Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Буздалов Максим Викторович

Применение генетических алгоритмов для генерации тестов, выявляющих неэффективные решения олимпиадных задач по программированию, на примере задачи о рюкзаке

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

Санкт-Петербург
2009

Содержание

Введение

Тестирование является важнейшей частью цикла разработки программного обеспечения (ПО). Оно занимает около 50 % времени и более 50 % стоимости разработки ПО. Так как качество тестирования сильно зависит от влияния человеческого фактора, исследователи и разработчики сосредоточили большие усилия над процессом автоматизации тестирования. В частности, среди новейших тенденций в этой области встречаются работы [1, 2] по *эволюционному* тестированию программ.

Олимпиадное движение в области информатики и программирования активно развивается как в России, так и в мире [3—5]. Олимпиады способствуют выявлению талантливых программистов среди школьников и студентов. Решаемые на этих олимпиадах задачи имеют большое число специфических особенностей. В частности, проверка решений задач на имеющихся тестах может быть полностью автоматизирована, в то время как создание тестов в настоящее время выполняется вручную. Качество проведения соревнований по программированию напрямую зависит от качества тестов, составленных к используемым на них задачам. Подготовка тестов к таким задачам является сложным и творческим процессом, и любая работа по автоматизации этого процесса сказывается позитивным образом на качестве соревнования в целом.

Задача о рюкзаке является хорошо изученной NP-полной задачей [6] с множеством применений. Существуют алгоритмы решения этой задачи, которые для большинства входных данных достаточно быстро находят оптимальный ответ [7]. Однако, для любого такого алгоритма существуют входные данные небольшого размера, на которых он работает очень долго (за время, экспоненциально зависящее от объема входных данных). Поиск и описание таких данных — необходимая часть анализа любого алгоритма решения задачи о рюкзаке [8]. Однако, нельзя не отметить тот факт,

что составление тестовых данных для задачи о рюкзаке является сложной задачей, в некоторых случаях нерешаемой традиционными методами. Тесты для задач этого класса обычно создают по некоторому шаблону или генерируют случайно [7], поэтому полнота наборов тестов, получаемых таким образом, чаще всего оказывается неудовлетворительной. Это может повлечь за собой переоценку эффективности алгоритма и тяжелые последствия при практическом его применении.

Генетические алгоритмы — широко развивающийся в последнее время класс оптимизационных алгоритмов, использующих идеи искусственного интеллекта и предназначенный для ведения направленного поиска [9, 10]. С их помощью могут быть найдены решения многих задач в различных областях, таких как составление расписаний, построение конечных автоматов, построение и настройка искусственных нейронных сетей. Генетические алгоритмы, в том числе такая перспективная разновидность, как *генетическое программирование*, успешно решают задачи тестирования программного обеспечения, такие как тестирование встроенных систем [**?**] и автоматическое построение модульных тестов [1, 2]. Однако для олимпиадных задач подходы, реализованные в указанных работах, неприменимы.

В настоящей работе предложен *метод генерации тестов* для олимпиадных задач по программированию с применением генетических алгоритмов. Эти тесты предназначены для выявления решений олимпиадных задач, неэффективных по времени выполнения. При таком подходе повышается степень автоматизации процесса генерации тестов и их качество. При анализе результатов, полученных при применении данного подхода к генерации тестов для задачи о рюкзаке, выявлен *новый класс* тестовых данных для задачи о рюкзаке, являющийся экстремально сложным для некоторых алгоритмов решения этой задачи. Предложенный метод был применен к генерации тестов для реальной олимпиадной задачи и позволил «убить» все известные решения [11].

Глава 1. Обзор предметной области

1.1. Тестирование программного обеспечения

Цель тестирования программного обеспечения (ПО) — удостовериться в том, что программа работает именно так, как было задумано, и что она не делает того, что ей не следует делать [12]. В общем виде эта задача алгоритмически неразрешима согласно теореме Райса [**?**]. Поэтому тестирование ПО до сих пор в значительной степени базируется на эвристическом подходе.

В книге [12], ставшей классикой, отмечено, 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. Популярность и эффективность таких методологий производства ПО, как *экстремальное программмирование* [13], показывают, что даже частичная автоматизация процесса тестирования существенно повышает качество создаваемого ПО.

1.1.1. Традиционные виды тестирования

В работе [12] рассматривается множество широко применяемых видов тестирования. Они включают в себя 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 же время, модульное тестирование [13] подразумевает написание человеком теста, который в дальнейшем автоматически запускается при проверке модуля программы.

1.1.2. Эволюционное тестирование

Если качество теста удается выразить каким-либо количественным критерием, то становится возможным поиск качественных тестов с помощью оптимизационных алгоритмов. В этой области тестирования находит применение генетическое программирование [1, 2]. Тесты при данном подходе выражены в виде программы, которая подготавливает систему к работе с некоторыми входными данными, запускает тестируемый программный модуль и проверяет результаты на соответствие спецификации.

При генерации модульных тестов с применением этого подхода человек не реализует тесты самостоятельно, а лишь направляет и контролирует процесс автоматического поиска теста. Однако ни критерии оптимизации, используемые в этом процессе, ни способы представления тестов не подходят для автоматической генерации тестов для олимпиадных задач.

1.2. Олимпиадные задачи по программированию

1.2.1. Олимпиадное движение

В мире проводится большое число олимпиад по программированию. Среди них можно отметить международную студенческую олимпиаду по программированию *International Collegiate Programming Contest* [3], проводимую *Association for Computing Machinery* (далее олимпиада будет упоминаться как *ACM ICPC*), с развитой сетью отборочных соревнований, международную олимпиаду школьников по информатике [4], соревнования, проводимые компанией *TopCoder* [14], интернет-олимпиады по инфор-

матике и программированию [5] и многие другие.

1.2.2. Задачи, используемые на олимпиадах

На большинстве олимпиад по программированию предлагается решить одну или несколько задач. Формулировка задачи в большинстве случаев предполагает чтение входных данных, удовлетворяющих условию задачи, получение требуемых результатов на основе этих данных и вывод результатов в формате, указанном в условии задачи. Решением задачи является программа, написанная на одном из алгоритмических языков (например, в соревновании *ACM ICPC* используются языки *C*, *C++* и *Java* [15]). Запуск программы, ввод и вывод данных осуществляются различными способами, наиболее распространенные из которых приведены ниже.

- Входные данные находятся в файле с заранее известным именем. Выходные данные также должны быть записаны в определенный файл. Решение при этом представляет собой консольное приложение.
- Вариация предыдущего подхода: один или оба файла заменяются на поток стандартного ввода (для входного файла) или вывода (для выходного файла). Этот и предыдущий подход характерен для соревнований формата *ACM ICPC* [3].
- Решение также представляет собой консольное приложение, однако для работы с проверяющей системой оно должно использовать функции некоторой библиотеки. Участникам олимпиады известно только описание этих функций, но не их реализация. С помощью этой библиотеки тестирующая система снабжает тестируемую программу входными данными, а также получает от нее результаты работы. Такой подход применяется для некоторых задач международной олимпиады школьников по информатике [4].
- Решение запускается с перенаправленными потоками стандартного ввода и вывода. К этим потокам подключена программа, написанная

жюри, которая снабжает тестируемую программу данными и получает от нее результаты по определенному, как правило текстовому, протоколу. Задачи такого типа были предложены в качестве эксперимента на полуфинале *ACM ICPC* в Северо-Западном регионе в 2008 году [16].

- Решение представляет собой класс, реализованный на одном из объектно-ориентированных языков программирования. В нем должен быть определен метод с заданной сигнатурой, который вызывается проверяющей системой. Аргументы этого метода являются входными данными для такого решения, а возвращаемое значение считается результатом работы. Такой подход используется системой соревнований *TopCoder* [14].

Программа тестируется на наборе из тестов, неизвестных участникам. На работу программы накладываются определенные ограничения: максимальное время выполнения, максимальный объем используемой памяти, запрет на выполнение определенных операций (например, работа с сетью, графикой, оконной подсистемой). Программа считается прошедшей определенный тест, если она при работе с ним не нарушила ограничений, завершилась корректно (без ошибок времени выполнения), и ее ответ признан правильным. О конкретных видах задач и ограничениях можно прочитать, например, на сайте олимпиады [15].

Решение таких задач развивает навыки программирования, исследовательской работы, а на некоторых соревнованиях — навыки работы в команде. В статье [17] изложен процесс решения отдельно взятой задачи при одиночной работе участника, а в статье [18] описана работа в команде.

1.2.3. Цель тестов к олимпиадным задачам

Проведение соревнований на высоком уровне предполагает качественную подготовку задач. Этот процесс включает в себя выбор интересных идей для будущих задач, написание условий и решений, а также

составление тестов. Из этих видов деятельности составление тестов кажется наиболее формализуемым.

В условии задачи на входные данные накладываются некоторые ограничения. Тем не менее, для большинства задач тестирование решения на всех возможных тестах, удовлетворяющих этим ограничениям, не представляется возможным. В то же время, цель составления набора тестов заключается в выявлении неверных и неэффективных решений. Следовательно, возникает задача 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 [3] до заданного процента от числа тестов [4]).

1.2.4. Традиционные методы подготовки тестов

Подготовка тестов к олимпиадной задаче является творческим процессом. По причине того, что этот процесс на настоящий момент практически не описан в литературе, будет приведено описание процесса подготовки тестов к задачам интернет-олимпиад по информатике и программированию, проводимых СПбГУ ИТМО [5], так как автор данной работы в течение длительного времени принимал участие в этом процессе.

Для каждой задачи в обязательном порядке пишутся несколько решений, известных как *решения жюри*. Среди них должны быть как корректные, так и неверные или неэффективные решения. Их цель — проверка качества тестов: любое корректное решение должно пройти все тесты, для любого некорректного решения должно существовать определенное число тестов, которое оно не проходит. Это число может зависеть от типа соревнования, а также от степени некорректности решения, но всегда должно отличаться от нуля.

Тесты могут иметь различные цели, такие как:

- выявление неверных решений (выдающих неверный ответ);

- выявление неэффективных решений (работающих за время, превышающее ограничения);
- выявление ошибок реализации в решении (с различными результатами, включающими ошибки времени выполнения);
- выявление неверно разобранных случаев.

Часть тестов пишется вручную. Такие тесты проверяют решения на корректность разбора случаев, встречающихся в задаче. Также эти тесты могут проверять корректность того, как решение работает на «минимальных» тестах, то есть таких, в которых некоторые характерные величины принимают минимальное значение.

Тесты большого размера вручную генерировать неэффективно. Вместо этого они генерируются согласно некоторому шаблону. Так, например, если в условии задачи фигурирует некоторый граф, то можно сгенерировать двоичное дерево, полный двудольный граф, полный граф [19] и другие виды графов. Некоторые шаблоны подразумевают множество вариантов реализации: так, например, дерево с N пронумерованными вершинами можно сгенерировать N^{N-2} различными способами [19]. В таком случае при генерации теста некоторые действия выполняются случайным образом.

Не все возможные случаи ошибок, встречающихся в решениях, можно покрыть тестами, сгенерированными предыдущими способами. Чтобы уменьшить количество таких случаев, создаются «случайные тесты». Например, если входными данными задачи является граф с N вершинами и M ребрами, то генерируется несколько тестов, в которых каждое ребро соединяет две произвольно взятые вершины. Назначение этих тестов — выявление в решении таких ошибок, которые могут не проявиться при прохождении тестов, составленных по шаблону.

Для некоторых задач возможно написать некорректное решение, которое не было предусмотрено жюри. В этом случае в наборе тестов может и не оказаться такого теста, на котором это решение не работает. Авторы

задач стремятся предотвратить такое развитие событий, для чего реализуют некоторые «эвристические» решения и генерируют тесты против них. Однако, поиск таких тестов может затянуться на неопределенное время. В связи с этим, на соревнованиях иногда встречаются задачи со слабым набором тестов, которые зачастую пропускают некорректные решения. Это, в свою очередь, приводит к тому, что «сильные» участники соревнований, ищущие корректные решения, не решают такие задачи, в то время как менее сильные участники успешно сдают различные некорректные решения. Все это существенно снижает качество соревнования.

1.3. Задача о рюкзаке

1.3.1. Постановка задачи

Задача о рюкзаке является известной задачей комбинаторной оптимизации. В простейшем виде она формулируется следующим образом. Дан рюкзак, вмещающий произвольный набор предметов суммарным весом не более W. Даны также n предметов с весами $w_j > 0$ и стоимостями $p_j > 0$. Необходимо

$$\begin{aligned} \text{максимизировать } & \sum_{j=1}^{n} p_j x_j \\ \text{при условиях } & \sum_{j=1}^{n} w_j x_j \le W \\ & x_j \in \{0;1\}, 1 \le j \le n \end{aligned}$$

Если стоимости предметов считать равными весам, то в результате получим *задачу о сумме подмножеств*:

$$\begin{aligned} \text{максимизировать } & \sum_{j=1}^{n} w_j x_j \\ \text{при условиях } & \sum_{j=1}^{n} w_j x_j \le W \\ & x_j \in \{0;1\}, 1 \le j \le n \end{aligned}$$

1.3.2. Применение задачи о рюкзаке в теории и на практике

Задача о рюкзаке имеет большое число практических применений. К ней сводятся различные задачи из таких областей, как логистика и планирование финансов [7]. Кроме того, задачу о равномерном распределении нагрузки между двумя процессорами можно рассматривать как экземпляр задачи о сумме подмножеств, как показано в работе [20]. Также, в статье [21] была предложена схема шифрования, сложность взлома которой зависит от сложности решения задачи о сумме подмножеств.

