Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

В. В. Каширин

Метод модификации оценочных функций для оптимизации работы генетических алгоритмов, генерирующих конечные автоматы

Бакалаврская работа

Руководитель – А. А. Шалыто

Санкт-Петербург
2008

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время широко исследуется возможность применения алгоритмов машинного обучения для генерации детерминированных конечных автоматов. В частности, для генерации детерминированных конечных автоматов успешно применялись генетические алгоритмы [1–4]. Объединение конечных автоматов и генетических алгоритмов позволяет получать эффективные решения нетривиальных для задач, при этом использование генетических алгоритмов позволяет не только получать необходимое решение, но и оптимизировать свойства генерируемых конечных автоматов.

Одной из актуальных задач в области генетических алгоритмов является разработка метода оптимизации работы генетического алгоритма. Существует несколько решений этой задачи, и целью этих решений является набор параметров генетического алгоритма, гарантирующий наибольшую скорость его работы. При этом возникают проблемы с трудоемкостью поиска оптимальных параметров.

В настоящей работе предлагается метод оптимизации работы генетических алгоритмов, основанный на модификации оценочной функции, используемой алгоритмами. Метод рассматривается в задаче генерации детерминированных конечных автоматов с помощью генетических алгоритмов, поскольку использование конечных автоматов позволяет более формально описать метод.

Рассмотрим основные задачи, решаемые в данной работе.

Первая задача – описание метода оптимизации работы генетического алгоритма с помощью модификации оценочной функции.

Вторая задача – построение алгоритма оптимизации генетического алгоритма с помощью описанного метода.

В главе 1 приводятся основные понятия и характеристики генетических алгоритмов, выполняется обзор существующих методов оптимизации работы генетических алгоритмов и осуществляется постановка задачи.

В главе 2 предлагается новый метод, позволяющий оптимизировать работу генетических алгоритмов, описывается алгоритм его работы.

В главе 3 приведено экспериментальное исследование работы предложенного метода на примере задачи об «Умном муравье».

ГЛАВА 1. ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ И СУЩЕСТВУЮЩИЕ МЕТОДЫ ОПТИМИЗАЦИИ ИХ РАБОТЫ

В этой главе кратко излагаются общие концепции генетических алгоритмов и рассматриваются существующие подходы к оптимизации их работы.

1.1. Генетические алгоритмы

Генетические алгоритмы (ГА) входят в группу эволюционных алгоритмов, которые, в свою очередь, являются подклассом эволюционных вычислений.

Эволюционные вычисления — это группа методов оптимизации, использующая понятие эволюции популяции особей. Эволюция представляет собой процесс естественного отбора, в ходе которого наиболее приспособленные к окружающей среде особи дают потомство, из которого формируется следующее поколение, при этом поколение потомков в среднем оказывается более приспособленным, чем поколение родителей.

Суть эволюционных алгоритмов сводится к следующему.

Фиксируется множество особей $\boldsymbol{X}$ (так же называемых хромосомами), обладающих некоторыми параметрами. Среди этих особей необходимо выбрать наилучшие по некоторому критерию. Вводится целевая функция, называемая функцией приспособленности, или фитнесс-функцией, которая формируется на основе параметров особей:

$\boldsymbol{F}$: $\boldsymbol{X} \rightarrow \boldsymbol{R}$ ($\boldsymbol{R}$ — множество действительных чисел) и каждой особи $x \in \boldsymbol{X}$ из множества $\boldsymbol{X}$ сопоставлено значение функции $\boldsymbol{F}(x)$.

Так же вводятся операции мутации и скрещивания.

Общий принцип работы эволюционного алгоритма можно изобразить в виде схемы на рис. 1.

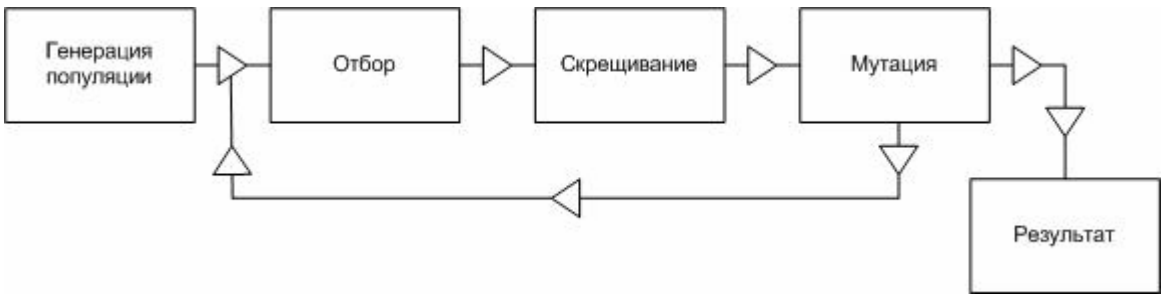

Рис.1. Общая последовательность работы эволюционного алгоритма

Таким образом, принцип действия эволюционного алгоритма таков:

1. Случайным образом генерируется начальная популяция;
2. Пока не выполняется критерий остановки, итеративно совершаются следующие шаги:
   2.1. Генерируется промежуточная популяция путем отбора текущего поколения;
   2.2. Особи промежуточной популяции скрещиваются. В результате этого формируется новое поколение;
   2.3. Производится мутация получившегося поколения;

Операции инициализации, скрещивания, и мутации называются генетическими операциями. Различные реализации генетических операторов, а так же представления особей порождают различные виды эволюционных алгоритмов.

Генетический алгоритм является одним из первых эволюционных алгоритмов. В генетическом алгоритме каждый параметр особи кодируется строкой битов, в результате особь представляет собой битовую строку, являющуюся конкатенацией строк упорядоченного набора параметров.

Особенности работы генетического алгоритма:

1. **Инициализация.** С помощью случайной генерации строк создается начальная популяция.
2. **Отбор**. Существует несколько возможных методов отбора особей, из них в последствии нам будут интересны следующие:

- *Метод рулетки* характеризуется тем, что шанс особи на выживание тем больше, чем выше значение ее функции приспособленности.
- *Метод элитизма* состоит в том, что выживает только фиксированное количество наиболее приспособленных особей, остальные отбрасываются.

3. **Скрещивание**. Эта операция осуществляется следующим образом: при скрещивании строк одинаковой длины часть символов копируется из одной строки, остальные символы копируются из другой строки. Таким образом, операции скрещивания различаются способом выбора скрещиваемых частей. Классический вариантом является одноточечный кроссовер: для родительских строк случайно выбирается точка раздела, и потомок получается путем соединения отсеченных частей. Также распространены двухточечный и однородный кроссовер.

4. **Мутация**. Чаще всего применяется простейший вид мутации – выбирается случайный бит в строке и инвертируется.

Родственником генетических алгоритмов является генетическое программирование, которое тоже является эволюционным алгоритмом и характеризуется тем, что в качестве особей выступают компьютерные программы. Развитием генетического программирования является эволюционный алгоритм, в котором особь представляет собой конечный детерминированный автомат. В обоих случаях генетические операции достаточно сильно отличаются от тех, что применяются в классическом генетическом алгоритме, но суть их остается прежней.

1.2. Проблемы оптимальности работы генетических алгоритмов

В генетических алгоритмах можно выделить характерные особенности, затрудняющие поиск наилучшего результата и ограничивающие эффективность их работы. Приведем основные особенности:

- При программировании работа ГА переносится на иное пространство поиска, и плохое кодирование может испортить или замедлить работу

алгоритма. Также плохое кодирование может затруднить идентификацию составляющих хорошего решения, в результате чего поиск имеет случайное и не сфокусированное поведение.

- Размер популяции конечен.
- Количество итераций конечно.
- Хорошие решения теряются или уничтожаются.
- Недостаточное покрытие пространства поиска.
- Слишком сильная или недостаточно сильная избирательность, что ведет к преждевременному схождению к, вероятно, неоптимальному результату, и случайному поиску соответственно.
- Несбалансированный отбор:
    - Несбалансированный отбор родителей.
    - Фитнесс-функция плохо взаимодействует с методами отбора.
    - Плохие генераторы случайных чисел.

