Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Рост Аркадий Юрьевич

Методы автоматизированного покрытия кода тестами на основе эволюционных алгоритмов

Научный руководитель: ассистент кафедры ТП М. В. Буздалов

Санкт-Петербург
2013

Содержание

Введение

При разработке современного программного обеспечения значительную часть времени и усилий занимает процесс тестирования. Зачастую он сводится к анализу поведения программы на специально подобранном наборе тестов. Поскольку перебор всех возможных ситуаций неосуществим на практике, стараются выбрать небольшое число тестов, таких чтобы по поведению программы на них можно было судить о ее поведении в целом.

Используются различные метрики покрытия кода для количественной оценки качества набора тестов. Качественным считается набор тестов, удовлетворяющий некоторому критерию полноты. Зачастую критерий полноты определяется с помощью выбранной метрики покрытия кода.

Автоматизация процесса построения набора тестов позволит существенно сократить затраты на тестирование. Существует множество подходов к автоматизированному построению набора тестов. Однако лишь немногие из них имеют программную реализацию, например Microsoft Pex. Все эти реализации нацелены на модульное тестирование, при котором покрытие фрагментов программы происходит независимо друг от друга. Вследствие этого большое число генерируемых тестов содержат недопустимые значения для тестируемых фрагментов кода, так как не учитывается внутренняя структура программы.

При тестировании некоторых программ, таких как решения олимпиадных задач, нет необходимости проводить модульное тестирование. Для полноценного покрытия тестами олимпиадных задач необходимо генерировать тесты, подходящие под условие задачи. В данной работе разрабатывается метод построения набора тестов на основе эволюционных алгоритмов, учитывающий особенности всей программы в целом.

Глава 1. Обзор предметной области

1.1. Метрики покрытия кода

Целью тестирования программного обеспечения является проверка требуемой функциональности. Существуют различные виды тестирования, например тестирование производительности или безопасности. Для оценки того, насколько хорошо данный набор тестов покрывает код, используются различные метрики. Зачастую полное покрытие кода в рамках выбранной метрики невозможно, поэтому набор тестов считается хорошим, если он покрывает 90 %–95 % кода.

Существует несколько различных способов задать метрику покрытия кода [1, 2]. Основные из них:

- покрытие классов/методов — проверяется, что каждый класс/метод вызывается;
- покрытие строк кода — проверяется, что каждая строка программы выполняется;
- покрытие условий — перебираются все значения условий;
- покрытие траекторий выполнения — все ли возможные траектории через заданную часть кода были пройдены.

Метрика покрытия классов/методов не применима для тестирования олимпиадных задач, поскольку при использовании данной метрики может быть не выявлено большое число ошибок. Однако, применение этих метрик оправдано при необходимости покрыть код, часть которого закрыта или не может быть профилирована по лицензионным соображениям.

В отличие от метрики покрытия строк кода, покрытие условий требует рассмотреть все варианты выполнения условия. Например, для условия $x \vee y$ при использовании метрики покрытия условий требуется рассмотреть варианты:

1. $x =$ `true`;

2. $x =$ `false`, $y =$ `true`;

3. $x =$ `false`, $y =$ `false`.

Однако, используя метрику покрытия строк кода, можно ограничиться рассмотрением первого и третьего вариантов.

Для рассмотрения всех возможных траекторий обычно требуется перебрать большее число вариантов, чем при покрытии условий. При этом число траекторий может расти экспоненциально, что требует большого времени выполнения тестов. Поэтому данную метрику применяют лишь при разработке систем с повышенными требованиями к надежности. В данной работе используется метрика покрытия условий.

1.2. Эволюционные алгоритмы

Эволюционный алгоритм [3] — это метод решения задач оптимизации. Данный подход основан на идеях, заимствованных из биологической эволюции: естественный отбор, мутация, скрещивания и наследование признаков. Каждая итерация алгоритма характеризуется набором особей, называемым поколением. На множестве особей вводят функции приспособленности, чтобы количественно оценивать, насколько заданная особь близка к верному решению. При помощи оператора скрещивания (кроссовера) по двум особям генерируется особь для следующего поколения. Оператор мутации вносит малые случайные изменения особи. Начальное поколение обычно формируется случайным образом. При выборе особей для создания нового поколения наиболее приспособленные имеют больше шансов. Общая схема эволюционного алгоритма представлена на листинге 1.2.1.

В качестве критерия останова часто используют следующие условия:

- найдено верное решение;
- достигнуто заданное количество поколений;

Листинг 1.2.1 Общая схема эволюционного алгоритма

```
1: Создать начальное поколение
2: Вычислить значение функции приспособленности для каждой особи
3: while (условие останова эволюционного алгоритма не выполнено) do
4:     Выбирается подмножество особей текущего поколения
5:     Применяя операторы мутации и кроссовера к выбранным особям, генерируются новые
6:     Вычисляется значение функции приспособленности для сгенерированных особей
7:     Формируется новое поколение, заменяя новыми особями наименее приспособленных
8: end while
```

- превышено заданное время работы;
- превышено заданное число вызовов функции приспособленности;
- за заданное число поколений не произошло улучшение.

Эволюционные алгоритмы применяются для решения задач, к которым не применимы традиционные методы оптимизации. Одной из областей применения эволюционных алгоритмов является автоматическая генерация тестов для программного обеспечения.

1.3. Существующие подходы

Рассмотрим некоторые методы, применяемые для автоматической генерации покрывающего набора тестов.

1.3.1. Применение алгоритмов оптимизации

Покрытие фрагмента кода тестом можно рассматривать как задачу оптимизации. В качестве оптимизируемой функции используется количественная оценка того, насколько сгенерированный тест покрывает заданный фрагмент кода.

Рассмотрим некоторые алгоритмы, применяемые для решения данной задачи [4, 5].

- Алгоритм восхождения на вершину (*Hill climbing*) [6]. На каждой итерации алгоритма генерируется набор "ближайших соседей", полученный при помощи небольших изменений текущего кандидата. Существует два способа выбора кандидата для следующей итерации. Можно перебирать соседей, пока не найдется лучший кандидат,

либо выбрать лучшего из всех соседей. Алгоритм завершает работу, если невозможно улучшить текущее решение.

