the input strand ι. We assume that all strands in the soup, i.e., the rule strands and the input strand, are present sufficiently many times.

All strands in the soup share the property of starting and ending with delimiters, and that delimiters appear only at the ends of a strand. When two strands σ and τ meet, they may bind and form a new strand which we call an *effective strand*, see Fig. 3. The binding occurs along some prefix of σ and a corresponding complementary suffix of τ (or the other way around, when the roles of σ and τ

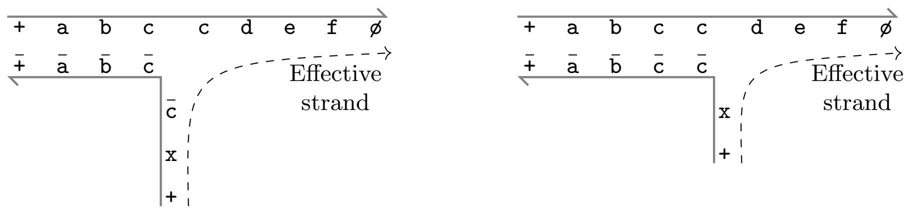

+ a b c c d e f ∅
$\bar{+}$ $\bar{a}$ $\bar{b}$ $\bar{c}$ $\bar{c}$ x +
Effective strand
+ a b c c d e f ∅
$\bar{+}$ $\bar{a}$ $\bar{b}$ $\bar{c}$ $\bar{c}$ x +
Effective strand

Fig. 3: Two examples of strand binding. On the left, not all possible domain bindings are involved, since one more `c` on the upper strand could bind with one more $\bar{\texttt{c}}$ of the lower strand; the resulting effective strand is `+x`$\bar{\texttt{c}}$`cdef∅`. On the right, all domains have bound; the resulting effective strand is `+xdef∅`;

are switched). This means that two strands always bind (at least) at two complementary delimiters. The effective strand resulting of this binding is composed of the unmatched prefix of τ and the unmatched suffix of σ, in this order. Note that the new effective strand also has the form of a rule strand. Effective strands behave like any other strand, and from now on we will not distinguish between strands and effective strands. We will use the terms *rule strand* and *input strand* to stress when a strand is not an effective one.

The process of strands binding with each other represents the computation performed by $\mathcal{A}$. The structure formed by such a binding process can be described as a tree, see Fig. 4.

Definition 1. *Fix a set S of strands. Let $T = (V, E, \mathrm{forth}, \mathrm{back})$ be a rooted ordered*[1] *tree, with nodes and edges in V and E, respectively, and two functions* forth, back*, each assigning a label from $(\mathcal{U} \cup \overline{\mathcal{U}})^* \dot\cup \{\mathrm{OPEN}\}$ to every edge in E, where* OPEN *is a special value. Denote by r the root of t, and for convenience consider r to also be a leaf. We say that T is an S-*assembly for σ *(or just* assembly for σ *when S is clear from the context) if T satisfies the following three conditions.*

(i) For all $e \in E$, if $\mathrm{back}(e) \neq \mathrm{OPEN}$, then $\mathrm{back}(e) = \overline{\mathrm{forth}(e)}$.

(ii) There is a unique path p from r to its rightmost descendant, s.t. the concatenated forth *labels on p are σ, all edges e on p have the label $\mathrm{back}(e) = \mathrm{OPEN}$, and no other edges in T have an* OPEN *label.*

(iii) Consider any two leafs t_1, t_2 with $t_2 \neq r$, so that t_1 is the last leaf visited before t_2 in a pre-order traversal of T, and denote by $t_{1,2}$ their nearest common ancestor. The word g obtained by concatenating the back *labels on the path from t_1 to $t_{1,2}$ and the* forth *labels on the path from $t_{1,2}$ to t_2 is in S.*

In this work, we focus on decision problems, i.e., problems where the output is either YES or NO. Let ι be an input strand. We say that $\mathcal{A}$ *accepts* ι if there is a $(\mathcal{A} \cup \{\iota\})$-assembly for `+∅`. Thus, the ability to produce the strand `+∅` corresponds to a YES output of $\mathcal{A}$, whereas the absence thereof corresponds to a NO output.

[1] A tree is ordered if the children of every node are ordered, e.g., from *left* to *right*.

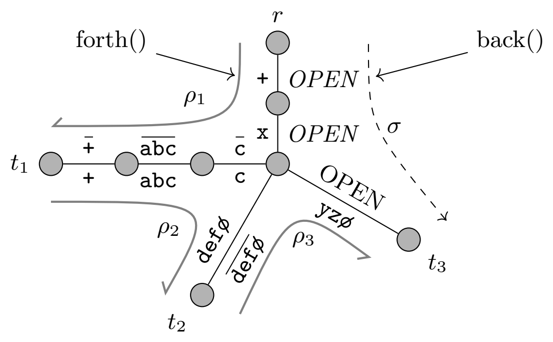


Fig. 4: An assembly T for $\sigma = \texttt{+xyz∅}$ with root r. The strands read in a pre-order traversal between the leaf pairs (r, t_1), (t_1, t_2), and (t_2, t_3) are $\rho_1 = \texttt{+x}\overline{\texttt{c}}\overline{\texttt{c}}\overline{\texttt{b}}\overline{\texttt{a}}\overline{\texttt{+}}$, $\rho_2 = \texttt{+abccdef∅}$, and $\rho_3 = \overline{\texttt{∅}}\overline{\texttt{f}}\overline{\texttt{e}}\overline{\texttt{d}}\texttt{yz∅}$, respectively.

We denote by $\mathcal{L}(\mathcal{A})$ the set of input strands accepted by $\mathcal{A}$, and say that $\mathcal{A}$ *decides* $\mathcal{L}(\mathcal{A})$. Note that in practice, the strand $\texttt{+∅}$ can be detected using standard fluorescence techniques (see, e.g., [1]).

Transition Sequences in Finite Automata. Let B be a deterministic finite automaton (DFA) with starting state s, accepting states F, and transition function δ. For any two states p and q in B, the word g is a *q-prefix (of B)* if $\delta(s, g) = q$; g is a *(p, q)-infix (of B)* if $\delta(p, g) = q$; and g is a *p-suffix (of B)* if $\delta(p, g) \in F$. Consider a word g and a decomposition $g_0 \ldots g_l$ of g into smaller words g_i. In that case, g is accepted by B if and only if there is a sequence of states $q_1, \ldots, q_l$ such that g_0 is a q_1-prefix, g_l is a q_l-suffix, and for $1 \leq i \leq l - 1$, g_i is a (q_i, q_{i+1})-infix. We denote by $\mathcal{L}(B)$ the *regular language* accepted by the automaton B. Please refer to a standard textbook (e.g., [17]) for a thorough introduction to formal language theory.

3 DNA Algorithms

When designing (DNA) algorithms it is convenient to use building blocks for solving reoccurring tasks. We will now introduce three such building blocks which we use in our algorithms, namely *gluing*, *substituting*, and *aggregating*. For that, in the remainder of this section, let $\texttt{d}_i$, $1 \leq i \leq 4$, be delimiters, and let p, x, s, y be arbitrary sequences of domains from Ψ^*.

Gluing. The gluing building block transforms the two strands $\sigma_1 = \texttt{d}_1 p x \texttt{d}_2$ and $\sigma_2 = \texttt{d}_3 y s \texttt{d}_4$ into the new strand $\texttt{d}_1 p s \texttt{d}_4$, formed of the prefix $\texttt{d}_1 p$ of σ_1 and the suffix $s \texttt{d}_4$ of σ_2. This is achieved by using the *gluing strand* $\overline{\texttt{d}_2}\overline{x}\overline{y}\overline{\texttt{d}_3}$. The strand binding is illustrated in Fig. 5a.

