Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 344 (2005) 279–297

Theoretical Computer Science

www.elsevier.com/locate/tcs

The computational power of Benenson automata

David Soloveichik*, Erik Winfree

Department of CNS, California Institute of Technology, MC 136-93, 1200 E. California Building, CA 91125, USA

Received 21 December 2004; received in revised form 29 July 2005; accepted 29 July 2005

Communicated by T. Yokomori

Abstract

The development of autonomous molecular computers capable of making independent decisions in vivo regarding local drug administration may revolutionize medical science. Recently Benenson et al. [An autonomous molecular computer for logical control of gene expression, Nature 429 (2004) 423–429.] have envisioned one form such a “smart drug” may take by implementing an in vitro scheme, in which a long DNA state molecule is cut repeatedly by a restriction enzyme in a manner dependent upon the presence of particular short DNA “rule molecules.” To analyze the potential of their scheme in terms of the kinds of computations it can perform, we study an abstraction assuming that a certain class of restriction enzymes is available and reactions occur without error. We also discuss how our molecular algorithms could perform with known restriction enzymes. By exhibiting a way to simulate arbitrary circuits, we show that these “Benenson automata” are capable of computing arbitrary Boolean functions. Further, we show that they are able to compute efficiently exactly those functions computable by log-depth circuits. Computationally, we formalize a new variant of limited width branching programs with a molecular implementation.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Molecular computing; DNA computing; Computational complexity; Restriction enzymes; Branching programs; Circuit complexity

1. Introduction

The goal of creating a molecular “smart drug” capable of making independent decisions in vivo regarding local drug administration has excited many researchers [10]. Recently,

* Corresponding author. Tel.: +1 626 395 6246.
E-mail addresses: dsolov@caltech.edu (D. Soloveichik), winfree@caltech.edu (E. Winfree).

Benenson et al. [4] (based on [5,3]) have envisioned what such an automaton may look like, and reported a partial implementation of the design in vitro. They made a system consisting of an enzyme and a set of DNA molecules which tests whether particular RNA molecules are present in high concentration and other particular RNA molecules are present in low concentrations, and releases an output DNA molecule in high concentration only if the condition is met. The actual computation process consists of the enzyme cutting a special DNA molecule in a manner ultimately determined by the concentrations of input mRNA molecules present in solution. The authors suggest that such a design, or a similar one, can be used to detect concentrations of specific mRNA transcripts that are indicative of cancer or other diseases, and that the output can take the form of a “therapeutic” ssDNA.

The key computational element in the scheme is an enzyme that cuts DNA in a controlled manner. Nature provides many biologically realizable methods of cutting DNA that can be adapted for computing. For instance, bacteria have evolved methods to cut the DNA of invading viruses (phages) with numerous enzymes called restriction enzymes. Most restriction enzymes cut double stranded DNA exclusively at sites where a specific sequence, called the recognition site, is found. Some restriction enzymes leave a so-called “sticky end overhang” which is a region of single stranded DNA at the end of a double stranded DNA molecule. Sticky ends are important because if there is another DNA molecule with a complementary sticky end, the two molecules can bind to each other forming a longer double stranded DNA strand.

Benenson et al. use type IIS restriction enzymes, which cut double stranded DNA at a specific distance away from their recognition sites in a particular direction [11]. These enzymes were first considered in molecular computation by Rothemund [9] in an non-autonomous simulation of a Turing machine. For an example of a type IIS restriction enzyme, consider *Fok*I which is known to cut in the manner shown in Fig. 1(a). Note that after *Fok*I cuts, the DNA molecule is left with a sticky end overhang of four bases. The automaton of Benenson et al. is based on a series of restriction enzyme cuts of a long *state molecule*. Each cut is initiated by the binding of a *cutting rule molecule* to the state molecule via matching sticky ends (Fig. 1(b)). Cutting rule molecules have an embedded restriction enzyme recognition site at a certain distance from their sticky end. The number of base pairs between the restriction enzyme recognition site and the sticky end on the cut rule molecule determines the number of bases that are cut away from the state molecule after the rule molecule attaches. Since the sequence of the sticky end on the state molecule determines which rule molecule attaches, it determines how many bases are cut off the state molecule in the presence of some set of rule molecules. Fig. 1(b) illustrates how TGGC can encode the “cut seven bases” operation when the appropriate cutting rule molecule is present. After each cut, a new sticky end is revealed which encodes the location of the next cut, and the process can continue.

Benenson et al. [4] describe how any set of RNA or DNA molecules can act as input to their automation. In particular, each input species converts some rule molecules that are initially inactive into active form, and inactivates others that are initially active. The net effect of multiple pre-processing steps is that the presence of input molecules in either high or low concentration determines which rule molecules will be available. Note that input is provided all at once, at the beginning of the computation; the activated rule molecules are used by the automaton as needed during the course of the computation.

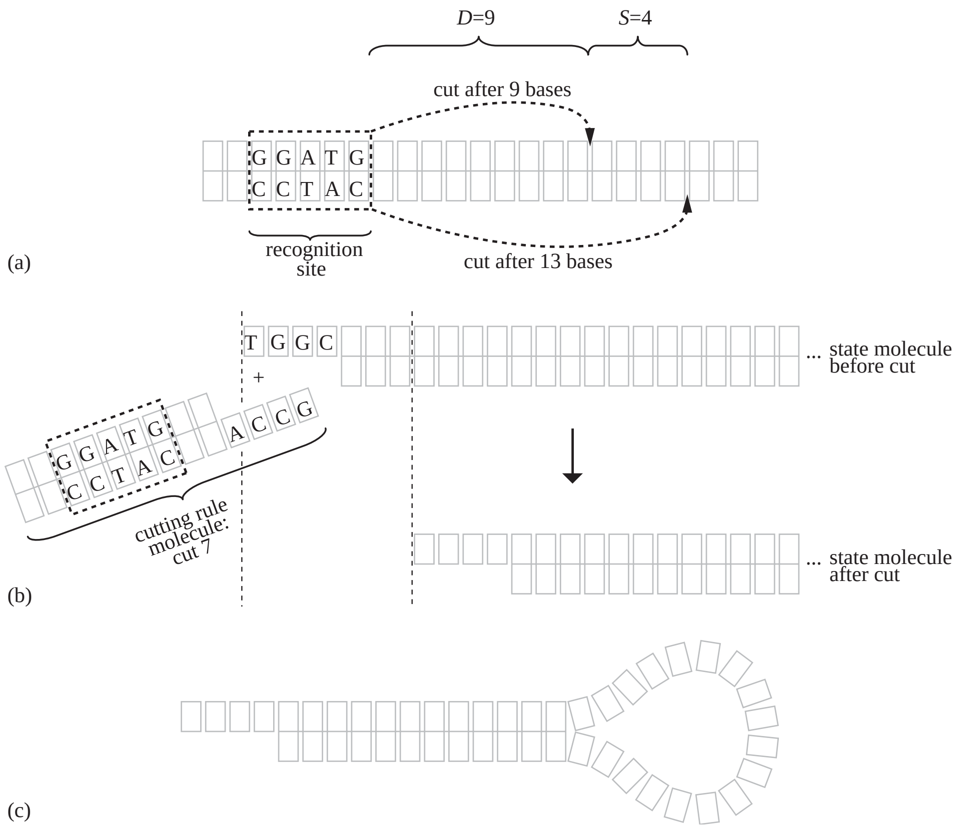


Fig. 1. (a) *Fok*I recognition and cut sites on a generic DNA substrate. The parameters *D* and *S* will be used to characterize restriction enzymes in this paper. *D* is called the cutting range and *S* the sticky end size. (b) Example of a cutting rule application. (c) Illustration of the output loop. Cutting far enough opens the loop. (In (a), (b), (c) the top strand is $5' \to 3'$.)