- Метод отжига [7]. В отличие от предыдущего алгоритма может выбрать худшее решение с некоторой вероятностью. Эта вероятность уменьшается с номером итерации алгоритма. Вследствие этого данный подход менее подвержен схождению в локальному оптимуму вместо глобального.
- Применение эволюционных алгоритмов для построения набора тестов получило широкое распространение [8—10]. Используются различные эволюционные операторы, функции приспособленности, а также по-разному кодируются особи. Однако не существует оптимального подхода.

1.3.2. Символьное выполнение

При символьном выполнении [11] моделируется выполнение программы, при котором часть входных переменных представляется в символьном виде. Можно выделить два основных подхода [12]: на основе статического или динамического анализа программы.

При статическом подходе для каждой возможной траектории выполнения с помощью символьного выполнения задаются ограничения на параметры тестируемой программы. Если для некоторой траектории возможно удовлетворить все ограничения, то вдоль нее будет сгенерирован тест.

Однако статический подход неприменим, если тестируемая программа содержит ограничения вне области видимости статического анализатора. Рассмотрим программу, приведенную на листинге 1.3.1.

Листинг 1.3.1: Пример, на котором не применим статический подход

```
def testMethod(x, y) {
    if (x == hash(y)) {
        return 0     // целевое ветвление
    }
    return 1
}
```

Допустим, что в целях безопасности, анализатор кода не имеет доступ к функции `hash`. Значит, без запуска программы невозможно подобрать `x` так, чтобы он был равен `hash(y)`.

Для обхода таких ситуаций используется динамический подход. При этом генерация теста происходит следующим образом:

1. Тестируемая программа выполняется на случайных входных данных.
2. Динамический анализатор записывает ограничения на входные параметры тестируемой программы вдоль траектории выполнения.
3. Удовлетворив ограничениям, подбирается такой тест, чтобы следующий запуск прошел по другой траектории.

Таким образом, при запуске программы 1.3.1 на случайных значениях, будет посчитано `hash(y)`. Взяв посчитанное значение `hash(y)` вместо `x`, можно удовлетворить ограничениям.

При использовании данного подхода возникают проблемы с масштабированием [13]. При тестировании более сложных программ возрастает как длина траекторий выполнения, так и число символьных переменных. При увеличении количества ограничений, накладываемых на параметры тестируемой программы, существенно увеличивается время, затрачиваемое на их удовлетворение.

1.4. Java Virtual Machine

В данной работе рассматривается покрытие тестами программ, работающих на *Java Virtual Machine (JVM)*. *JVM* [14] — основа *Java* платформы, отвечающая за аппаратную и операционную независимость программ. *JVM* имеет определенный набор инструкций и управляет памятью во время выполнения программы. *JVM* работает с файлами определенного бинарного формата, называемыми *class*-файлами, в которых содержатся последовательность инструкций, таблица символов и другая вспомогательная информация.

1.4.1. Типы данных

Типы данных *JVM* делятся на примитивные и ссылочные. Для каждой инструкции определен тип аргументов, к которым она применяется. Например, инструкции `iadd`, `ladd`, `fadd` и `dadd` складывают два численных значения и возвращают результат, но каждая из этих инструкций предназначена для определенного типа данных: `int`, `long`, `float` и `double` соответственно.

Примитивные типы данных делятся на численные, логический (`boolean`) тип и тип адреса возврата (`returnAddress`). В свою очередь, числовые типы данных делятся на:

- целочисленные
  - `byte`, от -127 до 128
  - `short`, от -2^{15} до $2^{15}-1$
  - `int`, от -2^{31} до $2^{31}-1$
  - `long`, от -2^{63} до $2^{63}-1$
  - `char`, символ Unicode, 2 байта
- числа с плавающей точкой, соответствующие стандарту IEEE 754
  - `float`, 32-битное вещественное число одинарной точности
  - `double`, 64-битное вещественное число двойной точности

Значениями, имеющими тип адреса возврата, являются указатели на инструкции *JVM*. В отличие от данных численных типов, значения типа адреса возврата не могут быть изменены в процессе работы программы.

Несмотря на то, что *JVM* декларирует тип `boolean`, для него предоставляется лишь ограниченная поддержка. *JVM* не имеет инструкций, специально предназначенных для значений типа `boolean`. Для работы со значениями типа `boolean` используются инструкции, предназначенные для работы со значениями типа `int`, причем `true` соответствует единица, а `false` — ноль.

Ссылочные типы данных делятся на классы, массивы и интерфейсы. Данными, имеющими такие типы, соответственно являются динамически создаваемые экземпляры классов, массивов, а также экземпляры классов и массивы, которые реализуют заданный интерфейс.

В процессе выполнения программы *JVM* некоторые типы данных хранятся одинаково, причем размер выделяемой памяти зависит от категории типа. Способ хранения типы данных зависит от соответствующего ему типа данных времени выполнения. Соответствие фактических типов данных и типов времени выполнения, а также категории указаны в таблице 1.1.

Таблица 1.1: Таблица соответствий типов данных *JVM*

Фактический тип	Тип времени выполнения	Категория
`boolean`	`int`	1
`byte`	`int`	1
`char`	`int`	1
`short`	`int`	1
`int`	`int`	1
`float`	`float`	1
`reference`	`reference`	1
`returnAddress`	`returnAddress`	1
`long`	`long`	2
`double`	`double`	2

1.4.2. Среда выполнения

Во время работы *JVM* использует память двух видов: куча и стек.

Куча предназначена для хранения экземпляров классов и массивов. Очистка кучи происходит автоматически с помощью сборщика мусора. Куча общая для всех потоков *JVM* и инициализируется при старте *JVM*.

В отличие от кучи, стек для каждого потока свой. Его инициализация происходит в момент создания потока.

При выполнении программы на каждый вызов метода создается фрейм — данные, необходимые для выполнения метода — который кладется на стек. После завершения выполнения метода фрейм удаляется со стека, вне зависимости от того, было ли брошено необработанное исключение или нет.

Фрейм, соответствующий методу, выполняемому в данный момент, называется активным. После завершения текущего метода, активным становится предыдущий фрейм. Каждый фрейм выделяется локально для каждого потока, поэтому остальные потоки не могут на него ссылаться.

Каждый фрейм содержит:

- набор локальных переменных;
- стек операндов;
- ссылку на набор констант текущего метода.

