*ISSN 1064-2307, Journal of Computer and Systems Sciences International, 2007, Vol. 46, No. 5, pp. 792–801. © Pleiades Publishing, Ltd., 2007.*
*Original Russian Text © P.G. Lobanov, A.A. Shalyto, 2007, published in Izvestiya Akademii Nauk. Teoriya i Sistemy Upravleniya, 2007, No. 5, pp. 127–136.*

**ARTIFICIAL INTELLIGENCE**

# Application of Genetic Algorithms for Automatic Construction of Finite-State Automata in the Problem of Flibs

**P. G. Lobanov and A. A. Shalyto**
*St.-Petersburg State University of Information Technologies, Mechanics, and Optics, ul. Sablinskaya 14, St. Petersburg, 197101 Russia*

Received January 9, 2007; in final form, May 28, 2007

**Abstract**—A genetic algorithm is proposed to solve the problem of flibs, which simulate the behavior of a simplest living being in a simplest environment. Computational experiments have been performed demonstrating the efficiency of this algorithm compared with the existing algorithm.

**DOI:** 10.1134/S1064230707050115

## INTRODUCTION

There are complex problems that can be efficiently solved with the help of finite-state automata [1]. In most cases, the synthesis of automata is performed heuristically. Therefore, it is of current interest to solve the problem of formalization of methods for the construction of automata and automation of these methods. There is a well-known class of problems (for example, prisoner's dilemma [2], "intelligent" ant [3], and synchronization [4] and classification of density [5, 6] for cellular automata), where genetic algorithms [7] can automatically construct automata solving these problems. In this paper, we consider a member of this class, the problem of flibs [8].

## 1. THE PROBLEM OF FLIBS

This problem is to simulate a simplest living being capable of predicting the periodic variations in a simplest environment. The living being is simulated by a finite-state automaton, and genetic algorithms make it possible to automatically construct an automaton predicting the environmental changes with a sufficient accuracy. Thus, one needs to generate a "device" (predictor) that estimates the future environmental changes with a maximum probability. This problem was solved in [8] by using one of the genetic algorithms [8]. Here, the accuracy of predictions by the constructed automaton is not sufficiently high. This is largely caused by the method used for constructing the next-generation entities (automata). The aim of this paper is to develop an approach eliminating this deficiency. We propose a C# program that implements both this and the existing approaches, which makes it possible to compare the approaches.

## 2. STATEMENT OF THE PROBLEM

An important feature of living beings is that they are able to predict the environmental changes. A simplest model of a living being can be represented by a finite-state automaton. Such finite-state automaton models are called flibs (an abbreviation of *finite living blobs*) [8].

The flib input variable can take one of the two values: 0 or 1. This variable describes the state of the environment at the current time instant. The environment is so simple that it has only two states. The flib changes its state and generates the value of the output variable. This value corresponds to the possible environmental state at the next time instant. The flib aims at predicting the actual state of the environment at the next time instant. This can be done due to the periodicity of its changes. Here, it is assumed that the more the accuracy of the flib-predicted environmental change, the higher its chances to survive and leave offspring.

## 3. THE DESIGN OF GENETIC ALGORITHMS IN THE PROBLEM OF FLIBS

Let us describe the designs of the existing and proposed algorithms.

### *3.1. The Existing Algorithm*

The search for an optimal predictor is performed by the following genetic algorithm [8].

(1) A generation of random flibs is created.

(2) The number of changes in the environmental state predicted correctly by each of these flibs is calculated.

(3) The worst and best predictors in the generation are determined.

(4) The best predictor is *crossed* with a randomly chosen flib.

(5) The need of applying an *operator of mutation* to the resulting flib is randomly determined. If needed, this operator is applied to the flib.

(6) The worst predictor in the generation is replaced by the flib resulting from crossover. After this replacement, it is assumed that a new generation is created.

(7) If one of the flibs reaches a 100%-prediction level or the program is terminated by the user, the algorithm stops. Otherwise, we go to item (2).

The algorithm for the generation of random flibs, as well as the operators of one-point crossover and mutation, will be considered in detail in Sections 4, 7, and 8.

