ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ ДЛЯ РЕАЛИЗАЦИИ СИСТЕМ СО СЛОЖНЫМ ПОВЕДЕНИЕМ

Н.И. Поликарпова, В.Н. Точилин
Научный руководитель – д.т.н., профессор А.А. Шалыто

Данная работа рассматривает применение методов генетического программирования для построения систем со сложным поведением. Предложено представление таких систем в форме хромосомы. Адаптированы и усовершенствованы операторы генетического программирования. Предложено решение проблемы экспоненциального роста объема хромосомы и размерности пространства поиска с увеличением количества параллельных входов автомата. Рассмотрен пример.

Введение

Программные и аппаратные системы, а также их отдельные элементы часто имеют *сложное поведение*. Таким свойством обладает большинство реактивных систем, устройства управления, сетевые протоколы, диалоговые окна, персонажи компьютерных игр и многие другие объекты и системы.

В последнее время для описания сущностей со сложным поведением предлагается использовать *автоматный подход* [1, 2]. В соответствии с этим подходом сущность представляется в виде *автоматизированного объекта* – совокупности *управляющего автомата* и *объекта управления*. Объект управления характеризуется множеством *вычислительных состояний*, а также двумя наборами функций: множеством *предикатов*, отображающих вычислительное состояние в логическое значение (истина или ложь), и множеством *действий*, позволяющих изменять вычислительное состояние. Управляющий автомат определяется конечным множеством *управляющих состояний*, *функцией переходов* и *функцией действий*. При этом объект управления может быть легко реализован традиционными методами (как программно, так и аппаратно), так как его поведение уже не является сложным: предикаты и действия представляют собой простые, короткие функции, практически не содержащие ветвлений. Вся «логика» оказывается сосредоточенной в управляющем автомате.

Описание логики в форме диаграммы переходов конечного автомата хорошо структурировано, разработан ряд подходов к его декомпозиции [1, 3]. Несмотря на это, опыт показывает, что построение управляющего автомата для сложной сущности или системы обычно требует от разработчика гораздо больше усилий и порождает больше ошибок, чем реализация объекта управления. Эту проблему авторы настоящей работы предлагают решать методами *генетического программирования* [4].

Основная идея генетического программирования состоит в построении программ путем применения генетических алгоритмов к некоторой *модели вычисления*. При этом разработчику программы остается лишь задать *оценочную функцию*, определяющую для каждого возможного результата вычисления в выбранной модели численное значение, называемое его *пригодностью* (*fitness*). Таким образом, развитие генетического программирования – важный шаг в направлении повышения уровня абстракции языков программирования.

В качестве моделей вычисления в генетическом программировании чаще всего используют деревья, графы, команды процессора – «низкоуровневые» модели, имеющие ограниченный набор элементарных операций (таких как, например, запись и чтение ячеек памяти, арифметические операции, вызовы подпрограмм и т.д.). Достоинство низкоуровневых моделей состоит в их универсальности: с их помощью можно построить любую программу целиком, единообразно, вне зависимости от специфики решаемой задачи. Однако, такие модели обладают и серьезными недостатками. Во-первых, построенная программа из-за отсутствия высокоуровневой структуры редко бывает понятна человеку, что исключает возможность ее дальнейшей модификации вручную,

обобщения полученного решения и т.п. Во-вторых, из-за того, что пространство допустимых программ в этом случае очень велико, генетическая оптимизация может потребовать длительного времени.

В настоящей работе предлагается объединить автоматный и генетический подходы. При этом в качестве оптимизируемой модели вычисления выступает автоматизированный объект. Реализацию объекта управления предлагается производить вручную, а после этого применять генетический алгоритм для автоматического построения управляющего автомата. Автомат – «высокоуровневый» вычислитель: его элементарные операции – предикаты и действия, специфичные для конкретной задачи. Именно поэтому предлагаемый подход лишен вышеупомянутых недостатков, характерных для низкоуровневого генетического программирования. Здесь машине поручается именно та часть работы по написанию программ, которая хуже всего получается у человека.

1. Обзор смежных работ

Эволюционной оптимизации моделей вычислений в виде конечных автоматов были посвящены многие исследования в различных направлениях *эволюционных вычислений*. Классификация по направлениям, приведенная на Рис. 1, сделает их рассмотрение более удобным.

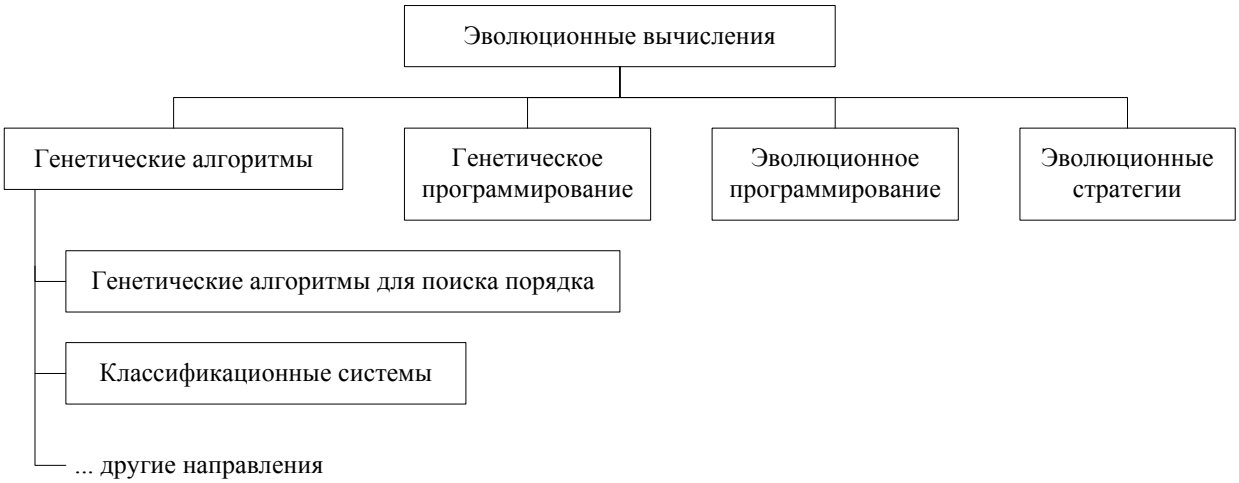

Рис. 1. Направления эволюционных вычислений

Генетические алгоритмы рассматривают методы функциональной оптимизации, основанные на модели естественной эволюции.

Классификационные системы работают преимущественно с пространством порождающих правил, а также конечных автоматов-*распознавателей*, определяющих принадлежность строки некоторому языку. Распознаватель не производит выходных воздействий, результат определяется состоянием автомата после обработки входной последовательности. В данном направлении как наиболее значимые можно выделить работы [5–12]. Более сложная форма конечного автомата – *преобразователь* (*transducer*) – отображает множество входных строк на множество выходных, возможно над другим алфавитом. Эволюционному построению преобразователей посвящена работа [13].