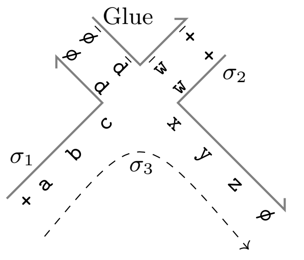
Substituting. The purpose of the substituting building block is to substitute a *whole* strand $\sigma_1 = \mathtt{d}_1 x \mathtt{d}_2$ with a strand $\sigma_2 = \mathtt{d}_3 y \mathtt{d}_4$. In general, this cannot be achieved with a single rule strand since more than two delimiter domains would be required to appear on it. Instead, in our algorithms, we introduce two new domains $\mathtt{u}$ and $\mathtt{v}$ for every substitution from x to y of the above form.

The replacement is now performed in three steps using three rule strands τ_1, τ_2, and τ_3 (illustrated in Fig. 5b) as follows. First, the strand $\tau_1 = \mathtt{d}_3\mathtt{u}\bar{\mathtt{x}}\overline{\mathtt{d}_1}$ binds with σ_1 to form the intermediate strand $\iota_1 = \mathtt{d}_3\mathtt{u}\mathtt{d}_2$. Next, the strand $\tau_2 = \overline{\mathtt{d}_2}\bar{\mathtt{u}}\mathtt{v}\mathtt{d}_4$ is used to form the intermediate strand $\iota_2 = \mathtt{d}_3\mathtt{v}\mathtt{d}_4$ by binding with ι_1. In the last step, the strand $\tau_3 = \mathtt{d}_3 y \bar{\mathtt{v}}\overline{\mathtt{d}_3}$ binds with ι_2 to form the desired strand σ_2. As a short-hand for these three rule strands, we write the *substitution rule* $\sigma_1 \to \sigma_2$, e.g., $\mathtt{+abc\emptyset} \to \mathtt{+z\emptyset}$.

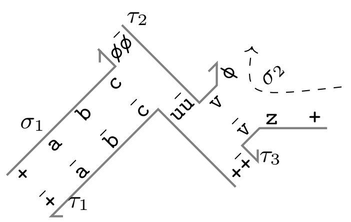
The above rule strands ensure that if only τ_1, or only τ_1 and τ_2 are applied, but not τ_3, then either $\mathtt{u}$ or $\mathtt{v}$ remain on the effective strand. Since for each substitution rule new domains $\mathtt{u}$ and $\mathtt{v}$ are introduced, they cannot be matched by any strand that is not from this substitution. Thus, we may assume that substitutions are either applied in full, or not at all. (The strands obtained by applying at most two of the three rules have no effect on the soup's output.)

Aggregating. By combining the two building blocks, it is possible to *aggregate* two whole strands $\sigma_1 = \mathtt{d}_1 x \mathtt{d}_2$ and $\sigma_2 = \mathtt{d}_3 y \mathtt{d}_4$ into a strand $\sigma_3 = \mathtt{d}_5 z \mathtt{d}_6$, i.e., σ_3 is only obtained if both σ_1 and σ_2 are present. For that, let $\mathtt{u}$ and $\mathtt{v}$ be new domains. We add the following strands: (1) the substitution $\sigma_1 \to \mathtt{+u}z\mathtt{d}_6$, (2) the substitution $\sigma_2 \to \mathtt{d}_5\mathtt{v}\emptyset$, and (3) the gluing strand $\bar{\emptyset}\bar{\mathtt{v}}\bar{\mathtt{u}}\bar{\mathtt{+}}$. We abbreviate this set of rules by writing $\sigma_1 + \sigma_2 \to \sigma_3$, and note that also larger aggregations are possible by applying the principle inductively.

Input Encoding. For the sake of simplicity we consider only unary inputs. That is, we fix a domain $\mathtt{a} \in \Sigma$ and represent an input number k by the input strand $\iota_k = \mathtt{+a}^k\emptyset$. We note that it is possible to design algorithms for the presented problems to work with binary inputs, i.e., input strands of the form $\mathtt{+}\{\mathtt{0},\mathtt{1}\}^*\emptyset$.



(a) Gluing the two strands σ_1 and σ_2 with $\bar{\emptyset}\bar{\mathtt{d}}\bar{\mathtt{w}}\bar{\mathtt{+}}$ yields the effective strand σ_3.



(b) Substituting the strand $\sigma_1 = \mathtt{+abc\emptyset}$ by $\sigma_2 = \mathtt{+z\emptyset}$ using the three strands τ_1, τ_2, and τ_3. The effective strand indicated by the dashed line is σ_2, as desired.

Fig. 5

Primality Testing. Using the above techniques, the idea for divisibility testing explained in Section 2 can be extended to obtain a DNA algorithm for primality testing. Let n be the largest possible input number, and let $\mathtt{a} \in \Sigma$ and $\mathtt{b}, \mathtt{f} \in \Lambda$ be domains. Our algorithm that tests any number $k \leq n$ for primality is composed of four kinds of rule strands.

Specifically, for any integer $2 \leq j \leq \sqrt{n}$ and integer $k \leq j - 1$, we add the following: (1) the *initialization* rule strands $\mathtt{+b}^j\mathtt{a}\bar{\mathtt{a}}\bar{\mathtt{+}}$, (2) the *division* rule strands $\mathtt{+b}^j\bar{\mathtt{a}}^j\bar{\mathtt{b}}^j\bar{\mathtt{+}}$, (3) the *remainder* rule strands $\mathtt{+b}^j\mathtt{a}^k\mathtt{\emptyset} \to \mathtt{+b}^j\mathtt{f}\mathtt{\emptyset}$, and (4) the *test* rules strands $\mathtt{+b}^2\mathtt{f}\mathtt{\emptyset} + \mathtt{+b}^3\mathtt{f}\mathtt{\emptyset} + \cdots + \mathtt{+b}^{\sqrt{n}}\mathtt{f}\mathtt{\emptyset} \to \mathtt{+}\mathtt{\emptyset}$. The task of the initialization rules (1) is to mark all input strands with one possible divisor j, thus creating an intermediate strand of the form $\mathtt{+b}^j\mathtt{a}^k\mathtt{\emptyset}$. The division rules (2) then divide the so marked intermediate strands by the divisor, by subtracting j once per application of the division rule strand $\mathtt{+b}^j\bar{\mathtt{a}}^j\bar{\mathtt{b}}^j\bar{\mathtt{+}}$. Should the division of k by j leave a remainder, the remainder rules (3) will create a strand indicating that j does not divide k. Finally, if j is not divisible by any such j, the test rules (4) will produce the desired YES-output.

This unary algorithm consists of $O(n)$ rules. Note that with this many rules it is possible to describe a simple DNA algorithm that matches every strand ι_p, with p prime, to produce a YES-output. With binary inputs it is possible to devise a primality testing algorithm using $O(\sqrt{n}\log n)$ rules, i.e., less than the number of primes $p \leq n$, which is $\Theta(n/\log n)$. Regardless of the way the input is presented, enumerating the rules for our primality testing algorithm requires knowledge of $\sqrt{n}$. An algorithm to actually compute $\sqrt{n}$ using DNA strands is presented in the appendix.

4 Regular DNA Algorithms

The algorithms we presented above use at most $O(n)$ rule strands, or $O(\sqrt{n}\log n)$ when binary input encoding is used. While the rule strands we use are of uniform nature, our algorithms' descriptions require an upper bound on the input size. Moreover, the corresponding rule sets form context-free or even context-sensitive languages, i.e., the rules for the two algorithms are "complex". One might hope that this complexity is not necessary. However, in this section we show that such complex rule sets cannot be avoided (for these problems), and that in fact rule sets that form a regular language are not very powerful algorithms. Specifically, we are going to establish the following theorem.

Theorem 1. *Let $\mathcal{A}$ be a DNA algorithm. If $\mathcal{A}$, interpreted as a set of strands, is a regular language, then $\mathcal{L}(\mathcal{A})$ is regular.*

For the remainder of this section, fix some DNA algorithm $\mathcal{A}$ such that $\mathcal{A}$ is a regular language. It will be convenient to assume that $\mathtt{+}\bar{\mathtt{+}}$ and $\bar{\mathtt{\emptyset}}\mathtt{\emptyset}$ are both in $\mathcal{A}$. While this changes neither the language detected by $\mathcal{A}$ nor the fact that $\mathcal{A}$ is a regular language, the assumption allows us to consider only $(\mathcal{A} \cup \{\iota\})$-assemblies for $\mathtt{+}\mathtt{\emptyset}$ that begin and end with strands from $\mathcal{A}$. We denote by $B = (Q, \mathcal{U} \cup \overline{\mathcal{U}}, \delta, s, F)$ a DFA satisfying $\mathcal{L}(B) = \mathcal{A}$.

