ИЗВЕСТИЯ РАН ТЕОРИЯ И СИСТЕМЫ УПРАВЛЕНИЯ, 2007, № 5, с 127–136

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

УДК 004 8

ИСПОЛЬЗОВАНИЕ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ ДЛЯ АВТОМАТИЧЕСКОГО ПОСТРОЕНИЯ КОНЕЧНЫХ АВТОМАТОВ В ЗАДАЧЕ О ФЛИБАХ

© 2007 г. П. Г. Лобанов, А. А. Шалыто

Санкт-Петербург, Санкт-Петербургский государственный ун-т информационных технологий механики и оптики

Поступила в редакцию 09.01.07 г., после доработки 28 05 07 г

Предложен новый генетический алгоритм для решения задачи о флибах, которые моделируют поведение простейшего живого существа в простейшей среде. Выполнены вычислительные эксперименты, демонстрирующие эффективность этого алгоритма по сравнению с известным.

Введение. Существуют сложные задачи, которые эффективно решаются с помощью конечных автоматов [1]. В большинстве случаев синтез автоматов в них выполняется эвристически. Поэтому весьма актуально решение проблемы формализации методов построения автоматов, а также автоматизации этих методов. Известен класс задач (например, итерированная дилемма узников [2], "умный" муравей [3], синхронизация [4] и классификация плотности [5] для клеточных автоматов), в которых генетические алгоритмы [6] способны автоматически строить автоматы, решающие эти задачи. В настоящей работе рассматривается еще один представитель этого класса – задача о флибах [7, 8].

1. Задача о флибах. Это моделирование простейшего живого существа, которое способно предсказывать имеющие периодичность изменения простейшей окружающей среды. Живое существо моделируется конечным автоматом, а генетические алгоритмы позволяют автоматически построить автомат, который прогнозирует изменения среды с достаточно высокой точностью. Таким образом, требуется сформировать "устройство" (предсказатель), которое оценивает будущие изменения среды с наибольшей вероятностью. В [8] для решения указанной проблемы используется один из генетических алгоритмов. При этом точность предсказания реализованного автомата является недостаточно высокой. Это во многом связано с методом, применяемым для построения следующего поколения особей (автоматов). Цель статьи – разработка подхода, устраняющего указанный недостаток. Создана программа на языке C#, которая реализует как предлагаемый, так и известный подходы, что позволяет их сравнивать.

2. Постановка задачи. Одна из важнейших способностей живых существ – предсказание изменений окружающей среды. В качестве простейшей модели живого существа можно использовать конечный автомат. В [8] такие конечноавтоматные модели названы *флибами* (сокращение от *finite living blobs* – конечные живые капельки).

На вход флиба подается переменная, которая принимает одно из двух значений – ноль или единица. Эта переменная характеризует состояние окружающей среды в текущий момент времени. Среда настолько проста, что имеет только два состояния. Флиб изменяет свое состояние и формирует значение выходной переменной. Это значение соответствует возможному состоянию среды в следующий момент времени. Задача флиба – предсказать, какое на самом деле состояние окружающей среды наступит в следующий момент времени. Это можно выполнить благодаря периодичности ее изменений. При этом считается, что чем точнее флиб предсказывает изменения среды, тем больше у него шансов выжить и оставить потомство.

3. Схемы работы генетических алгоритмов в задаче о флибах. Приведем схемы работы известного и предлагаемого алгоритмов.

3.1. Известный алгоритм. Для поиска оптимального предсказателя в [8] применяется следующий генетический алгоритм.

1. Создается поколение случайных флибов.

2. Подсчитывается, сколько изменений состояний окружающей среды правильно предсказывает каждый из этих флибов.

3. Определяются худший и лучший предсказатели в поколении.

4. Лучший предсказатель *скрещивается* с флибом, выбранным случайным образом.

5. Случайным образом определяется необходимость применения к полученному флибу *оператора мутации*. Если это подтверждается, то указанный оператор применяется к флибу.

6. Худший предсказатель в поколении заменяется флибом, полученным в результате скрещивания.

Таблица 1. Флиб с тремя состояниями

Состояния	0	1
A	1, B	0, A
B	0, C	0, A
C	1, A	0, B

После выполненной замены считается, что новое поколение сгенерировано.

7. Если один из флибов достиг уровня предсказания, равного 100%, или пользователь остановил программу, то алгоритм прекращает работу. В противном случае переходим к п. 2.

Алгоритм создания случайных флибов, а также работа операторов одноточечного скрещивания и мутации будут подробно рассмотрены в разд. 4, 7, 8.