Генетическое программирование было впервые упомянуто в книге [4]. Первоначально предлагалось представлять программу в виде дерева. В работе [14] рассматривалось представление в виде машинного кода, а в статье [15] впервые оптимизировалась модель в форме графа, который можно интерпретировать как диаграмму переходов конечного автомата. Модель оказалась достаточно удачной, и впоследствии использовалась практически без изменений в ряде работ [16–19]. В статьях [20–22] рассматривается совместное использование различных моделей вычислений, включая гра-

фы или их модификации. В работах [23, 24] в качестве модели вычисления используются клеточные автоматы. В работе [25] применяются ациклические графы для формирования эффективно распараллеливаемых программ. В статьях [26–28] рассматривается автоматическое построение компонент логических контроллеров в виде автоматов, а в работе [29] оценивается эффективность автоматных моделей применительно к различным задачам.

В работе [30] отмечается необходимость расширения генетического программирования для работы со сложными структурами данных и описываются достижения в данном направлении. Эта тема развивается в работах [15, 31, 32], где рассмотрено использование различных заданных структур данных. Отметим, что подход, предлагаемый в настоящей работе, решает эту задачу полностью, позволяя использовать в составе автоматизированного объекта любой объект управления, включая произвольные стандартные структуры данных и их комбинации.

В разделе *эволюционное программирование* сначала изучались методы создания искусственного интеллекта в форме эволюционирующей популяции автоматов, а впоследствии основной его задачей стала оптимизация числовых параметров. Характерной особенностью данного направления можно назвать отсутствие скрещивания, при этом оптимизация осуществляется только за счет мутации. В работах этого направления [33–42] автоматы обучаются предсказывать следующий символ входной последовательности, классифицировать последовательности и копировать поведение системы, заданной частичным набором входных и соответствующих им выходных воздействий. Задачи управления рассматриваются как обучение автомата повторению поведения системы и его анализу с целью выявления входных воздействий, приводящих к требуемому результату.

В работе [43], а также в ряде работ, не относящихся напрямую к эволюционному программированию, [44–46], автоматы находят применение в играх. Авторы работы [47] рассматривают адаптивные методы эволюционного программирования. В статьях [48, 49] воссоздание неизвестных систем в виде автоматов применяется для изучения поведения агентов с целью взаимодействия с ними и эффективного тестирования последовательных электрических схем, соответственно.

Эволюционные стратегии позволяют эффективно оптимизировать параметры системы. В рамках этого направления часто рассматриваются аппаратные системы, для которых характерна экспериментальная проверка пригодности (в процессе реальной работы или ее программной эмуляции), что делает количество таких проверок решающим для производительности.

Практически во всех рассмотренных исследованиях автомат в каждый момент времени обрабатывает только одну входную переменную. Исключения составляют лишь работы [45] (четыре параллельных троичных входа) и [29] (в качестве условия перехода допускается сравнение значений двух регистров). Теоретически, любое количество параллельных входов сводится к одному, в качестве воздействий которого выступают комбинации сигналов исходных параллельных входов. Однако размер алфавита полученного таким образом входа растет экспоненциально с увеличением количества исходных параллельных входов. В обоих упомянутых работах параллельные входы не приводят к недопустимо большому алфавиту, но для реальных систем эта проблема крайне актуальна.

Также в рассмотренных работах автомат на каждом шаге может выполнить не более одного действия из заданного множества. В таком случае любая комбинация действий, которые может потребоваться выполнить одновременно, также должна считаться элементарным действием. При этом требуется априорная информация обо всех возможных комбинациях действий, либо задание вместе с элементарными действиями всех их наборов, что приводит к экспоненциальному росту количества действий. Отме-

тим, что в работе [29] допускаются действия с аргументами, а также параллельно выполняемые автоматы, отвечающие за различные действия, что значительно ослабляет проблему.

Из всех перечисленных работ наилучшие результаты в аспекте автоматического построения части программы на высоком уровне абстракции получены в статьях [27, 28] применительно к созданию управляющей программы робота. Однако и эти работы не лишены упомянутых выше недостатков, относящихся к входным и выходным воздействиям. Насколько известно авторам, исследования в области эволюционного построения систем со сложным поведением, отличных от роботов, ранее не проводились.

2. Постановка задачи

Цель настоящей работы – проверить возможность эффективного применения генетических алгоритмов для построения логики автоматизированных объектов., для которых характерно наличие параллельных входов и выходов.

Сформулируем *задачу построения управляющего автомата* более формально. Пусть задан объект управления $O = \langle V, v_0, X, Z \rangle$, где V – множество вычислительных состояний (или *значений*), v_0 – *начальное значение*, $X = \{x_i : V \to \{0,1\}\}_{i=1}^{n}$ – множество предикатов, $Z = \{z_i : V \to V\}_{i=1}^{m}$ – множество действий. Также задана оценочная функция $\varphi : V \to \mathbf{R}^+$.

Объект O может управляться автоматом вида $A = \langle S, s_0, \Delta \rangle$, где S – конечное множество управляющих состояний, s_0 – *стартовое состояние*, $\Delta : S \times \{0,1\}^n \to S \times Z^*$ – *управляющая функция*. Управляющую функцию можно разложить на две компоненты: функцию действий $\zeta : S \times \{0,1\}^n \to Z^*$ и функцию переходов $\delta : S \times \{0,1\}^n \to S$.

Пусть объекту управления соответствует значение v, а управляющий автомат находится в состоянии s. В течении одного *шага* работы автоматизированного объекта автомат переходит в новое состояние $s_{new} = \delta(s, x_1(v), \ldots, x_n(v))$, а объект управления изменяет свое значение на $v_{new} = z^l(z^{l-1}(\ldots(z^1(v))\ldots))$, где $(z^1, z^2, \ldots z^l) = \zeta(s, x_1(v), \ldots, x_n(v))$.

Задача построения управляющего автомата состоит в том, чтобы найти автомат заданного вида такой, что за k шагов работы под управлением этого автомата объект O перейдет в вычислительное состояние с максимальной пригодностью ($\varphi(v) \to \max$).

Отметим, что здесь для простоты используется частный случай модели автоматизированного объекта.

В настоящей работе для решения поставленной задачи предлагается использовать генетическое программирование. В связи с этим возникают следующие задачи: выбор представления конечного автомата в виде *особи*; адаптация генетических операторов (*мутации* и *скрещивания*) для выбранного представления; выбор параметров генетической оптимизации, подходящих для автоматов.

3. Способы представления автомата

В классической интерпретации генетического алгоритма особь представляется в виде набора *хромосом*, каждая из которых записана как битовая строка. При этом генетические операторы определяются универсальным образом в терминах битовых строк. Управляющий автомат легко представить как набор состояний, в каждом из которых его поведение определяется сужением управляющей функции $\Delta_s : \{0,1\}^n \to S \times Z^*$,

$s \in S$. Таким образом, удобно сопоставить каждому состоянию хромосому. Однако, низкоуровневая запись хромосомы в виде битовой строки не очень удобна.

В генетическом программировании распространен другой подход: для каждого класса оптимизируемых моделей вычисления выбирается свое представление, обладающее высокоуровневой структурой. Определение генетических операторов в терминах этой структуры с учетом семантики хромосомы позволяет значительно ускорить процесс эволюции.

В данном разделе авторы предлагают два варианта представления состояния в виде хромосомы.

