Recombinase-based designs for a permutron and an automaton

Swapnil Bhatia,[*,†] Craig Laboda,[‡] Jenhan Tao,[¶] and Douglas Densmore[†,§]

Department of Electrical and Computer Engineering, Boston University, Boston MA, Department of Electrical Engineering, Duke University, Durham NC, Department of Bioengineering, University of California – Berkeley, Berkeley CA, and Bioinformatics Department, Boston University, Boston MA

E-mail: swapnilb@bu.edu

KEYWORDS: recombinase, permutation, finite state machine

Abstract

We present schematic design algorithms for two recombinase-based computing devices called the *automaton* and the *permutron*. As formulated here, an automaton is a finite state device that decides whether a sequence of input signals is one of a set of sequences of input signals it is designed to recognize. Our design may be viewed as a generalization of the counter by Friedland *et al.*[1] The permutron is a device that allows for the set of its constituent parts to be reordered into any desired sequence, and may thus be viewed as a possible extension of the designs discussed by Ham *et al.*[2] We present algorithms for designing an automaton and a permutron, provide example designs, discuss the complexity of our algorithms and the challenges in building these devices.

[*]To whom correspondence should be addressed
[†]Department of Electrical and Computer Engineering, Boston University, Boston MA
[‡]Duke University
[¶]University of California — Berkeley
[§]Boston University Bioinformatics

Introduction

Engineering a new capability in an organism often requires engineering three types of genetic modules: a *sensing* module that receives inputs, a *control* module that implements a mapping from inputs to outputs, and a *response* module that produces an output. In many cases, such a control module implements a discrete computation: conventionally, this is a finite sequence of transitions in a state space, driven by the input. There have been several successful demonstrations of control modules which implement computations that are temporally commutative and idempotent — the output is invariant to the order of presentation of inputs and repeated application of inputs. Implementing computations lacking these properties, however, requires genetic computing devices that are "stateful," meaning they can encode the computation executed thus far, and continue computing based on this encoded history and the next input. The double inversion state machine,[2] the genetic counter,[1] and rewritable storage[3] are some examples of such devices.

A Deterministic Finite Automaton (DFA) is the simplest such stateful computing device. A DFA is defined by an input *alphabet* set, a finite *state space* set, a *start state*, a set of final states, and a *transition function* that maps a state and an input to a new state. If a sequence of inputs drives the DFA from a starting state to a final state, then that sequence is said to be *accepted* by the DFA. For example, an idealized version of the single-inducer Digital Invertase Cascade Counter (DIC)[1] is a DFA with a unary alphabet $\{Ara\}$, three states, 0, 1, and 2, and the transition function $\{f(0,Ara) \mapsto 1, f(1,Ara) \mapsto 2\}$. State 0 is the start state, state 2 is the final state and the only acceptable sequence is (Ara,Ara). Thus, the DIC DFA recognizes the set of sequences $\{(Ara,Ara)\}$ and indicates acceptance by transitioning to a state in which GFP is transcribed. Here, we present an algorithm that takes as input an alphabet A and a set F of acceptable sequences of symbols from the alphabet, and produces as output the abstract design for a DFA that only accepts sequences in F. Thus, our algorithm proposes DFA designs for recognizing a large class of sets of input sequences. Our algorithm composes a DFA design using modules like the one shown in Figure 3.[4]

Ham *et al.*[2] demonstrate a different stateful device which transitions through a "full rank" state

space defined by the orientation of DNA segments between nested invertase recognition sites. Ham *et al.* demonstrate a device with two promoters and a bidirectional terminator that is reconfigured when treated with various sequences of invertases. By controlling the location and orientation of these elements via inversions, the transcription of coding sequences, and hence the behavior of the device, can be controlled and reconfigured. Here, we generalize the problem of designing such a device. For a sequence of n elements, the full space of their configurations is described by the group of signed permutations on n elements. We present an algorithm which, given a sequence of genetic modules, outputs a device design composed of those modules with invertase sites interleaved such that there is a sequence of invertase treatments that can transition the device from its initial permutation of modules to any desired permutation of those modules. We call such a device that can realize any permutation a *permutron*, after *shufflons*.[5,6] Our algorithm is derived from algorithms for sorting a list of elements.

Methods

Automaton algorithm

We define an *alphabet* to be a finite nonempty sequence of symbols. For the purpose of our current discussion, it may be a useful example to think of an alphabet as a finite set of inducer molecule species. Any finite sequence over an alphabet we call a *string*.