A single-stranded loop is attached to the end of the state DNA molecule (Fig. 1(c)). The loop is held closed by the remaining double stranded part of the state molecule— the so-called *stem*. If the state molecule is cut close enough to the loop, the loop is opened and released. Assuming the loop has a chemical function only when open (e.g. it is translated to create a protein or effectively acts as antisense DNA), this results in the production of the “therapeutic” molecule in an input-dependent manner. If the system worked without error, and supposing that the input RNA molecules are either present in high concentration or not at all, the output DNA molecule should be released if and only if a set of RNAs is present that results in a set of rule molecules that cut the state DNA molecule sufficiently far. To accommodate the possibility of error, which we ignore here, Benenson et al. implement two possible outputs that compete between each other, with the one produced in largest quantities “winning.”

We are interested in the class of computations that can be implemented using the approach developed by Benenson et al. [4]. One possibility of performing complex computations using this scheme is to use the output DNA molecule of one Benenson automaton as an input for another, allowing feed-forward circuits to be implemented. However, we would like to study the computational power of a system with a single state molecule. Showing how to compute complex functions with a single Benenson automaton examines the computational power of the basic unit of computation, and makes it clear how one can compute even more complex functions with many state molecules.

In the first part of this paper, we formalize the computational process implemented by Benenson et al. using a system with a single state molecule. As part of our abstraction, we are going to ignore concentration dependence and other analog operations such as those involving probabilistic competition between various reactions, and will focus on a binary model in which a reaction is either possible or not.[1] We treat the state molecule and the set of possible cutting rule molecules as a program specifying what computation is to be performed, while the input determines which rule molecules are active. Each rule molecule depends upon a specific input RNA species which either activates or deactivates it, or it may be always active. We will say that a Benenson automaton outputs 1 if at some point at least a total of p bases has been cut off, where p represents the point in the state string cutting beyond which opens the loop. Otherwise, we say it outputs a 0. Our constructions will cut the state molecule to leave no stem on a 1 output, and some length of stem otherwise.[2]

Like circuits, Benenson automata are best studied as a non-uniform computing model. But while the computational power of circuits is well characterized, the computational power of Benenson automata has not been studied. For example, while it was shown [4] how a single Benenson automaton can compute a conjunction of inputs (and negated inputs), it was not clear how a single Benenson automaton can compute a disjunction of conjunctions. While [5] and [3] show how finite automata can be simulated by a similar scheme,[3] a different input method is used. Here, we show that a Benenson automaton can simulate an arbitrary circuit, implying that it is capable of doing arbitrary non-uniform computation.

Lastly, we study the cost of implementing more complex computations (e.g. more complex diagnostic tests) using Benenson automata. While increasing the length of the state molecule is relatively easy and incurs approximately linear cost, increasing the size of the sticky ends or the range at which the restriction enzyme cuts requires discovering or creating new enzymes. Enzymes with very large cutting ranges that leave large sticky ends may not exist in nature, and while some success has been achieved in creating new restriction enzymes [6,7], engineering new restriction enzymes suitable for Benenson automata will require further technological advances.

[1] We will consider non-deterministic computation in which more than one cutting rule molecule can attach and cut. However, unlike [1] we will not assign probabilities to the various reactions and the output.

[2] Even with a stem remaining, the loop may still open at a certain rate (the "stem" must be long enough to keep the loop locked closed—see [4]). Nevertheless, our constructions can be modified to assure a longer remaining stem on a 0 output at the cost of using a few additional unique sticky ends (see Discussion).

[3] In contrast to [4], [1,3,5] treat the state molecule as an input string for a uniform computation, while the set of rule molecules is always the same and specifies the finite state machine computation to be performed. It is interesting to note the difference in the computational power of these two approaches. To implement a FSM with K symbols and N states, a type IIS restriction enzyme with cutting range N and sticky end size $O(\log KN)$ is sufficient and probably necessary.

Let us consider a family of boolean functions $\{f_n\}$, where $n = 1, 2, \ldots$ and $f_n : \{0, 1\}^n \to \{0, 1\}$. We show that any $\{f_n\}$ can be computed by a family of Benenson automata such that the size of the sticky ends grows only logarithmically with n and the range of enzyme cutting stays constant. (This is analogous to noting that any $\{f_n\}$ can be computed by a family of circuits using constant fan-in/fan-out, but it is non-trivial to prove.) If we restrict the length of the state molecule to be $poly(n)$, then the families of functions computable by these Benenson automata are exactly those computable by $\mathrm{O}(\log(n))$ depth circuits. These results are asymptotically optimal, since sticky end lengths must grow as $\log n$ in order to read all the input bits. We will also show that allowing the sticky end size to grow faster than $\mathrm{O}(\log n)$ does not increase computational power, and that allowing logarithmic cutting range cannot increase it significantly. Finally, we will define non-deterministic computation and prove that function families cannot be computed more efficiently using non-deterministic Benenson automata than deterministic ones.

Independent of the relevance of our formalization to biological computation, Benenson automata capture a model of string cutting with input-dependent cutting rules, and may be of interest as such.

2. Formalization of Benenson automata

We consider Benenson automata over a fixed alphabet Σ. For biological plausibility, one may want to consider $\Sigma = \{A, T, C, G\}$. However, our constructions assume only that $|\Sigma| \geqslant 3$. If so desired, all our results can be adapted to a binary alphabet by utilizing two bits to represent a single symbol, which entails changes in the constants used in the theorems.

Let $\mathbb{N}$ be the set of non-negative integers $\{0, 1, \ldots\}$. For any string $\sigma \in \Sigma^*$, $|\sigma|$ is the length of σ. For $j \in \mathbb{N}$ such that $j \leqslant |\sigma|$, we will use the notation $\sigma[j]$ to indicate the string that is left over after the first j symbols of σ are stripped off.

A Benenson automaton is parameterized by four numbers. Parameter n is the number of inputs that the automaton is sensitive to. Further, parameter S corresponds to the sticky end size, D to the maximum cutting range of the restriction enzyme (see Fig. 1(a)), and L to the length of the computational portion of the state molecule. A particular Benenson automaton is defined by specifying a state string σ and a selection of input-dependent cutting rules $\mathcal{R}$ as follows.

Definition 2.1. A *Benenson automaton* is a tuplet $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ where $n, S, D, L \in \mathbb{N}$, Σ is a finite alphabet, $\sigma \in \Sigma^L$ is a state string and $\mathcal{R} \subseteq \{0, \ldots, n\} \times \{0, 1\} \times \Sigma^S \times \{1, \ldots, D\}$ is a rule set using sticky ends of length S and maximum cutting distance D. Each rule (i, b, ω, d) specifies an input i, a binary value b, a sticky end ω, and a cutting distance d.

Interpreted as a DNA state molecule, $\sigma[j]$ represents the remaining portion of the molecule after j initial bases have been cut off. The first S symbols of $\sigma[j]$ represent the single stranded sticky end overhang. This revealed sticky end ω and the value of an input bit x_i determine where the next cut will be made by the application of some cutting rule (i, b, ω, d) which is applicable if $x_i = b$ and cuts at a distance d.

Definition 2.2. Given a Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$, for a binary input $x = x_1x_2 \dots x_n$, a rule $(i, b, \omega, d) \in \mathcal{R}$ *applies* to $\sigma[j]$, where $j \in \mathbb{N}$ s.t. $|\sigma[j]| \geqslant S + d$, if $x_i = b$ and ω is the initial *S* symbol portion of $\sigma[j]$. We write $\sigma[j] \rightarrow_x \sigma[j + d]$ iff there exists a rule $(i, b, \omega, d) \in \mathcal{R}$ that applies to $\sigma[j]$. Further, $\rightarrow_x^*$ is the reflexive transitive closure of $\rightarrow_x$.

