САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И
ПРОГРАММИРОВАНИЯ
КАФЕДРА «КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ»

ИГОРЬ АЛЕКСАНДРОВИЧ СИНЕВ

ПРИМЕНЕНИЕ ДВУХФАЗНОГО МЕТОДА ОПТИМИЗАЦИИ ДЛЯ ГЕНЕРАЦИИ ТЕСТОВ ДЛЯ ОЛИМПИАДНЫХ ЗАДАЧ ПО ТЕОРИИ ГРАФОВ

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

НАУЧНЫЙ РУКОВОДИТЕЛЬ – ДОКТ. ТЕХН. НАУК, ПРОФЕССОР
АНАТОЛИЙ АБРАМОВИЧ ШАЛЫТО

САНКТ-ПЕТЕРБУРГ
2010

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ 5
1.1. Тестирование ПО 5
1.1.1. Традиционные подходы к тестированию ПО 5
1.1.2. Эволюционные подходы к тестированию 6
1.1.3. Генерация случайных тестов, направляемая обратной связью 6
1.1.4. Использование алгоритмов поиска для построения тестов в применении к тестированию объектно-ориентированных контейнеров 6
1.1.5. Автоматизированное fuzz-тестирование белого ящика 8
1.2. Олимпиадные задачи 8
1.2.1. Об истории олимпиад 8
1.2.2. Олимпиадные задачи 9
1.2.3. Качество тестов к олимпиадным задачам 10
1.2.4. Задачи о графах 11
1.3. Изучение свойств моделей графов 12
1.4. Особенности тестирования олимпиадных задач 13
ГЛАВА 2. ПОСТАНОВКА ЗАДАЧИ И ОПИСАНИЕ ПРЕДЛАГАЕМОГО ПОДХОДА 16
2.1. Актуальность 16
2.2. Постановка задачи 17
2.3. Выбор критериев оптимизации 18
2.4. Генерирующие последовательности для графов 20
2.4.1. Описание базовых типов операций 21

2.4.2. Пример обработки последовательности 22
2.4.3. Применение генерирующих последовательностей для создания тестов к олимпиадным задачам 27
2.4.4. Обработка атрибутов 28
2.5. Решение задачи оптимизации на пространстве генерирующих последовательностей 29
2.5.1. Выбор алгоритма оптимизации 29
2.5.2. Описание метода 30
2.6. Описание построенной системы 32
ГЛАВА 3. ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ 34
3.1. Общие параметры генерации тестов 34
3.2. Задача tree 35
3.2.1. Условие задачи и решения 35
3.2.2. Построенные вручную тесты 35
3.2.3. Исследование статистики по тестам небольшого размера 40
3.3. Задача robot 41
3.3.1. Задание цветов ребер 42
3.3.2. Результаты тестирования 42
3.4. Задача Clique 43
3.4.1. Используемые элементарные операции 44
3.4.2. Результаты тестирования 44
ЗАКЛЮЧЕНИЕ 46
ИСТОЧНИКИ 47

ВВЕДЕНИЕ

Тестирование – неотъемлемая часть процесса разработки программного обеспечения, на которую по некоторым оценкам затрачивается более половины времени работы над проектами и стоимости этой работы. Для того, чтобы сократить эти расходы, применяются различные методы автоматизированного тестирования. Первые работы по автоматизированной генерации тестов начали появляться в 90-х годах. В последнее время эта область и, как ее часть, методы эволюционного тестирования динамично развиваются.

Соревнования по программированию (часто называемые олимпиадами) имеют уже достаточно долгую историю. Они способствуют выявлению талантливых программистов среди школьников и студентов, спонсорами соревнований выступают многие компании, работающие в сфере информационных технологий. На подготовку и проведение олимпиад уходит немало времени и сил организаторов, существенную часть которых занимает составление тестовых данных к задачам.

В настоящей работе изучается возможность адаптации идей, лежащих в основе современных методов автоматизированной генерации тестов применительно одному из классов олимпиадных задач – задачам по теории графов.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Тестирование ПО

Цель тестирования программного обеспечения – удостовериться в том, что программа работает именно так, как было задумано, и что она не делает того, что ей не следует делать [1]. В общем случае эта задача, даже в строго формализованном виде, алгоритмически неразрешима согласно теореме Райса. Поэтому тестирование ПО до сих пор в значительной степени базируется на эвристическом подходе.

В книге [1], ставшей классикой, упомянуто, что тестирование занимает около 50% времени и более 50% стоимости разработки программного обеспечения. Качество тестирования сильно подвержено влиянию человеческого фактора. По этой причине многие исследователи и разработчики прикладывают значительные усилия для автоматизации тестирования.

1.1.1. Традиционные подходы к тестированию ПО

Набор традиционных способов тестирования включает в себя [1] такие виды, как:

- чтение кода;
- тестирование «черного ящика»;
- тестирование «белого ящика»;
- модульное тестирование;
- функциональное тестирование;
- отладка.

Некоторые из этих видов тестирования, такие как чтение кода или отладка, требуют прямого участия человека в процессе, в то время как модульное

тестирование, при должной организации процесса производства, может быть полностью автоматизировано.

1.1.2. Эволюционные подходы к тестированию

Если качество теста или набора тестов удается выразить каким-либо количественным критерием, то становится возможным использование оптимизационных алгоритмов для поиска тестов, отвечающих заданным требованиям. Тесты при данном подходе выражены в виде программы, которая подготавливает систему к работе с некоторым тестовым случаем, запускает тестируемый метод класса или тестируемую процедуру и проверяет результаты на соответствие спецификации, что позволяет автоматизировать процесс тестирования.

1.1.3. Генерация случайных тестов, направляемая обратной связью

Суть данного метода состоит в инкрементальном построении тестовых сценариев, представленных в виде последовательностей вызовов простых функций, путем повторения генерации и запуска с отсеиванием избыточных или противоречащих контракту последовательностей. Авторы статьи [2] анализируют результаты, полученные при запуске созданного на базе этого метода инструмента *Randoop*, для нескольких объектных контейнеров на *Java*.

1.1.4. Использование алгоритмов поиска для построения тестов в применении к тестированию объектно-ориентированных контейнеров

В двух рассмотренных работах *Andrea Arcuri* [3, 4] производится сравнение нескольких методов поиска для автоматизированного построения тестов применительно к классу, реализующему красно-черное дерево (*RBTree)*.

Работа класса и покрытие веток исполнения в нем изучены на существующих тестах, и в качестве задачи, которую должны решать методы поиска, ставится достижение покрытия того блока кода в рассматриваемом классе, который вызывается реже всего (он является частью закрытого метода).

Набор алгоритмов, для которых производится сравнение, включает в себя:

1. Случайный поиск (Random Search).
2. Метод «подъема в гору» (Hill Climbing).
3. Простой эволюционный алгоритм ((1+1) Evolution Algorithm).
4. Генетический алгоритм (Genetic Algorithm).

Отмечено, что при наличии нескольких критериев оптимизации, решать, какой из них важнее, должен человек, это можно сделать, например, задав одну функцию от оптимизируемых величин, значение которой будет использоваться генератором для оценки теста.

Авторы указанных работ утверждают, что следует искать возможность сделать параметры оптимизации не дискретными, а непрерывными, или, как минимум, увеличить их допустимый диапазон значений, в результате чего поверхность оптимизации станет более гладкой. В рассматриваемых работах этот подход реализован путем перехода от подсчета числа покрытых веток исполнения, к оценке, насколько близко были пройдены те или иные ветки (branch distance).

Для анализа потока исполнения в тестируемый код вставляются дополнительные операторы инструментации (instrumentation).

С целью выявления не только эффективных, но и обладающих малым размером тестов, предложена функция оптимизации для теста, которая учитывает как полноту покрытия, так и размер теста:

$$f(S_i) = \operatorname{cov}(S_i) + \frac{1}{1 + len(S_i)} .$$

1.1.5. Автоматизированное fuzz-тестирование белого ящика

Некоторые ошибки в ветках исполнения встречаются редко, и их трудно обнаружить, например:

```
if (c[0] == 'b' && c[1] == 'a' && c[2] == 'd') {
    // rarely executed block
}
```

В статьях [5, 6] для проверки таких случаев предлагается использовать символьное вычисление (symbolic execution). Построение тестов для покрытия новых веток достигается с помощью решения систем ограничений на входные данные.

Отметим, что такой метод может использоваться и для олимпиадных задач, предоставляя возможности для построения тестов на граничные случаи (corner cases) и ошибки, связанные с неверными вычислениями индексов массивов или присвоением переменных. На момент начала настоящей работы эта тема не была широко освещена в статьях, и даже на момент ее завершения не удается найти открытых систем для символического исполнения, а создание такой системы с нуля выходило бы за рамки текущей работы.

Поэтому в настоящей работе не ставится цель превзойти методы, основанные на символьном вычислении в той области, где они могут успешно применяться, вместо этого круг исследуемых задач сужается в направлении генерации тестов на производительность, отсекающих как детерминированные, так и рандомизированные неверные решения.