Локальные переменные пронумерованы, начиная с нуля, и обращение к ним происходит по соответствующему номеру. *JVM* использует локальные переменные для передачи параметров при вызове метода, причем нулевым параметром является ссылка на объект, чей метод был вызван. Для хранения значений типа, относящегося первой категории, используется одна локальная переменная, а для значений типа, относящегося ко второй категории, используются две последовательные локальные переменные.

Максимальный размер стека операндов определяется на этапе компиляции. При выполнении инструкции аргументы забираются со стека операндов и результат выполнения кладется на стек. Кроме того, стек операндов используется для передачи аргументов вызываемым методам. При вызове метода аргументы забираются со стека операндов текущего фрейма, создается новый фрейм и аргументы сохраняются в локальных переменных созданного фрейма. Результат кладется на стек операндов того фрейма, из которого был вызван метод. После этого созданный фрейм удаляется. Значения типов данных, относящихся первой категории, занимают одну ячейку стека операндов, в то время как значения типов, относящихся второй категории — две.

Каждый класс и интерфейс имеет набор констант. В нем содержатся как численные константы, известные на момент компиляции, так и ссылки на методы и поля, значения которых вычисляются во время выполнения.

1.4.3. Набор инструкций

Перечислим инструкции *JVM*, которые затронуты в данной работе.

1.4.3.1. Инструкции сохранения и загрузки

Инструкции загрузки и сохранения предназначены для передачи значений между стеком операндов и локальными переменными:

- Загрузка значений из локальной переменной на стек операндов: `iload`, `fload`, `dload`, `lload`, `aload`;
- Сохранение значений со стека операндов в локальные переменные: `istore`, `fstore`, `dstore`, `lstore`, `astore`;
- Загрузка констант на стек операндов: `bipush`, `sipush`, `ldc`, `aconst_null`, `iconst`, `fconst`, `lconst`, `dconst`.

1.4.3.2. Инструкции ветвления

- Сравнение двух операндов типа `int`: `if_icmpeq` ($=$), `if_icmpne` ($\neq$), `if_icmplt` ($<$), `if_icmpge` ($\geq$), `if_icmpgt` ($>$), `if_icmple` ($\leq$);
- Сравнение операнда типа `int` с нулем: `ifeq`, `ifne`, `iflt`, `ifge`, `ifgt`, `ifle`;
- Сравнение ссылочных типов: `if_acmpeq`, `if_acmpne`, `ifnonnull`, `ifnull`;
- Сравнения: `dcmpg`, `dcmpl`, `fcmpg`, `fcmpl`, `lcmp`.

1.5. Scala

Scala [15] — статически типизированный язык программирования, сочетающий в себе возможности объектно-ориентированного и функционального программирования. Система типов *Scala* специально спроектирована для удобства создания компонентного программного обеспечения. При этом *Scala* полностью совместима с *Java*.

В отличие от *Java*, в *Scala* имеется поддержка типажей (`trait`), которые подобны интерфейсам, но могут содержать реализацию методов. Как и в случае с интерфейсами, возможно множественное наследование от типажей. При определении типажа можно указывать дополнительные типажи, необходимые создания его экземпляра. Это можно использовать для конфигурации эволюционного алгоритма (ЭА).

Рассмотрим пример, приведенный на листинге 1.5.2. При создании экземпляра ЭА (*EvolutionaryAlgorithm*) необходимо указать оператор мутации (*Mutation*).Таким образом, можно использовать один и тот же ЭА, но с различными операторами мутации. В приведенном примере определено два типажа мутации целого числа: *IncrementMutation* и *SquareMutation*. Экземпляры ЭА — *incEA* и *squareEA* — различаются используемыми операторами мутации.

Листинг 1.5.2: Пример конфигурации эволюционного алгоритма

```
trait Mutation[G] {
  def mutate(g : G) : G
}

trait EvolutionaryAlgorithm[G] {
  needs : Mutation[G] => // определение необходимого типажа
  ...
}

trait IncrementMutation extends Mutation[Int] {
  def mutate(g : Int) = g + 1
}

traint SquareMutation extends Mutation[Int] {
  def mutate(g : Int) = g * g
}

val incEA = new EvolutionaryAlgorithm[Int] with IncrementMutation

val squareEA = new EvolutionaryAlgorithm[Int] with SquareMutation
```

1.6. Выводы по главе 1

Рассмотрены некоторые существующие подходы к автоматизированному покрытию кода тестами. Описаны различные структурные метрики покрытия кода. Кратко описана спецификация *JVM*. Кратко описаны особенности языка программирования *Scala*.

Глава 2. Описание реализованного подхода

В данной главе описывается разработанный метод автоматизированного покрытия кода тестами на основе эволюционных алгоритмов.

2.1. Постановка задачи

Целью данной работы является создание платформы для автоматизированного покрытия программ тестами, учитывающими внутреннюю структуру тестируемой программы, на основе эволюционных алгоритмов и проверка разработанного метода на ряде модельных задач. Требования к данной работе:

- Разработать метод автоматизированного покрытия тестами кода программ, работающих на *JVM*.
- Провести сравнение разработанного метода с методом генерации случайных тестов на ряде модельных задач.
- Апробировать предложенный подход для покрытия тестами решения олимпиадной задачи Huzita Axiom 6.

2.2. Общая схема решения

На рисунке 2.1 показана общая схема решения. При загрузке *class*-файла тестируемой программы происходит модификация кода с целью получения траектории выполнения, а также считывания значений, переданных инструкциям ветвления, в процессе выполнения модифицированной программы. Заметим, что модификация кода тестируемой программы происходит таким образом, чтобы не изменился результат ее работы.

После загрузки модифицированного кода происходит инициализация списка ветвлений, покрываемых инструкций. Будем называть целью

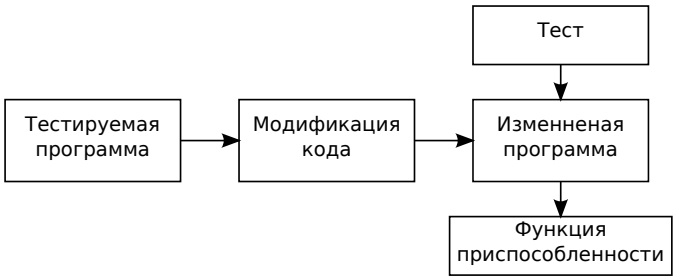

Рис. 2.1: Общая схема алгоритма

заданное ветвление выбранной инструкции. Дальнейшее решение сводится к поиску набора тестов, обеспечивающих покрытие каждой цели.