Consider any accepted input strand ι and an $(\mathcal{A} \cup \iota)$-assembly T of $+\emptyset$. Note that forth(e) contains a delimiter if and only if one endpoint of e is a leaf or the root. By definition, in a pre-order traversal of T the paths between two successive leafs are labeled with strands $(\rho_1, \ldots, \rho_l)$ in $\mathcal{A}$. For each strand $\rho_i \in \mathcal{A}$ there is an accepting transition sequence in B, which consequently corresponds to the path in T from which ρ_i was obtained. Our life would be simple if all $\rho_i \in \mathcal{A}$, since we would have to deal only with strands from a regular language. The main difficulty in our proof of Theorem 1 is to handle the case where $\rho_i = \iota$.

In our proof of Theorem 1 we use assemblies to investigate how exactly the strands from $\mathcal{A}$ and the input strands bind to form $+\emptyset$. To describe the language $\mathcal{L}(\mathcal{A})$, we consider all possible assemblies for $+\emptyset$ that can be formed with some input strand and strands from $\mathcal{A}$. More precisely, we ask the question: What are the possible input strands with which $+\emptyset$ can be assembled? A key ingredient to answering this question is the notion of a *junction*, which will allow us to answer the question in a recursive manner.

Definition 2. *Let Q be a set of states, let $v, w \in Q$, and let $J \subseteq Q \times Q$ with $(v, w) \notin J$. The triple (v, w, J) is called a* junction. *An* instance of the junction *(v, w, J) is a sequence $q = (q_1, \ldots, q_\ell)$ with entries in $Q \dot\cup \{\mathrm{NULL}\}$, where* NULL *is a special value not contained in Q, satisfying*

(i) $q_1 = v$ and $q_\ell = w$,

(ii) for all i, if $q_i \neq \mathrm{NULL}$, then either $(q_i, q_{i+1}) \in J \cup \{(v, w)\}$, or $q_{i+1} = \mathrm{NULL}$ and $(q_i, q_{i+2}) \in J \cup \{(v, w)\}$, and

(iii) all entries in q are pairwise distinct, except possibly q_1 and q_ℓ (when $v = w$).

The basic idea behind our proof is to define a language $\mathcal{I}$, such that (1) $\mathcal{I}$ is regular, (2) every ι accepted by $\mathcal{A}$ is in $\mathcal{I}$, and (3) every $\iota \in \mathcal{I}$ is accepted by $\mathcal{A}$. The definition of $\mathcal{I}$ is encapsulated in the recursive *sealing operator* $\mathcal{X}$, the exact definition of which is deferred to the appendix. Claim (1) will then be established by the fact that the recursion $\mathcal{X}$ is finite, whereas claims (2) and (3) will be confirmed by relating the recursion $\mathcal{X}$ with appropriate assemblies. For the latter two, the idea is to identify nodes in an assembly with junctions. Basically, $\mathcal{X}$ assigns a language to any junction (v, w, J). The intricate choice of $\mathcal{X}$ provides that $\mathcal{X}(v, w, J)$ contains exactly the input strands that "seal" the junction (v, w, J) with a sub-tree T' of some assembly T, such that in T' only junctions (v', w', J') with $(v', w') \notin J$ and $J' = J \setminus \{(v', w')\}$ appear.

We set the language $\mathcal{I}$ to

$$\mathcal{I} := \bigcup_{w : \delta(w, \emptyset) \in F} \mathcal{X}(\delta(s, +), w, Q \times Q \setminus \{(\delta(s, +), w)\}),$$

and follow the plan to prove Theorem 1 as outlined above. The first step is to show that $\mathcal{I}$ is regular, which is asserted by the following lemma.

Lemma 1. *For any $v, w \in Q$ and $J \subseteq Q \times Q \setminus \{(v, w)\}$, the language $\mathcal{X}(v, w, J)$ is regular.*

Lemma 2. *If $\iota \in \mathcal{I}$, then ι is accepted by $\mathcal{A}$.*

To obtain the opposite direction of the statement in Lemma 2 in our effort to establish Theorem 1, consider any $(A \cup \{\iota\})$-assembly T for +ø. We now describe a procedure that assigns an additional label to T, thus obtaining the *state-marked assembly* T'. Specifically, to each node u in T, we assign $\deg(u)+1$ labels $\mathrm{qmark}(u, 1), \ldots, \mathrm{qmark}(u, \deg(u)+1)$ from $Q \dot{\cup} \{\mathit{NULL}\}$, where $\deg(u)$ denotes the number of u's children.

To obtain the labels, we traverse T "together with B", feeding to B the forth labels when traversing an edge to the i^{th} child the first time (in forward direction), and set $\mathrm{qmark}(u, i)$ to B's current state. On the way back to u's parent, we assign the label $\mathrm{qmark}(u, \deg(u)+1)$, respectively. At leaf nodes, the DFA B is reset to its starting state.

We will later use the sequence defined for a node u by the qmark labels to obtain junctions and junction instances corresponding to u. For that, it is convenient to define two short-hands for reading the labels qmark. Consider any node u and denote by $u_1, \ldots, u_{\deg(u)}$ the children of u. The first short-hand, qpair, assigns to each edge (u, u_i) a pair of states as follows.

$$\mathrm{qpair}(u, u_i) := \begin{cases} (\mathrm{qmark}(u, i-1), \mathrm{qmark}(u, i+1)), & \text{if } \mathrm{qmark}(u, i) = \mathit{NULL} \\ (\mathrm{qmark}(u, i), \mathrm{qmark}(u, i+2)), & \text{if } \mathrm{qmark}(u, i+1) = \mathit{NULL} \\ (\mathrm{qmark}(u, i), \mathrm{qmark}(u, i+1)), & \text{otherwise,} \end{cases}$$

where we set $\mathrm{qmark}(u, 0) = \mathrm{qmark}(u, \deg(u)+2) = \mathit{NULL}$.

The second short-hand $\mathrm{junct}(u)$ assigns a junction to every node u. If u is a leaf, then the assignment depends on $\mathrm{qmark}(u', i+1)$, where u' is the parent of u, and u is the i^{th} child of u'. Specifically, $\mathrm{junct}(u) = (\mathrm{qmark}(u), x, \emptyset)$, where x is s if $\mathrm{qmark}(u', i+1) \in Q$ and NULL otherwise. If u is not a leaf, denote by v and w the first and the last non-NULL entry in the sequence of qmark labels for node u, respectively. Let further p be the path from T's root to u, and denote by $P \subseteq Q \times Q$ the set $\{\mathrm{qpair}(e) : e \in p \text{ and } \mathrm{qpair}(e) \text{ has no } \mathit{NULL} \text{ entries}\}$. The junction $\mathrm{junct}(u)$ is now $(v, w, Q \times Q \setminus (P \cup \{(v, w)\}))$.

Lemma 3. *Let T be a state-marked assembly for some strand σ. There is a state-marked assembly T' for σ such that*

(i) *all nodes v in T have $\deg(v) \neq 1$, except the root,*
(ii) *at every node v in T', the sequence $(\mathrm{qmark}(v, i), \ldots, \mathrm{qmark}(v, j))$ is an instance of the junction $(\mathrm{qmark}(v, i), \mathrm{qmark}(v, j), J)$ for some J, where i and j are the first and last index for which $\mathrm{qmark}(v, i)$ and $\mathrm{qmark}(v, j)$ are not* NULL, *and $i \in \{1, 2\}$, $j \in \{\deg(v), \deg(v)+1\}$, and*
(iii) *on every simple path P in T' from the root to a leaf, for every two edges e, e' on P, if all entries in $\mathrm{qpair}(e)$ and $\mathrm{qpair}(e')$ are from Q (i.e., not* NULL*), then it holds that $\mathrm{qpair}(e) \neq \mathrm{qpair}(e')$.*

The above technical Lemma 3, which essentially states the existence of a normal form for assemblies, is key in our proof for the missing part of Theorem 1, which is taken care of with the next lemma.