В теоретическом плане задача интересна как сама по себе, так и как подзадача для других комбинаторных задач (в частности, для получения верхних и нижних оценок в задачах, решаемых перебором с отсечениями). Так, в статье [22] было показано, что любая задача целочисленного программирования сводится к задаче о рюкзаке, а в работе [7] отмечено, что задача о рюкзаке является подзадачей при решении обобщенной задачи о назначениях.

1.3.3. Решения задачи о рюкзаке

Задача о рюкзаке является NP-полной [6]. Это означает, что для любого решения такой задачи существуют входные данные небольшого размера, на которых это решение работает долго. Тем не менее, можно реализовать такие решения, которые на большинстве входных данных будут работать за сравнительно небольшое время. В этом аспекте задача о рюкзаке известна как «самая легкая из NP-полных задач»: многие решения этой задачи работают на почти всех входных данных за время, пропорциональное их длине. Так, несколько решений, предложенных в работе [7], работают менее, чем за секунду, на большинстве входных данных с N порядка $10^5 \dots 10^6$ на обычном настольном компьютере. Разработка эффективных решений для этой задачи необходима в силу того, что любое улучшение времени работы существенно влияет на эффективность решения других теоретических и практических задач.

Среди решений задачи о рюкзаке можно выделить два класса: псевдополиномиальные решения и решения, использующие перебор с отсечениями.

Решения, принадлежащие первому классу, имеют асимптотическую сложность, зависящую не только от числа предметов N, но и от вместимости рюкзака W, максимального веса предмета w_{max}, максимальной стоимости предмета p_{max} и других подобных величин. Наиболее известное из таких решений [23] имеет асимптотическую сложность времени работы $O(N \cdot W)$ и использует динамическое программирование. В [7] предложен алгоритм с асимптотической сложностью времени работы $O(N \cdot w_{max}^2)$.

Решения, принадлежащие второму классу, используют другой принцип работы — «метод ветвей и границ». Такие решения чаще всего оформлены в виде рекурсивной процедуры. На каждом запуске этой процедуры они перебирают несколько возможных в данном случае альтернативных вариантов (например, положить ли некоторый предмет в рюкзак или отложить его), вызывая эту же процедуру с другими параметрами для каждой из таких альтернатив. Для того, чтобы уменьшить время работы алгоритма, некоторые возможные случаи исключаются из перебора на основе анализа верхних и нижних оценок на качество получаемого решения. Например, если можно доказать, что в текущем поддереве перебора нельзя получить более оптимальное решение, чем полученное ранее, то перебор в этом поддереве можно прекратить сразу. Эффективные решения, принадлежащие этому классу, описаны в работах [7], [20] и других источниках.

Некоторые решения могут изменять поведение в зависимости от получаемых оценок или комбинировать описанные подходы. Такие решения можно причислить сразу к двум описанным классам. Так, в работе [7] описано решение задачи о сумме подмножеств с асимптотической сложностью $O(\min(N \cdot W, 2^b + 2^{N-b}))$, где b — номер первого предмета, не помещающегося в рюкзак при «жадном» его заполнении.

1.4. Генетические алгоритмы

1.4.1. Концепция генетических алгоритмов

Основа концепции генетических алгоритмов была заложена в опубликованной Дж. Холландом (J. P. Holland) в 1975 году книге «Адаптация в естественных и искусственных системах» [9]. Генетические алгоритмы предназначены для решения задач оптимизации и используют механизмы, напоминающие механизмы естественной эволюции.

Потенциальные решения исследуемой задачи в генетическом алгоритме представлены в виде *особей*. Каждая особь кодируется с помощью полностью описывающего ее объекта — *хромосомы*. Совокупность особей, существующих одновременно в процессе работы алгоритма, называется *популяцией*.

Генетические алгоритмы используют следующие принципы, которым подчинена естественная эволюция [10]:

- Принцип выживания сильнейших. Этот принцип был сформулирован Ч. Дарвином в 1859 году в книге «Происхождение видов путем естественного отбора» [**?**]. Согласно этому принципу, особи, которые лучше приспособлены для существования в своей среде, выживают с большей вероятностью и имеют больше потомков. По аналогии с этим принципом, каждой особи генетического алгоритма назначается значение *приспособленности* — мера того, насколько решение, соответствующее этой особи, является оптимальным. Функция, по особи возвращающая значение приспособленности, называется *функцией приспособленности*, или *фитнесс-функцией*. Генетический алгоритм дает особям с лучшим значением приспособленности больше шансов на выживание и на генерацию потомства.
- Принцип скрещивания. В 1865 году Г. Менделем был открыт тот факт, что хромосома потомка состоит из частей хромосом родителей [**?**]. Формализация этого принципа в применении к генетическим

алгоритмам дает основу для *оператора скрещивания* (*кроссовера*).

- Принцип мутации. В 1900 году Х. де Фризом была открыта мутация генов [**?**]. Мутация служит причиной появления у потомка признаков, отсутствующих у родителей, а если она приводит к улучшению приспособленности, соответствующие признаки остаются в популяции. По аналогии с этим принципом, генетические алгоритмы используют подобный механизм для изменения свойств потомков и, как следствие, повышения разнообразия особей в популяции и получения новых направлений поиска.

Основные преимущества генетических алгоритмов таковы:

- Использование только значений функции в точках. В теории оптимизации известны методы оптимизации первого, второго, а также высших порядков [24], использующие информацию о поведении оптимизируемой функции в окрестности рассматриваемых точек. Однако, они накладывают определенные ограничения на оптимизируемую функцию, например, наличие в области оптимизации производных этой функции до некоторого порядка включительно. В отличие от них, генетические алгоритмы используют только значения функции. Таким образом, генетические алгоритмы могут быть использованы для решения задач оптимизации функций, имеющих разрывы, а также дискретных функций.
- Использование на каждой итерации алгоритма множества рассматриваемых значений. В отличие от методов локальной оптимизации (*метод имитации отжига* [25], *метод спуска* [26]), генетические алгоритмы ведут поиск по множеству различных направлений одновременно, имея поэтому меньший шанс сойтись к локальному, но не к глобальному оптимуму. Кроме того, за счет комбинаций субоптимальных решений из различных локальных оптимумов генетические алгоритмы могут получать более оптимальные решения.
- Возможность глубокого распараллеливания. Генетические алгорит-

мы легко модифицируются для выполнения на многопроцессорных и распределенных системах.

В то же время, генетическим алгоритмам присущи и некоторые недостатки:

- Высокая трудоемкость. Для поиска близкого к оптимальному решений нередко приходится рассматривать большое число особей. Хотя это число, как правило, значительно меньше размера допустимого пространства поиска, для многих задач оно все же остается большим [27].
- Сложность оценки применимости генетического алгоритма к конкретной задаче. Как правило, какие-либо теоретические оценки на вероятность и скорость сходимости к оптимальному решению можно сделать лишь в самых простых случаях. К примеру, так называемая *фундаментальная теорема генетических алгоритмов*, изложенная, например, в книге [10], дает некоторые вероятностные оценки на сходимость «классического» генетического алгоритма в применении к задаче угадывания строки-образца. Для большинства реальных задач, а также в случае более сложных реализаций генетических алгоритмов приходится довольствоваться лишь эвристическими соображениями.

Далее будут рассмотрены некоторые основные виды генетических алгоритмов и их компонент.

1.4.2. Классический генетический алгоритм

Классический алгоритм использует в качестве хромосом строки равной длины L над некоторым фиксированным алфавитом (обычно это двоичный алфавит $B = \{0, 1\}$). Функция приспособленности выбирается таким образом, чтобы ее значения были положительными, а большее значение функции соответствовало более оптимальному решению. На каждой итерации производятся следующие действия:

1. Формирование промежуточной популяции. Вероятность выбора особи в промежуточную популяцию пропорциональна ее значению приспособленности.
2. Скрещивание, или *кроссовер*. Из промежуточной популяции случайно выбирается две особи (пусть хромосома первой особи A равна $a_1a_2\ldots a_L$, а хромосома второй особи B равна $b_1b_2\ldots b_L$). Далее случайным образом выбирается индекс $k \in [1; L-1]$ и формируются две новые особи $AB = a_1\ldots a_k b_{k+1}\ldots b_L$ и $BA = b_1\ldots b_k a_{k+1}\ldots a_L$. Новые особи с некоторой вероятностью заменяют старые. Такой оператор скрещивания имеет название *одноточечный кроссовер*.
3. Мутация. Для каждой особи с некоторой вероятностью выполняется следующая операция. Вначале случайным образом выбирается индекс $k \in [1; L]$. Затем изменяется k-тый символ хромосомы (в случае двоичного алфавита на противоположный, в противном случае — на произвольный символ алфавита).

1.4.3. Операторы селекции

Для формирования промежуточной популяции, а также для выбора особей для скрещивания применяют различные операторы. Перечислим наиболее популярные из них:

- Метод «рулетки». Особь выбирается с вероятностью, пропорциональной ее значению приспособленности.
- Равномерный выбор. Любая особь выбирается равновероятно.
- Метод ранжирования. Особи сортируются в порядке улучшения значений приспособленности. Далее выбор особи происходит с вероятностью, увеличивающейся с увеличением индекса особи в отсортированной последовательности.
- Турнирный отбор. Для выбора особей, подлежащих скрещиванию, исходная популяция разбивается на группы небольшого размера m, в каждой из которых определяется лидер.

1.4.4. Операторы скрещивания

Для скрещивания хромосом, задаваемых строками, используют различные операторы скрещивания, например, такие:

- Одноточечный кроссовер. Его действие описано в разд. 1.4.2.
- Двухточечный кроссовер. Пусть даны хромосомы $A = a_1 \dots a_L$, $B = b_1 \dots b_L$. Случайным образом выбираются индексы i, j, такие что $1 \leq i < j < L$. Далее формируются две новые хромосомы $AB = a_1 \dots a_i b_{i+1} \dots b_j a_{j+1} \dots a_L$ и $BA = b_1 \dots b_i a_{i+1} \dots a_j b_{j+1} \dots b_L$.
- По аналогии с одноточечным и двухточечным кроссовером можно определить кроссовер для произвольного числа точек обмена.
- Однородный кроссовер. При формировании новых особей пара генов a_i, b_i случайным образом распределяется между потомками.
- Разновидность одно- или многоточечного кроссовера, применямая при работе с хромосомами переменной длины. Суть его заключается в том, что соответствующие индексы выбираются независимо для каждого из родителей.

1.4.5. Операторы мутации

Перечислим некоторые операторы мутации для битовых строк.

- Одиночная мутация. Сначала выбирается случайный индекс гена в хромосоме, затем значение этого гена с некоторой вероятностью инвертируется.
- Однородная мутация. Все гены хромосомы инвертируются равновероятно и независимо.
- Инвертирование интервала. Сначала выбираются два индекса $i, j : 1 \leq i \leq j \leq L$, затем с некоторой вероятностью все гены с индексами от i до j включительно инвертируются.
- Оператор инверсии. Случайно выбранная подстрока хромосомы разворачивается: $a_1 \dots a_{i-1} a_i a_{i+1} \dots a_{j-1} a_j a_{j+1} \dots a_n$ превращается в

$a_1 \dots a_{i-1} a_j a_{j-1} \dots a_{i+1} a_i a_{j+1} \dots a_n$.

1.4.6. Генетическое программирование

Особый вид генетических алгоритмов — *генетическое программирование* — использует в качестве хромосомы некоторое представление программы, например, синтаксическое дерево разбора программы [2] или конечный автомат [27]. Основы генетического программирования заложил Коза (Koza J. R.) в 1992 году. Структура хромосомы несет в себе специфику решаемой задачи и позволяет использовать проблемно-ориентированные операторы скрещивания и мутации, что позволяет сократить область поиска и ускорить нахождение оптимума.

1.5. Выводы по главе 1

Тестирование является одной из важнейших составляющих процесса разработки программного обеспечения. Повышение показателей автоматизации тестирования является одной из актуальных задач и способствует улучшению качества ПО и снижению стоимости его разработки.