3.2. Предлагаемый алгоритм. В настоящей работе для генерации нового поколения используется метод, отличающийся от описанного в разд. 3.1. При этом применяется турнирный отбор [9] и принцип элитизма (в новое поколение добавляется одна или несколько лучших особей из предыдущего поколения) [10]. Число флибов в поколении будем называть размером поколения.

Предлагаемый алгоритм имеет следующий вид.

1. Создается текущее поколение случайных флибов.

2. Подсчитывается, сколько изменений состояний окружающей среды правильно предскажет каждый из этих флибов.

3. Строится новое поколение флибов:

a) создается пустое новое поколение и в него добавляется лучший предсказатель из текущего поколения;

b) случайным образом из текущего поколения выделяются две пары флибов;

c) из каждой пары выбирается лучший предсказатель;

d) лучшие предсказатели из указанных пар *скрещиваются*;

e) случайным образом определяется целесообразность применения *оператора мутации* к полученному флибу. Если это необходимо, то указанный оператор применяется к флибу;

f) флиб подвергается новой операции – "Восстановление связей между состояниями автомата";

g) флиб добавляется в новое поколение;

h) переход к п. b, если размер нового поколения меньше размера текущего поколения.

4. Текущее поколение флибов заменяется новым.

5. Если число поколений меньше заданного пользователем, то возвращаемся к п. 2.

4. Реализация флиба. Опишем известный и предлагаемый методы реализации флиба.

4.1. Известный метод. Поведение флиба задается с помощью таблицы переходов и выходов. В табл. 1 в качестве примера приведен флиб с тремя состояниями: A, B и C. Представим эту таблицу в виде строки: `1B0A0C0A1A0B`, которая используется в [8]. Отметим, что число элементов в приведенной строке в 4 раза больше числа состояний флиба. Пронумеруем состояния флиба и элементы строки, задающей его. Первому состоянию и первому элементу строки присвоим номер ноль. Если флиб находится в состоянии с номером `s` и текущее состояний среды `i`, то состояние, в которое флиб перейдет, присутствует в элементе строки, задающей флиб. Этот элемент имеет номер `4s + 2i + 1`. Значение выходной переменной, генерируемой флибом, содержится в элементе с номером `4s + 2i`.

Для создания случайного флиба требуется задать значения элементов строки случайным образом. Приведем алгоритм создания случайного флиба, описанного строкой.

1. Цикл по всем элементам строки, задающей флиб:

a) если номер элемента четный, то элементу присваивается одно из возможных состояний среды, выбранное случайным образом;

b) в противном случае элементу присваивается одно из возможных состояний флиба, выбранное случайным образом.

4.2. Предлагаемый метод. В настоящей работе используется другой способ кодирования флиба, реализуемый с помощью трех классов – `Flib`, `State` и `Branch`, которые приведены в Приложении (листинг 1). В качестве главного класса при реализации флиба применяется класс `Flib`. Классы `State` и `Branch` реализуют его состояния и переходы соответственно. В каждом из этих классов имеется метод `Clone`, предназначенный для создания копий объектов.

Массив `_states` в классе `Flib` содержит состояния флиба. Поле `_curStateIndex` используется для хранения номера текущего состояния флиба в массиве `_states`. Число правильно предсказанных входных символов размещается в поле `_guessCount`. Метод `Step` переводит флиб в новое состояние и, при необходимости, изменяет количество правильно предсказанных символов. Метод `Nulling` возвращает флиб в начальное состояние и обнуляет число правильно предсказанных символов.

В массив `_branches` класса `State` включены дуги переходов из данного состояния. Номер элемента в массиве соответствует входной переменной. Переменные `_stateIndex` и `_output` класса `Branch` содержат номер состояния, в которое переходит флиб по этой дуге, и значение выходной переменной. Константа `TARGET_COUNT` определяет ко-

личества значений, которые может принимать выходная переменная, генерируемая флибом.

Опишем работу алгоритма создания случайного флиба.

1. Создаются объекты, соответствующие состояниям флиба.

2. Применительно к каждому объекту выполняется следующее:

a) для состояния формируются переходы из него;

b) для каждого перехода случайным образом определяются номер будущего состояния флиба и значение выходной переменной.

5. Генератор входного сигнала. В качестве входного сигнала для флибов (как и в [8]) используется повторяющаяся последовательность битов (битовая маска), задающая изменения состояний среды. Эта маска в программе зациклена. Код класса для генерации входного сигнала приведен в Приложении (листинг 2).