Lemma 4. *If $\iota \in \mathcal{L}(\mathcal{A})$, then $\iota \in \mathcal{I}$.*

Theorem 1 is now established with help of Lemmas 1, 2 and 4. Since finite languages are regular, it follows that $\mathcal{L}(\mathcal{A})$ is regular if the size of $\mathcal{A}$ is finite. Lastly, in the appendix we also establish the following.

Theorem 2. *Let $\mathcal{L}$ be a language with $\varepsilon \notin \mathcal{L}$. If $\mathcal{L}$ is regular, then there is a constant size DNA algorithm that decides $+\mathcal{L}\emptyset$.*

References

1. DNA technology. In: Lakowicz, J.R. (ed.) Principles of Fluorescence Spectroscopy, pp. 705–740. Springer US (2006)
2. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science 266(5187), 1021–1024 (1994)
3. Angluin, D., Aspnes, J., Eisenstat, D.: A simple population protocol for fast robust approximate majority. Distributed Computing 21(2), 87–102 (2008)
4. Bottoni, P., Labella, A., Manca, V., Mitrana, V.: Superposition based on Watson–Crick-like complementarity. Theory of Computing Systems 39(4), 503–524 (2006)
5. Breaker, R.R.: Engineered allosteric ribozymes as biosensor components. Current Opinion in Biotechnology 13(1), 31 – 39 (2002)
6. Cardelli, L., Csikász-Nagy, A.: The cell cycle switch computes approximate majority. Scientific Reports 2 (Sep 2012)
7. Kari, L., Păun, G., Rozenberg, G., Salomaa, A., Yu, S.: DNA computing, sticker systems, and universality. Acta Informatica 35(5), 401–420 (1998)
8. Kobayashi, S., Mitrana, V., Păun, G., Rozenberg, G.: Formal properties of PA-matching. Theoretical Computer Science 262(1-2), 117 – 131 (2001)
9. Lakin, M.R., Phillips, A.: Modelling, simulating and verifying turing-powerful strand displacement systems. In: Proc. of DNA Computing and Molecular Programming (2011)
10. Lipton, R.J.: DNA solution of hard computational problems. Science 268(5210), 542–545 (1995)
11. Manea, F., Martín-Vide, C., Mitrana, V.: Hairpin lengthening: language theoretic and algorithmic results. J. of Logic and Computation (2013)
12. Patitz, M.: An introduction to tile-based self-assembly and a survey of recent results. Natural Computing 13(2), 195–224 (2014)
13. Păun, G., Rozenberg, G., Yokomori, T.: Hairpin languages. Int. J. of Foundations of Computer Science 12(06), 837–847 (2001)
14. Qian, L., Winfree, E.: Scaling up digital circuit computation with DNA strand displacement cascades. Science 332(6034), 1196–1201 (2011)
15. Seelig, G., Soloveichik, D., Zhang, D.Y., Winfree, E.: Enzyme-free nucleic acid logic circuits. Science 314(5805), 1585–1588 (2006)
16. Seeman, N.C.: An overview of structural DNA nanotechnology. Molecular Biotechnology 37(3), 246–257 (2007)
17. Sipser, M.: Introduction to the Theory of Computation. International Thomson Publishing (1996)
18. Winfree, E., Yang, X., Seeman, N.C.: Universal computation via self-assembly of DNA: Some theory and experiments. In: DNA Based Computers II, volume 44 of DIMACS. pp. 191–213. American Mathematical Society (1996)
19. Zhang, D.Y., Turberfield, A.J., Yurke, B., Winfree, E.: Engineering entropy-driven reactions and networks catalyzed by DNA. Science 318(5853), 1121–1125 (2007)
20. Zhang, D.Y., Winfree, E.: Control of DNA strand displacement kinetics using toehold exchange. J. of the American Chemical Society 131(47), 17303–17314 (2009)

Appendix

A Reference Material

A.1 Biological Realization

The assumptions made in Section 2 are justified from a biological perspective. The ready availability of DNA duplicating technology makes it possible to produce all initially required rule strands (and the input strand) in sufficient numbers. This ensures that all possible bindings between two strands in the soup, originally present as well as effective, occur eventually. Moreover, the strands are fabricated so that all non-delimiter domains are covered with single domain strands prior to introducing the strands in the soup. The covers prevent any binding that does not involve two WK-complementary delimiter domains. Furthermore, these covers can be designed so that they lift off one by one when the strands bind past the delimiter. Our delimiters take on the function of *toeholds*, i.e., docking locations, that initiate binding between two strands, cf. [15,20].

Regarding transcription of domains into nucleotide sequences, special care must be taken. For instance, some (sub)sequences are problematic to manufacture. Please refer to [19], in particular the supplementary material, for a detailed description of this process. As in [15,19,20] we assume that single domains bind either completely or not at all.

If an algorithm is executed, and an output is created in the soup, how does one detect it? It is a standard technique in biochemistry to use fluorescence for such detection purposes. Chemical compounds can be crafted to react with a given strand type and activate a fluorescent (see, e.g., [1]). Thus, the strand $+\emptyset$ can be detected by the emitted light.

A.2 Regular Languages

A well established concept we use throughout this paper are *deterministic finite automata* (DFA). A DFA B is a 5-tuple $(Q, \Gamma, \delta, s, F)$, where the elements in the tuple are the set of states, the alphabet, the state transition function, the initial state, and the set of accepting states, respectively. The state transition function $\delta : Q \times \Gamma \to Q$ is extended to words $g = g_1 \dots g_l \in \Gamma^*$ by setting $\delta(q, g_1 \dots g_l) = \delta(\delta(q, g_1), g_2 \dots g_l)$. An automaton B accepts a word $g \in \Gamma^*$ if $\delta(s, g) \in F$, and $\mathcal{L}(B)$ is the *regular language* accepted by the automaton B. It is known that regular languages are closed under *regular operations*, which include finite intersections, unions, and complements. We note that for a regular language L, the language $\overline{L}$ obtained by taking the WK-complement of each word in L, is also regular. Finite automata are known to be robust to certain variations. One variation we will use is to permit ε-transitions, i.e., $\delta(q, \varepsilon) = q'$ can happen "spontaneously".

B Square Testing Algorithm

To design the rule strands for our primality testing algorithm required knowledge of $\sqrt{n}$. Therefore we tackle the problem of deciding the language of square numbers next.

Our algorithm is based on the observation that for any m, it holds that $\sum_{j=1}^{m-1} 2j = m^2 - m$. In other words, if a number n is a square, then it can be written as the sum of $\sqrt{n}$ and the even numbers smaller than $2\sqrt{n}$. The following algorithm reverses these steps and subtracts even numbers to find $\sqrt{n}$, using the following rules: (1) the initialization rule strand $\mathtt{+ba}\bar{\mathtt{a}}\bar{\mathtt{+}}$, (2) the subtraction rule strands $\mathtt{+b}^{j+1}\bar{\mathtt{a}}^{2j}\bar{\mathtt{b}}^{j}\bar{\mathtt{+}}$ for every j, and (3) the test rule strands $\mathtt{+b}^{j}\mathtt{a}^{j}\mathtt{\emptyset} \rightarrow \mathtt{+\emptyset}$. The total number of rule strands is thus $O(\sqrt{n})$.

The first strand that can bind to an input is the initialization rule (1), which just adds a $\mathtt{b}$ prefix to the input. The subtraction rules (2) then bind one after the other, starting with the strand obtained for $j = 1$; binding the subtraction rule containing j $\bar{\mathtt{b}}$s corresponds to subtracting $2j$ from ι. The same rule marks the resulting strand with $(j+1)$ $\mathtt{b}$ domains, so that the next subtraction rule (or the test rule) can bind. Last, the test rule binds when no subtraction rule can bind with the strand obtained from this process. Note that this algorithm can be adapted to return the actual square root of the tested number (rounded down). This is achieved by adding the rules $\mathtt{+b}^{j}\mathtt{a}^{l}\mathtt{\emptyset} \rightarrow \mathtt{+x}^{j}\mathtt{\emptyset}$ for $l \leq j$, i.e., a strand $\mathtt{+x}^{j}\mathtt{\emptyset}$ is interpreted as the output j.