Построение теста, покрывающего заданную цель, рассматривается как задачу оптимизации, решаемая с помощью эволюционных алгоритмов (ЭА). В качестве особи ЭА кодируется тест. С учетом требований к искомому тесту для особи определяются необходимые эволюционные операции: мутация и кроссовер.

Для покрытия каждой цели используется свой экземпляр ЭА. Экземпляры ЭА создаются лишь для непокрытых целей. Если в процессе работы находится тест, покрывающий ранее непокрытую цель, то такой тест сохраняется.

После получения набора тестов, покрывающего выбранные цели, производится его минимизация с целью уменьшения накладных расходов при дальнейшем тестировании.

Таким образом, для запуска алгоритма необходимо:

- задать список целей;
- сконфигурировать ЭА;
- обеспечить возможность многочисленных расчетов функции приспособленности.

2.3. Функция приспособленности

Функция приспособленности (ФП) дает количественную оценку того, насколько заданный тест покрывает выбранную цель. В данной работе ФП выбирается так, чтобы оценивать, насколько близка траектория выполнения программы к заданной цели.

2.3.1. Функция расстояния до ветви

У некоторых инструкций ветвления лишь определенные ветви способствуют покрытию тестами выбранной цели, поскольку только они приводят к выполнению нужного фрагмента кода. Функция расстояния до ветви определяется таким образом, что ее минимальное значение достигается при прохождении по желаемому ветвлению.

Рассмотрим построение функции расстояния до нужной ветви на примере метода, код которого приведен на листинге 2.3.1.

Листинг 2.3.1: Пример функции расстояния до ветви

```
def testMethod(x : Int) {
  if (10 <= x && x <= 15) {
    A()      // целевое ветвление
  }
  B()
}
```

На рисунке 2.2 представлена блок-схема тестируемого метода. Заметим, что условие $10 \leq x \leq 15$ заменяется двумя инструкциями ветвления на этапе компиляции. В качестве функции расстояния до ветви в данном случае подойдут $10 - x$ и $x - 15$ для первой и второй инструкции ветвления соответственно.

Качество теста оценивается в зависимости от выбора функции расстояния до ветви. Так, например, сравнение значений типов long, double и float происходит при помощи специализированных инструкций, как `lcmp`, `dcmpg`, `dcmpl`, `fcmpg`, `fcmpl`. Результатом выполнения этих инструкций будет ноль, единица или минус единица и ветвление потока управления происходит с использованием соответствующих инструкций сравнения с ну-

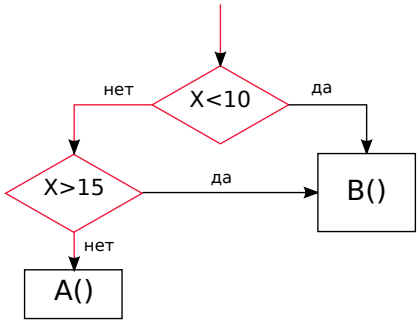

Рис. 2.2: Блок-схема для листинга 2.3.1. Малиновым цветом выделена целевая траектория выполнения.

лем. Если в качестве значения функции расстояния до ветви использовать значения, переданные в инструкции ветвления напрямую, то эволюционный алгоритм будет работать плохо. Вместо них лучше использовать значения, переданные непосредственно в специализированные инструкции сравнения. Поэтому при подсчете функции расстояния до ветви необходимо анализировать код тестируемой программы.

2.3.2. Следование опорной траектории

В данном случае предполагается, что траектория выполнения тестируемой программы должна совпадать с выбранной опорной траекторией. Опорная траектория либо задается человеком, либо генерируется автоматически на основе графа потока управления.

Рассмотрим опорную траекторию, состоящую из целей $C_1, C_2, .., C_k$. Пусть на некотором тесте была получена траектория $C_1, C_2, .., C_t, J, ..,$ где $J \neq C_{t+1}$. ФП представляется в виде пары двух чисел. Первое — длина общего префикса двух траекторий, а именно `t`. В качестве второго числа берется значение функции расстояния до ветви для цели C_t, поскольку именно в ней разошлись две траектории. При сравнении двух особей более приспособленной считается та, у которой первое число ФП больше. Если первые числа ФП равны, то лучше та особь, у которой второе число меньше.

Рассмотрим пример на рис. 2.3. Опорная траектория выделена малиновым цветом. Зелеными кружками помечена траектория выполнения программы. Инструкция, отмеченная темно-зеленым кружком, соответствует C_t и в ней происходит расхождение траекторий. Длина общего префикса отмеченных траекторий равна трем.

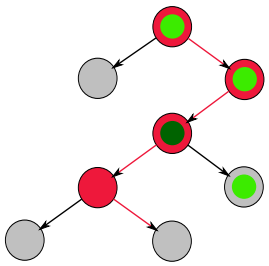

Рис. 2.3: Пример расхождения с опорной траекторией

Существуют другие варианты построения функции приспособленности на основе следования опорной траектории. Допустим, ни одна траектория выполнения программы не может следовать выбранной опорной траектории во всех инструкциях. В таком случае невозможно найти тест, покрывающий выбранную цель. Однако если в качестве первого числа функции приспособленности взять наибольший номер инструкции, общей для двух траекторий, то такой тест может быть сгенерирован.

Недостатком такого подхода является необходимость выбора опорной траектории, поскольку возможно большое число траекторий, проходящих по заданному ветвлению. При этом время работы алгоритма сильно зависит от выбора опорной траектории, так как не вдоль всех траекторий можно сгенерировать тест.

2.3.3. Приближение по многим направлениям

Обычно при покрытии выбранной цели нет предпочтений насчет траектории, вдоль которой надо покрывать. Таким образом, необходимо

выбрать опорную траекторию, либо задать функцию приспособленности, рассматривающую приближение вдоль всех возможных траекторий.

Расстоянием `d(x, y)` между инструкцией `x` и инструкцией `y` будем считать минимальную длину пути от инструкции `x` до инструкции `y` в графе потока управления. Если такой путь не существует, то $d(x, y) = \infty$. В качестве расстояния `d(T, x)` от траектории $T = t_1, t_2, ..., t_k$ до инструкции `x` возьмем $\min\limits_{i=1..k} d(t_i, x)$.

