Санкт-Петербургский государственный университет информационных технологий механики и оптики

На правах рукописи

Лобанов Павел Геннадьевич

Использование генетических алгоритмов для генерации конечных автоматов

Специальность 05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель –
доктор технических наук,
профессор Шалыто А. А.

Санкт-Петербург
2008

ОГЛАВЛЕНИЕ

Введение 5

Глава 1. Обзор задач, в которых генерация конечных автоматов осуществляется с помощью генетических алгоритмов 9

1.1. Генетические алгоритмы 9

1.2. Эксперименты Фогеля (регрессия) 12

1.3. Задача оптимизации функций 15

1.4. Автоматические переговоры 19

1.5. Распознавание нерегулярных языков 20

1.6. Проектирование логических схем для автоматов Мили 24

1.6.1. Задача кодирования состояний 25

1.7. Обзор методов генерации автоматов с помощью генетических алгоритмов 30

Выводы по главе 1 32

Глава 2. Генерация автоматов для решения задачи о флибах 34

2.1. Постановка задачи 34

2.2. Схемы работы генетических алгоритмов в задаче о флибах 35

2.2.1. Известный алгоритм 35

2.2.2. Предлагаемая модификация алгоритма 36

2.3. Реализация флиба 37

2.3.1. Известный метод 37

2.3.2. Предлагаемый метод 38

2.4. Генератор значений входной переменной 39

2.5. Функция приспособленности 39

2.6. Оператор одноточечного скрещивания 40

2.6.1. Известный алгоритм 40

2.6.2. Предлагаемый алгоритм 40

2.7. Оператор мутации....................41
2.7.1. Известный алгоритм....................41
2.7.2. Предлагаемый алгоритм....................41
2.8. Алгоритм восстановления связей между состояниями....................42
2.8.1. Программа для проведения экспериментов на флибах....................48
2.8.2. Эксперименты по применению алгоритма восстановления связей между состояниями....................49
2.9. Использование автоматов с флагами....................55
2.9.1. Эксперименты по использованию автоматов с флагами....................62
Выводы по главе 2....................67
Глава 3. Генерация автоматов для задачи об умном муравье....................68
3.1. Постановка задачи....................68
3.2. Известный генетический алгоритм....................70
3.2.1. Стратегия отбора....................70
3.2.2. Оператор скрещивания....................71
3.2.3. Оператор мутации....................72
3.3. Предложенные модификации алгоритма....................72
3.3.1. Сортировка состояний в порядке использования....................72
3.3.2. Оператор мутации «всегда вперед, если впереди еда»....................76
3.3.3. Уменьшение числа состояний....................77
3.4. Описание модели программы для сравнения эффективности генетических алгоритмов, решающих задачу об умном муравье....................77
3.5. Эксперименты....................82
Выводы по главе 3....................86
Глава 4. Генерация автоматов для задачи построения автопилота для упрощенной модели вертолета....................87
4.1. Постановка задачи....................87

4.2. Модель вертолета 90
4.3. Модель окружающей среды...92
4.4. Алгоритм построения автопилота 92
4.5. Общая схема генетического алгоритма...93
4.6. Описание программы 94
4.7. Эксперименты...95
Выводы по главе 4 98
Заключение...99
Литература..101
Приложения...105

ВВЕДЕНИЕ

Общая характеристика работы

Актуальность проблемы. В последнее время все шире применяется автоматное программирование, в рамках которого поведение программ описывается с помощью конечных детерминированных автоматов (в дальнейшем автоматов [1]).

Для многих задач автоматы удается строить эвристически, однако существуют задачи, для которых такое построение затруднительно или невозможно. Известны задачи (например, итерированная дилемма узников [2] и задача о «флибах» [3, 4]), в которых применение генетических алгоритмов (направленного перебора) позволяет генерировать автоматы. Это повышает уровень автоматизации *автоматных программ* и является одним из первых шагов к *автоматическому построению программ.*

Генетические алгоритмы [5] применяются при решении широкого круга задач. Однако при использовании классических генетических алгоритмов для генерации автоматов часто требуются большие вычислительные ресурсы, для того чтобы получить решение с необходимым уровнем качества.

Для повышения эффективности генетических алгоритмов требуется их модифицировать с учетом специфики задач, решаемых в диссертации.

Поэтому исследования, направленные на разработку более эффективных генетических алгоритмов для построения автоматов, весьма актуальны.

Цель диссертационной работы – разработка методов оптимизации генетических алгоритмов для построения автоматов.

Основные задачи исследования состоят в следующем:

1. Разработка новых операторов мутации, позволяющих повысить эффективность генетических алгоритмов, использующихся для построения автоматов, представленных в виде графов переходов.

2. Разработка функций приспособленности для повышения быстродействия генетических алгоритмов при построении автоматов.
3. Разработка модификаций генетических алгоритмов при построении автоматов для ряда задач.
4. Исследование эффективности генетических алгоритмов при построении двух разнотипных автоматов для одной из рассмотренных задач.

Методы исследования. В работе использовались методы теории автоматов, программирования и генетические алгоритмы.

Научная новизна. В работе получены новые научные результаты, которые выносятся на защиту:

1. Разработаны два оператора мутации для автоматов, представленных в виде графов переходов: восстановление связей между состояниями и сортировка состояний в порядке использования.
2. Предложена функция приспособленности, учитывающая число используемых состояний в автомате, которая позволяет для рассмотренного класса задач повысить эффективность генетических алгоритмов для генерации автоматов.
3. Разработаны модификации генетических алгоритмов для задач о флибах и умном муравье и для построения автопилота для упрощенной модели вертолета.
4. При решении задачи о флибах выполнено сравнение эффективности генетических алгоритмов, которые позволяют генерировать автоматы Мили и автоматы Мили с флагами. При этом показано, что точность предсказания флибом, который моделируется автоматом с флагами, выше.

Результаты диссертации получены в ходе работ по государственному контракту «Технология генетического программирования для генерации автоматов управления системами со сложным поведением» (шифр «2007-4-14-18-01-033»), проводимых СПбГУ ИТМО в 2007 – 2008 гг. в рамках Федеральной целевой программы

«Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России» на 2007 – 2012 гг.

Достоверность научных положений, выводов и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев, а также результатами экспериментов по использованию предложенных в диссертации методов.

Практическое значение работы заключается в том, что полученные результаты могут быть использованы для решения практических задач. Предложенные модификации генетических алгоритмов для построения автоматов, позволяют повысить эффективность работы алгоритмов, что подтверждается экспериментальными данными. Практическая ценность подтверждается внедрением результатов работы.

Внедрение результатов работы. Результаты, полученные в диссертации, используются на практике в компании *Транзас* (Санкт-Петербург) при разработке тренажера вертолета, а также в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО по курсу «Теория автоматов в программировании».

Апробация диссертации. Основные положения диссертационной работы докладывались на 4-й Всероссийской научной конференции «Управление и информационные технологии» СПбГЭТУ «ЛЭТИ» (2006), XXXVI научной и учебно-методической конференции профессорско-преподавательского и научного состава СПбГУ ИТМО (2007), IV межвузовской конференции молодых ученых СПбГУ ИТМО (2007), XIV Всероссийской научно-методической конференции «Телематика-2007» (Санкт-Петербург), XXXVII научной и учебно-методической конференции СПбГУ ИТМО (2008), V Всероссийской межвузовской конференции молодых ученых СПбГУ ИТМО (2008), XV международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке» (Санкт-Петербургский государственный политехнический университет, 2008).

Публикации. По теме диссертации опубликовано шесть печатных работ, в том числе три в изданиях из перечня ВАК («Известия РАН. Теория и системы управления» и «Научно-технический вестник СПбГУ ИТМО»).

Структура диссертации. Диссертация изложена на 114 страницах и состоит из введения, четырех глав и заключения. Список литературы содержит 32 наименования. Работа содержит 42 рисунков и 15 таблиц.

В первой главе приведен обзор задач, в которых автоматы строятся с помощью генетических алгоритмов. Вторая глава содержит описание методов генерации автоматов для задачи о флибах, которая появилась в ходе первых работ по моделированию способностей живых существ к предсказанию. Третья глава содержит описание методов генерации автоматов для задачи об умном муравье, которая стала классической в рассматриваемой области. В четвертой главе, используя сортировку состояний в порядке использования, предложенную в предыдущей главе, разработан генетический алгоритм генерации автоматов для задачи построения автопилота для упрощенной модели вертолета (в дальнейшем вертолета).

ГЛАВА 1. ОБЗОР ЗАДАЧ, В КОТОРЫХ ГЕНЕРАЦИЯ КОНЕЧНЫХ АВТОМАТОВ ОСУЩЕСТВЛЯЕТСЯ С ПОМОЩЬЮ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

1.1. Генетические алгоритмы

Для различных биологических объектов характерна оптимальность, а также согласованность и эффективностью их работы. Многих исследователей давно интересовал «философский» вопрос – возможно ли эффективное построение значимых и полезных для человека систем на основе принципов биологической эволюции?

Подобные идеи возникали у ряда исследователей. В 1962 г. Л. Фогель начал моделировать интеллектуальное поведение индивида и его развитие в процессе эволюции [6]. При этом поведение индивида задавалось конечным автоматом. Продолжая эти исследования, Л. Фогель, А. Оуэнс и М. Уолш предложили в 1966 г. схему эволюции конечных автоматов, решающих задачи предсказания [3]. Примерно в это же время (в середине 60-х годов) Дж. Холланд разработал новый метод поиска оптимальных решений – генетические алгоритмы. Результатом его исследований стала книга [7], вышедшая в 1975 г. Эти работы заложили основы эволюционных вычислений.

Приведем ряд определений, которые играют важную роль в теории эволюционных вычислений и необходимы для описания генетических алгоритмов.

Популяция – это совокупность особей на рассматриваемой стадии эволюции.

Индивид (особь) – единичный представитель популяции.

Хромосома – структура, содержащая генетический код индивида.

Ген – определенная часть хромосомы, кодирующая врожденное качество индивида.

Функция приспособленности (*fitness-функция, приспособленность*) - определяет, как близка особь к решению задачи.

Генетические алгоритмы [5] – это оптимизационный метод, базирующийся на принципах естественной эволюции популяции особей (индивидов). Задача оптимизации состоит в максимизации функции приспособленности (фитнес-функции). В процессе эволюции (в результате отбора, скрещивания и мутаций особей, а также других операторов генетических алгоритмов) происходит поиск особей с высокой приспособленностью. По сравнению с другими оптимизационными методами генетические алгоритмы имеют особенности: параллельный поиск, случайные мутации и скрещивание уже найденных хороших решений. Приведем описание генетических операторов.

Стратегия отбора определяет каким образом формируется новое поколение особей. Известны различные стратегии отбора, например, отбор отсечением [8], турнирный отбор [9], элитизм [9] и пропорциональный отбор [8]. Кратко опишем особенности каждой из этих стратегий:

- *Отбор отсечением* использует отсортированную по убыванию приспособленности популяцию. Число особей для скрещивания определяется в соответствии с *порогом*. Порог определяет какая доля особей, начиная с самой первой (самой приспособленной), будет принимать участие в отборе.
- *Турнирный отбор* работает следующим образом: из популяции случайным образом выбирается t особей. Наиболее приспособленная из этих особей допускается к скрещиванию (между особями проводится турнир).
- *Пропорциональный отбор* подразумевает, что сначала подсчитывается приспособленность каждой особи f_i и вычисляется средняя приспособленность особей в популяции $f_{cp} = \sum_{i=1}^{N} f_i$. После этого строится

промежуточная популяция, причем вероятность попадания особи в эту популяцию определятся соотношением $\frac{f_i}{f_{cp}}$. Например, если дробь равна 2.36, то особь будет добавлена в промежуточную популяцию два раза. С вероятностью 0.36 она будет добавлена третий раз. Особи для скрещивания выбираются из промежуточной популяции случайным равновероятным образом.

- *Элитизм* подразумевает, что несколько лучших родительских особей добавляются в новое поколение. Использование элитизма позволяет не потерять хорошее промежуточное решение, но в тоже время из-за него алгоритм может застрять в локальном экстремуме.

Как известно, в теории эволюции важную роль играет то, каким образом признаки родителей передаются потомкам. В генетических алгоритмах за передачу признаков отвечает оператор, который называется *скрещивание*. В литературе по генетическим алгоритмам этот оператор называют также кроссинговер, кроссовер или рекомбинация. Итак, скрещивание – процесс обмена генетическим материалом. Это операция, при которой две хромосомы обмениваются своими частями.

Мутация – случайное изменение одной или нескольких позиций в хромосоме. Мутации могут иметь как отрицательные, так и положительные последствия – приводить к появлению у индивида новых признаков. Таким образом, мутации являются двигателем естественного отбора, так как обеспечивают поддержание разнообразия особей в популяции.

Порядок генов в хромосоме является часто критическим. В ряде работ были предложены методы для *переупорядочения* позиций генов в хромосоме во время работы алгоритма. Цель переупорядочения состоит в том, чтобы попытаться найти порядок генов, который имеет лучший эволюционный потенциал. Одним из таких методов является инверсия, выполняющая реверс порядка генов между двумя случайно

выбранными позициями в хромосоме. Когда используются методы переупорядочения, гены должны содержать некоторую «метку», такую, чтобы их можно было правильно идентифицировать независимо от их позиции в хромосоме. Многие исследователи использовали инверсию в своих работах, однако мало кто из них пытался обосновать инверсию и определить ее вклад в работу генетического алгоритма в целом. Переупорядочение также значительно расширяет область поиска. Мало того, что генетический алгоритм пытается находить «хорошие» множества генов, но он также одновременно пробует находить хорошее упорядочение генов. Еще раз отметим, что оператор переупорядочения изменяет структуру хромосомы.

Как правило, выделяют еще один оператор для генетических алгоритмов – *оператор генерации случайной особи*, который может быть использован при создании начальной популяции, при пополнении популяции случайными особями и при некоторых мутациях.

Для генетических алгоритмов характерна относительно простая структура особей. *J. Koza* [10] предложил задавать особи в виде деревьев решений. Генетические алгоритмы, основанные на таких структурах, получили название генетическое программирование.

Термины «генетические алгоритмы» и «генетическое программирование» понимаются в широком смысле – для обозначения эволюционных вычислений. Для обозначения генетических алгоритмов, оперирующих с битовыми строками фиксированной длины (описанных, например, в работе [2]), используется термин «традиционные генетические алгоритмы».

1.2. Эксперименты Фогеля (регрессия)

Один из создателей эволюционного программирования Л.Фогель рассматривал интеллектуальное поведение индивида, как способность успешно предсказывать поведение среды, в которой он находится, и в соответствии с этим действовать. В 60-х

годах прошлого века Л. Фогель поставил ряд экспериментов [3] по созданию искусственных систем, способных адаптироваться к первоначально не известной им среде.

В проведенных экспериментах Л. Фогель моделировал поведение простейшего живого существа, названного «флиб», которое способно предсказывать изменения параметра среды, обладающего *периодичностью*. Это существо моделировалось *конечным автоматом* с действиями на переходах – автоматом Мили [11]. В качестве среды выступала последовательность символов над двоичным алфавитом, например, `(1111010010)`. На вход автомата в каждый момент времени подавалась битовое значение параметра окружающей среды. Автомат формировал значение выходной переменной – возможное значение рассматриваемого параметра в следующий момент времени. На рис. 1 приведен один из возможных автоматов, который построен для распознавания, указанной выше последовательности.

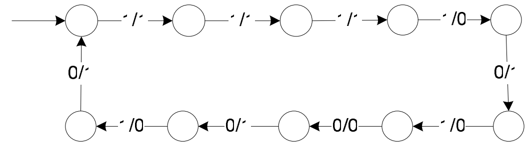

Рис. 1. Граф переходов эвристически построенного автомата

Задача состояла в эволюционном построении автомата, способного как можно более точно в смысле выбранной функции приспособленности (например, числа совпавших символов) предсказывать поведение среды – угадывать следующий символ последовательности. Кроме того, Л. Фогель накладывал ограничения на сложность порождаемого автомата, так как строить автоматы с числом состояний, равным длине обозреваемой последовательности, не представляет труда. Таким образом, предпочтение отдавалось автоматам, угадывающим как можно лучше, и в то же время имеющим как можно меньшее число состояний.

В начале эксперимента [3] задавалась периодическая последовательность символов над двоичным алфавитом, например (101110011101) и выбирался префикс данной последовательности малой длины. После этого создавалась популяция автоматов с небольшим числом состояний (например, пять). Затем каждый из автоматов путем одной из описанных ниже пяти мутаций (выбираемой случайным образом равномерно) производил потомка. Далее над потомком подобным образом производилось еще несколько мутаций (их число определялось случайным образом). Получившийся в результате мутаций автомат добавлялся в популяцию. При этом были допустимы следующие мутации автомата:

- добавление состояния;
- удаление состояния (в случае, если число состояний больше единицы);
- замена начального состояния (в случае, если число состояний больше единицы);
- замена перехода;
- замена действия на переходе.

После добавления потомков в популяцию на всех ее особях вычислялась функция приспособленности. Половина наиболее приспособленных особей переносилась в популяцию следующего поколения, а менее приспособленные автоматы – отбрасывались. Таким образом, размер популяции оставался постоянным. Стоит отметить, что в силу ограниченных возможностей компьютеров того времени, размер популяции был мал – всего несколько особей. Данный процесс продолжался до тех пор, пока не удавалось достичь желаемого результата – максимальное значение функции приспособленности не превосходило заданного порога. После этого к битовой последовательности, которая определяла среду, добавлялся очередной символ, и эволюционный процесс переходил в очередную фазу.

По мнению Л. Фогеля, результаты экспериментов показали, что эволюционное программирование может быть успешно применено для построения интеллектуальных

искусственных систем. При этом было отмечено, что построение вручную автоматов столь же результативных и простых, как те, что были построены эволюционным алгоритмом, является крайне сложной задачей.

1.3. Задача оптимизации функций

В работе [12] предлагается подход, позволяющий свести задачу оптимизации функций к задаче управления. При этом процесс нахождения глобального максимума сводится к перемещению по дискретизированному пространству поиска. Для управления перемещением вводится автомат Мили. Сформулируем рассматриваемую задачу: задано некоторое множество $\Omega \subset R^2$, и на нем определена функция $\Phi : \Omega \subset R_+$. Перемещение объекта управления по пространству Ω описывается следующим образом: $x_{t+1} = x_t + f(\Phi(x_t), \Phi(x_{t-1}))$, где $x_t \subset \Omega$, функция $f : R_+ \times R_+ \to \{x \mid x_t + x \in \Omega\}$ – стратегия перемещения. Таким образом, перемещение на текущем шаге (в текущий момент времени) зависит от значений функции Φ, исследованных на прошлом и позапрошлом шагах. Пусть $T \subset N$, $T > 1$ – время, отведенное на перемещение по пространству Ω. Задачей является максимизация функционала $\Psi(f, T) = \dfrac{\upsilon(\Phi(x_T) + 1)}{\max\limits_{0 \le t \le T} \Phi(x_t) + 1} + \max\limits_{0 \le t \le T} \Phi(x_t)$ по стратегии перемещения f. Таким образом, производится поиск стратегий, которые, с одной стороны, должны искать глобальный максимум (этому требованию соответствует второе слагаемое в сумме), а, с другой стороны, к окончанию поиска текущее положение x_T должно быть «недалеко» от максимального из исследованных (этому требованию соответствует первое слагаемое). Константа υ в числителе первого слагаемого устанавливает приоритет между первым и вторым требованиями.

Функция $\Psi(f, T)$ является функцией приспособленности. Стратегия перемещения f реализуется, как показано на рис. 2.

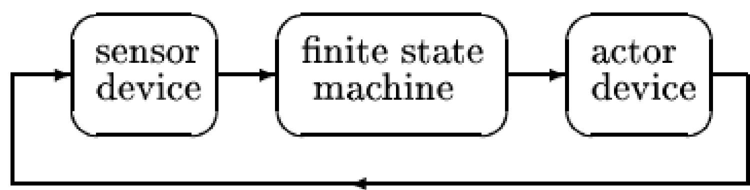

Рис. 2. Структура стратегии перемещения

Sensor device (преобразователь) переводит непрерывные значения $\Phi(x_t)$, $\Phi(x_{t-1})$ из R_+ во множество X дискретных входных воздействий автомата Мили A, реализуя функцию $\sigma : R_+^2 \to X$. Данная функция наряду с автоматом A выводится генетическим алгоритмом.

Finite state machine Mealy (автомат Мили) A задается множеством входных воздействий X, действий Y, состояний S, начальным состоянием $s_1 \in S$, функцией переходов автомата $\delta : X \times S \to S$ и функцией выходов автомата $\gamma : X \times S \to Y$.

Actor device (устройство перемещения) переводит выходные воздействия автомата в команду по перемещению объекта управления по двумерной сетке, дискретизирующей Ω. Устройство перемещения хранит информацию о текущем и предыдущем положениях устройства управления x_t и x_{t-1}, а также вектор перемещения d_t, который определяет как направление перемещения, так и величину шага, которая может быть равна шагу сетки, либо удвоенному шагу сетки. В зависимости от выходных воздействий автомата устройство перемещения оставляет устройство управления на месте $x_{t+1} = x_t$, либо перемещает его в одно из четырех ортогональных направлений. При этом шаг перемещения может остаться неизменным, либо удвоиться, либо уменьшиться вдвое. Определяется множество возможных перемещений $Y = \{F, B, L, R, S, \quad 2F, \frac{1}{2}F\}$, элементы которого соответствует перемещению вперед, назад, влево, вправо, вперед с удвоением шага, вперед с уменьшением шага вдвое.

Традиционное генетическое программирование конструирует функции σ, δ, γ одновременно – дерево, являющееся особью генетического программирования, определяет сразу три функции. Вершины дерева удовлетворяют следующей грамматике:

```
cond    ::= ≥0(<arith>,{<rule>},{<rule>})
rule    ::= γ(<state>,<output>) | δ (<state>,<state>) | <cond>
arith   ::= <arith> + <arith> | <arith> - <arith> | <arith> <arith> | <arith> % <arith> | A | B | c∈R
state   ::= s∈S
output  ::= y∈Y
```

Дерево параметризуется константами *A, B*, которые при вычислении заменяются на значения $\Phi(x_t)$, $\Phi(x_{t-1})$. Корнем дерева является *cond*-выражение. Пример дерева выражений приведен на рис. 3.