C Proof Details

C.1 The Sealing Operator $\mathcal{X}$

In the following, we set the empty union to $\emptyset$ and the empty intersection to $\mathtt{+}\Sigma^{*}\mathtt{\emptyset}$, i.e., all valid input strands. For the definition of $\mathcal{X}$, let $I(v, w, J)$ be the set of instances of the junction (v, w, J). The sealing operator is defined using two sub-operators C and D as follows.

$$\mathcal{X}(v, w, J) := \bigcup_{q \in I(v,w,J)} \left(\bigcap_{\substack{(q_i, q_{i+1}): \\ q_i \neq NULL \neq q_{i+1}}} C(q_i, q_{i+1}, J) \cap \bigcap_{i : q_i = NULL} D(q_{i-1}, q_{i+1}, J) \right).$$

For $C(x, y, J)$, we denote by $H_{x,y,J}$ the set of pairs $(z_1, z_2) \in J \setminus \{(x, y)\}$ for which there are two transition sequences $x \rightarrow z_1$, $z_2 \rightarrow y$ in B such that the corresponding words w_1 and w_2 satisfy $w_1 = \overline{w_2}$. Note that in particular $H_{x,y,J}$ is finite, since it is contained in J. Now, the sets C are

$$C(x, y, J) := \begin{cases} \mathtt{+}\Sigma^{*}\mathtt{\emptyset}, & \text{if } \exists (z_1, z_2) \in H_{x,y,J} \text{ s.t. } z_2 = s \text{ and } z_1 \in F \\ \displaystyle\bigcup_{(z_1, z_2) \in H_{x,y,J}} \mathcal{X}(z_1, z_2, J \setminus \{(x, y), (z_1, z_2)\}), & \text{otherwise.} \end{cases}$$

For $D(x, y, J)$, we construct a finite automaton $B_{K,x,y}$ with ε-transitions as follows: Let B_K be the automaton B supplemented with the transitions $\delta(k_1, \varepsilon) = k_2$ for all $(k_1, k_2) \in K$. Next, let B'_K be a copy of B_K, and denote by x' the copy of the state x in B'_K. The automaton $B_{K,x,y}$ is now obtained by taking B_K's starting state, B'_K's accepting states, and adding the transition $\delta(y, \varepsilon) = x'$. With this, the sets D are

$$D(x, y, J) := \bigcup_{K \subseteq J \setminus \{(x,y)\}} \left(\overline{\mathcal{L}(B_{K,x,y})} \cap \bigcap_{(z_1,z_2) \in K} \mathcal{X}(z_1, z_2, J \setminus \{(x, y), (z_1, z_2)\}) \right) .$$

C.2 Omitted Proofs

Lemma 1.

Proof (of Lemma 1). Note first that $+\Sigma^*\not{o}$, $\overline{\mathcal{L}(B_{K,x,y})}$, and $\emptyset$ are all regular languages. The language $\mathcal{X}(v, w, J)$ is obtained using finitely many regular operations on these languages, since the size of J decreases in every recursive step.

Lemma 2.

Proof (of Lemma 2). Let $v = \delta(s, +)$, let $W = \{w : \delta(w, \not{o}) \in F\}$, and let ι be an input strand. If $\iota \in \mathcal{I}$, then there is some $w \in W$ such that $\iota \in \mathcal{X}(v, w, Q \times Q \setminus \{(v, w)\})$. Fix any such w, and denote by L and R the two intersections in $\mathcal{X}(v, w, J)$, i.e.,

$$L = \bigcap_{\substack{(q_i, q_{i+1}): \\ q_i \neq NULL \neq q_{i+1}}} C(q_i, q_{i+1}, J) \qquad \text{and} \qquad R = \bigcap_{i : q_i = NULL} D(q_{i-1}, q_{i+1}, J) .$$

By the definition of $\mathcal{X}$, there is an instance q of the junction (v, w, J) such that ι is contained in both L and R. Our goal now is to recursively construct a $(\mathcal{A} \cup \{\iota\})$-assembly T for $+\not{o}$.

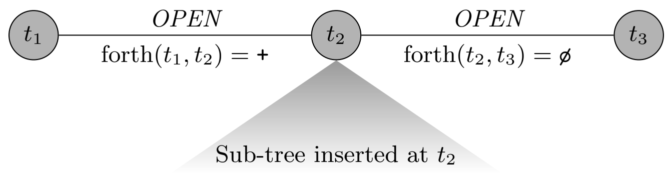


Fig. 6: Base case construction for the proof of Lemma 2. Starting at t_2, the rest of the assembly is created recursively. In the recursion, t_2 is considered as the junction $(v, w, Q \times Q \setminus \{(v, w)\})$, where v and w are such that $v = \delta(s, +)$ and $\delta(w, \not{o}) \in F$.

The construction of the assembly T begins as follows (depicted in Fig. 6). We first add three nodes t_1, t_2, t_3 and the edges $e_{1,2} = (t_1, t_2)$, and $e_{2,3} = (t_2, t_3)$ to T. The labels are $\text{forth}(e_{1,2}) = +$, $\text{forth}(e_{2,3}) = \emptyset$, and $\text{back}(e_{1,2}) = \text{back}(e_{2,3}) = \mathit{OPEN}$. We now consider the node t_2 as corresponding to the single instance q of the junction (v, w, J).

The remaining parts of T are constructed recursively. The construction follows the recursive definition of $\mathcal{X}$ and is therefore finite for the same reason the recursion $\mathcal{X}$ is finite, namely, because J decreases in every step. In the recursion, we will assign to every node t in T a sequence $(\phi_1, \ldots, \phi_m)$, where each ϕ_k consists of a junction (v, w, J) and a matching instance q. After assigning the sequence to t, the recursion enters t, and we handle each item ϕ_k in the sequence separately. The first sequence we consider is the one consisting of only q, which we assigned to node t_2. For an item ϕ_k, denote by $q = (q_1, \ldots, q_l)$ the instance of the corresponding junction (v, w, J). For $q_i \neq \mathit{NULL}$, we say that q_i is a *path case* if $q_{i+1} \neq \mathit{NULL}$, and that q_i is a *branch case* if $q_{i+1} = \mathit{NULL}$. We will consider these two cases for the states q_i in the order in which they appear.

To show that our construction is indeed an assembly, we consider three things. Firstly, we ensure that the edge labels forth and back are complementary (except for the two edges $e_{1,2}$ and $e_{2,3}$, which have an *OPEN* back label), which guarantees requirement (i) in Definition 1. Secondly, an *OPEN* label is assigned only in the base of the construction, thus ensuring requirement (ii) in Definition 1. Lastly, we guarantee requirement (iii) in Definition 1 inductively. For that, we consider T as being traversed in pre-order during the recursive construction.

For the base of the induction it is important to note that in the base construction involving the nodes t_1, t_2, and t_3, the label $\text{forth}(e_{1,2})$ is a v-prefix, and the label $\text{forth}(e_{2,3})$ is a w-suffix. The induction hypothesis for a node t and each considered pair (x, y) of states in a junction instance at t is thus that the labels read on the way from the previous leaf (or root) to t (in the pre-order traversal) form an x-prefix. To ensure the same for the next induction step, we need to show that the labels read on the way from the next leaf to t form a y-prefix. Moreover, we ensure that the last state in ϕ_k is the first state in ϕ_{k+1}. The induction is completed by the fact that w is the last state in the junction considered at t_2, so that the w-prefix and the w-suffix can be combined to form a strand in $\mathcal{A}$.