Массив символов, соответствующий битовой строке, передается в конструктор класса `SimpleSignalSource`. Свойства `Input` и `InputNext` возвращают входной символ для текущего и следующего моментов времени соответственно. Для перехода к следующему символу применяется метод `DoStep`, который необходимо вызвать до начала генерации входного сигнала. Метод `Nulling` сбрасывает генератор входного сигнала в начальное состояние.

6. Оценочная функция. Лучшим является тот флиб, который наиболее точно предсказывает входной сигнал. В качестве оценочной функции выступает количество правильно угаданных символов (поле `_guessCount` в классе `Flib`). После формирования нового поколения все флибы в нем устанавливаются в начальное состояние с помощью метода `Nulling`. Затем для определения приспособленности флибов каждому из них подается на вход несколько символов (по умолчанию их количество равно 100). После этого можно строить новое поколение решений, используя в качестве приспособленности флиба значение поля `_guessCount`, содержащее число правильно предсказанных символов.

7. Алгоритмы работы оператора одноточечного скрещивания. Опишем известный и предлагаемый алгоритмы работы оператора одноточечного скрещивания.

7.1. Известный алгоритм. В [8] формирование нового флиба из двух родительских выполняется с помощью оператора одноточечного скрещивания. Приведем описание алгоритма работы этого оператора для флиба, закодированного строкой длины m.

1. Выбирается случайное число j в интервале от 0 до $m - 1$.

2. Элементы с номерами, меньшими либо равными j, из строки, которая задает первый родительский флиб, копируются в строку, описывающую новый флиб.

3. Элементы с номерами, большими j, из строки, которая задает второй родительский флиб, копируются в строку, соответствующую новому флибу.

7.2. Предлагаемый алгоритм. Для применения оператора одноточечного скрещивания для флиба, реализованного в виде объекта класса `Flib`, алгоритм из [8] требуется модифицировать. При этом предлагаемый алгоритм имеет следующий вид.

1. Случайным образом выбирается номер одного из состояний нового флиба.

2. В этот флиб добавляются состояния из первого родителя, номера которых меньше выбранного номера, и состояния из второго родителя, номера которых больше выбранного номера.

3. Формируется и добавляется новое состояние, образованное из состояний первого и второго родителей, которые соответствуют выбранному номеру. Алгоритм формирования состояния аналогичен алгоритму работы оператора одноточечного скрещивания, который описан в разд. 7.1.

Реализация изложенного алгоритма приведена в Приложении (листинг 3).

8. Алгоритмы работы оператора мутации. Рассмотрим известный и предлагаемый алгоритмы работы оператора мутации.

8.1. Известный алгоритм. Алгоритм работы оператора мутации, используемый в [8], имеет следующий вид.

1. Случайным образом выбирается элемент в строке, задающей флиб.

2. Если номер элемента четный (в элементе содержится значение выходной переменной, генерируемой флибом), то значение переменной инвертируется.

3. Если номер элемента нечетный (в элементе содержится состояние флиба), то текущее состояние флиба заменяется следующим.

8.2. Предлагаемый алгоритм. В настоящей работе формируется следующий алгоритм работы оператора мутации.

1. Случайным образом устанавливается состояние флиба.

2. Случайным образом выбирается дуга из этого состояния.

3. Случайным образом определяется, что будет изменяться – значение выходной переменной, генерируемой флибом, или номер состояния, в которое он попадет при переходе по выбранной дуге:

a) если подтверждается, что изменяется значение выходной переменной, то ей присваивается зна-