Проблемы, относящиеся к кодированию, достаточно сложно поддаются анализу, и в целом носят ненаучный характер, так как относятся к вопросу реализации алгоритмов. Остальные же трудности относятся к самим алгоритмам и настройке параметров их работы. Проблемы ограниченного размера популяции и числа итераций не решаются как таковые, и чтобы как-то ускорить процесс вычислений применяют различные техники, например, разрабатывают параллельный ГА [5]. В целях улучшения параметров работы алгоритмов и разрабатываются различные способы оптимизации. Рассмотрим наиболее известный из них.

1.3. Мета-генетическое программирование

Идея мета-генетического программирования была впервые упомянута в работе Юргена Шмидхубера в 1987 году [6], и с тех пор получила достаточно сильное развитие [7–11]. В общих чертах идея проста: использовать один генетический алгоритм для оптимизации работы другого генетического алгоритма. Рассмотрим этот метод более подробно.

В генетическом программировании операторы обычно фиксированы, и задаются программистом. Зачастую это операторы инициализации и скрещивания, но могут так же применяться другие, например операторы мутации. Так же может быть создано и применено любое число других генетических операторов. От этих операторов во многом зависит успешность работы генетического алгоритма. Однако поскольку операторы, заданные программистом, могут быть неоптимальными для данного алгоритма, была придумана идея применения еще одного (мета-)генетического алгоритма, задачей которого является поиск оптимальных генетических операторов. Особенности работы мета-генетического алгоритма точно такие же, как и у простого ГА с учетом того, что необходимо придумать представление генетических операторов в виде хромосомы, и в качестве фитнесс-функции для каждой особи использовать его среднее или максимальное значение в популяции, выводимой с помощью генетического алгоритма, в котором применяются порожденные генетические операторы.

Развивая данный подход, можно создавать популяции операторов, воздействующих на популяции операторов, и т.д., и даже популяции воздействующие на самих себя.

В поддержку этого подхода говорит то, что не существует оптимальных операторов для всех видов задач, поэтому оказывается удобным найти оптимальный набор операторов для конкретной задачи, и использовать его для нахождения решения в этой и подобных ей.

Однако проблема данного метода заключается в высоких вычислительных затратах. Действительно, для того чтобы оценить каждую особь мета-алгоритма, необходимо протестировать её на достаточном числе итераций основного алгоритма. Поскольку реальные задачи зачастую работают со сложными объектами, тестирование которых представляет собой так же очень затратное действие, то при построении конфигурации одного генетического алгоритма с помощью другого генетического алгоритма общие

вычислительные затраты растут с высокой скоростью, пропорционально увеличению популяции одного и другого алгоритма.

1.4. Постановка задачи

Необходимо разработать метод оптимизации работы алгоритма генетического программирования, не использующий технику модификации генетических операторов и построения мета-генетического алгоритма, более эффективный в отношении вычислительных затрат и достаточно действенный в сравнении с рассмотренными методами.

Выводы по главе 1

1. Приведены общие сведения об эволюционных и генетических алгоритмах.
2. Выполнен краткий обзор проблем, из-за которых затрудняется поиск решения с помощью генетических алгоритмов.
3. Рассмотрены наиболее известные подходы к решению проблем поиска оптимального результата.
4. Выявлено, что эти методы требуют высоких вычислительных затрат, и не во всех случаях оправдывают свое применение.
5. Сформулирована задача, решаемая в работе.

ГЛАВА 2. МОДИФИКАЦИЯ ОЦЕНОЧНОЙ ФУНКЦИИ

В данной главе предлагается новый метод улучшения поиска оптимального результата, основанный на принципе модификации оценочной функции алгоритма. Применение метода рассматривается в задаче о генерации автоматов с помощью генетических алгоритмов.

2.1. Использование генетических алгоритмов для построения автоматов

В работах [1–4] изучается применение генетических алгоритмов к различным задачам, при этом решение в этих работах ищется в виде детерминированного конечного автомата. На рис. 2 изображено решение задачи о «флибах» [3].

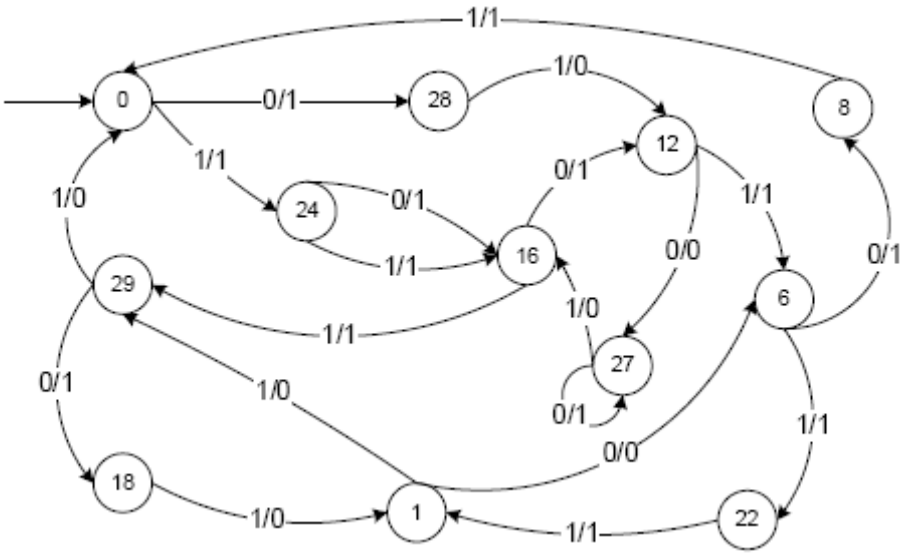

Рис.2. Решение задачи о «флибах»

В различных задачах варьируются виды используемых автоматов, и способы представления автоматов в памяти компьютера. В перечисленных работах используется конечный автомат с действиями на переходах (автомат Мили). На рис. 3 изображено решение задачи об «Умном муравье».

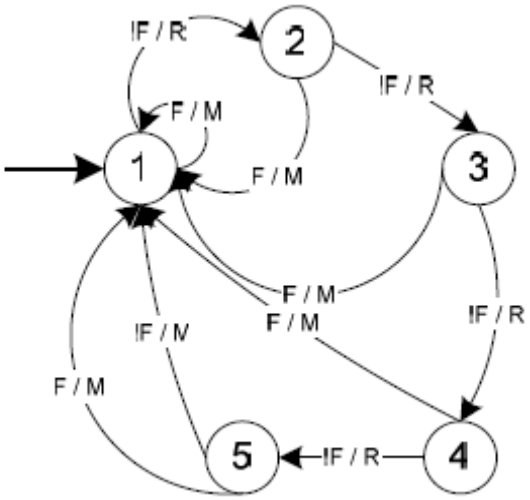

Рис.3. Решение задачи об «Умном муравье»

Пометки на переходах имеют формат *условие* / *действие*. Автомат, описывающий поведение объекта, обладает набором входных и выходных воздействий. Входные воздействия выступают в качестве условий в таком автомате, а выходные – в качестве действий.

В данной задаче входные воздействия:

• F – перед муравьем есть еда;

• !F – перед муравьем нет еды.

Выходные воздействия:

• M – «Сделать шаг вперед»;

• L – «Повернуть налево»;

• R – «Повернуть направо»;

• N – «Ничего не делать».

2.2. Качество оценочной функции

В генетических алгоритмах оценочная функция, или фитнесс-функция, непосредственно участвует в оценке полученных решений. Впоследствии, на основе полученной характеристики для каждой особи, осуществляется выборка особей для скрещивания и мутации, а так же выборка особей, автоматически переходящих в следующее поколение в случае, если используется принцип элитизма.

Поскольку генетические алгоритмы являются фактически способом решения оптимизационных задач, то традиционно в фитнесс-функции участвуют только оптимизируемые параметры, значения которых вычисляются в ходе тестирования особей очередного поколения.