A Deterministic Finite Automaton (DFA) is an abstract recognizer of a set of strings over an alphabet. More formally, a DFA D is defined by a quintuple (Q, A, δ, s, F) where Q is a finite nonempty set of states, A is an alphabet, $\delta : Q \times A \to Q$ is a transition function mapping a state and an input symbol to the next state, $s \in Q$ is the starting state of D, and $F \subseteq Q$ is the set of accepting states of D. Starting with the start state, and by iteratively applying the transition function, one obtains a path over the states of the DFA, comprising alternately of states and symbols from the alphabet. This is the computation performed by the DFA. For any such path, concatenating its input symbols defines a string. The set of all such strings defined by paths ending with a state

in F is the set of strings *recognized* or *accepted* by a DFA. We call this set of strings, *i.e.*, finite sequences of inducer species inputs to take the above example, the set of strings *acceptable* to the DFA.

Here, we only consider DFA that are acyclic, meaning there is no path in the DFA containing a state more than once, and present an algorithm for translating such a DFA into a recombinase-based genetic device. (We discuss the implications of acyclicity later.) The algorithm accepts as input a DFA as defined above, and produces as output the abstract sequence of part types defining the recombinase-based genetic device accepting the same set of strings as the DFA. We adopt the arbitrary convention that an acceptable string of inputs is indicated by the genetic device by a state in which GFP is expressed, and all other strings of inputs are indicated by a state in which GFP is not expressed. These states may be connected to downstream response modules, thus making the DFA a sophisticated control module capturing a class of common computations.

We describe an example design generated by the algorithm before we describe the algorithm. Let $\{a,b,c\}$ be an alphabet of three inducer species, and let the DFA shown in Figure 1 be the input DFA. The DFA has seven states with state 0 and state 5 defined to be the start and the set of accepting states, respectively. Tracing out all the paths between the start and the accepting state shows that the set of acceptable strings is $F = \{ac, abc, bab\}$. Given this input DFA, the algorithm produces a recombinase-based genetic device which responds to the inducer species in the alphabet, and expresses GFP if and only if exposed to inducer sequences from F. Thus, if the first inducer exposed to the device is of species a, then the device must move to a state in which it can process a c and accept, or process a b and wait for a c and accept. If the first inducer is a b, then it must move to a state in which it can process an a, and on receiving it, process a b and accept. If the first inducer is a c, then the device must move to a state from which no sequence of inducers can lead to the expression of GFP. This latter condition applies throughout the process: the device must transition to this "dead" state if any inducer other than the expected one is seen in any of the intermediate states. In the DFA, we show arcs that lead to the dead state in red and the dead state as a red diamond. (For clarity, we show the remaining arcs and the accepting state in green.) Once

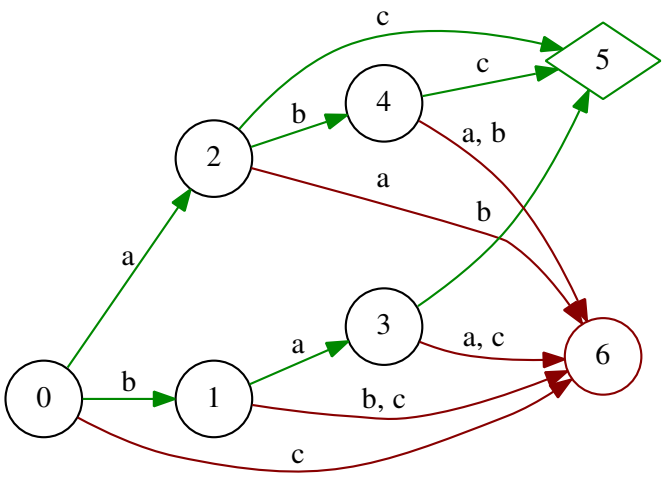

Figure 1: Example of a DFA

an accepting state or a "dead" state is reached, we consider the computation as having terminated and do not check for trailing symbols, though we could easily modify our design to include this feature.

In Figure 2, we show an intermediate abstract representation of the state machine shown in Figure 1. Each arc in the DFA is translated into a primitive computing device, like the one shown on the left in Figure 2. This device has three inputs and two outputs. The top two inputs are called *Enable* and *Disable*, left to right, respectively. The bottom two outputs are called *Enabler* and *Disabler*. The left input is the *Inducer* input. The device behaves as follows. If the Enable input is on, then both outputs are turned on (off) when the Inducer input is on (off). If the Disable input is on, then both outputs are turned off, regardless of the state of any of the other inputs.