Path Cases. For every path case q_i, we have that $\iota \in C(q_i, q_{i+1}, J)$. According to the case distinction in C's definition, this can happen in two sub-cases. In the first sub-case, there is a pair $(z_1, s) \in H_{q_i, q_{i+1}, J}$ with $z_1 \in F$. This means that there is a q_i-suffix g_1 and a q_{i+1}-prefix g_2 in B with $g_1 = \overline{g_2}$. The q_i branch of the current junction can be completed (or *sealed*) by adding a node t' and an edge (t, t'), and setting $\text{forth}(t, t') = g_1$ and $\text{back}(t, t') = g_2$. Correctness of this first sub-case now immediately follows from the induction hypothesis and the properties of g_1 and g_2.

In the second sub-case, there is a pair $(z_1, z_2) \in H_{q_i, q_{i+1}, J}$ such that $\iota \in \mathcal{X}(z_1, z_2, J')$, where $J' = J \setminus \{(q_i, q_{i+1}), (z_1, z_2)\}$. This means that there is a (q_i, z_1)-infix g_1 and a (z_2, q_{i+1})-infix g_2 in B with $g_1 = \overline{g_2}$. Again, we append to

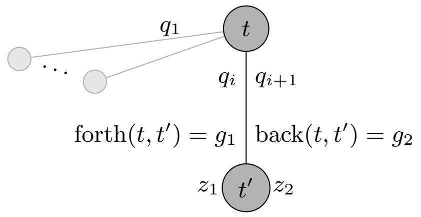


Fig. 7: The edge $e = (t, t')$ is inserted when q_i is a path case. The labels g_1 and g_2 are a (q_i, z_1)-infix and a (z_2, q_{i+1})-infix, respectively. In the recursion, all edges left of e were added to the assembly before inserting e, namely when treating q_j with $j < i$.

t a node t' and an edge (t, t'), and set $\mathrm{forth}(t, t') = g_1$ and $\mathrm{back}(t, t') = g_2$. Since $\iota \in \mathcal{X}(z_1, z_2, J')$, there is an instance q' of the junction (z_1, z_2, J') such that ι is in both intersections in $\mathcal{X}(z_1, z_2, J')$. We recursively consider the node t' as corresponding to the single junction instance q'.

The fact that g_1 is a (q_i, z_1)-infix together with the induction hypothesis guarantees the induction hypothesis for the recursive step with q'. The induction at q' together with the promise that g_2 is a (z_2, q_{i+1})-infix now guarantees the induction hypothesis at t for the next branch or path case.

Branch Cases. For every branch case q_i, we have that $\iota \in D(q_i, q_{i+2}, J)$. This means that there is a subset K of $J \setminus \{(q_i, q_{i+2})\}$ such that $\iota \in \overline{\mathcal{L}(B_{K,q_i,q_{i+2}})}$ and that for all pairs $(z_1, z_2) \in K$ it holds that $\iota \in \mathcal{X}(z_1, z_2, J')$, where $J' = J \setminus \{(q_i, q_{i+2}), (z_1, z_2)\}$.

Since $\iota \in \overline{\mathcal{L}(B_{K,q_i,q_{i+2}})}$, there is a transition sequence in $B_{K,q_i,q_{i+2}}$ for $\overline{\iota}$, the WK-complement of ι. Consider any such sequence a, and observe that due to the construction of $B_{K,q_i,q_{i+2}}$, the transition $\delta(q_{i+2}, \varepsilon) = q_i'$ must be taken exactly once in a, and that the first occurrence of q_i' is after that transition. Let b be the suffix of a starting at the first occurrence of q_i'. We partition b into *ε-blocks* and *non-ε-blocks*. An ε-block consists of contiguous ε-transitions, preceded and succeeded by non-ε-transitions. The remaining contiguous parts of b form the non-ε-blocks.

Let $n_1, \varepsilon_1, \dots, n_{h-1}, \varepsilon_{h-1}, n_h$ be the partition of b. We add to T the path $t = u_0, u_1, \dots, u_h$. Let g be the word corresponding to the transition sequence b, and let $g = g_1 g_2 \dots g_h$ be a division of g into smaller words such that g_i corresponds to the sub-sequence n_i. The labels $\mathrm{forth}(u_0, u_1), \dots, \mathrm{forth}(u_{h-1}, u_h)$ are set to $g_1, \dots, g_h$, respectively, and the labels $\mathrm{back}(u_j, u_{j+1})$ are set to $\overline{\mathrm{forth}(u_j, u_{j+1})}$, thus corresponding to a prefix of ι.

The idea is now to transform each ε-block into a sequence of junctions on a single node on the path $u_0, u_1, \dots, u_h$. Consider an ε-block ε_j, $1 \le j \le h - 1$, and let $\delta(z_1, \varepsilon) = z_2$, $\delta(z_2, \varepsilon) = z_3$, $\dots$, $\delta(z_{m-1}, \varepsilon) = z_m$ be the ε-transitions in ε_j. Recall that for all $1 \le k \le m$, the pair (z_k, z_{k+1}) is in K. Therefore, $\iota \in \mathcal{X}(z_k, z_{k+1}, J_k')$ with $J_k' = J \setminus \{(q_i, q_{i+2}), (z_k, z_{k+1})\}$. This in turn means that for each k, there is some instance of the junction (z_k, z_{k+1}, J_k') such that ι is in

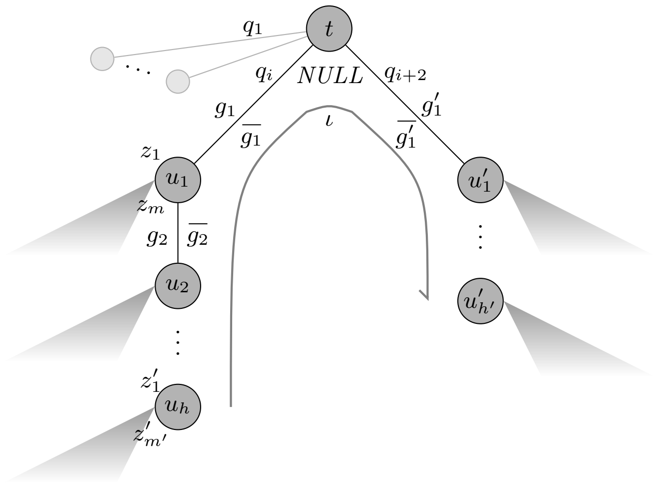


Fig. 8: The paths $u_1, \ldots, u_h$ and $u'_1, \ldots, u'_{h'}$ are inserted when q_i is a branch case. The triangles indicate the sub-trees that are inserted recursively.

both intersections in $\mathcal{X}(z_k, z_{k+1}, J'_k)$. Denote by ϕ_k these junction-instance-pairs. We recursively consider the node u_j as corresponding to the junction instance sequence $(\phi_1, \ldots, \phi_m)$, so that edges added to u_j in the recursion are inserted to the *left* of (u_j, u_{j+1}).

To see that the induction holds at node u_1, first observe that the last state in ϕ_k is equal to the first state in ϕ_{k+1}. Next, consider the first node u_1 at which junctions are inserted. The path from u_0 to u_1 is labeled with g_1 by forth. Let g_0 be the q_i-prefix in B guaranteed by the induction hypothesis. Due to the construction of $B_{K,q_i,q_{i+1}}$, the transition sequence n_1 in $B_{K,q_i,q_{i+1}}$ is also a transition sequence in B. Thus, the word $g_0 g_1$ is a z_1-prefix. This guarantees the induction hypothesis for the recursive step with ϕ_1. Inductively, it holds that the word y_1 read on the way to u_1 (in the pre-order traversal) from the previous leaf in the sub-tree corresponding to ϕ_m is a z_m-prefix in B. We may thus inductively consider $u_2, \ldots, u_{h-1}$ to obtain that at u_{h-1}, there is a $z'_{m'}$-prefix y_{h-1} in B, where $z'_{m'}$ is the last state considered when handling the junction sequence for u_{h-1}. Moreover, y_{h-1} is read on the way to u_{h-1} (in the pre-order traversal) from the previous leaf in the sub-tree corresponding to the pair $(z'_{m'-1}, z'_{m'})$. The induction at u_{h-1} is completed by the fact that $z_{m'}$ matches the first state in n_h, so that the $z_{m'}$-prefix y_{h-1} can be combined with the $z_{m'}$-suffix w_h to form a strand in $\mathcal{A}$.