Олимпиадное движение в области информатики и программирования является развитым движением в России и в мире, способствует выявлению талантливых и способных программистов. В силу своей специфики, процесс тестирования решений задач, предлагаемых на олимпиадах, подвержен глубокой автоматизации. Качество тестов является одним из главных показателей уровня проведения олимпиады. Подготовка тестов является творческим процессом, но для преодоления трудностей, связанных с составлением тестов против сложных эвристических решений, необходима разработка средств автоматизированного поиска таких тестов.

Задача о рюкзаке является известной, хорошо изученной и широко применяемой задачей комбинаторной оптимизации. В силу своей специфики, эта задача имеет множество решений, являющихся эффективными для подавляющего большинства входных данных. Однако в силу NP-полноты

задачи для любого решения можно построить такой набор входных данных, на котором это решение будет неэффективным. Поведение этих решений схоже с поведением решений олимпиадных задач по программированию, реализующих неэффективные алгоритмы с применением сложных эвристик. В силу вышесказанного, для проверки возможности генерации тестов, выявляющих неэффективные решения олимпиадных задач по программированию, в качестве олимпиадной задачи выбрана именно задача о рюкзаке.

Генетические алгоритмы являются современным семейством алгоритмов, решающих задачи оптимизации. Среди их преимуществ — возможность поиска оптимумов в задачах с дискретной областью определения функции и трудноопределимым локальным поведением. Предлагается использовать генетические алгоритмы для поиска тестов к олимпиадным задачам, максимизирующих время работы исследуемых решений.

Глава 2. Описание используемого подхода

2.1. Представление задачи о рюкзаке в виде олимпиадной задачи

Напомним постановку задачи о рюкзаке:

Дан рюкзак вместимостью W *и* n *предметов с весами* w_j *и стоимостями* p_j, $1 \leq j \leq n$. *Требуется минимизировать* $\sum\limits_{j=1}^{n} p_j x_j$ *при условиях* $\sum\limits_{j=1}^{n} w_j x_j \leq W$, $x_j \in \{0; 1\}$.

2.1.1. Выбор требуемого формата решения

Наиболее распространенным форматом решения является консольное приложение, читающее входные данные из файла и записывающее результаты вычислений также в файл. Этот подход имеет ряд преимуществ и недостатков. Преимущества его заключаются в следующем:

- Независимость логики решения задачи от логики тестирующей системы. Решение и проверяющая система должны лишь согласовать протоколы передачи данных, то есть форматы файлов.
- Нечувствительность тестирующей системы к критическим ошибкам времени выполнения, происходящим в решении. Такие ошибки, как нехватка памяти, или такие операции, как немедленное окончание работы приложения, не приводят к завершению работы проверяющей системы.
- Отсутствие зависимости работы решения от истории предыдущих запусков. Так как разным запускам решения соответствуют разные экземпляры программы, загруженные в память, то некорректное завершение одного из этих экземпляров или принудительное его завер-

шение проверяющей системой никак не повлияет на все остальные экземпляры, в том числе и еще не созданные.

Однако данный подход имеет и свои недостатки:

- Наличие дисковых операций ввода-вывода в цикле тестирования решения на тесте. Дисковые операции являются гораздо более медленными по сравнению с операциями, производимыми в оперативной памяти компьютера. Частично этот недостаток может быть скомпенсирован использованием каналов (pipes), хранящих свое содержимое в разделяемой области памяти.
- Наличие многократной конвертации представлений. Например, предмет в задаче о рюкзаке в памяти программы (как тестирующей, так и решающей) занимает не более 8 байт (два 32-битных целых числа), в то время как для записи в файл проверяющая программа должна преобразовать их в строковое представление, а для чтения из файла решение должно произвести обратную процедуру.
- Время, затрачиваемое на запуск решения как программы. Во время запуска программы операционная система производит большое число действий, требуемых для размещения программы в памяти. Эти действия в некоторых случаях могут занимать длительное время, например, при запуске решения на *Java*. Для небольших и/или простых тестов это время может существенно превышать время, затраченное на решение задачи.

В противоположность этому, можно рассмотреть другой формат решения, близкий к тому, что используется в *TopCoder* [14]. В этом случае в проверяющей системе имеется интерфейс, который должно реализовать любое решение. Этот интерфейс содержит метод, вызываемый проверяющей системой, в который в качестве аргументов передаются входные данные задачи, а возвращаемое значение является объектом класса, содержащим в себе ответ на задачу. Перечислим преимущества данного подхода:

- Отсутствие затрат на конвертацию данных. Входные данные пред-

ставлены (при надлежащем проектировании системы) в наиболее удобном для решения формате, требования к выходным данным также таковы, чтобы выполнение их приводило к минимальным дополнительным затратам времени и ресурсов.

- Отсутствие дополнительного времени на запуск приложения. Небольшие входные данные при таком подходе возможно решить за время порядка миллисекунды.
- Вся работа происходит в оперативной памяти, тем самым для выполнения одних и тех же операций потребуется меньше времени по сравнению с первым подходом.

Перечислим также и недостатки этого подхода:

- Решение может, случайно или преднамеренно, нарушить работу тестирующей системы, например, сделать системный вызов, приводящий к завершению приложения или войти в режим ожидания дисковой операции, приводя всю систему к зависанию. Наличие таких возможностей предполагает тщательный подход к проектированию проверяющих систем, работающих с решениями такого рода.
- Решение может поддерживать внутри себя определенное состояние. При непредвиденной ошибке времени выполнения или прерывании выполнения программы проверяющей системой решение может остаться в несогласованном состоянии, что помешает дальнейшим запускам этого решения на других наборах данных выполняться корректно. По тем же причинам возникают проблемы при параллельном тестировании.
- Решение зависит от проверяющей системы, поэтому при перепроектировании последней решения также должны быть изменены.

В условиях реального соревнования недостатки первого подхода не играют большой роли. Это связано с тем, что число тестов к задаче, как правило невелико, число участников также является сравнительно небольшим, в результате наличие затрат времени на побочную деятельность не

приводит к чрезмерной нагрузке на проверяющую систему. Недостатки же второго подхода устраняются различными методами.

При применении генетических алгоритмов возникает другая ситуация. Число тестов, на котором потребуется проверить решение, может быть велико по сравнению с соревнованием. В то же время, большая часть этих тестов потребует больше времени на работу с файлами или преобразование данных в строковый формат, увеличивая таким образом требуемое для выращивания поколения время в несколько раз.

В то же время, тесты, которые будут получены с использованием одного подхода, точно так же могут быть использованы в соревновании с тестирующей системой, использующей другой подход.

По описанным выше причинам, в работе реализован второй подход. В приложении 1 можно найти исходный код, соответствующий сущностям, характерным для задачи, и код интерфейса, который должны реализовывать решения.

Отметим еще два важных факта:

- Тесты, созданные с применением одного из подходов, могут с тем же успехом быть применены в системе соревнований, использующих другой подход. При этом, возможно, потребуется коррекция ограничений по времени и памяти.
- Не представляет особого труда реализовать адаптор решения, использующего первый подход, к тестирующей системе, использующей второй подход. Обратное же преобразование, как правило, повлечет за собой удвоение дополнительных расходов на чтение входных данных и запись выходных данных.

2.1.2. Выбор ограничений задачи

Современные методы решения задачи о рюкзаке [7] могут решать большие экземпляры задач за время, пропорциональное размеру ввода (порядка секунды для $N = 10^5$ на обыкновенном настольном компьютере).

Однако, лучшие из этих решений имеют оценку асимптотической сложности порядка $O(2^{\frac{N}{2}})$.

Большинство современных настольных компьютеров способны производить порядка 10^9 элементарных операций в секунду. Эксперименты с решениями задачи о рюкзаке показывают, что наихудшие экземпляры задачи о рюкзаке можно решить за разумное время при порядке ограничений $N \leq 50$. Даже при использовании сверхбыстрых многоядерных компьютеров или их кластеров по причине экспоненциального роста потребляемых ресурсов с увеличением N эта оценка не может быть сколь-нибудь существенно улучшена. Будем считать, что для генерируемых нами тестов будет выполняться ограничение $N \leq 50$, а при необходимости будем также уменьшать эту границу.

Также необходимо описать возможные значения таких величин из условия задачи, как вес и стоимость предмета, вместимость рюкзака. Для начала ограничимся *целочисленной* задачей о рюкзаке — будем считать веса и стоимости предметов, а также вместимость рюкзака, целыми величинами. Естественно считать веса и стоимости предметов положительными. Также будем считать, что $w_i, p_i \leq R = 10000$. Связано это с тем, что в большинстве решений неоднократно проводятся тесты верхних и нижних оценок, в которых участвуют квадраты величин порядка $N \cdot w_i$. Данные ограничения позволят реализовывать решения, используя встроенные целочисленные типы данных размерностью, не превышающей 64 бита. Кроме того, именно с такими ограничениями проводилась одна из серий тестирования решений в [7].

Так как решения запускаются в том же приложении, что и проверяющая система, проверка превышения предела используемой памяти затруднена. Поэтому ограничения на потребляемую память устанавливаться не будут.

В олимпиадных задачах ограничение по времени обычно находится в пределе от одной до десяти секунд. В нашей задаче мы будем варьировать

ограничение по времени в этих пределах в зависимости от поставленной задачи. С одной стороны, это время достаточно велико для того, чтобы при превышении данного ограничении на некотором тесте этот тест мог быть добавлен в окончательный набор тестов к задаче без дополнительного анализа устройства этого теста и построения эквивалентного ему теста большего размера. С другой стороны, это время достаточно мало для того, чтобы время подсчета функций приспособленности одного поколения в генетическом алгоритме, а следовательно и время выращивания требуемых тестов, находилось в разумных пределах.

2.2. Выбор представления теста в виде особи генетического алгоритма

2.2.1. Первоначальное упрощение

Тест для задачи о рюкзаке имеет один компонент, отличающийся от всего остального содержимого теста — вместимость рюкзака. Если каким-либо образом отказаться от того, чтобы явно хранить это число в особи генетического алгоритма, останется закодировать только набор предметов — однородную структуру, над которой легко производить различные операции, необходимые генетическому алгоритму.

Однако, в работах [7, 8] утверждается, что для большинства алгоритмов, использующих перебор и технику «ветвей и границ», наибольшую трудность представляют тесты, у которых вместимость рюкзака равна половине суммы весов всех предметов. Действительно, если вместимость составляет малую часть весов предметов или наоборот, почти ей равна, алгоритм может существенно сократить пространство перебора и уменьшить время работы.

В свете вышесказанного, предлагается кодировать в особи генетического алгоритма лишь набор предметов, а в качестве вместимости рюкзака брать половину суммы весов этих предметов.

2.2.2. О корреляции предметов и фундаментальной теореме генетических алгоритмов

К данной задаче можно применить вариацию классического генетического алгоритма, в которой вместо битовых строк рассматривались бы строки из предметов — пар чисел. Для классического генетического алгоритма утверждается ([10, 28]), что оптимальность решения достигается главным образом комбинированием схем небольшого порядка и длины. В применении же к задаче о рюкзаке такой подход кажется малоприемлемым — некоторые из типичных сложных тестов предполагают линейную зависимость стоимостей предметов от весов. В свете этого кажется целесообразным введение возможности группировать предметы, для того чтобы действие генетических операторов могло бы быть однородным относительно всех предметов одной группы.

2.2.3. Древовидные генераторы последовательностей

В данной работе особи генетического алгоритма имеют древовидную структуру. Каждая особь представляет собой подвешенное дерево, в котором каждой вершине соответствует некоторая последовательность предметов. Фактически, вершинами дерева являются объекты, которые обладают способностью генерировать последовательности предметов. Такие объекты называются *генераторы последовательностей*.

В листьях дерева находятся генераторы, генерирующие один, заранее заданный элемент. В узлах дерева находятся так называемые *композитные генераторы*. Каждый композитный генератор хранит в себе список генераторов, являющихся его потомками в дереве, и *операцию преобразования*, являющуюся функцией, изменяющей веса и/или стомости предметов. Пример генератора показан на рис. 2.1.

Последовательность, которую генерирует композитный генератор, описывается следующим образом:

- Конкатенируем все последовательности, сгенерированные

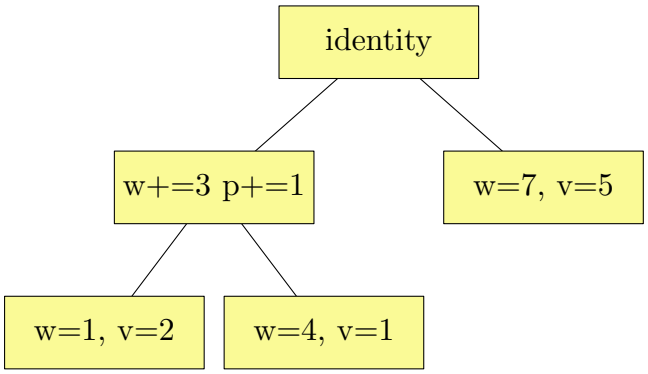

Рис. 2.1. Пример древовидного генератора

генераторами-потомками.