Однако зачастую особи обладают большим числом параметров, по которым их можно охарактеризовать, нежели те несколько, по которым их оценивает фитнесс-функция. При этом особи, имеющие одинаковое значение фитнесс-функции, могут оказаться сильно различающимися в остальных характеристиках, которыми можно их описывать. Одно значение фитнесс-функции не дает представления о поведении особи в среде, тогда как совокупность характеристик особи может позволить понять принципы взаимодействия ее со средой. При этом очевидно, что поведение особи и определяет, насколько приспособлена или неприспособленна особь к окружающей среде.

Таким образом, возникает вопрос о выявлении взаимосвязей между оптимизируемыми и косвенными характеристиками особи, и учете их при оценке особей очередного поколения.

Следствием неточной оценки особей является часть проблем, приведенных в главе 1 – плохая избирательность по пространству поиска, потеря хороших результатов вследствие этого, и т.п.

Одним из решений проблемы потери хороших результатов и учета различий особей, могло бы быть сравнение хромосом особей. Таким образом, в случае применения элитизма можно было бы избежать переноса в следующее поколение особей с идентичными хромосомами, с одновременной утерей особей с таким же значением функции приспособленности, но иным составом хромосомы. Преимуществом такого метода является то, что он не уменьшает общности применения алгоритма, поскольку рассматриваются лишь структуры данных. Недостатком данного метода является то, что при возрастании сложности структуры данных, описывающей хромосомы, так же увеличиваются затраты на операцию их сравнения. К тому же, с помощью

сравнения состава хромосом невозможно выделить практически никаких свойств, по которым можно было бы оценить преимущество одной особи по сравнению с другой, при условии, что они имеют одинаковое значение фитнесс-функции.

2.3. Метод модификации оценочной функции генетического алгоритма

Для решения проблемы точной оценки особей генетического алгоритма, генерирующего конечные автоматы, предлагается следующий метод, заключающийся в модификации оценочной функции.

Метод фактически состоит из двух задач – выделения дополнительных характеристик особи и выявления связи между их значениями и значениями оценочной функции. Вначале рассмотрим вопрос выделения характеристик.

Выделение дополнительных характеристик особи производится для каждой задачи отдельно, поскольку универсальных характеристик, присутствующих у особей во всех генетических алгоритмах, не существует. Однако в случае, если решения ГА ищутся в виде конечного автомата, можно гарантированно выделить набор дополнительных характеристик.

Предположим, что есть некоторая задача, которая решается с помощью генетического алгоритма, и её решения ищутся в виде конечных детерминированных автоматов.

Пусть $X = \{x_1, x_2, \dots, x_m\}$ – множество входных воздействий, $Z = \{z_1, z_2, \dots, z_n\}$ – множество выходных воздействий, $C = \{c_1, c_2, \dots, c_k\}$ – свойства взаимодействия особи со средой, не являющиеся входными или выходными воздействиями.

Мы можем вычислить, сколько раз выполняется свойство или совершается любое из воздействий в процессе тестирования особи в среде. Более того, можно вычислить, сколько раз выполняются более сложные условия. Фитнесс-функция, в свою очередь, является функцией от некоторых таких вычисленных значений. Например, в задаче о муравье традиционно

значение фитнесс-функции равно количеству съеденной еды. Однако если поведение муравьев описывается в виде автоматов, как в приведенном в начале главы примере, то их фитнесс-функцию можно описать как число переходов *F* / *M*.

Пусть S – функция, считающая число совершенных воздействий или число сколько раз выполнилось свойство, например: $S(x_1)$, $S(x_k / z_l)$, S(c) и т.п.

Введем так же функцию $P_x(y, z, \ldots) = S(y) + S(z) + \ldots$, которая вычисляется при очередном воздействии *x* или выполнении свойства *c*.

Можно построить и большее число вспомогательных функций, но тех, что мы ввели, будет достаточно для описания большого числа характеристик различных особей.

Далее построим многообразие введенных функций *S*, варьируя подставляемые переменные. Оно будет включать как функции от простых параметров, так и от непротиворечивых сочетаний входных и выходных воздействий.

С помощью построенного множества видов функций *S* можно конструировать функции типа *P*, однако очевидно, что количество возможных функций равно числу возможных подмножеств из множества функций типа *S*, то есть равно $(2^{|S|}) – 1$. Поэтому предполагается, что функции типа *S* и *P* будут конструироваться в зависимости от задачи, при помощи некоторых эвристических методов.

После этапа построения имеется два набора функций $S = \{S_{x1}(p), S_{x2}(p), ...\}$ и $P = \{P_{x1}(p), P_{x1x2}(p), ...\}$, которые теперь можно объединить в один. Назовем его *F*. Функции из *F* непосредственно связаны с характеристиками, поэтому будем называть *F* набором характеристик.

Следующей задачей метода является выявление связи между характеристиками, вычисляющимися с помощью построенных функций, и оптимизируемой характеристикой, вычисляемой с помощью фитнесс-функции. Для этого предлагается способ, заключающийся в последовательном отборе характеристик, в зависимости от степени их влияния на оптимизируемое

значение. Зафиксируем число итераций I, на которых будут тестироваться характеристики. Это число не обязательно делать большим, поскольку мы хотим лишь выделить общий характер связи, но оно не должно быть маленьким, поскольку на небольшом количестве итераций характер связи может быть не выявлен вовсе или выявлен неправильно. Так же мы должны выбрать число повторений каждого теста N, что необходимо для уточнения результата оценки. Зафиксируем число e, необходимое для учета погрешности. Таким образом, эти три значения, так же как и размер популяции в генетических алгоритмах, являются настроечными параметрами.

Введем обозначения:

F_0 – основная фитнесс-функция.

F_x – подстановка характеристики x.

$F_{1/x}$ – подстановка характеристики вида $1/x$.

F_m – модифицированная фитнесс-функция.

$F_{m,x}$ – модифицированная фитнесс-функция, включающая подстановку F_x.

На первой стадии алгоритма тестируются подстановки F_x и $F_{1/x}$ характеристик x из набора F. Они оцениваются относительно значений F_0 следующим образом.

Ведем следующую функцию $C(F_x)$:

$$C(F_x) = (F_{0,max}(I) + F_{0,av}(I)/4) / (F_{0,max}(1) + F_{0,av}(1)/4),$$

$F_{0,max}(I)$ – максимальное значение F_0 по популяции на I-ой, последней итерации алгоритма, использующего в качестве фитнесс-функции подстановку F_x.

$F_{0,av}(I)$ – среднее значение F_0 по популяции на I-ой, последней итерации алгоритма, использующего в качестве фитнесс-функции подстановку F_x.

$F_{0,max}(I)$, $F_{0,av}(I)$ – аналогично, но для первой итерации алгоритма.

Эти значения подсчитываются N раз для каждой подстановки, и усредняются. После этого они сравниваются со значением $C(F_0)$. Если $C(F_x) > C(F_0)$, или $|C(F_x) - C(F_0)| < e$, это означает, что характеристика x с подстановкой F_x непосредственно связана с основной фитнесс функцией.

Выявив все такие характеристики и их подстановки, сформируем начальную модифицированную фитнесс функцию:

$$Fm = F_0 + F_{x1} + \ldots + F_{xk}.$$

Найденные характеристики извлекаются из набора *F*. Таким образом, во множестве *F* остаются характеристики, которые на протяжении тестирования никак существенно не показали влияния на F_0, или изменения старой фитнесс-функции имели случайный характер. Это может объясняться либо тем, что эта характеристика никак не взаимосвязана с оптимизируемым свойством особи, либо тем, что зависимость не очевидная, и полезность данной характеристики может быть выявлена только в случае использования ее в совокупности с уже полученной модифицированной оценочной функцией. Оставшиеся характеристики будем считать второстепенными.