Our definition of Benenson automata (as well as the biochemical implementation) allows for conflicting cutting rules. For example, if the rule set contains rules $(1, 0, \omega, 4)$ and $(2, 1, \omega, 6)$, then either four or six bases may be cut off if the sticky end ω is revealed and $x_1 = 0, x_2 = 1$. An important class of Benenson automata are those in which it is impossible for conflicting cutting rules to apply simultaneously.

Definition 2.3. A Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ is said to be *deterministic* if $\forall x \in \{0, 1\}^n$ and $j \in \mathbb{N}$ s.t. $\sigma \rightarrow_x^* \sigma[j]$, there exists at most one $j' \in \mathbb{N}$ such that $\sigma[j] \rightarrow_x \sigma[j']$.

While in computer science non-determinism often seems to increase computational power, we will see this is not the case with Benenson automata. On the other hand, implementing deterministic Benenson automata may be advantageous because (assuming error-free operation) each state molecule is cut up in the same way and thus there is no need for a combinatorially large number of state molecules.

Cutting the state string far enough indicates a 1 output. We will think of Benenson automata computing boolean functions as follows:

Definition 2.4. For $p \in \mathbb{N}$, we say that a Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ *non-deterministically computes* a boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ *at position p* if $\forall x \in \{0, 1\}^n$, $f(x) = 1 \Leftrightarrow (\exists j \in \mathbb{N}, p \leqslant j \leqslant |\sigma|$ s.t. $\sigma \rightarrow_x^* \sigma[j])$. We will say simply that the Benenson automaton *non-deterministically computes f* if such a *p* exists.

Definition 2.5. For $p \in \mathbb{N}$, we say that a Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ *computes* a boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ *at position p* if the automaton is deterministic and $\forall x \in \{0, 1\}^n$, $f(x) = 1 \Leftrightarrow (\exists j \in \mathbb{N}, p \leqslant j \leqslant |\sigma|$ s.t. $\sigma \rightarrow_x^* \sigma[j])$. We will say simply that the Benenson automaton *computes f* if such a *p* exists.

Other reasonable output conventions have the same computational power. For example, the following lemma shows that Benenson automata cutting to exactly *p* symbols to output a 1 and never cutting to exactly *p* symbols to indicate a 0, can be easily modified to output according to our convention.

Lemma 2.1. *If for a deterministic Benenson automaton* $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ *and* $f : \{0, 1\}^n \rightarrow \{0, 1\}$, $\exists p \in \mathbb{N}$, $p \leqslant L$ *s.t.* $\forall x \in \{0, 1\}^n$, $\sigma \rightarrow_x^* \sigma[p] \Leftrightarrow f(x) = 1$, *then there is a Benenson automaton* $(S, D, p + S, \Sigma, n, \sigma', \mathcal{R})$ *that computes f.*

An identical lemma also holds for non-deterministic computation. The lemma is trivially proven by taking σ' to be the first $p + S$ symbols of σ. All our constructions of Section 4 will produce Benenson automata requiring Lemma 2.1 to satisfy our definition of computing boolean functions (Definition 2.5).

Note that interpreted as a DNA state molecule, the length of the remaining state string minus S represents the remaining double-stranded stem holding the output loop closed. Thus, as mentioned in the Introduction, automata from our constructions (like any automata produced by the above lemma) leave no stem only on a 1 output, allowing the loop to open.

In a biochemical implementation, it may seem that in order to change the input (say from being all zeros to all ones) it may be necessary to activate or inactivate a rule molecule for every cutting rule in $\mathcal{R}$. However, for certain Benenson automata much smaller changes need be made. Consider the example of an automaton whose rule set contains the rules $(1, 0, \omega, d)$ and $(1, 1, \omega, d)$. This pair of rules is really a single input-independent rule to cut d bases if sticky end ω is found no matter what the input is; thus, the cutting rule molecule for it can be always active in solution. The following definition quantifies the maximum "amount of effort" needed the change the input for a given Benenson automaton.

Definition 2.6. For $s \in \mathbb{N}$, a Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ is said to be *s-encoded* if for every input bit i, $1 \leqslant i \leqslant n$, there are at most s sticky ends $\omega \in \Sigma^S$ such that $\exists(i, b, \omega, d) \in \mathcal{R}$ but $(i, 1-b, \omega, d) \notin \mathcal{R}$.

An s-encoded automaton has at most s sticky ends "reading" any given input bit. In order to change the input, in a biochemical implementation of a deterministic s-encoded Benenson automaton, it is enough to activate or inactivate at most s pairs of rule molecules per changed bit.

3. Characterizing the computational power of Benenson automata

In Section 4 we show that to compute function families using Benenson automata, only logarithmic scaling of the restriction enzyme sticky end size, and no scaling of the maximum cutting distance is needed. This result holds no matter what the complexity of the function family is. Further, if the family of functions is computable by log-depth circuits, [4] then a state string of only polynomial size is required. All of our constructions use deterministic Benenson automata.

Theorem 3.1.

(a) *Any function* $f : \{0,1\}^n \to \{0,1\}$ *can be computed by a Benenson automaton with sticky end size* $S = \mathrm{O}(\log n)$ *and maximum cutting distance* $D = \mathrm{O}(1)$.

(b) *Families of functions computable by* $\mathrm{O}(\log n)$ *depth circuits can be computed by Benenson automata with sticky end size* $S = \mathrm{O}(\log n)$, *maximum cutting distance* $D = \mathrm{O}(1)$, *and state string length* $L = poly(n)$.

The constants implicit in both statements are rather small. (In this and in the following discussions we assume that the alphabet size $|\Sigma|$ is a constant.) Note that the sticky end

[4] For the purposes of this paper, circuits are feed-forward and consist of AND, OR, and NOT gates with fan-in bounded by 2. For an introduction to circuit complexity see for example [8].

size cannot be smaller than $\mathrm{O}(\log n)$ since there must be at least a different sticky end for each input bit (otherwise the input is not completely "read"). Thus, in computing arbitrary boolean functions, we cannot do better than $S = \mathrm{O}(\log n)$ and $D = \mathrm{O}(1)$.

Further, in Section 5 we prove that our computation of families of functions computable by log-depth circuits is optimal, and neither allowing non-determinism nor larger sticky ends adds computational power:

Theorem 3.2. *Families of functions computable, possibly non-deterministically, by Benenson automata with $D = \mathrm{O}(1)$, $L = poly(n)$ can be computed by* $\mathrm{O}(\log n)$*-depth circuits*.

Corollary 3.1. *Benenson automata with $S = \mathrm{O}(\log n)$, $D = \mathrm{O}(1)$, $L = poly(n)$ can compute the same class of families of functions as* $\mathrm{O}(\log n)$*-depth circuits*.

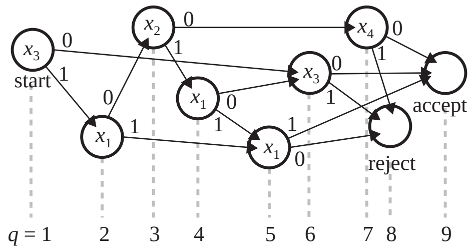
So if we consider only Benenson automata with $S = \mathrm{O}(\log n)$, $D = \mathrm{O}(1)$, $L = poly(n)$ efficient, then Benenson automata can compute a family of non-uniform functions efficiently if and only if it can be computed by a circuit of logarithmic depth. In Section 5, we will also show that relaxing this notion of efficiency to include logarithmic cutting range does not increase the computational power significantly.

