ISSN 1330-3651
UDC/UDK 519.713:004.383

ALGORITHMS AND DATA STRUCTURES FOR THE MODELLING OF DYNAMICAL SYSTEMS BY MEANS OF STOCHASTIC FINITE AUTOMATA

Robert Logozar

Original scientific paper

We elaborate on the design and time-space complexity of data structures and several original algorithms that resulted from the DSA program – a tool that simulates dynamical systems and produces their stochastic finite automata models according to the theory of ϵ-machines. An efficient batch iteration algorithm generates the system points and their binary symbols, and stores them in circular buffers realized as class member arrays. The words extracted from the time series are fed into a dynamically created binary tree of selected height. The tree is then searched for morphologically and stochastically unique subtrees or morphs, by aid of an algorithm that compares (sub)trees by their topology and the descendant nodes' conditional probabilities. The theoretical analysis of algorithms is corroborated by their execution time measurements. The paper exemplifies how an implementation of a scientific modelling tool like the DSA, generates a range of specific algorithmic solutions that can possibly find their broader use.

Keywords: *algorithms, data structures, time-space complexity, dynamical systems, stochastic finite automata*

Algoritmi i podatkovne strukture za modeliranje dinamičkih sustava s pomoću stohastičkih konačnih automata

Izvorni znanstveni članak

Razmatramo dizajn i vremensko-prostornu kompleksnost podatkovnih struktura i originalnih algoritama proisteklih iz DSA programa – alata za modeliranje dinamičkih sustava s pomoću stohastičkih konačnih automata prema teoriji ϵ-strojeva. Učinkovita grupna iteracija generira točke sustava i njihove binarne simbole, te ih pohranjuje u kružne spremnike realizirane s pomoću klasnih članskih poredaka. Riječima ekstrahiranim iz vremenskog niza hranimo dinamički kreirano stablo odabrane visine. U stablu potom nalazimo morfološki i stohastički jedinstvena podstabla ili morfove, uz pomoć algoritma koji uspoređuje (pod)stabla prema njihovim topološkim odrednicama i uvjetnim vjerojatnostima čvorova nasljednika. Teorijska analiza algoritama potkrijepljena je mjerenjem vremena njihovog izvođenja. Članak ilustrira kako implementacija znanstveno-programskog alata za modeliranje kao što je DSA, generira paletu specifičnih algoritamskih rješenja za koje možebitno postoji i šira uporaba.

Ključne riječi: *algoritmi, podatkovne strukture, vremensko-prostorna kompleksnost, dinamički sustavi, stohastički konačni automati*

1 Introduction

Modelling of dynamical systems by means of computation automata is introduced by the physicists John P. Crutchfield and his team of collaborators in the theory of *ϵ-machines* from the area of *computational mechanics* [1, 2]. Unlike the common way of solving the system differential (difference) equations numerically, the theory uses computing methods to analyse the time series emitted from a system, and the computational automata to present the system model. The introduced interdisciplinary approach brings in new and challenging problems to be analysed by the computational and information-theoretic tools, and opens a great arena for practical programming. This fusion of physics and computer science motivated our work, too, from which in this paper we present a variety of data structures and several original algorithms that were implemented in our *Dynamical Systems Automata* (DSA) program. Programming of a comprehensive modelling tool like this one requires a detailed knowledge of the theory behind it. A more in-depth background of the ϵ-machines theory modelling scheme, aimed for the readers from the computing, information-theoretic, and broader area, can be found in the author's paper [3].

The development of the DSA software started with the work presented in [4]. From the beginning the program was envisioned as more than just a theoretical tool with limited practical usability. With its graphical user interface (GUI), the DSA program enables interactive investigation of several implemented 1-dimensional dynamical systems, and should serve as a practical and user friendly scientific tool. It finds out the system's computational models up to the level of the stochastic finite automata, which present the crucial deterministic step of the theory (see the next section, or for details [5, 4]). Being based on the object oriented programming paradigm, the DSA program's foundations are prepared for easy upgrading, which includes installing new dynamical systems, adding new data presentations, and providing overall better functionality.

Aside from the many interesting features and capabilities of the DSA program from the programmer's and user's aspects, in this paper we take the role of a computer scientist who is interested in the implementation details of the program, and particularly in the design and efficiency of its core algorithms. This will be the main subject of our work. To physicists interested in the modelling of dynamical systems, such analysis can help to understand the underlying computing complexity needed to build the model, and to computer scientists it can offer an insight into several original algorithms that are needed in a comprehensive programming tool like this. Many of these ideas and solutions can find their application in general use, with little or no adaption. Furthermore, we have equipped the program with time measurement functionality for all of its key algorithms, so that, besides the modelling task, it can serve for the performance tests.

The article is organized as follows. Sec. 2 is devoted to an extra short exposé of the ϵ-machines modelling scheme, in order to provide an overview of the tasks that needs to be done by the main program modules. A brief introduction to the iterative systems is aimed to illustrate how the time series is obtained from a dynamical system. In Sec. 3 we describe the algorithms of the program module that iterates the system orbit points and provides their binary coding. Here the concept of batch iterations – based on the implementation of a circular buffer – is used to assure efficient generation of millions and billions of points if so needed. In Sec. 4 we deal with the dynamically created binary parse tree, which is fed with the words extracted

from the time series. The tree-feeding process is time consuming if one wants to achieve statistically relevant number of substrings. This is followed by an original nonrecursive algorithm for the (sub)tree comparison. The final stage of the modelling is finding the unique subtrees or morphs, which is presented in Sec. 5. The theoretical analysis is backed up by the concrete time measurements, and concluded in Sec. 6.

Because of the high volume of this applied algorithmic odyssey, in several places we shall be forced to skip more detailed approach in order to round up the subject. We hope this will be recognized not as a deficiency, but as a prospect for the future work. Also, we wish that a few original algorithms and programming solutions that follow will make the rest of the paper an interesting and useful reading.

2
Modelling of dynamical systems – from binary strings to the stochastic finite automata

To build computation automata models of dynamical systems, their orbit point coordinates should be encoded in symbolic presentation. The strings of symbols can then be analysed in a way analogous to the operation of computer language processors. Physically and epistemologically we can depict this as a *measuring instrument* that is installed by an observer and modeller of a dynamical system. The instrument measures the point coordinates and encodes them suitably, transforming the orbit into a *time series*. This is generally a string of symbols of indefinite length, in which a new symbol is added upon every discrete system iteration. From the time series we extract the strings of defined length, which shall be called words, and build our automata-based computational model. In the process we check the model size on different levels and try to make it minimal. On the other hand, we determine the quantity of information received from the measuring instrument and tune it in a way that the information received from the instrument is maximal. What we have described here briefly is known as the *measuring and modelling channel* [5, 3].

We shall elaborate shortly on each of the measuring and modelling stages, which are implemented in the three main modules of the DSA program. This will give us an insight in which data structures and algorithms should be used in each of the stages and the related modules.

2.1
Simulation of dynamical systems

The above described scheme depicts a general procedure in which we receive a time series from an *unknown* system. In order to test the model-building process and the algorithms it is based on, we shall start from some well known dynamical system. For this we provide numerical simulation by ***iteration*** of their orbit points. After an orbit point is determined, it can be "measured" and encoded, which simulates the tasks of the measuring instrument.[1)]

In the description of 1-dimensional dynamical systems we normally use the iterative equations of the form:

$$x_{n+1} = F(x_n), n \in N_0^+. \quad (1)$$

[1)] As we shall later, in order to speed everything up, organize all the tasks through the concept of *batch procedures*, so that a bulk of orbit points will be iterated first, and only after that, all of them will be measured by the binary instrument.

Here x_n is the system orbit point in the discrete time n, and x_{n+1} is the point in the next moment $n + 1$, obtained by the action of some mapping function F. A standard textbook example of nonlinear dynamical system with chaotic behaviour is the famous logistic function:

$$x_{n+1} = rx_n(1 - x_n), x_n \in [0,1]. \quad (2)$$

For it, the interesting range of parameter is $0 \le r \le 4$.

The function can be plotted in the x_{n+1} versus x_n graph, yielding the parabola as shown in Fig. 1. The graphical solution of the equation (2) with $r = 4{,}0$ is given for the first 13 points of the system orbit. The superimposed identity line $x_{n+1} = x_n$ serves to provide the iteration by mapping the function result $x_{n+1} = F(x_n)$ back to the abscissa domain, and to enable finding of the next point $x_{n+2} = F(x_{n+1})$. In the example shown, we start with the *seed point* $x_0 = 0{,}2$ and make 12 iterations for the total of 13 points. The last iterated point is x_{13} 0,99993842…. The system progresses further with a fully chaotic behaviour. The "graphical" solution presented in Fig. 1 is in fact drawn after the numerical calculations done by the DSA iteration module with the standard double floating precision.

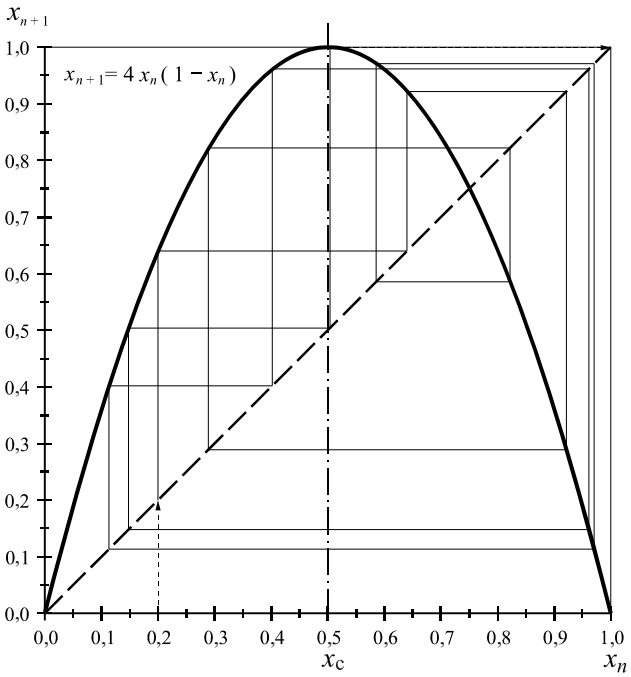

Figure 1 Graphical analysis of the logistic equation with $r = 4{,}0$. Starting from the seed $x_0 = 0{,}2$, the first 13 points are presented showing irregular behaviour. The points are "measured" by a course instrument and then encoded as the binary string: 0110111001011… (6 zeros follow before the next 1).

Here we departure from the classical numerical description of dynamical systems and from now on use the language of *symbolic dynamics* [6]. In order to discretize the state variable, the system's phase space is partitioned. To each of the partitions a symbol is attributed. The basic and simplest partitioning is binary, i.e. to the left and right subsets of a domain, measured from some *critical point* x_c defined by $F'(x_c) = 0$. To each subset we then attribute a symbol from a binary alphabet, like 0 and 1. In our example $x_c = 0{,}5$. Thus the seed point is encoded as 0, because $x_0 \in [0, 1/2)$, and the next point $x_1 = 0{,}64$ is encoded as 1. The 13 orbit points are then encoded as the binary string in the figure

caption. A more detailed formalism of the symbolic dynamics in the context of the theory of ϵ-machines can be found in [4].

2.2 The parse tree

From some final portion of the time series we extract words of constant length and analyse their interrelationship to get the insight in the system language. Because we yet have to learn the language, we build our fundamentally simple *language analyser* by feeding the words of binary symbols into the *grow and parse tree*. For the basic binary alphabet this is a ***binary tree*** with all branches of equal length. To simplify the terminology and to emphasize the tree similarity to language parsers, we shall call the tree shortly *parse tree*. Similarly, the process of the tree growing and of the (sub)word counting will be called *feeding* or *parsing*, according to the context.