На следующей стадии будем итеративно выделять второстепенные характеристики. Отличие от первого шага состоит в том, что характеристики будут вноситься уже в модифицированную функцию, при этом подстановка будет нормироваться по максимальному значению характеристики. Это сделано для того, чтобы новая подстановка не оказывала большого влияния на значение оценочной функции, так как характеристика может иметь очень большое значение, и ей подстановка может стать преобладающей величиной.

При тестировании сравниваются величины $C(F_m)$ и $C(F_{m,x})$, и в случае если нормированная подстановка *x* увеличила величину *C*, то в конце итерации она вводится в модифицированную оценочную функцию, а сама она характеристика *x* извлекается из *F*. Если подстановка никак не показала себя, то она тестируется еще раз, но уже внутри новой модифицированной функцией.

Алгоритм прекращается, если в *F* не осталось характеристик, или если их подстановки не оказывают никакого влияния на результат.

Обобщая, алгоритм поиска наилучшей фитнесс-функции можно записать в следующем виде:

1. Построить множество характеристик особей F, зафиксировать число итераций для тестирования I, число повторений тестирования N, и значение точности e.

2. Исследовать каждую характеристику из F, используя ее подстановку в качестве фитнесс-функции алгоритма, запуская его N раз на I итераций. Сравнить величины $C(F_x)$ и $C(F_0)$ для каждой характеристики. Характеристики, при подстановке которых значение $C(F_x)$ близко или превосходит $C(F_0)$, исключить из F, и построить с помощью этих подстановок и F_0 модифицированную оценочную функцию F_m.

3. Исследовать оставшиеся в F характеристики, вводя их нормированные подстановки в F_m, и сравнивая величины $C(F_m)$ и $C(F_{m,x})$. Повторять шаг 3 до тех пор, пока F не окажется пустым, либо пока в F не останутся характеристики, не оказывающие воздействия на модифицированную фитнесс-функцию.

2.4. Оценка трудоемкости метода

Оценим, сколько итераций генетического алгоритма будет выполнено на каждой итерации построения оценочной функции.

Пусть k – число характеристик, I – число итераций алгоритма при тестировании характеристики, N – число повторов тестирования. Тогда на первой итерации алгоритма, строящего оценочную функцию, будет произведено $k * I * N * 2$ итераций.

Если предположить, что на каждой итерации будет выбираться хотя бы одна характеристика, то общее число итераций будет равно $I * N * k * (k+1)$.

Для ускорения тестирования характеристик можно применять различные техники, например:

- В случае если за $I / 2$ итераций коэффициент воздействия b не изменился, можно отбросить эту подстановку, и вместо нее тестировать следующую на той же популяции.

- Если при первых тестированиях коэффициент b оказался значительно меньше единицы, то остальные итерации тестирования можно не проводить, поскольку маловероятно, что в последующих тестах он окажется выше единицы.

Можно разработать и другие способы уменьшить общее число итераций генетического алгоритма.

Рассмотрим, сколько итераций генетического алгоритма происходит при использовании другого метода оптимизации. Например, рассмотрим метод, при котором один генетический алгоритм (назовем его A_1) выращивает операторы скрещивания, работающие в другом генетическом алгоритме (A_2). Пусть второй генетический алгоритм аналогичен тому, работу которого мы оптимизируем.

Важным параметром настройки генетического алгоритма A_1 является число особей-операторов в популяции. Обозначим его n. На каждой итерации алгоритма A_1 каждая особь-оператор тестируется на алгоритме A_2, по тому же принципу, что используется и в нашем методе. Таким образом, на каждой итерации A_1 совершается $I * N * n$ итераций алгоритма A_2. Это число умножается на количество итераций алгоритма A_1, которое может быть фиксированным, или нефиксированным, в зависимости от условий остановки работы алгоритма.

Таким образом, требовательность разработанного алгоритма зависит от числа исследуемых характеристик, соответственно наш алгоритм будет выигрывать у рассмотренного аналога по трудоемкости, если число характеристик будет невелико, а число итераций, необходимое для поиска наилучшего результата для второго алгоритма будет достаточно большим.

Выводы по главе 2

1. Объяснен принцип работы метода.
2. Описан алгоритм построения модифицированной оценочной функции.

3. Рассмотрена трудоемкость алгоритма, проведено сравнение с трудоемкостью другого оптимизационного метода.

ГЛАВА 3. ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ РАБОТЫ МЕТОДА

В этой главе проводится экспериментальная оценка метода на примере задачи об «Умном муравье» и выполняется анализ полученных результатов.

3.1. Постановка задачи об умном муравье

Приведем краткое описание задачи «Умный муравей» на основе работы [12]. В традиционной постановке используется двумерный тор размером 32 на 32 клетки. На некоторых клетках поля расположена еда, которая изображается темно-серыми клетками на рис. 4.

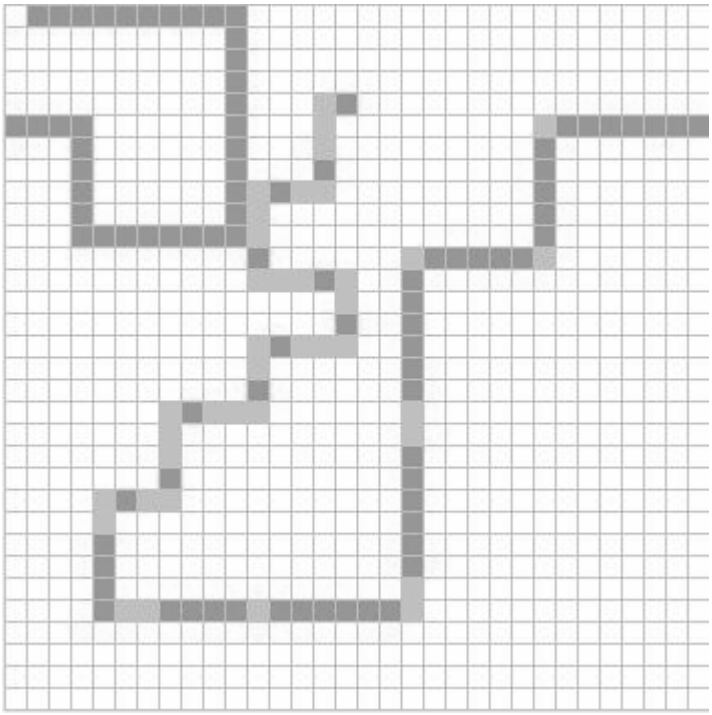

Рис. 4. Игровое поле

Еда расположена вдоль некоторой ломаной линии, но не на всех ее клетках. Клетки ломаной, на которых ее нет, изображены светло-серым. Белые клетки не содержат еду и не принадлежат ломаной. Всего на поле 89 клеток с едой.

В начале игры муравей находится в верхней левой клетке поля. Он занимает одну клетку поля и может смотреть в одном из четырех направлений –

север, запад, юг или восток. В начале игры муравей смотрит на восток. Муравей может определять находится ли яблоко непосредственно перед ним. За один ход муравей может совершить одно из четырех действий:

- сделать шаг вперед, съедая еду, если она там находится;
- повернуть налево;
- повернуть направо;
- ничего не делать.

Съеденная муравьем еда не восполняется. Муравей жив на протяжении всей игры и еда не является необходимым ресурсом для его жизни. Ломаная строго фиксирована. Муравей может ходить по любым клеткам поля.

Игра длится 200 ходов, на каждом из которых муравей совершает одно из четырех действий. По истечении 200 ходов подсчитывается количество еды, съеденной муравьем. Это значение и есть результат игры.

Цель игры – создать муравья, который за 200 ходов съест как можно больше еды.

Поведения муравья можно описать с помощью конечного автомата с действиями на переходах, у которого есть одна входная переменная логического типа (находится ли еда перед муравьем), а множество выходных воздействий состоит из четырех упомянутых выше действий. Пример такого автомата уже был приведен в главе 2.

3.2. Анализ решения задачи с помощью генетического алгоритма

В работе [3] приведено описание генетического алгоритма, решающего задачу об «умном муравье» и генерирующего в качестве решения конечный автомат Мили, описывающий поведение муравья.