The prefix of a (up to the transition $\delta(q_{i+2}, \varepsilon) = q'_i$) can be treated analogously, by inserting a path $(u'_0 = t, u'_1, \ldots, u'_{h'})$. The induction for the path $(u_0, \ldots, u_h)$ together with the induction for $(u'_0 = t, u'_1, \ldots, u'_{h'})$ yields two things. Firstly, that the back labels on the path from u_h's rightmost leaf to t concatenated with the forth labels on the path from t to $u'_{h'}$'s leftmost leaf form

ι; and secondly, that the path from u_1''s rightmost leaf to u_1' is back-labeled with a q_{i+2}-prefix. This concludes our effort to establish Lemma 2.

Lemma 3. Before presenting the missing proof, we describe precisely how the labels qmark are obtained. Consider a pre-order traversal of T. For convenience, we set $\delta(\mathit{NULL}, x) = \mathit{NULL}$ for all $x \in \Gamma^*$ and $\delta(q, \mathit{OPEN}) = \mathit{NULL}$ for all $q \in Q$. During the traversal, we keep track of a variable CUR_STATE which holds either a state from the set Q, or the value NULL. Initially, CUR_STATE is s, the starting state of B.

Before traversing an edge $e = (u, u_i)$ leading to the i^{th} child u_i of u (i.e., e is traversed in the direction away from the root), we assign the label $\text{qmark}(u, i) =$ CUR_STATE. Then, we update CUR_STATE $\leftarrow \delta($CUR_STATE$, \text{forth}(e))$ to continue our traversal at u_i. Before traversing an edge $e = (u_i, u)$ leading to u_i's parent in T (i.e., e is traversed in the direction back to the root) we assign the label $\text{qmark}(u_i, \deg(u_i) + 1) =$ CUR_STATE. Then, if u_i is not a leaf, we update CUR_STATE $\leftarrow \delta($CUR_STATE$, \text{back}(e))$. Otherwise u_i is a leaf, and we update CUR_STATE $\leftarrow \delta(s, \text{back}(e))$ if the strand obtained by reading the labels on the path to the next leaf is in $\mathcal{A}$, or CUR_STATE $\leftarrow \mathit{NULL}$ otherwise. The pre-order traversal then continues, visiting u again.

Proof (of Lemma 3). Part (i) is readily established by the observation that a path (u_1, u_2, u_3) with $\deg(u_2) = 1$ can be replaced by the edge $e = (u_1, u_2)$. The label forth(e) is the concatenation of forth(u_1, u_2) with forth(u_2, u_3), and the label back(e) is the concatenation of back(u_2, u_3) with back(u_1, u_2). To repair the state marks, simply apply the marking algorithm again.

To establish part (ii), let u be an inner node in T so that $\text{qmark}(u, i) = \text{qmark}(u, j) \neq \mathit{NULL}$ for some $1 \leq i < j \leq \deg(u)+1$, where $(i, j) \neq (1, \deg(u)+1)$. Denote by $e_1, \ldots, e_d$ the edges to u's children, and consider the tree T' obtained by removing all the sub-trees connected to u by $e_{i+1}, \ldots, e_j$. We claim that T' is an assembly of σ.

To see that this is true, consider the last leaf t_1 visited before e_i and the first leaf t_2 visited after e_j in a pre-order traversal of T. The back labels of the edges from t_1 to u correspond to a transition sequence from s to $\text{qmark}(u, i)$ in B, and the forth labels of the edges from u to t_2 correspond to a transition sequence in B from $\text{qmark}(u, j)$ to some $f \in F$. Since $\text{qmark}(u, i) = \text{qmark}(u, j)$, there is also a transition sequence from s to f labeled with the corresponding labels on the path from t_1 to t_2 (via u). Applying this technique inductively yields the desired claim. To repair the state marks, simply apply the marking algorithm again.

To establish part (iii), let T' be a state-marked assembly for σ satisfying both (i) and (ii) with the minimum number of nodes. Assume for the sake of contradiction that (iii) is false. This means that there is a path $p = (t_0, t_1, \ldots, t_l)$ in T' from the root t_0 to some leaf t_l such that $\text{qpair}(e) = \text{qpair}(e')$ for two edges e, e' on p, and NULL does not appear as an entry in $\text{qpair}(e)$. We argue that such a T' does not have a minimal number of nodes. Denote by $e = (x, y)$ and $e' = (x', y')$ the endpoints of the edges e and e', respectively, and assume w.l.o.g.

that e comes before e' on p. Denote by i and i' the index of e and e' among the edges to x's and x''s children, respectively. There are four cases, since one of $\mathrm{qmark}(x,i)$, $\mathrm{qmark}(x,i+1)$, and one of $\mathrm{qmark}(x',i')$, $\mathrm{qmark}(x',i'+1)$ could be *NULL*. We show how to deal with the case where none of the four is void; the other three cases can be handled in a similar manner.

Consider the tree T'' obtained by removing e, e', and the now disconnected sub-tree rooted at y, and inserting a new edge $f=(x,y')$ in the place of e among x's edges. The new edge is labeled $\mathrm{forth}(f)=\mathrm{forth}(e')$ and $\mathrm{back}(f)=\mathrm{back}(e')$. Clearly T'' has less nodes than T'. We claim that T'' is an assembly for σ. To see that, we only need to argue that the assembly properties hold at the modified nodes x and y'. Let (v,w,J) denote the junction $\mathrm{junct}(y')$ corresponding to y' in T', and let q_i, q_{i+1} be $\mathrm{qmark}(x,i)$ and $\mathrm{qmark}(x,i+1)$, where i is so that e is the i^{th} edge of x. Since $\mathrm{forth}(e')$ is a (q_i,v)-infix, and $\mathrm{back}(e')$ is a (w,q_{i+1})-infix, so are the labels assigned to f. This suffices to ensure that T'' is a state-marked assembly for σ, since no other parts in T'' are different from T'. Moreover, T'' still satisfies (i) and (ii), since node degrees only increased and the labels qmark were changed for neither x nor y'. This contradicts our assumption, thus concluding our proof for all three parts of Lemma 3.

Lemma 4.

Proof (of Lemma 4). Let T be an assembly for $\texttt{+}\not{o}$ in $\mathcal{A}\cup\{\iota\}$ as promised by Lemma 3. It is sufficient to show that $\iota\in\mathcal{X}(\delta(s,\texttt{+}),w,Q\times Q\setminus\{(v,w)\})$ for some w with $\delta(w,\not{o})\in F$. To see that this is true, denote by (t_1,t_2,t_3) the open path in T, i.e., the edges are labeled $\mathrm{forth}(t_1,t_2)=\texttt{+}$ and $\mathrm{forth}(t_2,t_3)=\not{o}$. It holds that the marking is $\mathrm{qmark}(t_1,1)=s$, $\mathrm{qmark}(t_3,1)=f$ for some $f\in F$. Denote by (v_2,w_2,J_2) the junction $\mathrm{junct}(t_2)$, and observe that $\delta(s,\texttt{+})=v_2$ and $\delta(w_2,\not{o})=f$, as desired, since T is an assembly. We establish the claim by showing that $\iota\in\mathcal{X}(v_2,w_2,Q\times Q\setminus\{(v_2,w_2)\})$. For that, we associate with each node x a junction instance, delivered by $\mathrm{qmark}(x,\circ)$, of the junction $\mathrm{junct}(x)$.

Again, the proof is by induction. For a node t in T, denote by $\varXi(t)$ the set of all descendants of t that are not leafs. The induction hypothesis at any node t is that for all $d\in\varXi(t)$, it holds that $\iota\in\mathcal{X}(\mathrm{junct}(d))$. The induction is based at all leaf nodes and their parents in T, for which the hypothesis holds vacuously.