1.2. Олимпиадные задачи

1.2.1. Об истории олимпиад

В начало списка наиболее известных ежегодных соревнований по программированию можно поставить студенческий чемпиона мира по

программированию ACM ICPC (Association for Computing Machinery International Collegiate Programming Contest), который [7] проводится с 1977 года и Международную олимпиаду школьников по информатике IOI (International Olympiad in Informatics), проводящуюся с 1989 года. Для того, чтобы попасть в финальный этап этих состязаний, участники должны достойно выступить на соревнованиях районного (внутриуниверситетского), городского и регионального уровня, но благодаря этому многие школьники и студенты могут попробовать свои силы в решении олимпиадных задач. Среди прочих можно отметить соревнования, проводимые компаниями TopCoder [8] и Google [9], интернет-олимпиады по информатике и программированию [10].

Наряду с олимпиадами по программированию проводятся также конкурсы по многим другим околокомпьютерным дисциплинам (например, [11]), но в настоящей работе речь будет идти именно об олимпиадах, где участникам за ограниченное время предлагается решить несколько задач, требующих как применения познаний в области алгоритмов, так и навыков по написанию сложных программ [7].

1.2.2. Олимпиадные задачи

Олимпиадные задачи могут быть очень разными. Иногда для решения задачи требуется вывести формулу, в других случаях необходимо запрограммировать аналог функционирующего компонента промышленной системы.

Для школьников, начинающих изучать программирование, соревнования могут проводиться и в формате теоретического тура – решая задачи, участники описывают алгоритм решения словами и записывая программный код на бумаге, после чего их работы оцениваются жюри.

Однако в большинстве случаев решения задач принимаются в виде исполняемых программ, при этом оценка решений производится посредством их

запуска на заранее подготовленных наборах тестовых данных. Необходимость использования такого подхода обусловлена тем, что в общем случае верификация программы (проверка соответствия того, что делает программа, ее спецификации; в случае решения задачи – того, что программа действительно решает поставленную задачу) невозможна.

Хотя, по сравнению с другими этапами подготовки задач (поиском идеи задачи, написанием решений), подготовку тестов может назвать относительно формализованной, она остается достаточно трудоемким творческим процессом.

Подробно о различных формах тестирования и о методе составления тестов рассказывается в актуальной работе М. Буздалова [12], и приводить здесь этот обзор в полном виде было бы излишним. Поскольку в настоящей работе решается задача оптимизации тестов, необходимо отдельно остановиться на оценке их качества.

1.2.3. Качество тестов к олимпиадным задачам

Основная цель составления набора тестов к любой задаче заключается в выявлении неверных и неэффективных решений. При этом для большинства задач проверка на всех возможных наборах входных данных невозможна. Следовательно, возникает необходимость выбрать такое подмножество допустимых тестов, чтобы как можно большее число неверных или неэффективных решений не прошло такие тесты (на разных соревнованиях минимальное требуемое число непройденных тестов может быть различным: от одного до половины всех тестов).

Тесты могут иметь разное назначение, например:

- выявление неверных решений (выдающих неверный ответ);
- выявление неэффективных решений (работающих за время, превышающее ограничения);

- выявление ошибок реализации решения (с различными результатами, включающими ошибки времени выполнения);
- выявление отдельно неверно разобранных случаев.

Критерии, по которым набор тестов к задаче можно назвать качественным, могут меняться от задачи к задаче, но в общем можно сказать, что тесты должны обладать следующими свойствами:

- они могут с достаточной достоверностью гарантировать отсев неверных решений (бывает, что неверные решения могут успешно проходить все тесты из набора);
- удовлетворять ограничениями на объем тестовых данных (чем больше тестов в наборе и чем больше их суммарный размер, тем больше ресурсов – времени и вычислительной мощности - уходит на проверку всех решений).

1.2.4. Задачи о графах

В настоящей работе рассматривается составление тестов для класса задач, основанных на теории графов. Теория графов — раздел дискретной математики, изучающий свойства графов [12]. Граф – это множество вершин (узлов), соединенных ребрами. Формально граф определяется как пара множеств $G = (V, E)$, где V есть подмножество любого счетного множества, а E — подмножество $V \times V$.

Выбор для рассмотрения именно этого класса задач обусловлен тем, что:

- входные данные даже для задач разного уровня и с принципиально разными решениями чаще всего достаточно мало отличаются от традиционного стандарта:
    - в условии определяется тип графа – граф общего вида или один из специальных:
        - дерево/лес (ациклические графы);

        - транспортная сеть (заданы начальный и конечный узлы, между которыми прокладываются пути);
        - двудольный граф;
    - указывается число вершин графа и перечисляются его ребра (для ориентированных графов – дуги); в некоторых случаях задается также функция на ребрах, которая может по смыслу соответствовать цвету или весу ребер (в общем случае, с каждым ребром ассоциируется некоторая дополнительная информация);
- таких задач достаточно много (исходя из рассмотрения задач архивов ACM ICPC NEERC [14] и интернет-олимпиад [10], а так же статистики TopCoder [8], можно оценить их долю как 15%).

1.3. Изучение свойств моделей графов

В то время как полностью и формально решить вопрос генерации тестовых графов на основе описанных в рассмотренной литературе методов нельзя, многие из описанных идей могут быть применены в исследуемой области.

В работах [15, 16] производится изучение свойств моделей для описания эволюционирующих графов, таких как граф ссылок между страницами в сети Интернет. Отмечаются несколько структурных свойств, в частности, модель «тематических групп» (hubs and referents). Эти свойства учитываются в предложенной модели добавления и удаления вершин, что позволяет отразить такие характеристики сети, как распределение вершин и вероятности образования кластеров различного размера. Целью исследования является улучшение алгоритмов, используемых в поисковых машинах.

В статье [17] рассматривается модель случайного графа с заданным распределением степеней вершин, изучаются свойства графов, построенных по

этой модели, например, максимальный и ожидаемый размеры компоненты связности или радиус графа относительно случайной вершины.

В работе [18] приводится метод построения по модели сетей, которые используются для оценки производительности алгоритмов, оптимизированных для решения прикладных задач (benchmarks). Предлагается генерировать так называемые *scale-free graphs* (графы с самоподобной структурой) со взвешенными ребрами (веса выбираются из набора *{0, 1}*). Для оценки разных типов алгоритмов создаются разные образцы, например, в один тип объединены алгоритмы поиска пути (алгоритм обхода графа в ширину, алгоритм Дейкстры), поскольку они работают с наборами вершин, операции с которыми выполняются посредством последовательных обращений.

1.4. Особенности тестирования олимпиадных задач

Если сравнивать генерацию тестов к олимпиадным задачам по теории графов с такой сравнительно изученной областью, как тестирование промышленного программного обеспечения, можно отметить несколько важных отличий.

Тестирование в промышленных системах характеризуется тем, что:

1. Обычно тестируется уже созданная система.
2. Программы, решающие прикладные задачи, обычно разбиты на отдельные модули и функции с определенным назначением, которые можно тестировать в отдельности (для этого создаются модульные тесты).
3. Наиболее изученной областью является генерация тестов, позволяющих выделить ошибки.
4. При создании тестов на производительность промышленного использования [18] оценивается скорость работы на специально

подобранных данных, моделирующих ожидаемые данные, которые программа будет обрабатывать во время эксплуатации.

При этом для олимпиадных задач по теории графов справедливо, что:

1. Тесты составляются до того, как проверяющая система получает решения участников. Отчасти это компенсируется тем, что при генерации тестов чаще всего подготавливаются типичные неправильные решения.
2. Решение олимпиадной задачи может состоять из одной процедуры, и тестирование его отдельных логических частей может быть невозможно.
3. Набор тестов для олимпиадной задачи должен отсеивать не только принципиально неверные, но и неэффективные решения.
4. Тесты к олимпиадным задачам, должны удовлетворять двум ограничениям. С одной стороны, заданные в условии задачи ограничения на максимальный объем входных данных должны позволять участнику соревнования, придумавшему верное решение и правильно оценившему время его работы в худшем случае, убедиться, что это решение будет укладываться и в заданные ограничения по времени исполнения. С другой стороны, даже оптимизированные неправильные решения не должны укладываться в указанные ограничения на достаточно большой доле тестов, из-за этого при фиксированных ограничениях на размер необходимо подбирать такие тесты, для которых время работы неправильных решений будет максимальным.

В заключение главы отметим, что автоматизация генерации тестов к олимпиадным задачам является актуальной задачей.

Методы, разработанные для тестирования промышленных систем, не представляется возможным напрямую применить для генерации тестов к

олимпиадным задачам, но многие идеи, лежащие в их основе, могут быть адаптированы для применения в области настоящей работы.

ГЛАВА 2. ПОСТАНОВКА ЗАДАЧИ И ОПИСАНИЕ ПРЕДЛАГАЕМОГО ПОДХОДА

2.1. Актуальность

Изучив существующие генераторы тестов для задач о графах можно отметить следующие их особенности, которые позволяют предположить, что в этой области есть возможность для автоматизации:

1. В большом числе генераторов используется инструкция начальной инициализации генератора случайных чисел. Значение параметра для такой инструкции часто подбирается вручную с целью выбора лучшего из нескольких похожих тестов.
2. Многие генераторы содержат подпрограммы, выполняющие одинаковые или очень похожие операции. При этом на данный момент выделение таких повторяющихся операций в библиотеки используется далеко не всегда.

В разд. 1.2 настоящей работы было отмечено, что создание тестов к олимпиадным задачам занимает существенную часть времени, затрачиваемого на подготовку олимпиад и, несмотря на большой накопленный в этой области опыт, на настоящий момент выполняется чаще всего эвристическими методами.