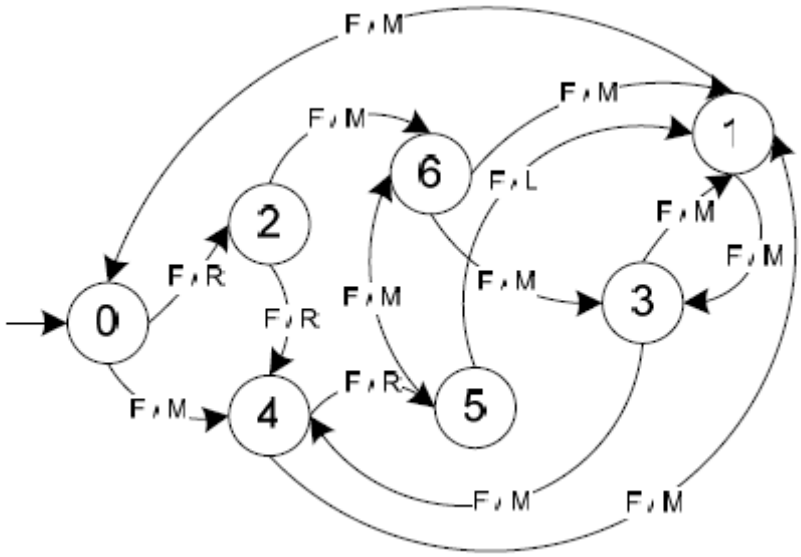

Рис. 5. Автомат, позволяющий съесть всю еду за 190 ходов

На основе предложенного в этой работе генетического алгоритма и был построен предложенный способ оптимизации. Рассмотрим особенности, которые были выявлены при изучении задачи об умном муравье и ее решении с помощью генетического алгоритма.

На рис. 6 изображен график распределения значений фитнесс-функций среди популяции размером 5000 особей на 20 итерации генетического алгоритма.

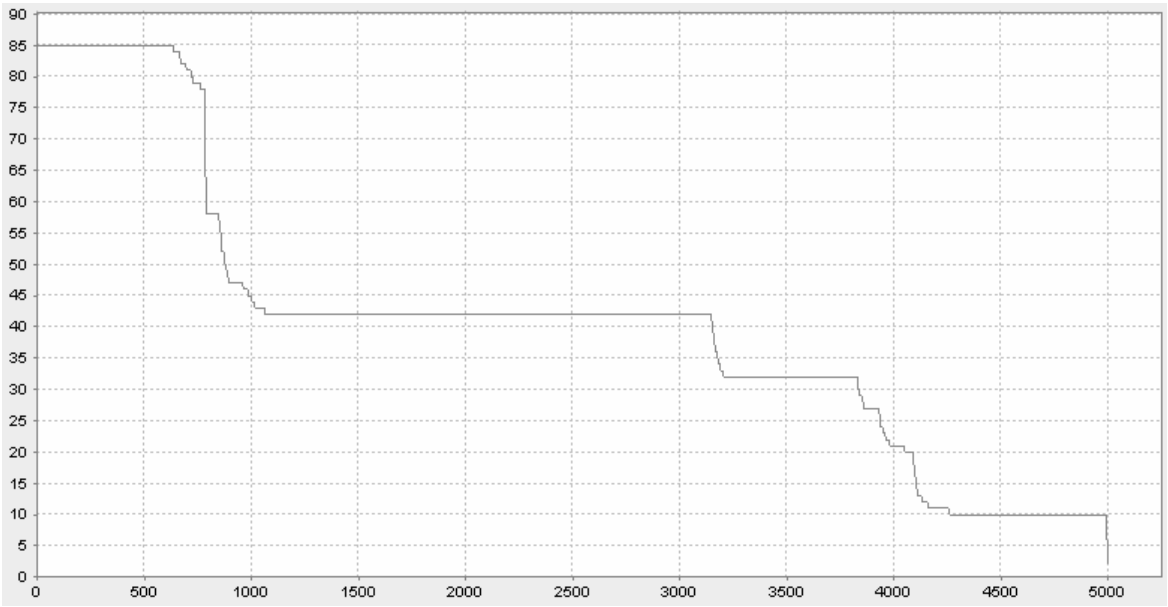

Рис. 6. График распределения значений фитнесса-функций

По графику видно, что в популяции преобладают группы особей со значениями фитнесс-функции 85, 42, 32, 10. Для алгоритма все особи внутри этих групп одинаковы между собой, поэтому нет большой разницы в том, какая именно из особей будет переведена в следующее поколение, выбрана в промежуточное поколение и т.д. Однако рассмотрим теперь немногочисленную группу особей со значением фитнесс-функции равным 80, таких особей всего 16. Рассмотрим теперь их поведение в среде. На рис. 7–10 изображены траектории, по которым движутся четыре разные особи со значением фитнесс-функции равным 80. Пронумеруем эти особи числами от единицы до четырех.