4. Simulating branching programs and circuits

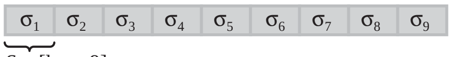
Benenson automata are closely related to the computational model known as branching programs. (For a review of branching programs see [12].) In the next section we show how arbitrary branching programs can be simulated. In the following two sections, we show how restricted classes of branching programs (fixed-width and permutation branching programs) can be simulated by Benenson automata with $S = \mathrm{O}(\log n)$ and $D = \mathrm{O}(1)$. Since fixed-width permutation branching programs are still powerful enough to compute arbitrary boolean functions (Section 4.4), Theorem 3.1(a) follows. Further, in Section 4.4 we will also see that fixed-width permutation branching programs of $poly(n)$ size can simulate $\mathrm{O}(\log n)$ depth circuits, implying Theorem 3.1(b).

4.1. General branching programs

A branching program is a directed acyclic graph with three types of nodes: variable, accept and reject (e.g. Fig. 2(a)). The variable nodes are labeled with an input variable x_i $(1 \leqslant i \leqslant n)$ and have two outgoing edges, one labeled 0 and the other 1, that lead to other (variable, accept or reject) nodes. The accept and reject nodes have no outgoing edges. One variable node with no incoming edges is designated the start node. The process of computation consists of starting at the start node and at every node x_i, following the outgoing edge whose label matches the value of the ith bit of the input. If an accept node is reached, we say that the branching program accepts the input x. Otherwise, a reject node is reached, and we say that the branching program rejects the input x. The function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ computed by a branching program is $f(x) = 1$ if x is accepted and 0 otherwise.



(a)

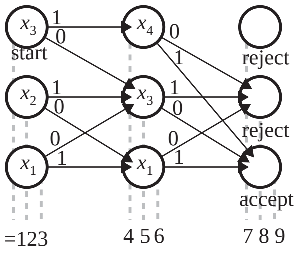


$S = [\log_{|\Sigma|} 9]$

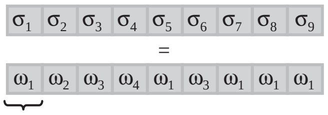
- Segments: All segments $\sigma_1, \ldots, \sigma_9$ are unique.
- Cutting rules:

σ_1: $(3, 0, \sigma_1, 5S)$, $(3, 1, \sigma_1, S)$
σ_2: $(1, 0, \sigma_2, S)$, $(1, 1, \sigma_2, 3S)$
σ_3: $(2, 0, \sigma_3, 4S)$, $(2, 1, \sigma_3, S)$
σ_4: $(1, 0, \sigma_4, 2S)$, $(1, 1, \sigma_4, S)$
σ_5: $(1, 0, \sigma_5, 3S)$, $(1, 1, \sigma_5, 4S)$
σ_6: $(3, 0, \sigma_6, 3S)$, $(3, 1, \sigma_6, 2S)$
σ_7: $(4, 0, \sigma_7, 2S)$, $(4, 1, \sigma_7, S)$

(b)



(c)



$S = [\log_{|\Sigma|} 4]$

- Segments:

$w_1 = \sigma_1 = \sigma_5 = \sigma_7 = \sigma_8 = \sigma_9$
$w_2 = \sigma_2$
$w_3 = \sigma_3 = \sigma_6$
$w_4 = \sigma_4$

- Cutting rules:

w_1: $(3, 0, w_1, 4S)$, $(3, 1, w_1, 3S)$
w_2: $(2, 0, w_2, 4S)$, $(2, 1, w_2, 3S)$
w_3: $(1, 0, w_3, 2S)$, $(1, 1, w_3, 3S)$
(d) w_4: $(4, 0, w_4, 4S)$, $(4, 1, w_4, 5S)$

Fig. 2. (a) An example of a general branching program of 9 nodes over 4 inputs and (b) the corresponding Benenson automaton. (c) An example of a width 3 branching program of 9 nodes over 4 inputs and (d) the corresponding Benenson automaton. Note that some nodes are inaccessible but these will be a small fraction for large programs. In both examples, $\sigma_1 \cdots \sigma_9 \to_x^* \sigma_9$ iff the branching program accepts x.

Because a branching program is a directed acyclic graph, we can index the nodes in such a way that we can never go from a node with a higher index to a node with a lower one (as shown in Fig. 2(a)). We can ensure that the first node is the start node and that there is only one accept node (convert all other accept nodes into variable nodes with all outgoing edges to this accept node). Let H be the total number of nodes in the given branching program. To each node with index $q \in \{1, \ldots, H\}$ we associate a unique string $\sigma_q \in \Sigma^*$ of length $S = \lceil \log_{|\Sigma|}(H) \rceil$. Let the state string σ be the concatenation of these segments in order: $\sigma_1 \ldots \sigma_H$. Thus, the size L of the state string is HS. For every variable node q labeled x_i, define $var(q) = i$. Further, for every variable node q, $goto_0(q) \in \{q+1, \ldots, H\}$ is the node targeted by the 0 outgoing edge and $goto_1(q) \in \{q+1, \ldots, H\}$ is the node targeted by the 1 outgoing edge of q. Using this notation, the rule set of our automaton consists of the following cutting rules. For every variable node q, there are two rules: $(var(q), 0, \sigma_q, (goto_0(q) - q)S)$ and $(var(q), 1, \sigma_q, (goto_1(q) - q)S)$. Depending on the branching program, the cutting distance may have to be as large as $(H-1)S$ if $goto_0(1) = H$ or $goto_1(1) = H$.

By construction, for any remaining portion of the state string $\sigma_q \cdots \sigma_{q'} \cdots \sigma_H$, we have that $\sigma_q \cdots \sigma_{q'} \cdots \sigma_H \rightarrow_x \sigma_{q'} \cdots \sigma_H$ iff the branching program goes to node q' from q on input x in one step. This implies that $\sigma_1 \cdots \sigma_q \cdots \sigma_H \rightarrow_x^* \sigma_q \cdots \sigma_H$ iff the branching program eventually goes from the start node to node q on input x. Thus, this Benenson automaton cuts to the beginning of the segment corresponding to the accept node iff the branching program accepts the input x. Thus, employing Lemma 2.1 (i.e. shortening the state string) we have a Benenson automaton computing the function f computed by the branching program. As there is exactly one outgoing edge from any variable node for each value of the read input bit, it follows that the resultant automaton is deterministic. See Fig. 2(a,b) for an example of a branching program and the corresponding Benenson automaton. Thus we have the following lemma:

Lemma 4.1. *For any function $f : \{0,1\}^n \rightarrow \{0,1\}$ computed by a branching program of H nodes and any alphabet Σ s.t. $|\Sigma| \geqslant 2$, there is a deterministic Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ with sticky end size $S = \lceil \log_{|\Sigma|}(H) \rceil$, maximum cutting distance $D = (H-1)S$, and state string length $L \leqslant HS$ computing f.*

Note that all three complexity parameters (S, D, and L) of Benenson automata needed to simulate general branching programs using the above construction scale with the size of the branching program. Thus, for families of functions for which the size of branching programs computing them increases very fast with n, new restriction enzymes must be developed that scale similarly. Consequently, this is not enough to prove Theorem 3.1(a).

4.2. Fixed-width branching programs

In this section, we demonstrate a sufficiently powerful subclass of branching programs whose simulation is possible by Benenson automata such that only the size L of the state string scales with the size of the branching program, while $S = \mathrm{O}(\log n)$ and $D = \mathrm{O}(1)$.