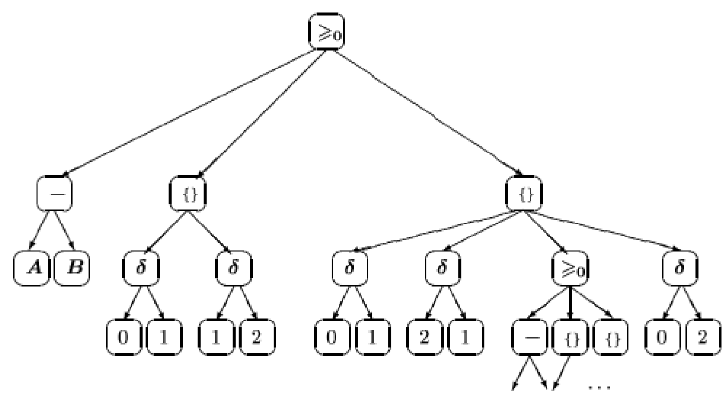

Рис. 3. Дерево выражений

Для наглядности иллюстрации в данном дереве не отображена функция γ. Результатом вычисления выражения приведенного вида после подстановки значений $\Phi(x_t)$ и $\Phi(x_{t-1})$ являются конечное множество переходов `(<state>,<state>)` и конечное множество выходных воздействий `(<state>,<output>)`, которые

определяют функции δ, γ. Если в множестве переходов (выходных воздействий) присутствуют два различных перехода (два различных выходных воздействия) из одного состояния, то выбирается самый левый из них (в приведенном примере - переход $\langle 0,1\rangle$).

Генетические операции (скрещивание, мутация) определены таким образом, чтобы при их выполнении порождались корректные деревья. Используются три дополнительные генетические операции над вершинами, являющимися списком инструкций (данные вершины обозначены символом {} на рис. 3): удаление поддерева, добавление поддерева, изменение порядка на поддеревьях.

Эксперименты проводились на трех различных функциях Φ:

$$\Phi_1(x) := -50\mu_1 \sum_{i=1}^{n} x_i \sin\sqrt{50x_i};$$

$$\Phi_2(x) := \mu_2 \sum_{i=1}^{n} \left(\frac{1}{4} x_i^2 - 10\cos(\pi x_i) + 10\right);$$

$$\Phi_3(x) := \mu_3 \sum_{i=1}^{n-1} \left(\frac{1}{4} x_{i+1} - \frac{1}{25} x_i^2\right) + \left(\frac{1}{5} x_i - 1\right)^2.$$

Параметр n принимал значение равное двум, $v = 10$, $T = 50$, а константы μ_i были выбраны таким образом, чтобы $\Phi_i \in [0,1]$. Пространством поиска являлось множество $\Omega := [0, 10]^2$. Автоматы содержали по 10 состояний. Популяция состояла из 10 автоматов, для каждого из которых случайным образом выбиралось $x_0 \in \Omega$ – положение объекта управления в начальный момент времени.

Графики первой и третьей функций приведены на рис. 4.

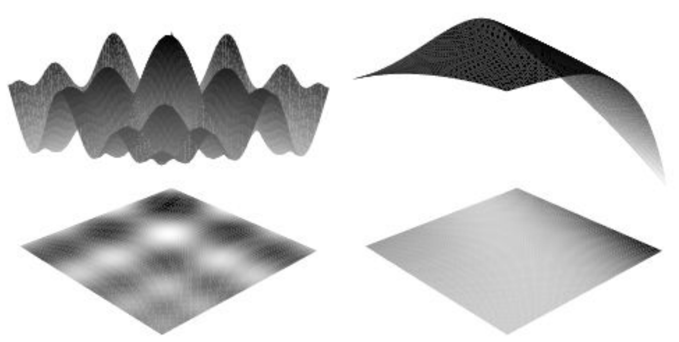

Рис. 4. Тестовые функции приспособленности Φ_1 и Φ_3

В результате экспериментов генетическое программирование сгенерировало стратегии перемещения f, для которых значение функции приспособленности близко к максимальному (единице). Кроме того, найденные стратегии являются устойчивыми как к изменению позиции x_0 объекта управления в начальный момент времени, так и к замене функции приспособленности Φ на другую, которая обладает аналогичными свойствами.

1.4. Автоматические переговоры

Приведем описание задачи автоматических переговоров [13]. Для начала выделяется фиксированный набор возможных предложений. После этого два переговорщика попеременно выдвигают одно из них. Любой из участников может либо принять предложение соперника, либо выдвинуть встречное предложение.

Каждый из переговорщиков стремится максимизировать известную только ему функцию полезности, определенную на множестве возможных предложений. Однако, если переговоры затягиваются, то переговорщики не получают никакой прибыли.

В работе [13] для выражения стратегий переговорщиков применены конечные автоматы. Входными и выходными воздействиями являются возможные предложения. Кроме того, вводится дополнительное выходное воздействие, соответствующее принятию предложения.

Приведем описание используемой операции скрещивания. Операция принимает на вход два автомата и выдает один. Для этого множество состояний каждого автомата разбивается на два подмножества. После этого подмножество первого автомата, содержащее указанную вершину, объединяется с подмножеством второго автомата, которое не содержит начальную вершину. Заметим, что некоторые ребра теперь ведут наружу или исходят из вершин, не представленных в полученном автомате. Эти ребра изменяются для корректности автомата. Предложенный метод напоминает скрещивание абстрактных графов, используемое в работе [14].

Разработанный метод был апробирован на достаточно большом наборе моделей переговоров. Среди тестов присутствовали как ситуации, в которых интересы переговорщиков совпадали, так и ситуации, в которых их интересы были диаметрально противоположными. Кроме того, была осуществлена проверка на сложных функциях оценки, для которых не удается построить оптимальную стратегию. В результате было подтверждено, что метод, представляющий стратегию с помощью конечных автоматов, применим к задаче автоматических переговоров.

1.5. Распознавание нерегулярных языков

На практике языки, которые необходимо классифицировать, редко являются регулярными [15]. Поэтому предпринимались попытки автоматизировать процесс построения распознавателей нерегулярных языков. Простейшим обобщением автомата на более широкий класс языков является *автомат с магазинной памятью*. Такой автомат отличается от конечного автомата тем, что может использовать в процессе работы стек: по комбинации текущего состояния, входного символа и символа на вершине стека автомат выбирает следующее состояние и, возможно, символ для записи в магазин. Может быть показано, что для любого контекстно-свободного языка можно построить недетерминированный автомат с магазинной памятью, распознающий требуемый язык. Построение же детерминированного автомата с магазинной памятью

возможно не всегда. Классическим примером такого языка является язык палиндромов [15].

В работе [16] рассматривается задача построения недетерминированного автомата с магазинной памятью по множеству примеров с помощью генетических алгоритмов. При этом используется построение автоматов, допускающих вход по пустому стеку. В автоматах запрещены ε-переходы. Может быть показано, что определенные таким образом недетерминированные автоматы с магазинной памятью по-прежнему способны распознавать любые контекстно-свободные языки.

Остановимся на функции приспособленности выводимых автоматов. В работе [16] утверждается, что доля верно классифицированных тестовых слов является плохой функцией приспособленности. Возрастание значения этой функции вовсе не означает, что автомат стал лучше распознавать как слова принадлежащие языку, так и не принадлежащие ему. Приведем построенную в этой работе функцию:

$$\mathtt{F}(\mathtt{M}, \mathtt{S}_+, \mathtt{S}_-) = (\mathtt{cor}(\mathtt{M}, \mathtt{S}_+) / 3 + \mathtt{pref}(\mathtt{M}, \mathtt{S}_+) / 3 + \mathtt{stack}(\mathtt{M}, \mathtt{S}_+) / 3) \times \mathtt{cor}(\mathtt{M}, \mathtt{S}_-).$$

Здесь $\mathtt{S}_+$ - множество примеров цепочек, принадлежащих языку, $\mathtt{S}_-$ - множество примеров, которые не принадлежат языку цепочек, а $\mathtt{cor}(\mathtt{M}, \mathtt{S})$ - доля верно распознанных автоматом $\mathtt{M}$ слов из множества $\mathtt{S}$.

Приведем описание функции $\mathtt{pref}(\mathtt{M}, \mathtt{S})$, смысл которой заключается в том, чтобы премировать автоматы, принимающие как можно более длинные префиксы слов, принадлежащих языку. Пусть при этом

$$\mathtt{pref}^*(\mathtt{M}, \mathtt{w}) = \mathtt{max}\left(\frac{|\mathtt{v}|}{|\mathtt{w}|}\right),$$ где $\mathtt{v}$ является префиксом $\mathtt{w}$ и принимается автоматом;

$$\mathtt{pref}(\mathtt{M}, \mathtt{S}) = \frac{\sum_{\mathtt{w} \in \mathtt{S}} \mathtt{pref}^*(\mathtt{M}, \mathtt{w})}{|\mathtt{S}|}.$$

Функция $\mathtt{stack}(\mathtt{M}, \mathtt{S}_+)$ премирует автоматы, имеющие меньшее число символов в стеке в конце разбора цепочек из множества примеров S. При этом

$\text{stack}^*(\text{M, w}) = \dfrac{1}{1+\gamma}$, где γ - число символов в стеке в конце разбора цепочки w;

$$\text{stack}(\text{M, S}) = \frac{\sum_{w \in S} \text{stack}^*(\text{M, w})}{|S|}.$$

Автоматы представляются с помощью строк. Для того чтобы избежать экспоненциального роста длины строки, в работе [16] введено ограничение на число переходов автомата. Автомат записывается в виде строки следующим образом: для каждого перехода выписываются начальное состояние, конечное состояние, входной символ, символ на вершине стека, следующее состояние и набор символов, которые размещаются в стеке. Максимальное число символов, которые можно положить в стек фиксируется. Поэтому все строки имеют одинаковую длину. В работе [16] в качестве оператора скрещивания используется двухточечное скрещивание строк, а качестве мутации – мутация над строками. Возможна запись, как в двоичную строку, так и в строку над числами.

Разработанный метод был апробирован на десяти тестовых языках. В набор входило шесть регулярных языков, три детерминированных контекстно-свободных языков и один недетерминированный контекстно-свободный язык. Для каждого из тестовых языков удалось построить соответствующий автомат с помощью предложенного генетического алгоритма. Сравнение использования битовых строк и строк над числами показало, что первые имеют преимущество.

В работе [17] предлагается другой способ применения генетического программирования для построения автоматов с магазинной памятью. В этой работе строится детерминированный автомат, тем самым, сужая класс языков до детерминированных контекстно-свободных грамматик.

В работе используется специфическое представление автомата с магазинной памятью, названное диаграммой псевдосостояний. Представление аналогично

диаграмме переходов конечного автомата Мура [11], но его состояния разделены на различные классы:

- READ-состояния - осуществляют чтение символа из входного потока. После этого в зависимости от прочитанного символа автомат переходит в следующее состояние.
- POP-состояния - осуществляют чтение символа с вершины стека. Переход выполняется в зависимости от прочитанного символа.
- PUSH-состояния - помещают в стек определенный символ. Они имеют единственный переход.
- ACCEPT-состояния - допускающие состояния автомата. Они не имеют выходов.
- REJECT-состояния - отвергающие состояния автомата. Они также не имеют выходов.

Пример диаграммы псевдосостояний приведен на рис. 5.

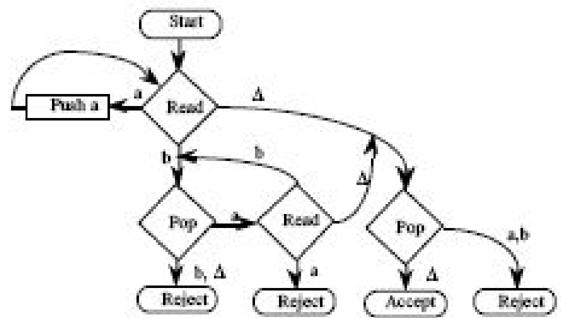

Рис. 5. Диаграмма переходов псевдосостояний

Диаграммы переходов получаются в результате вычисления выражений на специальном макроязыке *APDAL*. Выражения, соответствующие искомому автомату, выводятся с использованием генетического программирования. Для простоты входной алфавит автомата полагается равным $\{a, b\}$. Стековый алфавит совпадает с входным. Автор работы [17] вводит следующие инструкции:

- READ - создать новое READ-состояние.

- PUSHA - создать новое PUSH-состояние, помещающее в стек символ *a*.
- PUSHB - создать новое PUSH-состояние, помещающее в стек символ *b*.
- POP - создать новое POP-состояние.
- REJECT - создать новое отвергающее состояние.
- ACCEPT - создать новое допускающее состояние.

В качестве аргументов инструкциям передаются состояния, в которые должны вести переходы, исходящие из вершины. Таким образом, инструкции READ и POP имеют два аргумента, инструкции PUSHA и PUSHB - один аргумент, а инструкции REJECT и ACCEPT не имеют аргументов.

Вершины автомата создаются и нумеруются в порядке вычисления выражения. В процессе исполнения поддерживается специальная переменная LASTSTATE, соответствующая номеру последнего созданного состояния. Значение переменной может быть передано в инструкции языка с помощью макроса TOLAST. Макрос DEC уменьшает значение переменной LASTSTATE на единицу.

С помощью разработанного подхода в указанной работе удалось построить автомат, распознающий язык правильных скобочных последовательностей, и автомат, распознающий язык a^nb^n. Заметим, что язык a^nb^n является подмножеством языка правильных скобочных последовательностей.

1.6. Проектирование логических схем для автоматов Мили

При аппаратной реализации автоматов Мили возникает задача построения логической схемы, которая реализует данный автомат [18]. Эта задача обычно разбивается на ряд подзадач:

- минимизация число состояний автомата;
- кодирование состояний – сопоставление битовой строки, как номера, каждому состоянию;

- минимизация логической схемы;
- физическая реализация спроектированной схемы.

1.6.1. Задача кодирования состояний

Приведем пример из указанной работы, иллюстрирующий важность решения задачи кодирования состояний для минимизации числа логических элементов схемы, а, следовательно, площади ее физической реализации и стоимости изготовления. Задача состоит в том, чтобы сопоставить каждому состоянию автомата, имеющего N состояний, номер в диапазоне $[0..2^K - 1]$, где K – минимальное натуральное число, при котором $2^K \geq N$.

Табл. 1 задает некоторый автомат Мили.

Таблица 1. Таблица, задающая автомат Мили

Текущее состояние	Следующее состояние		Выходное воздействие	
Входное воздействие	0	1	0	1
q_0	q_0	q_0	0	0
q_1	q_2	q_2	0	1
q_2	q_0	q_0	1	0
q_3	q_2	q_2	1	1

Если произвести кодирование состояний автомата следующим образом: $\{s_0 = 00, s_1 = 11, s_2 = 01, s_3 = 10\}$, то будет построена схема, приведенная на рис. 6. Она содержит три элемента INV, пять элементов AND, три элемента OR и два D-триггера.

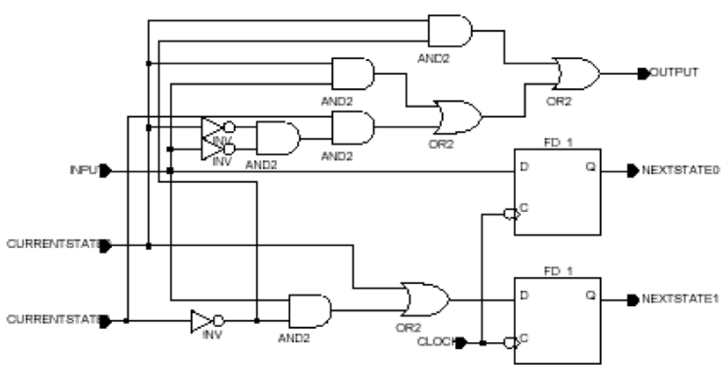

Рис. 6. Реализация автомата Мили при первом варианте
кодирования состояний

Если же закодировать состояния иначе: $\{s_0 = 00, s_1 = 10, s_2 = 01, s_3 = 11\}$, то возможна более экономичная реализация автомата (рис. 7), содержащая всего два элемента `INV`, пять элементов `AND`, два элемента `OR` и два `D`-триггера.

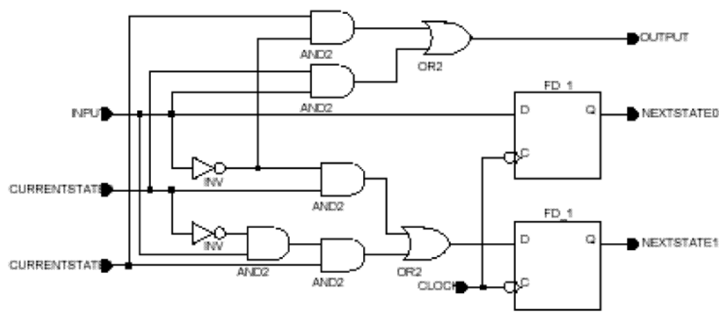

Рис. 7. Более экономичная реализация автомата Мили при втором варианте
кодирования состояний

Наивным решением задачи кодирования состояний является перебор всех возможных способов кодирования, построение минимальной схемы при заданном способе кодирования, выбор способа кодирования, при котором достигается минимизация числа элементов схемы. Однако данный подход имеет экспоненциальную сложность по числу состояний, в силу того, что такой сложностью обладает перебор всех возможных нумераций. Более того, установлено, что задача нумерации состояний (*State Assignment Problem*) является NP-полной.

В рассматриваемой работе применяется генетический алгоритм для решения задачи кодирования состояний. Особь генетического алгоритма, задающая решение

задачи, кодировалась в хромосому в виде строки длины N над целыми числами из диапазона $[0..2^K - 1]$. Пример кодирования особи указанным образом для автомата с шестью состояниями приведен в табл. 2.

Таблица 2. Таблица, задающая автомат Мили

S_0	S_1	S_2	S_3	S_4	S_5
4	2	1	0	7	6

Для скрещивания использовалась одноточечная, двухточечная и однородная рекомбинация строк. Мутация осуществлялась путем замены номера некоторого состояния (в позиции от 0 до N – 1) на другой допустимый номер. Как в случае мутации, так и в случае скрещивания могла получиться хромосома, определяющая некорректный способ кодирования, когда два состояния кодировались одним числом, в то время как некоторому состоянию не был сопоставлен никакой номер. В этом случае особь «штрафовалась», что препятствовало осуществлению недопустимых операций в дальнейшем.

Для вычисления функции приспособленности использовалась эвристика, которая дает хорошие результаты для задачи кодирования состояний, описанная в работе [19]. Данная эвристика формулируется следующим образом:

- два состояния a, b, для каждого из которых существует переход в состояние c, должны быть закодированы «близкими» значениями;
- два состояния a, b, в каждое из которых существует переход из состояния c, должны быть закодированы «близкими» значениями;
- у первого из указанных правил имеет приоритет над вторым.

«Близость» номеров состояний определяется расстоянием Хемминга между битовыми строками, задающими эти номера: номера, битовые строки которых отличаются не более чем в одной позиции, считаются «близкими», иначе не считаются таковыми. Например, номера 0101 и 1101 являются «близкими», в то время как номера 1100 и 1111 такими не являются. Для вычисления функции

приспособленности подсчитывалось число случаев x, противоречащих первому требованию, и число случаев y, противоречащих второму требованию. Если при этом хромосома соответствовала недопустимому способу кодирования (по описанным выше причинам), то ей назначался «штраф» – некоторое натуральное число z. При этом функция приспособленности принималась равной $f = 2x + y + z$.

Таким образом, наиболее «хорошие» особи имели малые значения функции приспособленности, а задачей генетического алгоритма являлось нахождение особи с минимальным значением функции приспособленности.

В результате проведенных экспериментов авторам рассматриваемой работы удалось на некоторых известных контрольных задачах (*benchmark*) получить значения функции приспособленности меньшие, чем те, что были получены известными ранее методами. В табл. 3 в столбце, обозначенном *Our GA*, приведены полученные результаты. В строках таблицы приведены сравнительные результаты по *benchmark*.

Таблица 3. Сравнение полученных решений с известными ранее

State machine	#AdjRes	Our GA	GA [5]	NOVA1	NOVA2
Shiftreg	24	0	0	8	0
Lion9	69	21	27	25	30
Train11	57	18	19	23	28
Bbara	225	127	130	135	149
Dk14	137	68	75	72	76
Bbsse	305	203	215	220	220
Donfile	408	241	267	326	291

Неудивительно, что программа авторов рассматриваемой статьи показала столь высокие результаты, если учесть, что другие подходы задавались лишь целью решить исходную задачу (кодирования состояний), а не минимизировать функцию приспособленности, определенную на основе эвристики решения исходной задачи.

В рассматриваемой работе также было предложено решение задачи минимизации логической схемы. Число элементов является лишь одним из критериев минимизации логической схемы. Рассмотренная задача кодирования состояний ориентирована именно на этот критерий. Другим критерием является минимизация

времени обработки сигнала. Различные логические элементы обладают различным временем обработки сигнала (задержки). В табл. 4 приведено время задержки для некоторых логических элементов, вместе с обозначениями самих элементов.

Таблица 4. Время задержки логических элементов

Name	Symbol	Gate Code	Gate Equiv.	Delay
NOT		0	1	0.0625
AND		1	2	0.209
OR		2	2	0.216
XOR		3	3	0.212
NAND		4	1	0.13
NOR		5	1	0.156
XNOR		6	3	0.211
MUX		7	3	0.212

Таким образом, в указанной работе решалась задача построения схемы, которая:

- соответствует автомату Мили;
- имеет минимальную площадь;
- имеет ограниченное время обработки сигнала.