3.1. Представление состояний: полные таблицы

Естественный способ записи хромосомы состояния – это табличное представление функции Δ_s. Таблица содержит 2^n строк (по одной для каждой возможной комбинации значений предикатов) и $m+1$ столбцов, в первом из которых записано значение функции переходов (номер целевого состояния), а совокупность остальных столбцов обозначает множество действий, которые необходимо выполнить на переходе. Пример полной таблицы одного состояния приведен на Рис. 2 (контуром обведена информативная часть таблицы, именно она и заносится в хромосому). Отметим, что в каждой строке таблицы записано множество действий, а не их последовательность, как это было определено в модели (разд. 2). Задание значения функции действий в виде множества значительно упрощает оператор скрещивания и повышает эффективность процесса эволюции. Выполнение на переходе последовательности действий эквивалентно осуществлению нескольких переходов, на которых выполняются множества действий. Таким образом, автомат исходной модели всегда может быть записан в предложенной выше табличной форме, возможно с добавлением нескольких состояний и переходов. После получения результата оптимизации лишние элементы автомата можно устранить, преобразовав множества действий в последовательности.

x_0	x_1	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

Рис. 2. Хромосома состояния: полная таблица (n = 2, m = 3, $|S|$ = 3)

Все таблицы, соответствующие состояниям одного автомата, имеют одинаковую размерность, так как число предикатов и действий объекта управления задано по условию задачи. Что касается управляющих состояний, наиболее эффективным является постепенное наращивание их числа в процессе оптимизации.

Опишем теперь генетические операторы над хромосомами состояний, записанными предложенным выше способом (в виде полных таблиц).

Алгоритм 1. Мутация полных таблиц. Алгоритм мутации состояния, представленного полной таблицей, описан на псевдокоде в листинге 1 и проиллюстрирован на Рис. 3 (здесь и далее на стрелках, обозначающих изменения значений в ячейках таблицы, написаны вероятности этих изменений). При мутации состояния с некоторой вероятностью может мутировать каждый элемент таблицы. При этом номер целевого состояния изменяется на любой из допустимых, а вместо исходного набора действий ге-

нерируется новый набор, вероятность появления единиц в котором равна доле единиц в исходном наборе.

Листинг 1. Мутация полных таблиц

```
State Mutate(State state)
{
	State mutant = state;
	for (для всех i: строк таблицы)
	{
		if (с вероятностью p1) {
			mutant[i].targetState = случайное число от 0 до nStates-1;
		}
		if (с вероятностью p2) {
			int nActsPresent = количество единиц в mutant[i].output;
			if ((nActsPresent == 0) || (nActsPresent == nActions)) {
				Index j = случайное число от 0 до nActions - 1;
				mutant[i].output[j] = !mutant[i].output[j];
			} else {
				for (для всех j: номеров действий) {
					mutant[i].output[j] = 1 с вероятностью
						nActsPresent/nActions и 0 иначе;
				}
			}
		}
	}
	return mutant;
}
```

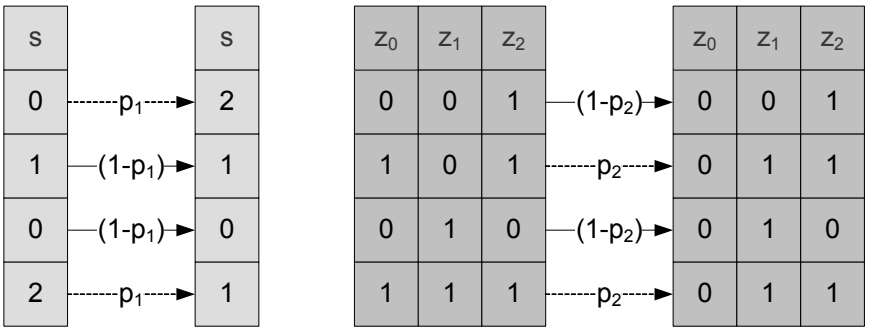

Рис. 3. Пример мутации полных таблиц

Алгоритм 2. Скрещивание полных таблиц. В настоящей работе рассматривается адаптация к предложенному представлению состояний одного способа скрещивания, известного как *одноточечное*. Руководствуясь схожими идеями, нетрудно адаптировать к табличному представлению и другие способы скрещивания. Алгоритм одноточечного скрещивания полных таблиц представлен в листинге 2 и проиллюстрирован на Рис. 4.

Листинг 2. Скрещивание полных таблиц

```
pair<State, State> Cross(State state1, State state2)
{
	State child1 = state1;
	State child2 = child1;
	int tableSize = размер таблицы;

	for (для всех j: столбцов таблицы)
	{
		int crossPoint = случайное число от 0 до tableSize;
		for (для всех i: строк таблицы от 0 до crossPoint - 1) {
			child1[i][j] = state1[i][j];
			child2[i][j] = state2[i][j];
```

```
            }
            for (для всех i: строк таблицы от crossPoint до tableSize - 1)
            {
                  child1[i][j] = state2[i][j];
                  child2[i][j] = state1[i][j];
            }
      }
      return make_pair(child1, child2);
}
```