Пусть выбрана цель `c` и заданному тесту `t` соответствует траектория выполнения `T`. Тогда в качестве значения функции приспособленности берется пара чисел, первое из которых равно `d(T, c)`, а второе — минимальному значению функции расстояния до ветви для инструкции `i` на траектории `T`, такой что $d(i, c) = d(T, c)$.

Рассмотрим пример на рис. 2.4. На нем изображены две различные траектории выполнения одного и того же фрагмента кода, выделенные зеленым цветом. Цель `c` выделена малиновым цветом. Темно-зеленым цветом выделена инструкция `i`. Обе траектории находятся на расстоянии до цели, равном одному условному переходу.

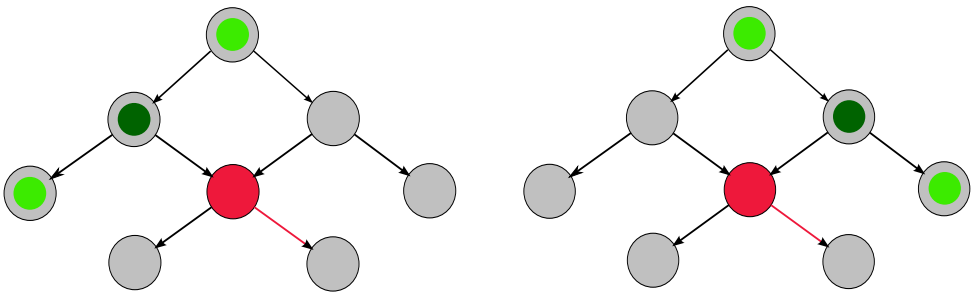

Рис. 2.4: Пример приближения по многим направлениям

В данном примере, при совпадении второго числа в функции приспособленности, количественные оценки для обеих траекторий будут одинаковые, но качественно эти траектории различаются. Таким образом, при данном определении функции приспособленности одинаковые количественные оценки могут соответствовать качественно различным тестам.

2.4. Модификация кода

Модификации кода тестируемой программы не должны изменять результат ее работы. Для работы с *class*-файлами используется библиотека ASM 4.0 [16], предоставляющая доступ к списку инструкций. Базовый интерфейс применяемых модификаций приведен на листинге 2.4.2.

Листинг 2.4.2: Типаж модификации кода тестируемой программы

```
import org.objectweb.asm.tree.{ClassNode, MethodNode}

trait CodeModification {
  protected def onMethod(cn : ClassNode, mn: MethodNode)

  def onClass(cn : ClassNode) {
    cn.methods.iterator.foreach{ x =>
      onMethod(cn, x.asInstanceOf[MethodNode])
    }
  }
}
```

Для модификации кода метода необходимо переопределить метод `onMethod`, входными аргументами которого являются ссылки на инструкции модифицируемого метода и класса, в котором этот метод определен. Для композиция двух модификаций метод `onMethod` определяется как последовательность вызовов методов `onMethod` этих модификаций. Класс композиции представлен на листинге 2.4.3.

Листинг 2.4.3: Композиция модификаций кода тестируемой программы

```
import org.objectweb.asm.tree.{ClassNode, MethodNode}

class Composition(a : CodeModification, b : CodeModification) extends CodeModification {
  protected def onMethod(cn : ClassNode, mn: MethodNode) {
    a.onMethod(cn, mn)
    b.onMethod(cn, mn)
  }
}
```

Работа с кодом программы разделена на два этапа: на первом формируется внутреннее представление программы в виде графа потока управления, на втором — код модифицируется с целью считывания траектории выполнения программы и другой информации о процессе выполнения.

2.4.1. Граф потока управления

Библиотека `ASM` предоставляет доступ к списку инструкций каждого метода. Однако для структурного тестирования необходимо построение графа потока управления.

Определение 2.1. Графом потока управления программы `F` называется ориентированный граф $G = (N, E, s, F)$, где `N` — множество вершин, `E` — множество ребер, `s` — входная инструкция, `F` — множество вершин, завершающих выполнение программы.

Каждой вершине $n \in N$ графа потока управления соответствует инструкция *JVM*. Ребром $e = (n_i, n_j)$ же является переход от инструкции n_i к инструкции n_j.

Для построения графа потока управления используется типаж анализатора кода, представленный на листинге 2.4.4. При помощи дополнительных типажей анализатор индексирует инструкции ветвления, а также сохраняет построенный граф потока управления.

Листинг 2.4.4: Типаж анализатора кода

```
import org.objectweb.asm.tree.{ClassNode, MethodNode}

trait AsmControlFlowAnalyzer extends CodeModification {
  needs : InstructionRegister with MethodRegister =>

  protected override def onMethod(cn : ClassNode, mn: MethodNode) {
    ...
  }
}
```

2.4.2. Считывание траектории выполнения программы

Для анализа выполнения программы на заданном тесте считывается траектория выполнения, а именно последовательность инструкций ветвле-

ния и значения переданные им во время выполнения. Для идентификации инструкций ветвления используется нумерация, введенная при построении графа потока управления.

Для считывания значений во время выполнения программы определен ряд методов, каждый из которых соответствует определенному типу инструкции ветвления. При работе с несколькими потоками последовательность инструкций ветвления будет составлена для каждого потока в отдельности [17].

Рассмотрим модификацию кода программы, необходимую для считывания траектории выполнения. Для каждого профилируемого метода вводятся дополнительные локальные переменные. Перед выполнением инструкции ветвления выполняются следующие действия:

1. Значения со стека операндов сохраняются в дополнительные локальные переменные.
2. Состояние стека операндов восстанавливается из локальных переменных.
3. Вызывается метод для считывания значений во время выполнения.
4. Восстанавливается состояние стека операндов для дальнейшей работы программы.

2.5. Минимизация набора тестов

Минимизация набора тестов является частным случаем задачи о минимальном покрытии множества. Пусть имеется множество тестов T. Для каждого теста $t \in T$ известно множество фрагментов кода, которые он покрывает. Требуется построить минимальное по мощности множество $T_{opt} \subset T$, такое что: $\bigcup\limits_{t \in T_{opt}} Cov(t) = \bigcup\limits_{t \in T} Cov(t)$.

В работе [18] показано, что данная задача является NP-полной. По причине чрезмерных затрат вычислительных ресурсов требуется применять приближенные методы.