Хромосома, кодирующая схему, задавалась матрицей ячеек. Каждая из ячеек содержала два или три (для элемента MUX) входных сигнала, логический элемент и выходной сигнал. Выходные сигналы ячеек предыдущего уровня являлись входными для ячеек следующего уровня. В соответствии с описанной структурой, ячейка кодировалась вектором целых чисел, соответствующих входным сигналам, логическому элементу и выходному сигналу. Пример хромосомы и соответствующей ей логической схемы приведен на рис. 8.

$\langle 1, 0, 2, 8\rangle$	$\langle 5, 10, 9, 12\rangle$	$\langle 7, 13, 14, 11, 16\rangle$
$\langle 2, 4, 3, 9\rangle$	$\langle 1, 8, 10, 13\rangle$	$\langle 3, 11, 12, 17\rangle$
$\langle 3, 1, 6, 10\rangle$	$\langle 4, 9, 8, 14\rangle$	$\langle 7, 15, 14, 15, 18\rangle$
$\langle 7, 5, 7, 7, 11\rangle$	$\langle 4, 10, 11, 15\rangle$	$\langle 1, 11, 15, 19\rangle$

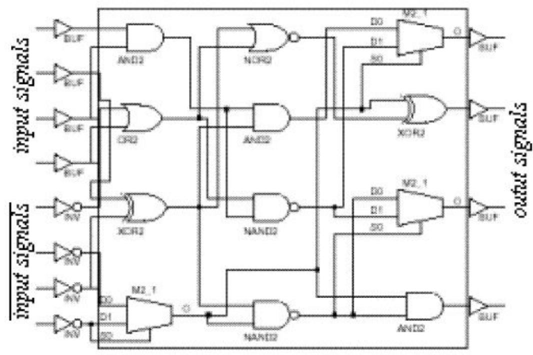

Рис. 8. Пример хромосомы, кодирующей схему

В качестве оператора скрещивания использовалась четырехточечная рекомбинация матриц. Оператор мутации изменял либо входные сигналы, либо логический элемент. При вычислении функции приспособленности учитывались с определенными весами число элементов в схеме и время обработки сигнала.

В работе [18] отмечено, что схемы, сгенерированные с помощью описанного выше эволюционного алгоритма, на ряде контрольных задач имели лучшие характеристики, нежели схемы, полученные другими методами.

1.7. Обзор методов генерации автоматов с помощью генетических алгоритмов

Методы генерации конечных автоматов можно разделить на несколько групп, в зависимости от того какой класс генетических алгоритмов используется:

- *Эволюционные алгоритмы* (по Л. Фогелю) [3] предполагают, что новые особи появляются только за счет операторов мутации. Алгоритмы данного класса не требуют много памяти для выполнения, однако работают они довольно медленно и легко могут «застрять» в локальном экстремуме.

- *Генетические алгоритмы* используют не только операторы мутации, но и операторы скрещивания. Генетические алгоритмы сходятся существенно быстрее, чем эволюционные, однако для их работы требуется популяция большого размера.
- *Генетическое программирование* является подклассом генетических алгоритмов, в котором особи кодируются с помощью деревьев решений.

Важную роль при разработке генетического алгоритма для конкретной задачи играет выбор способа кодирования особи. В тех случаях, когда решением задачи является автомат, наиболее часто используется один из следующих способов кодирования: битовые строки, граф переходов или дерево решений. В данной работе автоматы задаются графами переходов. Способ кодирования определяет, какие именно операторы скрещивания и мутации требуется использовать.

Правильный выбор типа кодирования и операторов скрещивания и мутации может значительно увеличить эффективность генетического алгоритма.

Необходимо отметить, что выбор функции приспособленности также играет важную роль. Например, функция приспособленности может быть заданной явно – возвращать вещественное значение, зависящее только от определенной особи и постоянное для этой особи. В качестве примера для такой функции может служить функция, получающая на вход битовую строку, задающую некоторый алгоритм, и возвращающая число, соответствующее эффективности этого алгоритма. В этом случае приспособленность особи определяется ее способностью решать поставленную задачу, например, отслеживать путь [15].

В тех случае, когда структуры, использующиеся для кодирования особей, имеют переменную длину, может потребоваться найти такое решение поставленной задачи, длина которого будет минимальной или хотя бы ограниченной разумными пределами. Это не сложно сделать, если включить в функцию приспособленности размер структуры, использующейся для кодирования особи. Пусть f_i определяет на сколько

хорошо i особь решает поставленную задачу и $length_i$ – размер структуры, использующейся для кодирования i особи. В качестве функции приспособленности можно использовать такую функцию F_i, что

1. $F_i < F_j$ для любых i и j, при которых $f_i < f_j$;
2. $F_i < F_j$ для любых i и j, при которых $f_i = f_j$ и $length_i < length_j$;
3. $F_i = F_j$ для любых i и j, при которых $f_i = f_j$ и $length_i = length_j$.

Построенная таким образом функция приспособленности, в первую очередь, отдает предпочтение тем особям, которые лучше решают поставленную задачу. При этом она также учитывает и размер структуры, использующейся для кодирования особи. Примером задачи, в которой была бы полезна такая функция приспособленности, является любая задача, в которой решение задано в виде конечного автомата. Как правило, желательно построить автомат с минимальным числом состояний. Число состояний автомата можно использовать вместо длины строки, если особь задается с помощью графа переходов.

Выводы по главе 1

1. Выполнен обзор задач решением, которых являются автоматы, генерирующиеся с помощью генетических алгоритмов.
2. Часто построение вручную автоматов, которые можно построить с помощью эволюционных алгоритмом, является крайне сложной, а, возможно, и невыполнимой задачей.
3. Выполнен обзор методов генерации конечных автоматов с помощью генетических алгоритмов для рассмотренных задач.
4. Выполненный обзор показал:

- имеющие место в прошлом ограничения вычислительной техники существенно ограничивали число особей в поколении и не позволяли решать задачи со сложными функциями приспособленности;
- для эффективной работы генетического алгоритма требуются операторы скрещивания и мутации, учитывающие особенности задачи. В случае, когда решением является конечный автомат, стандартных операторов, часто оказывается недостаточно.

5. Из изложенного следует, что требуется разработать новые методы, которые обеспечивают повышение эффективности работы генетических алгоритмов для генерации автоматов.

ГЛАВА 2. ГЕНЕРАЦИЯ АВТОМАТОВ ДЛЯ РЕШЕНИЯ ЗАДАЧИ О ФЛИБАХ

Задача о флибах – это задача моделирования элементарного живого существа, которое способно предсказывать имеющие периодичность изменения простейшей окружающей среды. Это существо моделируется конечным автоматом, а генетические алгоритмы позволяют автоматически построить автомат, который предсказывает изменения среды с достаточно высокой точностью. Таким образом, при решении данной задачи требуется построить «устройство» (предсказатель), которое предсказывает изменения среды с наибольшей вероятностью.

В работе [20] для решения указанной задачи используется один из генетических алгоритмов. Однако точность предсказания построенного автомата является недостаточно высокой. Это во многом связано с методом, используемым для построения следующего поколения особей (автоматов).

2.1. Постановка задачи

Одна из важнейших способностей живых существ – предсказание изменений окружающей среды. В качестве простейшей модели живого существа можно использовать конечный автомат. В работе [20] такие конечноавтоматные модели названы *флибами* (сокращение от *finite living blobs* – конечные живые капельки).

На вход флиба подается переменная, которая принимает одно из двух значения – ноль или единица. Эта переменная соответствует состоянию 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 изменений состояний среды. При этом считается, что чем точнее флиб предсказывает изменения среды, тем больше у него шансов выжить и оставить потомство.

2.2. Схемы работы генетических алгоритмов в задаче о флибах

Приведем схемы работы известного и предлагаемого алгоритмов.

2.2.1. Известный алгоритм

Для поиска оптимального предсказателя в работе [20] предлагается использовать 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 необходимо, то указанный оператор применяется к флибу.
6. Худший предсказатель в поколении заменяется флибом, полученный в результате скрещивания. 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 пункту 2.

Алгоритм создания случайных флибов, а также работа операторов одноточечного скрещивания и мутации, будут подробно рассмотрены в разд. 2.3, 2.6, 2.7.

2.2.2. Предлагаемая модификация алгоритма

В настоящей работе предлагается применять метод генерации нового поколения, отличающийся от метода, описанного в разд. 2.2.1. Предлагаемый метод использует турнирный отбор [21] и принцип элитизма (в новое поколение добавляется одна или несколько лучших особей из предыдущего поколения) [22].

Число флибов в поколении будем называть размером поколения.

Предлагаемая модификация алгоритма 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 выбираются две пары флибов.
   c. Из каждой пары флибов выбирается лучший предсказатель.
   d. Лучшие предсказатели из указанных пар *скрещиваются*.
   e. Случайным образом определяется необходимость применения *оператора мутации* к полученному флибу. Если это необходимо, то указанный оператор применяется к флибу.
   f. К флибу применяется *новый оператор мутации* – «восстановление связей между состояниями автомата».
   g. Флиб добавляется в новое поколение.
   h. Переходим к пункту b, если размер нового поколения меньше размера текущего поколения.
4. Текущее поколение флибов заменяется новым.
5. Если число поколений меньше заданного пользователем, то переходим к пункту 2.

2.3. Реализация флиба

Опишем известный и предлагаемый методы реализации флиба.

2.3.1. Известный метод

Поведение флиба описывается с помощью таблицы переходов и выходов. В табл. 5 в качестве примера приведен флиб с тремя состояниями A, B и C.

Таблица 5. Таблица переходов и выходов для флиба с тремя состояниями

Состояние	Значение входной переменной			
	0		1	
	Значение выходной переменной	Следующее состояние	Значение выходной переменной	Следующее состояние
A	1	B	0	A
B	0	C	0	A
C	1	A	0	B

Представим эту таблицу в виде строки: `1B0A0C0A1A0B` (такое представление используется в работе [20]). Отметим, что число элементов в приведенной строке в четыре 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, содержится 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, заданного 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. Если номер элемента нечетный, то элементу присваивается одно из возможных состояний флиба, выбранное случайным образом.

2.3.2. Предлагаемый метод

В настоящей работе предлагается использовать другой способ кодирования флиба – автомат, заданный с помощью графа переходов, который реализуется 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 число правильно предсказанных символов. Метод `Nulling` возвращает флиб в начальное состояние, и обнуляет число правильно предсказанных символов.

В массиве `_branches` класса `State` содержатся 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 число значений, которые может принимать выходная переменная, генерируемая флибом.

Опишем работу алгоритма создания случайного флиба.

1. Создаются объекты, соответствующие состояниям флиба.
2. Для каждого объекта выполняется следующее:
   - для состояния формируются переходы из него.

Для каждого перехода случайным образом определяются номер состояния, в которое переходит флиб и значение выходной переменной.

2.4. Генератор значений входной переменной

В качестве значений входной переменной для флибов (как и в работе [20]) 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 используется метод DoStep. Этот метод необходимо вызвать перед началом использования генератора входного сигнала. Метод Nulling сбрасывает генератор входного сигнала в начальное состояние.

2.5. Функция приспособленности

Лучшим предсказателем является тот флиб, который наиболее точно предсказывает входной сигнал. В качестве функции приспособленности используется число 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 входных символов (по умолчанию их число равно 100). После этого можно строить новое поколение решений, используя в качестве приспособленности флиба значение поля `_guessCount`, содержащее число правильно предсказанных символов.

2.6. Оператор одноточечного скрещивания

Опишем известный и предлагаемый алгоритмы работы оператора одноточечного скрещивания.

2.6.1. Известный алгоритм

В работе [20] формирование нового флибов из двух существующих 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, задающей первый родительский флиб, копируются в строку, задающую новый флиб.
3. Элементы с номерами большими `j` из строки, задающей второй родительский флиб, копируются в строку, задающую новый флиб.

2.6.2. Предлагаемый алгоритм

Для применения оператора одноточечного скрещивания для флиба, закодированного в виде графа переходов, алгоритм, описанный в работе [20], требуется модифицировать. 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. 2.6.1

Реализация изложенного алгоритма приведена в Приложении (Листинг 3).

2.7. Оператор мутации

Опишем известный и предлагаемый алгоритмы работы оператора мутации.

2.7.1. Известный алгоритм

Алгоритм работы оператора мутации для флиба, заданного битовой строкой, который используется в работе [20], 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.

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

2.7.2. Предлагаемый алгоритм

В настоящей работе для флиба, поведения которого описывается автоматом, заданным графом переходов, предлагается использовать следующий алгоритм работы оператора мутации.

1. Случайным образом выбирается состояние флиба.
2. Случайным образом выбирается дуга из выбранного состояния флиба.

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 было определено, что изменяется значение выходной переменной, то ей присваивается значение состояния среды, выбранное случайным образом.
    b. Если было определено, что изменяется 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).

2.8. Алгоритм восстановления связей между состояниями

При генерации генетическим алгоритмом нового поколения решений, применении операторов скрещивания и мутации, переходы в автоматах изменяются случайным образом. При таком изменении переходов, в автомате, как правило, возникают состояния, в которые невозможно попасть из начального состояния при любой последовательности значений входных переменных. Будем называть такие состояния *недоступными*. Состояния, в которые можно попасть из начального состояния при некоторой последовательности значений входных переменных, будем называть *доступными*. Оператор восстановления связей между состояния изменяет переходы в автомате таким образом, чтобы в нем не было недоступных состояний.

Рассмотрим работу оператора восстановления связей между состояниями на примере задачи о флибах. Класс `FlibRestorer`, реализующий алгоритм восстановления связей между состояниями, приведен в Приложении (Листинг 5). Опишем алгоритм реализации предлагаемого оператора мутации.

1. Формируется список доступных состояний (метод `InitIndexesList`). Для этого можно, например, использовать функцию рекурсивного обхода графа `AddIndex`.
2. Выполняется цикл по всем состояниям. Если текущее состояние не входит в число доступных состояний, то для него выполняются следующие операции:
   a. Случайным образом выбирается состояние из списка доступных состояний.
   b. Выбирается случайным образом один переход из выбранного состояния.
   c. В текущем состоянии заменяется один переход из этого состояния на переход, который ведет в то же состояние, что и переход, выбранный в пункте b.
   d. Выбранный в пункте b переход заменяется переходом, который ведет в текущее состояние.
   e. Обновляется список доступных состояний. В него добавляется текущее состояние и все состояния, в которые можно попасть из него.

Рассмотрим работу описанного выше алгоритма на примере. На рис. 9 изображен граф переходов для флиба полученного в результате работы генетического алгоритма. Начальное состояние автомата – **A**.

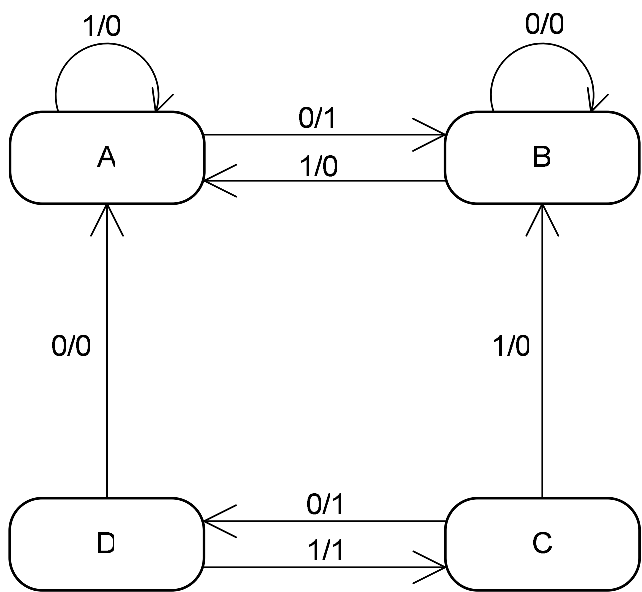

Рис. 9. Граф переходов для флиба полученного в результате работы генетического алгоритма

Для автомата, граф переходов которого изображен на рис. 9, состояния **A** и **B** являются доступными состояниями, в то время как состояния **C** и **D** – недоступные. На рис. 10 доступные состояния закрашены светло-серым цветом. Недоступные состояния изображены темно-серыми.

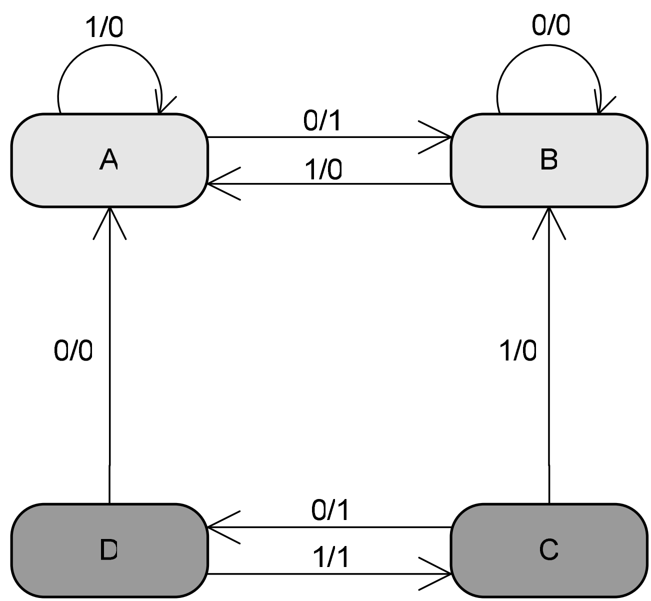

Рис. 10. Доступные (светло-серые) и недоступные (темно-серые) состояния автомата

Алгоритм восстановления связей между состояниями перебирает все недоступные состояния. Первым идет состояние **C**. Случайным образом выбирается состояние из списка доступных состояний (например, состояние **B**). Выбирается случайным образом переход из состояния **B**. В качестве примера выберем переход, который соответствует единичному значению входной переменной и ведет в состояние **A**. Он заменяется на переход, соответствующий такому же значению выходной переменной, и ведет в состояние **C**. Один из переходов, который ведет из состояния **C** (для примера возьмем переход, соответствующий единичному значению входной переменной), заменяется аналогичным переходом, ведущим в состояние **A**.

На рис. 11 показан результат добавления состояния **C** в список доступных состояний. Серым пунктиром показаны переходы, которые были удалены в результате этой операции. Черным пунктиром обозначены новые переходы. В состояние **C** автомат попадет из начального состояния **A**, если на его вход, например, будет подана последовательность значений входной переменной вида `01`.

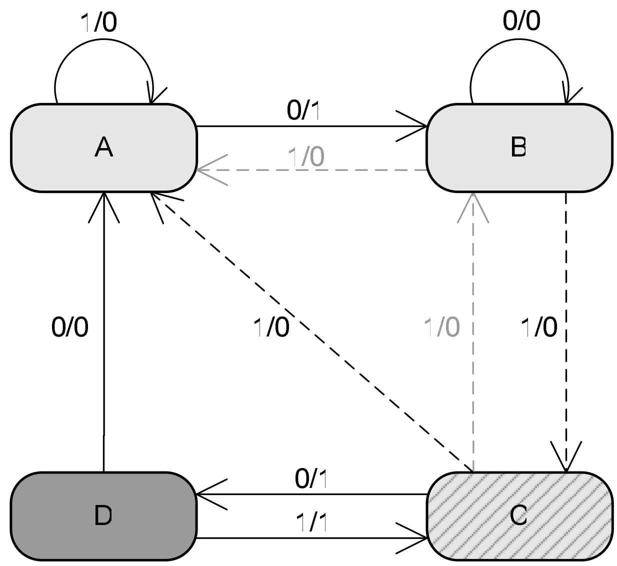

Рис. 11. Добавление состояние **C** в список доступных состояний

После замены переходов состояние **C** стало доступным. В состояние **D** можно попасть из состояния **C.** Следовательно, состояние **D** также стало доступным. Действительно, в состояние **D** можно перейти из начального состояния **A**, если подать, например, последовательность значений входной переменной вида `0011`.

Так как после обновления списка доступных состояний все состояния стали доступными, то алгоритм завершает свою работу. Граф переходов для автомата, получившегося в результате работы алгоритмам восстановления связей между состояниями, изображен на рис. 12.

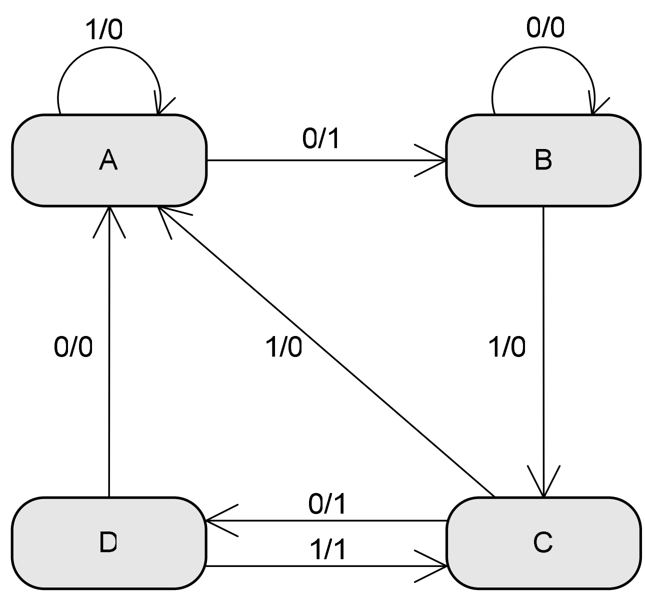

Рис. 12. Результат работы алгоритма восстановления связей между состояниями

Алгоритм восстановления связей между состояниями можно использовать как в качестве дополнительного, так и в качестве основного оператора мутации. Иногда может оказаться целесообразным проверять, не ухудшилась ли приспособленность особи после выполнения восстановления связей между состояниями. Если приспособленность особи ухудшилась, то граф переходов возвращается в исходное состояние.

Часто может оказаться целесообразным увеличить число доступных состояний автомата, оставив часть состояний недоступными. Такой подход требует меньше вычислительных ресурсов и вносит меньше изменений в поведение автомата.