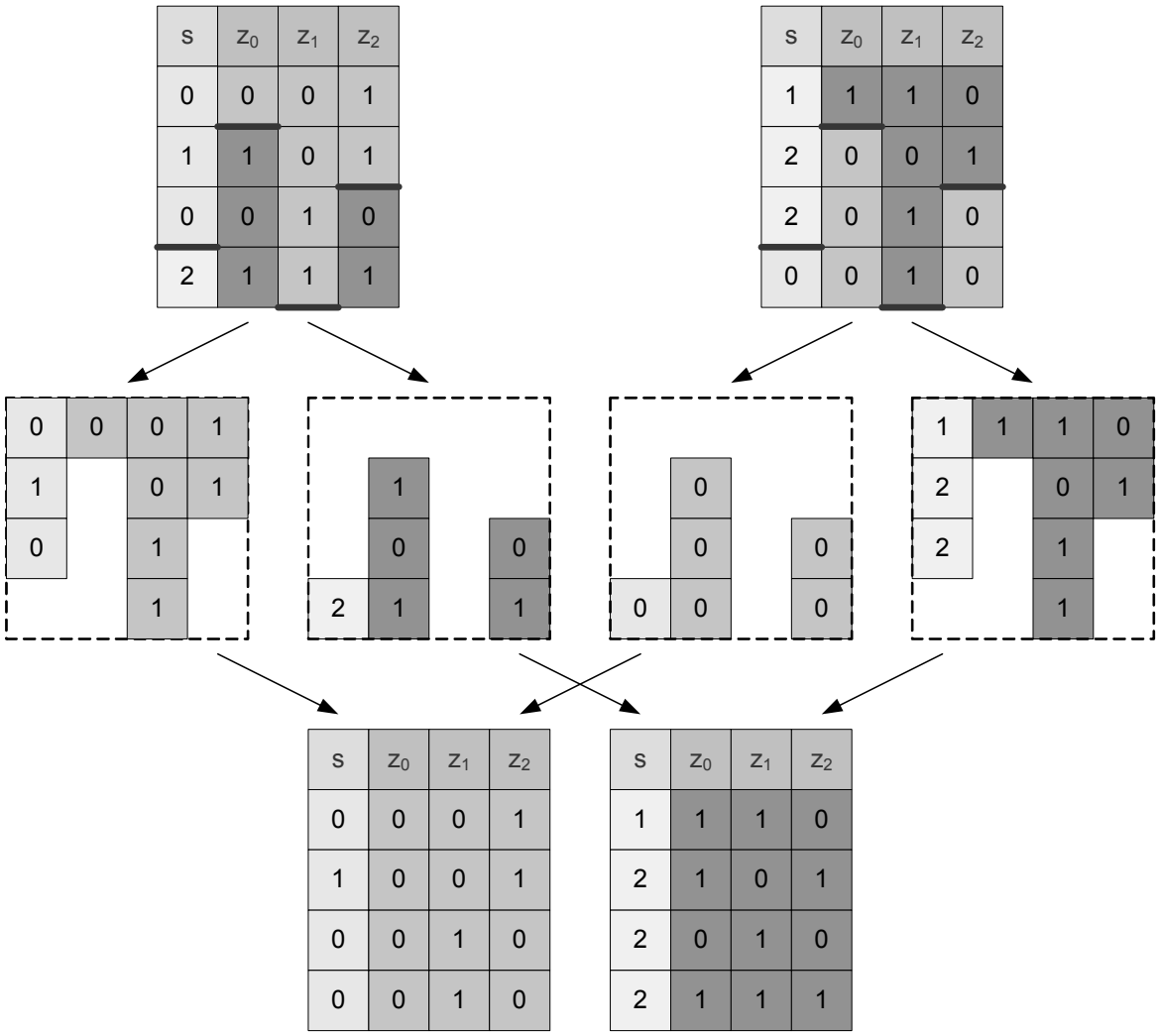

Рис. 4. Пример скрещивания полных таблиц

Основная проблема, возникающая при использовании полных таблиц рассмотренного вида – это экспоненциальный рост размерности хромосомы с увеличением числа предикатов объекта управления (напомним, что количество строк в таблице 2^n, где n – число предикатов). Опыт показывает, что в реальных задачах управляющие автоматы, построенные вручную, имеют гораздо меньше переходов, чем $|S| \cdot 2^n$. Причина, по мнению авторов, в том, что в большинстве задач предикаты имеют «локальную природу» по отношению к управляющим состояниям. В каждом состоянии *значимым* является лишь определенный, небольшой поднабор предикатов, остальные же не влияют на значение управляющей функции. Именно это свойство позволяет существенно сократить размер описания состояний. Кроме того, навязывание этого свойства в процессе оптимизации позволяет получить результат, более похожий на автомат, построенный вручную, а значит, более понятный человеку.

3.2. Представление состояний: сокращенные таблицы

Свойство локальности предикатов можно использовать для сокращения описания управляющего состояния разными способами. Авторами выбран один из подходов, при котором количество значимых в состоянии предикатов ограничивается некоторой константой r. К таблице, задающей сужение управляющей функции на данное состояние, в этом случае добавляется битовый вектор, описывающий множество значимых предикатов (Рис. 5).

x_0	x_1	x_2	x_3	x_4	x_5
0	1	0	1	0	0

x_1	x_3	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

Рис. 5. Хромосома состояния: сокращенная таблица (n=6, m=3, r=2, $|S|$=3)

Число строк таблицы в этом случае 2^r, однако, константа r обычно невелика. Ее выбор зависит от сложности задачи. Как показывает опыт, для большинства автоматизированных объектов среднее по всем состояниям значение r не больше пяти.

Опишем генетические операторы над хромосомами состояний, записанными в виде сокращенных таблиц.

Алгоритм 3. Мутация сокращенных таблиц. По сравнению с представлением в виде полной таблицы, добавилась возможность мутации множества значимых предикатов. При этом каждый из значимых предикатов с некоторой вероятностью заменяется другим, не принадлежащим множеству (Рис. 6).

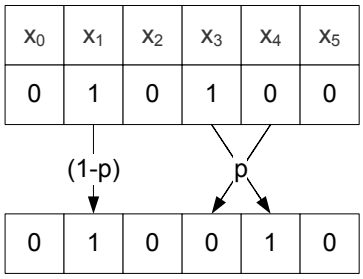

Рис. 6. Пример мутации множества значимых предикатов

Мутация самой сокращенной таблицы происходит так же, как мутация полной таблицы. Описание алгоритма приведено в листинге 3.

Листинг 3. Мутация сокращенных таблиц

```
State Mutate(State state)
{
      State mutant = state;
      if (с вероятностью p) {
            int from, to;
            случайно выбрать from и to, так что
                  (mutant.predicates[from]==1)&&
                  (mutant.predicates[to]==0);
            mutant.predicates[from] = 0;
            mutant.predicates[to] = 1;
```

```
        }
        // Остальная часть хромосомы мутирует, как в случае полных таблиц
        mutant.table = Mutate(mutant.table);
        return mutant;
}
```

Алгоритм 4. Скрещивание сокращенных таблиц. Это наиболее сложный из предлагаемых алгоритмов. Основная последовательность его шагов отражена в листинге 4.

Листинг 4. Скрещивание сокращенных таблиц

```
pair<State, State> Cross(State state1, State state2)
{
        State child1 = state1;
        State child2 = child1;

        ChoosePreds(state1.predicates, state2.predicates,
                child1.predicates, child2.predicates);

        int crossPoint = случайное число от 0 до tableSize;
        FillChildTable(state1, state2, child1, crossPoint);
        FillChildTable(state1, state2, child2, crossPoint);
        return make_pair(child1, child2);
}
```

Поскольку родительские хромосомы, представленные сокращенными таблицами, могут иметь разные множества значимых предикатов, сначала необходимо выбрать, какие из этих предикатов будут значимы для хромосом детей. Функция `ChoosePreds`, осуществляющая этот выбор, представлена в листинге 5.

Листинг 5. Выбор значимых предикатов детей при скрещивании сокращенных таблиц

```
void ChoosePreds(Predicates p1, Predicates p2, Predicates ch1, Predicates
ch2)
{
        for (для всех i: номеров предикатов) {
                if (p1[i] && p2[i]) {          // Предикат от обоих родителей
                        ch1[i] = ch2[i] = true; // достается обоим детям
                        запоминаем, что в наборах предикатов детей стало
                        на 1 меньше места;
                }
        }
        for (для всех i: номеров предикатов) {
                if (p1[i] != p2[i]) {
                        Predicates* pCh;
                        if (у обоих детей есть место) {
                                pCh = равновероятно любой ребенок;
                        } else {
                                pCh = тот ребенок, у которого еще есть место;
                        }
                        (*pCh)[i] = true;
                        запоминаем, у кого стало меньше места;
                }
        }
}
```

Работа функции `ChoosePreds` для родительских хромосом, представленных на Рис. 7, проиллюстрирована на Рис. 8.

x_0	x_1	x_2	x_3	x_4	x_5
0	1	0	1	0	0

x_0	x_1	x_2	x_3	x_4	x_5
1	1	0	0	0	0

x_1	x_3	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

x_0	x_1	s	z_0	z_1	z_2
0	0	1	1	1	0
0	1	2	0	0	1
1	0	2	0	1	0
1	1	0	0	1	0

Рис. 7. Родительские хромосомы, представленные сокращенными таблицами

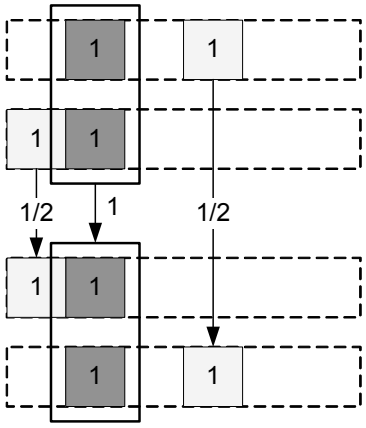

Рис. 8. Пример выбора значимых предикатов детей

После выбора значимых предикатов заполняются таблицы обоих детей. Алгоритм заполнения представлен в листинге 6.