The notion of *investigating the structure of the received data by the data itself and by no other premises* is recursive per se. The tree can be also very simply defined recursively [7, 8], meaning that it is recursive by its very nature. Thus the tree is indeed the right data structure for the first and crucial level of our model. And within the main parse tree we shall search for the smaller trees of fixed height to explore the "structure within structure", which will be explained in the next subsection.

As was hinted above, in difference to predefined parse trees associated with predefined languages, the ϵ-machine parse tree is initially empty. Its purpose is, at least at first, to learn the system language by reading the words extracted from the time series that was emitted by the system. So, we feed the tree with words w_D of length D extracted from the time series. For every new word and its substrings not previously found in the time series, the tree will grow new nodes, while for the repeating words (and its substrings) the counts in the corresponding nodes will be incremented.

More precisely, every prefix w_l of the word w_D with length $l \leq D$ defines a node in the parse tree. Including the empty prefix e which defines the root node, and the word itself which defines a leaf, there are $D + 1$ prefixes in every such word. So when we parse the word, its prefixes relate to nodes. If a node did not exist before, it is created and its counter is set to 1. If a node already existed, its counter is incremented. The sum of counts on all tree levels is the same (preserved). Obviously, which nodes will be created and how much they will be fed, is dependent on the structure of the time series.

Feeding of the grow and parse tree of height (depth) $D = 5$ from a source that can be described as "no consecutive zeros" is shown in Fig. 2. The source is *regular* in the sense of the theory of computation, since it can be described by the regular expression: (0 + e)(1 + 10)*. This simple system of regular behaviour is more suitable to guide us through the modelling procedure than the logistic mapping. There are 16 substrings or words of length 5 that could be fed from the above series of 20 symbols. Each of the $w_{D=5}$ words defines a unique path and a unique leaf among the 2^5 possible leaves at the depth 5. Here every word defines 6 nodes, including the root node at zeroth level. E.g. for the first word in our example the nodes encountered are: e, 0, 01, 011, 0110, 01101. The non-existing nodes are designated by thin circles. There are no paths leading to them from the root.

Normally, a leaf of a binary tree is a terminal node with no descendants. Since all our input words w_D are of the same length D, our parse tree is a special binary tree with all its leaves at the same level $l = D$. We call it ***binary ϵ-tree***, or simply ***ϵ-tree***, and within it we define ***ϵ-subtrees*** of height $L \leq D$. For both of these a leaf is every node at level D. Many of the conclusions made for our specialized (sub)trees are valid also for the general trees by releasing the leaf criterion to the standard one (for details on ϵ-(sub)trees see [11]).

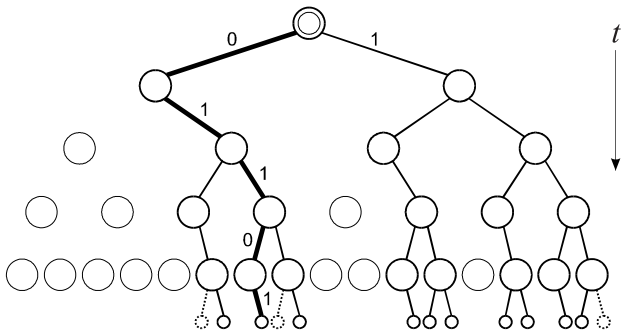

Figure 2 Feeding of the binary ϵ-tree of depth $D = 5$ with words from the time series s = 01101011011110110101… generated by the system "no consecutive zeros". The first word 01101 creates the nodes on the bolded path. The last word defines the sixth leaf from the left. The nodes and paths that are allowed by the rule, but do not (yet) appear in s, are shown in dotted lines.

By providing abundant feeds of our parse tree with long enough portions of the time series, we assure to record all the word prefixes w_l that appear in the language of the observed dynamical system. Furthermore, from the counts that record the prefix appearances in the tree nodes we can calculate their full statistics. This includes the a-posteriori probabilities for the next symbol or a string conditioned to some previous prefix (node).

2.3 The structure within structure – morphs and the stochastic finite automata

The *ϵ-machine* must be an unbiased model based solely on the data received from the system. Within the structure of our parse tree we search for subtrees of height L, $1 \leq L \leq D$, which are unique in either their "morphology", or in their stochastic properties. We shall call such unique subtree a ***morph***.

Precisely stated, a morph is a class of δ-equivalence for all subtrees having the same topology of nodes, and having all the node probabilities conditioned to the subtree root within some parameter δ, $0 \leq \delta \leq 1$. Since every subtree can be identified by the string $\bar{s}$ that describes its root node starting from the parse tree root, two subtrees defined by $\bar{s}$ and $\bar{s}'$ are δ-equal if the probabilities for the same downward paths or branches $s_t^{\rightarrow}$ coming from $\bar{s}$ and $\bar{s}'$ are within δ:

$$\bar{s} \sim \bar{s}' \Leftrightarrow |\Pr(s_t^{\rightarrow} \mid \bar{s}) - \Pr(s_t^{\rightarrow} \mid \bar{s}')| \leq \delta . \qquad (3)$$

The downward branches $s_t^{\rightarrow}$ are all possible paths leading to the roots of the possible subtrees of depth L, which themselves define further subtrees, etc. The subtree $\bar{s}'$ which is different from any previously found morph $\bar{s}$ according to the condition (3), defines a new class of δ-equivalence and becomes its representative. A morph represents past-independent δ-equal prospects for the future. In other words, it is a causal state with equal conditional probabilities for its substrings. As such, morphs

are a natural choice for the description of the system states [5].

δ-"equivalence" $\bar{s} \sim \bar{s}'$ is not a true equivalence relation, because generally it is not transitive[2]. This can influence the completeness of certain algorithms for finding them (confer 5.2). By allowing certain δ-tolerance in equation (3), i.e. by putting $\delta > 0$, we comply to the experimental and computational reality that only finite precision is achievable. When modelling, approximations are unavoidable. Without them the models would diverge in the number of states and in complexity.

The procedure of finding morphs starts with the root node and the subtree it defines. As the first one, it is always unique and hence a morph. Then the root's left and right child subtrees are compared to the root morph. If they are different (and unique), the process continues with their subtrees while there are new morphs and while we can go further down the parse tree.

In the example from Fig. 2 we search for the subtrees of height $L = 2$. There we can spot 3 different subtrees, as shown in Fig. 3. *A* is the root morph, which corresponds to the initial state when no previous symbols are emitted from the system and no symbols are received by the modeller. Because of that, the probabilities for emitting 0 or 1 are based on the fact that on average our system emits twice as much 1s as 0s. On the morph diagram this is denoted as $0|\frac{1}{3}$ and $1|\frac{2}{3}$. *B* is the left subtree of *A* that occurs after a 0 is received. The system rule says that a 0 must be followed by a 1, and hence the sure transition to the right node. Because of the morphological difference to the root morph this is the second morph. After 1, the system can emit either a 0 or a 1 with equal probabilities. Subtree *C* is morphologically equal to *A* since it has exactly the same transitions and consequently the same nodes. But, according to the system rule, because it occurs after a 1 was received, it has equal probabilities for emitting a 0 or a 1. In other words, it has equal probabilities for going to its left or right submorph.

If we look to what states the system goes after emitting a 0 or a 1, we can easily conclude the transitions from the system states. From *A* on emitting a 0 we get to *B*, and on emitting a 1 we get to *C*. From *B* the only, sure, transition is to *C*, and from *C* we can get to either *B* or *C* with equal probabilities. We note that the state *A* is transient. After leaving it – and that happens immediately after receiving the first symbol when the phase is caught – the system oscillates between the *B* and *C* states only.

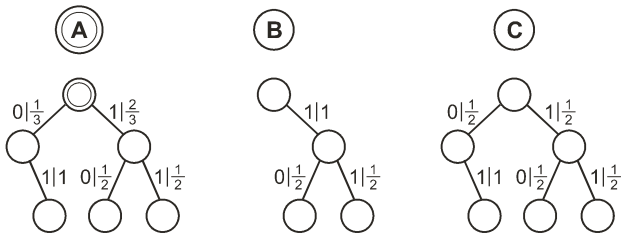

Figure 3 Subtrees of height (depth) $L = 2$ found as unique morphs (system states) in the parse tree from Fig. 2. *A* is the root morph, *B* and *C* are its left and right subtrees. Having the same structure of nodes, *A* and *C* are morphologically equivalent, but with different transition probabilities for the generation of 0s and 1s.

[2] $\bar{s}_1 \sim \bar{s}_2$ & $\bar{s}_2 \sim \bar{s}_3$ does not necessarily imply that $\bar{s}_1 \sim \bar{s}_3$, because of the possible buildup of the δ-tolerance.

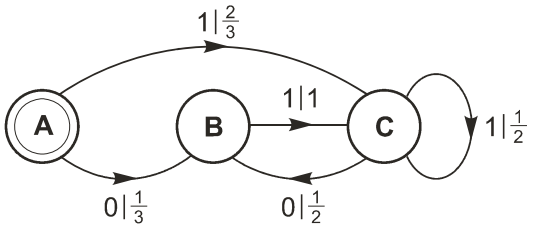

Figure 4 Stochastic Finite Automata (SFA) for the rule-defined system "no consecutive zeros". The states are circles. Here the double circle denotes the start state (in FA it usually denotes the final or accepting state!). Each transition is labelled with the symbol that initiates it, and with the transition probability.

The above analysis leads us to the fully defined ***Stochastic Finite Automaton*** (SFA) for the "no consecutive zeros" system, presented by the cyclic directed labelled graph (LDG) in Fig. 4. The SFA accurately depicts the system behaviour, and does that with the minimal number of states. The SFA are analogous to the finite automata (FA) with addition of the transition probabilities and the lack of the final state. A definition of the SFAs and their comparison to the FAs is given in [4].

3 Simulation of the System and Measuring Channel: From Orbit Points to the Time Series

The algorithms from the DSA program that will be presented here are implemented in C++ language[3] by strictly obeying the object-oriented paradigm. They can be pictured as parts of the three program modules which follow the modelling stages depicted in the three subsections of the previous section 2. The modeller picks up one of the implemented systems, initiates the generation of the time series, and gets the model in the form of an SFA with given parameters. Thus, as initial step mentioned in 2.1, the program is expected to provide the iteration of dynamical systems and the generation of the time series. This serves the purpose of testing the program in the developing phase, and the research of the systems later on. Also we need to simulate the binary instrument measurements to make our measuring and modelling channel complete. So we start with description of dynamical system iteration.

3.1 Iteration of Dynamical Systems

The basic functionality for dynamical system iteration is encapsulated in the abstract base class `CDynSys1D` (*Dynamical System, 1-Dimensional*). The class is realized through its two-level subclasses as is illustrated in Fig. 5. The 2nd level abstract subclasses with extensions `_0Par` and `_1Par` unify the functionality for dynamical systems with zero and one parameter, while `_RlRnd` does that for the rule-defined systems with randomly generated numbers. Their subclasses finally implement the concrete 1-dimensional dynamical systems with their specific iteration functions $F(x_n)$ (confer eq. (1)).