Использование алгоритма восстановления связей между состояниями позволяет увеличить среднее число состояний в автоматах поколения. Чем больше состояний у автомата, тем меньшему числу изменений он подвергается и, как следствие, тем меньше меняется его поведение. В итоге генетический алгоритм начинает отдавать предпочтение автоматам с большим числом состояний и соответственно с более сложным поведением. Это полезно если приближенным решением задачи является

автомат с небольшим числом состояний, а любое лучшее решение должно иметь значительно большее число состояний. В этом случае генетический алгоритм легко может застрять в локальном оптимуме. Эта проблема возникает из-за того, что стандартный оператор мутации (изменяет значение выходной переменной на переходе или состояние, в которое осуществляется переход) при однократном применении крайне редко может значительно увеличить число состояний в автомате. Алгоритм восстановления связей между состояниями решает эту проблему.

2.8.1. Программа для проведения экспериментов на флибах

По рассмотренным алгоритмам (известному и предлагаемому) написана программа, окно пользовательского интерфейса которой приведено на рис. 13.

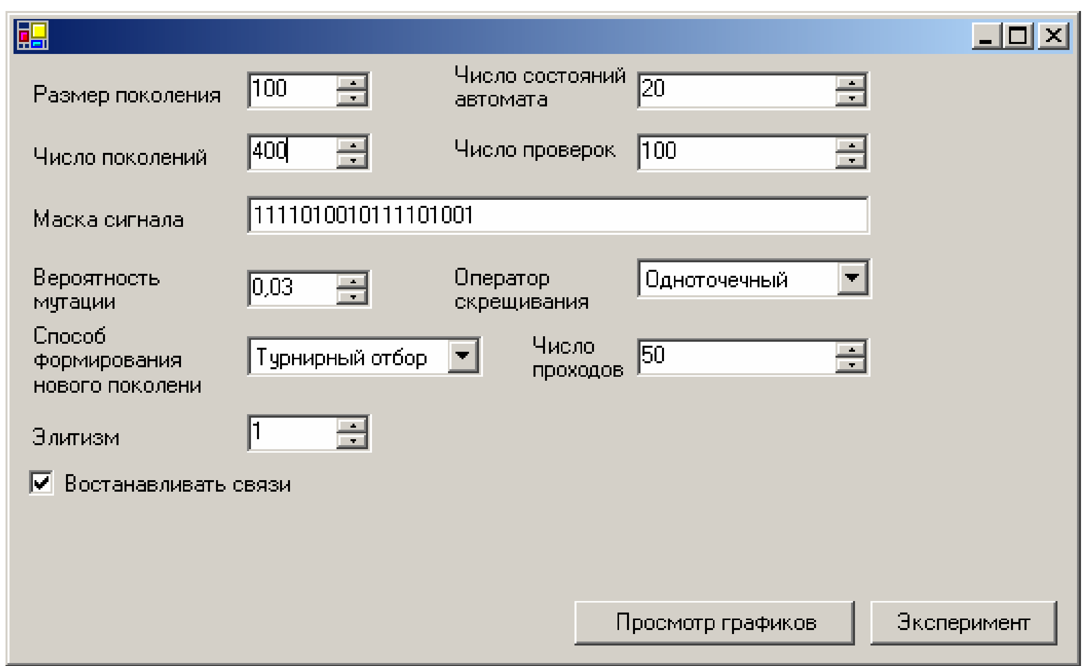

Рис. 13. Окно пользовательского интерфейса

Программа позволяет реализовать известный и предлагаемый алгоритмы с помощью выпадающего списка «Способ формирования нового поколения». При этом в предлагаемом алгоритме может быть изменен компонент стратегии отбора «элитизм» в диапазоне от нуля до размера поколения. Программа позволяет выбрать размер

поколения и число поколений. Также может быть выбрана вероятность мутации (в диапазоне от нуля до единицы) и тип оператора скрещивания (одноточечный и двухточечный). В программе имеется возможность задавать битовую маску, определяющую среду. Для каждого флиба может быть задано число его состояний, и определено будет ли использоваться оператор восстановления связей между состояниями.

Для того чтобы сравнивать эффективность работы генетических алгоритмов принято многократно прогонять их на одинаковых наборах тестовых данных и сравнивать усредненные результаты. Поэтому в программе реализована возможность автоматического прогона предлагаемой модификации генетического алгоритма заданное число раз и получения усредненных результатов. Эта модификация может быть запущен на число проходов, которое экспериментатор хочет установить.

Приведенные в приложении листинги составляют ядро разработанной программы для предлагаемого подхода.

2.8.2. Эксперименты по применению алгоритма восстановления связей между состояниями

Эксперименты, описываемые в этом разделе, проводились с помощью предложенной модификации генетического алгоритма, решающего задачу о флибах. При формировании нового поколения применялся турнирный отбор [21] и элитизм (в новое поколение добавляется одна или несколько лучших особей из предыдущего поколения) [22]. Для формирования новой особи использовался одноточечный оператор скрещивания. Все эксперименты проводились при размере поколения 100 и вероятности применения одноточечного оператора мутации 0,03. Число воздействий среды на флиб выбрано равным 100. Поэтому число правильно предсказанных символов по величине равно точности предсказания символов в процентах. В таблицах с результатами экспериментов, которые приведены ниже, указаны минимальные, максимальные и средние точности предсказания автоматически построенных флибов.

На графиках ось абсцисс соответствует номерам поколений, а ось ординат – числу правильно предсказанных символов лучшим предсказателем в каждом поколении. При построении графиков использовались *усредненные данные*, полученные при проведении 50 экспериментов с одинаковыми начальными параметрами. Пунктир использован для графиков, соответствующих стандартному генетическому алгоритму. Для случая, когда применяется оператор мутации «восстановление связей между состояниями», графики построены непрерывными линиями.

Ниже приводятся результаты трех экспериментов, которые отличаются между собой выбранным числом поколений, битовой маской и числом состояний флиба.

2.8.2.1. Первый эксперимент

Эксперимент производился для 400 поколений. Битовая маска, задающая среду, имеет вид – `11110100101111101001`. Число состояний флиба – 20.

Графики для первого эксперимента приведены на рис. 14.

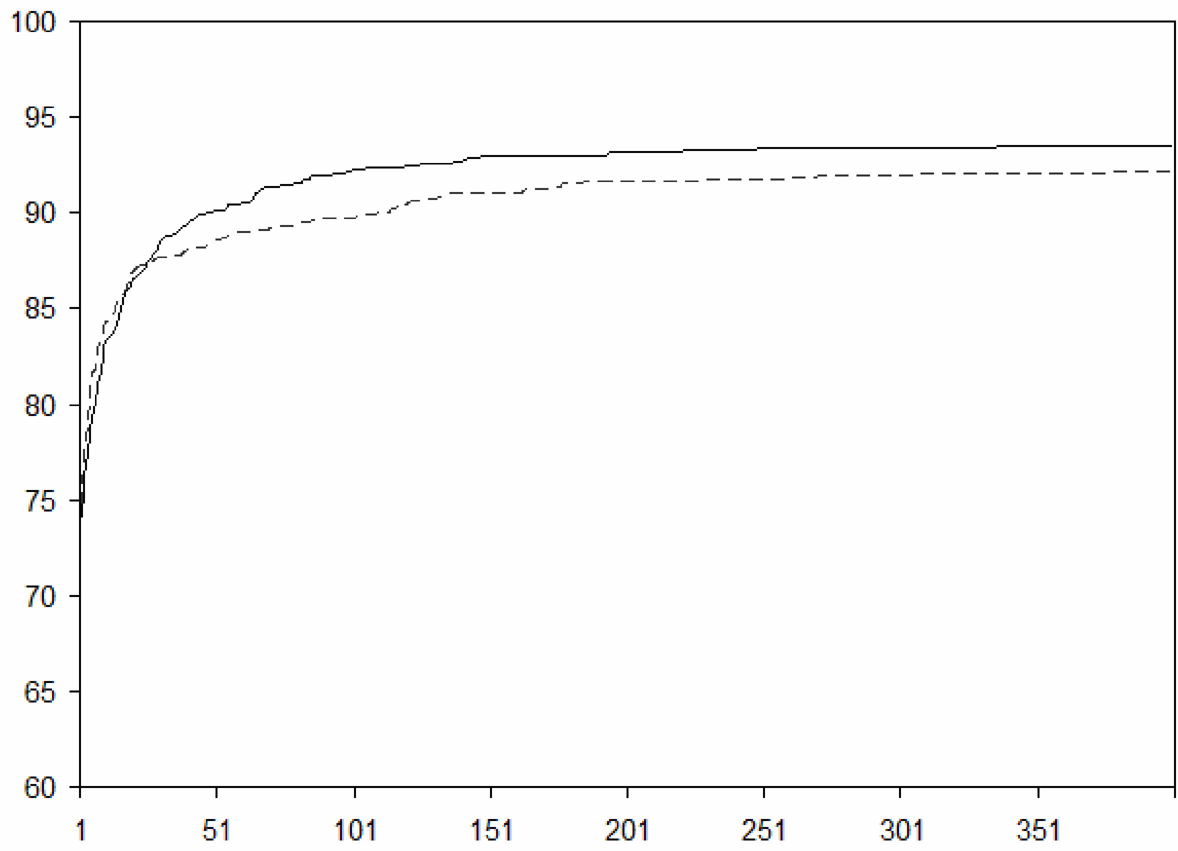

Рис. 14. Графики усредненных результатов для первого эксперимента по применению оператора мутации «восстановления связей между состояниями» (пунктир используется для стандартного генетического алгоритма)

Из приведенных графиков видно, что использование алгоритма восстановления связей между состояниями позволяет повысить среднюю эффективность генетического алгоритма.

Результаты первого эксперимента приведены в табл. 6.

Таблица 6. Результаты первого эксперимента по применению оператора мутации «восстановления связей между состояниями»

Восстановление связей между состояниями	Худший результат	Усредненный результат	Лучший результат
Нет	83	92,26	100
Да	84	93,48	100

2.8.2.2. Второй эксперимент

Эксперимент производился для 400 поколений. Битовая маска, задающая среду, имеет вид – `11110100101110`. Число состояний флиба – 10. Отметим, что битовая маска в этом эксперименте короче, чем в предыдущем.

Графики для второго эксперимента приведены на рис. 15.

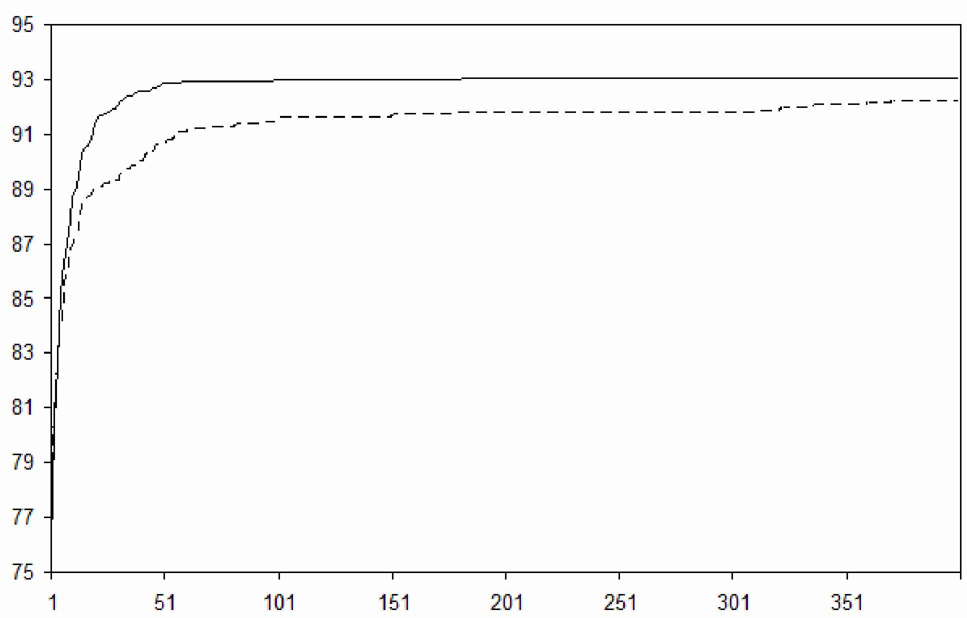

Рис. 15. Графики усредненных результатов для второго эксперимента по применению оператора мутации «восстановления связей между состояниями» (пунктир используется для стандартного генетического алгоритма)

Результаты второго эксперимента приведены в табл. 7.

Таблица 7. Результаты второго эксперимента по применению алгоритма оператора мутации «восстановления связей между состояниями»

Восстановление связей между состояниями	Худший результат	Усредненный результат	Лучший результат
Нет	86	92,2	94
Да	87	93,06	94

Значения, приведенные в столбце "Усредненный результат", соответствуют крайним правым точкам графиков на рис. 15.

2.8.2.3. Третий эксперимент

Эксперимент производился для 1600 поколений. Битовая маска, задающая среду, имеет вид – `10101110110001110111110011001`. Число состояний флиба – 30. Отметим, что битовая маска в этом эксперименте содержит большее число разрядов, чем в обоих предыдущих.

Графики для третьего эксперимента приведены на рис. 16.

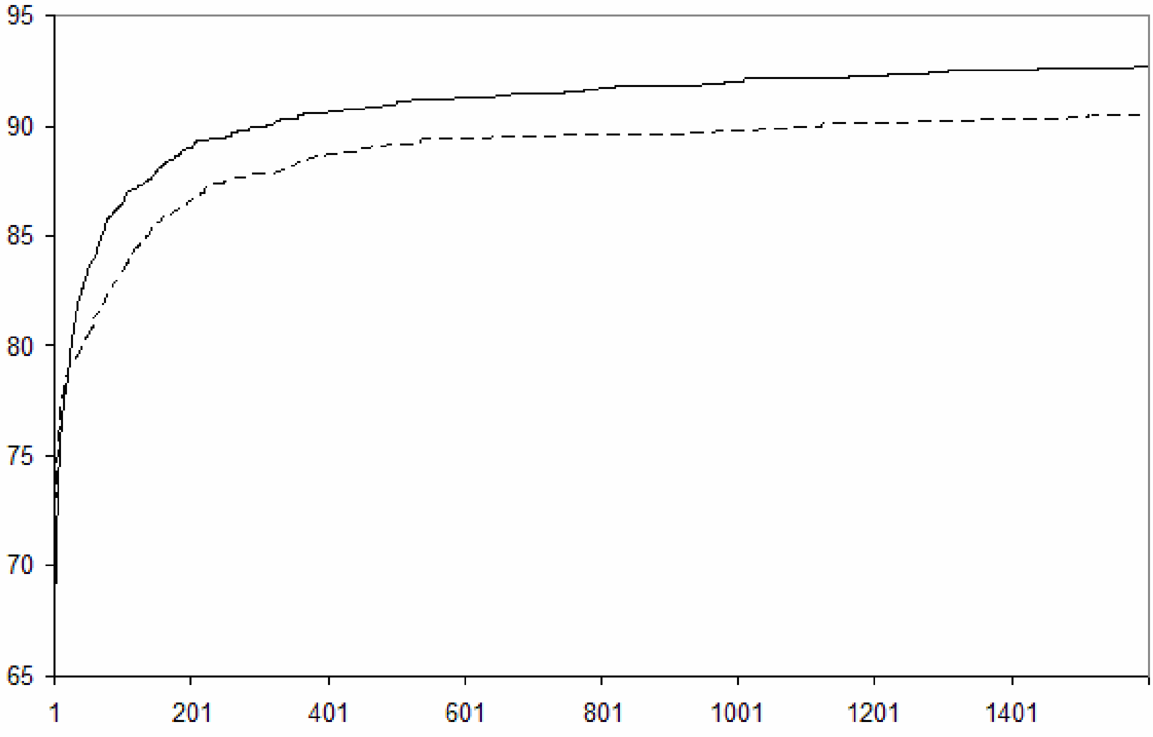

Рис. 16. Графики усредненных результатов для третьего эксперимента по применению оператора мутации «восстановления связей между состояниями» (пунктир используется для стандартного генетического алгоритма)

Результаты третьего эксперимента приведены в табл. 8. Выигрыш от применения оператора мутации «восстановление связей между состояниями» в данном эксперименте больше, чем в двух предыдущих, благодаря увеличению числа поколений и усложнению задачи.

Таблица 8. Результаты второго эксперимента по применению оператора мутации «восстановления связей между состояниями»

Восстановление связей между состояниями	Худший результат	Усредненный результат	Лучший результат
Нет	83	90,44	97
Да	85	92,72	97

Проведенные эксперименты показали, что оператор мутации «восстановление связей между состояниями» повышает эффективность работы генетического алгоритма как при различных битовых масках, задающих среду, так и при различном числе состояний флиба.

2.9. Использование автоматов с флагами

Поведение конечного автомата зависит от его состояний и переходов между ними. Каждый переход определяет в какое состояние перейдет автомат при некотором условии (функция, принимающая значения входных переменных и возвращающая булевское значение) и какие действия выполнятся при переходе.

Чем более сложную задачу решает автомат, тем, как правило, больше число его состояний. Назовем автоматом с флагами автомат, использующий вектор флаговых (задержанных) переменных `F`, который несет информацию о значениях предыдущих входных воздействий автомата. Если автоматы без флагов зависят от предыдущего состояния и входного воздействия, то автоматы с флагами зависят еще и от предыстории входных воздействий.

На рис. 17 изображена структурная схема для автомата без флагов.

Рис. 17. Структурная схема для автомата без флагов

Флаговые переменные вместе с входными переменными и текущим состоянием автомата определяют в какое состояние перейдет автомат в следующий момент времени и какие выходные переменные он сформирует. На рис. 18 изображена структурная схема для автомата с флагами.

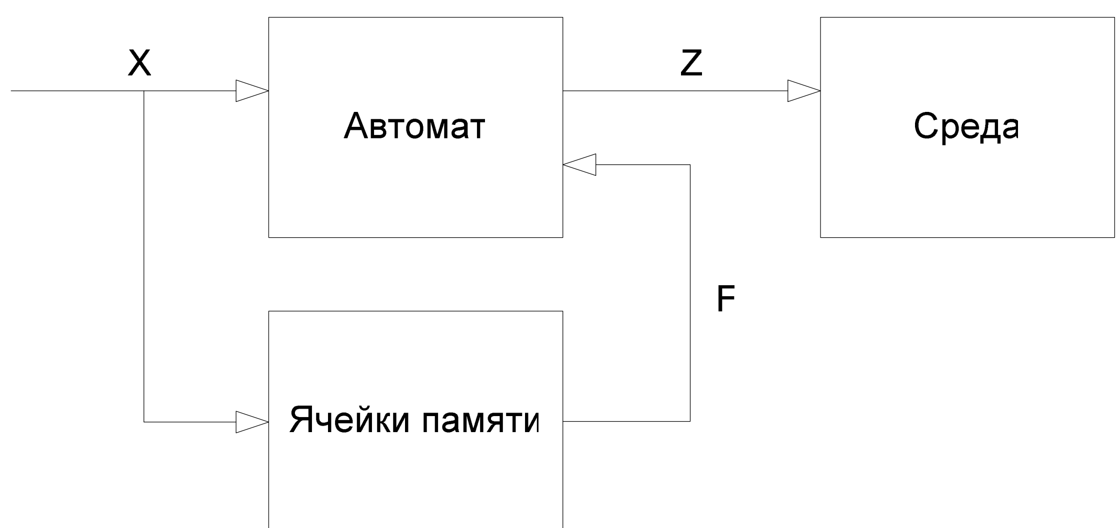

Рис. 18. Структурная схема для автомата с флагами

Использование флаговых переменных часто позволяет уменьшить число состояний в автомате. Это, правда, приводит к усложнению его графа переходов. При этом становится труднее понять алгоритм работы автомата. Однако, этот недостаток не является существенным для случая автоматической генерации автоматов.

Рассмотрим, как можно использовать автоматы с флагами при решении задачи о флибах [23]. Поведение флиба описывается с помощью таблицы переходов и выходов. На рис. 19 изображен граф переходов для флиба, таблица переходов и выходов для которого приведена в табл. 5.

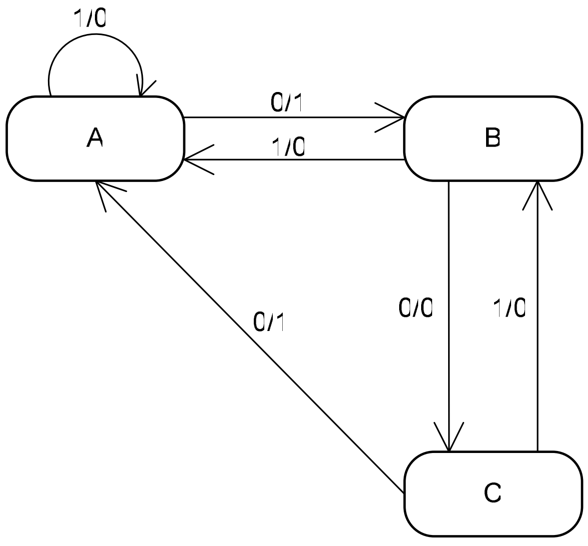

Рис. 19. Граф переходов для флиба с тремя состояниями

Во флибе флаговые переменные используются для хранения предыстории окружающей среды. Во флаговой переменной с номером один будем хранить состояние среды на предыдущий момент времени, а во флаговой переменной с номером n – состояние среды на n шагов назад.

Пусть x и $F = (f_1, \ldots, f_n)$ – состояние среды и вектор флаговых переменных в текущий момент времени соответственно. Тогда вектор флаговых переменных для следующего момента времени будет иметь вид $F' = (f_1' = x, f_2' = f_1, \ldots, f_n' = f_{n-1})$. При программной реализации флаговые переменные можно хранить в массиве. В этом случае переход от F к F' осуществляется с помощью удаления последнего элемента массива и вставки в начало массива текущего состояния среды x.

Введение каждой флаговой переменной увеличивает число переходов из каждого состояния флиба вдвое. При n флаговых переменных число переходов из каждого состояния флиба будет равно 2^{n+1}. Номер перехода, который должен использоваться в текущий момент времени, можно вычислить по формуле

$x + 2^1 f_1 + 2^2 f_2 + \ldots + 2^n f_n$. Номер перехода может быть выражен также записанным в двоичном виде числом $f_n \ldots f_2 f_1 x$.

Далее флиб, который моделируется с помощью автомата с флагами, будем называть флибом с флагами. В табл. 9 приведен пример таблицы переходов для флиба с тремя состояниями и одной флаговой переменной, которая на текущем шаге используется в качестве одной из входных переменных.