In the general case discussed in Section 4.1, our cutting range had to be large because we had no restriction on the connectivity of the branching program and may have needed to jump far. Further, we used a different sticky end for each node because there may be many different “connectivity patterns.” Restricting the connectivity of a branching program in a particular way permits optimizing the construction to significantly decrease S and D. In fact, both will loose their dependence on the size of the branching program. In the final construction, the sticky end size S will depend only on the size of the input n and the cutting range will be a constant.

A width J, length K branching program consists of K layers of J nodes each (e.g. Fig. 2(c)). The total number of nodes is $H = KJ$. We will think of J as a constant since for our purposes $J \leqslant 5$ will be enough. Nodes in each layer have outgoing edges only to the next layer, and every node in the last layer is either accepting or rejecting. We can ensure that the first node in the first layer is the start node and that the last layer has a single accept node. (Otherwise, the branching program can be trivially modified.) It turns out that width 5 branching programs are sufficiently powerful to simulate any circuit (Section 4.4). Further, the results of Section 5 ensure that we have not restricted our model of computation too much; more general Benenson automata cannot compute more efficiently.

Given a width *J* branching programs, we index nodes consecutively from each layer: the *j*th node in layer *k* obtains index $q = (k-1)J + j$. We use the same cutting rules as before, and construct the state string identically to the previous section, but with the following difference. Instead of using a unique segment for each node in the branching program as we did in the previous section, we let $\sigma_q = \sigma_{q'}$ iff $var(q) = var(q')$, $goto_0(q) - q = goto_0(q') - q'$ and $goto_1(q) - q = goto_1(q') - q'$. In other words, we allow the segments to be the same if their cutting rules have the same behavior. This does not change the behavior of the automaton but allows us to use fewer unique segments, thereby decreasing *S*. For a width *J* branching program, $goto_0(q) - q$ and $goto_1(q) - q$ range from 1 to $2J - 1$. So we need no more than $n(2J-1)^2$ different segments, which implies that at most we need $S = \lceil \log_\Sigma(n(2J-1)^2) \rceil$. (The segments corresponding to the accept and reject nodes can be anything as long as we cannot go from a reject node to the accept node. We can choose a segment such that $goto_0(q) - q, goto_1(q) - q \geqslant J$.) Note that the resultant automaton is $(2J-1)^2$-encoded as $goto_0(q) - q$ and $goto_1(q) - q$ range from 1 to $2J - 1$. Further, the maximum cutting distance needs to be at most $D = (2J-1)S$ since in the worst case we need to go from the first node of a layer to the last node of the next layer. See Fig. 2(c,d) for an example of how a fixed-width branching program can be converted to a Benenson automaton.

As a result of the above optimizations for fixed-width branching programs, the sticky end size *S* and the maximum cutting distance *D* loose their dependence on the length of the branching program *K*. Assuming the width *J* is fixed, this means that the choice of the restriction enzyme is independent of the size of the branching program and is dependent only on the number of input bits *n*.

Lemma 4.2. *For any function* $f : \{0,1\}^n \to \{0,1\}$ *computed by a branching program of width J and length K, and any alphabet* Σ *s.t.* $|\Sigma| \geqslant 2$, *there is a* $(2J-1)^2$*-encoded deterministic Benenson automaton* $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ *that uses sticky end size* $S = \lceil \log_\Sigma(n(2J-1)^2) \rceil$, *maximum cutting distance* $D = (2J-1)S$, *and state string length* $L \leqslant KJS$ *computing f.*

The constructions described above rely on being able to skip entire segments in a single cut. It seems that the cutting range must be at least logarithmic in *n*, since the size of the segments is logarithmic in *n* to be able to read all the input variables. However, in the following we describe the construction in which the maximum cutting distance *D* is dependent only on the width *J* and no longer on *n*, and is thus shorter than the segments. As before, we will still have that $\sigma = \sigma_1 \cdots \sigma_q \cdots \sigma_H \to_x^* \sigma_q \cdots \sigma_H$ iff the branching program eventually goes from the start node to node *q* on input *x*. However, while previously following a single arrow on the branching program corresponded to the application of a single cutting rule, now it will involve the application of many. We will separate the cutting rules into two logical types: *segment* cutting rules and *skip* cutting rules. If previously the applicable cutting rule removed $(goto_0(q) - q)$ or $(goto_1(q) - q)$ entire segments, now the corresponding segment cutting rule only removes $(goto_0(q) - q)$ or $(goto_1(q) - q)$ symbols from the beginning of the current segment σ_q. How can the cutting of *d* symbols from the beginning of a segment result in the eventual cutting of *d* entire segments? This is accomplished by the skip cutting rules as follows (see also Fig. 3).

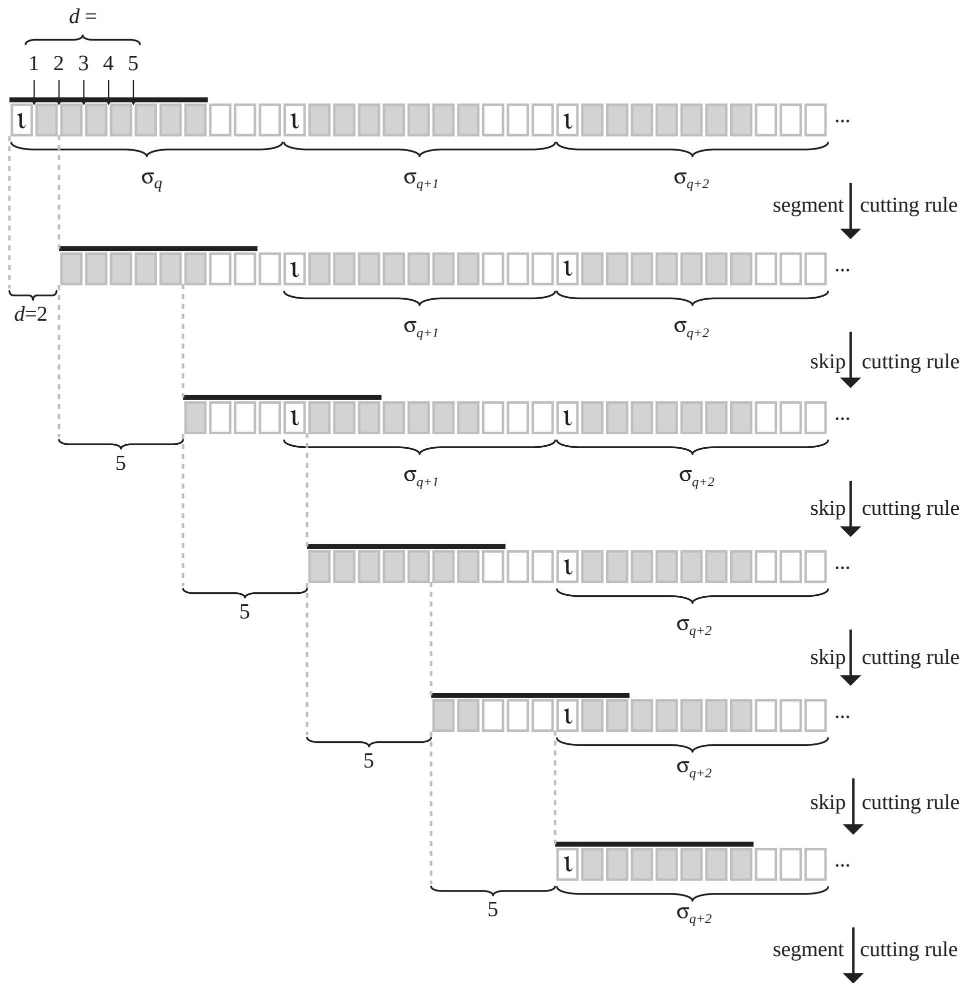


Fig. 3. An example of a segment cutting rule application and the subsequent application of skip cutting rules. In this case, $D = 5$, $k = 2$ and the size of the segments is $m = D \cdot k + 1 = 11$. The sticky end size is $S = 8$; the black horizontal lines above the state string show the sticky end in each step. The grayed squares comprise τ_q, τ_{q+1}, and τ_{q+2} that, together with a bit of input, determine which segment cutting rule is applicable. The empty white squares comprise v.