### *3.2. The Proposed Algorithm*

In this paper, to create a new generation, we use a method distinct from the one described in Subsection 3.1. Here, we apply the tournament selection [9] and principle of elitism (the new generation is added by a single or several better entities of the previous generation) [10]. The number of flibs in the generation is called the size of the generation.

The proposed algorithm has the following form.

(1) A current generation of random flibs is created.

(2) The number of changes in the environmental state predicted correctly by each of these flibs is calculated.

(3) A new generation of flibs is constructed:

(a) a new empty generation is created and the best predictor from the previous generation is added into it;

(b) two pairs of flibs are randomly taken from the current generation;

(c) the best predictor of each pair is selected;

(d) the best predictors of these pair are *crossed*;

(e) one randomly determines the need of applying the *mutation operator* to the resulting flib. If needed, this operator is applied to the flib;

(f) a new operation ("restoration of links between automaton states") is applied to the flib;

(g) the flib is added to the new generation;

(h) go to item (b) if the size of the new generation is smaller than the size of the current generation.

(4) The current generation of flibs is replaced by the new generation.

(5) If the number of generations is smaller than the user-defined number, return to item (2).

## 4. FLIB IMPLEMENTATION

Let us describe the existing and proposed methods for the flib implementation.

**Table 1.** A three-state flib

| State | 0 | 1 |
|---|---|---|
| A | 1, B | 0, A |
| B | 0, C | 0, A |
| C | 1, A | 0, B |

### *4.1. The Existing Method*

The behavior of the flib is given by the table of branches and exits. Table 1 presents an example of a flib with three states: *A*, *B*, and *C*. Let us represent this table as the string 1B0A0C0A1A0B used in [8]. The number of elements in this string is four times greater than the number of flib states. Let us enumerate the flib states and elements of its specifying string. The first state and first element are assigned to zero. If the flib is in the state with a number $s$ and the current state of the environment is $i$, then the future state to which the flib will pass is contained in the element of the string specifying the flib. The number of this element is $4s + 2i + 1$. The value of the output variable generated by the flib is contained in the element with the number $4s + 2i$.

To create a random flib, it is required that the string elements be given randomly. Let us describe the algorithm for creating a random flib given by the string.

(1) A loop over all elements of the string specifying the flib:

(a) if the number of an element is even, the element is assigned to one of the possible environmental states chosen randomly;

(b) otherwise, the element is assigned to one of the possible flib states chosen randomly.

### *4.2. The Proposed Method*

In this paper, we use another method of flib encoding, which is implemented with three classes: *Flib, State*, and *Branch* (see Listing 1 in the Appendix). The main class of the flib implementation is taken to be the class *Flib.* The classes *State* and *Branch* implement its states and branches, respectively. Each of these classes has a method *Clone* designed for cloning objects.

The array *_states* in the class *Flib* contains the flib states. The field *_curStateIndex* is used to store the number of the current state of the flib in the array*_states.* The number of correctly predicted input characters is stored in the filed *_guessCount.* The method *Step* transfers the flib to a new state and, if necessary, changes the number of correctly predicted characters. The method *Nulling* returns the flib to the original state and nulls the number of correctly predicted input characters.

The array *_branches* of the class *State* includes arcs of changes from a given state. The number of an element in the array corresponds to the input variable. The variables *_stateIndex* and *_output* of the class *Branch* contain the number of the state reached by the flib going along this arc and the value of the input variable. The constant *TARGET_COUNT* defines the numbers of values that can be taken by the output variable generated by the flib.

Let us describe the algorithm for creating a random flib.

(1) Objects corresponding to the flib states are created.

(2) Each object undergoes the following operation:

(a) for a state, the transfers from it are generated;

(b) for each transfer, the number of the future state of the flib and value of the output variable are determined randomly.

## 5. GENERATOR OF INPUT SIGNAL

As an input signal for flibs (like in [8]), we use a recurrent sequence of bits (bit mask) specifying changes in the environmental states. This mask is looped in the program. The code class for generating the input signal is presented in Listing 2 of the Appendix.