Таблица 9. Таблица переходов и выходов для флиба с тремя состояниями и одной флаговой переменной

Состояние	Значение входной переменной *s* и флаговой переменной *f*							
	f = 0, x = 0		f = 0, s = 1		F = 1, s = 0		f = 1, s = 1	
	Значение выходной переменной и следующее состояние		Значение выходной переменной и следующее состояние		Значение выходной переменной и следующее состояние		Значение выходной переменной и следующее состояние	
A	1	A	1	B	0	B	0	A
B	1	B	1	C	0	C	0	A
C	0	B	0	C	1	A	1	B

На рис. 20 изображен граф переходов для флиба, таблица переходов которого приведена в табл. 9.

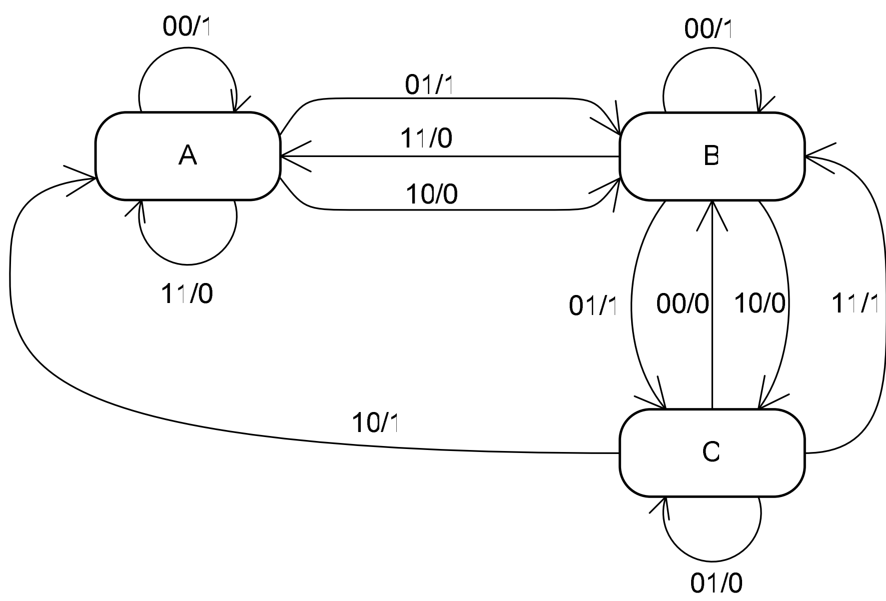

Рис. 20. Граф переходов для флиба с тремя состояниями и одной флаговой переменой

В общем случае, входная переменная может принимать произвольное число значений. Пусть существует m допустимых комбинации значений входных переменных, и все эти комбинации пронумерованы от 0 до $m - 1$. Тогда при n флаговых переменных число переходов из каждого состояния автомата будет равно m^{n+1}. Номер перехода, который должен использоваться в текущий момент времени, можно вычислить по формуле $x + m^1 f_1 + m^2 f_2 + \ldots + m^n f_n$, где x – номер текущей комбинации значений входных переменных, а $F = (f_1, \ldots, f_n)$ – вектор флаговых переменных.

Флаговые переменные позволяют существенно уменьшить число состояний в автомате. Рассмотрим эту особенность на простом примере. Пусть требуется построить флиб для простейшей среды, заданной с помощью повторяющейся битовой маски `11100`. При использовании обычного автомата потребуются, по крайней мере, три состояния, для того чтобы флиб смог предсказывать изменения среды со сто процентной точностью. Пример такого флиба приведен на рис. 21.

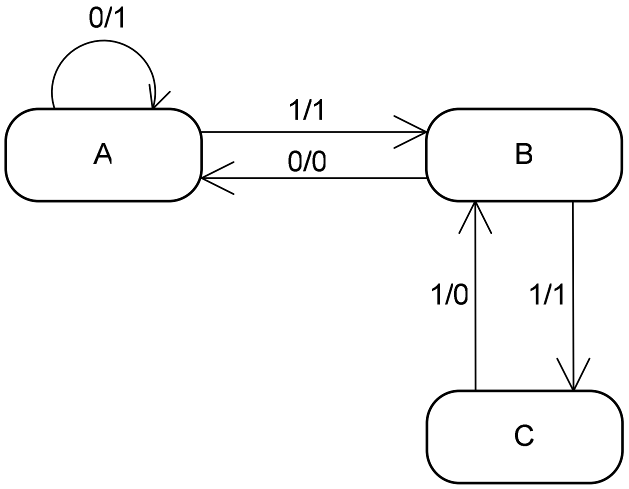

Рис. 21. Граф переходов флиба с одной входной переменной для среды заданной битовой маской 11100

Начальное состояние автомата **А**. Отметим, что для рассматриваемой маски среды один из переходов в состоянии **С** не определен.

При использовании одной входной и двух флаговых переменных достаточно одного состояния, для того чтобы построить флиб с точностью предсказания равной ста процентам. Граф переходов для такого флиба с флагами изображен на рис. 22. Как и в предыдущем случае, все неопределенные переходы были удалены, для того чтобы упростить рисунок.

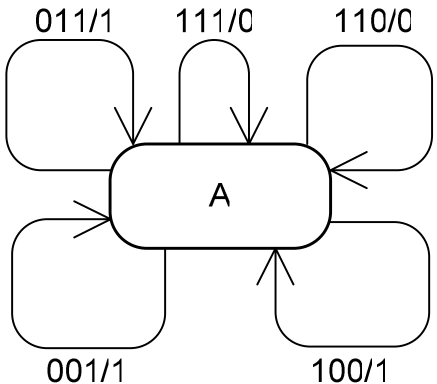

Рис. 22. Граф переходов флиба с одной входной и двумя флаговыми переменными для среды заданной битовой маской 11100

Как видно из приведенного примера, использование флаговых переменных позволяет уменьшить число состояний автомата, но увеличить число переходов между состояниями. Благодаря этому в автоматах уменьшается число *недоступных состояний*.

При n флаговых переменных число переходов из каждого состояния автомата будет равно m^{n+1}, где m – число допустимых комбинации значений входных переменных. Объем памяти, необходимой для хранения графа переходов, пропорционален числу переходов из каждого состояния. Создание новой особи из двух родительских также выполняется за время пропорциональное числу переходов из каждого состояния автомата. Для того чтобы генетический алгоритм перебирал решения для автоматов с флагами со скоростью не меньшей, чем для автоматов без флагов, число состояний в автоматах с флагами должно не больше $\frac{s}{m^{n+1}}$, где s – максимальное число состояний для автомата без флагов.

2.9.1. Эксперименты по использованию автоматов с флагами

Эксперименты, описываемые в этом разделе, проводились с помощью генетического алгоритма, решающего задачу о флибах. При формировании нового поколения применялся турнирный отбор [21] и элитизм [22]. Для формирования новой особи использовался одноточечный оператор скрещивания. Все эксперименты проводились при размере поколения 100 и вероятности применения одноточечного оператора мутации 0,03. Число воздействий среды на флиб выбрано равным 100. Благодаря этому число правильно предсказанных символов по величине равно точности предсказания символов в процентах. В таблицах с результатами экспериментов приводится минимальные, максимальные и средние точности предсказания автоматически построенных флибов. В качестве битовой маски использовалась строка `111101001011110100 1`.

На графиках по оси абсцисс отложены номера поколений, 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.

Чем больше состояний у автомата, тем более сложным может быть его поведение. Исходя из этого, можно попытаться улучшить точность предсказания с помощью увеличения числа состояний флиба. В табл. 10 приведены результаты экспериментов для флибов, заданных автоматом без флагов. Число состояний автоматов варьировалось в зависимости от эксперимента в диапазоне от 10 до 80.

Таблица 10. Результаты экспериментов над флибами без флагов с разным числом состояний

Число состояний	Худший результат	Усредненный результат	Лучший результат
10	89	93,76	95
20	90	94,2	95
40	89	94,58	100
80	89	96	100

Как видно из таблицы, увеличение числа состояний действительно несколько улучшает точность найденного предсказателя.

Графики для экспериментов над флибами с разным числом состояний приведены на рис. 23. Точками изображен график для результатов эксперимента над флибами с 10 состояниями. Штрихпунктирная линия используется для флибов с 20 состояниями, а пунктирная линия для флибов с 40 состояниями. График для результатов эксперимента над флибами, имеющими 80 состояний, изображен сплошной линией.

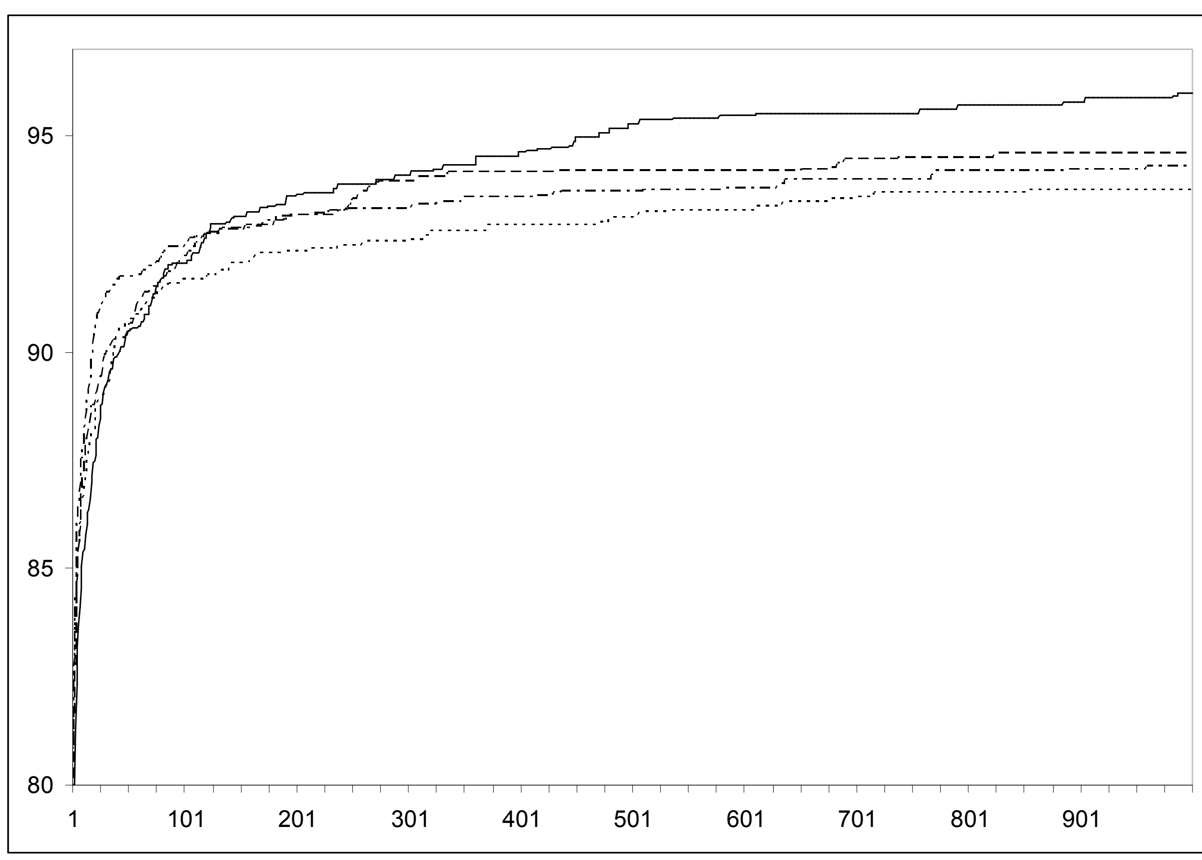

Рис. 23. Графики усредненных результатов для экспериментов над флибами без флагов с разным числом состояний

Основным недостатком увеличения числа состояний является рост объема вычислений и, как следствие, увеличение времени, необходимого для построения предсказателя.

Перейдем к рассмотрению флибов с флагами. В табл. 11 приведены результаты экспериментов над флибами с разным числом состояний и флагов.

Таблица 11. Результаты экспериментов над флибами с разным числом состояний и флагов

Число состояний	Число флагов	Худший результат	Усредненный результат	Лучший результат
80	0	89	96	100
40	1	90	96,82	100
20	2	95	98,54	100
10	3	95	99,74	100

Графики для экспериментов над флибами с разным числом состояний и флагов приведены на рис. 24. Точками изображен график для результатов эксперимента над флибами с 80 состояниями без флагов. Штрихпунктирная линия используется для флибов с 40 состояниями и одним флагом, а пунктирная линия – флибы с 20 состояниями и двумя флагами. График результатов эксперимента над флибами с 10 состояниями и тремя флагами изображен сплошной линией.

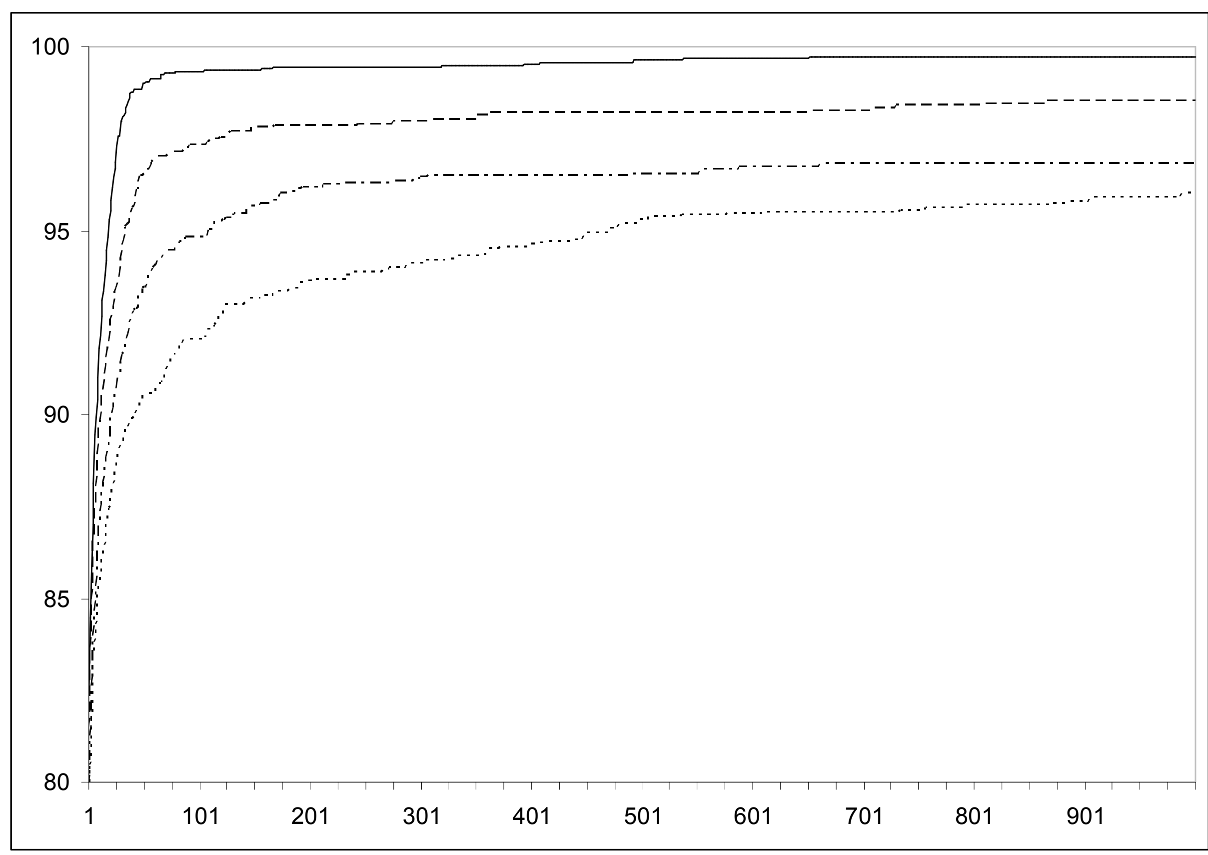

Рис. 24. Графики усредненных результатов для экспериментов над флибами с разным числом состояний и флагов

Необходимо отметить, что число состояний и флагов в этих экспериментах выбрано таким образом, что общее число дуг переходов во всех флибах остается одинаковым.

Из приведенных таблиц и графиков следует, что применение автоматов с флагами для моделирования флибов позволяет строить более точные предсказатели.

Эксперименты показывают, что применение автоматов с флагами является более эффективным методом для улучшения точности работы генетического алгоритма по сравнению с увеличением числа состояний автоматов.

Выводы по главе 2

1. Предложен новый оператор мутации для автоматов, заданных с помощью графов переходов – восстановление связей между состояниями. Принцип его работы рассмотрен на примере.
2. Разработана модификация генетического алгоритма, использующая предложенный оператор мутации.
3. Разработана программа, позволяющая сравнивать эффективность известного и модифицированного генетических алгоритмов.
4. Приведены результаты компьютерных экспериментов, которые показывают, что предложенный оператор мутации повышает эффективность работы генетического алгоритма как при различных битовых масках, задающих среду, так и при различном числе состояний флибов.
5. Для задачи о флибах выполнено сравнение эффективности генетических алгоритмов, которые позволяют генерировать автоматы Мили и автоматы Мили с флагами. Результаты компьютерных экспериментов, показывают, что точность предсказания флибом, который моделируется автоматом с флагами, выше.

ГЛАВА 3. ГЕНЕРАЦИЯ АВТОМАТОВ ДЛЯ ЗАДАЧИ ОБ УМНОМ МУРАВЬЕ

3.1. ПОСТАНОВКА ЗАДАЧИ

На тороидальном поле размером 32 на 32 вдоль определенной ломаной линии («тропы») расположено 89 ячеек с едой. Расположение ячеек с едой на «тропе» фиксировано. На рис. 25 приведено пример «тропы», называемый в литературе «тропой Джона Мьюра» (*John Muir Trail*) [24], которая образована черными и серыми ячейками. При этом каждая черная ячейка содержит одну единицу еды.

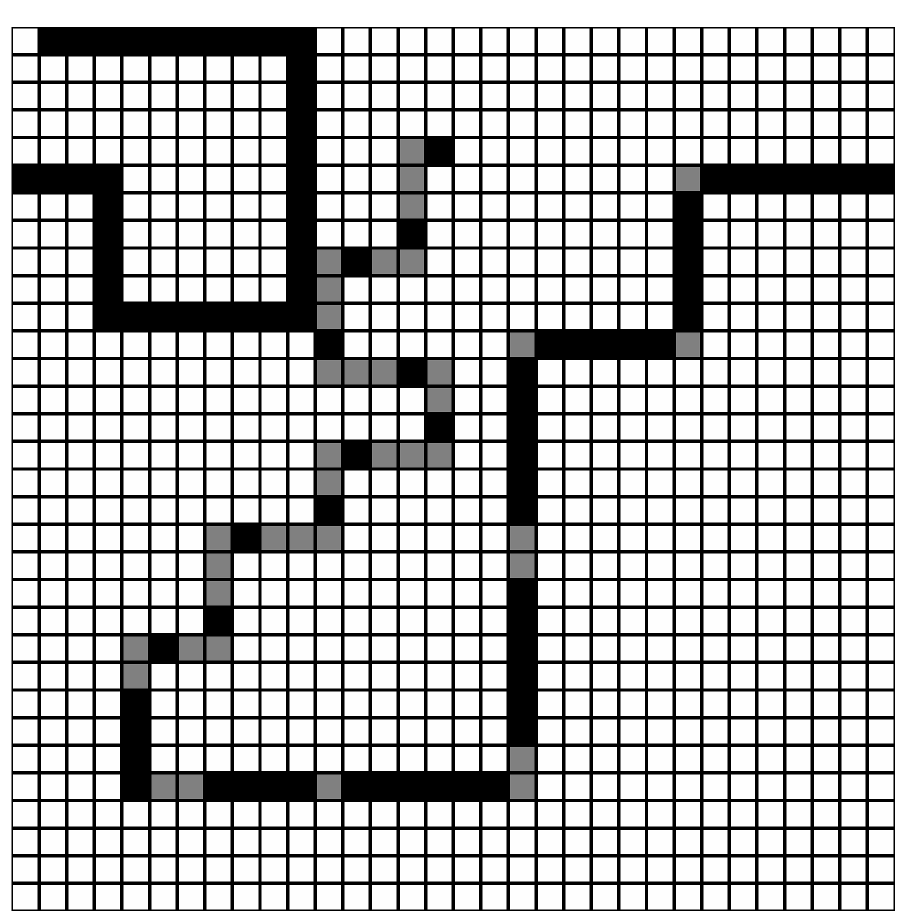

Рис. 25. Тропа Джона Мьюра (John Muir Trail)

Перед первым шагом муравей находится в верхней левой ячейке и смотрит направо. Он может определить есть ли еда в ячейке, находящейся непосредственно

перед ним. За ход муравей может сделать одно из следующих трех действий – подвинуться вперед, повернуться налево, повернуться направо.

Для того чтобы съесть еду в ячейке, муравью необходимо попасть в нее. После этого ячейка с едой считается пустой и становится неотличимой для муравья от изначально пустых ячеек. Муравей должен съесть всю еду не более чем за 200 ходов.

Муравей определяет значение входной битовой переменной (ноль – еды в ячейке перед муравьем нет, единица – еда в ячейке перед муравьем есть). Значение этой переменной используется в качестве входного воздействия для конечного автомата, моделирующего поведения муравья. После получения входного воздействия автомат генерирует выходную переменную (действие, которое может осуществить муравей – *вперед*, *налево* или *направо*) и переходит в новое состояние.

Пример автомата с четырьмя состояниями, моделирующего поведение муравья, построенный в работе [10] эвристически, приведен на рис. 26. Однако муравей с таким поведением задачу не решает, так как за 200 ходов съедает только 42 единицы еды.