Рассмотрим жадный алгоритм, решающий данную задачу:

1. Пусть множество X — множество выбранных тестов, $P = \bigcup_{t \in T} Cov(t)$, а множество Z — множество покрытых им фрагментов кода. Изначально $X, Z = \emptyset$.
2. Если $Z = P$, прекратить работу, решение задачи $T_{opt} = X$.
3. Выбрать такое $t \in T$, что $Cov(t) \cap (P/Z)$ максимально.
4. $X = X \cup t$, $Z = Z \cup Cov(t)$, перейти к шагу 2.

Время работы простейшая реализация данного алгоритма имеет асимптотическую оценку $O(|T|^2 \cdot |P|)$. В работе [19] показано, что данный алгоритм генерирует в худшем случае ответ, превосходящий оптимальный в $O(log|P|)$ раз. Однако на практике для многих входных данных он дает ответ, отличающийся от оптимального не более чем на 10%.

2.6. Выводы по главе 2

Формализована цель работы: создание платформы для автоматизированного покрытия программ тестами на основе эволюционных алгоритмов. Описан предлагаемый подход. Предложен способ задания функции приспособленности, рассматривающий приближение по многим направлениям.

Глава 3. Результаты

В данном разделе описываются результаты экспериментов, демонстрирующих работу предложенного метода генерации покрывающего набора тестов.

3.1. Покрытие тестами модельных задач

Исследования проводились на пяти модельных задачах, взятых с сайта инструментального средства Microsoft Pex `http://pexforfun.com`.

3.1.1. Описание модельных задач

Для каждой задачи приводится исходный код на языке $C\#$, описывается способ кодирования теста и оператор мутации. Для построения покрывающего набора тестов использовалась `(1+1)`-эволюционная стратегия.

В рассматриваемых задачах при применении оператора мутации для изменения целого числа `x` к нему добавлялось число вида $(r(19) - 9) \cdot 10^{r(10)}$, где $r(a)$ — случайное целое число в диапазоне $[0, a)$. Если `x` должен находиться в диапазоне от -10^5 до 10^5, то он заменяется на $\max(-10^5, \min(10^5, x + (r(19) - 9) \cdot 10^{r(4)}))$.

3.1.1.1. Задача 1

Код тестируемой программы приведен на листинге 3.1.1. Наиболее сложным для покрытия является случай, когда сумма элементов массива равняется 42. В качестве тестов генерируются целочисленные массивы длины шесть. Оператор мутации выбирает случайный элемент массива и изменяет его описанным ранее способом.

Листинг 3.1.1: Код задачи 1 с сайта pexforfun

```
using System;

public class Program {
  //#What values of v to cause the exception? Ask Pex to find out!#
  public static int Puzzle(int[] v) {
    int sum = 0;
    foreach (int x in v)
      sum += x;
    if (sum == 42)
      throw new Exception("hidden bug!");
    return sum;
  }
}
```

3.1.1.2. Задача 2

Код тестируемой программы приведен на листинге 3.1.2. Сложность данной задачи заключается в покрытии случая, когда элемент списка, следующий за указанным, на единицу больше. Тестом является список, состоящий из шести целых чисел, и число из множества $\{0, 1, 2, 3, 4\}$ — в качестве индекса элемента в списке. Оператор мутации случайным образом выбирает другой индекс, либо изменяет элемент списка, выбранный случайным образом.

Листинг 3.1.2: Код задачи 2 с сайта pexforfun

```
using System;
using System.Collections.Generic;

public class Program
{
  //# What values of list and i can cause exceptions? Ask Pex to find out!#
  public static void Puzzle(List<int> list, int i)
  {
    if (list[i] + 1 == list[i + 1])
      throw new Exception("hidden bug!");
  }
}
```

3.1.1.3. Задача 3

Код тестируемой программы приведен на листинге 3.1.3. Сложнее всего покрыть случай, когда сумма выбранного элемента массива и 27277 равняется 42. Тест представляется в виде массива из шести целых чисел и числа из множества $\{0, 1, 2, 3, 4, 5\}$ — индекса элемента в массиве. Опера-

тор мутации случайным образом выбирает другой индекс, либо изменяет элемент массива, выбранный случайным образом.

Листинг 3.1.3: Код задачи 3 с сайта pexforfun

```
using System;

public class Program {
  //# What values of v and i can cause an exception? Ask Pex to find out!#
  public static void Puzzle(int[] v, int i) {
    if (v[i] + 27277 == 42)
      throw new Exception("hidden bug!");
  }
}
```

3.1.1.4. Задача 4

Код тестируемой программы приведен на листинге 3.1.4. Чтобы полностью покрыть данную задачу, необходимо подобрать корень линейного уравнения. В качестве теста берется целое число в диапазоне от -10^5 до 10^5. Оператор мутации изменяет текущее решение описанным ранее образом.

Листинг 3.1.4: Код задачи 4 с сайта pexforfun

```
using System;

public class Program
{
  public static void Puzzle(int x)
  {
    // What value of x solves this equation? Ask Pex to find out!
    if (x * 3 + 27 == 153)
      Console.WriteLine("equation solved");
  }
}
```

3.1.1.5. Задача 5

Код тестируемой программы приведен на листинге 3.1.5. Сложнее всего покрыть тестом строку, в которой производится вывод на консоль. Это эквивалентно решению нелинейного уравнения второй степени в целых числах с двумя переменными и наличием ограничений. Тест представляется в виде кортежа из двух целых чисел в диапазоне от -10^5 до 10^5. Оператор мутации изменяет одно из них.

Листинг 3.1.5: Код задачи 5 с сайта pexforfun

```
using System;

public class Program {
  public static void Puzzle(int x, int y) {
    //# What values of x and y solve this equation? Ask Pex to find out!#
    if (x >= 0 && x <= 100 &&
        y >= 0 && y <= 100 &&
        x * y - 37 * y + 71 * x - 2627 == 0)
      Console.WriteLine("equation solved");
  }
}
```

3.1.2. Результаты эксперимента

Для каждой из рассмотренных задач было проведено 1000 запусков эволюционного алгоритма. В результате каждого из запусков был сгенерирован покрывающий набор тестов. В таблице 3.1 приведены минимальное, среднее и максимальное число вычислений функции приспособленности (ФП).