- Заменяем в получившейся последовательности каждый предмет предметом, полученным действием на него операцией преобразования.
- Полученная последовательность является результатом.

Набор предметов, соответствующий данной особи, состоит из предметов, входящих в последовательность, генерируемую корневым генератором этой особи.

Описанный подход напоминает генетическое программирование (см. разд. 1.4.6) с той особенностью, что значением функции в узле является не число, а последовательность предметов.

2.2.4. Операторы скрещивания и мутации

В качестве оператора скрещивания используется стандартный для генетического программирования оператор обмена поддеревьями. Выбор поддерева описывается следующим образом: если в данный момент алгоритм рассматривает лист дерева, то он и будет выбран в качестве поддерева. Если же алгоритм рассматривает узел дерева, то с вероятностью 0.5 выбирается поддерево с корнем в этом узле, иначе равновероятно выбирается один из потомков и процедура выбора продолжается. Оператор скрещивания проиллюстрирован на рис. 2.2.

Операторов мутации используется несколько. Первый из них заме-

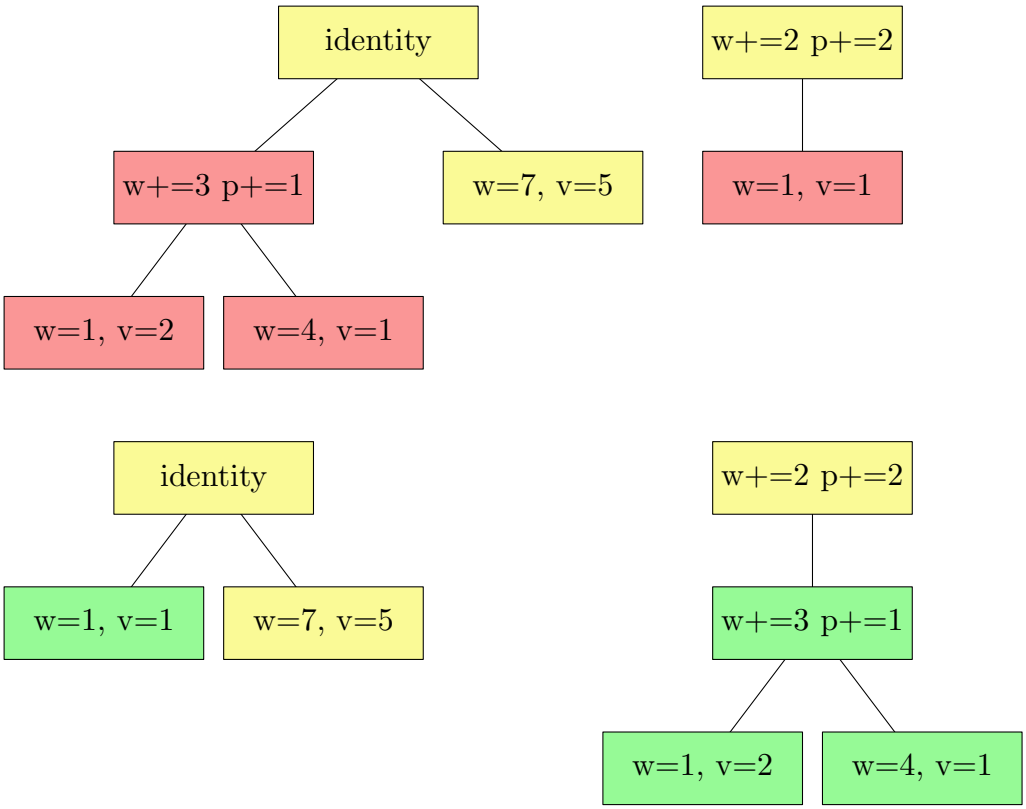

Рис. 2.2. Действие оператора скрещивания

няет выбранное поддерево на сгенерированное случайным образом дерево той же величины, как показано на рис. 2.3. Выбор поддерева производится тем же способом, как и в разд. 2.2.4.

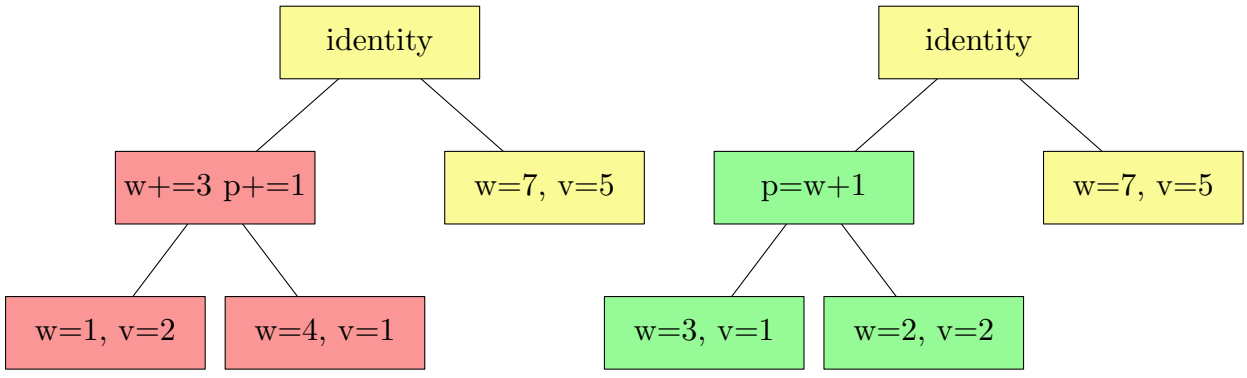

Рис. 2.3. Действие первого варианта оператора мутации

Второй оператор заменяет генератор одного предмета на генератор предмета, отличающегося от него на малую случайную величину. Композитный генератор он заменяет на другой композитный генератор с теми же потомками и операцией преобразования, равновероятно выбранной из

списка, приведенного в разд. 2.2.5. Пример приведен на рис. 2.4.

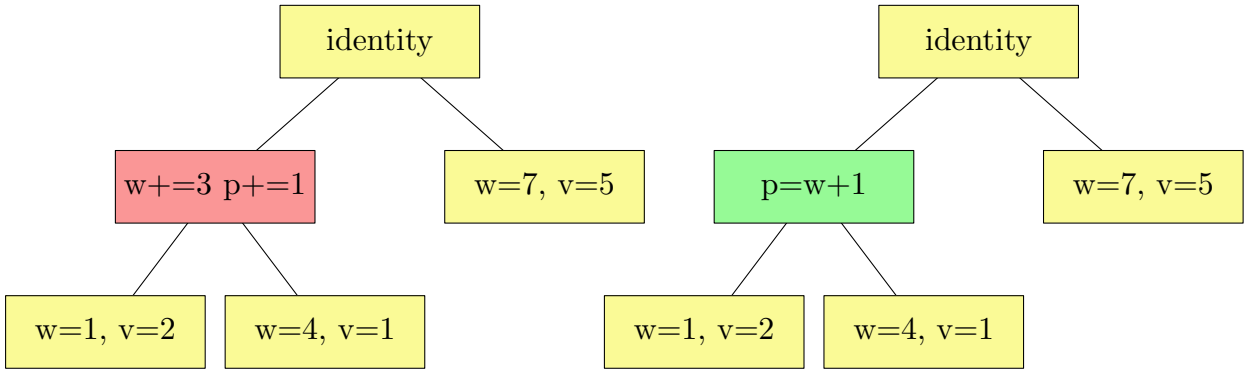

Рис. 2.4. Действие второго варианта оператора мутации

2.2.5. Операции преобразования

Используются следующие операции преобразования:

- Тождественное преобразование. Возвращает аргумент неизменным.
- Операция сдвига. Прибавляет к весу любого предмета число dW, а к стоимости — число dP. Оба этих числа — фиксированные параметры операции, которые выбираются независимым образом равновероятно из интервала $[-50; 50]$.
- Операция линеаризации. Оставляет неизменным вес w_i, делая стоимость равной $p_i = w_i + \beta$, где β — фиксированный параметр операции, выбираемый равновероятно из диапазона $[-50; 50]$.

2.2.6. Учет ограничений задачи

В процессе генерации последовательностей, а также выполнения генетических операторов, необходимо удовлетворять ограничениям задачи. В частности, вес и стоимость каждого предмета не должна выходить за рамки $[1; R]$, а число предметов не должно превышать N.

Первое из этих требований достигается путем возвращения значений веса и стоимости предметов в требуемые рамки после применения операции преобразования: $\tilde{p_i} = \min(R, \max(1, p_i))$, $\tilde{w_i} = \min(R, \max(1, w_i))$.

Выполнение ограничения на число предметов поддерживается следующим образом: после операции, влияющей на длину генерируемой последовательности (например, кроссовер), генератор подвергается преобразованию — *редуцированию* — после которого он генерирует только первые N элементов из старого списка. Делается это путем исключения соответствующих поддеревьев. Пример редуцирования показан на рис. 2.5.

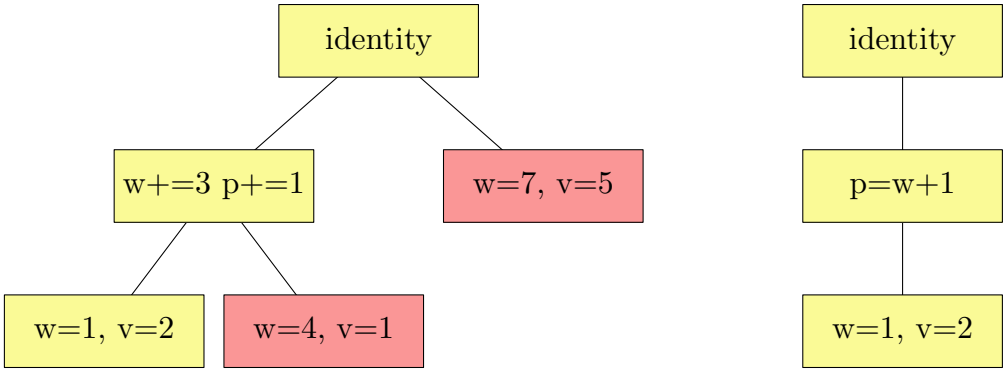

Рис. 2.5. Редуцирование генератора

Отметим, что существуют и иные способы выполнения ограничений. К примеру, особи, генерирующие не удовлетворяющую ограничениям последовательность, присваивается минимальное (нулевое) значение приспособленности. Этот подход имеет преимущества в тех случаях, когда иными способами поддерживать выполнимость ограничений затруднительно. Его недостаток заключается в том, что достаточно большое число потомков (до половины) могут иметь минимальную приспособленность: действительно, если выживают особи с числом предметов, близких к максимуму, то после кроссовера вероятность того, что у одного из потомков максимум будет превышен, очень велика.

2.2.7. Неизменяемость генераторов

Генераторы последовательностей выполнены неизменяемыми. Этот подход имеет ряд больших преимуществ перед изменяемым дизайном, среди них такие:

- Возможность кэширования результатов запросов по поддереву. В

частности, это позволяет зафиксировать для всей особи значение функции приспособленности, не вычисляя ее многократно.

- Разделение одного и того же поддерева между разными особями, в которые оно входит. Это позволяет экономить ресурсы памяти, требуемые для хранения поколения. Грамотное использование языка *Java* [29] со встроенным сборщиком мусора предотвращает от возможных утечек памяти при реализации таких структур.
- Непреднамеренное изменение объекта в случае ошибок в реализации генетических операторов невозможно, поэтому устраняются такие нежелательные эффекты, как «порча генофонда» из-за ошибок в программе, которые в противном случае очень трудно выявить и устранить.

В общих чертах, генераторы последовательностей следуют паттерну проектирования *Flyweight* [30].

2.3. Особенности примененного генетического алгоритма

2.3.1. Общее устройство генетического алгоритма

Примененный генетический алгоритм имеет стандартную схему. Число особей в одном поколении ограничено сотней особей. Алгоритм построения следующего поколения таков:

1. Число генерируемых потомков выбирается равным числу особей в текущем поколении.
2. Родители для генерации пары потомков выбираются методом турнирного отбора. При этом случайным образом выбирается восемь особей, затем по олимпийской системе происходит отбор. Из двух особей с вероятностью 0.9 побеждает сильнейший, с вероятностью 0.1 — слабейший. Затем оператор кроссовера создает двух потомков. Каждый из них с вероятностью 0.01 подвергается оператору мута-

ции. Оба потомка попадают во временный список, содержащий кандидатов на прохождение в следующее поколение. Операторы кроссовера и мутации описаны в части 2.2.4.

3. Предыдущий пункт повторяется столько раз, чтобы набралось необходимое число потомков.
4. Во временный список добавляется 25 случайно сгенерированных особей.
5. Во временный список добавляются все особи из текущего поколения.
6. В следующее поколение проходит не более 100 лучших особей из временного списка.

Алгоритм начинает работу с пустого поколения. Процесс генерации новых поколений продолжается до тех пор, пока не будет найден тест, на котором тестируемое решение не укладывается в отведенные ограничения по времени, или пока не закончится отведенное для процесса время.