For the induction step, consider a non-leaf node t. By the induction hypothesis, we have that $\iota\in\mathcal{X}(\mathrm{junct}(d))$ for all $d\in\varXi(t)$. Let (v,w,J) denote $\mathrm{junct}(t)$, and denote by $q=(q_1,\ldots,q_l)$ the corresponding junction instance guaranteed for t by property (ii) of Lemma 3. The goal is now to show that $\iota\in\mathcal{X}(v,w,J)$. For that, it suffices to show that $\iota\in L$ and $\iota\in R$, with

$$L=\bigcap_{\substack{1\le i\le l-1:\\ q_i\ne NULL\ne q_{i+1}}} C(q_i,q_{i+1},J)$$

and

$$R=\bigcap_{\substack{2\le i\le l-1:\\ q_i=NULL}} D(q_{i-1},q_{i+1},J)$$

If $q_i \neq NULL \neq q_{i+1}$, then we say that (q_i, q_{i+1}) is a C-case. If $q_i = NULL$, then q_{i-1} and q_{i+1} are both not $NULL$ and we say that (q_{i-1}, q_{i+1}) is a D-case.

We first show that for every C-case (q_i, q_{i+1}), it holds that $\iota \in C(q_i, q_{i+1})$. Associated with a C-case (q_i, q_{i+1}) in the sequence q, there is an edge $e = (t, d)$ so that $\mathrm{qpair}(e) = (q_i, q_{i+1})$. Denote by (z_1, z_2, J') the triple $\mathrm{junct}(d)$. If d is a leaf, then $z_1 \in F$ and $z_2 = s$, and thus $\iota \in C(q_i, q_{i+1}, J) = +\Sigma^*\emptyset$, as desired. Otherwise, if d is not a leaf, then $\mathrm{forth}(e)$ and $\mathrm{back}(e)$ are (q_i, z_1)- and (z_2, q_{i+1})-infixes, respectively. Thus, (z_1, z_2) is a pair in $H_{q_i,q_{i+1},J}$, and since $\iota \in \mathcal{X}(z_1, z_2, J')$, the input strand ι is also in the union $C(q_i, q_{i+1}, J)$.

It is left to show that for every D-case pair (q_{i-1}, q_{i+1}), it holds that $\iota \in D(q_{i-1}, q_{i+1})$. Associated with a D-case (q_{i-1}, q_{i+1}) in the sequence q, there are two successive edges $e = (t, d)$ and $e' = (t, d')$ so that $\mathrm{qpair}(e) = \mathrm{qpair}(e') = (q_i, q_{i+1})$. Let x denote the rightmost leaf in the sub-tree rooted at d, and let y denote the leftmost leaf in the sub-tree rooted at d', correspondingly. Denote by p the simple path $(x = u_1, u_2, \ldots, u_{j-1} = d, u_j = t, u_{j+1} = d', \ldots, u_k = y)$. We choose $K = \{(z_1, z_2) : \text{there is some } J' \text{ so that } (z_1, z_2, J') \text{ is a junction } \mathrm{junct}(u_i) \text{ for } 2 \leq i \leq k-1\}$. Note that indeed $K \subseteq J \setminus \{(q_{i-1}, q_{i+1})\}$ due to property (iii) of Lemma 3.

We have to show two things, namely that $\iota \in \overline{\mathcal{L}(B_{K,q_{i-1},q_{i+1}})}$, and that $\iota \in \mathcal{X}(z_1, z_2, J \setminus \{(q_{i-1}, q_{i+1}), (z_1, z_2)\})$ for all $(z_1, z_2) \in K$. The first claim can be established by considering the word $\overline{\iota}$, which can be obtained from concatenating the back labels from y to t, and the forth labels from t to x. From the construction of $B_{K,q_{i-1},q_{i+1}}$ we obtain that $\overline{\iota}$ is in $\mathcal{L}(B_{K,q_{i-1},q_{i+1}})$, and therefore $\iota \in \overline{\mathcal{L}(B_{K,q_{i-1},q_{i+1}})}$. For the second claim consider a pair $(z_1, z_2) \in K$. There is a node u_h on p for which $\mathrm{junct}(u_h) = (z_1, z_2, J')$, where $J' = J \setminus \{(q_{i-1}, q_{i+1}), (z_1, z_2)\}$. Thus, $\iota \in \mathcal{X}(z_1, z_2, J')$ by applying the induction hypothesis. We have established that the strand ι is in the intersections in $D(q_{i-1}, q_{i+1}, J)$ for K as chosen above. It thus follows that ι is also in the union. Since we have considered the whole sequence q, this concludes our proof of the lemma.

Theorem 2.

Proof (of Theorem 2). Let $B = (Q, \Gamma, \delta, s, F)$ be a deterministic finite automaton that accepts $\mathcal{L}$. Our goal is to construct an algorithm $\mathcal{A}$ that decides $\mathcal{L}$ by simulating B. The idea behind our construction of $\mathcal{A}$ is to mimic the transition function δ. For each possible transition $\delta(q, x) = q'$, with $q, q' \in Q$ and $x \in \Sigma$, the set $\mathcal{A}$ contains the *transition strand* $+q'\overline{x}\overline{q}\overline{+}$. Additionally, $\mathcal{A}$ contains the *starting strand* $+s\overline{+}$, for the starting state s, and the *accepting strands* $\overline{\emptyset}\overline{f}\emptyset$, for each state $f \in F$. For correctness we have to show two things. Firstly, that for every word $w \in \mathcal{L}$ the algorithm $\mathcal{A}$ accepts the word $+w\emptyset$, and secondly that any word $+w\emptyset$ accepted by $\mathcal{A}$, the word w is also accepted by B.

For the first part, let $t = (q_0, \ldots, q_l)$ be the transition sequence in B for any word $w \in \mathcal{L}$ with $q_0 = s$ and $q_l \in F$. Denote by $w_1 \ldots w_l$ the characters in w. We now argue that the effective strand $+\emptyset$ can be obtained by binding the strand $\iota = +w\emptyset$ with strands from $\mathcal{A}$, mimicking the transition sequence t. First, the starting strand $+s\overline{+}$ binds to ι, forming the new effective strand $\iota_0 = +sw_1 \ldots w_l\emptyset$.

We obtain the next effective strand ι_{i+1} by binding ι_i with the transition strand $\mathtt{+}q_{i+1}\bar{\mathtt{w}}_i\overline{q_i}\bar{\mathtt{+}}$, so that ι_{i+1} has the form $\mathtt{+}q_{i+1}w_{i+1}\dots w_l\mathtt{\emptyset}$. This is repeated until the strand $\mathtt{+}q_l\mathtt{\emptyset}$ is obtained, which then binds with $\bar{\mathtt{\emptyset}}\overline{q_l}\mathtt{\emptyset}$, yielding the desired effective strand $\mathtt{+\emptyset}$.

We begin our argument for the second part with the observation that strands in $\mathcal{A}$ cannot form the effective strand $\mathtt{+\emptyset}$ by themselves. Therefore, any assembly T for $\mathtt{+\emptyset}$ must involve at least one instance of the input strand. Moreover, such a T cannot involve more than one input strand, since each input strand introduces a delimiter $\mathtt{\emptyset}$, but the only strands that can match with $\mathtt{\emptyset}$, namely the accepting strands, also introduce a new unmatched delimiter $\mathtt{\emptyset}$. For this reason T must use exactly one accepting strand, and similarly exactly one starting strand.

Let $\iota = \mathtt{+}w_1\dots w_l\mathtt{\emptyset}$ be an input strand accepted by $\mathcal{A}$, and consider an assembly T for $\mathtt{+\emptyset}$. For any instance of ι in T, all symbols in $w = w_1\dots w_l$ must be matched, since only $\mathtt{+\emptyset}$ remains as effective strand. The only strands that contain domains from $\overline{\Gamma}$ are the transition strands, thus transition strands must be used to bind with each w_i. In particular, exactly l transition strands must be used in T, since less would leave unmatched symbols from Γ, whereas more would leave unmatched symbols from $\overline{\Gamma}$. Each transition strand t_i that binds with w_i introduces exactly two domains $\mathtt{p}$ and $\bar{\mathtt{q}}$ corresponding to states. To reach a valid assembly, two neighboring strands t_i, t_{i+1} must therefore bind along these domains. Since the first strand to match with ι is the starting strand, and because an accepting state can only be matched with an accepting strand, the assembly T therefore corresponds to a transition sequence in B for w.