This abstract machine implements the state machine in the following way. The current state of the DFA is encoded by having the Enable input of those primitive blocks that represent its outgoing arcs being turned on. This implies that when one of the possible Inducer input is applied, the Enabler and Disabler outputs of that primitive block are turned on. These Enabler outputs in turn enable the outgoing arcs of the new state and the Disabler outputs disable the other outgoing arcs from the old state. For example, when the inducer *"a"* is applied in state 0, the primitive block labled 02 produces its Enabler and Disabler outputs. These disable the arcs going from state 0 to states 1 and 6, and enable the outgoing arcs from state 2. In this way, state 2 becomes the current

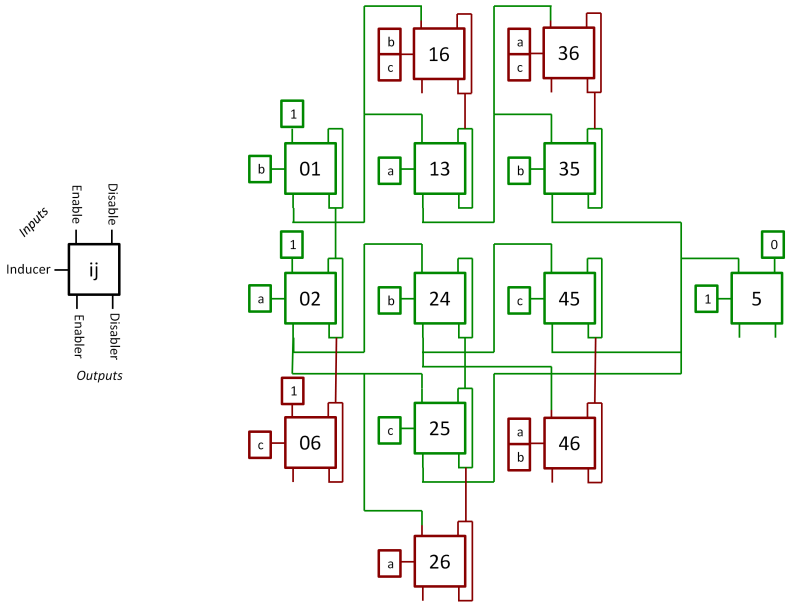

Figure 2: Abstract state machine circuit

state, and the process continues until either state 5 or state 6 is reached.

Algorithm 1 lists the three main steps in translating a DFA to a recombinase-based genetic device. Conceptually, each primitive block in the abstract machine is translated into a genetic module in the final design. The algorithm uses modules like the one shown in Figure 3 which we call a τ-*module* to implement a primitive block. The algorithm proceeds by examining each arc in the DFA and translating it into a τ-module. Each state in the DFA is represented by a τ-module in which the terminator t_j immediately downstream of the promoter p_i is excised. This corresponds to the Enable input of a primitive block in the abstract machine being turned on. The transition from one state to another in the DFA requires reading the next input symbol and following the arc labeled by it to a new state. This is implemented in the genetic device in the following way. The next input symbol is read by the device by having the promoter p_i in the τ-module for the current state be of a type that is activated by the molecule species corresponding to that symbol. The input-driven transition is encoded in the activatable promoter and in the terminator excision effected by the downstream recombinases expressed by the promoter. The transition into a state is implemented by the genetic device by excising the terminator immediately downstream of the

activatable promoter in the τ-module encoding that state. Once the promoter is activated, the recombinases excising the terminator from a new destination state is expressed. This corresponds to the Enabler output of a primitive block in the abstract machine being turned on. In a DFA, a transition from a state x to a state y also implies that other transitions that were possible from x are no longer possible when in state y. The algorithm encodes this in the genetic device by implementing excision machinery that excises the promoters from the τ-modules of all such states related to the previous state. This corresponds to the Disabler output and the Disable input of primitive blocks in the abstract machine.

Figure (put this figure in) shows the output of the algorithm on the DFA from Figure 1.