Как показано в упомянутых работах о моделях случайных графов [17], для таких графов известно, многие их параметры, которые могут определять пригодность графа как теста для задачи (например, диаметра или средней степени вершины), в среднем попадают лишь в небольшой поддиапазон допустимых значений (экспериментальные результаты, говорящие в пользу этого рассуждения, приведены в разд. 3.2.3). Поэтому генерация больших тестов в форме случайных графов как таковых не всегда перспективна.

Все вышесказанное позволяет заключить, что:

1. Создание системы, позволяющей сократить время на создание тестов для задач по теории графов, является актуальной задачей.
2. В то время как задача о генерации тестов к произвольным задачам является трудноразрешимой, благодаря особенностям задач по теории графов, разработка специального метода для этой области кажется перспективной.

Помимо генерации тестов к олимпиадным задачам, такой метод может, будучи развитым, найти применение в области составления тестов на производительность для промышленных систем, работающих с графами.

2.2. Постановка задачи

Как было показано в предыдущем разделе, актуальной является задача разработки метода, который бы позволил сократить время, которое затрачивается на генерацию задач по теории графов.

В близкой по тематике работе М. Буздалова [12] был разработан успешно примененный метод генерации тестов к задаче о рюкзаке. В то время как приведенные там результаты имеют как методологическую, так и практическую ценность, а многие идеи использованы и в настоящей работе, способов обобщения предложенного подхода на другие задачи на момент окончания той работы предложено не было. При этом изучение возможности применения таких методов для составления тестов к другим олимпиадным задачам, вероятнее всего, не может быть осуществлено в рамках работы по подготовки олимпиады из-за ограничений по времени, поэтому перед настоящей работой ставится задача выявить возможности для генерации тестов без проведения детального анализа задач.

Следовательно, необходимо предложить метод, базирующихся на механизмах, работающих для достаточно общего случая, и показать, как его можно применить на конкретных примерах.

Одной из сложных задач для генерации тестов с помощью методов поиска является построение тестов на крайние случаи, которые могут быть маловероятно достижимы как для последовательной оптимизации, так и для перебора. Как отмечалось выше, наиболее перспективными для решения этих задач кажутся методы типа динамического fuzz-тестирования (fuzz testing) с использованием символьного вычисления [5]. Их применение так же позволило бы эффективно создавать тесты для проверки программ на ошибки типа *+/-1*. Однако изучение применения таких методов выходит за рамки настоящей работы.

Во многих случаях тест к задаче, помимо графа, может содержать и некоторые дополнительные параметры задачи. Для того, чтобы к этим задачам можно было применять универсальный метод, можно задавать такие параметры в качестве начальных данных для генератора, или вычислять значения параметров в процессе построения теста, или генерировать их по построенному тесту..

Так как в процессе подготовки задачи иногда производится написание и тестирование разных решений и, в некоторых случаях, модификация установленных в первом приближении ограничений, желательно, чтобы после запуска программы генерации тестов она могла выдавать удовлетворительные результаты за минуты и хорошие результаты за часы. Из-за этого применение более сложных методов оптимизации, таких как генетическое программирование, может не всегда быть оправданным.

2.3. Выбор критериев оптимизации

Перед тем, как описывать работу генератора тестов, необходимо установить критерии, по которым оценивается качество теста.

Как было отмечено в [3], оптимизацию следует проводить по одному критерию. В случае необходимости оценки нескольких параметров теста, следует задать для генератора целевую функцию, зависящую от этих критериев тем или

иным образом и, в зависимости от выбора этой функции, получать на выходе тесты.

В соответствии с разд. 1.2.3, наиболее подходящими критериями оценки теста в общем случае можно назвать время работы решения и объем используемой памяти. В некоторых случаях время работы можно более точно оценить с помощью инструментации, например, через число выполняемых операций или выполненных перезапусков рандомизированного алгоритма. Для тех задач, ответ на которые может содержать большие числа или должен быть посчитан с высокой точностью, целесообразным может быть подбор тестов, на которых экстремального значения достигает некоторая функция от ответа. Другим возможным критерием общего типа могло бы быть количество отсеиваемых мутантов (решений, получаемых из правильного путем небольшого случайно сгенерированного изменения), но в настоящей работе этот подход не рассматривается.

Можно отметить, что подсчет оценочной функции может производиться как посредством запуска решения на тесте, так и прямым вычислением ее значения для тестового графа, если это возможно.

Базовые характеристики, такие как время работы и объем использованной памяти, могут быть получены и в общем виде. Для вычисления специальных функций требуется написание отдельных подпрограмм.

В обоих случаях критерии оценки качества и ограничения на размер теста являются входными данными для генератора.

Общая схема работы генератора тестов на производительность такова:

Шаг 1. Выбирается или случайным образом строится исходный экземпляр теста (или, в случае использования генетического алгоритма, популяция тестов).

Шаг 2. Для теста или тестов из текущего набора вычисляется значение функции, подлежащей оптимизации (в случае ГА – функция приспособленности).

Шаг 3. Если достигнут установленный предел по времени или количеству шагов генерации, алгоритм возвращает оптимальный из ранее найденных тестов и завершает работу.

Шаг 4. Для случайного поиска выполняется переход на шаг 1, остальные алгоритмы модифицируют или заменяют на новые некоторые тесты (или единственный тест) набора и выполняют переход на шаг 2.

В настоящей работе не ставится задача превзойти генерацию тестов вручную: во многих случаях, проанализировав неэффективное решение задачи, можно составить тест, для которого оптимальность по заданным критериям может быть доказана математически. Однако такой анализ необходимо выполнять отдельно для каждой задачи, поэтому можно надеяться, что метод, который позволил бы существенно уменьшить усилия по генерации достаточно хороших тестов для широкого класса задач, найдет применение.

2.4. Генерирующие последовательности для графов

Тестовые графы могут содержать тысячи вершин и десятки тысяч ребер. В этом случае определить способ работы алгоритмов поиска, работающих напрямую с графами, затруднительно.

Проанализировав существующие генераторы, можно видеть, что генерируемые тестовые графы имеют регулярную структуру и часто состоят из последовательного вызова нескольких процедур (например, создания нескольких частей графа и добавления ребер между ними).

Кроме того, для пояснения предложенного способа, заметим, что работу многих алгоритмов на графах можно приближенно описать последовательностью следующих операций:

- выбор начальной вершины;
- выполнение необходимых вычислений;

- переход к следующей вершине, возврат к предыдущей вершине, или выбор новой начальной вершины.

Предположим, что решение получает входные данные не из файла, а через запросы к проверяющей системе (например, запрос списка номеров вершин, инцидентных заданной), и проверяющая система может моделировать его работу. Тогда при неограниченных ресурсах проверяющая система могла бы для ответа на запрос решения перебирать все возможные варианты ответа на него, не противоречащие ответам на предыдущие запросы, и выдавать тот из них, при котором время работы решения будет максимально.

Трудоемкость такого способа сопоставима с полным перебором всех возможных наборов входных данных, однако приведенные выше рассуждения показывают, что «хороший» тест может быть построен из меньшего «хорошего» теста операциями, подобными добавлению соседей к выбираемым некоторым образом вершинам графа.

Поэтому предлагается генерировать не сами графы, а последовательности элементарных операций, результатом работы которых являются графы. Эти элементарные операции должны быть, с одной стороны, достаточно эффективными, с другой стороны, переиспользуемыми.

2.4.1. Описание базовых типов операций

В качестве базовых типов операций, из которых составляются генерирующие последовательности, предлагается использовать следующие четыре:

1. `S`[*elect*] – *выбор вершины*. Способом, определенным для конкретной операции, выбирается для передачи последующим операциям одна из вершин графа.
2. `T`[*ransform*] – *элементарное преобразование*. К последней выбранной вершине применяется указанное преобразование. Это преобразование может

затрагивать не только указанную вершину, но должно быть локальным относительно нее. Последовательности, для которых какой-либо из операций `T` не предшествует никакая `S`, являются некорректными.

3. `L`[*oop*] – *последовательное повторение*. У операции два параметра, количество повторений и повторяемая подпоследовательность, начинающаяся не с операции преобразования `T`. Запоминается выбранная первой операцией вершина, после чего заданное количество раз повторяется заданная подпоследовательность.
4. `R`[*ecursion*] – *рекурсивное повторение*. Параметры этой операции такие же, как и у предыдущей. Отличие от типа `L` состоит в том, что выбор первой вершины не запоминается, а производится каждый раз заново.

В случае, если число повторений может быть определено зафиксированным при построении последовательности числом, множество допустимых значений для него можно ограничить значениями по логарифмической шкале. Это существенно сокращает размер пространства поиска, не ограничивая возможности для теста обнаруживать асимптотические свойства алгоритма. Если требуется, количество повторений может определяться и динамически (разд. 2.4.4).

Последовательность таких операций удобно представить в виде дерева: листьями будут пары ‹`S`, `T`›, промежуточными вершинами – операции `L` и `R`, служебная вершина-корень объединяет вершины первого уровня. О перспективности использования древовидного представления программ можно судить так же по результатам, опубликованными в работе *J. R. Koza* [19], где эта идея положена в основу метода генетического программирования.

2.4.2. Пример обработки последовательности

Для пояснения работы предлагаемого метода рассмотрим пример последовательности и опишем для этого примера процесс исполнения. Определим по одной простой операции для типов `S` и `T`. Для типа `S` это будет `selectLast`,

выбирающая последнюю созданную вершину. Для типа `T` это будет `connectToNew`, добавляющая новую вершину и соединяющую ее с выбранной соответствующей операцией `S`. Подробнее об этих и других типах операций будет рассказываться при рассмотрении конкретных задач в главе 3.