Листинг 6. Заполнение таблиц детей при скрещивании сокращенных таблиц

```
void FillChildTable(State s1, State s2, State& child, int crossPoint)
{
      for (для всех i: строк таблицы child) {
            vector<int> lines1 = выбрать строки таблицы s1, в которых
                  предикаты, значимые для child, имеют те же значения,
                  что в строке i, причем, если предикат значим для обоих
                  родителей и i >= crossPoint, то его значение
                  не учитывается;
            vector<int> lines2 = выбрать строки таблицы s2, в которых
                  предикаты, значимые для child, имеют те же значения,
                  что в строке i, причем, если предикат значим для обоих
                  родителей и i < crossPoint, то его значение
                  не учитывается;

            vector<Probability> p1(nStates);
            vector<Probability> p2(nStates);
            for (для всех j из lines1) {
                  p1[целевое состояние у s1 в строке j] += 1.0;
            }
            for (для всех j из lines2) {
                  p2[целевое состояние у s2 в строке j] += 1.0;
            }
            Поделить значения p1 на число строк из lines1;
            Поделить значения p2 на число строк из lines2;
            vector<Probability> p = p1 + p2;
```

```
            child[i].targetState = выбрать случайно с распределением
                  вероятностей p;

            for (для всех k: номеров действий) {
                  Probability q1, q2;
                  for (для всех j из lines1) {
                        q1 += s1[j].output[k];
                  }
                  for (для всех j из lines2) {
                        q2 += s2[j].output[k];
                  }
                  Поделить q1 на число строк из lines1;
                  Поделить q2 на число строк из lines2;
                  child[i].output[k] = 1 с вероятностью (q1 + q2)/2
                        и 0 иначе;
            }
      }
}
```

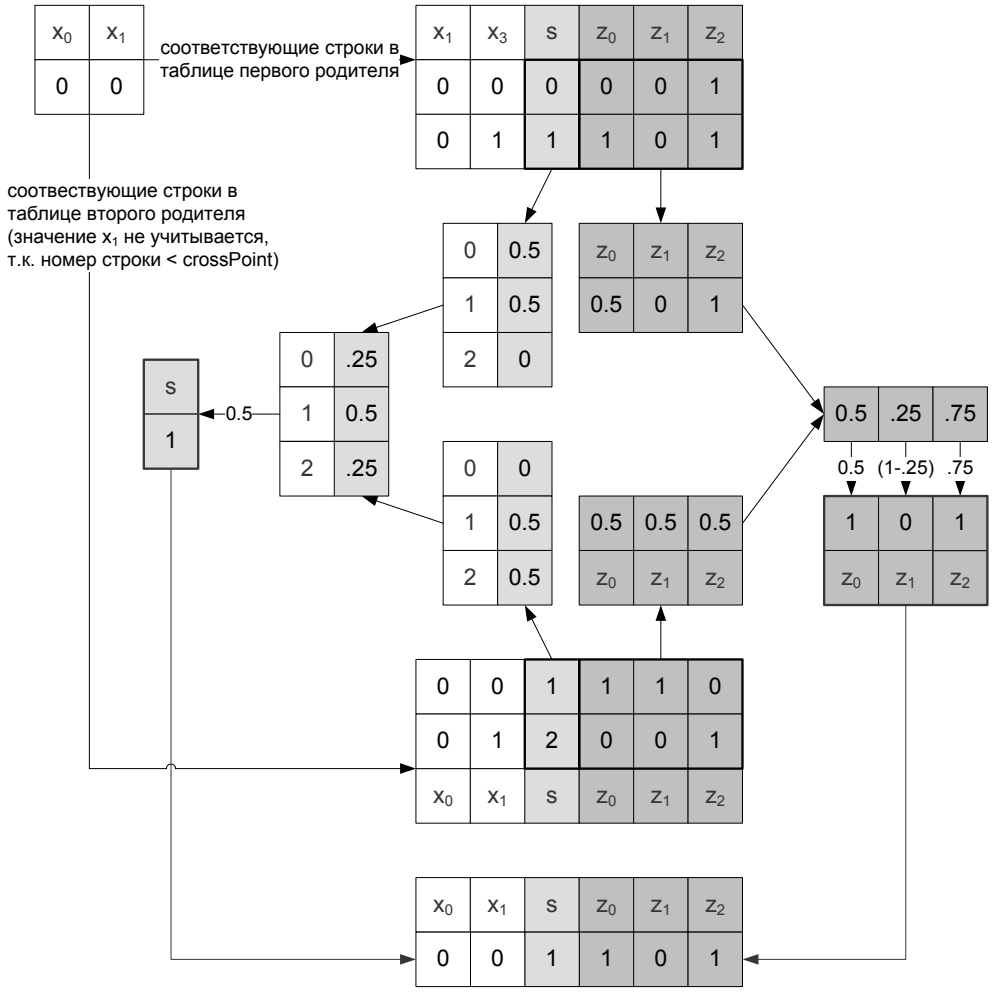

Рис. 9. Пример заполнения строки таблицы ребенка при скрещивании сокращенных таблиц

Иллюстрация примера заполнения первой строки таблицы одного из детей приведена на Рис. 9. В данной реализации оператора скрещивания на значения каждой строки таблицы ребенка влияют значения нескольких строк родительских таблиц. При этом конкретное значение, помещаемое в ячейку таблицы ребенка, определяется «голосованием» всех влияющих на нее ячеек родительских таблиц.

В описанном выше варианте алгоритма все состояния автомата имеют равное количество значимых предикатов (r – константа для всего процесса оптимизации). Однако предложенный алгоритм скрещивания легко расширяется на случай разного количества значимых предикатов у пары родителей. В этом случае необходимо добавить процедуру выбора r для каждого из детей. Такой выбор целесообразно делать случайным образом с математическим ожиданием, равным среднему арифметическому значений r пары родителей.

4. Детали применения алгоритма

4.1. Мутация, зависящая от пригодности

Что касается стратегий и параметров генетической оптимизации, основная особенность реализованного авторами подхода состоит во введении зависимости интенсивности мутаций особи от ее пригодности.

Применение интенсивной мутации к плохо приспособленным особям повышает эффективность эволюции, позволяет покидать локальные оптимумы и ослабляет проблему преждевременной сходимости популяции, тогда как для особей с высокой пригодностью оптимальны менее интенсивные мутации.

В Евклидовом пространстве оптимальный модуль вектора мутации можно с высокой точностью оценить сверху расстоянием между особью и глобальным оптимумом оценочной функции. Это расстояние в задаче не известно, однако можно ожидать тенденции к его сокращению с ростом пригодности особи. Если оценочная функция непрерывна и возрастает с приближением к оптимуму, можно проанализировать зависимость эффективности мутации от ее модуля. График на Рис. **10** отражает зависимость математического ожидания относительного сокращения расстояния до оптимума в результате попытки применения мутации.