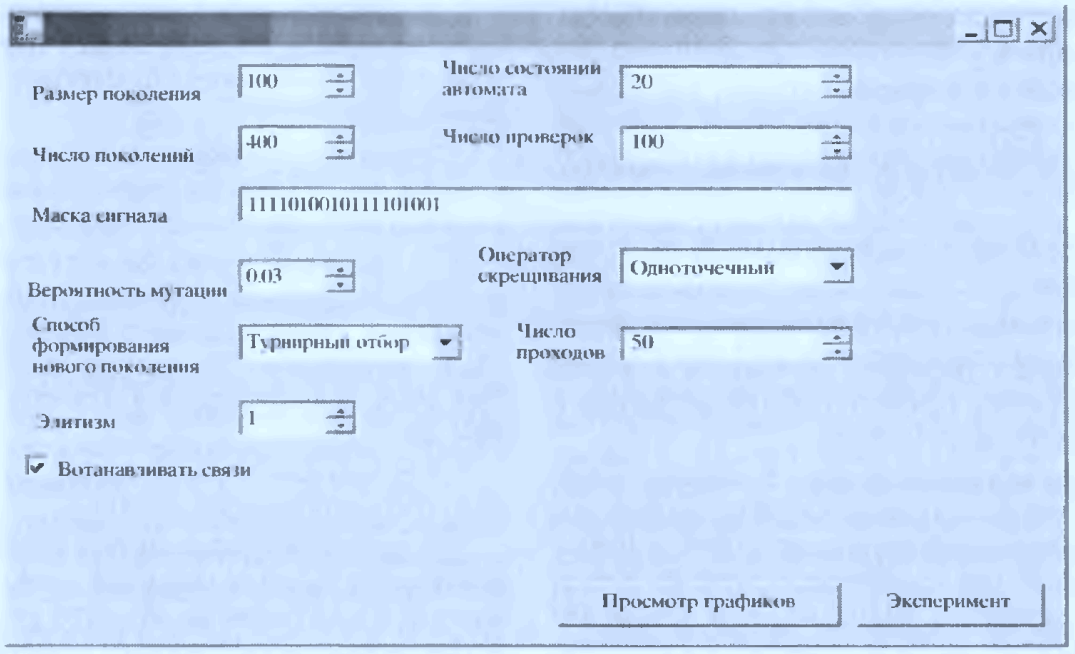

Рис. 1. Окно пользовательского интерфейса.

чение состояния среды, выбранное случайным образом;

b) если было определено, что корректируется номер состояния, то номеру состояния, в которое переходит флиб, присваивается номер случайно выбранного состояния флиба.

Класс, реализующий данный оператор мутации, приведен в Приложении (листинг 4).

9. Восстановление связей между состояниями. При генерации нового поколения за счет применения операторов скрещивания и мутации дуги переходов в флибах изменяются случайным образом: могут возникать состояния, в которые невозможно попасть из начального при любой последовательности изменений состояний среды. Будем называть их *недоступными*. Состояния, достигаемые из начального состояния при некоторой последовательности изменений состояний среды, являются *доступными*. Алгоритм восстановления связей между состояниями изменяет дуги переходов во флибе таким образом, чтобы в нем не было недоступных состояний.

Класс `FlibRestorer`, реализующий алгоритм восстановления связей между состояниями, приведен в Приложении (листинг 5). Предлагаемый алгоритм работает следующим образом.

1. Формируется список *доступных состояний* (метод `InitIndexesList`). Для этого используется функция рекурсивного обхода состояний `AddIndex`.

2. Выполняется полный цикл. Если текущее состояние не входит в число *доступных*, то для него задействованы следующие операции:

a) случайным образом выбирается состояние из списка *доступных состояний*;

b) также случайным образом определяется одна дуга из выбранного состояния;

c) в текущем состоянии заменяется одна дуга из этого состояния на дугу, которая ведет в то же состояние, что и дуга, полученная в п. b);

d) выбранная в п. b) дуга заменяется дугой, которая ведет в текущее состояние;

e) обновляется список *доступных состояний*. В него добавляется текущее состояние и все состояния, в которые можно попасть из него.

10. Программа для проведения экспериментов на флибах. По рассмотренным алгоритмам (известному и предлагаемому) написана программа, окно пользовательского интерфейса которой приведено на рис. 1. Программа позволяет реализовать эти алгоритмы с помощью выпадающего списка "Способ формирования нового поколения". При этом в предлагаемом алгоритме может быть изменен элитизм в диапазоне от нуля до размера поколения. Программа допускает выбор размера поколения и числа поколений. Также может быть установлена вероятность мутации (в диапазоне от нуля до единицы) и тип оператора скрещивания (одноточечный и двухточечный). Программа позволяет формировать битовую маску, описывающую среду. Для каждого флиба может быть задано число его состояний и определено, будет ли использоваться алгоритм восстановления связей между состояниями.

Для того чтобы сравнивать эффективность работы генетических алгоритмов, принято многократно прогонять их на одинаковых наборах тестовых данных и сопоставлять усредненные результа-

ты. Поэтому в программе реализована возможность автоматического прогона алгоритма заданное число раз и получения усредненных результатов. Алгоритм может быть запущен на число проходов, которое экспериментатор хочет установить. Приведенные в Приложении листинги составляют ядро разработанной программы для предлагаемого подхода.