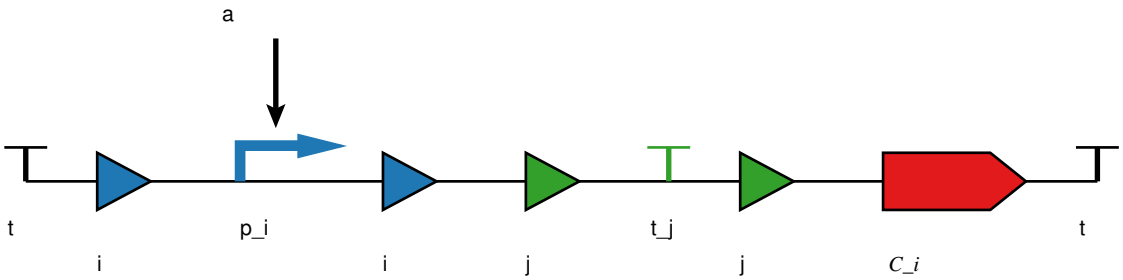

Figure 3: A module used by our DFA algorithm

Permutron algorithms

We now present an algorithm for designing a permutron comprising $n > 1$ DNA elements. The permutron requires two algorithms: one to design the permutron, and another to generate the invertase treatment sequence for achieving a desired permutation. We call the former algorithm the *design algorithm* and the latter the *key generating algorithm*. Our design algorithm produces a design that can utilize a sorting algorithm known as the "Pancake Sorting" algorithm for key generation. We first present this sorting algorithm.

Given a permutation of integers $1, \ldots, n$, the Pancake Sorting algorithm sorts the permutation by prefix reversals. That is, in each move, the algorithm is allowed to reverse a prefix, *i.e.*, if an element is in the prefix, then so is every element before it, of the current permutation. For example,

Algorithm 1 DETERMINISTIC FINITE AUTOMATON GENERATOR

Require: Deterministic acyclic finite automaton $\mathscr{A} = (Q, \Sigma, \delta, s, F)$

$r \leftarrow 0$, $i \leftarrow 0$
$A \leftarrow \varepsilon$
{For each state in the DFA}
for each state s in Q **do**

  {Step 1: Create τ-modules for each arc.}
  {For each arc from the current state}
  for each arc $e_i = (s, y, \alpha)$ **do**
    {Add new distinct recombinase sites to flank the promoter and the terminator of the τ-module}
    $p_i \leftarrow r$, $t_i \leftarrow r+1$
    {Add the recombinase coding sequence to self-excise the promoter in the τ-module}
    $C_i \leftarrow C_i \cup p_i$
    $r \leftarrow r+2$
  end for

  {Step 2: Tweak τ-modules to excise every other arc except the one chosen by the input.}
  {For each arc e_i from a state s}
  for each arc $e_i = (s, y, \alpha)$ **do**
    {For every other arc e_j from the same state s}
    for each arc $e_j = (s, w, \alpha) \neq e_i$ **do**
      {Add the recombinase coding sequence that excises the promoter from arc e_j, to the τ-module for current arc, i.e., e_i}
      $C_i = C_i \cup \{p_j\}$
    end for
  end for

  {Step 3: Tweak τ-modules to excise terminators from downstream τ-modules enabled by the current input.}
  for each pair of arcs $e_i = (x, s, \alpha)$ and $e_j = (s, y, \beta)$ **do**
    {If arc e_i and e_j are connected by a state, then add the recombinase coding sequence that excises the terminator from the τ-module representing e_j to the τ-module for arc e_i}
    $C_i \leftarrow C_i \cup \{t_j\}$
  end for

  $T_i \leftarrow \tau\text{-}module(\alpha, p_i, t_i, C_i)$ {Generate the τ-module}
end for
Output $T_1 \cdot T_2 \cdot \cdots \cdot T_{|\delta|}$

given the permutation $(1,3,2)$ on three elements, the algorithm would proceed as follows to sort it to the permutation $(1,2,3)$: $(1,3,2) \rightarrow (3',1',2) \rightarrow (2',1,3) \rightarrow (1',2,3) \rightarrow (1,2,3)$. (We use the "′" mark to indicate an inverted element.) If we reverse this sequence of transitions, then we obtain a way of achieving the permutation $(1,3,2)$ starting from the initial permutation $(1,2,3)$. As illustrated by this example, it can be shown that there is a sequence of transitions from $(1,2,3)$ to any of its permutations, and more generally, that this holds in the group of n elements.

It is straightforward to show that $2n-3$ prefix inversions are sufficient to sort a permutation on n elements using the above algorithm. This is because the algorithm brings each part to the head of the permutation using one inversion, and then moves it to its appropriate place using another inversion. This process is repeated $n-2$ times, and the last two elements can be reoriented with a single inversion. While more complex algorithms[7,8] can sort any permutation in about $1.6n$ inversions, we do not explore here the feasibility of developing a design algorithm that can use them for key generation.