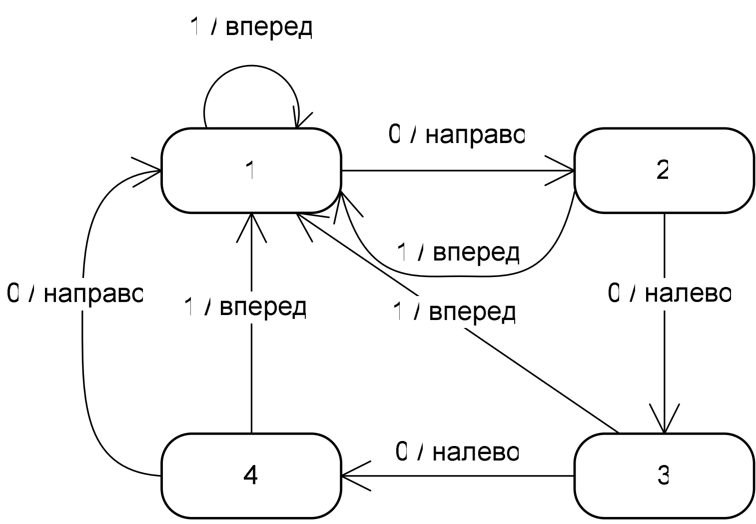

Рис. 26. Автомат с четырьмя состояниями, моделирующий поведение умного муравья

3.2. Известный генетический алгоритм

Конечный автомат, моделирующий поведение муравья, является решением задачи об умном муравье. Чем больше еды съедает муравей за 200 ходов, тем лучше решение (в качестве значения функции приспособленности [10, 24] используется число единиц еды, съеденной муравьем). Автомат задается набором состояний, пронумерованных от нуля. Из каждого состояния выходит две дуги, так как возможно два входных воздействия – ноль и единица. Дуга определяет номер состояния, в которое перейдет автомат, и значение выходной переменной, генерируемой им при переходе (*вперед*, *налево* или *направо*).

3.2.1. Стратегия отбора

Как и в классическом алгоритме из работы [10], в диссертации используется стратегия отбора особей для следующего поколения – отбор отсечением. При использовании такого отбора родительские решения выбираются из группы лучших решений текущего поколения. Размер этой группы решений (порог отсечения) составляет 20% от размера поколения. Все решения из группы лучших решений имеют одинаковые шансы быть выбранными в качестве родительских решений.

Приведем описание предлагаемого генетического алгоритма, использующего отбор отсечением:

1. Текущее поколение решений заполняется случайными решениями.
2. Из текущего поколения решений отбирается группа лучших решений.
3. Формируется новое поколение решений:
    a. Создается пустое новое поколение решений.
    b. Из группы лучших решений случайным образом выбирается пара решений.
    c. Формируется новое решение с помощью применения n-точечного оператора скрещивания [24] к двум выбранным решениям.

d. К новому решению применяются операторы мутации [24] (*n*-точечный оператор мутации и *новые* операторы мутации).
e. Новое решение добавляется в новое поколение решений.
f. Если размер нового поколения меньше размера текущего поколения, то переходим к пункту b.

4. Новое поколение становится текущим.
5. Если число созданных поколений меньше заданного пользователем, то переходим к пункту 2.

3.2.2. Оператор скрещивания

В данной работе, как и в работе [10], используется *n*–точечный оператор скрещивания. Приведем описание алгоритма работы этого оператора:

1. В качестве нового решения берется копия первого из выбранных решений.
2. Осуществляется цикл по всем состояниям нового решения. Для каждого состояния:
   a. Выполняется цикл по всем дугам состояния. Для каждой дуги:
      i. Случайным образом определяется, требуется ли изменить номер состояния, в которое переходит автомат по дуге. Если это требуется, то он изменятся на номер состояния из соответствующей дуги второго решения.
      ii. Случайным образом определяется, требуется ли изменить значение выходной переменной, генерируемое при переходе автомата по дуге. Если это требуется, то значение выходной переменной изменяется на значение выходной переменной, полученной из соответствующей дуги второго решения.

Для *n*–точечного оператора скрещивания задается вероятность его применения к каждому элементу автомата.

3.2.3. Оператор мутации

В данной работе, как и в работе [10], используется *n*–точечный оператор мутации. Приведем описание алгоритма этого оператора:

1. Осуществляется цикл по всем состояниям автомата:
   a. Выполняется цикл по всем дугам переходов в состоянии:
      i. Случайным образом определяется необходимость изменение индекса состояния, в которое автомат переходит по дуге.
      ii. Если требуется изменить индекс состояния, то он изменяется на индекс состояния, выбранный случайным образом.
      iii. Случайным образом определятся необходимость изменения значения выходной переменной, генерируемой автоматом при переходе по дуге.
      iv. Если требуется изменить значение выходной переменной, то оно изменяется на одно из возможных значений, выбранное случайным образом.

Для *n*–точечного оператора мутации задается вероятность его применения к элементу автомата. В отличие от обычного оператора мутации, такой оператор может изменить сразу несколько элементов в автомате.

3.3. Предложенные модификации алгоритма

3.3.1. Сортировка состояний в порядке использования

Во многих задачах в большинстве решений, которые перебирает генетический алгоритм, автоматы в процессе вычисления значения функции приспособленности не попадают в часть состояний. Далее состояния, из которых выполняется переход хотя бы на одном шаге при вычислении функции приспособленности, будем называть *используемыми*, а все остальные состояния – *неиспользуемыми*.

Если при вычислении функции приспособленности на вход автомата подаются все возможные последовательности входных значений, то *неиспользуемые состояния* и переходы, связанные с ними, не влияют на поведение автомата.

Приведем алгоритм реализации нового оператора мутации – сортировка состояний в порядке использования:

1. Создается пустой словарь пар номеров: [«старый номер состояния» – «новый номер состояния»].
2. Моделируется работа автомата. Перед каждым переходом выполняем следующее: если в словаре нет пары, в которой первый элемент равен текущему номеру состояния, то в него добавляется пара [текущий номер состояния – число пар в словаре].
3. Выполняется цикл по всем состояниям автомата. Для каждого состояния: если в словаре нет пары, в которой первый элемент равен номеру состояния, то в него добавляется пара [номер состояния – число пар в словаре].
4. Согласно словарю изменяется порядок состояний и номера состояний, в которые ведут переходы.

Состояния, для которых в словарь добавляются пары в пункте 2 – используемые. Состояния, для которых в словарь добавляются пары в пункте 3 –неиспользуемые.

Автоматы, моделирующие флибы, как правило, существенно проще, чем автоматы, моделирующие умного муравья. Поэтому рассмотрим работу алгоритма сортировки состояний в порядке использования на примере задачи о флибах.

Пусть среда задана с помощью повторяющейся битовой маски `11100`. В процессе работы генетического алгоритма был получен флиб, граф переходов для которого изображен на рис. 27.

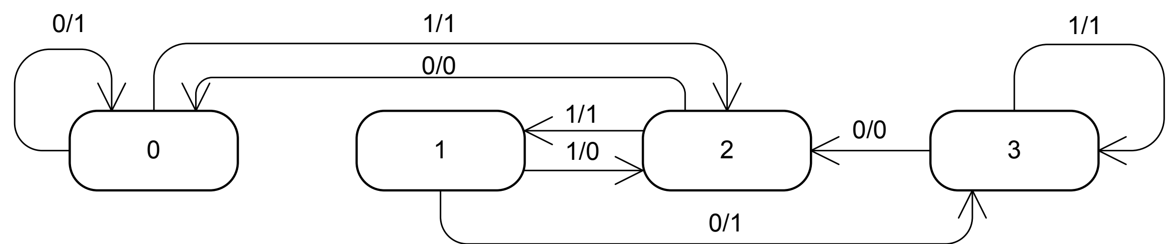

Рис. 27. Граф переходов флиба полученного в процессе работы генетического алгоритма (среда заданна битовой маской 11100)

Состояния флиба пронумерованы цифрами от нуля до трех. Начальное состояние нулевое.

Создается пустой словарь пар [«старый номер состояния» – «новый номер состояния»].

Начинается моделирование работы флиба. На вход автомату подается единица. Автомат переходит в состояние **2** и в словарь добавляется пара **[0, 0]**. На следующем шаге моделирования автомат переходит в состояние **1**, а в словарь добавляется пара **[2, 1]**. Далее автомат переходит в состояние **2** и в словарь добавляется пара **[1, 2]**. При переходе из состояния **2** в состояние **0** в словарь ничего не добавляется, так как для состояний **2** в словаре уже есть пара.

При дальнейшем моделирование работы флиба, пары в словарь не добавляются, так как при среде, заданной битовой маской `11100`, флиб никогда не попадет в состояние **3**, а для всех остальных состояний пары в словаре уже есть.

После моделирования работы флиба словарь пар номеров будет иметь следующий вид:

{[0, 0], [1, 2], [2, 1]}

Осуществляется цикл по всем состояниям. Так как в словаре нет пары только для состояния **3**, то в него добавляется пара **[3, 3]**. Теперь словарь принимает следующий вид:

{[0, 0], [1, 2], [2, 1], [3, 3]}

Осталось изменить порядок состояний и номера состояний в переходах согласно словарю. Состояния **1** и **2** меняются местами и номерами. Граф переходов для флиба, получившегося в результате работы алгоритма, приведен на рис. 28.

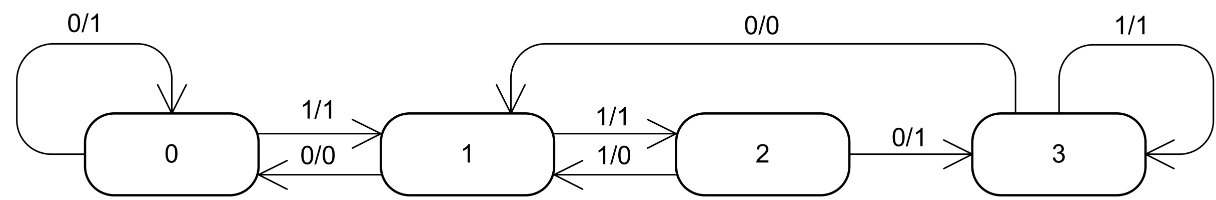

Рис. 28. Граф переходов флиба после применения оператора мутации «сортировки состояний в порядке использования» (среда заданна битовой маской 11100)

После применения алгоритма сортировки состояний можно построить автомат, имеющий такое же поведение, как и автомат, найденный с помощью классического генетического алгоритма [10], но имеющий меньшее число состояний. Для этого в автомате, состояния которого отсортированы в порядке использования, достаточно удалить все неиспользуемые состояния и заменить переходы в них переходами в начальное состояние.

В нашем случае одно неиспользуемое состояние – **3**. В состояние **3** ведет только один переход из состояния **2**. Удалим состояние **3** и заменим переход в него на переход в состояние **0**. Результат изображен на рис. 29.

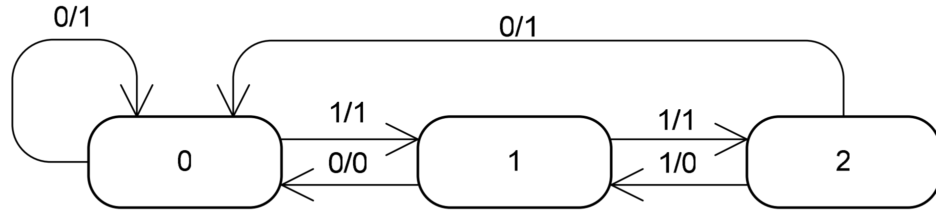

Рис. 29. Граф переходов флиба после удаления неиспользуемых состояний (среда заданна битовой маской 11100)

Так как при моделировании работы автомата переход из состояния **2,** соответствующий значению входной переменной **0**, ни разу не выполнялся, то поведение автомата не изменилось.

Оператор мутации «сортировка состояний в порядке использования» приводит к одному виду автоматы с одинаковым поведением, но разной пометкой вершин. Например, поведение автомата, граф переходов, для которого приведен на рис. 30, аналогично поведению автомата, приведенного на рис. 19. Однако генетический алгоритм будет считать эти автоматы разными решениями. Применение алгоритма сортировки состояний в порядке использования приведет оба автомата к одному виду.

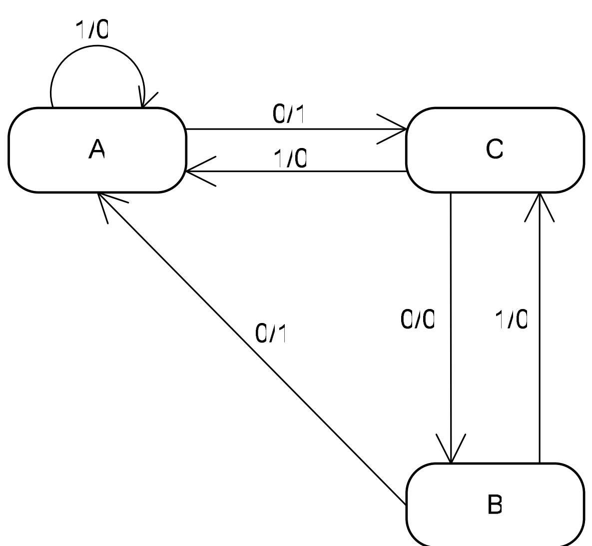

Рис. 30. Граф переходов автомата с тремя состояниями

Таким образом, использование оператора мутации «сортировка состояний автомата в порядке использования» позволяет сократить пространство поиска, в котором генетический алгоритм перебирает решения.

3.3.2. Оператор мутации «всегда вперед, если впереди еда»

После изучения автоматов, которые решают задачу об умном муравье, было замечено, что эти муравьи всегда идут вперед, если перед ними еда. Для того чтобы

достичь такого поведения был добавлен еще один оператор мутации. Этот оператор изменяет значение выходной переменной на значение «вперед» для всех переходов, помеченных входным воздействием единица (в ячейке перед муравьем еда).

3.3.3. Уменьшение числа состояний

В настоящей работе изменена функция приспособленности из работы [10] таким образом, что наиболее приспособленными оказываются те решения, у которых меньше *используемых состояний*. Для этого в функцию приспособленности (число съеденных единиц еды) добавляется соотношение:

$$\frac{1}{1 + \texttt{useful_state_count}},$$ где `useful_state_count` – число используемых состояний.

Так как это соотношение всегда меньше единицы, то наиболее приспособленными будут те муравьи, которые съедают больше еды за 200 ходов. Однако, если количество съеденной еды одинаково, то предпочтение будет отдаваться автоматам с меньшим числом используемых состояний.

3.4. Описание программы для сравнения эффективности генетических алгоритмов, решающих задачу об умном муравье

В программе, написанной на языке C#, автоматы реализуются с помощью трех классов: `StateMachine`, `State` и `Branch`, которые приведены на диаграмме классов (рис. 31).

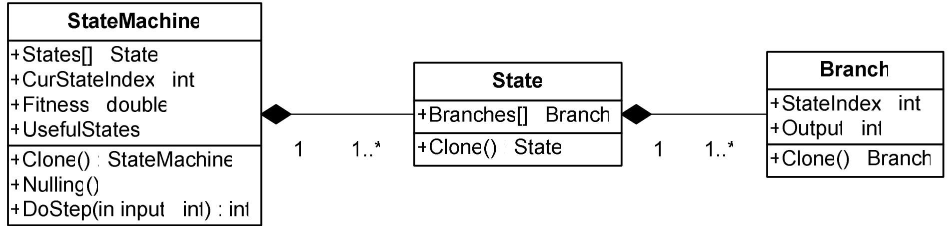

Рис. 31. Диаграмма классов для реализации автомата

В качестве главного класса при реализации автомата применяется класс StateMachine. 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 StateMachine содержит состояния автомата. Свойство CurStateIndex используется для хранения номера текущего состояния автомата в массиве States. Доступ к значению функции приспособленности можно получить с помощью свойства Fitness. Словарь пар номеров [«старый номер состояния» – «новый номер состояния»] (далее словарь используемых состояний) доступен через свойство UsefulStates.

Метод DoStep переводит автомат в новое состояние, изменяет словарь используемых состояний и возвращает значение выходной переменной, генерируемой автоматом. Метод Nulling возвращает автомат в начальное состояние.

В массиве Branches класса State содержатся дуги переходов из данного состояния. Номер элемента в массиве соответствует значению входной переменной.

Переменные StateIndex и Output класса Branch свойства номер состояния, в которое переходит флиб по этой дуге, и значение выходной переменной.

Для реализации стратегий отбора используются классы, изображенные на рис. 32. Эти классы реализуют интерфейс ISelection.

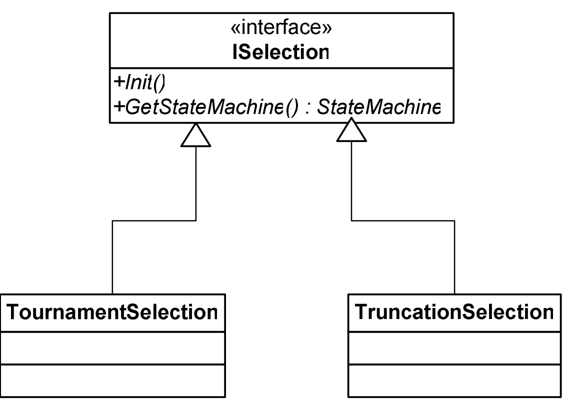

Рис. 32. Диаграмма классов для реализации стратегий отбора

В метод `Init` передаются автоматы текущего поколения, и происходит инициализация стратегии отбора. Метод `GetStateMachine` возвращает родительскую особь для скрещивания. Класс `TournamentSelection` (Листинг 6) реализует турнирный отбор [25], а класс `TruncationSelection` – стратегия отбора отсечением [24] (Листинг 7).

Для реализации операторов скрещивания используются классы, изображенные на рис. 33. Эти классы реализуют интерфейс `ICrossover`.

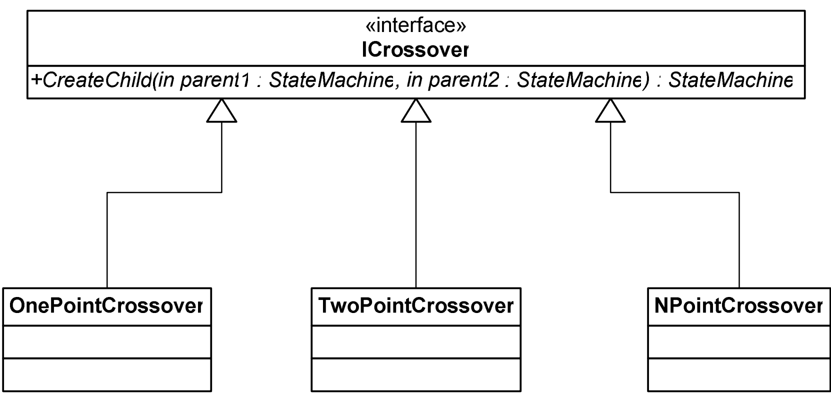

Рис. 33. Диаграмма классов для реализации операторов скрещивания

Метод `CreateChild` создает новый автомата из двух родительских автоматов. Классы `OnePointCrossover` и `TwoPointCrossover` реализуют одноточечный и двухточечный операторы скрещивания. Класс `NPointCrossover` реализует *n*–точечный оператор скрещивания (Листинг 8).

Для реализации операторов мутации используются классы, изображенные на рис. 34. Эти классы реализуют интерфейс `IMutation`.

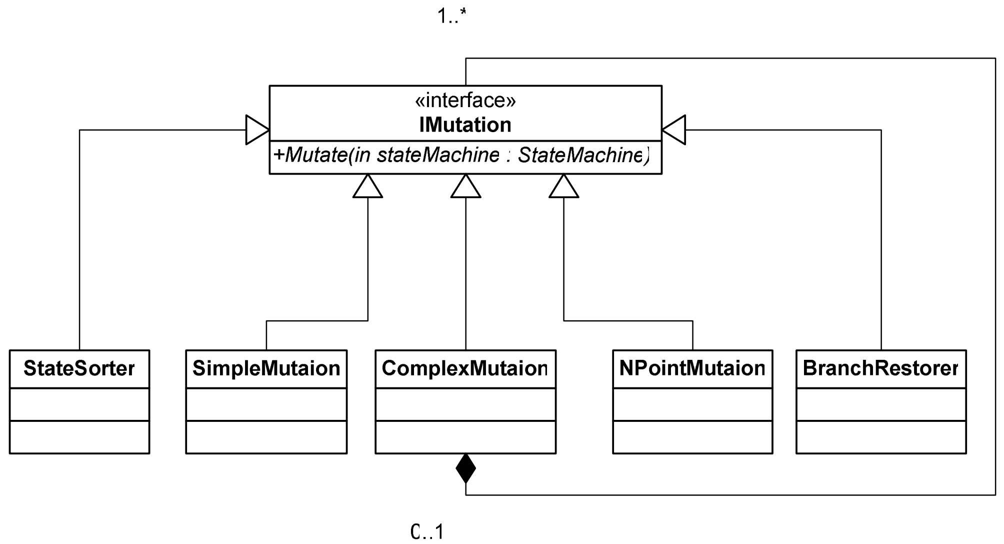

Рис. 34. Диаграмма классов для реализации операторов мутации

Метод `Mutate` выполняет операцию мутации над переданным в него автоматом. Классы `SimpleMutation` и `NPointMutation` реализуют стандартный и *n*–точечный операторы мутации [24] (Листинг 9) соответственно.

Для реализации оператора мутации «восстановление связей между состояниями» используется класс `BranchRestorer`. Класс `StaterSorter` реализует оператор мутации «сортировка состояний в порядке использования».

Для выполнения нескольких операций мутации над одним автоматом применяется класс `ComplexMutation`.

Классы, реализующие вычисление значения функции приспособленности, изображены на рис. 35. Эти классы реализуют интерфейс `IFitness`.

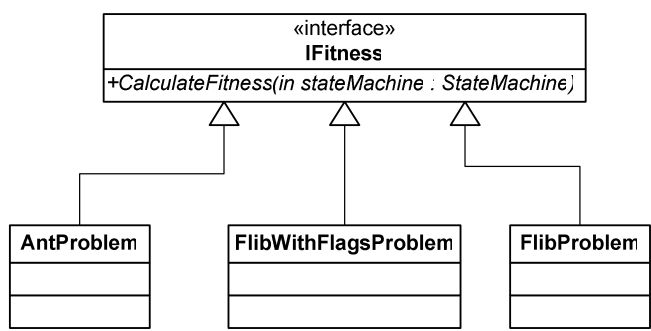

Рис. 35. Диаграмма классов для реализации вычисления значения функции приспособленности

Метод `CalculateFitness` вычисляет значение функции приспособленности для переданного в него автомата. Класс `AntProblem` моделирует поведение умного муравья и определяет значение функции приспособленности для него. Класс `FlibProblem` выполняет ту же функцию для флибов без флагов. Класс `FlibWithFlagsProblem` вычисляет указанную функцию для флибов, реализованных автоматами с флагами.