## 6. EVALUATION FUNCTION

The best flib is that with a maximum of correct predictions of the input signal. The evaluation function is taken to be the number of correctly predicted characters (the field *_guessCount* in the class *Flib*). When a new generation is created, all the flibs available there are put into the original state with the help of the method *Nulling*. Then, the fitness of flibs is determined by sending to their inputs several input characters (by default, their number is equal to 100). After this, a new generation of solutions can be constructed, using as the flib fitness the value of the field *_guessCount*, containing the number of correctly predicted characters.

## 7. ALGORITHMS OF THE OPERATOR OF ONE-POINT CROSSOVER

Let us consider the existing and proposed algorithms for the operator of one-point crossover.

### *7.1. The Existing Algorithm*

In [8], the new flib is generated from two parents by an operator of one-point crossover. Let us describe the algorithm of this operator for a flib encoded by a string of length $m$.

(1) A random number $j$ is chosen in the range between 0 and $m - 1$.

(2) The elements with numbers smaller than or equal to $j$ of the string that specifies the first parent flib are copied into the string describing the new flib.

(3) The elements with numbers greater than $j$ of the string that specifies the second parent flib are copied into the string corresponding to the new flib.

### *7.2. The Proposed Algorithm*

To use the operator of one-point crossover for the flib implemented as an object of the class *Flib*, the algorithm of [8] should be modified. Here, the proposed algorithm has the following form.

(1) The number of a state of the new flib is chosen randomly.

(2) This flib is added by the states of the first parent, with their numbers being smaller than the chosen number, and by the states of the second parent, with their numbers being higher than the chosen number.

(3) A new state generated from the states of the first and second parents (corresponding to the chosen number) is formed and added. The algorithm of the state formation is similar to the algorithm of the operator of one-point crossover described in Subsection 7.1.

The description of this algorithm can be found in Listing 3 of the Appendix.

## 8. ALGORITHMS OF THE MUTATION OPERATOR

Let us consider the existing and proposed algorithms for the mutation operator.

### *8.1. The Existing Algorithm*

The algorithm of the mutation operator used in [8] has the following form.

(1) An element of the string specifying the flib is chosen randomly.

(2) If the number of the element is even (the element contains the value of the output variable generated by the flib), the value of the variable is inverted.

(3) If the number of the element is odd (the element contains the flib state), the current state of the flib is replaced by the next one.

### *8.2. The Proposed Algorithm*

In this paper, the following algorithm of the mutation operator is proposed.

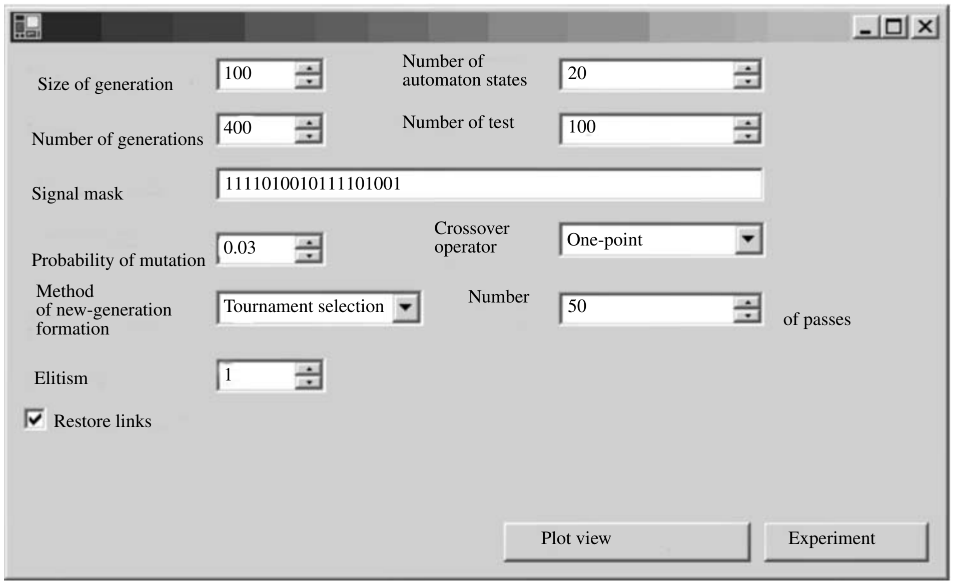
(1) The flib state is set randomly.