A new symbol $\imath \in \Sigma$ marks the beginning of each segment, while the rest of the segment uses symbols in $\Sigma - \{\imath\}$. A skip cutting rule is always applicable if the first symbol of the revealed sticky end is not $\imath$, while segment cutting rules are only applicable if the first symbol of the revealed sticky end is $\imath$. All skip cutting rules cut exactly D symbols. We use segments of length $m = D \cdot k + 1$ for some integer $k \geqslant 1$. After the application of some segment cutting rule removes d initial symbols of the state string, exactly $d \cdot k$ applications of skip cutting rules follow because after $d \cdot k \cdot D + d = d \cdot m$ symbols have been removed, it follows that *d* entire segments (each of length *m*) have been cut off and a new segment cutting rule is applicable. No segment cutting rule is applicable before then since this is the first time the first symbol of the revealed sticky end is $\imath$.

Formally, we use segments of the form $\sigma_q = \imath\tau_q\nu$ where $\tau_q, \nu \in (\Sigma - \{\imath\})^*$ and ν is an arbitrary string such that $|\sigma_q| = D \cdot k + 1$ for some integer $k \geqslant 1$. The strings τ_q are chosen such that $\tau_q = \tau_{q'}$ iff $var(q) = var(q')$, $goto_0(q) - q = goto_0(q') - q'$ and $goto_1(q) - q = goto_1(q') - q'$. For each variable node *q*, the segment cutting rules are: $(var(q), 0, \imath\tau_q, (goto_0(q) - q))$ and $(var(q), 1, \imath\tau_q, (goto_1(q) - q))$. Since we have at most $n(2J-1)^2$ unique τ_q's and we also need to read the $\imath$, we need the sticky end to be of size $S = 1 + \lceil \log_{|\Sigma|-1}(n(2J-1)^2) \rceil$. In the worst case, as before, we have $goto_{0/1}(q) - q = 2J - 1$ and so the maximum cutting distance needs to be $D = 2J - 1$ so that we can skip $2J - 1$ segments. Since the skip cutting rules should be independent of the input, for every $\omega \in \Sigma^S$ s.t. the first symbol of ω is not $\imath$, we can use the following two rules: $(1, 0, \omega, D)$ and $(1, 1, \omega, D)$. Note that since for both segment and skip cutting rules, there is at most one cutting rule applicable at any time, and because a segment cutting rule cannot be applicable at the same time as a skip cutting rule, it follows that our construction yields a deterministic Benenson automaton.

With the above trick (of course after applying Lemma 2.1), we have the following lemma for fixed-width branching programs:

Lemma 4.3. *For any function* $f : \{0,1\}^n \to \{0,1\}$ *computed by a branching program of width J and length K, and any alphabet* Σ *s.t.* $|\Sigma| \geqslant 3$, *there is a* $(2J-1)^2$*-encoded deterministic Benenson automaton* $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ *that uses sticky end size* $S = 1 + \lceil \log_{|\Sigma|-1}(n(2J-1)^2) \rceil$, *maximum cutting distance* $D = 2J - 1$, *and state string length* $L \leqslant KJS$ *computing f.*

Lemma 4.3 together with Barrington's theorem (Lemma 4.5) is enough to prove both parts of Theorem 3.1. However, we first optimize our construction even further to obtain better constants.

4.3. *Permutation branching programs*

We can obtain better constants if we restrict the branching program even more. Again, in the next section we will see that even with this restriction, branching programs can simulate circuits.

First, we need a notation for the context of layered branching programs. For node *j* in layer *k* let $goto_0(k, j) = j'$ if the j'th node in layer $k + 1$ is targeted by the 0 outgoing edge of this node; $goto_1(k, j)$ is defined analogously. A width *J* permutation branching program is a width *J* branching program such that for all layers *k*, the sequences $goto_0(k, 1), \ldots, goto_0(k, J)$ and $goto_1(k, 1), \ldots, goto_1(k, J)$ are permutations of $1, \ldots, J$. Further, there is exactly one accept node in the last layer (this can no longer be trivially assumed). It turns out that width 5 permutation branching programs are still sufficiently powerful to simulate any circuit (Section 4.4). In Section 5, we will confirm that we have not restricted our model of computation too much: efficient Benenson automata cannot simulate anything more powerful than permutation branching programs.

For permutation branching programs we can use fewer unique sequences for the τ_q's than for general fixed width branching programs. It is easy to see that for every permutation branching program, there is another permutation branching program of the same width and length that accepts the same inputs as the original program but for all layers k, $goto_0(k, \cdot)$ is the identity permutation (i.e. $goto_0(k, j) = j$). In this case, since $goto_0(q)-q$ is always J, we need at most $n(2J-1)$ unique τ_q's. Thus, sticky ends of size $S = 1+\lceil\log_{|\Sigma|-1}(n(2J-1))\rceil$ are sufficient and our automaton is $(2J-1)$-encoded. This leads to the following lemma:

Lemma 4.4. *For any function $f : \{0,1\}^n \to \{0,1\}$ computed by a permutation branching program of width J and length K, and any alphabet Σ s.t. $|\Sigma| \geqslant 3$, there is a $(2J-1)$-encoded deterministic Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ with sticky end size $S = 1+\lceil\log_{|\Sigma|-1}(n(2J-1))\rceil$, maximum cutting distance $D = 2J-1$, and state string length $L \leqslant KJS$ computing f.*

4.4. Simulating Circuits

While it may seem that fixed-width permutation branching programs are a very weak model of computation, it turns out that to simulate circuits, width 5 permutation branching programs is all we need:

Lemma 4.5 (*Barrington [2]*). *A function $f : \{0,1\}^n \to \{0,1\}$ computed by a circuit of depth C can be computed by a length 4^C width 5 permutation branching program.*

Corollary 4.1 (*of Lemmas 4.4 and 4.5*). *For any function $f : \{0,1\}^n \to \{0,1\}$ computed by a circuit of depth C, and any alphabet Σ s.t. $|\Sigma| \geqslant 3$, there is a 9-encoded deterministic Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ with sticky end size $S = 1+\lceil\log_{|\Sigma|-1}(9n)\rceil$, maximum cutting distance $D = 9$, and state string length $L = 4^C 5S$ computing f.*

This provides an alternative proof of Theorem 3.1 and implies, for instance, that a Benenson automaton using the restriction enzyme *Fok*I can do arbitrary 3-bit computation. Any increase in the sticky end size, exponentially increases the number of inputs that can be handled. If an enzyme is discovered that cuts 9 bases away like *Fok*I but leaves size 7 sticky ends, then it can do all 81-bit computation.

Letting $C = O(\log n)$, this proves Theorem 3.1(b). Theorem 3.1(a), of course, follows trivially since the complexity of the circuit (depth C) enters only in the length of the state string.

4.5. Achieving 1-encoded automata

If it is essential that the Benenson automaton be 1-encoded, the scheme from Section 4.3 can be adapted at the expense of slightly increasing the maximum cutting range D and the length of the state string L. The modification actually decreases the size of the sticky ends.