Основной класс `GAEngine`, реализующий работу генетического алгоритма, изображен на рис. 36.

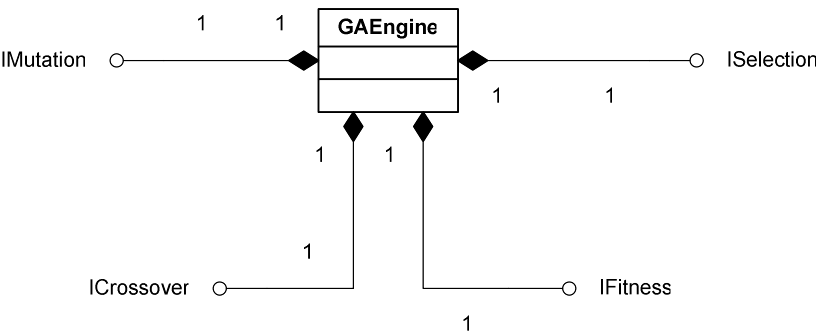

Рис. 36. Диаграмма классов для реализации генетического алгоритма

Класс `GAEngine` не зависит от того, какие именно операторы скрещивания, мутации, стратегии отбора и функции приспособленности используются. Такой подход позволяет легко модифицировать генетический алгоритм.

3.5. Эксперименты

Все эксперименты, о которых идет речь ниже, проводились при размере поколения 10000 и пороге отсечения 2000. Вероятность применения оператора скрещивания к элементу автомата выбрана равной 0.04, а вероятность применения оператора мутации – 0.02. Число состояний автомата было ограничено двадцатью. Эксперименты проводились на компьютере с процессором *AMD Athlon 3800+*.

В табл. 12 приведены результаты экспериментов, в которых использовалась функция приспособленности из работы [10]. Число поколений в экспериментах, в которых использовались предложенные в работе операторы мутации, было выбрано равным 100. Число поколений в экспериментах только с *n*–точечным оператором мутации не превышало двухсот. Каждый эксперимент без дополнительных операторов мутации длился примерно 310 с (время построения двухсот поколений). Продолжительность каждого эксперимента с дополнительными операторами мутации составила около 270 с (время построения ста поколений).

В столбце «Номер поколения» таблицы приведен номер поколения, в котором был найден лучший автомат в эксперименте. В столбце «Результат» приведено число ячеек с едой, съеденных лучшим муравьем в эксперименте. Число используемых состояний лучшего автомата, полученного в эксперименте, приведено в столбце «Число используемых состояний». В столбце «Время поиска» приводится время в секундах, прошедшее от начала эксперимента до построения поколения, в котором был найден лучший автомат в эксперименте.

Таблица 12. Результаты экспериментов с использованием стандартной функции приспособленности

	Номер эксперимента	Номер поколения	Результат	Число используемых состояний	Время поиска
Без применения оператора «сортировка состояний в порядке использования»	1	137	87	15	225
	2	180	88	15	218
	3	73	88	12	137
	4	72	86	12	114
	5	68	89	12	104
С применением оператора «сортировка состояний в порядке использования»	1	47	89	12	154
	2	49	89	11	160
	3	56	89	13	182
	4	51	89	12	145
	5	67	89	13	189

Как следует из приведенных данных, использование предложенных операторов мутации позволило при всех запусках найти автоматы, которые решают поставленную задачу. Известный алгоритм с этим не справился в четырех экспериментах из пяти, даже используя вдвое большее число поколений. Это, по всей видимости, связано с тем, что в данных экспериментах размер поколения был значительно меньше размера поколения, использующегося в работе [10].

В табл. 13 приведены результаты экспериментов, в которых использовалась предложенная в работе функция приспособленности. Число поколений в экспериментах, проводившихся с предложенными в работе операторами мутации,

было ограничено значением 100. Число поколений в экспериментах, в которых использовался только n–точечный оператор мутации, не превышало двухсот. Лучшим результатом считался муравей, который съедает максимальное число единиц еды при возможно меньшем числе используемых состояний. Каждый эксперимент без предложенных операторов мутации длился примерно 320 с (время построения двухсот поколений). Продолжительность каждого эксперимента с дополнительными операторами мутации составила около 290 с (время построения ста поколений).

Таблица 13. Результаты экспериментов с использованием предложенной функции приспособленности

	Номер эксперимента	Номер поколения	Результат	Число используемых состояний	Время поиска
Без применения оператора «сортировка состояний в порядке использования»	1	143	89	11	237
	2	144	89	9	263
	3	138	89	10	259
	4	99	89	13	152
	5	158	89	10	252
С применением оператора «сортировка состояний в порядке использования»	1	61	89	10	161
	2	85	89	10	263
	3	70	89	10	227
	4	63	89	10	176
	5	96	89	9	269

Как следует из приведенных в табл. 13 данных, изменение функции приспособленности позволило получить автоматы с меньшим числом используемых

состояний по сравнению с предыдущим случаем. Необходимо отметить, что алгоритм, использующий дополнительные операторы мутации, работает более стабильно и с измененной функцией приспособленности.

Сравнив данные, приведенные в табл. 12, 13, отметим, что изменение функции приспособленности не только не ухудшило эффективность алгоритма, но даже улучшило ее.

В табл. 14 приведен автомат, который решает задачу об умном муравье для «тропы Джона Мьюра» и имеет минимальной число используемых состояний из автоматов сгенерированных программой, разработанной автором.

Таблица 14. Таблица переходов и выходов для лучшего из сгенерированных автоматов

	В ячейке перед муравьем нет еды		В ячейке перед муравьем есть еда	
Номер состояния	Номер следующего состояния	Выходная переменная	Выходная переменная	Номер следующего состояния
0	направо	6	вперед	1
1	вперед	2	вперед	2
2	вперед	8	вперед	3
3	вперед	2	вперед	4
4	направо	6	вперед	5
5	налево	4	вперед	3
6	направо	7	вперед	1
7	налево	2	вперед	1
8	налево	0	вперед	4

Отметим, что число состояний в автомате, приведенном в табл. 14 (девять состояний), меньше числа состояний в автоматах, полученных в работах [24, 26] (одиннадцать состояний).

В заключение отметим, что задача поиска муравья с наименьшим числом состояний не ставилась [27].

Выводы по главе 3

1. В главе приведено описание классического генетического алгоритма для задачи об умном муравье.
2. Предложен новый оператор мутации для автоматов, заданных с помощью графов переходов – сортировка состояний в порядке использования. Алгоритм его работы рассмотрен на примере.
3. Предложена новая функция приспособленности, учитывающая число используемых состояний в автомате, которая позволила улучшить эффективность предлагаемой модификации генетического алгоритма.
4. Приведены результаты компьютерных экспериментов, которые показали, что предложенная модификация генетического алгоритма, использующая новые операторы мутации и функцию приспособленности, эффективнее классического.

ГЛАВА 4. ГЕНЕРАЦИЯ АВТОМАТОВ ДЛЯ ЗАДАЧИ ПОСТРОЕНИЯ АВТОПИЛОТА ДЛЯ УПРОЩЕННОЙ МОДЕЛИ ВЕРТОЛЕТА

4.1. ПОСТАНОВКА ЗАДАЧИ

Требуется построить автопилот для простейшей модели вертолета (далее вертолета), перемещающейся на плоскости. За один шаг вертолет может повернуться на некоторый фиксированный угол и изменить скорость своего движения (ускориться или замедлиться). Минимальная скорость, с которой может перемещаться вертолет $\mathtt{V}_{\mathtt{min}} = 10^{-4}$, максимальная – $\mathtt{V}_{\mathtt{max}} = 2$ и ускорение $\mathtt{a} = 0.1$ (все величины в настоящей главе, в том числе и на осях графиков, приведены в *условных единицах*). Требуется построить автопилот, который за отведенное время полета, проведет вертолет в определенном порядке через максимальное число заранее заданных меток. Метки нумеруются, а вертолет их проходит в порядке возрастания номеров. При этом вертолет не может пропускать метки, но, так как время полета ограничено, то возможна ситуация, что он может не успеть пройти их все. Считается, что вертолет прошел метку, если он оказался от нее на расстоянии не более 0.5.

Вертолет и окружающая его среда представляются моделями, образующими взаимодействующую систему (рис. 37).

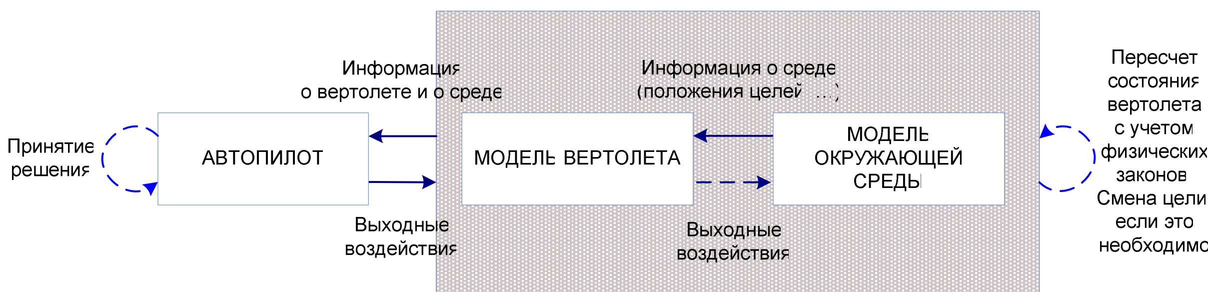

Рис. 37. Структурная схема системы

Модель вертолета, используя полученную от среды информацию, вычисляет необходимые ей данные о текущем состоянии системы. Она использует часть этих

данных для их передачи автопилоту в качестве входных воздействий с целью анализа текущей ситуации и принятия решений. Модель окружающей среды «знает», что представляют собой эти воздействия с точки зрения определенных в ней физических законов, и в соответствии с ними пересчитывает положение и скорость вертолета.

Входные данные (воздействия) (рис. 38) автопилота представляют собой единственную переменную (*номер сектора обзора*): положение текущей цели относительно вертолета, заданное углом между направлением движения вертолета и направлением на метку (рис. 38, а). Сектора всегда неподвижны относительно вертолета, а вертолет летит не по границе двух секторов, а посередине одного из них (рис. 38, б).

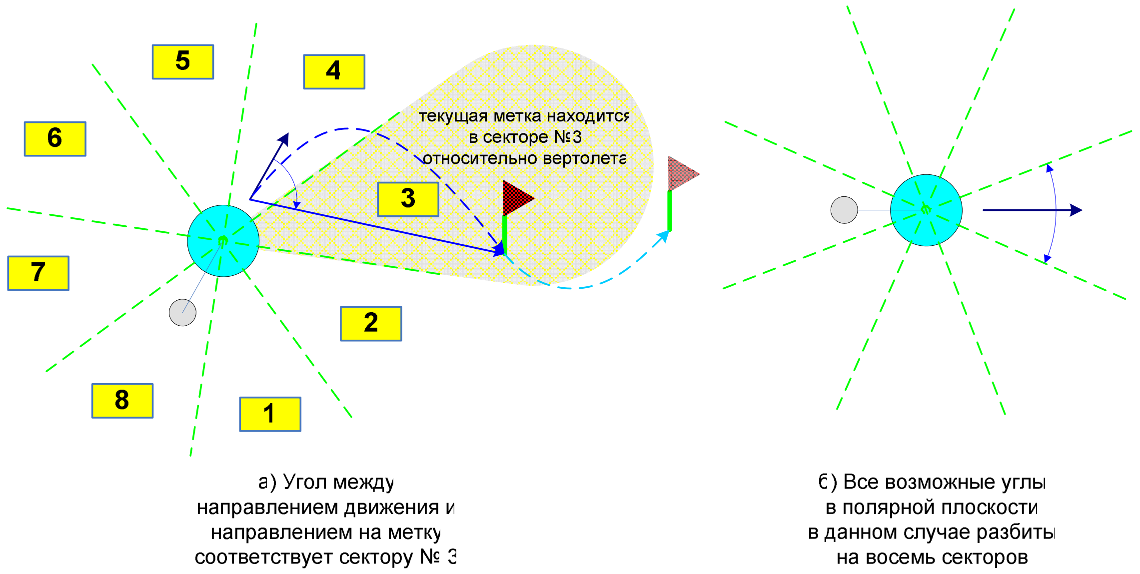

Рис. 38. Входные данные автопилота

В качестве выходных в работе выбраны два воздействия, которые действуют относительно текущего вектора движения вертолета: «Изменить скорость» (придать ускорение в направлении вектора движения) и «Повернуть» (повернуть вектор движения на некоторый угол). Текущие значения ускорения и угла поворота никак не связаны с их предыдущими значениями. В каждый момент времени вертолет знает о положении только одной метки. Таким образом, следующее состояние вертолета

зависит от его положения в пространстве, скорости движения и текущей метки. При достижении вертолетом метки, текущей становится следующая по порядку. В результате направление на метку изменяется.

В качестве модели автопилота используется конечный автомат. Его входные и выходные воздействия должны быть дискретными. Для этого пространство вокруг вертолета разбивается на сектора обзора вертолета, число которых определяется заранее и является настраиваемым параметром.

Автомат содержит некоторое число состояний, ограниченное сверху. Из каждого состояния графа переходов исходят дуги (переходы), число которых равно числу секторов. В качестве значения входной переменной, помечающей соответствующий переход, выступает номер сектора, в котором находится текущая для вертолета метка. Каждый переход переводит автомат в новое состояние. Он помечен конкретным набором выходных воздействий. Поскольку автомат не использует память для хранения информации о глубокой предыстории (зависит только от предыдущего состояния), выбор следующего состояния производится автоматом лишь на основе входного воздействия и текущего состояния. Состояния *косвенно* отражают информацию о текущем положении вертолета, его скорости и предыстории переходов между состояниями. Рис. 39 иллюстрирует работу автопилота, заданного конечным автоматом.

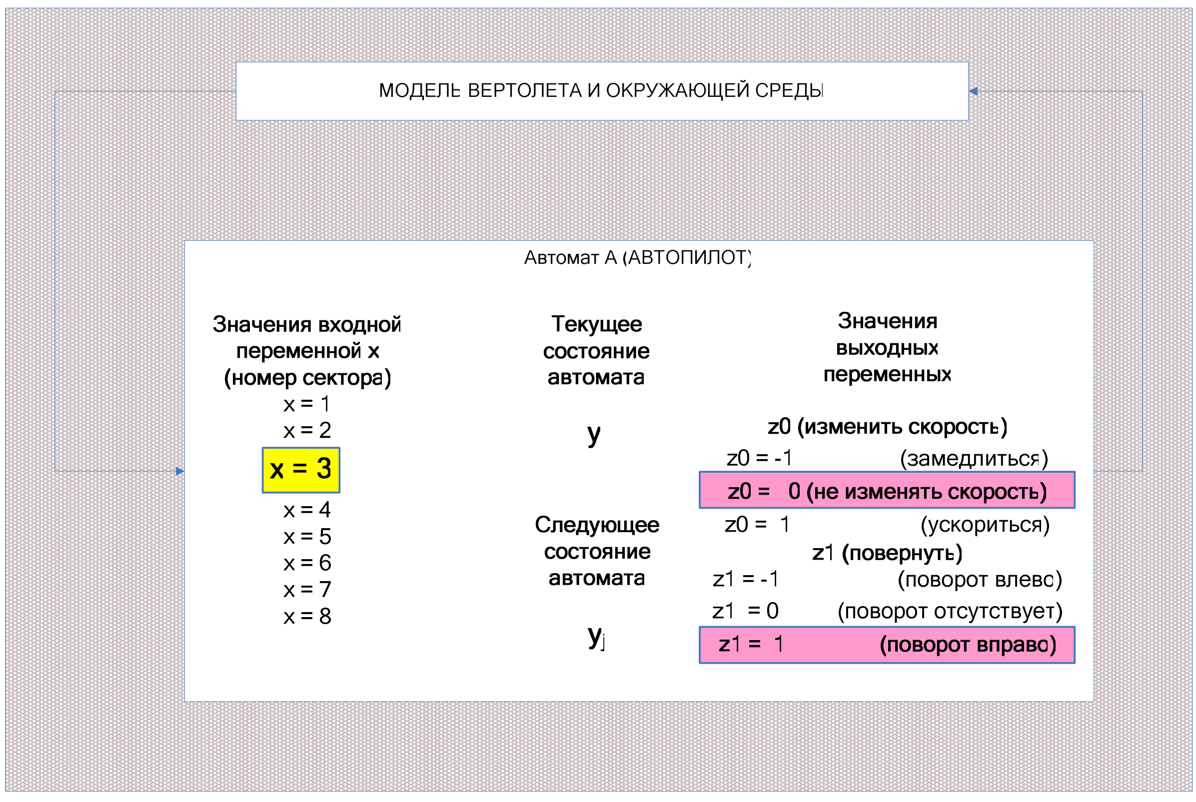

Рис. 39. Работа автопилота, заданного конечным автоматом

4.2. Модель вертолета

Модель вертолета содержит данные о положении вертолета в пространстве, его скорости и направлении движения. Положение вертолета в пространстве хранится в виде точки $|x_h, y_h|$. Скорость и направление движения вертолета задаются вектором $|v_x, v_y|$, который называется вектором скорости. Скорость вертолета определяется соотношением: $V = \sqrt{v_x^2 + v_y^2}$. Так как по условию задачи скорость вертолета не может быть меньше 10^{-4}, то вектор скорости всегда однозначно определяет направление движения вертолета.

Модель вертолета отвечает за изменение его положения в пространстве и пересчет скорости. Положение вертолета в следующий момент времени $|x'_h, y'_h|$ вычисляется по формуле: $|x'_h, y'_h| = |x_h + v_x, y_h + v_y|$.

Определение вектора скорости $|v'_x, v'_y|$ для следующего момента времени выполняется за два шага. Сначала с помощью формулы:

$$|v''_x, v''_y| = \left| v_x \frac{V + 0.1 z_0}{V}, v_y \frac{V + 0.1 z_0}{V} \right|$$

определятся *промежуточный вектор*, учитывающий изменение скорости вертолета, задаваемое значением выходной переменной z0.

Затем определяется искомый вектор с помощью поворота *промежуточного вектора*:

$$|v'_x, v'_y| = \left| v''_x + \frac{0.1 z_1 v''_y}{V''}, v''_y - \frac{0.1 z_1 v''_x}{V''} \right|,$$

где z_1 – значение выходной переменной z1, а V'' – длина промежуточного вектора $|v''_x, v''_y|$.

Если длина полученного вектора V'' оказывается меньше минимальной скорости V_{min}, то вектор скорости $|v'_x, v'_y|$ вычисляется по формуле:

$$|v'_x, v'_y| = \left| v'_x \frac{V_{min}}{V'}, v'_y \frac{V_{min}}{V'} \right|.$$

Если V' больше максимальной скорости V_{max}, то $|v'_x, v'_y|$ изменяется по формуле:

$$|v'_x, v'_y| = \left| v'_x \frac{V_{max}}{V'}, v'_y \frac{V_{max}}{V'} \right|.$$

4.3. Модель окружающей среды

В модели окружающей среды хранятся координаты меток вертолета $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ и номер текущей цели k. Когда расстояние от вертолета до текущей метки становится меньше 0.5 ($\sqrt{(x_k - x_h) + (y_k - y_h)} < 0.5$), номер k увеличивается на единицу.

Модель среды отвечает за определение номера сектора, в котором находится текущая для вертолета метка. Сначала вычисляется угол α между направлением движения вертолета и направлением на текущую метку. Для этого используется формула:

$$\alpha = \mathrm{arctg}\left(\frac{v_x(y_k - y_h) - v_y(x_k - x_h)}{v_x(x_k - x_h) + v_y(y_k - y_h)}\right).$$

Номер сектора n вычисляется по формуле:

$$n = \left\lfloor \frac{\frac{\alpha}{\pi} + 1}{2m} \right\rfloor + 1,$$

где m – число секторов обзора.

4.4. Алгоритм построения автопилота

Задача автопилота – провести вертолет через максимальное число наперед заданных целей в определенном порядке. Как было отмечено выше, время полета ограничено. Лучшим является автопилот, который провел вертолет через большее число меток за отведенное время. Функция приспособленности [10] должна быть максимально гладкой, что позволяет улучшить точность сравнения особей. При этом если число пройденных меток одинаково, то лучшим считается автопилот, для которого расстояние до следующей цели к моменту завершения полета минимально.

В текущей главе хромосому будем задавать в виде графа переходов конечного автомата. Такое задание хромосомы определяется тем, что в рассматриваемой задаче переходы помечаются только одной входной переменной. При этом гены хромосомы – компоненты графа переходов (например, значения входной переменной).

4.5. Общая схема генетического алгоритма

На рис. 40 приведена общая схема работы генетического алгоритма [10].