2.3.2. Стагнация и меры, принимаемые для борьбы с ней

Стагнация — хорошо известное явление, признаком которого является остановка развития поколения. Чаще всего это происходит, когда алгоритм находит локальный оптимум, и все его особи сосредоточиваются в его окрестности. При этом теряется разнообразие генов в популяции, и вывести алгоритм из этого состояния бывает крайне трудно. Стагнации особенно подвержены генетические алгоритмы, использующие элементы стратегии элитизма [10], в том числе и используемый в данной работе.

Для предотвращения стагнации в данной работе используется следующий способ. Если в течение 50 поколений не происходит изменения максимума функции приспособленности по поколению, то поколение уничтожается и алгоритм начинает выращивать особи «с нуля». Однако, наилучшая особь из уничтоженного поколения сохраняется в особом списке. Когда в новых поколениях максимальное значение функции приспособленности превысит это значение у сохраненной особи, она будет исключена

из списка и добавлена в следующее поколение. Таким образом, лучшие результаты, достигнутые на неудачных циклах генетического алгоритма, сохраняются для того, чтобы быть полезными в дальнейшем, но придерживаются до того момента, когда они перестанут доминировать в новых поколениях. Практика показывает, что в некоторых случаях добавление оптимума из предыдущего цикла развития может спровоцировать лавинообразный рост функции приспособленности.

2.3.3. Функция приспособленности

Для выявления неэффективных решений кажется естественным в качестве функции приспособленности выбрать время, затраченное решением на данном тесте. Однако у этого подхода есть серьезные недостатки:

- Точность измерения времени оказывается недостаточной, чтобы отличить хорошие решения от плохих, особенно на начальном этапе.
- Измеряемая величина имеет свойство варьироваться в зависимости от различных условий. Даже если использовать системную функцию операционной системы, измеряющую время процессора, затраченное в определенном процессе, значения, измеренные в разные моменты времени, все равно отличаются. В табл. 2.1, 2.2 приведены результаты эксперимента по определению стабильности измерения времени работы некоторых процессов на различных операционных системах.

Таблица 2.1. Измерения времени работы процесса — ОС Linux, частота ядра 100 Гц

№ теста	Время исполнения теста, мс											
1	0	10	10	10	10	10	10	10	10	10	10	10
2	60	50	60	60	60	60	60	60	60	50	60	60
3	200	200	200	200	200	200	200	200	200	200	210	200
4	470	480	470	470	470	470	470	470	470	470	480	470
5	930	920	930	920	930	920	920	930	920	920	920	920
6	1630	1620	1640	1620	1610	1640	1600	1600	1600	1610	1620	1630

Описанные недостатки могут привести к выделению «ложного лидера» — ничем не выделяющаяся особь получит огромное преимущество

Таблица 2.2. Измерения времени работы процесса — ОС Windows Vista SP1

№ теста	Время исполнения теста, мс											
1	15	0	0	15	0	0	15	0	0	15	15	0
2	31	46	31	46	31	31	46	31	46	31	46	46
3	140	140	140	171	156	171	156	156	125	125	171	140
4	296	359	375	421	406	312	265	312	296	265	281	375
5	656	656	609	453	625	640	640	500	593	593	578	656
6	1125	968	1078	1093	1109	1125	1140	1140	1031	1125	1156	1093

над остальными, что приводит к стагнации уже на начальной стадии.

Для исправления этих недостатков предлагается использовать иной подход. Для этого тестируемые решения необходимо снабдить счетчиком «элементарных действий». В некоторых местах тестируемого решения этот счетчик необходимо увеличивать на величину, характеризующую количество операций, произведенных алгоритмом. Один из наиболее простых вариантов счетчика — подсчет числа вызовов определенной процедуры. Однако, счетчик может быть устроен гораздо сложнее. В конце работы решение возвращает значение этого счетчика вместе с ответом, а проверяющая система читает это значение и на его основе формирует значение функции приспособленности.

Хотя подход требует некоторой предварительной работы с тестируемыми решениями и с архитектурой проверяющей системы, преимущества, которые он дает, перекрывают его недостатки. Число операций, при должной реализации, сильно коррелирует с временем работы программы и практически является характеризующей его величиной. В то же время, оно не имеет флуктуаций во времени и имеет гораздо больший диапазон значений, что стабилизирует работу генетического алгоритма.

2.4. Выводы по главе 2

Для того чтобы работать с задачей о рюкзаке, как с олимпиадной задачей по программированию, выполнено сведение задачи о рюкзаке к олимпиадной задаче. Это сведение учитывает особенности тестирующей системы, применяемой при поиске тестов для неэффективных решений.

Описан алгоритм кодирования теста для задачи о рюкзаке в виде особи генетического алгоритма. При этом вместимость рюкзака поставлена в зависимость от остальных данных задачи, что упростило представление теста. Применение древовидной структуры особи, напоминающей особь генетического программирования, позволило сделать операторы скрещивания и мутации проблемно-ориентированными.

Решен вопрос о конкретном виде функции приспособленности особи. При этом устранены причины, приводящие к стагнации генетического алгоритма на ранних стадиях эволюции популяции.

Глава 3. Генерация тестовых данных для задачи о рюкзаке

3.1. Выбор алгоритмов решения задачи о рюкзаке для тестирования

При выборе алгоритмов для тестирования следует учитывать то, что время работы некоторых алгоритмов мало зависит от структуры входных данных и определяется главным образом характерными величинами теста. Таков, например, алгоритм, упомянутый в работе [23], время работы которого есть $\Omega(N \cdot W)$. Анализируя эффективность алгоритма такого класса для решения данной задачи, можно исходить из чисто асимптотических оценок на время работы. Так, вышеупомянутый алгоритм при работе на обычном настольном компьютере может считаться достаточно эффективным при ограничениях на данные $N \leq 100$, $W \leq 10^5$ и ограничении на время работы 2 секунды. В то же время, при ограничениях на данные $N \leq 2000$, $W \leq 10^6$ он уже не будет укладываться в ограничение по времени, если исходить из асимптотической оценки. При этом любой тест, достигающий ограничений на данные, заставит данное решение работать долго, и поиск таких тестов становится достаточно простой операцией.

По этой причине нецелесообразно генерировать тесты для алгоритма [23], как и для большинства решений с псевдополиномиальной асимптотической оценкой, генетическими алгоритмами — в случае подозрения на неэффективность такого алгоритма, тесты для него находятся традиционным путем.

Напротив, имеет смысл рассматривать алгоритмы, использующие технику «ветвей и границ», а также различные отсечения и эвристики. В случае подозрения на неэффективность такого алгоритма поиск тестов представляет собой нетривиальную задачу. Тесты, сгенерированные случайно, почти наверняка окажутся простыми, а при выборе шаблона для

генерации тестов эффект зачастую оказывается таким же, если не прибегать к глубокому теоретическому анализу свойств задачи и тестируемого алгоритма. Умение генерировать тесты против таких алгоритмов становится особенно ценным, если вспомнить, что похожими свойствами обладают «медленные» решения со встроенными эвристиками, посылаемые участниками олимпиад по программированию в надежде на то, что тесты окажутся недостаточно качественными и решение будет зачтено.

Исходя из описанных соображений, для тестирования выбраны следующие алгоритмы:

1. Простой алгоритм перебора с несколькими отсечениями ветвей.
2. Полная реализация алгоритма `EXPKNAP`, приведенного в [7].
3. Частичная реализация алгоритма `EXPKNAP`.
4. Корректная реализация алгоритма `HARDKNAP`, приведенного в [7].
5. Реализация алгоритма `HARDKNAP` с ошибкой в отсечении по нижней границе.

   Ниже кратко описан каждый из этих алгоритмов.

3.1.1. Простой алгоритм

Алгоритм предполагает, что предметы отсортированы в порядке невозрастания *эффективности* $e_j = p_j/w_j$. На каждом запуске основной процедуры перебора решается вопрос о том, взять ли предмет с номером i в рюкзак или отложить его. При этом используются три отсечения:

1. Если при прибавлении веса предмета к суммарному весу уже набранных предметов емкость рюкзака будет превышена, то не следует брать данный предмет.
2. Если все предметы с индексами $j \geq i$ помещаются в рюкзак на данном этапе, то не следует рассматривать случай, когда мы не берем предмет с индексом i.

3. Если при прибавлении стоимостей всех предметов с индексами $j \geq i$ к суммарной стоимости уже набранных предметов известное решение не сможет быть улучшено, то никакие случаи на данном этапе рассматривать не нужно.

3.1.2. Полная реализация алгоритма EXPKNAP

Алгоритм EXPKNAP является одним из алгоритмов решения задачи о рюкзаке, использующим концепцию *ядра* [7].

Алгоритмы такого типа предполагают, что предметы отсортированы в порядке невозрастания *эффективности* $e_j = p_j/w_j$. Рассмотрим «жадное» решение задачи, а именно, будем класть в рюкзак предметы, начиная с первого, до тех пор, пока предметы помещаются в рюкзаке. Первый предмет, не поместившийся в рюкзак, называется *разбивающим* (*break item*).

Было замечено [7], что для большинства входных данных оптимальное решение отличается от найденного таким образом жадного решения в небольшом числе позиций, причем все эти позиции располагаются близко к разбивающему предмету. Это дает возможность решить лишь небольшую подзадачу исходной задачи на предметах, находящихся в непосредственной близости от разбивающего. Такие предметы составляют *ядро* задачи. Различные алгоритмы по-разному определяют размер этого ядра, основываясь на эвристиках или сравнении верхних и нижних оценок на достижимое решение при некоторых ограничениях. В алгоритме EXPKNAP предлагается концепция *расширяемого ядра*. Также этот алгоритм использует *редукции*, дающие возможность зафиксировать некоторые предметы в определенном состоянии (взять в рюкзак или наоборот, не взять).

3.1.3. Частичная реализация алгоритма EXPKNAP

В частичной реализации не используются редукции, а также упрощено получение первоначальной нижней оценки. Кроме того, если полная реализация использует частичную сортировку предметов, тем самым для

простых данных имея оценку времени работы $O(N)$, то частичная реализация полностью сортирует предметы, тем самым работая на простых данных за $O(N \log N)$.

3.1.4. Корректная реализация алгоритма HARDKNAP

Алгоритм HARDKNAP [7] предназначен для эффективного решения сложных типов данных. Алгоритм построен по схеме «разделяй-и-властвуй»: задача делится на две подзадачи равного размера, которые в свою очередь также рекурсивно делятся на пары подзадач. Подзадачей является задача следующего вида: найти возможные пары $(w; p)$, представимые в виде:

$$w = \sum_{j=s}^{t} w_j x_j \qquad 1 \le s \le t \le n, 0 \le w \le W$$

$$p = \sum_{j=s}^{t} p_j x_j \qquad x_j \in \{0; 1\}, s \le j \le t$$

При слиянии результатов подзадач производятся оптимизации. Например, если (w_1, p_1) и (w_2, p_2) таковы, что $w_1 \le w_2$, а $p_1 \ge p_2$, то не имеет смысл хранить пару (w_2, p_2), так как она заведомо не лучше первой [7]. Кроме того, если верхняя оценка на решение для некоторой пары при текущих s и t не превосходит лучшего известного решения, то такую пару тоже можно исключить из рассмотрения.

Этот алгоритм имеет оценку времени работы $O(2^{\frac{N}{2}})$ в худшем случае, а для простых тестов работает за время порядка $O(N^2)$. Верхняя оценка теоретически и практически достижима, но тесты, на которых происходит достижение оценки, имеют веса и стоимости, растущие экспоненциально от N.

3.1.5. Реализация алгоритма HARDKNAP с ошибкой

Ошибка в данной реализации алгоритма заключается в том, что при оптимизации множества пар пара (w, p) исключается из рассмотре-

ния в том случае, когда верхняя оценка для этой пары *хуже* достигнутого решения, в то время, как в корректной реализации это происходит, когда верхняя оценка *не лучше* достигнутого решения. Эта ошибка не влияет на корректность ответа, но может серьезно повлиять на время работы программы за счет увеличения перебираемого множества.

Ошибки такого рода могут часто встречаться при реализации различных алгоритмов, в том числе и на олимпиадах по программированию. Последствия таких ошибок могут быть совершенно непредсказуемы.

3.2. Выбор тестовых данных

Для того, чтобы сделать выводы об эффективности применения генетических алгоритмов для генерации тестов, необходимо произвести поиск тестов не только с применением генетических алгоритмов, но также и традиционными методами.

В данной работе рассмотрены следующие традиционные методы генерации тестов:

- аналитическое построение тестов;
- генерация случайных тестов;
- генерация случайных тестов по шаблону.