На рис. 1 изображен пример генерирующей последовательности, для которого будет дано пояснение работы алгоритма.

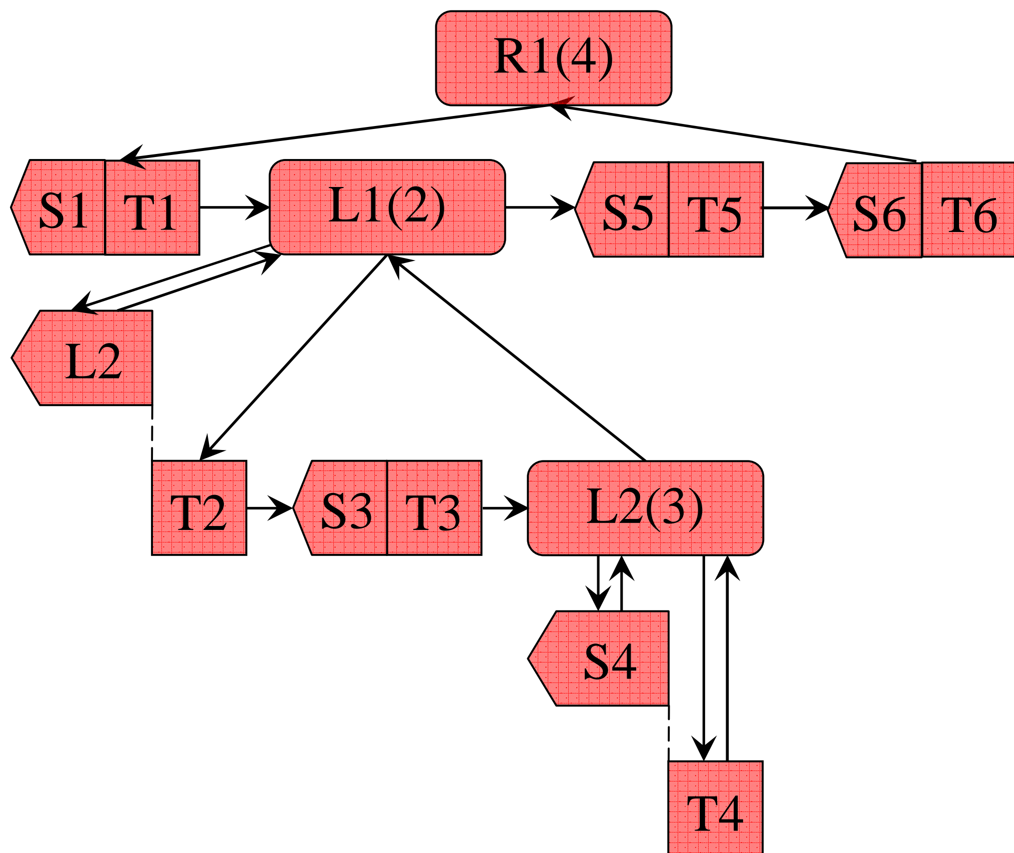

Рис. 1. Пример последовательности

Исполнение данной последовательности начинается с верхней операции, и в процессе исполнения производится переход по стрелкам. При исполнении операции `R(`n`)` операции из поддерева выполняются n раз, после чего управление

передается дальше. Это же верно и для операции `L(n)` – за исключением того, что первая операция `S` выполняется только один раз.

Граф, генерируемый этой последовательностью, содержит 27 вершин, поэтому процесс генерации будет демонстрироваться по этапам.

Проследим первый этап исполнения последовательности, заканчивающийся после первого исполнения цикла `L2` (эта часть последовательности изображена на рис. 2):

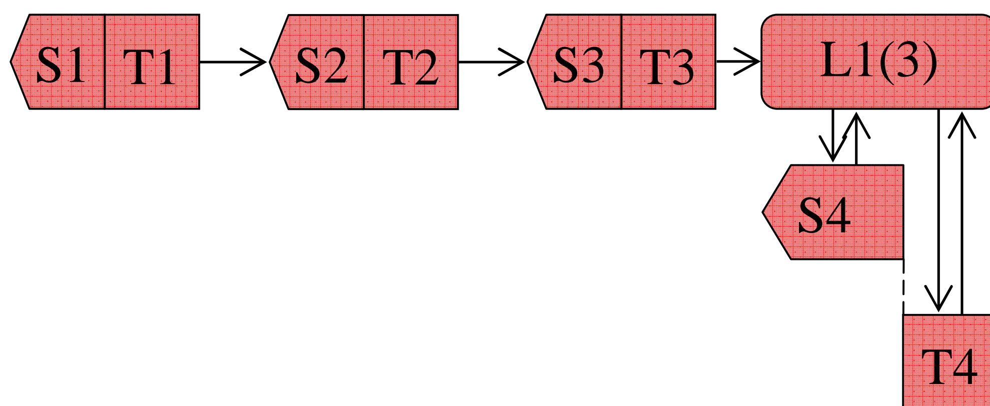

Рис. 2. Процесс исполнения последовательности (этап 1)

- подпоследовательность ‹`S1`,`T1`› к выбранной единственной вершине добавляет ещё одну;
- ко второй вершине подпоследовательность ‹`S2`,`T2`› добавляет третью вершину;
- четвёртая вершина добавляется подпоследовательностью ‹`S3`,`T3`› при выполнении цикла `L1(3)`, состоящего только из одной подпоследовательности ‹`L1`,`T4`›, с помощью операции `L1` три раза выбирается одна и та же вершина 4, к которой с помощью операции `T4` добавляются вершины 5, 6 и 7 (Рис. 4.а).

К концу второго этапа исполнения последовательности (рис. 3) будет один

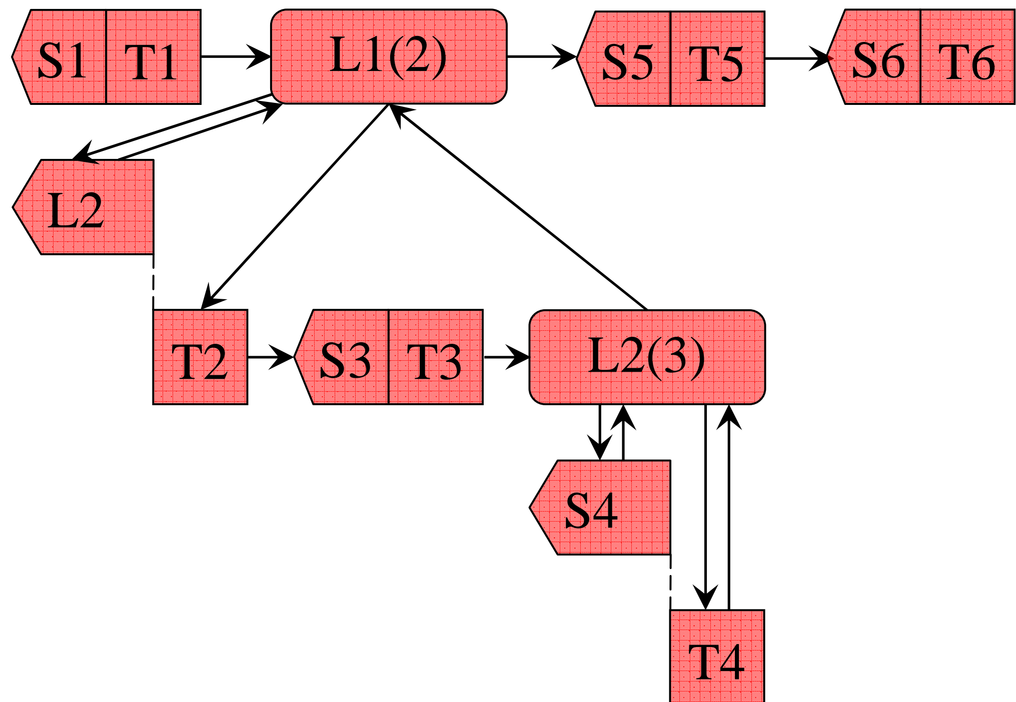

Рис. 3. Второй этап исполнения последовательности из примера

раз исполнена последовательность, образующая тело операции рекурсии `R1` имеются два цикла: `L1(2)` повторяющийся два раза, и вложенный в него цикл `L2(3)`, который повторяется три раза внутри цикла `L1`. В результате выполнения данной последовательности:

- сначала добавляется одна вершина;
- затем эта вершина выбирается в качестве базовой для цикла `L1(2)`, внутри которого дважды выполняется последовательность аналогичная рассмотренной выше;
- в завершении данной последовательности к последней из построенных вершин графа с помощью подпоследовательностей ‹`S5,T5`› – ‹`S6,T6`› достраиваются ещё две вершины (Рис. 4.б).

На третьем этапе цикл `R1` выполняется второй раз, завершается выполнение последовательности, результат можно видеть на (Рис. 4.в).