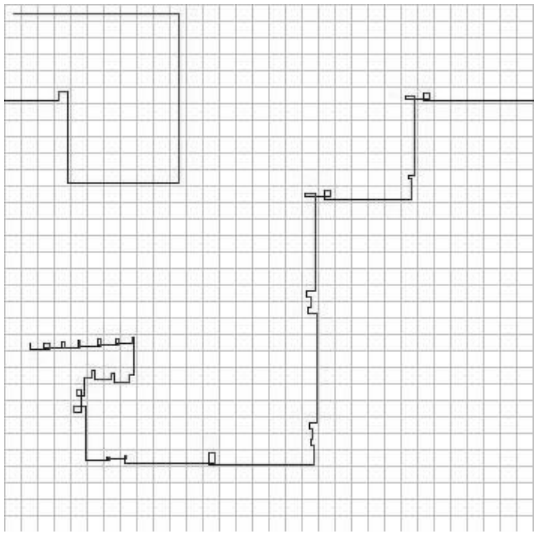

Рис. 7. Траектория особи 1

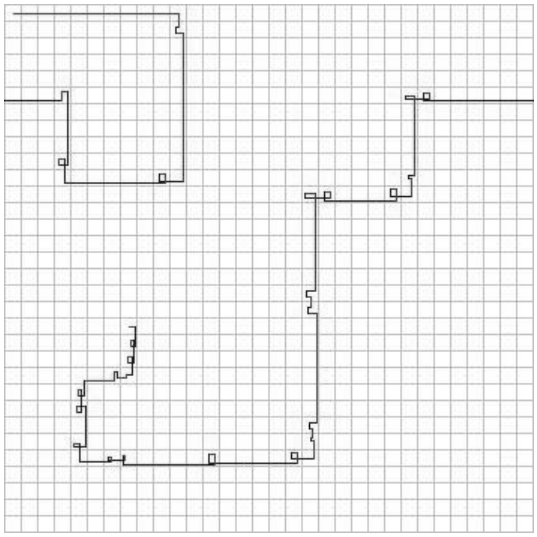

Рис. 8. Траектория особи 2

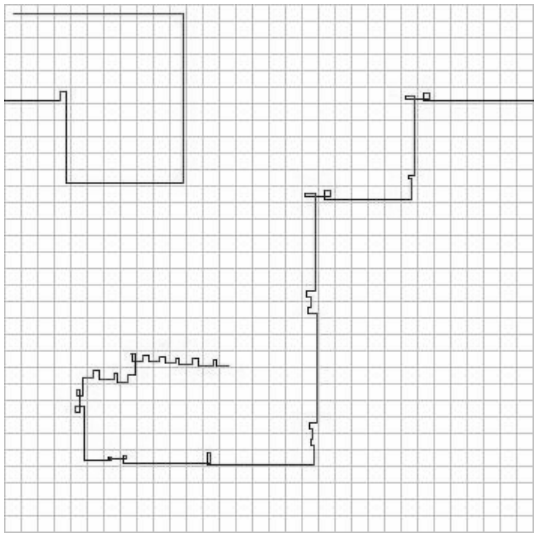

Рис. 9. Траектория особи 3

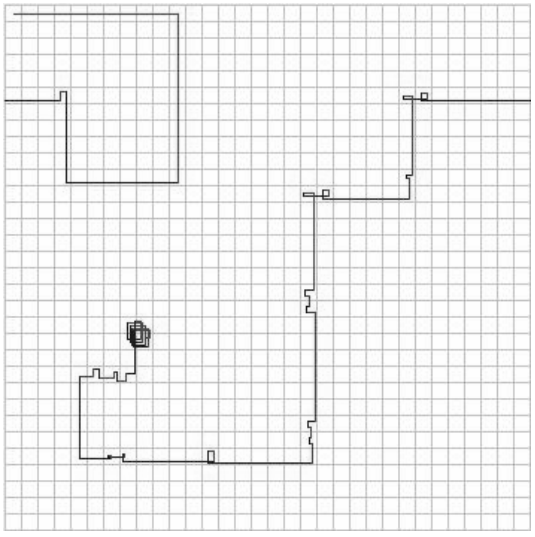

Рис. 10. Траектория особи 4

По рис. 7–10 видно, насколько различается поведение особей в конце траектории их движения. Очевидно, особи различаются своим строением, которое диктует их поведение в среде. Однако, сравнивая построенные автоматы, мы сможем лишь выделить особенности строения, отличающие одни автоматы от других, но не сможем из этого сделать достаточно полезных выводов. Более того, даже если бы особенности строения автоматов могли говорить о приспособленности особи к среде, то процедура оценки этих характеристик была бы чересчур дорогостоящей.

Рассмотрим некоторые количественные характеристики этих особей. В табл. 1 приведены некоторые характеристики особей, вычисляемые в процессе тестирования особи в среде.

Таблица 1. Количественные характеристики, описывающие муравьев

Особь/ характеристика	Количество съеденной еды	Номер шага, на котором муравей последний раз был на линии	По какому числу клеток линии прошел муравей	Число поворотов
1	80	169	95	99
2	80	198	96	104
3	80	168	94	100
4	80	162	96	86

Из таблицы видно, что их характеристики сильно различаются. Отсюда можно сделать вывод, что особи с одинаковым значением оценочной функции можно ранжировать при помощи учета подобных характеристик. Но при этом возникает вопрос, какие признаки стоит считать лучшими или худшими. Под понятием «лучше» (или «хуже») будем подразумевать, что выраженность этого признака означает потенциально большую приспособленность особи к среде, и что в случае применения генетических операторов к этой особи результат будет лучше, чем в случае применения генетических операторов к особи с таким же значением функции приспособленности, но с менее выраженным признаком.

3.3. Реализация метода модификации оценочной функции алгоритма

В качестве основы для исследования и реализации предлагаемого метода был выбран фреймворк *GAAP*, разработанный Е. А. Мандриковым и В. А. Кулевым (http://gaap.net.ru:8080/). Использование данного фреймворка позволило сконцентрироваться на реализации метода и не задумываться над реализацией генетического алгоритма и генетических операторов. Наличие во фреймоворке тестовых задач, таких как задача об «умном муравье», и как следствие объектов управления позволило абстрагировать предлагаемый метод от конкретики решаемой задачи.

Для применения метода потребовалось реализовать всего два класса: специальную фитнесс-функцию, которую можно модифицировать, и класс-тестер, реализующий сам метод построения фитнесс-функции.

Тестер обрабатывает данные, поставляемые объектом, оценивающим характеристики полученных решений, который для каждой задачи различен. Тестер на основе данных, полученных от такого объекта из конкретной задачи, последовательно конструирует модифицированную фитнесс-функцию, которая используется при поиске решений конкретной задачи с помощью генетических алгоритмов.

Тестер реализует алгоритм, описанный в главе 2. На каждой итерации тестер изучает степень воздействия определенной характеристики из набора на качество поиска результата.

Исходные коды классов, реализующих оценочную функцию (*AntFitnessFunction.java*) и тестер (*TestRunner.java*) приведены в Приложении.

3.4. Результаты работы метода

Метод тестировался на уже описанной задаче об «умном муравье», среда представляла собой поле, указанное на рис. 4.

Параметры настройки алгоритма были выбраны следующие: размер популяции – 5000, число тестирований одной характеристики – 2, число

итераций при тестировании характеристики – 150. Из характеристик, генерируемых объектом, собирающим информацию о решениях, алгоритм выделил следующие параметры.

- `FOOD_COUNT (FC)` – количество съеденной еды. Этот параметр учитывается в традиционной фитнесс-функции.
- `LAST_FOOD (LF)` – номер шага, на котором объект последний раз съедал еду.
- `PATH_PASSED (PP)` – число клеток кривой, на которых побывал объект.
- `ROTATIONS (R)` – число поворотов.

На основе данных, полученных при тестировании, тестер сконструировал следующую фитнесс-функцию, подсчитывающую приспособленность особи *o*:

$$F^*(o) = FC + 1 / LF_{norm} + PP + 1 / R_{norm}.$$

Приставка *norm* означает, что величина нормирована по своему максимальному значению.

Сравним эффективность работы алгоритма, использующего обычную и модифицированную функцию. Первое сравнение проведем на том же поле.
В табл. 2 приведены данные по экспериментам.

Таблица 2. Результаты тестирования обычной и модифицированной фитнесс-функции

	Размер популяции	Число тестирований	Среднее число итераций, прошедших до получения решения
F_0	6000	7	353
F*	6000	7	201

Следующие тестирования проводились уже не на том поле, на котором строилась модифицированная оценочная функция, а на ином.

Первый вариант – поле *Santa-Fe*. На рис. 11 изображена конфигурация этого поля.

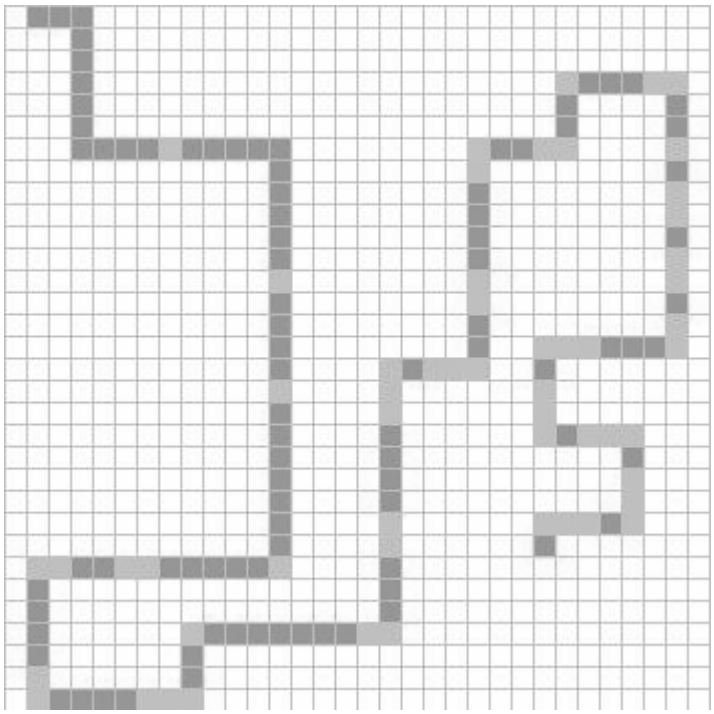

Рис. 11. Поле *Santa-Fe*

Условия – 91 клетка с едой, 300 шагов.

В табл. 3 приведены результаты тестирования.

Таблица 3. Результаты тестирования функций на поле *Santa Fe*

	Размер популяции	Число тестирований	Среднее число итераций, прошедших до получения решения
F_0	6000	5	60
F*	6000	5	36

И, наконец, наиболее сложное для тестирования поле – *Los-Altos*. На рис. 12 изображена его конфигурация.