3.2.1. Аналитическое построение тестов

Об аналитическом построение тестов можно прочитать в работе [8], посвященной тяжелым экземплярам задачи о рюкзаке. В этой статье рассматривается класс алгоритмов, названных *рекурсивными*. После анализа таких алгоритмов приводится набор свойств данных для задачи о рюкзаке, достаточных, чтобы любой рекурсивный алгоритм работал на этих данных достаточно долго. В частности, приводится элегантный способ построения сложных наборов данных, принадлежащий Тодду (Todd). Однако серьезным недостатком таких тестов с точки зрения данной работы заключается

в том, что величины весов и стоимостей растут экспоненциально с N. Ниже приведены характеристики предметов, получаемых способом Тодда:

$$p_j = w_j = 2^{K+N+1} + 2^{K+j} + 1, \text{ где } K = \lfloor \log_2 N \rfloor$$

Уже для сравнительно небольшого значения $N = 20$ значения, получаемые с помощью этих формул, превосходят $3 \cdot 10^7$. Таким образом, аналитически построенные тяжелые тесты не подходят для рассматриваемой задачи с изложенными выше ограничениями.

3.2.2. Генерация случайных тестов по шаблону

Генерация тестов по шаблону предполагает нахождение аналитическим путем шаблона, предположительно соответствующего тяжелым тестам, а затем создание и проверку на сложность множества конкретных тестов, соответствующих этому шаблону.

Для задачи о рюкзаке известно [7] несколько типичных шаблонов для создания тестовых данных. Эти шаблоны часто используются для демонстрации алгоритмов решения задачи о рюкзаке. Эти шаблоны таковы:

- Некоррелированные данные. Вес и стоимость предмета выбираются равновероятно из интервала $[1; R]$ независимо друг от друга и от других предметов.
- Слабо коррелированные данные. Вес предмета w_j выбирается равновероятно из интервала $[1 + \Delta; R - \Delta]$, а стоимость p_j выбирается равной $w_j + \delta$, где δ выбирается равновероятно из интервала $[-\Delta; \Delta]$. Величина Δ выбирается один раз для набора данных равновероятно из интервала $[1; 50]$.
- Сильно коррелированные данные. Вес предмета w_j выбирается равновероятно из интервала $[\max(1, 1 - \Delta); \min(R, R - \Delta)]$, а стоимость p_j выбирается равной $w_j + \Delta$. Величина Δ выбирается один раз для набора данных равновероятно из интервала $[-50; 50]$.
- Задачи о сумме подмножеств. Вес выбирается равновероятно из интервала $[1; R]$, стоимость полагается равной весу.

Во всех этих шаблонах вместимость рюкзака выбирается как половина суммы весов всех предметов.

Из вышеперечисленного, шаблон «некоррелированные данные» может быть рассмотрен как генерация чисто случайных тестов. Поэтому не имеет смысла выделять чисто случайные тесты в отдельный класс.

3.2.3. Тесты, генерируемые генетически

Необходимо также определиться с тем, какие именно тесты будут генерироваться с помощью генетических алгоритмов. Представляют интерес две разновидности тестов:

- Общие тесты. Никаких предположений о соотношении веса и стоимости предмета не делаются.
- Тесты для задачи о сумме подмножеств. Для получения таких тестов схема генерации особей генетического алгоритма изменяется очевидным образом.

3.3. Постановка эксперимента и анализ его результатов

3.3.1. Условия эксперимента

В эксперименте участвовали пять решений задачи о рюкзаке, перечисленных в разд. 3.1.1-3.1.5. Каждое из этих решение тестировалось на большом числе тестов каждого вида из представленных в разд. 3.2.2, а также подвергалось взаимодействию с двумя вариантами генетического алгоритма из разд. 3.2.3. Общее время воздействия каждого типа тестов на решение было ограничено по времени. Ограничение на значения весов и стоимостей R было зафиксировано ($R = 10000$), а ограничения на число предметов, на время выполнения и на время воздействия тестов варьировались, как указано в табл. 3.1.

Вычисления проводились на компьютере с параметрами, указанными в табл. 3.2.

Таблица 3.1. Ограничения

Максимальное N	Ограничение по времени	Время тестирования
25	2 сек	1 час
30	5 сек	2 часа
40	10 сек	3 часа
50	20 сек	4 часа

Таблица 3.2. Параметры используемого компьютера

Параметр	Значение
Тип процессора	Intel Core 2 Duo 6400
Тактовая частота процессора	2.13 ГГц
Объем оперативной памяти	1 Гб
Операционная система	Linux 2.6.22 (32 бит)

3.3.2. Часть результатов и выводы из них

В табл. 3.3 приведены некоторые результаты для $N = 25$. Из этих результатов видно, что для алгоритма `HARDKNAP`, как корректного, так и с ошибкой в реализации, генетический алгоритм добивается тех же результатов, что и генерация тестов традиционными способом. Для остальных же алгоритмов тесты, сгенерированные генетическим алгоритмом, превосходят по сложности тесты, найденные традиционным способом, иногда на порядок.

Таблица 3.3. Результаты, $N = 25$

Алгоритм	Некоррелированные		Коррелированные		Сумма подмножеств		Генетические	
	операций	мс	операций	мс	операций	мс	операций	мс
SimpleBranch	1911129	40	21849069	430	18041309	380	21841664	450
ExpKnapPart	1520	0	404877	20	694342	20	10950992	380
ExpKnap	1173	0	456577	20	285325	20	11732142	570
HardKnapPart	17255	0	100924	0	101081	10	100891	10
HardKnap	3885	0	258502	20	262104	10	255065	20

Тест, полученный генетическим алгоритмом для алгоритма `EXPKNAP`, имеет $N = 25$, $W = 58129$ и распределение весов, указанное в табл. 3.4. Из его рассмотрения видно, что в тесте содержится много одинаковых предметов. Кроме того, этот тест удовлетворяет трем из четырех свойств трудных тестов, изложенных в работе [8].

Для проверки гипотезы о том, что большое число совпадающих предметов может сделать тест труднее для некоторых алгоритмов, в пе-

Таблица 3.4. Тест, полученный генетическим алгоритмом

Вес	Стоимость	Число предметов	Число предметов в оптимальном решении
4704	4705	10	5
3153	3154	7	2
3193	3194	3	2
7020	7021	2	1
8635	8636	1	0
5955	5956	1	1
8939	8940	1	1

речень тестовых данных был добавлен еще один тип тестов:

- Тесты с малым разнообразием предметов. Выбираются два предмета с весами и стоимостями, распределенными равномерно в допустимых пределах. Для первого предмета в тест добавляется количество его копий, равномерно выбираемое из диапазона $[1; N-1]$, остальные предметы делаются копиями второго предмета.

Отметим, что данный тип тестов к задаче о рюкзаке достаточно вероятен в практических приложениях задачи о рюкзаке. Действительно, такие тестовые данные могут получаться, если поставлена задача загрузить, например, корабль, большим числом контейнеров двух типов.

3.3.3. Основные результаты

В табл. 3.5 и 3.6 приведены результаты генерации тестов. Из них следует, что для алгоритмов семейства `HARDKNAP` генетический алгоритм генерирует тесты того же порядка сложности, что и лучшие из тестов, сгенерированных традиционным способом. Что же касается остальных рассматриваемых алгоритмов, для них самыми сложными являются тесты, сгенерированные по новому шаблону (малое разнообразие предметов). Генетический алгоритм генерирует тесты примерно такого же порядка сложности, а все остальные шаблоны тестов генерируют существенно более простые тесты.

При рассмотрении отдельно задачи о сумме подмножеств (два тестирующих запуска: традиционный и генетический) наблюдается уверенное превосходство подхода к генерации тестов с применением генетического

Таблица 3.5. Сводная таблица результатов тестирования (значения приспособленности)

Тип тестов	SimpleBranch	ExpKnapPart	ExpKnap	HardKnapPart	HardKnap
N = 25					
Некоррелированные	1911129	1520	1173	17255	3885
Коррелированные	21849069	404877	456577	100924	258502
Сумма подмножеств	18041309	694342	285325	101081	262104
Малое разнообразие	33308926	16777217	17132585	1276	2260
Генетические общие	21841664	10950992	11732142	100891	255065
Ген., сумма подмн-в	29360131	13358856	12665753	101130	266112
N = 30					
Некоррелированные	18955091	2806	1997	51474	4404
Коррелированные	TL	2651394	2278654	599896	1354040
Сумма подмножеств	TL	426786	340007	597063	1392054
Малое разнообразие	TL	TL	TL	1857	3095
Генетические общие	TL	TL	TL	595956	1234443
Ген., сумма подмн-в	TL	TL	TL	602433	1416982
N = 40					
Некоррелированные	TL	3473	3353	130719	7339
Коррелированные	TL	19739109	73428000	15677670	6295984
Сумма подмножеств	TL	615357	778820	16250915	188499
Малое разнообразие	TL	TL	TL	4333	6721
Генетические общие	TL	182376230	TL	12157417	4468909
Ген., сумма подмн-в	TL	TL	TL	16187562	4615618
N = 50					
Некоррелированные	TL	5887	4079	306605	10775
Коррелированные	TL	175820961	193782996	TL	23018237
Сумма подмножеств	TL	439628	1382559	TL	2551084
Малое разнообразие	TL	TL	TL	7441	10751
Генетические общие	TL	249482851	259466527	147939527	16405923
Ген., сумма подмн-в	TL	232480165	TL	TL	2930031

Примечание. Как TL отмечены те тестовые запуски, на которых ограничение по времени было превышено на найденном тесте, и следовательно, число операций не определено.

алгоритма над традиционным подходом.

Можно, однако, выделить два недостатка метода, предложенного в данной работе. Первый из них — некоторая нестабильность работы генетического алгоритма, выражающаяся в том, что окончательный результат при фиксированном времени генерации сильно варьируется от запуска к запуску. Эта черта свойственна генетическим алгоритмам в целом как эвристическим алгоритмам, использующим генератор случайных чисел.

Второй недостаток метода — при увеличении размера задачи (в данном случае резмером задачи считается максимальное число предметов) алгоритму становится труднее находить оптимум. Этому можно дать двоякое объяснение. Во-первых, это еще одна особенность оптимизационных алгоритмов вообще — доля входных данных, близких к оптимуму, с увеличе-

Таблица 3.6. Сводная таблица результатов тестирования (время работы на тесте, мс)

Тип тестов	SimpleBranch	ExpKnapPart	ExpKnap	HardKnapPart	HardKnap
N = 25					
Некоррелированные	40	0	0	0	0
Коррелированные	430	20	20	0	20
Сумма подмножеств	380	20	20	10	10
Малое разнообразие	660	540	820	0	0
Генетические общие	450	380	570	10	20
Ген., сумма подмн-в	580	450	620	10	20
N = 30					
Некоррелированные	410	0	0	0	0
Коррелированные	TL	120	110	50	100
Сумма подмножеств	TL	20	20	50	100
Малое разнообразие	TL	TL	TL	0	0
Генетические общие	TL	TL	TL	50	90
Ген., сумма подмн-в	TL	TL	TL	50	110
N = 40					
Некоррелированные	TL	0	0	10	0
Коррелированные	TL	890	3370	1470	540
Сумма подмножеств	TL	20	40	1510	10
Малое разнообразие	TL	TL	TL	0	0
Генетические общие	TL	7530	TL	1230	430
Ген., сумма подмн-в	TL	TL	TL	1650	390
N = 50					
Некоррелированные	TL	0	0	30	0
Коррелированные	TL	7820	8920	TL	1950
Сумма подмножеств	TL	20	70	TL	180
Малое разнообразие	TL	TL	TL	0	0
Генетические общие	TL	11000	11450	13340	1390
Ген., сумма подмн-в	TL	7910	TL	TL	220

Примечание. Как TL отмечены те тестовые запуски, на которых ограничение по времени было превышено на найденном тесте, и следовательно, время выполнения не определено.

нием размеров задачи существенно снижается. Во-вторых, время, затрачиваемое на подсчет функции приспособленности у поколения, растет для данной задачи экспоненциально. Этот эффект почти не затрагивает генераторы случайных тестов по шаблону, так как доля сложных тестов среди них в целом мала, но существенно влияет на генетический алгоритм, так как рост приспособленности поколения приводит к увеличению времени, требуемого для расчетов.

Несмотря на указанные недостатки, по результатам эксперимента генетический алгоритм показал себя с хорошей стороны. Для сравнительно эффективных решений тесты, генерируемые им, имеют тот же порядок сложности, что и тесты, сгенерированные традиционным методом, в то время как для сравнительно неэффективных решений он находит гораздо

более сложные тесты.

3.4. Выводы по главе 3

Для проверки эффективности подхода, описываемого в данной работе, были выбраны алгоритмы решения задачи о рюкзаке, подлежащие тестированию. При этом учтена зависимость того или иного решения от внутренней структуры теста.

В качестве сравнения с применяемым подходом, были выбраны некоторые способы генерации тестов, соответствующие традиционному подходу. Для эксперимента было отобрано три способа генерации тестов с применением традиционного подхода и два способа с применением генетического алгоритма.