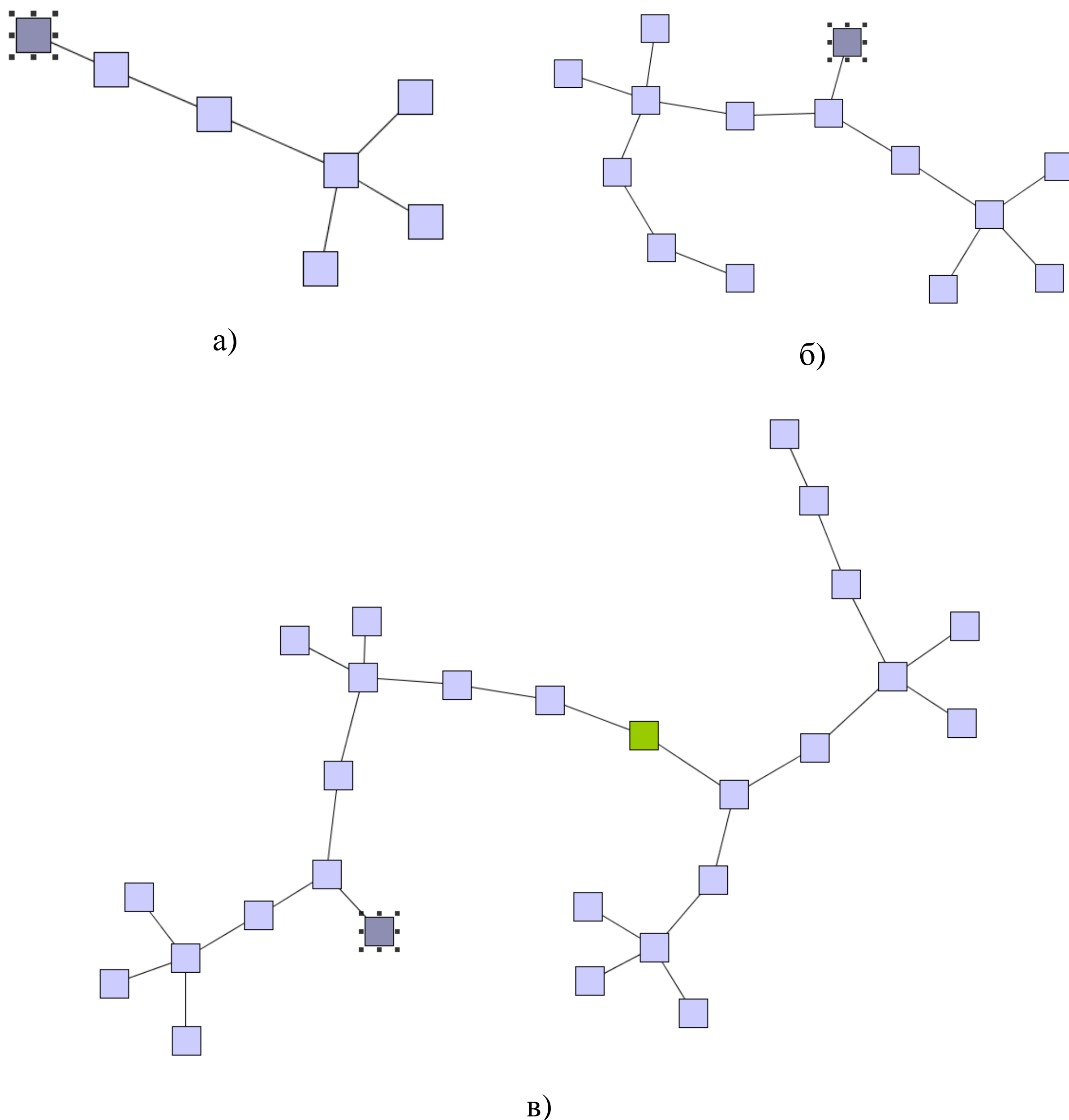

Рис. 4. Построение дерева в процессе исполнения последовательности (Рис. 2). Этот и последующие рисунки выполнены с использованием пакета *yED*

В заключении отметим, что для загрузки и сохранения таких последовательностей удобно использовать формат описания на языке *XML*. Структура вложенных тегов позволяет достаточно удобно описать генерирующую последовательность. Для приведенного примера она записывается в виде:

```
<operations xmlns="http://is.ifmo.ru/testsgen/graph">
 <O><R count="2">
   <O><ST select="selectLast"
                         transform="connectToNew"/></O>
   <O><L count="2">
     <ST select="selectLast" transform="connectToNew"/>
     <O><ST select="selectLast"
                         transform="connectToNew"/></O>
     <O><L count="3">
        <ST select="selectLast" transform="connectToNew"/>
        </L></O>
     </L></O>
  <O><ST select="selectLast" transform="connectToNew"/></O>
  <O><ST select="selectLast" transform="connectToNew"/></O>
</R></O>
</operations>
```

2.4.3. Применение генерирующих последовательностей для создания тестов к олимпиадным задачам

Описанный язык генерации последовательностей является достаточно общим, и для того, чтобы применить его для создания тестов, необходимо решить некоторые практические вопросы.

Размер графа, получающегося в результате генерации, может не подходить под ограничения на максимальный размер, указанные в условии задачи. В случае, когда размер результата интерпретации последовательности операций превышает максимальные установленные ограничения, генерация останавливается.

Оценочная функция последовательности может учитывать как качество генерируемого теста, так и размер самой последовательности, что должно позволять генерировать качественные тесты с достаточно простой, поддающейся анализу структурой.

Для большинства программ можно построить эквивалентные. Например, в приведенной программе можно было бы заменить две первые операции `ST` операцией `R` с числом повторений равным двум. Однако представление не является избыточным, в общем случае указанное преобразование может менять результат исполнения последовательности, так как операции выбора и преобразования могут быть разными или же различаться может способ обработки атрибутов.

Одной из оптимизаций, не влияющих на выразительную мощность языка последовательностей, является запрет таких операций `R`, подпоследовательность которых состоит так же из одной операции `R`, так как эту конструкцию можно заменить на одну операцию `R` с большим числом повторений.

2.4.4. Обработка атрибутов

Для обеспечения возможности работы с цветами и весами ребер, или построения графов с определенной структурой, вершинам и ребрам графа могут назначаться атрибуты (метки), обработка которых зависит от конкретной задачи. Значения этих атрибутов формируются специальной операцией, которая может быть отдельно указана в качестве параметра для любой операции преобразования. Уже вычисленные атрибуты могут использоваться последующими операциями преобразования и выбора для определения действия, а так же функциями, определяющими количество повторений циклов. Конкретный пример того, как обрабатываются атрибуты, можно найти в разд. 3.3.

2.5. Решение задачи оптимизации на пространстве генерирующих последовательностей

После того, как определены процесс генерации графа по последовательности и способ вычисления оценочной функции, для поиска оптимальных графов можно применять стандартные методы поиска.

В качестве малых изменений последовательности на данный момент используются:

- замена одной из операций `S` и `T` на другую операцию этого типа;
- замена числа повторений для одного из операторов `R` или `L`.

2.5.1. Выбор алгоритма оптимизации

Как отмечалось выше, использование генетического программирования может не всегда оказаться достаточно эффективным: в случае, если проверяемое на неэффективность решение работает на тесте секунду, за 10 минут можно выполнить лишь 600 вычислений функции приспособленности, и эта величина может быть недостаточной для работы генетического алгоритма. Поэтому для применения необходимо было выбрать один из методов случайного поиска.

Обзор и оценки для различных методов поиска них приводятся в статьях (20, 21). При этом в работах по случайному поиску [22], и по генерации тестов [3] подчеркивается важность адаптации общего метода оптимизации к конкретной задаче.

Для использования вместе с предложенной в настоящей работе моделью генерирующих последовательностей была выбрана модификация двухфазного метода оптимизации [23]. Исследование применимости подобных методов производится в статье [24], однако отмечается, что предложенные там для общего случая оценки необходимо проверять на практике.

2.5.2. Описание метода

Модификация генерирующих последовательностей может выполняться несколькими способами: случайной сменой структуры (при этом необходимо генерировать заново и параметры циклов), заменой операций S и T при неизменной структуре (сохранении типов операций) и изменением границ циклов. Можно заметить, что модификации структуры последовательности (например, замена операции L на R), существенно влияют на вид результирующего графа. В настоящей работе незначительные изменения структуры не рассматриваются, при необходимости перехода к принципиально другой генерирующей последовательности производится ее полная перегенерация.

Для определения того, какую из модификаций следует выполнить, предлагается сравнивать приближенные оценки на ожидаемое значение функции после каждого из этих типов модификаций последовательности, вычисляемые на базе накопленной статистики двум множествам последовательностей: по всем рассмотренным и по последовательностям с той же структурой. При этом если текущее значение функции хуже, чем для определенной доли рассмотренных примеров, выполняется существенная модификация последовательности, в противном случае осуществляется ее незначительная модификация. Для определения пороговых значений для квантиля, со значением которого сравнивается значение оценочной функции, используются эвристически построенные функции, графики которых приведены на рис. 5.