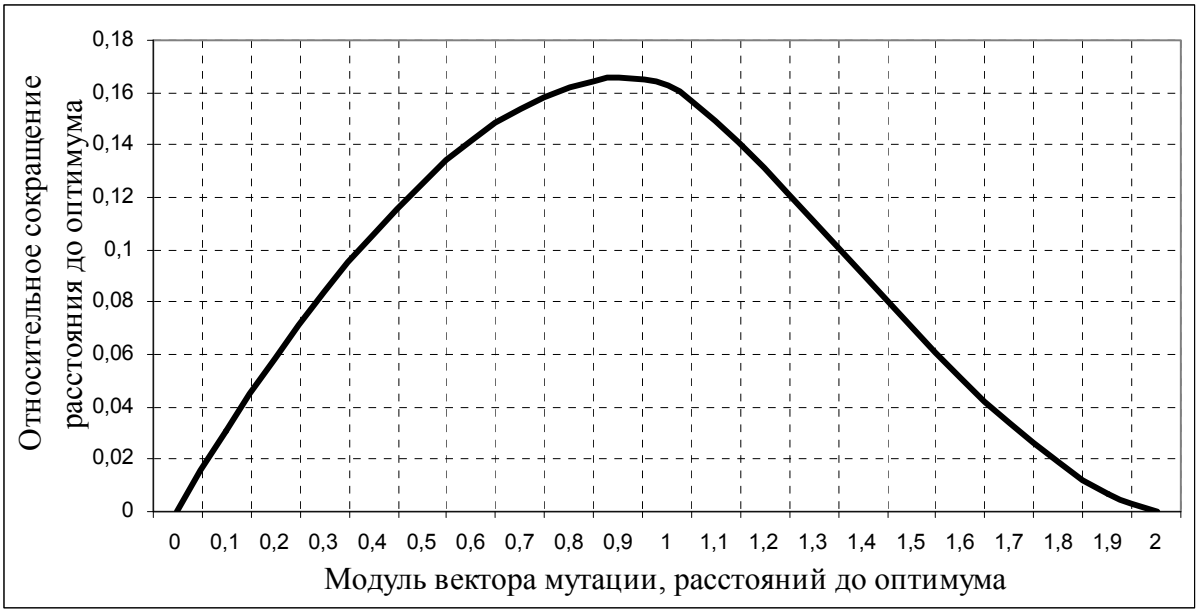

Рис. 10. Математическое ожидание сокращения расстояния до оптимума за одну операцию оценки пригодности

Можно сделать вывод, что оптимальная интенсивность мутации склонна к обратной зависимости от значения оценочной функции. В экспериментах авторы использовали интенсивность мутации, обратно пропорциональную пригодности.

4.2. Модификация оценочной функции

Задание оценочной функции на множестве вычислительных состояний объекта управления позволяет оптимизировать *поведенческие* свойства управляющего автомата. Однако, в целях улучшения понятности построенного результата оптимизации человеку, авторы считают целесообразным оптимизировать также *структурные* свойства автомата. Предлагается ввести набор стандартных структурных характеристик (например, количество достижимых управляющих состояний, суммарное число действий на переходах), которые, по желанию разработчика, будут влиять на оценку пригодности особей.

5. Экспериментальная проверка предложенных методов

В целях экспериментальной проверки предложенных методов авторами был реализован каркас *AutoGen*, позволяющий производить построение логики системы со сложным поведением, задавая в качестве входных данных только реализацию объекта управления и оценочной функции.

С помощью каркаса был реализован демонстрационный пример – построение автомата управления виртуальной «разливочной линией». Объект управления разливочной линии состоит из транспортера, на котором установлены бутылки, и дозирующей емкости (бака) с верхним (впускным) и нижним (выпускным) клапаном. Объем бака соответствует объему каждой бутылки. Перед началом работы все бутылки пусты.

Работой разливочной линии можно управлять, открывая и закрывая клапаны, а также запуская и останавливая транспортер. Управляющий автомат имеет пять двоичных входов от сигнализаторов, показывающих, правильно ли стоит бутылка под нижним клапаном, движение транспортера, опустошение и заполнение дозирующей емкости. Пятый вход автомата не несет информации и добавлен в экспериментальных целях.

Верхний клапан соединяет дозирующую емкость с трубой, подающей жидкость. Нижний клапан позволяет жидкости выливаться из бака и заполнять находящуюся под ним бутылку либо проливаться на транспортер, если бутылка не установлена. При наличии полной бутылки под нижним клапаном жидкость через клапан не проходит. Перед запуском линии бак заполнен и под нижним клапаном установлена пустая бутылка. Разливочная линия схематично изображена на Рис. 11.

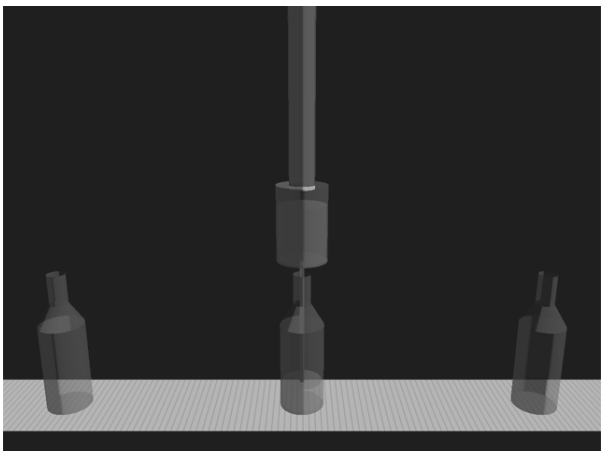

Рис. 11. Разливочная линия

Схема связей управляющего автомата разливочной линии показана на Рис. 12.

Рис. 12. Схема связей управляющего автомата разливочной линии

Задача состоит в заполнении максимального количества бутылок за определенный промежуток времени. Один из автоматов, решающих поставленную задачу, был построен авторами вручную (Рис. 13). Автомат содержит пять состояний. Здесь и далее состояние стартовым является первое состояние.

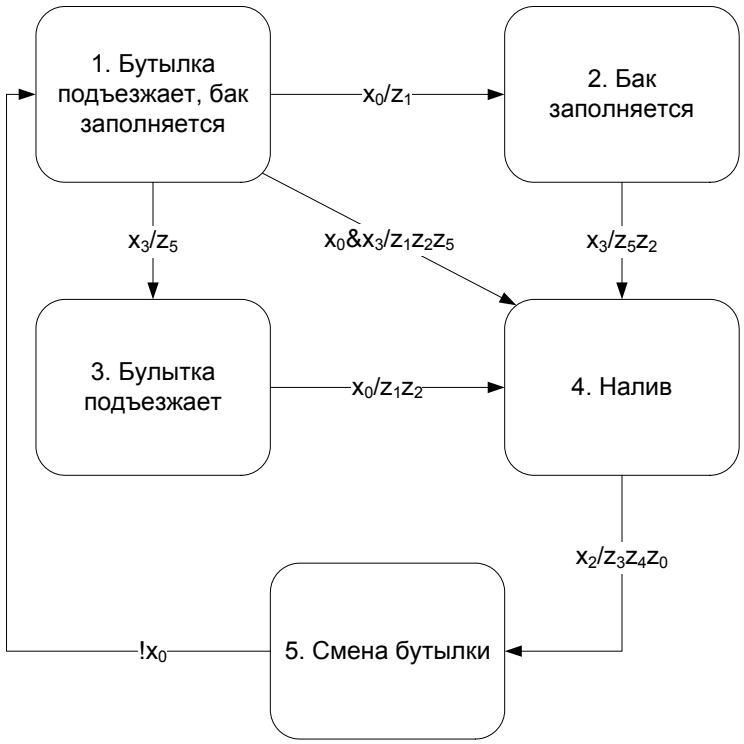

Рис. 13. Управляющий автомат разливочной линии, построенный вручную ($|S| = 5$, $\max(r) = 2$)

После этого с помощью каркаса *AutoGen* были автоматически построены автоматы с представлением состояний в виде полных и сокращенных таблиц. Автомат, представленный полными таблицами, имеет по 32 перехода из каждого состояния, что делает бессмысленным его изображение и анализ.