(2) An arc is chosen from this state randomly.

(3) It is determined randomly what will be changed, the value of the output variable generated by the flib or the number of the state reached by the flib going along this arc:

(a) if it has been confirmed that the value of the output variable is to be changed, this variable is assigned to the value of the environmental state chosen randomly;

(b) if it has been decided that the number of the state is to be corrected, the number of the state reached by the flib is assigned with the number of a randomly chosen state of the flib.



Size of generation 100

Number of automaton states 20

Number of generations 400

Number of test 100

Signal mask 1111010010111101001

Probability of mutation 0.03

Crossover operator One-point

Method of new-generation formation Tournament selection

Number 50 of passes

Elitism 1

[✓] Restore links

**Fig. 1.** User-interface window.

The class implementing this mutation operator can be found in Listing 4 of the Appendix.

## 9. RESTORATION OF LINKS BETWEEN STATES

When a new generation created by the operators of crossover and mutation, the branching arcs in the flibs are changed randomly. This can lead to states that cannot be reached from the original state for any sequence of changes in the environmental states. These states are called *inaccessible* states. The states that can be reached from the original state for some sequence of changes in the environmental states are called *accessible* states. The algorithm for the restoration of the links between states changes the branching arcs in the flib so that there will be no inaccessible states.

The class *FlibRestorer*, implementing the algorithm of restoration between states, can be found in Listing 5 of the Appendix. The proposed algorithm operates in the following way.

(1) The list of *accessible states* is formed (the method *InitIndexList*). This is performed using the function *AddIndex* of recursive traversal of states.

(2) A complete loop is done. If the current state is not *accessible*, the following operations are enabled for it:

(a) a random state is chosen from the list of *accessible states*;

(b) a likewise random arc is determined from the chosen state;

(c) in the current state, an arc of this state is replaced by another leading to the same state as the arc obtained in item (b);

(d) the arc chosen in item (b) is replaced by another arc leading to the current state;

(e) the best of *accessible states* is updated by appending the current state and all the states that can be reached from it.

## 10. PROGRAM FOR EXPERIMENTS ON FLIBS

Based on the considered algorithms (existing and proposed), we had written a program with its user interface having the form shown in Fig. 1. This program makes it possible to implement the algorithms with the help of the dropdown list “Method of Formation of New Generation”. In the proposed algorithm, the elitism can be changed in the range from zero to the size of the generation. The program admits the choice of the generation size and the number of generations. Also, one can set the mutation probability (in the range between zero and unity) and the type of crossover operator (one-point or two-point). The program can generate a bit mask describing the environment. For each flib, one can set the number of its states and determine whether the algorithm of link restoration will be used.

To compare the efficiency of genetic algorithms, it is conventional to conduct repeated runs on the same sets of test data and compare the averaged results. Thus, the program is capable of doing an automatic run of the algorithm at a given number of times and yielding averaged results. The algorithm can be executed for a user-defined number of runs. The listings presented in the Appendix constitute the core of the program for the proposed approach.

**Table 2.** The results of the first experiment

| Algorithm | Restoring links between states | Results | | |
|---|---|---|---|---|
| | | worst | averaged | best |
| Existing | – | 70 | 78.3 | 88 |
| Proposed | – | 83 | 92.26 | 100 |
| | + | 84 | 93.48 | 100 |

**Table 3.** The results of the second experiment

| Algorithm | Restoring links between states | Results | | |
|---|---|---|---|---|
| | | worst | averaged | best |
| Existing | – | 71 | 82.2 | 93 |
| Proposed | – | 86 | 92.2 | 94 |
| | + | 87 | 93.06 | 94 |

**Table 4.** The results of the third experiment

| Algorithm | Restoring links between states | Results | | |
|---|---|---|---|---|
| | | worst | averaged | best |
| Existing | – | 70 | 75.86 | 87 |
| Proposed | – | 83 | 90.44 | 97 |
| | + | 86 | 92.72 | 97 |

## 11. GENERAL REQUIREMENTS TO EXPERIMENTS

All the experiments was conducted for a generation size of 100 and a mutation probability of 0.03. The number of environmental influences on the flib is taken to be equal to 100. Therefore, the number of correctly predicted characters is equal to the accuracy of character prediction in percents. Tables 2–4 present the experimental results with an indication of the minimum, maximum, and average accuracy of prediction of automatically generated flibs.