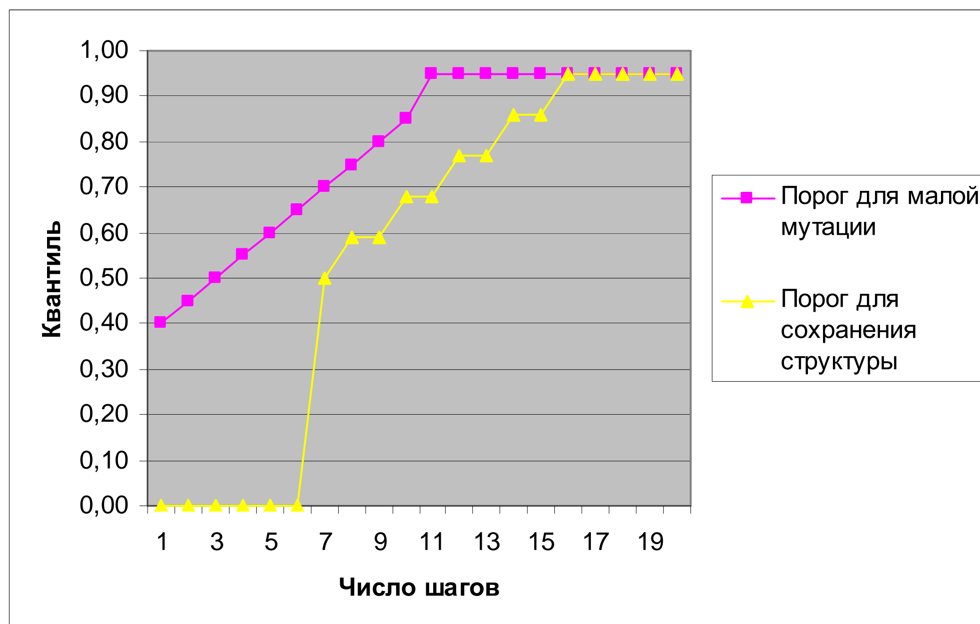

Рис. 5. Последовательность, генерирующая тест для задачи `tree`

График демонстрирует зависимость между количеством шагов оптимизации, выполненном алгоритмом в локальной области поиска, и порядками квантилей, посчитанных на основе множества уже обработанных точек, со значениями (x_{struct} и x_{neighb}) которых сравнивается значение оптимизационной функции в текущей точке f_{cur}, для определения следующего шага:

1. Если $f_{cur} < x_{struct}$, выполняется перегенерация структуры.
2. Если $x_{struct} \leq f_{cur} < x_{neighb}$, выполняется перегенерация параметров без перегенерации структуры.
3. Если $x_{neighb} \leq f_{cur}$, выполняется малая модификация параметров.

На примере задачи, рассматриваемой в разд. 3.4, анализ статистики работы программы показывает, что применение описанного метода увеличивает среднее

значение целевой функции на 30% сравнению со полностью случайным поиском (*Pure Random Search*).

2.6. Описание построенной системы

Для интерпретации генерирующих последовательностей программа строит в памяти дерево, соответствующее этой последовательности, узлами которого являются экземпляры классов, соответствующих описанным выше операциям.

Для чтения и записи последовательностей в формате *XML* используется пакет *XmlBeans* [25] который по составленной *XML Schema* генерирует классы на языке *Java* для работы с соответствующими *XML*-документами. Для вызова этого пакета используется *Apache Ant*.

Операции `S` и `T` работают с изменяющейся параметризованной моделью графа, что позволяет обрабатывать разные представления графа и обрабатывать разные типы атрибутов. При обработке операции `L` первая выбранная вершина кэшируется используется на последующих итерациях.

Ниже описаны интерфейсы, которые следует имплементировать для поддержки новых операций.

Интерфейс класса для выбора вершины:

```
public interface ISelector {
    <VD, ED> GraphGenerationModel<VD, ED>.VertexView
              selectVertex(
                 GraphGenerationModel<VD, ED> gm);
}
```

Интерфейс класса, выполняющего преобразование:

```
public interface ITransformer {
    <VD, ED> void transform(
         GraphGenerationModel<VD, ED> ggm,
         GraphGenerationModel<VD, ED>.VertexView v,
         IAttributeProcessor<VD, ED> ap);
}
```

Интерфейс класса, выполняющего вычисление атрибутов:

```
public interface IAttributeProcessor<VD, ED> {
    public void process(GraphGenerationModel<VD, ED> ggm,
                 GraphGenerationModel<VD, ED>.VertexView v1,
                 GraphGenerationModel<VD, ED>.VertexView v2,
                 GraphGenerationModel<VD, ED>.EdgeView e12,
                 GraphGenerationModel<VD, ED>.EdgeView e21);
}
```

Разработка построенной системы осуществлялась с использованием среды *Intellij IDEA 9.0.2*.

По результатам генерации тестов программа выводит статистику, накопленную в процессе генерации, и для лучшего из созданных тестов сохраняет сформированный входной файл и *XML*-описание генерирующей его последовательности.

ГЛАВА 3. ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ

3.1. Общие параметры генерации тестов

Время работы решения может существенно зависеть от мощности вычислительной машины, на которой оно запускается. Поэтому при генерации тестов, отсекающих неэффективные решения правильно анализировать не абсолютное значение затраченного времени, а соотношение между временем работы предположительно неэффективного решения на сгенерированном тесте и временем работы наиболее медленного из правильных решений на худшем для него тесте, а не абсолютную величину затраченного времени.

В случае, если время работы решения или используемая память экспоненциально зависят от размера входного графа, даже при относительно небольшом изменении его размера время работы меняется существенно. При этом сложно учитывать влияние таких процессов, как чтение и запись данных. Тогда, например, при изменении границ выполнения циклов, размер теста может изменяться на величину, достаточную для того, чтобы упомянутые выше погрешности были существенными. Из-за этого эффективность малых модификаций последовательности оценивать сложно, и эффективность двухфазного метода оптимизации может снизиться..

Переходя к описанию работы предлагаемого метода для конкретных задач, ограничимся сначала рассмотрением задач на деревья. Повторяя рассуждение из разд. 2.4 о способе построении оптимального теста, для этого случая можно показать, что достаточным является одно преобразование: добавление ребра от существующей вершины к новой. Проанализировав генераторы к рассмотренным задачам, можно видеть, что при этом достаточными должны быть две операции

выбора: взятие последней сгенерированной вершины и взятие случайной вершины.

Оценки времени для следующих задач были получены на компьютере *Intel(R) Core(TM)2 Duo 1.50GHz*.

3.2. ЗАДАЧА TREE

Задача была предложена для решения участникам Летних учебно-тренировочных сборов для школьников 2005 года [26]

3.2.1. Условие задачи и решения

В задаче требуется по заданному дереву, содержащему n ($1 \le n \le 250$) вершин, и числу p ($1 \le p \le n$) найти минимальный по размеру набор ребер такой, что, если эти ребра удалить из дерева, то как минимум одна из образовавшихся компонент связности будет содержать ровно p вершин.

Эта задача может быть решена методом динамического программирования., время работы такого решения будет $n^2 \cdot p$.

Среди решений, представленных участниками, встречаются неэффективные переборные решения, много и решений, выдающих неверные ответы на некоторых тестах. При этом интересно, что некоторые решения работают неверно только на небольшом числе тестов.

3.2.2. Построенные вручную тесты

Исследование возможности применения предлагаемого метода для данной задачи показало следующее:

- все решения, которые не проходят какой-либо из тестов к задаче по причине использования неэффективного алгоритма, не проходят хотя бы один из трех созданных вручную тестов (см. далее);

- с помощью созданного теста была найдена ошибка, заключающаяся в неверном ограничении на максимальную доступную память программы в решении жюри, использовавшемся для генерации тестов.

Два из созданных вручную тестов были максимально простыми:

- Дерево, в котором все листья находятся на расстоянии одного ребра от корня:

```
<operations xmlns="http://is.ifmo.ru/testsgen/graph">
    <O><L count="250">
        <O><ST select="selectLast"
               transform="connectToNew"/></O>
    </L></O>
</operations>
```

- Дерево, построенное по методу выбора случайного родителя:

```
<operations xmlns="http://is.ifmo.ru/testsgen/graph">
    <constraints maxGraphSize="250"/>
    <O><R count="250">
        <O><ST select="selectRandom"
               transform="connectToNew"/></O>
    </R></O>
</operations>
```

Два представления этого дерева изображены на рис. 6, 7.:

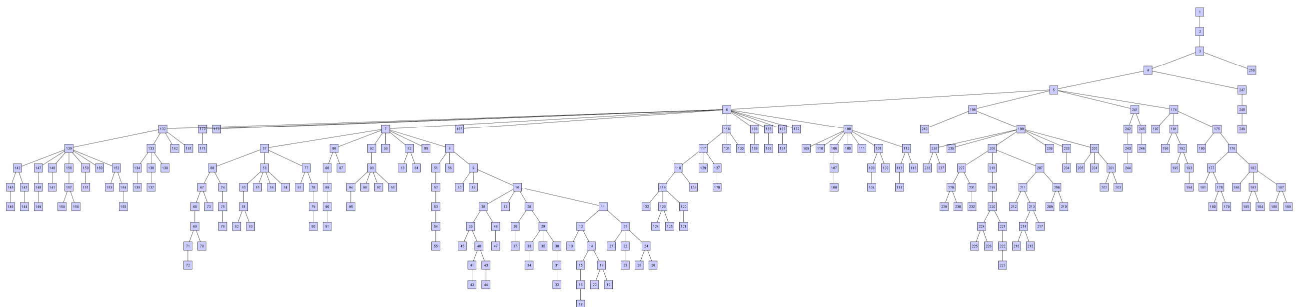

Рис. 6. Дерево из 250 вершин, построенное методом случайного выбора родителя

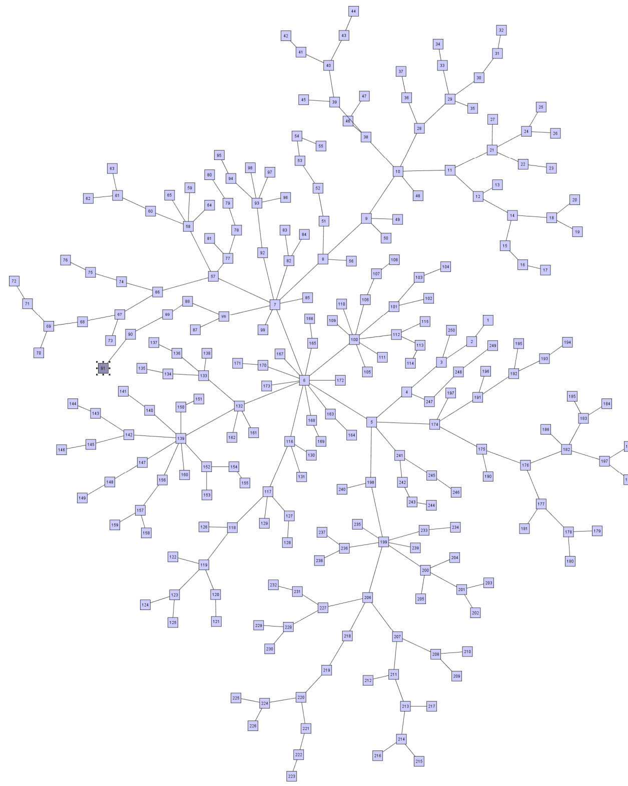

Рис. 7. Другая укладка случайного дерева из 250 вершин

Третья последовательность (рис. 8) была построена эвристическим образом для генерации теста к рассматриваемой задаче.

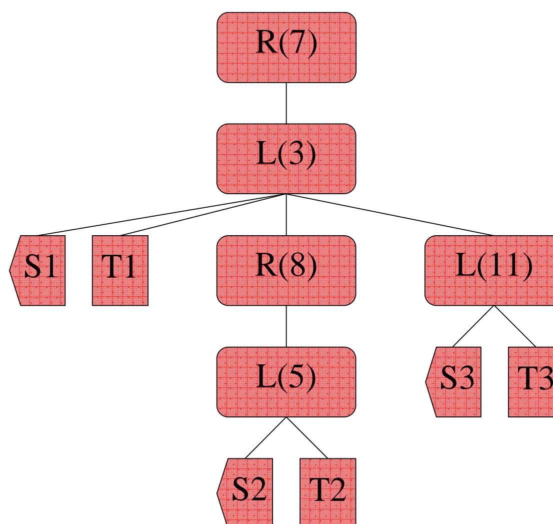

Рис. 8. Последовательность, генерирующая тест для задачи `tree`

На рис. 9 представлен вид дерева, генерируемого эту последовательность на четырех промежуточных стадиях генерации, и полное результирующее дерево в двух представлениях.

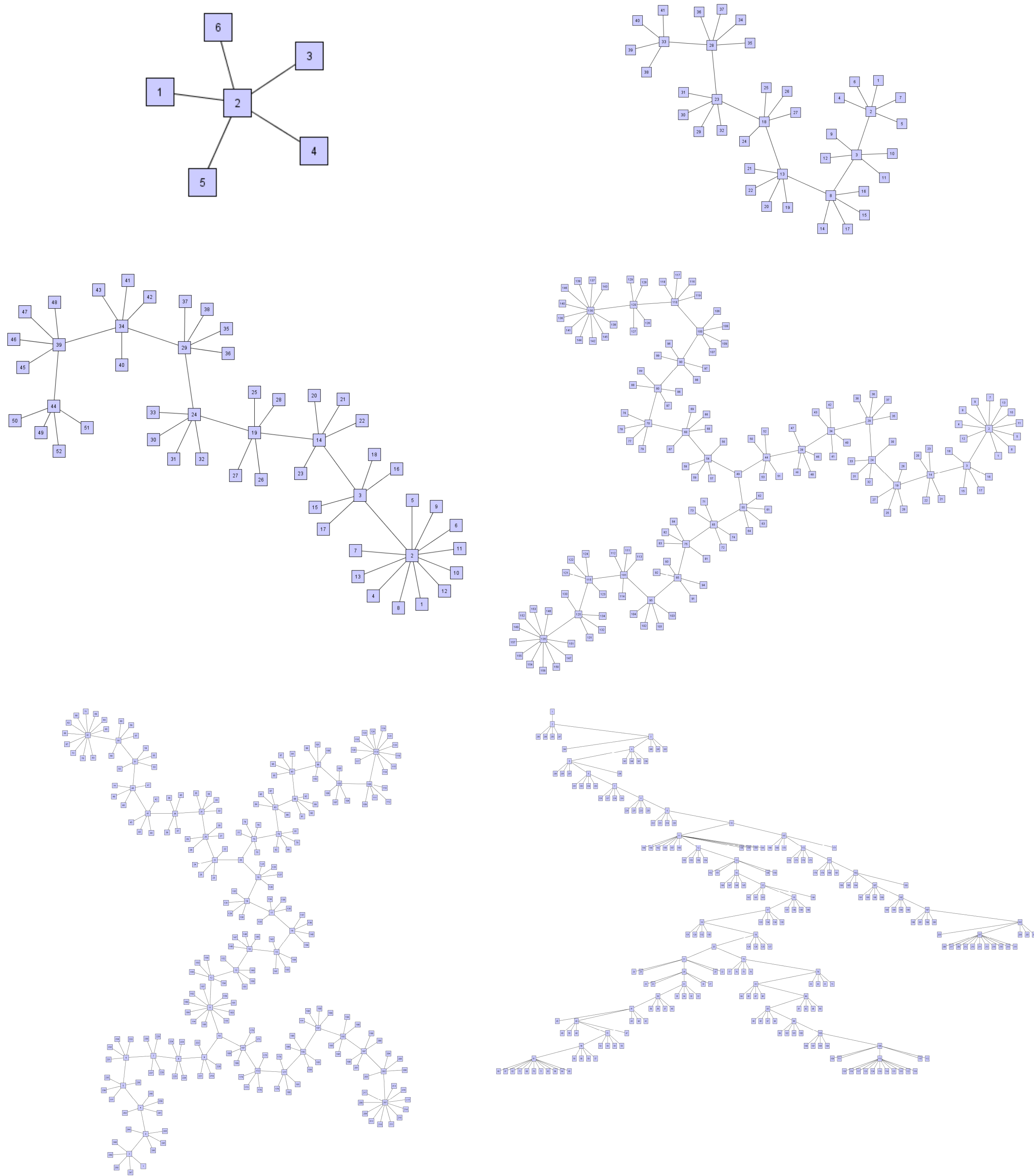

Рис. 9. Дерево, сгенерированное по второму примеру программы (рис. 8)

Для трех указанных деревьев было построено по девять тестов с разными значениями параметра *p*.

Проверка показала, что этот набор тестов выявил все неэффективные решения, а так же все кроме одного решения среди содержащих ошибки.

Кроме того, оказалось, что на трех из этих тестов решение жюри, использовавшееся для построения ответов, завершается с ошибкой из-за превышения указанного в заголовке программы размера, отведенного на стек.

3.2.3. Исследование статистики по тестам небольшого размера

В рассмотренной задаче не было найдено возможности для применения метода автоматизированной генерации тестов с использованием поиска. Однако, простое представление выходных данных позволяет исследовать работу решений на всех возможных экземплярах входных данных, размер которых ограничен небольшим числом и, тем самым, сравнить методы генерации тестов с помощью генерирующих последовательностей и традиционных методов случайного построения тестов.

Те из решений участников сборов, которые проходили как минимум половину тестов жюри, но не все из этих тестов, были запущены на наборе тестов, полученных перечислением всех различных деревьев без выделенного корня размера 12. Для каждого дерева по всем тестам с этим деревом (с разными значениями *p*) была подсчитана сумма количеств решений, отсекаемых тестом (эту величину можно назвать качеством дерева как теста).

Кроме того, для деревьев были подсчитаны вероятности быть сгенерированными двумя классическими случайными алгоритмами (выбора случайного родителя и проведения случайного ребра, не добавляющего цикла), а так же случайным образом выбранной генерирующей последовательностью для последовательностей длиною от трех до семи операций.

Оказалось, что:

- тесты, которые наиболее редко генерируются случайными алгоритмами, достаточно часто получаются в результате работы генерирующих последовательностей;
- значения корреляции между качеством дерева и вероятностью быть сгенерированными для генерирующих последовательностей и для двух случайных алгоритмов составляют –0.9, –0.5 и 0.13 соответственно.

Второй из этих результатов позволяет предположить, что, даже в случае невозможности построения достаточно достоверной оценочной функции для теста, генерирующие последовательности могут быть использованы наряду со случайными тестами (многие из которых также могут быть описаны на языке генерирующих последовательностей с использованием операции `selectRandom`), для генерации предварительных наборов тестам к некоторым задачам.

Полную версию таблицы с данными, по которым были сделаны эти выводы, можно найти в электронной таблице, выложенной в сети Интернет [27].

3.3. Задача ROBOT

Эта задача с городской олимпиады школьников Санкт-Петербурга по информатике 2005 года [33].

В задаче задается дерево из ($1 \le n \le 100000$) вершин, для каждого из ребер которого указан один из c ($1 \le c \le 100000$) цветов. Необходимо перечислить набор вершин, для которых при запуске поиска в глубину с началом в этой вершине не возникает ситуации, когда среди непросмотренных ребер из вершины хотя бы два имеют один цвет.