In order to achieve the statistical relevance of the reconstructed models, abundant feeds of the parse tree are required. This in turn requires a very large number of orbit points to be iterated. To have N_w words of length D,

[3] Microsoft Visual Studio© developing tool is used, for the Wintel platform GUI.

```
CDynSys1D:
  CDynSys1D_RlRnd:
     CDS1DRR_BernoulliSer
     CDS1DRR_0IfPrevIs1
     CDS1DRR_NoConsec0s
     CDS1DRR_EvenNumOfConsec1s
       ...   ...    ...
  CDynSys1D_0Par:
     CDS1D0P_DoublingFunc
       ...   ...    ...
  CDynSys1D_1Par:
     CDS1D1P_AbsoluteValue
     CDS1D1P_LogisticMap
     CDS1D1P_QuadraticMap
     ...     ...    ...
```

Figure 5 The hierarchy of the Dynamical System Classes. The abstract, base, classes are underlined. New concrete classes can be added by providing the overrides for the iteration functions only. All other functionality is implemented in the base classes.

$n_p \geq N_w + D - 1$ points must be generated. To assure enough counts in the leaves of tall and diversified trees, 10^8 to 10^9, or even more, words may be needed.

In our modelling, once the words are fed into the tree, the points are not needed any more. Hence the idea to implement a circular buffer which will enable the iteration of arbitrarily large number of points (limited only by the integer data type that counts them – e.g. in the parse tree root node), while in the same time reducing the algorithm space complexity.

To minimize the execution time, every detail is adjusted for the greatest speed of the algorithm. Thus, the main array for storing the orbit points is organized as a class-bound (non-dynamic) member, with fixed number $N_{CB,max}$ of maximal elements of type `double`. The user can change the actual size of the buffer N_{CB} to a number smaller than the provided maximum (e.g. for the testing purposes). Of course, because the array is not dynamic, a new compilation of the program is required to change the maximal buffer size, but this is a small sacrifice for having the fastest possible access to the array elements. The present choice is $N_{CB,max} = 2M + 32$ elements.

Naturally connected to the circular buffer data structure is the concept of *batch iteration*, in which the whole buffer is filled with the orbit points regardless of how many of them are momentarily requested by the tree feed process (see 4.1). The rationale for this is obvious: what may seem as a redundant approach when a small number of points is needed, will pay its way when millions and billions of iterations are needed. For such demanding calculations we need to reduce the number of intermittent function calls and other slowing mechanisms to a minimum. Modern computers have the higher levels cache memories that can contain considerable amounts of data and programming code, which can further contribute to the efficiency of the batch iteration concept.

Before getting into performance details we shall inspect a few basic organizational aspects of the batch iteration. The used functions are presented in Listing 1.[4)] The function `InitIteration` sets the pointer to the array of orbit points of the type `double`, resets the number of batch iterations done to 0, and the number of iterated points to 1 (to account for the seed point). This and the `BatchIteration` function make the `FirstBatchIteration`, which simply fills the point array to the full. Every next batch iteration must be done with `RepeatBatchIteration`, which uses the function `PrepBatchIterRepeat` to move the last `uStrl - 1` points to the beginning of the circular buffer. This is necessary to maintain consistency of the extracted words of length $D =$ `uStrl`. The further iterations that are done by the `BatchIteration` function will start from the point with index `uStrl - 2` as the initial value (complying to the zero-based C/C++ indexing style), and calculate the points with indices from `uStrl - 1` up till the end of the circular buffer array.

Listing 1 The base class `CDynSys1D` functions used in the dynamical system batch iterations

```
// DECLARATIONS
// Iteration initialization:
InitIteration(double x0);
// Prepare the batch iteration repeat:
PrepBatchIterRepeat(UINT uStrl);
// Base class batch iteration for common
// functionality:
BatchIteration(UINT uStrl);
// Declarations and implementations:
FirstBatchIteration()
  { InitIteration(); BatchIteration(2);}
RepeatBatchIteration(UINT uStrl)
  { PrepBatchIterRepeat(uStrl);
    BatchIteration(uStrl);}
// Comprehensive functions (see Listing A1):
DoABatchIterInclBI(bool bFrmLstPnt,UINT uStrl);
    // Performs one batch iteration that calcula-
   // tes all the points in the circular buffer.
DoBatchItersInclBI(bool bFrmLstPnt,
                UINT uStrl, UINT uNBIt);
   // Performs uNBIt iterations, refilling the
  // circular buffer each time.
```

Listing 2 The concrete batch iteration for the logistic mapping implemented by overloading the base class `BatchIteration` function

```
void CDS1D1P_LogisticMap::
   BatchIteration(double dPar, UINT uStrl)
{
   _ASSERT(uStrl >= 1);
   if(uStrl == 1) uStrl++;
   double* p  = Getpdx() + uStrl - 1;
   double* p1 = Getpdx() + GetuNArrPts();
   double x;

   for (; p < p1; p++)
   {
       x = *(p - 1);
      *p = dPar*x*(1.0 - x);
   }
   CDynSys1D::BatchIteration(uStrl);
}
```

To illustrate the functionality of the above base class functions, we give the batch iteration functions for the two exemplary systems from 2.1. The first one is 1-parameter logistic mapping presented in Listing 2. The algorithms are given directly in the C++ code, counting on the good language definement, precision, and readability by most of the readers. The first argument `dPar` is the single parameter for this dynamical system class, and the second is the string length `uStrl`. The accessor function `Getpdx` provides the pointer to the base class orbit point array and `GetuNArrPts` gets the number of points in the circular buffer. Let us note that after all the preparations, the batch iteration boils down to a `for` loop with the simplest and fastest possible realization of the iteration function. The access to the array elements is always done by dereferenced pointers, and all unnecessary operations and function calls are strictly

[4)] The term Listing will be used for the literal or slightly modified excerpts or integral parts of a program code.

avoided. After the points are calculated in the for loop, the base class function is called which does the common jobs of keeping track of the total number of iterated points, counting the finished batch iterations, and providing the debug asserts.

The other mentioned override of the `BatchIteration` function in the `CDS1DRR_NoConsec0s` class starts in the same way, as shown in the Listing A2 of Appendix A. For all subclasses of `CDynSys1D_RlRnd` class the orbit points are within the unity interval $x_n \in [0, 1]$, $n = 0, 1, 2, \ldots$, and are obtained by adequately normalized random numbers. Because of the calls of the random generator function and the need to obey the system rule, the resulting algorithm is more complex than the one for the logistic map. Not shown in the Listing A2 (because of the length), is that the actual versions of these functions enable also probability-biasing by setting parameter $b \in [0, 1]$ to a value different from 1/2 for the fair coin. b sets the probability that an orbit point, which is randomly picked from the unity interval, is in its first half:

$$b = \Pr(x_n \in [0,1/2)), n = 0,1,2,\ldots. \tag{4}$$

The definition is motivated by a simple idea: if the obtained point x_n is to be differentiated later by a binary instrument adjusted to the middle of the unity interval (see below), b is the probability for 0s, and $1 - b$ is the probability for 1s. The interesting analysis of the probability-bias implementation will be omitted from here.

As a quick discourse, let us note that despite the rule-defined iterations are algorithmically much more complex than the quadratic functions like the presented logistic map, regarding the richness and complexity of the phenomena produced the situation is just the opposite. The iteration of the logistic equation leads to incomparably wider range of orbits: from the simple fixed point, to periodic and very complex, and finally to fully chaotic and almost random behaviour [6].

3.2
Binary measurement

The functionality of the measuring instrument is provided by the `CBinInstrument` class. According to the name, its only purpose is to make the binary discrimination of the system orbit points according to the binary measurement and encoding function:

$$f_{\mathrm{BI}}(x_n) = \begin{cases} 0, x_n \in [0, x_{\mathrm{BI}}), \\ 1, x_n \in [x_{\mathrm{BI}}, 1]. \end{cases} \tag{5}$$

$x_{\mathrm{BI}} \in [0, 1]$ is the adjustment point of the binary instrument. In most cases it must be kept equal to the critical point of the system: $x_{\mathrm{BI}} = x_{\mathrm{c}}$, in order to fulfill the requirement that the partition of the phase space is *generating* [6, 4 ch. 2]. However, for the rule-defined systems of the class `CDynSys1D_RlRnd` this can be freely altered. One purpose of tempering the instrument adjustment to a value different from the standard one (in this case $x_{\mathrm{c}} = 0{,}5$) could be to provide compensation for an "unfair" bias with parameter $b \neq 0{,}5$, as was discussed in the previous subsection.

The binary measurement and encoding (5) is realized by the simple function `BinMeasurement` in the batch manner, i.e. for all the orbit points of the class's circular buffer at once. An array of bytes (Visual C++ specific 8-bit integer type BYTE) is filled with binary values 0 or 1. With this, the dynamical system's time series is determined and prepared for further analysis. Its continuity and consistency for the extraction of the words of length D is assured by the `RepeatBatchIteration` function – here the points are only encoded. The function is placed right after the batch iteration within the aggregate functions `DoABatchIterInclBI`, which runs one, and `DoBatchItersInclBI`, which runs specified number of batch iterations. These functions wrap up all the functionality needed for the system batch iteration(s). The first of them is shown in Listing A1, illustrating also the implementation of the time measurements.

3.3
Complexity analysis of iteration algorithms

We shall analyze the space-time complexity of the iteration and other algorithms by introducing functions: $f_{\mathrm{TIME}}(n) = T(n)$, and $f_{\mathrm{SPACE}}(n) = S(n)$. Both crucial algorithms from this program module: for the calculation of orbit points and for their binary encoding, are simple algorithmic *iterations* (IT) with time complexity $T_{\mathrm{IT}}(n_{\mathrm{p}})$ being linear in the number of points n_{p}:

$$T_{\mathrm{IT}}(n) = t_{\mathrm{IT1}} n_{\mathrm{p}} + t_{\mathrm{r}D}(n_{\mathrm{p}} / N_{\mathrm{CB}}) + t_{\mathrm{IT0}} = O(n_{\mathrm{p}}). \tag{6a}$$

The constant t_{IT1} in front of the linear term in n_{p} comprises the time needed for the calculation of one iteration point, and t_{IT0} is the initial preparatory time needed for the first batch iteration. $n_{\mathrm{p}}/N_{\mathrm{CB}}$ is integer division with result k for $kN_{\mathrm{CB}} \le n_{\mathrm{p}} < (k+1)N_{\mathrm{CB}}$, $k = 0, 1, 2, \ldots$.

A total of $k + 1$ new batch iterations are done to assure that the N_{CB} buffer points are filled (confer 3.1). $t_{\mathrm{r}D}$ refers to the time needed to prepare every next batch iteration after the first one for the words of length D, i.e. to remove $D - 1$ points from the end to the beginning of the buffer. It depends on the word length as $t_{\mathrm{r}D} = t_{\mathrm{r}}(D - 1)$, where t_{r} is the time constant for removal of one point. We can expect that $t_r < t_{\mathrm{IT1}}$. When $n_{\mathrm{p}}/N_{\mathrm{CB}} >> 1$, the integer division in (6a) can be approximated with normal fraction in order to rearrange the formula as:

$$T_{\mathrm{IT}}(n_{\mathrm{p}}) \approx \left(t_{\mathrm{IT1}} + \frac{t_{\mathrm{r}}(D-1)}{N_{\mathrm{CB}}} \right) n_{\mathrm{p}} + t_{\mathrm{IT0}} = O(n_{\mathrm{p}}). \tag{6b}$$

As stated above, t_{IT0} is the time needed to call a few concrete class functions and their base class versions that are performed just once before the group of batch iterations. This can be ignored for $n_{\mathrm{p}} >> 1$, and especially for $n_{\mathrm{p}} >> N_{\mathrm{CB}} >> 1$. The second term in the parenthesis can be disregarded for $D >> N_{\mathrm{CB}}$, which is the condition for an efficient circular buffer. E.g. for a buffer with $n_{\mathrm{p}} \approx 2\mathrm{M}$ points and the word length $D = 16$, after performing 2M iterations we make only 15 removals of the points from the end to the start of the array (see Tab. 1 and the discussion preceding it).