11. Общие требования к экспериментам. Все эксперименты проводились при размере поколения 100 и вероятности применения оператора мутации 0.03. Число воздействий среды на флиб выбрано равным 100. Поэтому количество правильно предсказанных символов по величине равно точности предсказания символов в процентах. В табл. 2–4 приведены результаты экспериментов и указаны минимальные, максимальные и средние точности предсказания автоматически построенных флибов.

На графиках (рис. 2–4) по оси абсцисс отложены номера поколений, а по оси ординат – число правильно предсказанных символов лучшим предсказателем в каждом поколении. При построении графиков использовались **усредненные данные**, полученные при проведении 50 экспериментов с одинаковыми начальными параметрами. Графики для известного алгоритма изображены точками. Пунктиром выделены графики, соответствующие алгоритму, предложенному в работе, в котором не применяется операция восстановления связей между состояниями. Для случая, когда эта операция присутствует, графики построены непрерывными линиями. Ниже приводятся результаты трех экспериментов, которые отличаются между собой выбранным числом поколений, битовой маской и числом состояний флиба.

11.1. Первый эксперимент. Эксперимент производился для 400 поколений. Число состояний флиба равно 20. Битовая маска, задающая среду, имеет вид

Таблица 2. Результаты первого эксперимента

Алгоритм	Восстановление связей между состояниями	Результат		
		худший	усредненный	лучший
Известный	–	70	78.3	88
Предложенный	–	83	92.26	100
	+	84	93.48	100

Таблица 3. Результаты второго эксперимента

Алгоритм	Восстановление связей между состояниями	Результат		
		худший	усредненный	лучший
Известный	–	71	82.2	93
Предложенный	–	86	92.2	94
	+	87	93.06	94

Таблица 4. Результаты третьего эксперимента

Алгоритм	Восстановление связей между состояниями	Результат		
		худший	усредненный	лучший
Известный	–	70	75.86	87
Предложенный	–	83	90.44	97
	+	86	92.72	97

```
11110100101111101001.
```

Графики для первого эксперимента приведены на рис. 2. Из них следует, что предложенный в работе метод генерации нового поколения позволяет значительно повысить точность предсказания по сравнению с известным подходом. Использование алго-

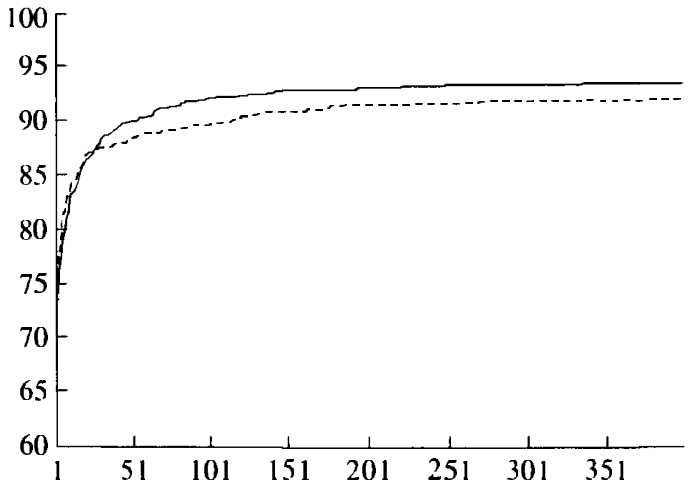

Рис. 2. Графики усредненных результатов предсказаний для первого эксперимента.