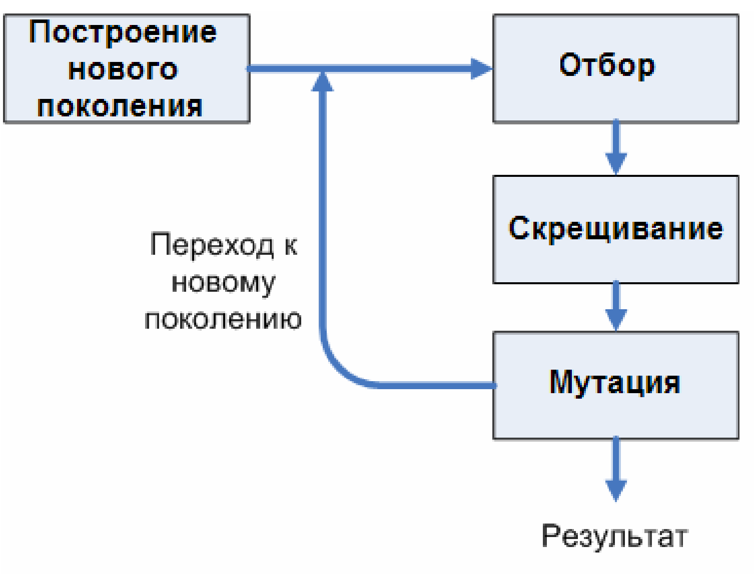

Рис. 40. Общая схема генетического алгоритма

При решении данной задачи в качестве стратегии отбора особей для построения следующего поколения используется отбор отсечением [24]. При применении такой стратегии отбора родительские решения выбираются из группы лучших решений текущего поколения. Размер этой группы (порог отсечения) выбран равным 30% от размера поколения. Все автоматы из группы лучших решений имеют одинаковые шансы быть выбранными в качестве родительских особей.

Подробное описание генетического алгоритма, использующего стратегию отбора отсечением, приведено в разд. 3.2.1. Необходимо отметить, что в данном алгоритме к каждому новому решению последовательно применяются операторы мутации, в

качестве которых выступают *n*-точечный оператор (разд. 3.2.3) и предлагаемый автором работы оператор мутации «сортировка состояний в порядке использования» (разд. 3.3.1).

4.6. Описание программы

Для проведения экспериментов по генерации автопилотов была написана программа на языке *Java*. Программа позволяет задавать время полета, верхний предел числа состояний автомата, размер поколения, максимальное число поколений, число секторов, вероятности применения для *n*-точечных операторов мутации и скрещивания. Также в ней можно изменить порог для стратегии отбора отсечением. При проведении экспериментов при необходимости можно изменять расположение меток вертолета.

Для кодирования графов переходов автоматов используется три класса: `StateMachine`, `State` и `Branch`. В качестве главного класса автомата применяется класс `StateMachine`. Классы `State` и `Branch` реализуют его состояния и переходы соответственно.

Массив `states` в классе `StateMachine` содержит состояния автомата автопилота. Поле `currentStateIndex` используется для хранения номера текущего состояния автомата. В массив `brunches` класса `State` включены переходы из данного состояния.

Поле `stateIndex` в классе `Branch` содержит номер состояния, в которое переходит автомат. В массиве `outputs` хранятся значения выходных переменных, генерируемых при переходе.

При запуске программа создает два файла. В первом из них содержится траектория полета вертолета для автопилота, построенного с помощью генетического алгоритма. Во второй файл выводится таблица переходов и выходов для лучшего

автомата. Перед формированием таблицы производится сортировка состояний автомата в порядке использования, и удаляются неиспользуемые состояния.

4.7. Эксперименты

Все эксперименты проводились при размере поколения 300 и пороге отсечения 90. Максимальное число поколений – 200. Вероятность применения оператора скрещивания к переходу автомата – 0.04, а вероятность применения n-точечного оператора мутации – 0.02. Число состояний автомата не больше 15. Время полета – 200.

На рис. 41 изображены 20 меток, пронумерованных в том порядке, в котором через них должен пролететь вертолет. Точка с координатами (0; 0) является исходной. В начале движения вертолет ориентирован вправо и движется в этом направлении с минимальной скоростью, указанной в разд. 4.1.

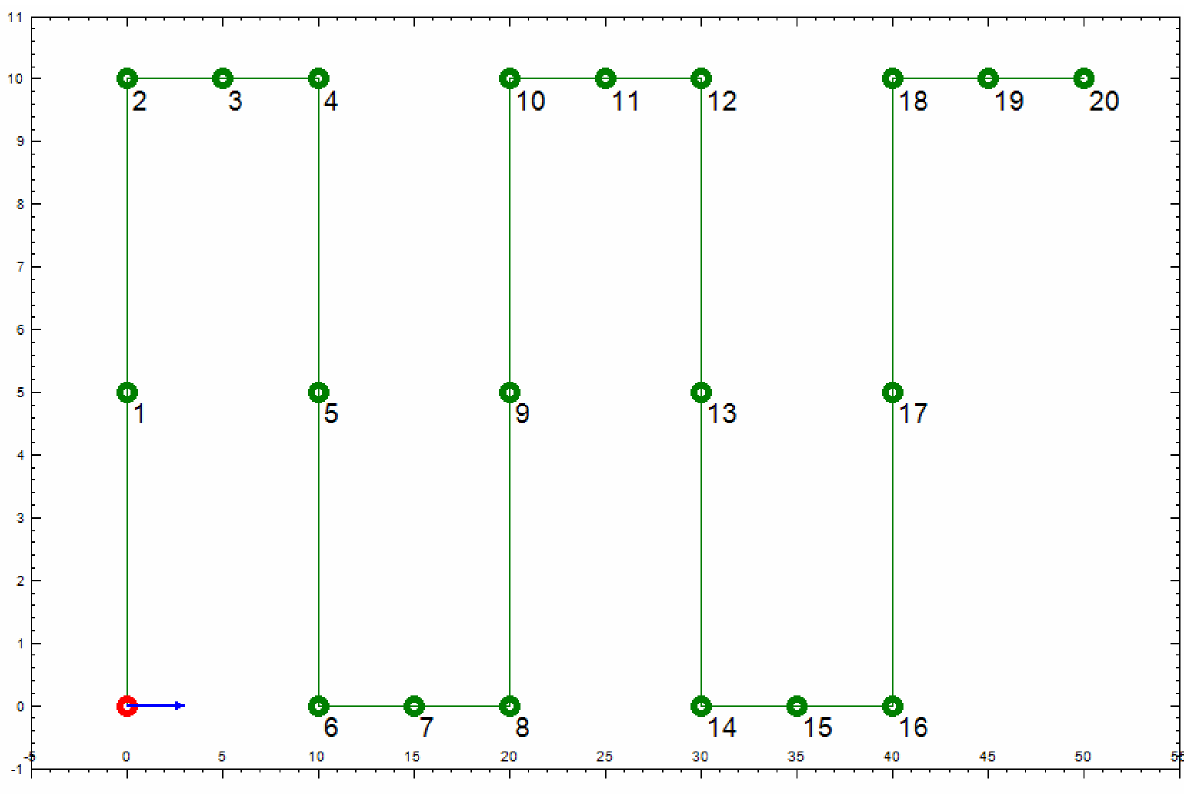

Рис. 41. Метки, проходимые вертолетом

При проведении экспериментов изменялось число секторов обзора вертолета (четыре или шесть), а также использовался или не использовался оператор мутации «сортировка состояний в порядке использования». Эксперименты с каждым набором этих параметров алгоритма проводились по 10 раз.

В табл. 15 приведены результаты экспериментов. Под результатом эксперимента понимается число меток, через которые пролетел вертолет с автопилотом, построенным с помощью предложенного генетического алгоритма. Столбец «худший» – худший результат из десяти экспериментов, столбец «лучший» – лучший результат этих экспериментов, а столбец «средний» – усредненный результат.

Таблица 15. Результаты экспериментов по построению автопилота для вертолета

Число секторов	Оператор мутации «сортировка состояний в порядке использования»	Результат		
		худший	средний	лучший
4	Нет	12	14.1	16
	Есть	12	14.7	17
6	Нет	11	15.6	18
	Есть	14	16.6	18

Из этой таблицы видно, что использование предлагаемого оператора мутации улучшает эффективность работы генетического алгоритма.

На рис. 42 изображена траектория полета вертолета, управляемого лучшим из построенных автопилотов.

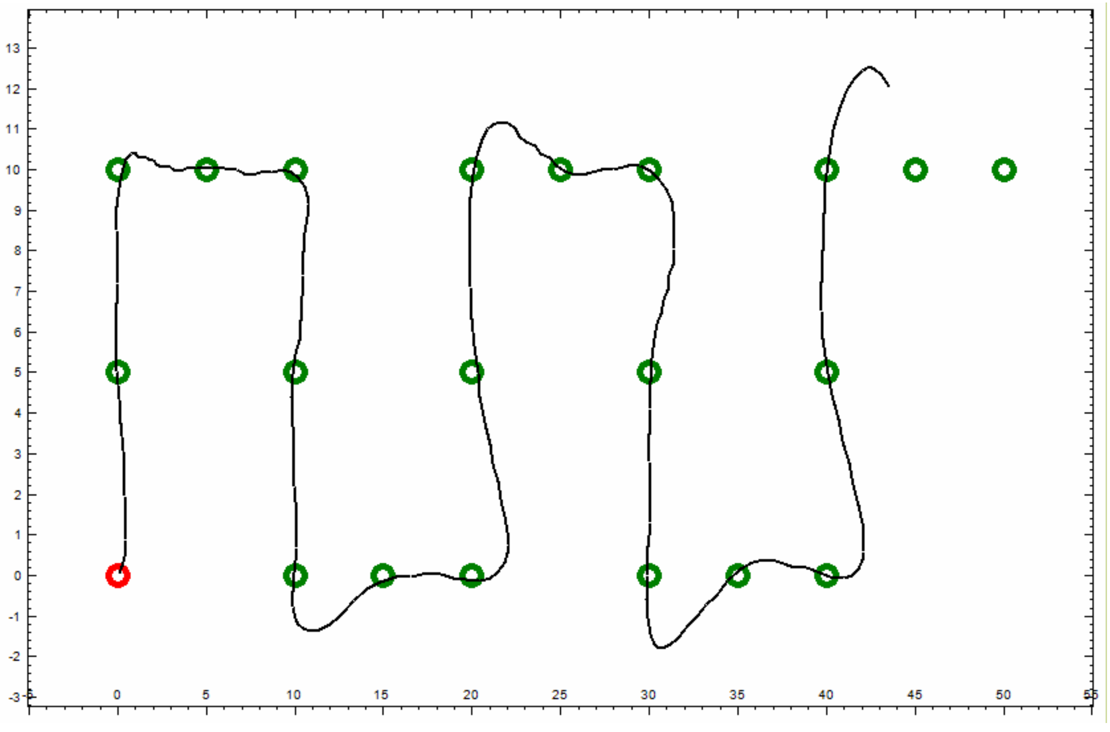

Рис. 42. Траектория полета лучшего вертолета

Также был выполнен эксперимент по использованию восьми секторов обзора вертолета, однако, несмотря на увеличение времени работы генетического алгоритма, существенного улучшения работы автопилота не наблюдалось.

С помощью предложенного генетического алгоритма был построен автопилот с 12 состояниями, который проводит вертолет через первые 18 целей из 20 за отведенное время. Установлено, что при увеличении времени полета на 33 условных единицы, полученный автопилот обеспечит прохождение всех 20 меток.

Выводы по главе 4

1. Приведена постановка задачи построения автопилота для упрощенной модели вертолета.
2. Разработан генетический алгоритм генерации автоматов для задачи построения автопилота для упрощенной модели вертолета, который использует предложенный в предыдущей главе оператор мутации «сортировка состояний в порядке использования».
3. Разработана программа, которая позволяет проводить компьютерные эксперименты по генерации автоматов для поставленной задачи.
4. Приведены результаты компьютерных экспериментов, которые показывают, что использование предложенного оператора мутации улучшает эффективность работы разработанного генетического алгоритма.

ЗАКЛЮЧЕНИЕ

В диссертации получены следующие научные результаты:

1. Разработаны новые операторы мутации для конечных автоматов, представленных в виде графов переходов: восстановление связей между состояниями и сортировка состояний в порядке использования.
2. Предложена новая функция приспособленности, учитывающая число используемых состояний в автомате, которая позволяет повысить эффективность генетических алгоритмов для генерации автоматов.
3. Разработаны модификации генетических алгоритмов для задач о флибах и умном муравье и для построения автопилота для упрощенной модели вертолета.
4. При решении задачи о флибах выполнено сравнение эффективности генетических алгоритмов, которые позволяют генерировать автоматы Мили и автоматы Мили с флагами. Показано, что точность предсказания флибом, который моделируется автоматом с флагами, выше.

Результаты, полученные в диссертации, используются на практике в компании *Транзас* при разработке тренажера вертолета и в учебном процессе в СПбГУ ИТМО. Акты внедрения прилагаются.

Все основные результаты, полученные в работе, опубликованы. Приведем информацию о личном вкладе автора в работы, выполненные в соавторстве.

В работах [23, 28] автором предложены новый оператор мутации «восстановления связей между состояниями» и модификация генетического алгоритма для решения задачи о флибах.

В работах [29, 30] автором предложены новый оператор мутации «сортировка состояний в порядке использования», новая функция приспособленности,

учитывающая число используемых состояний, а также модификация генетического алгоритма для задачи об умном муравье.

В работе [31] автором для задачи о флибах выполнено сравнение эффективности генетических алгоритмов, которые позволяют генерировать автоматы Мили и автоматы Мили с флагами.

В работе [32] автором предложен генетический алгоритм, использующий оператор мутации «сортировка состояний в порядке использования», для построения автопилота для упрощенной модели вертолета.

ЛИТЕРАТУРА

1. *Шалыто А. А.* Технология автоматного программирования / Труды первой Всероссийской конференции “Методы и средства обработки информации”. МГУ. 2003, с. 150-152. http://is.ifmo.ru/works/tech_aut_prog
2. *Mitchell M.* An Introduction to Genetic Algorithms. MA: The MIT Press, 1996.
3. *Fogel L., Owens A., Walsh M.* Artificial Intelligence through Simulated Evolution. NY: Wiley. 1966. (Фогель Л., Оуэнс А., Уолш М. Искусственный интеллект и эволюционное моделирование. М.: Мир, 1969).
4. *Букатова И. Л.* Эволюционное моделирование и его приложения. М.: Наука, 1979.
5. *Гладков Л. А., Курейчик В. В., Курейчик В. М.* Генетические алгоритмы. М.: Физматлит, 2006.
6. *Fogel L.* Autonomous Automata // Industrial Research. 1962. V.4, pp. 14–19.
7. *Hillis W.* Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure / In Artificial Life II. MA: Addison-Wesley, 1992. pp. 313−324.
8. *Whitley, D.* A Genetic Algorithm Tutorial, Technical Report CS-93-103, Colorado State University, 1993.
9. *Blickle, T., and Thiele, L.* A Comparison of Selection Schemes used in Genetic Algorithms. Technical Report №. 11. Gloriastrasse 35, CH-8092 Zurich: Swiss Federal Institute of Technology (ETH) Zurich, Computer Engineering and Communications Networks Lab (TIK), 1995.
10. *Koza J. R.* Genetic programming. On the Programming of Computers by Means of Natural Selection. MA.: The MIT Press, 1998.
11. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.

12. *Frey C., Leugering G.* Evolving Strategies for Global Optimization – A Finite State Machine Approach /Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001). Morgan Kaufmann. 2001, pp. 27–33. http://citeseer.ist.psu.edu/456373.html
13. *Tu M., Wolff E., Lamersdorf W.* Genetic Algorithms for Automated Negotiations: A FSM-based Application Approach /Proceedings of the 11-th International Conference on Database and Expert Systems. 2000. http://ieeexplore.ieee.org/iel5/7035/18943/00875153.pdf
14. *Naidoo A., Pillay N.* The Induction of Finite Transducers Using Genetic Programming / Proceedings of Euro GP. Springer. 2007. http://saturn.cs.unp.ac.za/~nelishiap/papers/eurogp07.pdf
15. *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, техника реализации и инструменты. М.: Вильямс, 2001.
16. *Holland J.* ECHO: Explorations of Evolution in a Miniature World / Proceedings of the Second Conference on Artificial Life. Redwood City. CA: Addison-Wesley, 1990.
17. *Zomorodian A.* Context-free Language Induction by Evolution of Deterministic Push-Down Automata Using Genetic Programming. http://www.cs.dartmouth.edu/~afra/papers/aaai96/aaai96.pdf
18. *Nedjah N., Mourelle L.* Mealy Finite State Machines: An Evolutionary Approach // International Journal of Innovative Computing, Information and Control. 2006. V.2, № 4, pp. 58-67.
19. *Armstrong, D.* A programmed algorithm for assigning internal codes to sequential machines // IRE Transactions on Electronic Computers. EC 11. 1962. № 4, pp. 466-472.
20. *Воронин О., Дьюдни А.* Дарвинизм в программировании // Мой компьютер. 2004. № 35, с. 18-21. http://www.mycomp.kiev.ua/text/7458

21. *Miller B., Goldberg M.* Genetic algorithms, tournament selection, and the effects of noise // Complex Systems. 1995. 3, pp. 193–212.
22. *De Jong K.* An analysis of the behavior of a class of genetic adaptive systems. PhD thesis. Univ. Michigan. Ann Arbor, 1975.
23. *Лобанов П. Г., Шалыто А. А.* Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о флибах / Сборник докладов 4-й Всероссийской научной конференции «Управление и информационные технологии» (УИТ-2006). СПбГЭТУ «ЛЭТИ». 2006, с. 144–149. http://is.ifmo.ru/works/flib
24. *Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A.* The Genesys System: Evolution as a Theme in Artificial Life. 1992. www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
25. *Miller B., Goldberg M.* Genetic algorithms, tournament selection, and the effects of noise // Complex Systems. 1995. V. 9. № 3, pp. 193–212.
26. *Angeline P. J., Pollack J.* Evolutionary Module Acquisition / Proceedings of the Second Annual Conference on Evolutionary Programming. 2003. http://www.demo.cs.brandeis.edu/papers/ep93.pdf
27. *Chambers L.* Practical handbook of genetic algorithms. Boca Raton. CRC Press, 1995.
28. *Лобанов П. Г., Шалыто А. А.* Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о флибах // Известия РАН. Теория и системы управления. 2007. № 5, с. 127-136.
29. *Лобанов П. Г.* Использование генетических алгоритмов для решения задачи об умном муравье // Научно-технический вестник СПбГУ ИТМО. Вып. 39. Исследования в области информационных технологий. Труды молодых ученых. 2007, с. 215-221.

30. *Лобанов П. Г., Шалыто А. А.* Использование генетических алгоритмов для решения задачи об умном муравье / XIV Всероссийская научно-методическая конференция «Телематика-2007». СПб. 2007. Т.2, с. 426, 427.
31. *Лобанов П. Г., Шалыто А. А.* Использование автоматов с флагами для решения задачи о флибах // Научно-технический вестник СПбГУ ИТМО. Вып. 42. Фундаментальные и прикладные исследования информационных систем и технологий. 2007, с. 67-74.
32. *Лобанов П. Г., Сытник С. А.* Использование генетических алгоритмов для построения автопилота для простейшего вертолета / Материалы XV международной научно-методической конференции «Высокие интеллектуальные технологии в образовании и науке». Санкт-Петербургский государственный политехнический университет. 2008. Т.1, с. 298.

ПРИЛОЖЕНИЯ

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. Одноточечный оператор скрещивания

```
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;
        }
    }
}
```

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[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));
        }
    }
}
```

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

Листинг 6. Турнирный отбор

```
namespace Ant.GAEngine.Selection {
    public class TournamentSelection : ISelection {
        private Random _random;
        private IList<StateMachine> _generation;

        public void Init(Random random, ICollection<StateMachine> curGeneration)
{
            _random = random;
            _generation = new List<StateMachine>(curGeneration);
        }

        public StateMachine GetStateMachine() {
            StateMachine firstPretendent =
_generation[_random.Next(_generation.Count)];
            StateMachine secondPretendent =
_generation[_random.Next(_generation.Count)];

            return (firstPretendent.Fitness > secondPretendent.Fitness) ?
firstPretendent : secondPretendent;
        }
    }
}
```

Листинг 7. Отбор отсечением

```
namespace Ant.GAEngine.Selection {
    public class TruncationSelection : ISelection {
        private readonly int _threshold;
        private IList<StateMachine> _list;
        private Random _random;

        public TruncationSelection(int threshold) {
            _threshold = threshold;
        }

        public void Init(Random random, ICollection<StateMachine> curGeneration)
{
            _random = random;
            List<StateMachine> sortedList = new
List<StateMachine>(curGeneration);
            sortedList.Sort(StateMachine.StateMachineFitnessComparison);
            _list = sortedList.GetRange(0, Math.Min(_threshold,
sortedList.Count));
        }

        public StateMachine GetStateMachine() {
            return _list[_random.Next(_list.Count)];
        }
    }
}
```

Листинг 8. n-точечный оператор скрещивания

```
namespace Ant.GAEngine.Crossover {
    public class NPointCrossover : ICrossover {
        private readonly double _probability;

        public NPointCrossover(double probability) {
            _probability = probability;
        }

        public StateMachine CreateChild(Random random, StateMachine firstParent,
StateMachine secondParent) {
            StateMachine result = firstParent.Clone();
            for (int state = 0; state < result.States.Length; state++) {
                for (int branch = 0; branch <
result.States[state].Branches.Length; branch++) {
                    if (random.NextDouble() < _probability) {
                        result.States[state].Branches[branch] =
                            new
Branch(secondParent.States[state].Branches[branch].StateIndex,

result.States[state].Branches[branch].Output);
                    }

                    if (random.NextDouble() < _probability) {
                        result.States[state].Branches[branch] =
                            new
Branch(result.States[state].Branches[branch].StateIndex,

secondParent.States[state].Branches[branch].Output);
                    }
                }
            }

            return result;
        }
    }
}
```

Листинг 9. n-точечный оператор мутации

```
namespace Ant.GAEngine.Mutation {
    public class NPointMutation : IMutation {
        private readonly double _mutationProbability;

        public NPointMutation(double mutationProbability) {
            _mutationProbability = mutationProbability;
        }

        public void Mutate(Random random, StateMachine stateMachine) {
            FSMParams fsmParams = stateMachine.FsmParams;

            foreach (State state in stateMachine.States) {
                for (int i = 0; i < state.Branches.Length; i++) {
                    if (random.NextDouble() < _mutationProbability)
                        state.Branches[i] = new
Branch(random.Next(fsmParams.StateCount), state.Branches[i].Output);

                    if (random.NextDouble() < _mutationProbability)
                        state.Branches[i] = new
Branch(state.Branches[i].StateIndex, random.Next(fsmParams.OutputCount));
                }
            }
        }
    }
}
```