The algorithmic complexity is bounded from below by the very nature of the problem, and since it is linear in n_{p} it is satisfactory. Our programming efforts described in the previous subsections were aimed to assure the full functionality of iterations for many different types of 1D dynamical systems by as little repetition of the code as possible, while in the same time keeping the time

coefficients t as low as possible. The improvements in the constant factors do not show in the big O-notation, since it shows only the order of magnitude of the time dependency. The little o-notation $f(n) = o(g(n))$ checks if $\lim_{n\to\infty}(f(n)/g(n)) = 1$, which for the polynomial functions means that their highest order term factors are the same [9]. In our case it is the t_{IT1} that is crucial for the large number of iterations, so we can write $T_{IT}(n_p) = o(t_{IT1} n_p)$.

Regarding the space complexity, by the use of the fixed size circular buffer the lowest possible order of space magnitude is accomplished. If s_p is the memory space needed for one orbit point, and s_{vr} is the size of a typical variable, the iteration space is:

$$S_{IT}(n_p) = s_p N_{CB,max} + s_{vr} N_{Loc.Var.} = O(1). \quad (7)$$

The number of local and auxiliary variables $N_{Loc.Var.}$ can be ignored. The requirement $N_{CB} >> D$ keeps the size of the circular buffer significant, but independent of the number of points. With the above choice of approx. 2M points, the space of 16 MB is spent, which can be considered as quite small for the modern computers.

Execution time measurements for the batch iterations with different sizes of the circular buffer are provided in Tab. 1. It is easily spotted that the small buffers are inefficient. The value of 30 is a drastic example. E.g. for the logistic map, by enlarging the buffer size from 40 to 400 elements the speed increased 5,5 times. For the more demanding iteration function of the no consecutive 0s system, the time proportion of the buffer organizing functions is lesser, so that improvement is 2,7 times. We can conclude a rule of a thumb that by making the buffer 10 times as big as the word length, the times improve significantly, and by making it 100 times as big, we approach the size of the very effective buffer. Further buffer enlargements result in no significant time improvements (compare also Tab. 2 in subsection 4.1).

Table 1 Execution times for the batch iterations of the orbit points. The word length was $D = 20$. The circular buffer size varied from 30 to 10 000 points. The minimum number of batch iterations needed to provide the total of at least 10^6 orbit points was done (shown in second column). The third column presents the number of actually iterated points. The left of the two time columns is the execution time in seconds, and the right one is the average time per one point in microseconds.[5)]

$D = 20$, $N_{pts} \approx 10^6$			*Logistic Map*		*No Consec. 0s*	
Cir. Bff. Size	Batch Iters.	Num. of Iter. Pts.	Δt /s	$\frac{\Delta t}{N_{pts}}$ /μs	Δt /s	$\frac{\Delta t}{N_{pts}}$ /μs
30	90910	1000028	285,0	0,2850	337,0	0,3370
40	47620	1000038	152,0	0,1520	211,0	0,2110
50	32259	1000047	111,0	0,1110	165,0	0,1650
80	16394	1000052	65,4	0,0654	118,0	0,1170
100	12346	1000044	54,3	0,0543	106,0	0,1060
200	5525	1000043	35,2	0,0352	85,5	0,0855
400	2625	1000143	27,4	0,0274	77,4	0,0774
1000	1020	1000638	23,1	0,0231	73,2	0,0731
2000	505	1000423	21,3	0,0213	71,6	0,0716
10000	101	1008099	20,5	0,0203	70,6	0,0700

[5)] The testing is performed on Wintel platform: Pentium 4 on 2,6 GHz, 1 GB RAM, with Windows XP operating system.

The complexities $T_{BI}(n_p)$ and $S_{BI}(n_p)$ for the binary measurements are fully analogous to the values derived in (6) and (7), but with lower time and space constants. The total cost of the system simulation and the preparation of the times series is the sum of the complexities for the iteration and binary measurement:

$$T_{IT+BI}(n_p) = T_{IT}(n_p) + T_{BI}(n_p) = O(n_p), \quad (8a)$$

$$S_{IT+BI}(n_p) = S_{IT}(n_p) + S_{BI}(n_p) = O(1). \quad (8b)$$

The run time measurements for the iteration and binary encoding as functions of the number N_w of words are shown in Tab. 2 below. Since $N_w \approx n_p$ (see also the next section), the results of the constant time per word confirm (8a), because $T_{IT+BI}(n_p) \approx T_{IT+BI}(N_w) = O(N_w)$.

4 Tree Classes and the Parse Tree Feeding

The tree data structures are realized with three classes whose hierarchy is shown in Fig. 6. The base class is abstract `CABinTree` class. The name is derived from *Automata Binary Tree* to distinguish our tree from the general binary tree (Sec. 2.2, 2.3 [11]). The base class contains all the common functionality for the two concrete subclasses. `CMainABinTree` implements the grow and parse tree, which is in the programming context called the *main tree*. The class `CSubABinTree` realizes the general concept of a subtree of some other tree. A tree can be its own subtree. Subtrees are primarily needed in the process of morph extraction, but serve also for other purposes.

```
CABinTree:
   CSubABinTree
   CMainABinTree
```

Figure 6 The hierarchy of the Binary Tree Classes

The base class `CABinTree` uses the fundamental, recursively defined class `CBinTreeNode`, which implements a tree node as the basic constituent of a tree (confer Listing 3 below). A tree node can be considered as a minimal and trivial subtree that has height 0. For the nodes we have chosen dynamical allocation of memory. The rationale for that is simple: the concrete trees of height D may vary extremely in their size, from the minimal D nodes for the fixed points 00 … 0 and 1 … 1, to the maximum of $2^{D+1} - 1$ nodes for the *full* or *complete* tree that has all possible nodes. Thus, in general case we must be prepared for the growth that is exponential in D. The penalty for the dynamical allocation of nodes will be bigger execution time constants in the tree feeding process, as will be seen in 4.1.

The main piece of information stored in the tree node is the counter of the word prefixes, which are, as shown in 2.2, equivalent to the tree nodes. So they will be shortly called node counts. Furthermore, to organize a binary tree structure, the minimal information needed is the position of the two descendant nodes. Since the nodes are dynamically created, the most natural realization of the tree structure is to provide pointers to the node's left and right children. To provide greater flexibility of several algorithms depending on this fundamental class, two more member variables are added. The first is the pointer to the parent node. It makes the implementation of the nonrecursive traversals, as well as a few other functions (e.g. for the moving among the tree nodes), much simpler. The second is the node level that

simplifies and speeds up some functions needed in efficient representation of the parse tree. The member variables of the class `CBinTreeNode` are shown in Listing 3.

In the implementation of the tree node as this, a pointer to nonexisting child is assumed to be `NULL`. For the general binary trees, if both pointers to children are `NULL` the node is a leaf. For our Automata Binary Trees or ϵ-trees, the leaves are all nodes at the level equal to the tree height (depth) D, as is already commented in 2.2. So, when dealing with ϵ-subtrees, we can inspect the node absolute level, deduct from it the absolute level of the subtree root, and get the relative node level. If this is equal to the subtree height L, the node is a leaf regardless of the values of its child pointers.

Prior to moving to the main parse tree and its feeding, let us briefly mention the tree traversal functions implemented in the base `CABinTree` class. In the initial concept of the DSA program these functions were envisioned to be nonrecursive. The reason for that was to prevent possible stack overflows due to the nested recursive calls in complex algorithms. The nonrecursive traversal functions have more complex algorithms, but are easy to use. They can be explicitly written for every type of traversal and, furthermore, they can return the pointer to the next node that has to be visited. The latter functionality cannot be straightforwardly realized for the recursive traversals. The traversal order of recursive solutions is embodied in the recursive function itself, as the relative position of the task-performing statements to the recursive calls [9, 10, 11].

Listing 3 Member variables of the `CBinTreeNode` class

```
class CBinTreeNode
{
private:
  UINT m_uCount;   // Node count.
  UINT m_uAbsLev; // Absolute node level.
          // Root nodes have m_uAbsLev:
         //  = 0 for the trees of mainTree type;
        //  >= 0 for the trees of subTree type.
  CBinTreeNode* m_pParent;   // Ptr. to parent.
  CBinTreeNode* m_pLChild;  // Ptr. to L. child.
  CBinTreeNode* m_pRChild; //  Ptr. to R. child.
   // ...        ...
  // ...         ...
}
```

For each of the three main traversal types a nonrecursive algorithm was designed, resulting in `TravrsPreOrd`, `TravrsInOrd`, and `TravrsPostOrd` nonrecursive functions. They all utilize the self referenced pointers in our tree nodes, including the pointer to the parent node. When using these functions, each of them must be preceded by its own initialization function: `InitPreOrd`, `InitInOrd`, and `InitPostOrd`, respectively. The practical ability of the nonrecursive traversal functions to return the pointer to the next node was then used in the tree comparison algorithm implemented in the `CSubABinTree` subclass, and in some other tasks. One of them is the manual step-by-step traversal through the parse tree. The nonrecursive algorithms were then compared against their recursive versions for a few tasks organized within the testing functions. This interesting subject of general applicability is discussed in a separate paper [11].

4.1 The tree feeding algorithm

Having prepared the `CMainABinTree` class to accept words from a time series, we are ready to "grow and feed" the tree. The nodes are dynamically created and their counts are incremented by Tree Feed (TF) algorithm, as outlined in 2.2 and in the description of the node class. Feeding of the tree with one word is implemented by the function `FeedWithABinString` shown in the Listing 4. The only function argument is the pointer to the byte containing the first symbol in the word. Other needed parameters are mostly the member variables of the base class `CABinTree`, and a few remaining are from the node class `CBinTreeNode`. As can be read from the code, they are all accessed by the accessor functions to straiten the encapsulation of the classes. All trivial and simple functions, and particularly the accessors, are declared `inline`.[6] Thus, regarding the access time to the variables, it will be the same as they are directly addressed from the place of the function call.

The algorithm follows the idea from Fig. 2. The procedure is slightly different for the leaves, so we inspect $D - 1$ symbols first. If the first symbol is a 0 (1), the left (right) child of the current node is checked. If it does not exist, it is created with its level recorded and the counter set to 1. In parallel we build the tree statistics stored in the base class member variable of the `CTreeStatistics` type. It counts the left and right nodes. The last symbol corresponding to a leaf is treated separately, to avoid constant level checking. The procedure is mostly the same, with the only difference that in the tree statistics the number of left and right leaves is updated.

When the modeller requests that the parse tree is fed with n_w words, the function `FeedWithABinString` is called n_w times within an iteration loop. The true solution gets complicated because for large n_w we also have to update the iteration and binary measurement circular buffers by doing necessary new batch iterations. This job is done by separate organizing functions within the `CMainABinTree` class. Despite the fact that batch iteration may produce more points than needed (on average a half of the buffer size), those functions assure that the `FeedWithABinString` function will be executed exactly n_w times, feeding n_w words into the main tree.

The Tree Feed algorithm time complexity for one word $T_{TF,w}$, expressed by complexity $T_{TF,s}$ for one symbol that reduces to the time constant t_s, is:

$$T_{TF,w}(D) = DT_{TF,s} = t_s D = O(D). \tag{9}$$

A comprehensive algorithm that needs time t_w to extract one word, has time complexity T_{TF} for n_w words:

$$T_{TF}(n_w, D) = t_w n_w + t_s D n_w = (t_w + t_s D) n_w \tag{10a}$$

$$= O(n_w) + O(D n_w) = O(n_w) = O(n_s). \tag{10b}$$

All zero-order terms time constants were ignored here, and $n_s = n_w D$ is the total number of symbols.