We now describe informally an abstract design produced by the design algorithm, before describing the design algorithm formally. The design algorithm uses the "pancake inversion motif" shown in Figure 4 as a building block for enabling the inversions needed by the pancake sorting algorithm. Each instance of this motif comprises two nested pairs of invertase sites recognized by distinct invertases, illustrated by the green and blue sites in Figure 4. The algorithm assumes that any element could participate in any of the $2n-3$ sufficient inversions, and therefore composes $2n-3$ such motifs instances per element. Two such motif instances i and $i+1$ ($1 \leq i, i+1 \leq 2n-3$) are composed by nesting the green sites rightward: the left green site of instance $i+1$ is appended to the left green site of instance i and the right green site of instance $i+1$ is appended to the right green site of instance i. The blue sites of instance i are enclosed within the blue sites of instance $i+1$. The red pair of sites are also distinct and appear only once per element. We now describe the purpose of these sites in a motif instance. The green sites are used to bring the element to the head of the permutation. The blue sites are used to ensure that this "bring to head" machinery is always arranged in the correct orientation, regardless of other inversions. The red sites are used to flip the

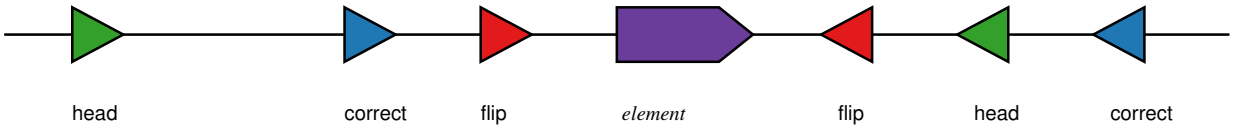

Figure 4: An instance of the inversion motif used by the design algorithm

orientation of the individual element in the final permutation, if necessary.

Figure 5 shows an abstract three-element example design as described above marked as "Initial." The initial design has the permutation $(1,2,3)$ of elements. (In this figure, these elements are shown as coding sequences, but these could be any segments of DNA.) The figures below show the sequence of transitions needed to reach the permutation $(3,1,2)$. Treating the initial design with the invertase recognizing the site "head32" inverts and brings element 2 to the head of the permutation. Treating this design with the invertase recognizing the site "head23" brings element 2 to the end of the permutation, which is its final place in the desired permutation. It turns out that all elements are in the corrent position, though element 3 is in an incorrect orientation. This is fixed by treating the design with invertase "flip3" which flips element 3 into the correct orientation.

More formally, let $P = \{1,\ldots,n\}$ be a set of part identifiers. (We will refer to these as parts for brevity.) A *permutation on P* is a sequence $\sigma_1,\sigma_2,\ldots,\sigma_n$, where each $\sigma_i \in P$ and $\sigma_i \neq \sigma_j$ whenever $i \neq j$, $1,\leq i,j \leq n$. The permutation $1,2,\ldots,n$ is called the *identity* permutation, denoted by ι. Let $\sigma = \sigma_1,\ldots,\sigma_n$ denote a permutation of n parts.

Algorithm 2 PERMUTRON DESIGN

Require: Set of DNA elements $\{e_1,\ldots,e_n\}$, $n > 1$.

$\tau \leftarrow \varepsilon$
for $i = 1,\ldots,n$ **do**
  $e_i \leftarrow \mathtt{flip_i} \cdot e_i \cdot \mathtt{flip_i}'$
  for $j = 1,\ldots,2n-3$ **do**
    $\tau \leftarrow \tau \cdot \mathtt{head}_{ij}$
    $e_i \leftarrow e_i \cdot \mathtt{head}'_{ij}$
    $e_i \leftarrow \mathtt{correct_{ij}} \cdot e_i \cdot \mathtt{correct}'_{\mathtt{ij}}$
  end for
end for
return $\tau \cdot e_1 \cdot e_2 \cdot \cdots \cdot e_n$

The Key Generation algorithm is as shown below:

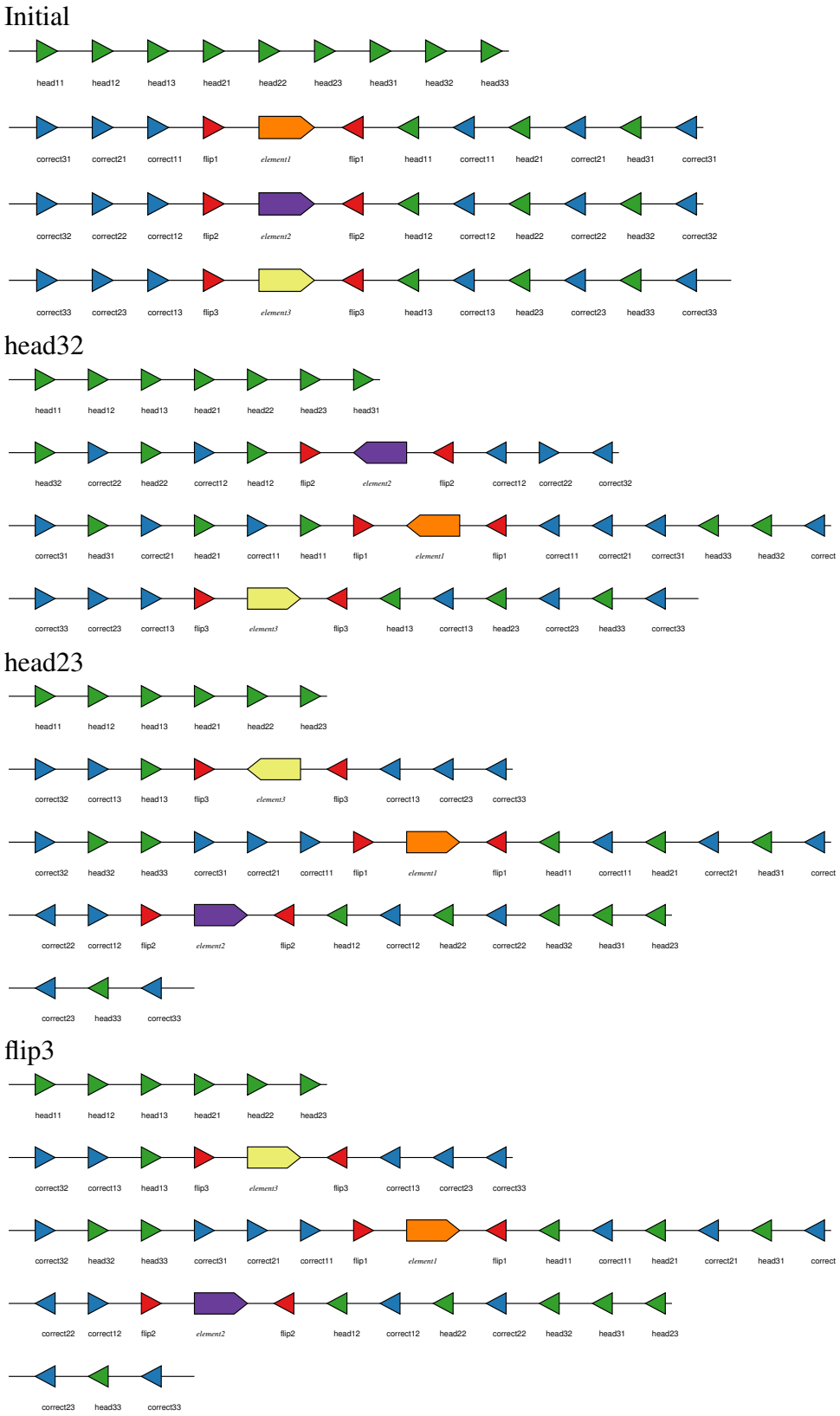

Figure 5: A three-element example output by the design algorithm

Algorithm 3 KEY GENERATOR

Require: Permutron design $\tau \cdot e_1 \cdot e_2 \cdot \cdots \cdot e_n$, $\sigma : \{1,\ldots,n\} \to \{1,\ldots,n\}$, a bijection

1: **for** $i = 1,\ldots,n-1$ **do**
2: For $1 \leq j \leq n-i+1$, find the largest value $\sigma(j)$.
3: **if** $j > 1$ **then**
4: Reverse the prefix ending at $\sigma(j)$.
5: **end if**
6: Reverse the prefix ending at $\sigma(n-i+1)$.
7: **end for**

Results and Discussion

Automaton results

We demonstrate our automaton design and the design algorithm by presenting the design for building a DFA recognizing any set of strings up to three symbols in length, over a binary alphabet $A = \{a,b\}$. As illustrated in Figure 6, there are 15 possible strings of length no greater than three over this alphabet. There are, thus, $2^{15} = 32,768$ possible sets of strings $\{\}, \{a\}, \{b\}, \{aa\}$, $\{ab\}, \{ba\}$, $\{bb\}, \{aa,ab\}, \{aa,ba\}, \ldots, \{a,b,aa,ab,ba,bb,aaa,aab,\ldots,bbb\}$ that could be used to design a DFA recognizing the given set of strings.