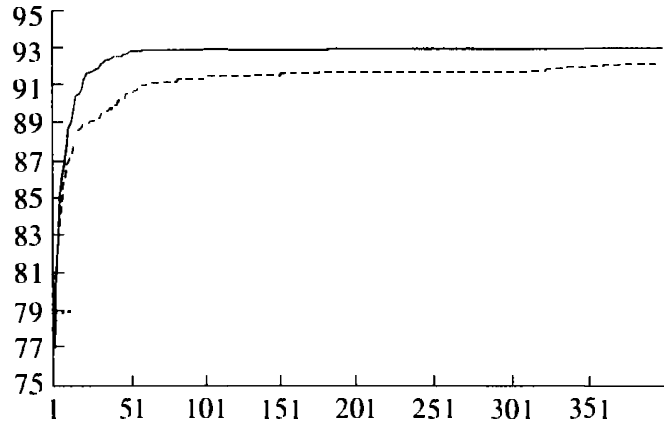

Рис. 3. Графики усредненных результатов предсказаний для второго эксперимента

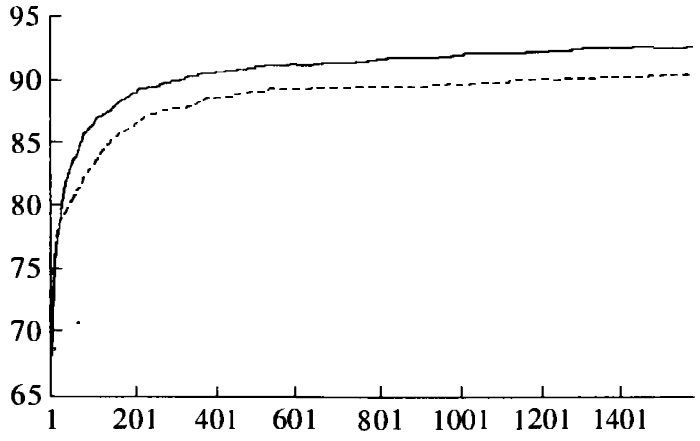

Рис. 4. Графики усредненных результатов предсказаний для третьего эксперимента.

ритма восстановления связей между состояниями также увеличивает эффективность предлагаемого алгоритма. Результаты первого эксперимента представлены в табл. 2.

11.2. Второй эксперимент. Эксперимент производился для 400 поколений. Число состояний флиба равно 10. Битовую маску, задающую среду, запишем как

```
111101001011110.
```

Отметим, что битовая маска в этом эксперименте короче, чем в предыдущем. Графики для второго эксперимента приведены на рис. 3. Результаты второго эксперимента отражает табл. 3. Из ее рассмотрения следует, что предложенный алгоритм превосходит известный по всем параметрам. Значения, представленные в столбце "Усредненный результат", соответствуют крайним правым точкам графиков на рис. 3.

11.3. Третий эксперимент. Эксперимент производился для 1600 поколений. Число состояний флиба равно 30. Битовая маска, задающая среду, имеет вид

```
10101111011000111101111110011001.
```

Отметим, что битовая маска в этом эксперименте имеет большее число разрядов, чем в двух предыдущих. Графики для третьего эксперимента приведены на рис. 4. Результаты третьего эксперимента сведены в табл. 4.

Заключение. Проведенные эксперименты показали, что предложенный метод эффективнее известного как при различных битовых масках, задающих среду, так и различном числе состояний флиба.

ПРИЛОЖЕНИЕ

Листинг 1. Реализация флиба

```
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; }
    }
  }
}
```

Листинг 2. Генератор входного сигнала

```
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;
    }
  }
}
```

Листинг 3. Оператор скрещивания

```
namespace Flibs {
  public class SimpleCrossover : ICrossover {
    public SimpleCrossover() {
    }
    public virtual Flib CreateChild(Random random, Flib firstParent, Flib second-
Parent) {
      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], firstPar-
ent.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;
    }
  }
}
```

Листинг 4. Оператор мутации

```
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] = MutateBranch(random, state.Branches[branchIn-
dex], 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));
    }
  }
}
```

Листинг 5. Восстановление связей между состояниями

```
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[in-
dex].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] = 0;
      foreach (Branch branch in flib.States[index].Branches) {
        if (!indexes.Contains(branch.StateIndex))
          AddIndex(indexes, branch.StateIndex, flib);
      }
    }
} }
```

СПИСОК ЛИТЕРАТУРЫ

1. *Шалыто А А* Технология автоматного программирования // Мир ПК. 2003. № 10. С. 74–78.
2. *Mitchell M* An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press, 1996.
3. *Langdon W , Poli R* Better Trained Ants for Genetic Programming. Birmingham: University of Birmingham, 1998.
4. *Wolfram S* A New Kind of Science. Champaign. Wolfram Media, 2002.
5. *Mitchell M , Crutchfield J , Hraber P* Evolving cellular automata to perform computations // Physica D. 1993. V. 75. P. 361–391.
6. *Whitley D* A Genetic Algorithm Tutorial // Statistics and Computing. 1994. V. 4. P. 65–85.
7. *Фогель Л , Оуэнс А , Уолш М.* Искусственный интелект и эволюционное моделирование. М.: Мир, 1969.
8. *Воронин О , Дьюдни А* Дарвинизм в программировании // Мой компьютер. 2004. № 35.
9. *Miller B , Goldberg M.* Genetic algorithms, tournament selection, and the effects of noise // Complex Systems. 1995. V. 3. P. 193–212.
10. *De Jong K* An analysis of the behavior of a class of genetic adaptive systems. PhD thesis. Ann Arbor: Univ. Michigan, 1975.