We provide a sketch of the construction; the details are carried over from the previous sections. The main idea is to use a pair of segments $\sigma_q = \imath\tau_q$ and $\sigma'_q = \imath\tau'_q$, where τ_q, $\tau'_q \in (\Sigma - \{\imath\})^*$, for each node q of the permutation branching program, rather than a single

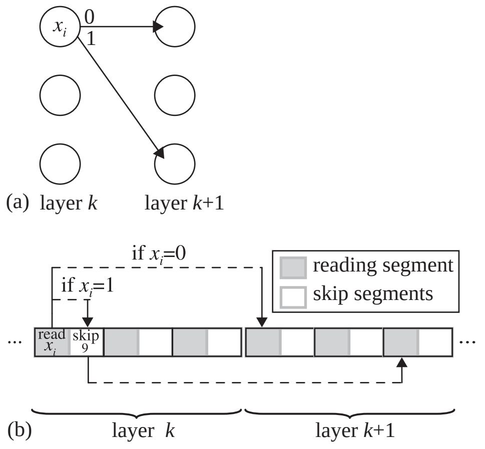


Fig. 4. Illustration of the construction achieving 1-encoded automata. (a) The portion of the branching program being simulated. In this case the width of the branching program is $J = 3$. (b) The relevant portion of the Benenson automaton. Note that each skip illustrated by the dashed lines consists of many cuts like those illustrated in Fig. 3.

segment as before (see Fig. 4). The first segment of the pair σ_q (the *reading segment*) reads the corresponding variable and skips either $2J$ segments if $x_i = 0$ or goes to the next segment if $x_i = 1$. Thus, the segment cutting rules for this segment are: $(i, 0, \imath\tau_q, 2J)$ and $(i, 1, \imath\tau_q, 1)$. Segment σ'_q (the *skip segment*) encodes an input-independent skip of $2(goto_1(q) - q) - 1$ segments to go to the correct reading segment. Thus, for the skip segment we can use the following segment cutting rules: $(1, 0, \imath\tau'_q, 2(goto_1(q) - q) - 1)$ and $(1, 1, \imath\tau'_q, 2(goto_1(q) - q) - 1)$. We need at most $n + 2J - 1$ unique segment types: n to read all the variables, and $2J - 1$ to be able to skip $2(goto_1(q) - q) - 1$ segments for all the values of $(goto_1(q) - q)$ which ranges from 1 to $2J - 1$. The maximum number of segments to skip is $2(2J - 1) - 1 = 4J - 3$. Note that there is at most one reading segment per input bit and thus the construction is 1-encoded.

Lemma 4.6. *For any function $f : \{0, 1\}^n \to \{0, 1\}$ computed by a permutation branching program of width J and length K, and any alphabet Σ s.t. $|\Sigma| \geqslant 3$, there is 1-encoded deterministic Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ with $S = 1 + \lceil \log_{|\Sigma|-1}(n + 2J - 1)\rceil$, $D = 4J - 3$, and $L \leqslant 2KJS$ computing f.*

This implies, for instance, that 1-encoded Benenson automata using restriction enzyme *Fok*I can simulate any width 3 permutation branching program over 22 inputs.

Corollary 4.2 (*of Lemmas 4.6 and 4.5*). *For any function $f : \{0, 1\}^n \to \{0, 1\}$ computed by a circuit of depth C, and any alphabet Σ s.t. $|\Sigma| \geqslant 3$, there is a 1-encoded deterministic Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ with $S = 1 + \lceil \log_{|\Sigma|-1}(n + 9)\rceil$, $D = 17$, and $L = 4^C 10S$ computing f.*

This implies, for example, that if a DNA restriction enzyme can be found that leaves sticky ends of size 4 like *Fok*I but cuts 17 bases away, then this enzyme can do all 18 bit computation with 1-encoded Benenson automata.

5. Shallow circuits to simulate Benenson automata

We will now show that our constructions from the previous section are asymptotically optimal.

Lemma 5.1. *A function* $f : \{0, 1\}^n \to \{0, 1\}$ *computed, possibly non-deterministically, by a Benenson automaton* $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ *can be computed by a* $\mathrm{O}(\log (L/D) \log D + D)$ *depth,* $\mathrm{O}(D4^D L)$ *size circuit.*

To see that this Lemma implies Theorem 3.2, take $D = \mathrm{O}(1)$, $S = \mathrm{O}(\log n)$, and $L = poly(n)$. Further, this Lemma implies that allowing non-determinism does not increase the computational power of Benenson automata. Likewise, note that sticky end size *S* does not affect the complexity of the circuit simulating a Benenson automaton. This implies that increasing the sticky end size to be larger than $\mathrm{O}(\log n)$ does not increase computational power.

Finally, Lemma 5.1 implies that Benenson automata using maximum cutting distance $D = \mathrm{O}(\log n)$, and state string length $L = poly(n)$ cannot be much more powerful than Benenson automata with $D = \mathrm{O}(1)$, and $L = poly(n)$. Specifically, $\forall \varepsilon > 0$, functions computable by Benenson automata with $D = \mathrm{O}(\log n)$, and $L = poly(n)$ are computable by $\mathrm{O}(\log^{1+\varepsilon} n)$ depth, $poly(n)$ size circuits.

Let us be given a Benenson automaton $(S, D, L, \Sigma, n, \sigma, \mathcal{R})$ computing, possibly non-deterministically, a boolean function *f* at position *p*. Observe that in order to check if, for a given input, the state string can be cut to or beyond *p*, it is enough to check if it can be cut to *p* or the following *D* symbols. The idea of our construction is that we split the state string into segments of length *D* and compute for all cut locations in every segment where the possible cuts in the next segment can be (for the given input). Then this information can be composed using a binary tree of matrix multiplications to reveal all possible cuts in the *D* symbols following *p* starting with the full state string. Making the segments shorter than *D* allows the possibility that a cut entirely bypasses a segment thereby fouling the composition, and making them longer than *D* makes the construction less efficient (i.e. results in a deeper circuit). This proof is similar to the argument that poly-length fixed-width branching programs can be simulated by log-depth circuits (e.g. [2]), in which the construction computes a binary tree of compositions of permutations rather than matrix multiplications.

For convenience let us assume *p* is divisible by *D* (say $Q = p/D$) and that $|\sigma| \geqslant p + D$. For q and $q' \in \mathbb{N}$, $q < q' \leqslant Q$, define a $D \times D$ binary matrix $T_{q,q'}(x)$ in which the *h*th bit (0 indexed) of the *j*th row (0 indexed) is 1 iff $\sigma[qD + j] \to_x^* \sigma[q'D + h]$. Observe that $T_{q,q'}(x) \times T_{q',q''}(x) = T_{q,q''}(x)$ where in the matrix multiplication $+$ is logical OR and $\cdot$ is logical AND. Therefore, $f(x) = 1$ iff there is at least one 1 in the 0th row of $T_{1,Q}(x) = T_{1,2}(x) \times T_{2,3}(x) \times \cdots \times T_{Q-1,Q}(x)$. If *p* is not divisible by *D* or $|\sigma| < p + D$, we can let the first or last of these matrices be smaller as necessary.