Первоначальные испытания показали, что для некоторых алгоритмов генетический алгоритм сгенерировал тесты, по качеству многократно превосходящие тесты, созданные традиционными методами. Анализ этих тестов показал, что имеет смысл выделить новый класс тестов для задачи о рюкзаке. Новый класс тестов был добавлен к имеющимся способам генерации тестов.

Дальнейшие испытания показали целесообразность подхода с использованием генетических алгоритмов. Кроме того, новый класс тестов показал себя крайне сложным для некоторых из тестируемых алгоритмов.

Глава 4. Генерация тестов для олимпиадной задачи

4.1. Описание олимпиадной задачи

Для апробации описываемого подхода в условиях реальных олимпиадных задач была взята задача «Ships. Version 2». С текстом условия этой задачи можно ознакомиться на сайте *Timus Online Judge* [31], где она размещена под номером 1394 [11].

Более формальная версия условия задачи звучит следующим образом. Дано N ($2 \leq N \leq 99$) предметов с весами w_i, $1 \leq w_i \leq 100$. Также дано M ($2 \leq M \leq 9$) рюкзаков с вместимостями $c_j \geq 1$. Известно, что $\sum_{i=1}^{N} w_i = \sum_{j=1}^{M} c_i$. Требуется поместить все данные предметы в данные рюкзаки. Гарантируется, что искомое размещение существует. Ограничение по времени — 1 секунда, ограничение по памяти — 16 мегабайт.

Из изложенного следует, что рассматриваемая задача является частным случаем *задачи о мультирюкзаке* [7]. В терминах задачи о мультирюкзаке, дополнительные ограничения состоят в том, что для всех предметов их вес равен стоимости, и все рюкзаки должны быть заполнены.

Данная задача NP-полна, так как она принадлежит классу NP и к ней сводится NP-полная задача о сумме подмножеств. Кроме того, задача о мультирюкзаке является NP-трудной *в сильном смысле*, то есть для нее неизвестны решения, работающие с полиномиальной оценкой от размерности задачи и ограничений на веса и стоимости предметов. Вполне естественно ожидать от рассматриваемой задачи такого же свойства. Из этого следует, что при данных ограничениях задачи маловероятно существование решения, которое укладывалось бы в ограничения по времени и памяти на всех возможных тестах. Однако, существует большое число различных эвристических решений, для которых трудно составить такой тест.

Впервые эта задача была предложена на сборах болгарской команды школьников, а позже была выложена на [32]. Набор тестов, прилагаемый к этой задаче, оказался достаточно слабым, и через некоторое время задача с тем же условием и усиленным набором тестов была выложена под другим номером [11]. Некоторое время после этого тесты, генерируемые против некоторых из решений, добавлялись в этот набор. По состоянию на 15 июня 2009 года, в наборе присутствовало 47 тестов, из 3100 посланных на проверку решений было принято 260.

Исследования, описанные в данной главе, имеют целью сгенерировать новые тесты, чтобы максимально возможное число принятых решений не прошло хотя бы один из таких тестов.

4.2. Описание генетического алгоритма

4.2.1. Общая схема алгоритма

Генетический алгоритм, примененный в данной части, незначительно отличается от алгоритма, описанного в разд. 2. Ниже перечислены особенности описываемой версии генетического алгоритма:

- число особей в поколении выбирается в пределах от 300 до 500;
- ограничение на время работы решения на одном тесте выбирается в пределах от двух до 40 секунд (подробнее в разд. 4.3.1);
- ограничение на время работы генетического алгоритма не ставятся, решение о прерывании работы алгоритма принимает пользователь.

4.2.2. Представление теста в виде особи генетического алгоритма

Для генерации тестов для рассматриваемой задачи предлагается сначала представить тест в виде последовательности чисел, а затем полученную последовательность закодировать древовидным генератором последовательности, описанным в разд. 2.2.3.

4.2.2.1. Представление теста в виде последовательности чисел

Для того чтобы генетический алгоритм работал наиболее эффективно, необходимо сузить пространство поиска алгоритма. В частности, целесообразно выразить некоторые данные, присутствующие в тесте, через другие такие данные.

В условии задачи сказано, что сумма весов предметов должна равняться сумме вместимостей рюкзаков, и должен существовать способ разместить все предметы по рюкзакам. Поэтому целесообразно закодировать тестовые данные таким образом, чтобы эти условия выполнялись «по построению» и нужда в проверке этих условий отпала.

Предлагается следующий способ построения теста по последовательности чисел. Будем считать, что последовательность состоит из целых чисел из диапазона $[0; 100]$. При этом положительным числам соответствуют предметы с весом, равным соответствующему числу. Нули же, встречающиеся в этой последовательности, являются разделителями последовательности на группы положительных чисел. Каждой такой группе сопоставлен рюкзак с вместимостью, равной сумме чисел этой группы. На рис. 4.1 проиллюстрирован пример последовательности и соответствующего ей теста.

Веса предметов: $3, 7, 5, 1, 2, 4, 7, 6, 1, 3, 8$

Последовательность: $0, \underbrace{3, 7, 5}_{15}, 0, \underbrace{1, 2}_{3}, 0, \underbrace{4, 7, 6}_{17}, 0, 0, \underbrace{1, 3, 8}_{12}$

Вместимости рюкзаков: $15, 3, 17, 12$

Рис. 4.1. Числовая последовательность и генерируемый ей тест

Если не учитывать порядок следования весов предметов и вместимостей рюкзаков в тесте, то любой тест, удовлетворяющий ограничениям, может быть закодирован в виде последовательности чисел. Хотя для любого теста существует бесконечно много кодирующих его последовательностей, среди них всегда найдется последовательность длиной $N + M - 1$.

В противоположность этому, не любая возможная последовательность генерирует тест, удовлетворяющий ограничениям. А именно, число предметов и число рюкзаков могут быть как слишком малыми, так и слишком большими. Однако, требование о равенстве сумм весов предметов и сумм вместимостей рюкзаков, а также требование о существовании ответа, всегда удовлетворены. В связи с этим, предлагается отбраковывать последовательности, генерирующие тесты, которые не удовлетворяют ограничениям. Наиболее простым способом сделать это является обнуление значения приспособленности у особи, соответствующей «плохой» последовательности.

Среди преимуществ данного подхода — однородность данных в последовательности. А именно, структура последовательности (в частности, способность производить тесты) не изменяется при удалении любого элемента последовательности или добавлении элемента в произвольное место последовательности. Это существенно упрощает проектирование генетических операторов.

4.2.2.2. Кодирование последовательности чисел древовидным генератором последовательности

Для того, чтобы иметь возможность группировать блоки элементов последовательности, предлагается использовать для кодирования числовой последовательности в генетическом алгоритме хорошо себя зарекомендовавшие древовидные генераторы последовательностей, описанные в разд. 2.2.3. В листьях генератора хранятся целые числа, в узлах — тождественные преобразования. Пример генератора приведен на рис. 4.2.

4.2.3. Функция приспособленности

Как и в разд. 2.3.3, в качестве функции приспособленности выбрано число, приближенно характеризующее число операций, выполненных тестируемом алгоритмом на данном тесте.

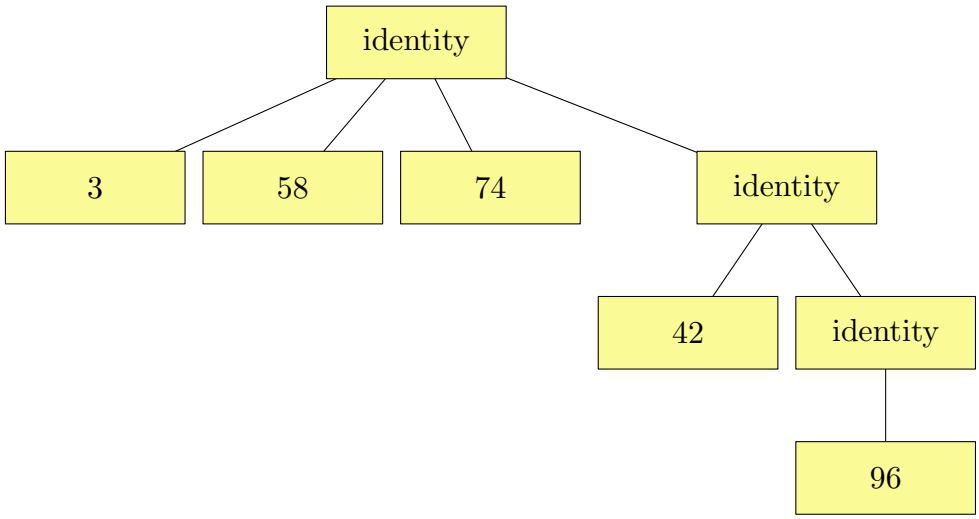

Рис. 4.2. Пример древовидного генератора

Вопрос о том, что считать функцией приспособленности, принимается индивидуально для каждого тестируемого решения. Однако, можно выделить общие закономерности.

В качестве функции приспособленности можно выбрать число раз, когда были выполнены «узкие места» алгоритма. При таком подходе счетчик числа операций инкрементируется внутри самого глубокого уровня вложенности некоторого числа циклов, встречающихся в программе. Итоговое значение счетчика в этом случае почти точно пропорционально времени работы программы. Далее будем ссылаться на этот подход выбора функции приспособленности как на *прямой подсчет*.

Для рекурсивных алгоритмов решения в качестве функции приспособленности можно выбрать число вызовов рекурсивной функции (или одной из таких функций в случае, если их несколько) в течение времени работы алгоритма. Этот подход далее будет именоваться *подсчетом вызовов*.

Часто также встречаются программы, запускающие один и тот же алгоритм на разных вариантах упорядочения входных данных. Так, многие эвристические алгоритмы перебирают случайные перестановки предметов и/или рюкзаков и для каждой такой перестановки пытаются произвести поиск решения. В таких случаях в качестве функции приспособленности

выбирать число таких запусков в течение времени работы алгоритма. Этот подход будет далее именоваться *подсчетом запусков*.

В некоторых случаях целесообразно использовать комбинации описанных выше подходов.

4.3. Сложности в реализации подхода и методы борьбы с ними

4.3.1. Применение избыточных ограничений

На время работы решений, и в частности на то, укладывается ли конкретное решение в ограничение по времени, существенно влияет конфигурация сервера, на котором происходит тестирование, а также версии и настроек компилятора или интерпретатора используемого языка программирования. Кроме того, всегда существует возможность улучшения комплектации сервера или обновления программного обеспечения. Исходя из этих соображений, при генерации теста следует добиваться превышения не времени, указанного как ограничение в условии задачи, а несколько большего времени. При работе с различными решениями использовались ограничения по времени от двух до 40 секунд, в зависимости от стабильности времени их работы.

4.3.2. Портирование решений во внутрисистемный формат

Как описано в разд. 2.1.1, решения могут быть представлены в нескольких форматах, включающих в себя консольное приложение и реализацию интерфейса на языке *Java*. Так как для этой задачи решения уже были реализованы в виде отдельных консольных приложений на языках *Pascal* и *C++*, выбор второго из приведенных форматов автоматически влечет за собой дополнительные расходы на портирование существующих решений на язык *Java*, в то время как использование решений в близком к исходному виде влечет дополнительные затраты ресурсов системы на создание большого числа приложений за короткий период времени.

В описанных условиях выбор между этими вариантами, вообще говоря, не может быть обусловлен чисто теоретическими соображениями. В работе использованы оба описанных подхода.

4.3.3. Рандомизированные решения

Генетический алгоритм, используемый в данной работе, сохраняет значение приспособленности, вычисленное один раз для особи, в течение жизни всей особи. Этот подход позволяет сэкономить время, проведенное в подсчете функции приспособленности. В то же время, при этом предполагается, что значение приспособленности не зависит от времени, в которое производится его вычисление.

Однако, это предположение неверно, если тестируемое решение использует генератор случайных чисел, инициализируемый системным таймером (или какой-либо другой функцией времени). В этом случае число операций, выполненных алгоритмом, а значит и значение приспособленности, сильно зависит от времени запуска решения. Такое поведение может дестабилизировать работу генетического алгоритма. Кроме того, любой тест оказывается «трудным» с некоторой вероятностью, которая может быть мала или велика.

Не существует универсального способа решить эту проблему. В данной работе используется следующий метод: генератор случайных чисел инициализируется фиксированным числом вместо системного таймера. Тест считается сложным, если время работы решения на нем в несколько раз больше ограничения по времени (например, в десять раз). Такое допущение имеет смысл, если алгоритм состоит из нескольких итераций, каждая из которых инициализируется с помощью генератора случайных чисел и способна выдать ответ в случае удачной инициализации. В качестве функции приспособленности в этом случае целесообразно выбирать «подсчет запусков». Если значение приспособленности для некоторого генератора случайных чисел велико, то вероятность удачного запуска мала,

и для произвольного генератора случайных чисел вероятность удачного запуска будет также мала.