Although quite obvious, the derivation was made explicit to prove that the Tree Feed algorithm was linear in the total number of words and symbols that are fed into the tree.

[6] For pure accessors this was always possible. For the calls to subclass overrides, this was sometimes prevented by the linker.

Listing 4 Feeding the parse tree with a word containing D binary symbols. The main tree height = D = `GetuHeight()`

```
UINT CMainABinTree::FeedWithABinString(BYTE* p)
{
  CBinTreeNode* pNewNode;
  CBinTreeNode* pChild;
  CBinTreeNode* pNdC = GetpRoot();
  UINT j  = 0, j1 = GetuHeight() - 1;
   // Start from the root and construct new or
  // increment the old nodes below the root:
  for ( ; j < j1; j++)
  {
    if (*p == 0x00) //  The symbol is 0.
    {
      pChild = pNdC->GetpLChild();
      if ( pChild == NULL) // Make a new L. Child
      {
         pNewNode = new CBinTreeNode(j + 1);
         m_TreeStat.IncrLNodes(); // Optional!
         pNewNode->SetpParent(pNdC);
         pNdC->SetpLChild(pNewNode);
      }
      else  // Increment old Left Child:
         pChild->IncrnCount();
      pNdC = pNdC->GetpLChild(); // Move to L. Ch.
    }
    else // *p == 0x01, the symbol is 1.
    {
      pChild = pNdC->GetpRChild();
      if (pChild == NULL) // Make a new R. Child
      {
         pNewNode = new CBinTreeNode(j + 1);
         m_TreeStat.IncrRNodes(); // Optional!
         pNewNode->SetpParent(pNdC);
         pNdC->SetpRChild(pNewNode);
      }
      else  // Increment old Right Child:
         pChild->IncrnCount();
      pNdC = pNdC->GetpRChild(); // Move to R. Ch.
    }
    p++;
  } // end for
  // Leaf level: construct new or increment old
  // leaves, following the above procedure for the
  // nodes, additionally update the leaf counts.
        . . .   . . .   . . .
  // A successful feed: increment the root count:
  GetpRoot->IncrCount();
  return 1; // One string was fed.
}
```

It is the additional demand of the model precision that will make the feed time critical in the modelling process. Namely, if we want to provide statistical relevance of the node-to-node transition probabilities, and achieve better accuracy of the model by using lower δ parameter (eq. 3), a sufficient number of counts must be recorded in the nodes of the subtrees that will be compared. The number N_{Nds} of nodes in a binary ϵ-tree of heigth D is in the range: $D \le N_{\mathrm{Nds}} \le 2^{D+1} - 1$, of which 1 to 2^l are on the level l, l=0, 1 … D. If we want to achieve the δ-accuracy, we must require that some $k_{FF}\delta^{-1}$ counts are in every leaf, where $k_{\mathrm{FF}} \ge 1$ is a *feed factor*. E.g. we can set it to 10 to assure sufficient number of counts and conditional probability accuracy to be better than δ for about an order of magnitude. Obviously, the feed factor should be set to even higher values, because we are just guessing how the counts will be distributed in the nodes. In the best case with only D nodes, we shall need the feed with $k_{\mathrm{FF}}\delta^{-1}$ words, or $k_{\mathrm{FF}}\delta^{-1}D$ symbols. In the worst case with 2^l nodes on the level l, by requiring on average $k_{\mathrm{FF}}\delta^{-1}$ counts in every node, the conservation of counts gives:

$$n_{\mathrm{w}} \ge k_{\mathrm{FF}} \times \delta^{-1} 2^l. \tag{11}$$

For the above criterion to hold also on the leaf level $l = D$, the request is to feed $n_{\mathrm{w}} \ge k_{\mathrm{FF}}\delta^{-1}2^D$ words, and that is exponential in D. If we combine this with eqs. (10), we can summarize the best, worst and average time complexities of the statistically proper or *abundant feed*:

$$k_{\mathrm{FF}}\delta^{-1}t_{\mathrm{s}}D \le T'_{\mathrm{TF,abn}}(D) \le k_{\mathrm{FF}}\delta^{-1}t_{\mathrm{s}}D2^D, \tag{12a}$$

$$O(D) \le T'_{\mathrm{TF,abn}}(D) \le O(D2^D), \tag{12b}$$

$$\left\langle T'_{\mathrm{TF,abn}}(D)\right\rangle_{\mathrm{ave}} \approx k_{\mathrm{FF}}\delta^{-1}t_{\mathrm{s}}D2^{D-1} = O(D2^{D-1}). \tag{12c}$$

The space complexity of the algorithm is proportional to the number of nodes needed to represent the parse tree as: $S_{\mathrm{TF}} = s_{\mathrm{Nd}}N_{\mathrm{Nds}}$, where s_{Nd} is the space needed to store one tree node. Since the number of the tree nodes varies in the range from D to $2^{D+1} - 1$, the space complexity with the free terms ignored is:

$$s_{\mathrm{Nd}}D \le S_{\mathrm{TF}}(n_{\mathrm{w}}, D) \le s_{\mathrm{Nd}}(2^{D+1} - 1), \tag{13a}$$

$$O(D) \le S_{\mathrm{TF}}(n_{\mathrm{w}}, D) \le O(2^{D+1}). \tag{13b}$$

The lower bound is for the two trivial systems with constant points, and the upper bound is for the full tree. The average case is, of course, still exponential in the tree height D:

$$\left\langle S_{\mathrm{TF}}(n_{\mathrm{w}}, D)\right\rangle_{\mathrm{ave}} \approx s_{\mathrm{Nd}}2^D = O(2^D). \tag{13c}$$

Let us stress that the exponential growth in the time and space complexity is in the tree height D, which is the size of our model on this stage (confer Sec. 2), and not in the number of symbols. So, the exponential growth is not a consequence of the algorithmic inefficiency, but of the request for the modelling accuracy. For the theory of ϵ-machines the important thing is that the space complexity in this stage of the model becomes independent from the number of words and symbols emitted by a system and read into the parse tree (at least for the nondiverging systems). Having provided the abundant tree feeds, we expect that $n_{\mathrm{w}} >> D$, and furthermore that $n_{\mathrm{w}} >> 2^{D+1}$. Thus, although the number of tree nodes may grow exponentially with the tree height, it should be much smaller than the number of words and symbols extracted from the time series. We expect that D is chosen to be relatively small, so that the compression of the system presentation is accomplished.

Tab. 2 shows the results of the execution time measurements for three different feeds, performed on our two exemplary dynamical systems. The time measurement function accuracy is below 10^{-8} s, but in the multitasking environment the repeated measurements vary significantly. The expected precaution measures were done by turning off all other applications and several unnecessary processes. Still, in the functions relying on dynamically allocated memory, like in the Tree Feed algorithm, fluctuations of up to ±10 % can be observed.

Besides the unavoidable obstacles that influence the stability of the measurements, the results are very informative. The right column of each feed shows the execution times per one word. Batch iteration plus binary measurement times were already discussed in 3.3. The results for the Tree Feed algorithm prove that $T_{\mathrm{TF}}(n_{\mathrm{w}}, D) = O(n_{\mathrm{w}})$, which is compliant with (10). Furthermore, the Tree Feed algorithm gets faster with the larger number of words, and this is probably because the operating system dedicates the memory accessing

Table 2 Execution times for the batch iteration with binary measurement (Itr. + B.M) and the Tree Feed algorithm, measured for the two exemplary dynamical systems. The tree was fed with 10^4, 10^6, and 10^8 words of length $D = 20$, after skipping the first 100 transient points. The circular buffer size was 10124 elements for the first feed, and the default value of 1048608 elements for the other two. The times spent on the overhead functions and total execution times are included. The right column of each feed is the average time per one word.

$N_w =$	10^4		10^6		10^8	
Dyn. Sys. & Algorithm	Δt /s	$\frac{\Delta t}{N_w}$ /μs	Δt /s	$\frac{\Delta t}{N_w}$ /μs	Δt /s	$\frac{\Delta t}{N_w}$ /μs
Logist. Map						
Itr. + B.M.	0,0004	0,035	0,0359	0,036	3,49	0,035
Tree Feed	0,0523	5,228	2,1636	2,164	119,79	1,198
Overhead	0,0012	0,121	0,0025	0,003	0,10	0,001
Total	0,0539	5,384	2,2020	2,203	123,38	1,234
No cons. 0s						
Itr. + B.M.	0,0008	0,083	0,0826	0,083	8,20	0,082
Tree feed	0,0187	1,867	0,5460	0,546	51,20	0,512
Overhead	0,0016	0,163	0,0048	0,005	0,12	0,001
Total	0,0211	2,113	0,6334	0,634	59,52	0,595

mechanism to the currently running and dominating process. This is followed by a significant decrease of the overhead functions' average time per one word.

Another interesting thing is that the access to the tree nodes by words from a chaotic system like logistic map is more than 2 times longer than for the regular system "no consecutive zeros".

Additional measurements showed that the execution time dependence on D is linear for $D \leq 12$, i.e. for the total occupied memory up to 80 KB. However, it tends to raise more than we would expect from a linear time complexity for $D > 12$ and for larger amounts of dynamically allocated memory. We can suppose that this is due to the behaviour of the memory manager, but this requires further exploration and confirmation.

4.2 The tree comparison algorithm

Once the parse tree is fed with words, we seek for the subtrees of unique morphology and transition probabilities (confer 2.3). In that process the subtrees must be mutually compared. This is done by *Compare Trees* (CT) algorithm implemented in the subclass `CSubABinTree` function `CompareTo`, presented in the Listing 5. This nonrecursive function operates on `this` subtree and compares it to the subtree defined by `pSTr` pointer. Before explaining it, let us stress that the objects of the class `CSubABinTree` use the parse tree nodes that were formed in the tree feed process. So, the subtrees do not create any additional nodes. On creation, only the inherited base class member variables, like height, root pointer, etc., and two more subclass members are added. E.g. it is practical to have the root node string, which, together with the height, uniquely defines the subtree within the main tree. Another thing that may be, and is normally done on the subtree construction, is gathering its statistics. The (sub)tree statistics can be used for the tree comparison, and can be also exposed in the DSA program afterwards.

The function `CompareTo` not only discriminates the subtrees according δ-equivalence relation in (3), it also provides the modeller with more useful information. If the subtrees are not δ-equal, we would like to know whether that happened because of the morphological difference or because of different transition probabilities for the morphologically equal subtrees, as will be discussed soon.

The algorithm starts by setting the Boolean traversal control variable to `false`, and the floating point flag `fFlg` of combined purpose to zero. Via the pointers `pNdC` and `pNdCT` we shall operate with two tree nodes: the first one being from "`this`" tree, and the other from the comparison tree. Before starting the node by node comparison, we may try to terminate the algorithm early. That is, if we have built the tree statistics on creation of the subtree, we can now use it to immediately discriminate the trees with different number of the left and right nodes or the left and right leaves. In that case the special flag value of 2,0 is returned as a sign of the morphological difference. This can spare a lot of time for the large subtrees, especially in the cases of the rich-structured parse trees with diversified subtrees. However, in the worst case of a full parse tree, this will not help at all. In that case all the subtrees will be full too, and they will have the same statistics (check the Tab. 4 in 5.2).