The plots in Figs. 2–4 show the number of generations (axis of abscissa) and the number of correctly predicted characters by the best predictor in each generation (axis of ordinates). The plots include *averaged data* obtained from 50 experiments with the same initial parameters. The plots for the existing algorithm are shown by dots. The dashed lines correspond to the algorithm proposed in this paper (no operation of restoration of links between states is used here). For the case when this operation has been used, solid lines are depicted. Below, we present the results of three experiments differing in the chosen number of generations, bit mask, and number of flib states.

### *11.1. The First Experiment*

Here, the number of generations is 400, the number of flib states is 20, and the bit mask specifying the environment has the form

11110100101111101001.

The plots of the first experiment are shown in Fig. 2. It follows from these plots that the algorithm proposed in this paper for the formation of a new generation significantly enhances the accuracy of prediction as compared to the existing approach. The use of the algorithm of link restoration also enhances the efficiency of the proposed algorithm. The results of the first experiment are presented in Table 2.

### *11.2. The Second Experiment*

Here, the number of generations is 400, the number of flib states is 10, and the bit mask specifying the environment has the form

111101001011110.

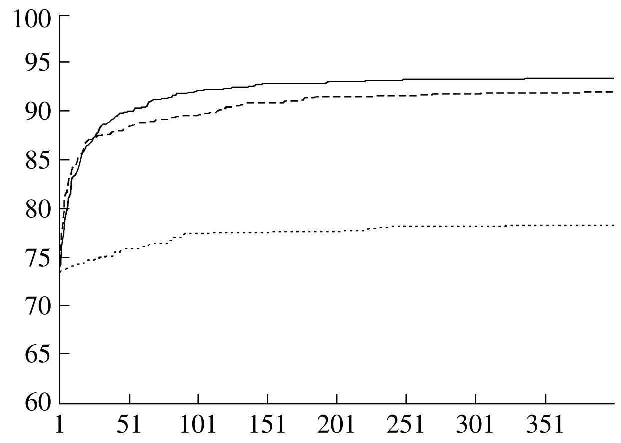


**Fig. 2.** Plots of averaged prediction data for the first experiment.

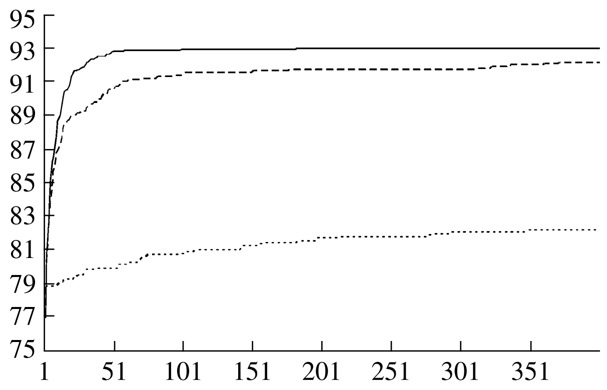


**Fig. 3.** Plots of averaged prediction data for the second experiment.

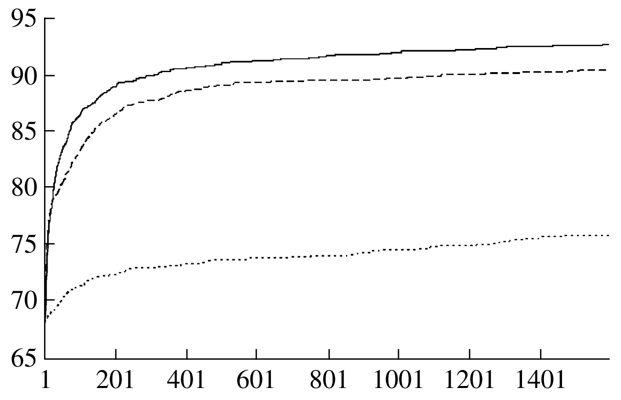


**Fig. 4.** Plots of averaged prediction data for the third experiment.

Let us note that the bit mask in this experiment is shorter than in the first experiment. The plots of the second experiment are shown in Fig. 3. The results of the second experiment are presented in Table 3. It follows from these results that the proposed algorithm is superior to the existing one in all parameters. The values given in the column “Averaged Data” correspond to the right-most points of lines in Fig. 3.

### *11.3. The Third Experiment*

Here, the number of generations is 1600, the number of flib states is 30, and the bit mask specifying the environment has the form

10101111011000111101111100110 01.

Let us note that the bit mask in this experiment is longer than in the first and second experiments. The plots of the third experiment are shown in Fig. 4. The results of the third experiment are presented in Table 4.

## CONCLUSIONS

The experiments have shown that the proposed method is more efficient than the existing one for both different bit masks specifying the environment and different numbers of flib states.

## APPENDIX

***Listing 1. Flib Implementation***.

```
namespace Flibs {
    public class Flib {
    private State [] _states;
    private int _curStateIndex = 0;
    private double _guessCount;
    public Flib (int stateCount) {