Таблица 3.1: Число вычислений ФП на модельных задачах

№ задачи	ФП, мин.	ФП, среднее	ФП, макс.
1	33	340	1449
2	2	660	3909
3	286	2604	13078
4	20	201	679
5	298	1429	4086

Покрытие модельных задач было осуществлено с помощью тестов, сгенерированных случайным образом. При этом, чтобы было возможным полное покрытие заданного кода, пространство поиска тестов было уменьшено. Так, например, для задачи 3.1.1.1 массив заполнялся целыми числами в диапазоне от $-5 \cdot 10^5$ до $5 \cdot 10^5$. При этом на каждом запуске генерировалось в 10 раз больше тестов, чем максимальное число вызовов ФП для соответствующей задачи. В таблице 3.2 приведены полученные результаты, усредненные по 1000 запускам.

Из результатов эксперимента можно сделать вывод, что применение эволюционного алгоритма для генерации тестов, покрывающих заданные строки кода, весьма эффективно даже для сложных условий.

Таблица 3.2: Результаты покрытия модельных задач случайными тестами

№ задачи	Покрытие, %	Число тестов на каждом запуске
1	75.3	15000
2	52.7	40000
3	55.6	130000
4	51.4	7000
5	66.7	41000

3.2. Покрытие тестами олимпиадных задач

Исследования проводились на основе задачи Huzita Axiom 6, предложенной на NEERC 2011.

3.2.1. Условие задачи

Заданы две прямые l_1 и l_2 и две точки p_1 и p_2. Необходимо найти прямую, по которой можно сложить плоскость так, что точка p_1 попадет на прямую l_1, а точка p_2 попадет на прямую l_2. Прямые задаются с помощью двух точек. Пример условия задачи приведен на рис. 3.1.

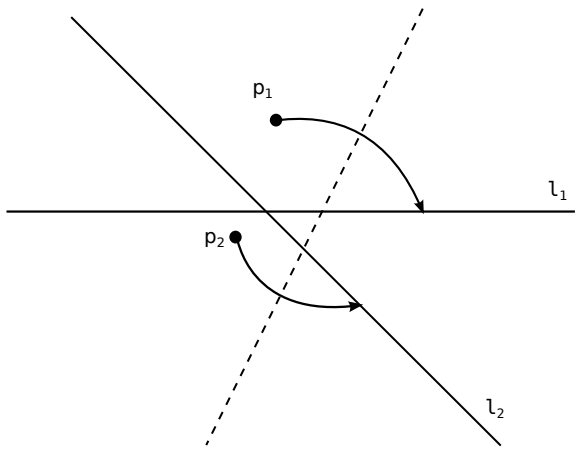

Рис. 3.1: Иллюстрация к задаче Huzita Axiom 6

3.2.2. Описание эксперимента

Для тестирования решений задачи был разработан интерфейс *Solution*, приведенный на листинге 3.2.6. Решения реализуют данный интерфейс. Входные данные(тест) передаются в виде строки, а результат выдается в виде списка строк.

Листинг 3.2.6: Интерфейс решения задачи Huzita Axiom 6

```
public interface Solution {
    public List<String> solve(String input) throws IOException;
}
```

Для построения покрывающего набора тестов использовалась (1+1)-эволюционная стратегия. Пример конфигурации ЭА, приведен на листинге 3.2.7.

Листинг 3.2.7: Конфигурация эволюционного алгоритма для задачи Huzita Axiom 6

```
class HuzitaConfig[C <: Solution](val cl : Class[C], override val maxGenerationCount : Long)
  extends TargetAwareComputationStateImpl
  with CoverageConfig[TestData, DistDiffFitness]
  with OnePlusOneES
  with TraceProvider
  with Mutation
  with FastRandomDelegate
  with CoverageFitnessComparator.LeastDistLeastAbsDiff[DistDiffFitness]
  with GenotypeGenerator
  with GenotypeWrapper
  with TotalFitnessCallCount
{
  private[this] def nestedClasses(top : Class[_]) : Seq[Class[_]] = {
    top.getDeclaredClasses.toSeq.map(x => nestedClasses(x)).flatten :+ top
  }

  override def targetMethods: Seq[Method] = nestedClasses(cl).map(_.getDeclaredMethods).flatten

  override def targetConstructors: Seq[Constructor[_]] =
      nestedClasses(cl).map(_.getDeclaredConstructors).flatten

  def newGenotype(): TestData = TestData.genTestData()

  private[this] lazy val mh = MethodHandles.lookup().findVirtual(cl, "solve",
      MethodType.methodType(classOf[java.util.List[String]], classOf[String]))

  def wrap(genotype: TestData): Seq[AnyRef] = genotype.toTestString :: Nil

  def mutate(source: TestData): TestData = source.mutate()

  def evaluateFitness(genotype: TestData): DistDiffFitness = computeFitness(wrap(genotype))

  protected def invoke(args: Seq[AnyRef]) {
    mh.invokeWithArguments(cl.newInstance() +: args : _*)
  }
}
```

Точка задается двумя целочисленными координатами. При мутации точки изменяются обе ее координаты. Линия определяется с помощью двух

точек, и при мутации изменяется каждая из них. Тест, используемый в качестве особи ЭА, задается с помощью двух прямых — l_1 и l_2 — и двух точек —p_1 и p_2. Рассматривается пять типов операторов мутации тестовых данных. Мутируют:

1. либо l_1 и p_1, либо l_2 и p_2;
2. либо l_1 и l_2, либо p_1 и p_2;
3. одна из l_1, l_2, p_1, p_2;
4. все прямые и все точки;
5. одна из l_1 и p_1 и одна из l_2 и p_2.

3.2.3. Результаты

Для каждого эксперимента было проведено по 1000 запусков. В таблицах 3.3, 3.4 и 3.5 приведены результаты работы ЭА, когда координаты точек лежат в диапазоне от -10 до 10, от -100 до 100 и от -1000 до 1000 соответственно. В первой колонке указан номер оператора мутации, во второй — среднее число вычислений ФП, а в третьей — усредненный процент покрытия кода. Можно видеть, что лучшие результаты были получены при использовании пятого оператора мутации.