If the tree statistics is not different, the subtrees are to be compared node by node. We first initialize the node pointers to the tree roots by applying `InitPreOrd` function. The subtree root nodes are compared first and separately from the other nodes, because their relations to the parents must be ignored (compare this to the definition of morphs as past independent causal states in 2.3). The descendants in both roots are checked by function `HasChildren` for the four possible outcomes: none, left only, right only, or both. If this is not the same for both subtrees, the algorithm terminates again with `fFlg = 2.0`. If the children structure is the same, the nonrecursive preorder traversal move is made on both nodes. If both of the traversals are not done (`bTrv1Done == bTrv2Done == false`) the `while` loop is entered. For the non-root nodes, besides checking the children structure, the character of the nodes themselves with respect to their parents is mutually compared. They should both be the same type of children to their parent nodes: either both left or both right child. If this is not true, the subtrees are morphologically different and the `fFlg = 2.0` is returned again. Else, we calculate the both nodes' probabilities with respect to their root nodes by `NodeProblty` function. It gives the probability of a substring starting in the root and ending in the observed node of the subtree. The absolute value of the probability difference is stored in `fFlg`, and then checked if it is greater than the maximal probability difference `fDiffMax`. If so, the node string relative to the subtree root is determined by function `NodeFullString` and stored in the `sMaxDiffNd` variable. After the full topological and stochastic comparison of the nodes, the traversal move is made. The `while` loop is continued till at least one of the traversals is done. On the exit from the `while` loop, the flag is set to the `fDiffMax` value if both traversals ended at the same time, and if not, on the morphologically-different flag value of 2,0.

Let us summarize the flag values of the CT algorithm. If `fFlg = 2.0` the subtrees are morphologically different. If the flag is within the domain [0, 1] of the probability function, this is the maximal probability difference found between two corresponding nodes of the two compared subtrees, and `sMaxDiffNd` is the substring which uniquely defines the nodes' position. From the CT algorithm it can be easily proven that if the function returned `fFlg = fDiffMax`, which is a valid probability, then the subtrees must be morphologically identical. Namely, the same type of

Listing 5 Function that implements the Compare Trees (CT) algorithm for comparison of two subtrees

```
float CSubABinTree::CompareTo(const
  CSubABinTree* pSTr, CString& sMaxDiffNd) const
{
  bool bTrv1Done = false, bTrv2Done = false;
  float fDiffMax = 0.f;
  float fFlg;
  CBinTreeNode* pNdC; // Ptr. to the current node
                      // of "this" tree.
  CBinTreeNode* pNdCT; // Ptr. to the current node
                       // of *pSTr tree.
  // The tree statistics comparison (optional):
  if(!(m_TreeStat == pSTr->GetTreeStatistics()))
     return fFlg = 2.f; // Different tree stat.
  pNdC = InitPreOrd(); // Init. preorder traversal
  pNdCT = pSTr->InitPreOrd(); // for both trees.
  // Compare roots:
  if(HasChildren(pNdC) == pSTr>HasChildren(pNdCT))
  {
    pNdC = TravrsPreOrd(pNdC, bTrv1Done);
    pNdCT = pSTr->TravrsPreOrd(pNdCT, bTrv2Done);
    while(!bTrv1Done && !bTrv2Done)
    { // Check nodes below root:
      if(  HasChildren(pNdC) ==
           pSTr->HasChildren(pNdCT) &&
     (IsLChild(pNdC) && pSTr->IsLChild(pNdCT) ||
      IsRChild(pNdC) && pSTr->IsRChild(pNdCT)) )
       // The nodes are topologically equal,
       // compare their probabilities:
      {
        fFlg = float(fabs(NodeProblty(pNdC) -
                    pSTr->NodeProblty(pNdCT)) );
        if(fFlg > fDiffMax)
        {
          fDiffMax = fFlg;
          sMaxDiffNd = NodeFullString(pNdC);
        }
      }
      else
      {
        return fFlg = 2.f; // Morph-different.
      }
      pNdC = TravrsPreOrd(pNdC, bTrv1Done);
      pNdCT = pSTr->TravrsPreOrd(pNdCT, bTrv2Done);
    } // (while)
    if(bTrv1Done && bTrv2Done) fFlg = fDiffMax;
    else fFlg = 2.f;
  }
  else  // Difference at the root level:
  {
    fFlg = 2.f; // Morphologically different.
  }
  return fFlg;
}
```

traversal passed through all their nodes, showing the same node topology for each node pair. So, the returned flag does show the maximal probability difference between the nodes defined by the string `sMaxDiffNd` in the morphologically equivalent subtrees.

Two important things must be stressed about the CT algorithm as realized by the `CompareTo` function:

i. It terminates either on the first morphological (topological) difference found between the tree nodes, or:
ii. It does a complete comparison of all the nodes in two morphologically equivalent subtrees, returning the maximal probability difference and the corresponding node string, as was just proven above.

Instead of solution (ii) above, the comparison could have terminated as soon as the probability difference greater than δ is found. But in that, on average faster solution, only the partial node comparison is performed. Thus, we can possibly overlook the bigger probability differences and even morphological discrepancies further down the tree. In other words, not only that such an algorithm does not provide valuable modelling information (stated in the point (ii) above), but it also cannot determine if the subtrees are morphologically or only stochastically different.

The time complexity of the compare trees algorithm is analogous to the space complexity of the tree feed algorithm in 4.1. It is linear in the number N_{Nds} of the tree nodes: $T(N_{\text{Nds}}) = O(N_{\text{Nds}})$. For a subtree of height L having from L to $2^{L+1} - 1$ nodes, the time complexity $T_{\text{CT}}(L)$ of the tree comparison algorithm is in the range:

$$t_{\text{NdC}} L \le T_{\text{CT}}(L) \le t_{\text{NdC}}(2^{L+1} - 1), \tag{14a}$$

$$O(L) \le T_{\text{CT}}(L) \le O(2^{L+1}). \tag{14b}$$

Here t_{NdC} is the average time needed for comparison of one pair of the related tree nodes. Taking into account that we can discriminate two morphologically different subtrees by comparing their tree statistics, we get even better lower bound:

$$O(1) \le t_{\text{NdC}} L \le T_{\text{CT}}(L). \tag{14c}$$

This leads to the average time complexity of:

$$\langle T_{\text{CT}}(L) \rangle_{\text{ave}} \approx t_{\text{NdC}} 2^L = O(2^L). \tag{14d}$$

The CT algorithm operates on the space of two subtrees, $S_{\text{CT}} = O(2^{L+1})$, plus the small fixed space for storing the subtree data. Since the subtrees use the main tree nodes, its actual space cost is ignorable: $S_{\text{CT,cost}} = O(1)$.

5
Finding morphs and the system's SFA

Being equipped with the (sub)tree comparison algorithm, we are ready to search for all the subtrees which are either morphologically unique, or have unique conditional probabilities of their nodes. To assure proper encapsulation of the extracted morphs, they are organized in class `CATrMorphs` (Automata Tree Morphs). The technical name reflects the fact that in general we search and find several unique subtrees, or morphs. Only if they satisfy the additional condition that their transitions are mutually closed, i.e. that each transition from every morph goes to another defined morph, the `CATrMorphs` object will present a regular SFA model of the system. The class and its most important member variables are shown in Listing 6. Most of the variables used there have a clear meaning.

The concept of *traversal tree*, a type of subtree to which the pointer `m_pTravrsTree` points to, needs some clarification. This is a subtree that grows from the main tree root node and has height $D - L$. The leaves of the traversal tree are the maximal-level nodes (the lowest on the picture) from which we can construct subtrees of height L as potential morph candidates. So, when the morph-finding algorithm reaches the traversal tree leaf level $l = D - L$, it cannot continue further down because the subtrees below would have height less than L and could not be regular morphs. For the same reason the transitions from the traversal tree leaves to their child subtrees cannot be considered as regular either. We say that such transitions "do not close" to the previously found morphs. Thus, as soon as the morph searching algorithm gets to the leaf level of the traversal tree, the morphs do not

Listing 6 The Automata Tree Morphs class for the unique subtrees (morphs) and the Stochastic Finite Automaton (SFA)

```
class CATrMorphs : public CObject
{
  CMainABinTree* m_pMainTree; // Ptr. to main tr.
  UINT m_uMTrHeight; // Main tree height D.
  UINT m_uHeight;   // Subtree (morph) height L.
  float m_fDelta; // Delta probab. param. δ.

  CSubABinTree* m_pTrvrsTree; // Pointer to the
  // traversal tree of height D - L.
  bool m_bSearchDone; // True if the morph search
                      // is done, otherwise false.
  bool m_bMrphTrnsClsd; // True if morph transiti-
                // ons "close" to the found morphs.
  //    ...     ...     ...
  //Typed ptr. array of morphs (unique subtrees):
  CTypedPtrArray<CTypedPtrArray<CObArray,
                 CSubABinTree*> m_MrphPtrArr;
  // Other member variables and functions:
  //    ...     ...     ...
        ...     ...     ...
}
```

Listing 7 Finding all morphs in the parse tree

```
void CATrMorphs::FindAllMorphs(float fDelta)
{
  // Start from the root morph;
  if( FindRootMorph()!= NULL )
  {
    int iMI = 0; // Morph index.
    do
       FindChildMorphs(fDelta, iMI++);
    while( iMI < m_MrphPtrArr.GetSize() );
  }
  m_bSearchDone = true; // Member variable.
}
```

form a regular SFA. This is recorded by setting the `m_bMrphTrnsClsd` variable to `false`.

5.1 Find Morphs algorithm

To find the unique morphs and store all their relevant data, we proceed according to the following sketch of the *Find Morphs Algorithm* (FM):

FM.1 Create a subtree of height L, $1 \le L \le D$, with the root in the main tree root node. By definition this is a unique morph, so store it in the list at position $i = 0$;

FM.2 For the i-th morph inspect if its child subtrees of height L can be created (for the morph's root at level l the condition is $l \le D - L$). If not, go to FM.3. If yes, create a temporary subtree from the root's left child node (if it exists), and proclaim it to be a morph candidate. Compare the candidate will all the morphs in the list. If it is unique, either morphologically or only stochastically, add it to the list and store the morph's transition data in the list of the morph's transitions (the list parallel to the morph list). If the morph candidate is not unique, delete it. Repeat the same procedure for the right child subtree (if it exists). At level $l < D - L$ at least one candidate must exist, but it does not have to be unique. The result of this step is that either 0, 1 or 2 morphs were added to the list, together with their transition probabilities.

FM.3 Increment the index $i \leftarrow i + 1$. Check if new morphs were added to the list in FM.2: if $i <$ *number of list elements* go back to step FM.2, else finish the algorithm.

The step FM.1 is realized by the simple function `FindRootMorph`. The core of the algorithm in step FM.2 is implemented by function `FindChildMorphs`. It is straightforward but a bit lengthy, so it is left for the Appendix B (Listing B1). The function should be easily readable by following the idea outlined in FM.2, and the comments in the code. The whole algorithm is governed by `FindAllMorphs` function, which can be very elegantly realized in C++. The Listing 7 shows its slightly simplified version without the general, prolonged argument list, and a few `ASSERT` statements for checking the internal consistency. In short, the function `FindRootMorph` creates the root morph and returns the pointer to its root node (the same as the main tree root node). If it is not `NULL`, we start the morph search and iteratively call `FindChildMorphs` function from within the `do - while` loop. `m_MrphPtrArray` member variable is a dynamical pointer array template of type `CTypedPtrArray`, and it holds the pointers to the created morphs. According to FM.3, in the `while` condition we compare the incremented index `iMI` to the current size of the dynamical array. If there are remaining morphs enlisted by the function `FindChildMorphs`, the algorithm continues the search for their child morphs. Else it ends with the total of N_{Mrs} morphs found.