Автомат, представленный сокращенными таблицами, оказался значительно проще. Один из результатов оптимизации изображен на Рис. 14.

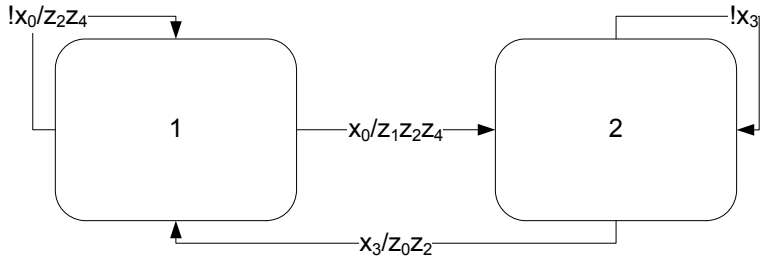

Рис. 14. Управляющий автомат разливочной линии, полученный в результате генетической оптимизации (|*S*| = 2, *r* = 1)

Анализ полученного автомата показал, что его работоспособность вызвана неявными зависимостями в эмуляторе разливочной линии. Например, этот автомат в процессе работы использует тот факт, что пропускная способность верхнего клапана вдвое больше, чем у нижнего. Таким образом, было установлено, что первоначальная реализация оценочной функции не полностью соответствует словесной формулировке задачи. Этот недостаток был впоследствии устранен путем внесения в параметры эмулятора (такие как скорость движения транспортера, пропускная способность клапанов) псевдослучайных отклонений. Результатом повторного применения генетической оптимизации стал автомат, граф переходов которого изображен на Рис. **15**.

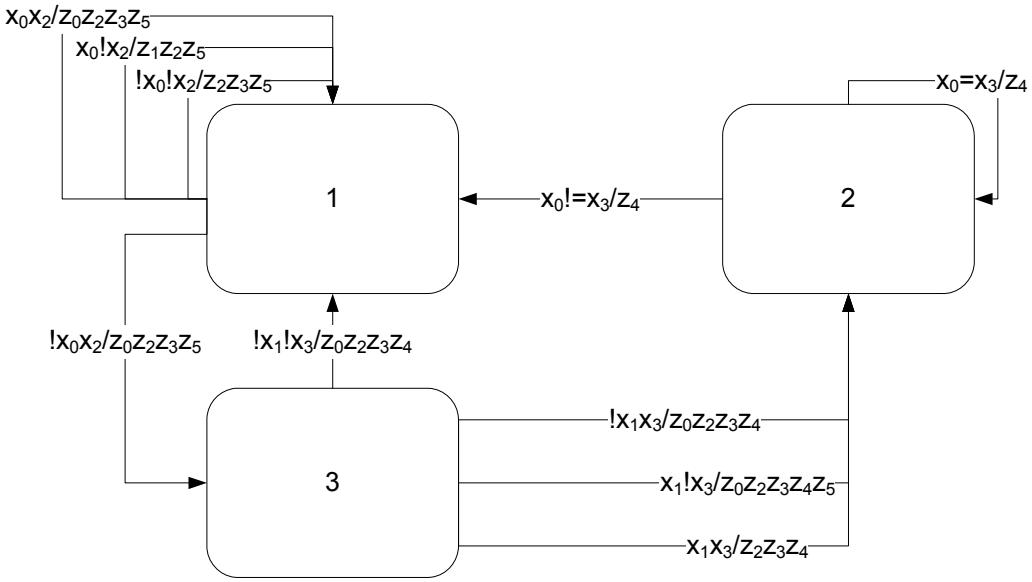

Рис. 15. Управляющий автомат "случайной" разливочной линии, полученный в результате генетической оптимизации (|*S*| = 3, *r* = 2)

В результате эксперимента авторы пришли к выводу о применимости методов генетического программирования для построения логики систем со сложным поведением. Представление состояний автоматов в форме сокращенных таблиц и разработанные для них генетические операторы показали работоспособность и эффективность. Применение предложенного подхода позволяет значительно сократить размерность пространства поиска и, соответственно, быстрее найти оптимальный управляющий автомат, пригодный для анализа и модификации.

Заключение

В работе предложены методы использования генетического программирования для построения систем со сложным поведением, теоретически обоснована целесообразность применения этих методов, а также проведена экспериментальная проверка их эффективности.

Авторы предполагают, что для дальнейшего развития генетического подхода к построению управляющих автоматов имеются широкие возможности.

Во-первых, необходимо изучить и реализовать другие решения проблемы экспоненциального роста размерности хромосомы состояния. Например, вместо ограничения числа предикатов, значимых в состоянии, можно ограничить число переходов из данного состояния (или суммарное число переходов в автомате), и задавать управляющую функцию в виде списка переходов, а не в виде таблицы. Авторам известны исследования представления управляющей функции в виде *дерева решений*.

Во-вторых, необходимо уделить внимание проблеме «долгого» вычисления оценочной функции. В генетическом программировании каждое вычисление оценочной функции подразумевает запуск модели вычисления. В случае автоматизированного объекта вычисление пригодности – это эмуляция определенного числа шагов работы заданного объекта управления совместно с оцениваемым автоматом. Такая эмуляция требует больших временных затрат по сравнению с другими этапами генетической оптимизации. Поэтому необходимо применять подходящие стратегии оценивания: турнирный метод, изменение числа шагов эмуляции в ходе оптимизации.

В-третьих, можно рассмотреть различные более общие постановки задачи построения систем со сложным поведением. В реализованном варианте вся структура объекта управления, включая предикаты и действия, задается разработчиком вручную. В то же время, для полного описания задачи достаточно задать лишь множество вычислительных состояний объекта управления и оценочную функцию на этом множестве. При таком подходе предикаты и действия могут строиться методами генетического программирования, но на основе других моделей вычисления, более подходящих для представления коротких функций, не содержащих логики. К таким моделям относятся, например, нейронные сети, язык ассемблера, упрощенные версии языков программирования высокого уровня и многие другие. Преимущества автоматного подхода к реализации систем со сложным поведением в этом случае не теряются, так как априори навязывается разделение системы на управляющий автомат и объект управления, и для каждой из этих компонент используется наиболее подходящая модель вычисления. Построенные автоматически предикаты и действия с большой вероятностью будут понятны человеку, поскольку это простые, короткие функции. Описание автомата, в свою очередь, остается высокоуровневым, как и в частном случае, подробно описанном в настоящей работе.

Можно рассмотреть также промежуточную постановку задачи, в которой структура предикатов и действий задается вручную, а некоторый набор их параметров подвергается оптимизации.

Наконец, необходимо рассмотреть подходы к представлению результатов генетической оптимизации в виде, более понятном человеку. В частности, необходимо автоматически строить граф переходов полученного автомата, а для сложных автоматов – производить их декомпозицию.