Таблица 3.3: ЭА, диапазон от -10 до 10

№ мутации	ФП, среднее	Покрытие, %
1	$2,5 \cdot 10^4$	95,8
2	$2,9 \cdot 10^4$	95
3	$4,5 \cdot 10^4$	89,3
4	$2,5 \cdot 10^4$	95,9
5	$2,2 \cdot 10^4$	96,6

Таблица 3.4: ЭА, диапазон от -100 до 100

№ мутации	ФП, среднее	Покрытие, %
1	$5,7 \cdot 10^5$	87,3
2	$4,7 \cdot 10^5$	89,7
3	$5,7 \cdot 10^5$	87,3
4	$6,7 \cdot 10^5$	88,3
5	$4,2 \cdot 10^5$	91,3

В таблице 3.6 приведены результаты покрытия кода случайными тестами в сравнении с результатами ЭА. В первой колонке указан диапазон

Таблица 3.5: ЭА, диапазон от -1000 до 1000

№ мутации	ФП, среднее	Покрытие, %
1	$7,4 \cdot 10^5$	81,7
2	$6,7 \cdot 10^5$	87,3
3	$6,2 \cdot 10^5$	88
4	$7,9 \cdot 10^5$	79,3
5	$5,9 \cdot 10^5$	90

Таблица 3.6: Покрытие решения Huzita Axiom 6 случайными тестами

Диапазон	Число тестов	Покрытие, %	Покрытие ЭА, %
-10 до 10	$5 \cdot 10^5$	97,7	96,6
-100 до 100	10^6	80,3	91,3
-1000 до 1000	10^6	74,3	90

координат, во второй — число случайно генерируемых тестов, в третьей — усредненный процент покрытия кода случайными тестами, а в четвертой — лучший усредненный процент покрытия кода с помощью ЭА.

Из результатов эксперимента можно сделать вывод, что время и качество работы ЭА сильно зависит от выбора оператора мутации. При увеличении множества допустимых тестов, ЭА работает стабильнее и достигает лучших результатов, чем случайная генерация тестов.

3.3. Выводы по главе 3

Были проведены эксперименты, демонстрирующие работу предложенного подхода для автоматизированного покрытия кода тестами на основе эволюционных алгоритмов. В результате экспериментов, проведенных на модельных задачах, было получено, что разработанный метод эффективен для покрытия даже сложных условий. Также применимость разработанного подхода была протестирована на олимпиадной задаче Huzita Axiom 6. Было получено, что эффективность данного метода сильно зависит от выбора эволюционных операторов. Можно видеть, что при увеличении множества допустимых тестов результаты предложенного подхода превосходят результаты, полученные при покрытии кода случайными тестами.

Заключение

В данной работе создана платформа для автоматизированного покрытия кода тестами на основе эволюционных алгоритмов. Разработан новый способ построения функции приспособленности: используется не опорная траектория, а рассматривается приближение к цели по многим направлениям.

Предложенный метод был успешно применен для покрытия тестами ряда модельных задач. Было проведено сравнение разработанного подхода с методом генерации случайных тестов. В результате экспериментов было получено, что в отличие от метода генерации случайных тестов, предложенный подход обеспечивает полное покрытие кода.

Кроме того, разработанный метод был применен для покрытия тестами решения олимпиадной задачи Huzita Axiom 6. В ходе экспериментов было установлено, что результат работы предложенного подхода зависит от выбора эволюционных операторов. Предложенный метод показал хорошие результаты даже для больших диапазонов допустимых тестов, в то время как метод генерации случайных тестов применим лишь для малых диапазонов.

Таким образом, данная работа полностью удовлетворяет поставленным требованиям.

Список литературы

1. Structural coverage metrics. http://www.ipl.com/pdf/p0823.pdf.
2. *Weiser M. D.*, *Gannon J. D.*, *McMullin P. R.* Comparison of Structural Test Coverage Metrics // IEEE Softw. 1985. №2. С. 80—85. ISSN: 0740-7459.
3. *Скобцов Ю. А.* Основы эволюционных вычислений. Донецк: ДонНТУ, 2008.
4. *Harman M.*, *Mansouri S. A.*, *Zhang Y.* Search-based software engineering: Trends, techniques and applications // ACM Comput. Surv. 2012. №1. 11:1—11:61. ISSN: 0360-0300.
5. *McMinn P.* Search-based Software Test Data Generation: A Survey // Software testing, verification and reliability. 2004. С. 105—156.
6. *Harman M.*, *McMinn P.* A Theoretical and Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation / International Symposium on Software Testing and Analysis (ISSTA 2007). London, UK: ACM, 2007. С. 73—83.
7. *Kirkpatrick S.*, *Gelatt C. D.*, *Vecchi M. P.* Optimization by Simulated Annealing // Science. 1983. New Series №4598. С. 671—680. ISSN: 00368075.
8. *Tonella P.* Evolutionary testing of classes / Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004. С. 119—128. ISBN: 1-58113-820-2.
9. *Fraser G.*, *Arcuri A.* Whole Test Suite Generation. Тех. отч. Madrid, Spain, 2011.
10. *Tracey N.*, *Clark J.*, *Mander K.*, *Mcdermid J.*, *York H.* An automated framework for structural test-data generation / Proceedings of the International Conference on Automated Software Engineering; IEEE. 1998.
11. *Baars A.*, *Harman M.*, *Hassoun Y.*, *Lakhotia K.*, *McMinn P.*, *Tonella P.*, *Vos T.* Symbolic search-based testing / Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011. С. 53—62. ISBN: 978-1-4577-1638-6.
12. *Godefroid P.*, *Halleux P.*, *Nori A. V.*, *Rajamani S. K.*, *Schulte W.*, *Tillmann N.*, *Levin M. Y.* Automating Software Testing Using Program Analysis // IEEE Softw. 2008. №5. С. 30—37. ISSN: 0740-7459.
13. *Xu R.-G.* Symbolic execution algorithms for test generation. AAI3410346. Дис. ... док. Los Angeles, CA, USA, 2009. ISBN: 978-1-124-00942-1.
14. *Lindholm T.*, *Yellin F.*, *Bracha G.*, *Buckley A.* The Java Virtual Machine Specification, Java SE 7 Edition. Pearson Education, 2013. ISBN: 9780133260465.
15. Scala language. http://www.scala-lang.org/.
16. ASM library. http://asm.ow2.org/.
17. ThreadLocal. http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html.
18. *Karp R. M.* Reducibility Among Combinatorial Problems. В: 50 Years of Integer Programming 1958-2008. Под ред. Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi и Laurence A. Wolsey. Springer Berlin Heidelberg, 2010. С. 219—241. ISBN: 978-3-540-68279-0.
19. *Raz R.*, *Safra S.* A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP / Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. STOC '97. El Paso, Texas, USA: ACM, 1997. С. 475—484. ISBN: 0-89791-888-6.