Having searched through the parse tree for all the unique subtrees, our modelling result is finished. In a short digression from the algorithmic topic, let's take a look at the final result of the DSA program. Fig. 7 shows an example of the model report from the DSA program. The SFA of the testing "no consecutive 0s" system from Sec. 2 is fully and accurately reproduced ($D = 5$, $L = 2$, $\delta = 0{,}001$). After an abundant feed with 10^7 strings the reconstructed SFA is within $\approx 10^{-4}$ of the predicted model from Fig. 4. The IDs of the left and right submorphs and their conditional probabilities fully define the SFA. Several additional data that are extracted during the modelling and are shown in the SFA report window can help an intelligent modeller (agent) to obtain better results within acceptable modelling space and time.

5.2 The analysis of the Find Morphs algorithm

As first we shall comment on a special property of the above algorithm. From the FM.2 step (confer also Listing B1) we conclude that the algorithm will not generate new morphs if the morph candidates are recognized as equal to the already existing morphs. This will result in closing of the morphs' transitions between insofar found morphs and in the termination of the algorithm. The subtrees that are lower than the last compared morph candidates are not investigated at all. In some branches of the main tree this may happen later, and in the others sooner, depending on the main tree structure. We summarize the consequences of this in the following conclusions:

i. The FM algorithm is ***superficial***. In general case it does not investigate all subtrees of height L within the main tree of height D ($L \le D$; $D, L = 1, 2, \ldots$).
ii. Property (i) helps the termination of the algorithm, and the formalization of the found subtrees and their mutual transitions as a regular SFA.
iii. The investigated correlations within the words w_D of length D can be as short as $L + 1$ (the root morph child subtrees are always investigated), and can prolong down to the length D if all possible subtrees are investigated.
iv. The lack of thoroughness can be justified by the fact that as we go down the traversal tree and the main trees, the statistical significance and the probability accuracy decreases as the counts in the tree nodes decrease.

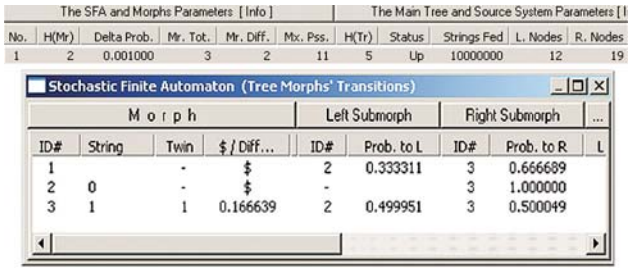

Figure 7 The SFA model for the system "no consecutive zeros" obtained by the DSA program. The outer dialog box shows that one model (a `CATrMorphs` object) was created, with $L = 2$, $\delta = 0{,}001$. The total of 3 morphs are found of which 2 are morphologically unique, within the main tree of height $D = 5$. In the inner box, the 3 found morphs are listed. $ denotes morphologically unique morphs. Otherwise ID of the earliest "twin" is written, together with the maximal probability discrepancy between the morph and its twin. The IDs of the left and right submorph and their conditional probabilities fully define the system's SFA.

To inspect the influence of the above shortcomings, and enable a thorough comparison of all the subtrees, full-traversal versions of the FM algorithm are designed (*Thorough Find Morphs* algorithms). Their description and analysis, including some epistemological implications that can be deducted, will be presented elsewhere.

The time complexity analysis of the Find Morphs algorithm is lengthy, so here we present only the results.

With the tasks of the FM.2 step that only check the existence and prepare the morph candidates (accomplished within the `FindChildMorphs` function), we shall associate time constant t_{FCM}. Here the subtree comparison is not included. In each step maximally 2 morph candidates can be found. On their creation, time t_{NdTS} is spent per candidate node to gather the subtree statistics. Both candidates are compared to all the previously found morphs. If they are recognized as unique, they are added to the morph list. Accounting for the worst cases of both the FCM and the CT algorithms, the time complexity for the full parse tree with $N_{\mathrm{Nds}} \approx 2^{D+1}$ nodes is:

$$\begin{aligned} T_{\mathrm{FM}}(D,L) &\approx t_{\mathrm{NdC}}\, 2^{D+1}(2^{D-L+1}+1) + t_{\mathrm{NdTS}} 2^{D+2} + t_{\mathrm{FCM}} 2^{D-L} \\ &\approx t_{\mathrm{NdC}} 2^{2D-L+2} \approx t_{\mathrm{NdC}} 2^{-L} N_{\mathrm{Nds}}^2 \\ &= O(2^{2D-L+2}) = O(N_{\mathrm{Nds}}^{2-x}),\ \ x = L/(D+1). \end{aligned} \tag{15}$$

The assumption here is $2^{D-1} >> 1$. This worst case happens if the probability differences greater than δ are found between otherwise morphologically equal (full) subtrees. Obviously, this can be caused by too low δ, as shown in the last column of Tab. 4 for logistic map. The time complexity is severely exponential in the model parameters, but less than quadratic in the number of the tree nodes. If we assume that the average number of the subtree comparisons occurred, and further, that on average only one new morph candidate is found, the complexity gradually decreases. The typical cases are summarized in Tab. 3. The best cases are for periodical systems (case 1), and when the root morph candidates are equal to the root morph (case 2). Depending on the tree structure, other low-order time complexities can occur. In case 3 the subtrees are discriminated by comparing their statistics in time T_{CTS} independent of their structure. The cost is time T_{TS} needed for gathering the node statistics. However, it is spent only once on the creation of the subtree.

As for the space complexity, FM algorithm uses the tree nodes from the main tree. Thus it operates mainly within the space of complexity as determined in (13). The little constant extra space for the morph candidate's subtree data

Table 3 The range of the time complexities $T_{\mathrm{FM}}(D,L)$ for the Find Morphs (FM) algorithm, from the best to worst case

Case	$O(D, L)$	*Assumption / Description*
1.	$O(L^3)$ $O(L^4)$	Period-n sys.: tree stat.-discrimination, $L = n$, subtr. compar. "node by node".
2.	$O(2^{L+1})$,...	Mrph.cnd. = root morph (superficiality).
3.	$O(2^D)$ $O(2^{2D-2L-2})$	$T_{\mathrm{TS}} \geq T_{\mathrm{CTS}}$, $2^{D-2L-2} < 1$, [tree-stat. $T_{\mathrm{CTS}} \geq T_{\mathrm{TS}}$, $2^{D-2L-2} \geq 1$, comp.].
4.	$O(2^{2D-L-2})$	$2^{D-L-2} >> 1$, as (5), but aver. time T_{CT}.
5.	$O(2^{2D-L-1})$	$2^{D-L-2} >> 1$, 2 cand., 1 new morph.
6.	$O(2^{2D-L+1})$	$2^{D-L} >> 1$, as (7), but aver. time T_{CT}.
7.	$O(2^{2D-L+2})$	$2^{D-L} >> 1$, worst case, max. subtrees.

can be neglected. The total of N_{Mrs} morphs and their transitions are stored in the lists of size $O(N_{\mathrm{Mrs}})$, with $N_{\mathrm{Mrs}} \leq 2^{D-L+1} - 1$. The number N_{Mrs} is mostly governed by the character of the system, and only indirectly by D and L. If we introduce the space constant s_{Mr} as the total space needed to store the morph's pointers and transition data of one morph, for the space cost $S_{\mathrm{FM,cost}}$ we can write:

$$S_{\mathrm{FM,cost}}(D,L) \approx s_{\mathrm{Mr}} N_{\mathrm{Mrs}} = O(N_{\mathrm{Mrs}}). \tag{16}$$

Unless the number of found morphs explodes, the algorithm mostly uses the storage of the tree feed algorithm.

The execution times behave as expected from the previous deliberation. If the number of words fed into the tree is abundant with respect to the selected δ (confer 4.1), the number of morphs found will reflect the structure of the source. If the tree feed is insufficient, the number of morphs can explode because of the insignificant statistics. In this case we must either release δ, or enlarge the tree feed, or lessen the correlation length L (if allowed).

In Tab. 4 we have summarized the results of the FM algorithm for three different feeds of the parse tree. For the logistic map, the lowest feed with 10^4 words is insufficient for the morphs with $L = 10$, even with the loose $\delta = 0{,}01$. The morphological differences appear on the levels $l \geq 10$ because some nodes are not formed yet. Thus the algorithm inevitably finds that all possible subtrees are morphologically unique morphs. By lowering L to 9 we diminish the depth of our search and the length of correlation, to find only 1 morph. For the system "no consecutive zeros" the feed with 10^4 words is sufficient to produce a regular SFA for $L = 10$, $\delta = 0{,}01$, but with nonminimal number of 4 *states*. For the higher accuracy of $\delta = 0{,}001$, from the possible 375 subtrees even 276 are morphologically unique, contributing to the total of 287 morphs (76,5 %).

With the feed of 10^6 words, we can establish some of our results on the "medium" accuracy level (the left columns). However, for both systems we get an explosion of the morphs for the higher accuracy (the right columns). For the logistic map, the maximal number of morphologically different morphs (2047) indicates that the subtrees on the deeper levels are not yet full. The explosion in the number of subtrees checked and the morphs found resulted also in the blast of the execution times. For the first system the number of subtrees recorded and checked rose from 3 to the maximal 2047, and the execution time rose by factor of almost 10^4 (≈13 times more). This should resemble the jump from the cases 1 and 2 to the case 3 in Tab. 3.

Even bigger growth factor of execution times can be spotted in the last two columns for the logistic map – $7{,}3\times10^5$. Since in that case all the subtrees are full and have

Table 4 Results of the Find Morphs (FM) algorithm for the two exemplary dynamical systems. The tree of height $D = 20$ was fed with N_w words, to obtain morphs of height L and δ-accuracy. The maximal number of subtrees $N_{St,max}$ (equal to the number of nodes in the traversal tree) is 2047 for the first system and 375 for the second. *Legend*: $N_{St,rc}$= number of subtrees recorded and checked, $N_{Mrs,md}$ = num. of morphologically different morphs, $N_{Mrs,tot} = N_{Mrs}$ = total num. of morphs, ($N_{Mrs,md} \le N_{Mrs,tot} \le N_{St,rc} \le N_{St,max}$), T_{FM}/s = execution time in seconds. Regular SFAs are marked with the check sign.

N_w =	10^4		10^6		10^8	
Logist. Map	✓		✓		✓	
L =	9	10	10	10	10	10
δ =	0,01	0,01	0,001	0,0001	0,0001	0,00001
$N_{St,rc}$ =	3	2047	3	2047	3	2047
$N_{Mrs,md}$ =	1	2047	1	2047	1	1
$N_{Mrs,tot}$=	1	2047	1	2047	1	2047
T_{FM}/s	0,0010	0,1780	0,0022	18,58	0,0023	1826,8
No cons. 0s	✓		✓		✓	
L =	10	10	10	10	10	10
δ =	0,01	0,001	0,005	0,001	0,0001	0,00001
$N_{St,rc}$ =	8	288	6	257	6	288
$N_{Mrs,md}$ =	2	276	2	5	2	276
$N_{Mrs,tot}$=	4	287	3	255	3	287
T_{FM}/s	0,0014	0,0180	0,0009	2,111	0,0009	2,724

the same morphology ($N_{Mrs,md} = 1$), all the subtrees' nodes are compared. Every encountered subtree is enlisted, thus enlarging the number of the subtree comparisons for 2 with every added morph. This corresponds to the worst case in Tab. 3. To back up these conjectures with more quantitative arguments, we would need to have better insight in the time constants, especially those connected with the dynamical memory managing of the operating system (confer comments under Tab. 2).