    _states = new State[stateCount];
    }
    public Flib(State [] states) {
    _states = states;
    }
    public void Step(int input, int output) {
    Branch branch = _states[_curStateIndex].Branches[input];
    if (branch.Output == output) _guessCount++;
    _curStateIndex = branch.StateIndex;
    }

    public double Fitness {
    get { return _guessCount; }
    }
    public void Nulling() {
    _guessCount = 0;
    _curStateIndex = 0;
    }
    public State[] States {
    get { return _states; }
    }
    public Flib Clone() {
```

```
Flib flib = new Flib(_states.Length);
flib._curStateIndex = _curStateIndex;
flib,_guessCount = _guessCount;
for (int i = 0; i < _states. Length; i++) {
flib.States [i] = _states[i].Clone();
}
return flib;
}
}
public class Branch {
public const int TARGET_COUNT = 2;
private int _stateIndex;
private int _output;
public Branch(int stateIndex, int output) {
_stateIndex = stateIndex;
_output = output;
}
public Branch Clone() {
return new Branch(_stateIndex, _output);
}
public int StateIndex {
get { return _stateIndex; }
}
public int Output {
get { return _output; }
}
}
public class State {
private Branch[] _branches;
public State () {
_branches = new Branch[Branch.TARGET_COUNT];
}
public State Clone() {
State state = new State();
state._branches = new Branch[_branches.Length];
for (int i = 0; i < _branches.Length; i++) {
state._branches[i] = _branches[i].Clone();
}
return state;
}
public Branch [ ] Branches {
get { return _branches; }
}
}
}
```

***Listing 2. Generator of Input Signal.***

```
namespace Flibs {
    public class SimpleSignalSource : ISignalSource {
    private int _state;
    private int[] _mask;
    public void DoStep() {
    _state++;
    }
    public SimpleSignalSource(int[] mask) {
    Nulling();
    _mask = mask;
    }
    public int Input {
    get { return _mask[_state%_mask.Length]; }
    }
    public int InputNext {
    get { return _mask[(_state + 1)%_mask.Length]; }
    }
    public void Nulling() {
    _state = -1; }
    }
    }
    }
```

***Listing 3. Crossover Operator.***

```
namespace Flibs {
    public class SimpleCrossover : ICrossover {
    public SimpleCrossover() {
    }
    public virtual Flib CreateChild(Random random, Flib firstParent, Flib secondParent) {
    Flib result = new Flib(firstParent.States.Length);
    int bound = random.Next(result.States.Length);
    for(int i = 0; i < result.States.Length; i++) {
    if(i < bound)
    result.States[i] = firstParent.States [i].Clone();
    else if(i > bound)
    result.States[i] = secondParent.States[i] .Clone (); else
    result.States[i] = CreateState(random, firstParent.States[i], firstParent.States[i]);
    }
    return result;
    }
    private State CreateState(Random random, State firstState, State secondState) {
    State result = new State();
    int bound = random.Next(result.Branches.Length);
    for(int i = 0; i < result.Branches.Length; i++) {
    if(i < bound)
    result.Branches[i] = firstState.Branches[i].Clone();
    else
    result .Branches [i] = secondState.Branches [i] .Clone ();
    }
    return result;
    }
    }
    }
```