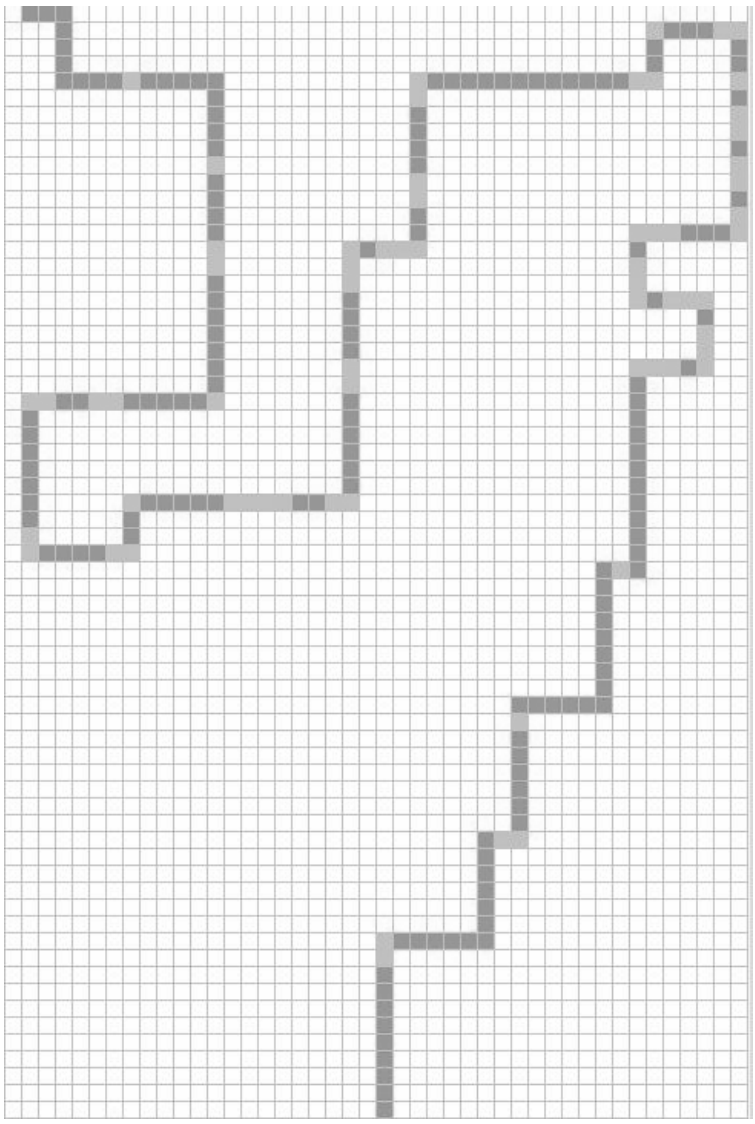

Рис. 12. Поле *Los-Altos*

Условия задачи – 400 шагов, 156 клеток с едой.

Это поле достаточно сложно в плане получения решения, позволяющего съесть всю еду за 400 шагов. Простая схема тестирования здесь не показательна, поскольку решение в виде конечного автомата на данный момент получено не было. Однако применение модифицированной оценочной функции позволило существенно продвинуться в направлении получения решения.

Сначала была протестирована обычная оценочная функция. При размере популяции в 6000 особей, наилучшее решение съедало 129 единиц еды, и в среднем на получение этого решения уходило около 1500 итераций.

Применение модифицированной оценочной функции позволило получить решение, съедающее 141 единицу еды. Причем на получение этого решения в

среднем уходило 150 итераций, а минимальное число итераций, за которое было найдено это решение, равнялось 115. Траектория движения особи, съедающей 141 единицу еды, изображена на рис. 13.

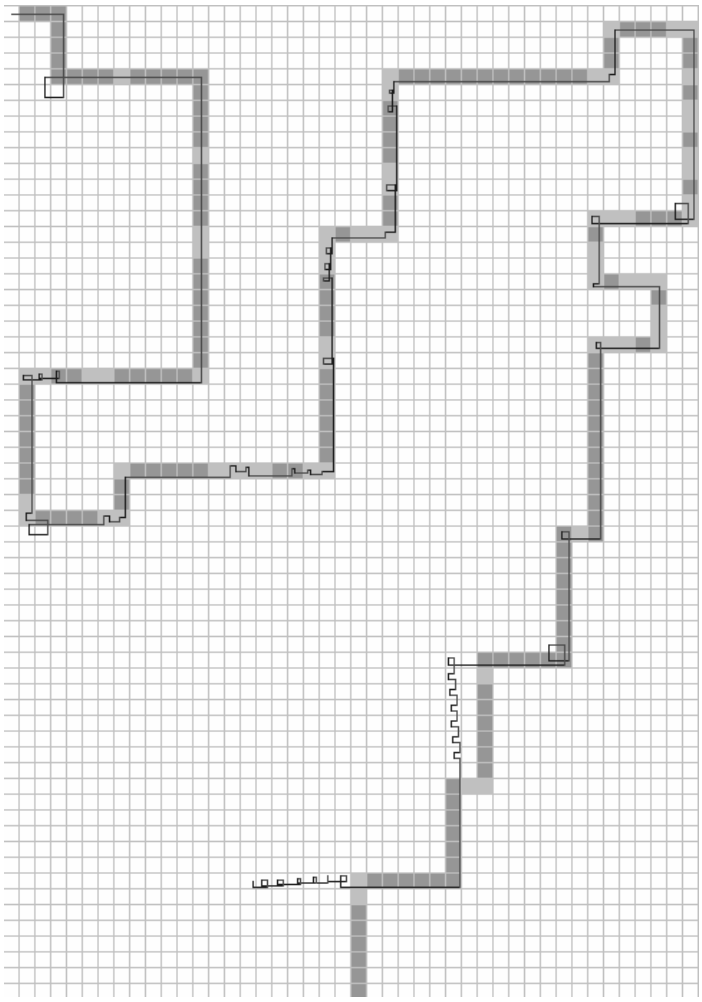

Рис. 13. Решение на поле *Los-Altos,* съедающее 141 единицу еды

Выводы по главе 3

1. Продемонстрирована связь между дополнительными характеристиками особи и ее поведением.
2. Было показано преимущество использования модифицированной оценочной функции по сравнению с традиционной.

ЗАКЛЮЧЕНИЕ

В данной работе получены следующие результаты.

1. Построен метод, оптимизирующий работу генетического алгоритма путем модификации оценочной функции.
2. Показано преимущество использования модифицированной фитнесс-функции.
3. Приведено экспериментальное сравнение модифицированной оценочной функции приспособленности с обычной фитнесс-функцией, построенной для задачи об умном муравье.

ИСТОЧНИКИ

1. *Поликарпова Н. И., Точилин В. Н., Шалыто А. А.* Разработка библиотеки для генерации управляющих автоматов методом генетического программирования. http://is.ifmo.ru/download/polikarpova(LETI).pdf

2. *Бедный Ю. Д., Шалыто А. А.* Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей». http://is.ifmo.ru/works/_ant.pdf

3. *Мандриков Е. А., Кулев В. А., Шалыто А. А.* Построение автоматов с помощью генетических алгоритмов для решения задачи о «флибах» / Сборник докладов X-ой Международной конференции по мягким вычислениям и измерениям. Том 1. СПбГЭТУ «ЛЭТИ». 2007, с. 293–296. http://is.ifmo.ru/download/flibs.pdf

4. *Царев Ф. Н., Шалыто А. А.* Применение генетического программирования для генерации автомата в задаче об «умном муравье» / Сборнике трудов IV-ой Международной научно-практической конференции «Интегрированные модели и мягкие вычисления в искусственном интеллекте». Том 2. М.: Физматлит. 2007, с. 590–597. http://is.ifmo.ru/genalg/_ant_ga.pdf

5. *Kent S., Dracopoulos D.*. Speeding up Genetic Programming: A Parallel BSP implementation. Brunel University, Department of Computer Science and Information Systems.