6 Conclusion

A programming endeavour that implements an innovating scientific application requires the use of suitable data structures and algorithms, many of which have to be tailor-made. In our implementation of the ϵ-machines theory by the DSA program this is shown to the full. An ϵ-machine model of a dynamical system is expressed directly by the interrelationship of the data structures excerpted from the input data. After simulating a dynamical system and providing a time series that is being fed into an ϵ-parse tree, morphologically and stochastically unique ϵ-subtrees are found. They are recognized as system states of the stochastic finite automaton model. The realization of this interdisciplinary modelling scheme gave rise to development of several original programming concepts and solutions. A few of the included algorithms can be considered as rather fundamental and generally applicable.

The used data structures and the accompanying algorithms are presented and analysed for their time and space complexities. Besides the calculated time dependencies, time measurements for the crucial modelling steps were given. This was used to illustrate the influence of the size of our circular buffer on the performance of the batch iterations. Also, we have explained the properties of the time-critical Tree Feed algorithm. It is linear with respect to the number of words fed into the tree, but becomes exponential in the tree height because of the statistical demands of our modelling scheme. The time measurements back up our predictions accurately for the algorithms that use nondynamic memory (the system iteration and binary measurements), and approximately for those with data stored in dynamically allocated memory (the Tree Feed and other algorithms operating on the tree structures).

Among more complex algorithms we have presented an original nonrecursive version of the Compare Trees algorithm. It first compares two binary (sub)trees for having identical morphological structure, and if so, it determines the greatest conditional probability difference between the corresponding nodes. The algorithm is thorough, in the sense that for the two morphologically equal trees, it will traverse through all their nodes, using the nonrecursive preorder tree traversal algorithm [11].

Our modelling scheme culminates in the Find Morphs algorithm. Its crucial function is `FindChildMorphs`, which uses the above Compare Trees algorithm to compare a morph candidate subtree to all the previously found unique subtrees or morphs, and to enlist it to the morph list if it is unique itself. The function is called from within the `FindAllMorphs` function, which is characterized by a superficial algorithm that will stop if no new morphs were added to the list. The derived time complexity order of magnitude for this algorithm ranges from the constant, to linear, and finally to exponential dependence in the tree height. It is found to be generally consistent with the measured execution times.

In this comprehensive exposure of the subject we have tried to give the complete picture. This forced us to have left out several interesting details and a few special subjects for separate elaborations. From those, the comparison of the recursive versus nonrecursive traversals is given in [11]. In the similar manner the implemented nonrecursive version of the Compare Trees algorithm should be compared to its recursive version. Another interesting topic is the presentation and analysis of the Thorough Find Morphs algorithm announced in 5.2.

We hope that this presentation of our work provided valuable information to the readers from both the physical-modelling and the computer science provenance. Additionally, if some of the presented data structures and algorithms find a more general use, the primary goal of this paper would be even surpassed.

7 References

[1] Crutchfield, J. P.; Young, K. Inferring Statistical Complexity. // Physical Review Letters, 63, 2(1989), pp. 105-108.

[2] Crutchfield, J. P. Computational mechanics publications: http://csc.ucdavis.edu/~chaos/chaos/pubs.htm (May 2012).

[3] Logozar, R.; Lovrencic, A. The Modeling and Complexity of Dynamical Systems by Means of Computation and Information Theories. // Journal of Information and Organizational Sciences (JIOS), 35, 2(2011), pp. 173-196.

[4] Logozar, R. Modeling of Dynamical Systems by Stochastic Finite Automata, Master Thesis (in Croatian), Faculty of Electrical Engineering and Computing, University of Zagreb, 1999.

[5] Crutchfield, J. P. Knowledge and Meaning. // Modeling Complex Phenomena, Lam, L.; Naroditsky, V. editors. Springer, Berlin, (1992), pp. 66-101.

[6] Devaney, R. L. An Introduction to Chaotic Dynamical Systems, 2nd Edition. Addison-Wesley, Redwood City, California, 1989.

[7] Horowitz, E; Sahni, S; Rajasekaran, S. Computer Algorithms, Computer Science Press, New York, 1998.

[8] Aho, A.; Hopcroft J. E.; Ullman J. D. Data Structures and Algorithms. Addison-Wesley, Reading, MA, 1983, reprinted with corrections 1987.

[9] Cormen, T. H. et al. Introduction to Algorithms, 3rd edition, MIT Press Cambridge MA, 2009.
[10] Wirth, N. Algorithms & Data Structures (New Edition), Prentice/Hall International, London, 1986.
[11] Logozar, R. Recursive and Nonrecursive Traversal Algorithms for Dynamically Created Binary Trees, to be published in Computer Technology and Application (CTA), David Publishing, Vol. 3, No. 5.

The DSA program availability

All interested readers are welcome to contact the author at his address below for a current version of the Dynamical Systems Automata program. It may be freely used for scientific purposes.

Author's Address

Robert Logozar, MS. in CS.
Polytechnic of Varazdin
J. Krizanica 33
HR-42000 Varazdin, Croatia
robert.logozar@velv.hr

Appendix A

Listing A1. A single batch iteration (either the first one or some later), which includes the binary measurement. This function illustrates the execution time measurement by help of the `CHRTimer` class object. The local variables storing the measured times and the numbers of iterations done are updated by `UpdateFeedTmAndNStrFed1` function.

```
UINT CDynSys1D::DoABatchIterInclBI
     (CBinInstrument* pBI, bool bFrmLastPnt,
      UINT uStrl)
{
 UINT uNItr;
 CHRTimer hrTimer; //
 double dtsDur; // Temporary time duration.
 if( m_uNBtItrDone == 0 || !bFrmLastPnt )
 {
  hrTimer.StartTimer();
  uNItr = FirstBatchIteration();
  dtsDur = hrTimer.StopTimer();
 }
 else
 {
  hrTimer.StartTimer();
  PrepBatchIterRepeat(uStrl);
  uNItr = BatchIteration(uStrl);
  dtsDur = hrTimer.StopTimer();
 }
 UpdateFeedTmAndNStrFed1(dtsDur, uNItr,
      m_dPtItrTm1, m_uNPtItrtdTm1,
      m_dPtItrTm2, m_uNPtItrtdTm2);
 if(m_bInclBinInstr)
  pBI->BinMeasurments(this); //Bin. measurement.

 return uNItr;
}
```

Listing A2. Iteration of the rule-defined system "no consecutive zeros" with randomly generated numbers. For the consistency reasons the random numbers are normalized to the interval [0, 1) (statements for `dNorm` and `d2Norm`).

```
void CDS1DRR_NoConsec0s::
  BatchIteration(UINT uStrl)
{
 // Regular expression: (0 + e)(1 + 10)*
 _ASSERT( uStrl >= 1);
 if(uStrl == 1) uStrl++;
 const double cd05 = 0.5;
 const double cd20 = 2.0;
 double* p = Getpdx() + uStrl - 1;
 double* p1 = Getpdx()+ GetuNArrPts();
 double dNorm = RAND_MAX + cdSmall1;
 double d2Norm = 2.0 * RAND_MAX + cdSmall1;
 double dx;
 // Seed is initialized by InitIteration func.
 // Checking the previous point:
 if(*(p - 1) >= cd05)
 {
  dx = double(rand())/dNorm; // 2nd parenth.
  if(dx < cd05) *(p++) = dx;
 }
 // 2nd parenthesis in the regular expression:
 while(p < p1 - 1)
 {
  if(dx >= cd05 )  // The prev. rand. num. (1)
    *(p++) = dx; // stay unused, use it here.
  else    // Gener. 1, if not done before, or:
    *(p++) = cd05 + double(rand())/d2Norm;

  dx = double(rand())/dNorm;
  if(dx < cd05) *(p++) = dx; // - generate 10.
 }
 // The case when 1 symbol was generated:
 if(p < p1) *p++ = cd05 + double(rand())/d2Norm;
 ASSERT(p == p1);
 CDynSys1D::BatchIteration(uStrl);
}
```

Appendix B

Listing B1. Finding the child morphs (according to FM.2).

```
hasChldrnType CATrMorphs::FindChildMorphs
      (const float fDelta, const int iMI)
{
 CSubABinTree* pSubTr;
 UINT uNMrphs = mrphTrnsPtrArr.GetSize();
 float fFlg;        // Local flag variable.
 float fMinDiff;  // Minimal delta-difference.
 int iMinDiff;   // Index of the morph for
                 // which fMinDiff is found.
 CString sMaxDiffNd;
 hasChldrnType chldType, newMorphs;
 CBinTreeNode* pMRtNdNw;  // A node pointer.
 CTrMorphsTrans* pTrnNw; // A mrph. trans. ptr.
 const CBinTreeNode* pMRtNdMI =
                  mrphPtrArr[iMI]->GetpRoot();
 // Pointer to the morph's transition:
 CTrMorphsTrans* pTrnMI = mrphTrnsPtrArr[iMI];
 newMorphs = none; // Initialization.
 // If there are no children, by def. of the
 // CABinTree this is the tree leaf level:
 chldType = pTrvrsTree->HasChildren(pMRtNdMI);
 if (chldType == none)
 {
  m_bMrphTrnsClsd = false;
  switch(pMRtNdMI->HasChildren())
  {
  case leftOnly: pTrnMI->m_iLSubMrph = -2;
       break;
  case rightOnly: pTrnMI->m_iRSubMrph = -2;
       break;
  case both: pTrnMI->m_iLSubMrph = -2;
             pTrnMI->m_iRSubMrph = -2;
       break;
  }
 }
 else
 {
  if( (chldType == leftOnly) ||
      (chldType == both) )  // Check the left
  {                         // child subtree.
    fFlg = fMinDiff = 2.f;
    iMinDiff = -1;
    sMaxDiffNd = "";
    pMRtNdNw = pMRtNdMI->GetpLChild();
    pSubTr = NewSubTrMrphCand(pMRtNdNw);
    // Compare the subtree to all the morphs
    // in the array:
    UINT uI = 0; // uI index is needed below.
    for( ; (fFlg > fDelta) && (uI < uNMrphs);
           uI++)
    {
```

```
      fFlg = pSubTr->CompareTo(mrphPtrArr[uI],
                               sMaxDiffNd);
        if(fFlg < fMinDiff)
        {
          fMinDiff = fFlg; iMinDiff = uI;
        }
      }
      if(fFlg <= fDelta) // Subtree delta-equal
      {                  // to (uI - 1)-th morph.
        if(pTrnMI->m_iLSubMrph == -1)
        {
          pTrnMI->m_iLSubMrph = uI - 1;
          pTrnMI->m_fLSubProb =
                      pSubTr->SubTreeProblty();
        }
        delete pSubTr; // Delete after new!
        m_bMrphTrnsClsd = true; // The child
          // subtree is a previously found morph.
          // The morph transitions are closed.
       }
       else // The subtree is a new morph!
       {
          pTrnMI->m_iLSubMrph =
                      mrphPtrArr.Add(pSubTr);
          pTrnMI->m_fLSubProb =
                      pSubTr->SubTreeProblty();
          pTrnNw = new CTrMorphsTrans
                    (pSubTr->GetsRoot(), fMinDiff);
          pTrnNw->m_iPrntMrph = iMI;
          pTrnNw->m_iMinDiff = iMinDiff;
          pTrnNw->m_sMaxDiffNd = sMaxDiffNd;
          mrphTrnsPtrArr.Add(pTrnNw);
          newMorphs = leftOnly;
       }
     }
    if( (chldType == rightOnly) ||
        (chldType == both) )  // Check the right
    {                         // child subtree.
      // The code for the right child subtree -
      // - analogous to the code for the left
      // subtree:
        ...        ...        ...
    }
  } // (else)
  return newMorphs;
}
```