Литература

1. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
2. Шалыто А.А. Технология автоматного программирования // Мир ПК. 2003.
3. Harel D., Polity M. Modeling Reactive Systems with Statecharts. The Statemate Approach. New York: McGraw-Hill, 1998.
4. Koza J. Genetic Programming: On the programming of Computers by Means of Natural Selection. Cambridge: MIT Press, 1992.
5. Gold E.M. Language Identification in the Limit // Information and Control. 1967. № 10.
6. Belz A. Computational Learning of Finite-State Models for Natural Language Processing. PhD thesis. University of Sussex. 2000.
7. Clelland C.H., Newlands D.A. Pfsa modelling of behavioural sequences by evolutionary programming // Complex'94 – Second Australian Conference on Complex Systems. IOS Press, 1994.
8. Das S., Mozer M.C. A Unified Gradient-Descent/Clustering Architecture for Finite State Machine Induction // Advances in Neural Information Processing Systems. 1994.
9. Lankhorst M.M. A Genetic Algorithm for the Induction of Nondeterministic Pushdown Automata. Computing Science Report. University of Groningen Department of Computing Science. 1995.
10. Belz A., Eskikaya B. A genetic algorithm for finite state automata induction with an application to phonotactics // ESSLLI-98 Workshop on Automated Acquisition of Syntax and Parsing. Saarbruecken, 1998.
11. Ashlock D., Wittrock A., Wen T-J. Training finite state machines to improve PCR primer design // Congress on Evolutionary Computation (CEC'02). 2002.
12. Ashlock D.A., Emrich S.J., Bryden K.M. and others A comparison of evolved finite state classifiers and interpolated markov models for improving PCR primer design // 2004 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB'04). 2004.
13. Lucas S.M. Evolving Finite State Transducers: Some Initial Explorations // Genetic Programming: 6th European Conference (EuroGP'03). Berlin: Springer, 2003.
14. Kinnear K.E. Advances in Genetic Programming. Cambridge: MIT Press, 1994.
15. Teller A., Veloso M. PADO: A New Learning Architecture for Object Recognition // Symbolic Visual Learning. New York: Oxford University Press, 1996.
16. Banzhaf W., Nordin P., Keller R., Francone F.D. Genetic Programming – An Introduction. On the automatic Evolution of Computer Programs and its Application. San Francisco: Morgan Kaufmann Publishers, 1998.
17. Kantschik W., Dittrich P., Brameier M. Empirical Analysis of Different Levels of Meta-Evolution // Congress on Evolutionary Computation. 1999.
18. Kantschik W., Dittrich P., Brameier M., Banzhaf W. Meta-Evolution in Graph GP // Genetic Programming: Second European Workshop (EuroGP'99). 1999.
19. Teller A., Veloso M. Internal Reinforcement in a Connectionist Genetic Programming Approach // Artificial Intelligence. 2000.
20. Brameier M., Kantschik W., Dittrich P., Banzhaf W. SYSGP – A C++ library of different GP variants. Internal Report. Univ. of Dortmund. 1998.
21. Benson K. Evolving Automatic Target Detection Algorithms that Logically Combine Decision Spaces // Eleventh British Machine Vision Conference. 2000.
22. Kantschik W., Banzhaf W. Linear-Graph GP. A new GP Structure // 4th European Conference on Genetic Programming (EuroGP'02). 2002.
23. Brave S. Evolving Deterministic Finite Automata Using Cellular Encoding // Genetic Programming 1996: First Annual Conference. 1996.

24. Miller J.F., Thomson P. A Developmental method for growing Graphs and Circuits // Evolvable Systems: From Biology to Hardware. Berlin: Springer, 2003.
25. Poli R. Evolution of Graph-like Programs with Parallel Distributed Genetic Programming // Genetic Algorithms: Seventh International Conference. 1997.
26. Frey C., Leugering G. Evolving Strategies for Global Optimization. A Finite State Machine Approach // Genetic and Evolutionary Computation Conference (GECCO-2001). Morgan Kaufmann, 2001.
27. Petrovic P. Simulated evolution of distributed FSA behaviour-based arbitration // The Eighth Scandinavian Conference on Artificial Intelligence (SCAI'03). 2003.
28. Petrovic P. Evolving automatons for distributed behavior arbitration. Technical Report. Norwegian University of Science and Technology. 2005.
29. Petrovic P. Comparing Finite-State Automata Representation with GP-trees. Technical report. Norwegian University of Science and Technology. 2006.
30. Koza J.R. Future Work and Practical Applications of Genetic Programming // Handbook of Evolutionary Computation. Bristol: IOP Publishing Ltd, 1997.
31. Handley S. A new class of function sets for solving sequence problems // Genetic Programming 1996: First Annual Conference. 1996.
32. Langdon W. B. Genetic Programming and Data Structures. Boston: Kluwer, 1998.
33. Fogel L.J. Autonomous Automata // Industrial Research. 1962. № 4.
34. Fogel L.J. On the organization of intellect. PhD thesis. University of California. 1964.
35. Fogel L.J., Owens A.J., Walsh M.J. On the Evolution of Artificial Intelligence // 5th National Symposium on Human Factors in Electronics. San Diego, 1964.
36. Fogel L.J., Owens A.J., Walsh M.J. Artificial Intelligence through a Simulation of Evolution // Biophysics and Cybernetic Systems. London: Macmillan & Co, 1965.
37. Fogel L.J., Owens A.J., Walsh M.J. Artificial Intelligence through Simulated Evolution. New York: Wiley, 1966.
38. Fogel L.J. Extending Communication and Control through Simulated Evolution // Bioengineering – An Engineering View: Symp. Engineering Significance of the Biological Sciences. 1968.
39. Lutter B.E., Huntsinger R.C. Engineering applications of finite automata // Simulation. 1969. № 1.
40. Dearholt D.W. Some Experiments on Generalization Using Evolving Automata // 9th Intern. Conf. on System Sciences. Honolulu, 1976.
41. Atmar J.W. Speculation on the Evolution of intelligence and its possible realization in machine form. PhD thesis. New Mexico State University. 1976.
42. Takeuchi A. Evolutionary Automata – Comparison of Automaton Behavior and Restle's Learning Model // Information Science. 1980. № 2.
43. Burgin G.H. On playing two-person zero-sum games against nonminimax players // IEEE Trans. on Systems Science and Cybernetics. 1969. № 4.
44. Miller J.H. The Coevolution of Automata in the Repeated Prisoner's Dilemma. Working Paper. Santa Fe Institute. 1989.
45. Spears W.M., Gordon D.F. Evolving Finite-State Machine Strategies for Protecting Resources // International Syposium on Methodologies for Intelligent Systems. 2000.
46. Ashlock D. Evolutionary Computation for Modeling and Optimization. New York: Springer, 2006.
47. Fogel L.J., Angeline P.J., Fogel D.B. A Preliminary Investigation on Extending Evolutionary Programming to Include Self-adaptation on Finite State Machines // Informatica. 1994. № 4.
48. Carmel D., Markovitch S. Learning models of intelligent agents // Thirteenth National Conference on Artificial Intelligence. 1996.
49. Hsiao M.S. Sequential Circuit Test Generation Using Genetic Techniques. PhD thesis. University of Illinois at Urbana-Champaign. 1997.