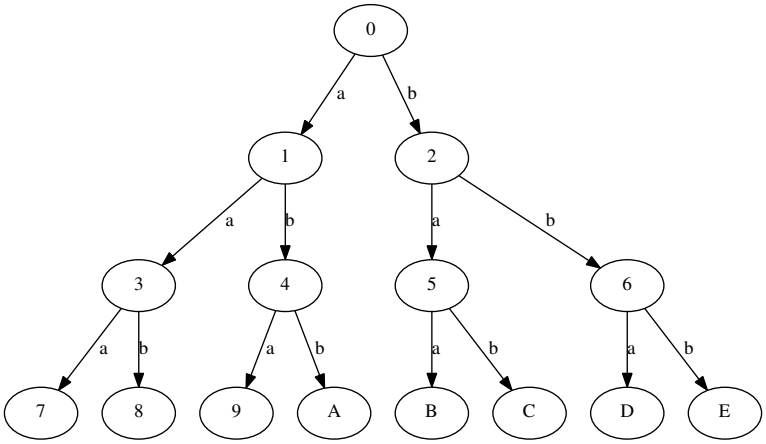

Figure 6: A tree of all strings of upto three symbols on a binary alphabet.

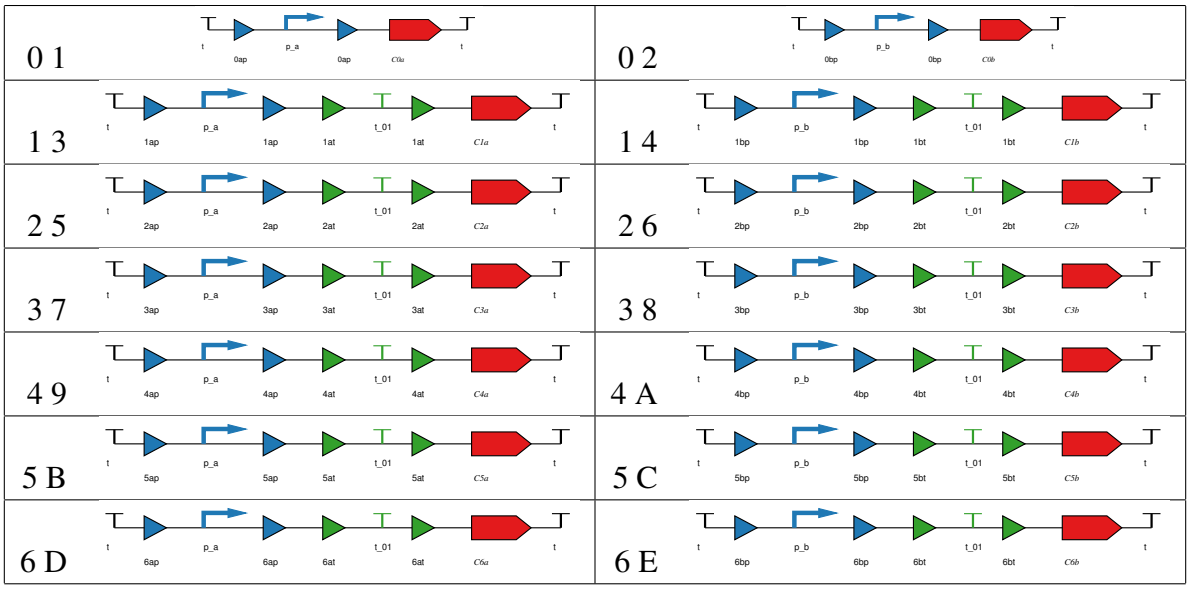

Figure 7: All τ-modules needed to realize the tree in Figure 6.

Recombinase set	recognizes
C0a	0ap, 0bp, 1at, 1bt
C0b	0ap, 0bp, 2at, 2bt
C1a	1ap, 1bp, 3at, 3bt
C1b	1ap, 1bp, 4at, 4bt
C3a	=7
C3b	=8
C4a	=9
C4b	=A
C2a	2ap, 2bp, 5at, 5bt
C2b	2ap, 2bp, 6at, 6bt
C5a	=B
C5b	=C
C6a	=D
C6b	=E