Практика показывает, что описанный метод является достаточно эффективным для практически реализуемых решений.

4.3.4. Отсечения по времени

Некоторые решения основаны на следующем подходе. Используется два и более различных алгоритмов решения, но каждому из них дается внутреннее ограничение по времени. Так, если первый из алгоритмов не нашел решения за отведенное ему время, то он прекращает работу, и решение запускает следующий алгоритм, и так далее. Такие решения достаточно сложны для генерации тестов против них. Кроме того, время их работы гораздо сильнее зависит от конфигурации тестирующего сервера, чем у иных решений. Связано это с тем, что для некоторых тестов лишь один алгоритм из представленных в таком решении может за время, близкое к отведенному, найти правильный ответ, и на одном компьютере он успевает это сделать, а на другом — нет.

Предлагается следующий подход к генерации тестов против таких решений. Для каждого из вложенных алгоритмов время, отведенное на их работу, увеличивается в несколько (от двух до десяти) раз, и общее ограничение по времени увеличивается соответствующим образом. Это стабилизирует поведение решений при тестировании на практике. Кроме того, тест, генерируемый против такого решения, оказывается трудным сразу для большого количества вариантов этого решения, варьирующих ограничения по времени для вложенных алгоритмов, в том числе для всех таких решений при таком выборе внутренних ограничений, когда каждое из них превосходит общее ограничение по времени.

4.4. Описание и результаты эксперимента

4.4.1. Описание эксперимента

Целью эксперимента было сгенерировать такие тесты, чтобы максимально возможное число решений, прошедших уже имеющиеся тесты, не прошли хотя бы один из таких тестов. Для этого из имеющихся на сервере зачтенных решений было выбрано 25 решений. Некоторые из них были выбраны для генерации тестов против них, остальные решения использовались для оценки эффективности полученных тестов. По мере генерации тесты отправлялись на проверяющий сервер, где производилось перетестирование имеющихся решений.

Каждый тест генерировался против какого-то одного решения. Для этого в решении производились небольшие изменения, необходимость и содержание которых описаны в разд. 4.2.3 и 4.3. Исходя из специфики решения, определялось ограничение по времени работы на тесте. Далее производился запуск генетического алгоритма. Он останавливается либо автоматически, в случае превышения решением времени работы на некотором сгенерированном тесте, либо вручную, по команде пользователя. Время работы генетического алгоритма варьировалось от одного-двух часов до суток.

Всего было сгенерировано 28 тестов. Три из них генерировались на основе трех решений, показавших на ранее существовавших тестах наихудшее время работы. Эти решения были портированы на язык *Java*, как изложено в разд. 4.3.2. Остальные тесты генерировались против семи решений (против каждого было сгенерировано несколько тестов), причем эти решения были оставлены в формате консольного приложения и изменены минимальным образом.

Помимо этого, был проведен эксперимент по генерации случайных тестов и запуске на них одного из предоставленных решений с наихудшим временем работы на тестах. Этот эксперимент ставил целью сравнение тра-

диционного и генетического подходов при генерации тестов. Процесс проходил в течение 26 часов.

4.4.2. Результаты эксперимента

В результате перетестирования решений из имевшихся на момент начала тестирования принятых решений ни одно не прошло сгенерированный набор тестов.

По результатам тестирования среди сгенерированных тестов было отобрано 11 лучших. Они получили номера с 48 по 58 в порядке уменьшения сложности. В табл. 4.1 для каждого теста показано, сколько решений не проходят этот тест, но проходят все предыдущие (далее такой тест будем называть «первый трудный тест»).

Таблица 4.1. Число решений, для которых тест является первым трудным

Номер теста	48	49	50	51	52	53	54	55	56	57	58
Число решений	217	15	8	4	0	1	5	2	2	4	1

В табл. 4.2 приведены данные по времени работы для одного из решений на сгенерированных тестах.

Сравнительный эксперимент со случайными тестами показал следующие результаты. За 26 часов, в течение которых проходил эксперимент, было проверено 17921000 тестов. Максимальное время работы решения на каком-либо из этих тестов составляет 382 миллисекунды, что более чем вдвое меньше ограничения по времени. Данное сравнение наглядно показывает преимущество подхода с использованием генетических алгоритмов перед традиционным подходом.

4.5. Выводы по главе 4

Подход, описанный в разд. 2 и испытанный в разд. 3 на примере задачи о рюкзаке, применен к реальной олимпиадной задаче [11] с формулировкой, близкой к задаче о рюкзаке (фактически, частный случай задачи о мультирюкзаке). Из анализа сложности задачи вытекает то, что при огра-

Таблица 4.2. Время работы решения на тестах, мс

Номер теста	48	49	50	51	52	53	54	55	56	57	58
937951	234	234	250	234	218	15	15	15	46	15	TL
1048182	156	156	156	156	156	265	TL	125	78	15	328
1053557	187	171	187	156	171	31	15	TL	140	15	15
1243136	TL	343	TL	687	328	15	15	296	593	15	15
1289928	WA	WA	TL	718	WA	15	15	265	125	15	15
1290261	TL	265	TL	TL	TL	15	15	296	312	15	15
1322878	TL	TL	TL	TL	TL	15	15	TL	78	15	15
1328152	TL	TL	TL	TL	TL	140	31	750	TL	15	TL
1394345	TL	TL	TL	TL	TL	15	15	TL	31	15	46
1682998	TL	TL	TL	TL	TL	15	15	TL	46	15	453
1700721	TL	TL	234	TL	TL	15	15	125	TL	15	15
1841778	TL	TL	TL	359	TL	15	15	281	109	15	31
2004976	TL	TL	TL	TL	TL	62	15	TL	TL	15	671
2006419	TL	TL	TL	TL	TL	156	15	TL	TL	15	93
2052162	TL	TL	TL	TL	TL	593	15	TL	156	15	TL
2072685	437	312	437	312	312	328	328	328	281	TL	15
2072705	390	390	390	375	375	281	281	281	265	TL	15
2141473	421	TL	765	TL	296	15	15	171	TL	15	15
2208365	TL	TL	TL	TL	TL	15	15	TL	TL	15	15
2293082	TL	1000	TL	TL	875	TL	TL	TL	484	15	218
2293085	TL	937	TL	TL	812	984	TL	TL	375	15	125
2354409	TL	TL	TL	TL	TL	109	31	TL	78	15	187
2438565	TL	TL	TL	218	TL	15	15	TL	31	15	31
2526559	TL	TL	TL	TL	TL	15	15	TL	TL	15	15
2558302	TL	TL	TL	TL	TL	15	15	TL	TL	15	93
2584211	WA	WA	WA	WA	WA	125	125	WA	125	15	WA
2616776	TL	TL	TL	TL	TL	15	15	TL	TL	15	62
2616813	TL	TL	TL	TL	TL	15	15	TL	TL	15	31
2628204	609	265	TL	TL	TL	515	453	TL	TL	15	15
2628278	TL	TL	TL	TL	TL	15	15	TL	TL	15	15
2633193	TL	937	TL	TL	812	984	TL	TL	390	15	125
2640878	203	937	375	TL	TL	15	15	109	109	15	15

Примечание. Отметка TL означает, что решение превысило ограничение по времени на тесте. Отметка WA означает, что решение дало неверный ответ на тесте.

ничениях, данных в условии задачи, ее маловероятно решить, уложившись в ограничения. Тем не менее, в тестирующей системе находилось более 250 решений, прошедших все имеющиеся на тот момент тесты.

С помощью генетического алгоритма удалось сгенерировать набор тестов, на котором все решения, принятые по результатам проверки на предыдущих тестах, получили отрицательный вердикт. Эти тесты добавлены в набор, тем самым качество тестов к этой задаче существенно повысилось.

Заключение

В работе предложен метод генерации тестов к олимпиадным задачам с использованием генетических алгоритмов. Произведено сравнение тестов, генерируемых данным методом, с тестами, генерируемыми традиционными методами. Сравнение показало эффективность предложенного метода по сравнению с традиционными методами.

При анализе результатов генерации тестов выделен новый класс входных данных для задачи о рюкзаке, являющийся сложным для некоторых алгоритмов решения этой задачи. Данный класс был построен на базе тестов, генерируемых генетическим алгоритмом.

Предложенным методом были также сгенерированы новые тесты к реальной олимпиадной задаче. По результатам тестирования все решения, прошедшие имеющийся набор тестов, не прошли сгенерированные таким образом тесты.

Направления дальнейшего исследования по задачам, приведенным в данной работе, таковы:

- генерация тестов для других классов задач;
- генерация тестов, трудных одновременно для нескольких неэффективных решений;
- исследование верхних оценок на время работы различных алгоритмов с помощью генетических алгоритмов;
- улучшение стабильности работы генетических алгоритмов для типов функций приспособленности, время работы которых возрастает вместе со значением функции.

Список литературы

1. *Alander J. T.*, *Mantere T.*, *Turunen P.* Genetic Algorithm Based Software Testing. URL: http://citeseer.ist.psu.edu/40769.html.
2. *Tonella P.* Evolutionary testing of classes / ISSTA. ISSTA. 2004. Pp. 119–128.
3. ACM International Collegiate Programming Contest. URL: http://en.wikipedia.org/wiki/ACM_ICPC.
4. International Olympiad in Informatics. URL: http://www.ioinformatics.org.
5. Интернет-олимпиады по информатике. URL: http://neerc.ifmo.ru/school/io/.
6. *Гэри М.*, *Джонсон Д.* Вычислительные машины и труднорешаемые задачи. Москва: Издательство «Мир», 1982.
7. *Pisinger D.* Algorithms for Knapsack Problems. PhD thesis. University of Copenhagen, 1995.
8. *Chvatal V.* Hard Knapsack Problems // Operations Research. 1980. No.6. Pp. 1402–1411.
9. *Holland J. P.* Adaptation in Natural and Artificial Systems. University of Michigan, 1975.
10. *Скобцов Ю. А.* Основы эволюционных вычислений. Донецк: ДонНТУ, 2008.
11. Задача «Ships. Version 2». URL: http://acm.timus.ru/problem.aspx?num=1394.
12. *Myers G. J.* The Art of Software Testing, Second Edition. John Wiley & Sons, Inc., 2004.
13. *Бек К.* Экстремальное программирование. Питер, 2002.
14. TopCoder. URL: http://www.topcoder.com/tc.
15. Правила проведения полуфинала NEERC. URL: http://neerc.ifmo.ru/information/contest-rules.html.
16. Задачи NEERC-2008. URL: http://neerc.ifmo.ru/regional/problems.pdf.
17. *Оршанский С. А.* О решении олимпиадных задач по программированию формата ACM ICPC // Мир ПК. 2005. №9.
18. *Акишев И. Р.* Об опыте участия в командных соревнованиях по программированию формата ACM // Методическая газета для учителей «Информатика». 2008. №19. С. 20—28.
19. *Харари Ф.* Теория графов. М.: Едиториал УРСС, 2003.
20. *Martello S.*, *Toth P.* A mixture of dynamic programming and branch-and-bound for the subset-sum problem // Management Science. 1984. No.30. Pp. 765–771.
21. *Diffie W.*, *Hellman M. E.* New Directions in Cryptography. 1976.
22. *Mathews G. B.* On the Partition of Numbers / Proceedings of the London Mathematical Society. Proceedings of the London Mathematical Society. No.28. 1897. Pp. 486–490.
23. *Bellman R. E.* Dynamic Programming. Princeton, NJ: Princeton University Press, 1957.
24. *Гилл Ф.*, *Мюррей У.*, *Райт М.* Практическая оптимизация. М.: Мир, 1985.
25. *Kirkpatrick S.*, *Gelatt C. D.*, *Vecchi M. P.* Optimization by Simulated Annealing // Science. 1983. No.4598. Pp. 671–680.
26. Hill Climbing. URL: http://en.wikipedia.org/wiki/Hill_climbing.
27. *Царев Ф. Н.*, *Шалыто А. А.* Применение генетического программирования для генерации автомата в задаче об «Умном муравье» / Труды IV Международной научно-практической конференции «Интегрированные модели и мягкие вычисления в искусственном интеллекте». Труды IV Международной научно-практической конференции «Интегрированные модели и мягкие вычисления в искусственном интеллекте». М.: Физматлит, 2007. С. 590—597.
28. *Forrest S.*, *Mitchell M.* Relative Building-Block Fitness and the Building-Block Hypothesis. 1993.
29. *Gosling J.*, *Joy B.*, *Steele G.*, *Bracha G.* The Java Language Specification, Third Edition. Boston, Mass.: Addison-Wesley, 2005. URL: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.
30. *Гамма Э.*, *Хелм Р.*, *Джонсон Р.*, *Влиссидес Д.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.

31. *Timus Online Judge*. Архив задач с проверяющей системой. URL: http://acm.timus.ru.
32. Задача «Ships». URL: http://acm.timus.ru/problem.aspx?num=1115.