***Listing 4. Mutation Operator.***

```
namespace Flibs {
```

```
public, class SimpleMutation : IMutation {
private double _mutationProbability;
public SimpleMutation(double mutationProbability) {
_mutationProbability = mutationProbability;
}
public void Mutate(Random random, Flib flib) {
if(random.NextDouble() > _mutationProbability) return;
MutateState(random, flib.States[random.Next(flib.States.Length)], flib.States.Length);
}
private void MutateState(Random random, State state, int. stateCount)
{
int branchIndex = random.Next(Branch.TARGET_COUNT); state.Branches[branchIndex] = Mutate-
Branch(random,
state.Branches[branchIndex], stateCount);
}
private Branch MutateBranch(Random random, Branch branch, int stateCount) {
if(random.NextDouble() < 0.5)
return new Branch(random.Next(stateCount),
branch.Output);
else
return new Branch(branch.StateIndex, random.Next(Branch.TARGET_COUNT));
}
}
}
```

***Listing 4. Restoring Links Between States.***

```
namespace Flibs {
public class FlibRestorer {
public void Restore(Random random, Flib flib) {
SortedList indexes = InitIndexesList(flib);
for (int i = 0; i < flib.States.Length; i++) {
if (!indexes.Contains(i)) {
int index = (int)
indexes.GetKey(random.Next(indexes.Count));
int branchNum =
random.Next(Branch.TARGET_COUNT);
flib.States[i].Branches[branchNum] = new
Branch(flib.States[index].Branches[branchNum].StateIndex,
flib.States[i].Branches[branchNum].Output);
flib.States[index].Branches[branchNum] = new Branch(i, flib.States[index].Branches[branchNum].Output);
AddIndex(indexes, i, flib);
}
}
}
private SortedList InitIndexesList(Flib flib) {
SortedList indexes = new SortedList();
int index = 0;
AddIndex(indexes, index, flib);
return indexes; }
private void AddIndex(SortedList indexes, int index, Flib flib) {
indexes[index] = b;
```

```
foreach (Branch branch in flib.States[index].Branches) {
if (!indexes.Contains(branch.StateIndex))

                    AddIndex(indexes, branch.StateIndex, flib);
```

## REFERENCES

1. A. A. Shalyto, "Technology of Automaton Programming," Mir PK, No. 10, 74–78 (2003).
2. M. Mitchell, *An Introduction to Genetic Algorithms* (MA: MIT Press, Cambridge, 1996).
3. W. Langdon and R. Poli, *Better Trained Ants for Genetic Programming* (University of Birmingham, Birmingham, 1998).
4. S. A. Wolfram, *A New Kind of Science* (Wolfram Media, Champaign, 2002).
5. M. Mitchell, J. Crutchfield, and P. Hraber, "Evolving Cellular Automata to Perform Computations," Phys. D. (Amsterdam) **75**, 361–391 (1993).
6. Yu. D. Bednyi, *Application of Genetic Algorithms for Solving a Problem on Cell Automata.* The Problem of Density Classification for Cellular Automata. Bachelor Thesis (SPbGU ITMO, St. Petersburg, 2006) [in Russian].
7. D. Whitley, "A Genetic Algorithm Tutorial," Statistics and Computing **4**, 65–85 (1994).
8. O. Voronin and A. D'yudni, "Darwinism in Programming," Moi Komp'yuter, No. 35 (2004).
9. B. Miller and M. Goldberg, "Genetic Algorithms, Tournament Selection, and the Effects of Noise," Complex Systems **3,** 193–212 (1995).
10. K. De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems. PhD Thesis* (Univ. of Michigan, Ann Arbor, 1975).