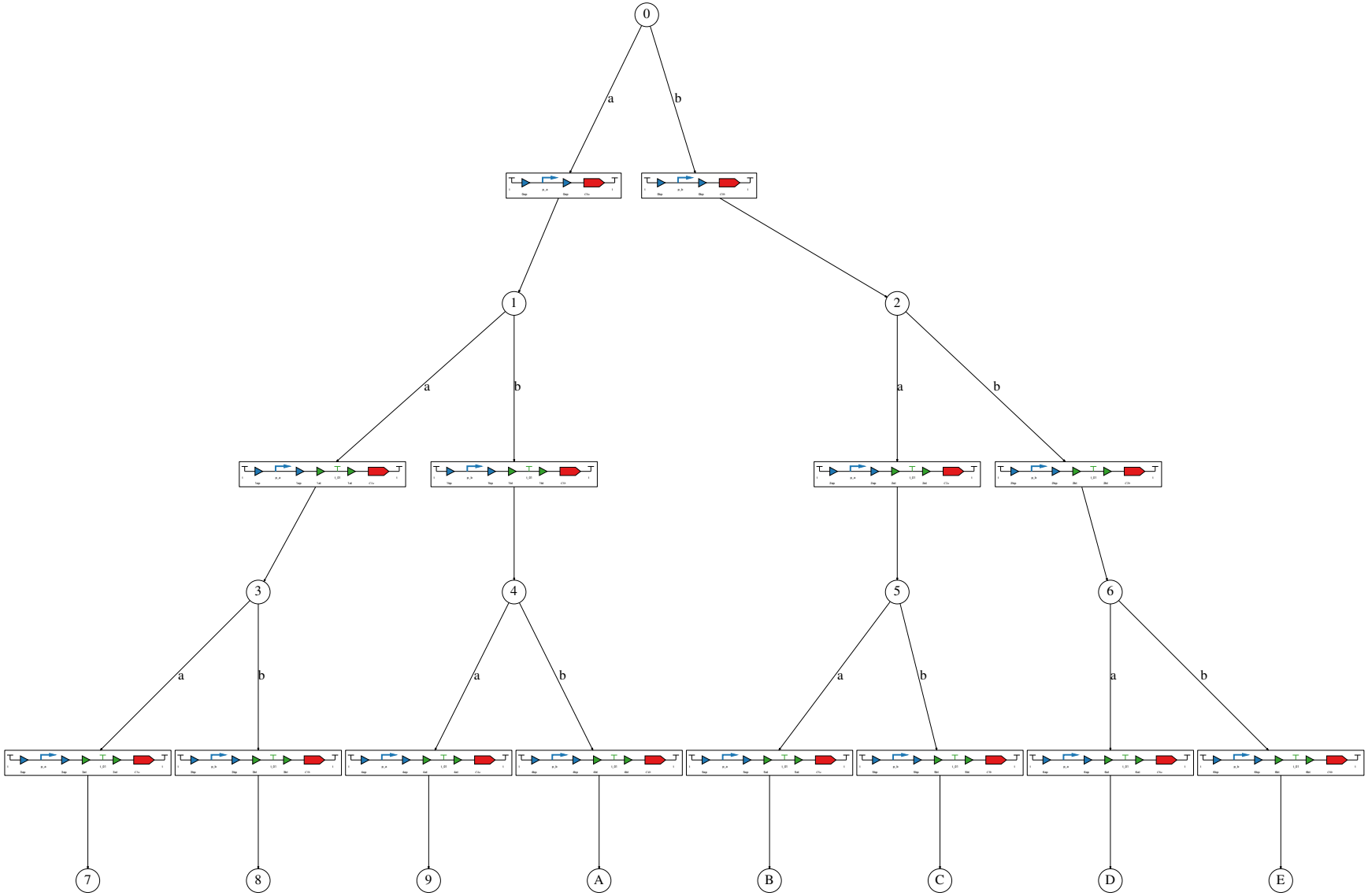

Figure 8: All τ-modules needed to realize the tree in Figure 6.

References

(1) Friedland, A. E.; Lu, T. K.; Wang, X.; Shi, D.; Church, G.; Collins, J. J. *Science* **2009**, *324*, 1199–1202.

(2) Timothy, S. H.; Lee, S. K.; Keasling, J. D.; Arkin, A. P. *PLoS One* **2008**, *3*, e2815.

(3) Bonnet, J.; Subsoontorn, P.; Endy, D. *Proceedings of the National Academy of Science, USA* **2012**,

(4) Siuti, P.; Yazbek, J.; Lu, T. K. *Nature Biotechnology* **2013**, *31*, 448–452.

(5) Komano, T. *Annual Review of Genetics* **1999**, *33*, 171–191.

(6) Dworkin, J.; Blaser, M. J. *Microbiology* **1997**, *94*, 985–990.

(7) Gates, W. H.; Papadimitriou, C. *Discrete Mathematics* **1979**, *27*, 47–57.

(8) Chitturi, B.; Fahle, W.; Meng, Z.; Morales, L.; Shields, C. O.; Sudborough, I. H.; Voit, W. *Theoretical Computer Science* **2009**, *410*, 3372–3390.