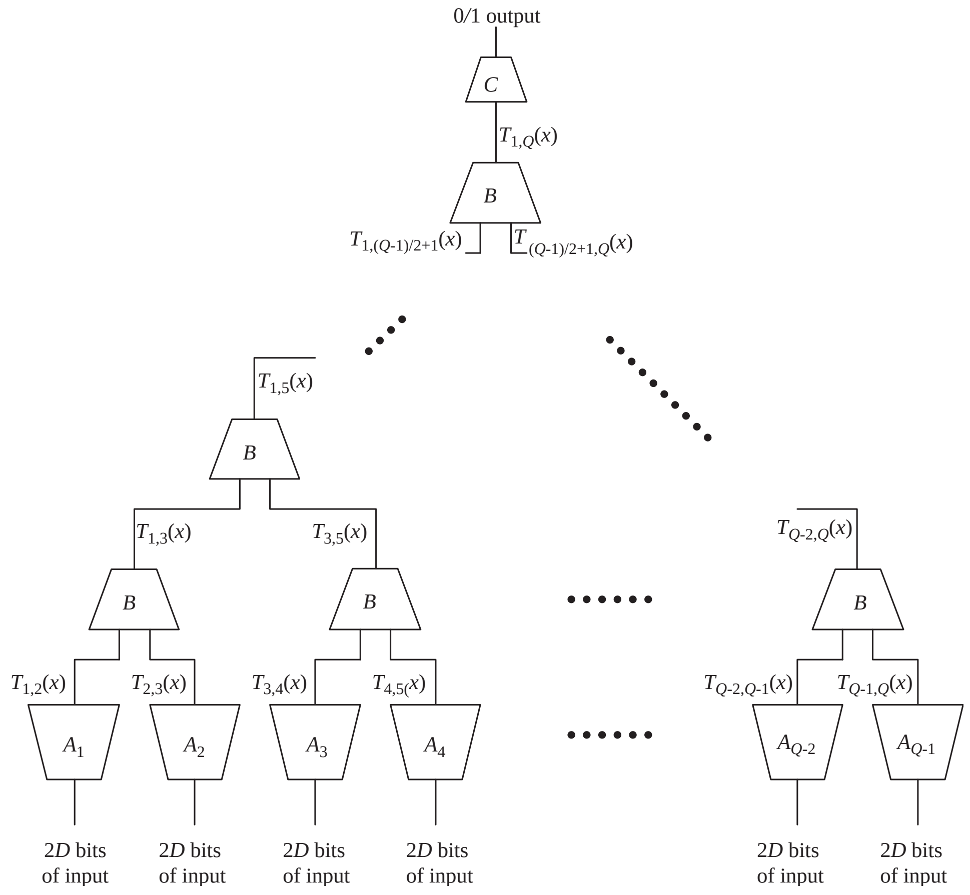


Fig. 5. Circuit outline for simulating a Benenson automaton. The T lines represent a bundle of at most D^2 wires. Input lines represent a bundle of at most $2D$ wires (a different subset for each gadget, possibly overlapping).

We can create a shallow binary tree computing the product $T_{1,Q}$. For clarity of exposition, let us assume that $Q-1$ is a power of 2. Our circuit consists of gadgets A_q $(1 \leqslant q \leqslant Q-1)$, gadgets B and gadget C (see Fig. 5). The input and output lines of gadgets B represent a matrix $T_{q,q'}(x)$ for a range of segments q to q' as shown in Fig. 5. To compute the initial series of matrices, each gadget A_q needs only to know at most $2D$ bits of input x on which $T_{q,q+1}(x)$ may depend (a segment of length D can read at most D input bits). Each gadget A_q can be a selector circuit that uses the relevant input bits to select one of 2^{2D} possible hardwired outputs (different for each A_q). These gadgets A_q have depth $O(D)$ and size $O(D^2 4^D)$. The output of gadget B is the product of its first input matrix by the second input matrix, where $+$ is logical OR and $\cdot$ is logical AND. Gadget B can be made of depth $O(\log D)$ and size $O(D^3)$. Gadget C outputs 1 iff there is at least one 1 in the 0th row of its input matrix.

6. Discussion

This work generalizes the non-uniform model of computation based on the work of Benenson et al. [4] and characterizes its computational power. We considered restriction enzymes with variable reach and sticky end size, and studied how the complexity of the possible computation scales with these parameters. In particular, we showed that Benenson automata can simulate arbitrary circuits and that polynomial length Benenson automata with constant cutting range are equivalent to fixed-width branching programs and therefore equivalent to log-depth circuits. We achieve these asymptotic results with good constants suggesting that the insights and constructions developed here may have applications.

There may be ways to reduce the constants in our results even further. Although the fixed-width permutation branching programs produced via Barrington's theorem have the same variable read by every node in a layer, this fact is not used in our constructions. Exploiting it may achieve smaller sticky end size or maximum cutting distance.

As mentioned in the Introduction, in a biochemical implementation of our constructions the last possible cut in the case that $f(x) = 0$ may have to be sufficiently far away from the output loop to prevent its erroneous opening. By using a few extra unique sticky ends we can achieve this with our constructions. For example, by adding one more unique sticky end corresponding to the reject states and making sure the accept state is last, we can ensure that in the constructions simulating general branching programs and fixed-width branching programs the last possible cut in the case $f(x) = 0$ is at least the length of a segment away ($\geqslant S, D$) from the last cut in the case $f(x) = 1$.

Some Benenson automata may pose practical problems for existing or future restriction enzymes not discussed in this paper. For example, a cutting rule with $d = 1$ would require a single base adjacent to a nick to be cleaved off each strand, which may not be biochemically plausible for certain restriction enzymes (a ligation enzyme may have to be used). Such issues must be considered carefully for an experimental implementation.

The major problem with directly implementing our construction is the potential of an error during the attachment of the rule molecule and during cuts. While a practical implementation of a Benenson automaton [4] has to work reliably despite high error rates, our formalization does not take the possibility of erroneous cutting into account. Further work is needed to formalize and characterize effective error-robust computation with Benenson automata. Similarly, while it is the easiest study of the binary model in which a reaction either occurs or not, a model of analog concentration comparisons may better match some types of tests implemented by Benenson et al.

Acknowledgements

We thank Georg Seelig for first bringing Benenson et al.'s work to our attention. Further, we thank two anonymous reviewers for very detailed reading of this paper and useful suggestions. This research was supported by NIH training Grant MH19138-15.

References

[1] R. Adar, Y. Benenson, G. Linshiz, A. Rosner, N. Tishby, E. Shapiro, Stochastic computing with biomolecular automata, Proc. Nat. Acad. Sci. 101 (2004) 9960–9965.

[2] D.A. Barrington, Bounded-width polynomial-sized branching programs recognize exactly those languages in NC^1, J. Comput. Systems Sci. 38 (1988) 150–164.
[3] Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, E. Shapiro, DNA molecule provides a computing machine with both data and fuel, Proc. Nat. Acad. Sci. 100 (2003) 2191–2196.
[4] Y. Benenson, B. Gil, U. Ben-dor, R. Adar, E. Shapiro, An autonomous molecular computer for logical control of gene expression, Nature 429 (2004) 423–429.
[5] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, E. Shapiro, Programmable and autonomous computing machine made of biomolecules, Nature 414 (2001) 430–434.
[6] T. Inui, H. Ikeda, Y. Nakamura, The design of an artificial restriction enzyme having simultaneous DNA cleavage activity, Nucelic Acids Symp. Ser. 44 (2000) 141–142.
[7] M. Komiyama, DNA manipulation using artificial restriction enzymes, Tanpakushitsu Kakusan Koso. 50 (2005) 81–86 (in Japanese).
[8] C.H. Papadimitriou, Elements of the Theory of Computation, Prentice-Hall, Englewood Cliffs, NJ, 1997.
[9] P.W.K. Rothemund, A DNA and restriction enzyme implementation of turing machines, in: DNA-Based Computers, Proceedings of a DIMACS Workshop, vol. 27, 1995, pp. 75–119.
[10] M.N. Stojanovic, T.E. Mitchell, D. Stefanovic, Deoxyribozyme-based logic gates, J. Amer. Chem. Soc. 124 (2002) 3555–3561.
[11] H. Sugisaki, S. Kanazawa, New restriction endonucleases from Flavobacterium okeanokoites (FokI) and Micrococcus luteus (MluI), Gene 16 (1981) 73–78.
[12] I. Wegener, Branching Programs and Binary Decision Diagrams, SIAM Monographs on Discrete Mathematics and Applications, 2000.