Наиболее эффективное решение этой задачи, за которое участники могли получить полный балл, использует специальный обход в глубину графа с анализом цветов ребер и работает за $O(n)$.

Большинство неэффективных решений не работают на достаточно большой доле тестов, и для поиска тестов, требующих применения оптимизационного метода, было выбрано самое быстрое из субоптимальных решений. Среди тестов жюри таких четыре из 40.

3.3.1. Задание цветов ребер

Существенным отличием этой задачи от предыдущей, помимо другой асимптотической оценки оптимального решения, является необходимость задания цветов для ребер.

Согласно предложению из разд. 2.4.4, для этого будет определена операция вычисления атрибутов. Для назначения цветов ребер графа был определен тип атрибута `ColorEN`. Вычислять значение этого атрибута при проведении нового ребра могут три операции (возможно, здесь следует напомнить, что добавлять ребра может только одна операция `connectToNew`):

- `asParent` (копирование значения от родительской вершины);
- `other` (выбор цвета, отличающегося от всех цветов уже проведенных от предыдущей вершины ребер);
- `random` (выбор цвета случайным образом).

3.3.2. Результаты тестирования

В результате работы генератора тестов был построен тест, на котором упомянутое выше решение работает 4 с, что повторяет по этому критерию значение для лучшего из тестов, на которых проверялись решения участников олимпиады. Время работы решений жюри на самых трудных тестах составляет 0,6 с.

Генерирующая последовательность построенного теста выглядит следующим образом:

```
<operations xmlns="http://is.ifmo.ru/testsgen/graph">
  <constraints maxGraphSize="100000"/>
  <O><R count="90000">
    <O><ST select="selectLast" transform="connectToNew"
              attributeProcessor="colorEN_other"/></O>
  </R></O>
  <O><L count="100">
    <ST select="selectLast" transform="connectToNew"
              attributeProcessor="colorEN_asParent"/>
    <O><R count="10">
      <O><ST select="selectRandom" transform="connectToNew"
              attributeProcessor="colorEN_asParent"/></O>
    </R></O>
  </L></O>
</operations>
```

Побочным эффектом, по совпадению, было обнаружение теста, на котором эталонное решение жюри уже к этой задаче выходило за пределы указанного в заголовке решения объема памяти.

3.4. Задача Clique

По заданному графу из n ($1 \leq n \leq 50$) вершин требуется посчитать количество непустых полных подграфов (клик) в этом графе. Эту задача находится на известном в олимпиадной среде сайте УрГУ *acm.timus.ru* [32]

Правильное решение задачи построено с использованием метода *Meet-in-the-middle*. Этот метод, наиболее известный благодаря основанному на нем методу метод криптографической атаки [28] впервые был опубликован в статье [29], а сейчас входит во многие курсы по Computer Science [30] и является ключевой идеей для решения целого класса олимпиадных задач (например, [31]).

3.4.1. Используемые элементарные операции

В отличие от двух предыдущих рассмотренных задач, которые работали с деревьями, для генерации тестов к этой задаче требуется составление графов общего вида.

Исходя из этого были добавлены новые элементарные операции:

- операция выбора `selectLeastDegree`, выбирающая одну из вершин графа с минимальной степенью;
- операция преобразования `connectToLeastDegree`, соединяющая выбранную вершину ребром с одной из вершин с минимальной степенью (или со следующей в порядке возрастания степенью, если минимальной степенью на момент выполнения операции обладает текущая вершина);
- операция преобразования `connectToRandom`, соединяющая вершину со случайно выбранной;
- операция преобразования `copyAllFromRandom`, копирующая все ребра от выбранной случайным образом вершины;
- операция преобразования `copyAllToNew`, создающая новую вершину и копирующая все ребра от выбранной вершины.

3.4.2. Результаты тестирования

Результаты первых запусков программы генерации были неудовлетворительными. После этого в процесс генерации тестов было внесено изменение, соответствующее следующему эвристическому соображению: некоторые алгоритмы плохо справляются с более плотными графами, поэтому может быть целесообразно вместе с графом рассматривать инвертированный граф. Такое изменение позволяет описывать графы с большим числом ребер более короткими генерирующими последовательностями.

После этого был получен удовлетворительный результат: системой сгенерирован тест, на котором время работы неправильного решения в 1,5 раза превосходит максимальное время работы правильного решения на тестах, удовлетворяющих указанным ограничениям.

Генерирующая последовательность для построенного теста выглядит следующим образом:

```
<operations xmlns="http://is.ifmo.ru/testsgen/graph">
  <constraints maxGraphSize="50"/>
  <O><ST select="selectLeastDegree"
               transform="copyAllFromRandom"/></O>
  <O><L count="33">
      <ST select="selectLeastDegree"
               transform="connectToNew"/>
      <O><ST select="selectLast"
               transform="copyAllFromRandom"/></O>
      <O><ST select="selectLast"
               transform="connectToNew"/></O>
  </L></O>
</operations>
```

ЗАКЛЮЧЕНИЕ

В работе предложен способ построения тестов для задач по теории графов с помощью генерирующих последовательностей. В комбинации с двухфазным методом оптимизации, он позволяет систематизировать процесс построения эффективных тестов. На примере трех разных задач продемонстрирована возможность применения этого метода для поиска тестов, оптимальных по заданному критерию, и, в некоторых случаях, даже в отсутствии такого критерия.

ИСТОЧНИКИ

1. *G. J. Myers*. The Art of Software Testing. John Wiley & Sons, Inc., 2004.
2. *C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball.* Feedback-directed Random Test Generation
3. *A. Arcuri.* Insight Knowledge in Search Based Software Testing
4. *A. Arcuri,* X. Yao. Search Based Software Testing of Object-Oriented Containers
5. *P. Godefroid, M. Y. Levin, D. Molnar.* Automated Whitebox Fuzz Testing
6. *P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, W. Tillmann, M. Y. Levin.* Automating Software Testing Using Program Analysis
7. *Оршанский С. А.* О решении олимпиадных задач по программированию формата ACM ICPC //Мир ПК. 2005. № 9.
8. *TopCoder.* http://www.topcoder.com/tc.
9. *Google Code Jam.* http://www.google.com/codejam
10. *Интернет-олмпиады по информатике.* http://neerc.ifmo.ru/school/io/
11. TopCoder Studio. http://www.topcoder.com/studio
12. *Буздалов М. В.* Применение генетических алгоритмов для генерации тестов, выявляющих неэффективные решения олимпиадных задач по программированию, на примере задачи о рюкзаке. СПбГУ ИТМО. Бакалаврская работа. 2009.
13. *Харари Ф.* Теория графов. М.: Едиториал УРСС, 2003.
14. *North-East Europeran Regional Contest*. http://neerc.ifmo.ru
15. *R. Kumar, P. Raghavan, S. Rajagopalan, A. S. Tomkins, D. Sivakumar, E. Upfa* . Stochastic models for the web graph.
16. J. *M. Klenberg, R. Kumar, P. Raghavan, S. Rajagopalan, A. S. Tomkins.* The Web as a graph: measurements, models, and methods.
17. *M. E. J. Newman.* Random graphs as models of networks

18. *A. B. Yoo, Y. Liu, S. Vaidya, S. Poole.* A New Benchmark For Evaluation Of Graph-Theoretic Algorithms.
19. *J. Koza* Genetic programming. On the Programming of Computers by Means of Natural Selection. MA: The MIT Press, 1998.
20. *Z. B. Zabinsky*. Random Search Algorithms
21. *Z. B. Zabinsky, R.L. Smith, J.F. McDonald, H.E. Romeijn, and D.E. Kaufman.* Improving Hit and Run for Global Optimization. Journal of Global Optimization, 3, 171 – 192. 1993.
22. *M. S. Sarma.* On the convergence of the Baba and Dorea random optimization methods.
23. *F. Schoen*. Two-phase Methods for Global Optimization. Handbook of Global Optimization/ Volume 2/ Kluwer Academic Publishers, Netherlands, 2002, pp. 151 – 177.
24. *M. Muselli.* A Theoretical Approach to restart in Global Optimization.
25. *Страница проекта* XmlBeans.
26. *Летние учебно-тренировочные сборы* для школьников 2005 года.
27. *Ссылка на таблицу*, содержащую подробную статистическую информацию об оценке качества деревьев малого размера как тестов для задачи Tree. https://spreadsheets.google.com/ccc?key=0AsQaOfyDLOBMdFk4TWZsV3ZZRGY4U0VCay1wam5iaUE&hl=en&authkey=CKTfq9MN
28. *Wikipedia*. http://en.wikipedia.org/wiki/Meet-in-the-middle_attack
29. *E. Horowitz, and S. Sahni.* Computing Partitions with Applications to the Knapsack Problem. Journal of the ACM. V. 21, №. 2. April 1974, pp 277 – 292.
30. *G. J. Woeginge.,* Exact Algorithms for NP-Hard Problems., Lecture Notes in Computer Science, Springer Verlag Heidelberg, vol. 2570, pp. 185 – 207, 2003.
31. *Разбор задач TopCoder* High School Competition, 2007. http://www.topcoder.com/tc?module=Static&d1=hs&d2=match_editorials&d3=tchs07Semi

32. *Timus Online Judge*, архив задач с проверяющей системой. http://acm.timus.ru.
33. *Санкт-Петербургская городская олимпиада школьников по информатике*. 2005 год. http://neerc.ifmo.ru/school