6. *Schmidhuber J.* Evolutionary principles in self-referential learning. Institut f. Informatik, Tech. Univ. Munich. 1987, с. 7–13.

7. *Edmonds B.* Meta-Genetic Programming: Co-evolving the Operators of Variation. Centre for Policy Modeling, Manchester Metropolitan Univ, 2001.

8. *Diosan L., Oltean M.* Evolving Crossover Operators for Function Optimization. Department of Computer Science, Faculty of Mathematics and Computer Science, Babes-Bolyai University, 2006.

9. *Oltean M.* Evolving Evolutionary Algorithms using Linear Genetic Programming. http://www.scribd.com/doc/260024/Evolving-Evolutionary-Algorithms-using-Linear-Genetic-Programming

10. O*ltean M.* Evolving Evolutionary Algorithms using Multi Expression Programming. http://www.scribd.com/doc/260631/Evolving-Evolutionary-Algorithms-using-Multi-Expression-Programming

11. *Whitley D., Richards M., Beveridge R.* Alternative evolutionary algorithms for evolving programs: evolution strategies and steady state GP. http://www.cs.bham.ac.uk/~wbl/biblio/gecco2006/docs/p919.pdf

12. *Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A*. The Genesys System. 1992.

Приложение. Исходные коды

AntFitnessFunction.java

```
package org.gaap.examples.ant;

import org.gaap.ga.FitnessFunction;

public final class AntFitnessFunction implements FitnessFunction {

    private double[] params;
    private double[] max;
    public static final double C = 0.0001;

    public AntFitnessFunction(double[] params) {
        this.params = params;
        max = fill(new double[params.length], 1, params.length);
    }

    private double[] fill(double[] array, double value, int count) {
        for (int i = 0; i < count; i++) {
            array[i] = value;
        }
        return array;
    }

    public AntFitnessFunction() {
        this.params = fill(new double[params.length], 1, 1);
    }

    @Override
    public double calculate(Object[] data) {
        double res = 0;

        int paramsCount = data.length;
        if (params.length != paramsCount) {
            throw new IllegalStateException("Wrong number of parameters");
        }

        for (int i = 0; i < paramsCount; i++) {
            double t = ((Integer) data[i]).doubleValue();

            if (t > max[i]) {
                max[i] = t;
            }
            if (params[i] == C) {
                res += t;
            } else if (params[i] == -C) {
                res += 1 / t;
            } else if (params[i] > 0) {
                res += params[i] * t;
            } else if ((params[i] < 0) && (t > 0)) {
                res += (-params[i]) * (max[i] - t) / max[i];
            }
        }
        return res;
    }
}
```

TestRunner.java

```
package org.gaap.examples.ant;

import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
import org.gaap.CommandLineLoader;
import org.gaap.fsm.impl.simple.SimpleCrossover;
import org.gaap.fsm.impl.simple.SimpleMutation;
import org.gaap.fsm.impl.simple.SimpleOperators;
import org.gaap.ga.Ga;
import org.gaap.ga.Offspring;
import org.gaap.ga.impl.DefaultBreeder;
import org.gaap.ga.impl.DefaultGa;
import org.gaap.ga.impl.selectors.RouletteSelector;
import org.gaap.ga.utils.LeveledListener;

import java.net.MalformedURLException;
import java.net.URL;

public class TestRunner {

    private static Logger LOG = Logger.getLogger(TestRunner.class);

    private static AntTester antTester;
    private static AntFoodDataProvider antFood = new AntFoodDataProvider();
    private static DefaultGa ga = new DefaultGa();

    private static int iterationCount = 0;
    private static int repetitions = 2;
    private static int maxIteration = 30;

    private static double[] result;
    private static double e = 0.1;

    public static void main(String[] args) throws MalformedURLException {
        run();
    }

    private static void run() throws MalformedURLException {

        URL log =
        CommandLineLoader.class.getResource("/gaap-examples-log4j.xml");
        if (log != null) {
            DOMConfigurator.configure(log);
        }

        ga.addListener(new GaListener());
        ga.selectors.add(new RouletteSelector(0.5));
        ga.selectors.add(new RouletteSelector(0.5));
        ga.operators.add(new SimpleCrossover());
        ga.operators.add(new SimpleMutation());


         antTester = new AntTester
                (TestRunner.class.getResource("john-muir.txt"));
        result = new double[antTester.getFieldNames().length];
        double[][] testCases = createTestCases(result.length);
```

```
ga.breeder = new DefaultBreeder(antTester, 0.9);
double[] testResults = new double[2 * antTester.getFieldNames().length];

for (int j = 0; j < testCases.length; j++) {
    LOG.info("TestCase: " + (j + 1));
    testResults[j] = testFunction(testCases[j]);
}

//Вычисление главных характеристик
result[0] = AntFitnessFunction.C;
for (int k = 1; k < testResults.length; k++) {
    if ((testResults[k] > testResults[0]) ||
        Math.abs(testResults[k] - testResults[0]) < e) {
        if (k < result.length) {
            result[k] = AntFitnessFunction.C;
        } else {
            if (testResults[k] > testResults[k % result.length]) {
                result[k % result.length] = -AntFitnessFunction.C;
            }
        }
    }
}

double oldResult;
double newResult;
int paramCount;

//вычисление второстепенных характеристик
do {
    newResult = testFunction(result);
    oldResult = newResult;
    paramCount = 0;

    for (int p = 0; p < result.length; p++) {
        if (result[p] == 0) {
            paramCount++;

            result[p] = 1;
            double test1 = testFunction(result);
            double test1c = test1 / newResult;


            result[p] = -1;
            double test2 = testFunction(result);
            double test2c = test2 / newResult;
            if (test1c > 1) {
                result[p] = test1c;
                newResult = test1;
            }

            if (test2c > test1c && test2c > 1) {
                result[p] = -test2c;
                newResult = test2;
            }

            if (result[p] == 1 || result[p] == -1) {
                result[p] = 0;
            }
        }
    }

    LOG.info("Old Result: " + oldResult);
    LOG.info("New Result: " + newResult);
```

```
        } while (Math.abs(newResult - oldResult) > 0.05 || paramCount != 0);

        String res = "";
        for (double aResult : result) {
            res = res + ", " + aResult;
        }

        LOG.info("Result coefficients: " + res);
    }


    private static double[][] createTestCases(int l) {
        double[][] testCases = new double[2 * l][l];

        for (int y = 0; y < l; y++) {
            testCases[y][y] = AntFitnessFunction.C;
        }
        for (int y = l; y < 2 * l; y++) {
            testCases[y][y % result.length] = -AntFitnessFunction.C;
        }

        return testCases;
    }

    private static double testFunction(double[] function) {
        double testResult = 0;
        for (int r = 0; r < repetitions; r++) {
            iterationCount = 0;
            ga.breeder.setFitnessFunction(new AntFitnessFunction(function));
            ga.setPopulation(ga.breeder.breed(new Offspring(
               SimpleOperators.random(15, 2, 1, 2000).toArray()), null));
            double coefficient = (Double) (antFood.getData(ga))[0] +
               (((Double) (antFood.getData(ga))[1]) / 4);
            ga.run();
            double max = (Double) (antFood.getData(ga))[0];
            double average = ((Double) (antFood.getData(ga))[1]) / 4;
            coefficient = (max + average) / coefficient;
            testResult += coefficient;
        }
        testResult = testResult / repetitions;
        return testResult;
    }

    private static class GaListener implements LeveledListener<Ga> {
        public int getLevel() {
            return Ga.LEVEL_MODIFY;
        }

        public void fired(Ga sender) {
            if (sender != ga) {
                throw new IllegalStateException();
            }

            iterationCount++;
            LOG.info("Iteration " + iterationCount);
            Number[] b = antFood.getData(ga);
            LOG.info("Food: " + b[0] + ":" + b[1] + ":" + b[2] + "\n");

            if ((iterationCount == maxIteration)) {
                ga.stop();
            }
        }
    }
}
```