Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Буздалов Максим Викторович

Генерация тестов для олимпиадных задач по теории графов с использованием эволюционных алгоритмов

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

Санкт-Петербург
2011

Содержание

Введение

Олимпиадное движение в области информатики и программирования активно развивается как в России, так и в мире [1–4]. 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 положительно сказывается на качестве соревнования в целом.

Эволюционные алгоритмы — широко развивающийся в последнее время класс оптимизационных алгоритмов, использующих идеи искусственного интеллекта и предназначенных для ведения направленного поиска [5–9]. 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. Эволюционные алгоритмы, в частности, генетическое программирование [10], успешно решают задачи тестирования программного обеспечения, такие как тестирование встроенных систем и автоматическое построение модульных тестов [11, 12]. Однако для олимпиадных задач подходы, реализованные в указанных работах, неприменимы.

В настоящей работе предложен *метод генерации тестов* для олимпиадных задач по программированию по теории графов с использованием эволюционных алгоритмов. Эти тесты предназначены для выявления ре-

шений олимпиадных задач, неэффективных по времени выполнения или по используемой памяти. При таком подходе повышается степень автоматизации процесса генерации тестов и их качество. Предложенный метод был применен к генерации тестов для реальной олимпиадной задачи [13], при этом из имевшихся зачтенных решений новые тесты не прошло более половины. Также метод успешно применен при генерации тестов для задачи о поиске максимального потока в сети.

Глава 1. Обзор предметной области

1.1. Олимпиады по программированию

В мире проводится большое число олимпиад по программированию. Среди них можно отметить международную студенческую олимпиаду по программированию *International Collegiate Programming Contest* [1], проводимую *Association for Computing Machinery* (далее олимпиада будет упоминаться как *ACM ICPC*), с развитой сетью отборочных соревнований, международную олимпиаду школьников по информатике (Information Olympiad in Informatics, IOI) [2], соревнования, проводимые компаниями *TopCoder* [14] и *Google* [15], интернет-олимпиады по информатике [3] и многие другие.

1.1.1. Олимпиадные задачи

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 языков программирования (например, в соревновании *ACM ICPC* используются языки *C*, *C++* и *Java* [4]).

Решение тестируется на наборе из тестов, неизвестных участникам. На его работу накладываются определенные ограничения: максимальное время выполнения, максимальный объем используемой памяти, запрет на выполнение определенных операций (например, работа с сетью, графикой, оконной подсистемой). Ограничение на время выполнения обычно равно 1 – 3 секундам, ограничение на память, как правило, выбирается равным 64, 128 или 256 мегабайтам.

Решение считается прошедшим определенный тест, если оно при работе с ним не нарушило ограничений, завершилось корректно (без ошибок времени выполнения), и выданный им ответ признан правильным. На наборе тестов решение обычно оценивается одним из двух способов:

- Решение считается корректным, если оно прошло все тесты. Если хотя бы один из тестов не пройден, решение считается некорректным, при этом участнику возвращается вердикт, вынесенной тестирующей системой на этом тесте и, опционально, номер теста. Такой способ используется в олимпиадах формата ICPC [1, 4] и в соревнованиях TopCoder [14].
- Каждому тесту назначается определенное число баллов. Баллы, полученные решением, определяются как сумма баллов по всем тестам, пройденным этим решением. Чем больше баллов набрало решение, тем «более правильным» оно является. Такой способ используется в олимпиадах формата IOI [2].

Решение таких задач развивает навыки программирования, исследовательской работы, а на некоторых соревнованиях — навыки работы в команде. В статье [16] изложен процесс решения отдельно взятой задачи при одиночной работе участника, а в статье [17] описана работа в команде.

1.1.2. Решения олимпиадных задач

Решение, которое для всех возможных (в частности, удовлетворяющих ограничениям задачи) входных данных выдает верный ответ, укладываясь при этом в ограничения на время и память, считается корректным. Любое другое решение является некорректным. Некорректные решения можно разделить на два типа: неверные и неэффективные (по времени или по памяти).

Решение считается неверным, если выполнено хотя бы одно из двух условий:

- Решение реализует неверный алгоритм. Например, в задаче о рюк-

заке [18] решения, реализующие различные жадные алгоритмы [19], являются неверными, так как для них всегда существуют тесты, на которых они выдают неверный ответ.

- Решение реализует алгоритм с ошибкой в реализации. Ошибка может приводить к выводу неверного ответа (например, переполнение целочисленных типов), к зацикливанию (программа никогда не останавливается), к ошибке времени выполнения (например, выход за границы массива).

Решение считается *неэффективным по времени*, если для любых входных данных, удовлетворяющих ограничению задачи, оно работает конечное время и выдает верный ответ, но при этом для некоторых таких данных время его работы превышает ограничение по времени. Так, например, для сортировки массива из N чисел при ограничении времени работы в две секунды и выполнении программы на персональном компьютере с тактовой частотой процессора порядка 2 – 3 ГГц решение, реализующее алгоритм сортировки вставками [19], является неэффективным по времени, если $N \leq 10^5$, а решение, реализующее алгоритм сортировки слиянием [19], является эффективным. Если же $N \leq 10^3$, то оба решения признаются эффективными.

Решение считается *неэффективным по памяти*, если для любых входных данных, удовлетворяющих ограничению задачи, оно работает конечное время и выдает верный ответ, но при этом для некоторых таких данных объем используемой им памяти превышает ограничение по памяти.

Неэффективное решение может быть неэффективным как по времени, так и по памяти. В этом случае, если нехватка памяти проявляет себя при меньших ограничениях, то считается, что оно неэффективно по памяти; если первой сказывается нехватка времени, то решение неэффективно по времени. Иначе тип решения не может быть определен, и для него могут быть составлены тесты, приводящие к превышению ограничения как

по времени, так и по памяти.

1.1.3. Тесты для олимпиадных задач

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. В настоящей работе предлагается новый метод автоматизированного составления тестов против неэффективных решений.

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, так как число таких тестов чрезмерно велико. В силу этого из всех возможных тестов необходимо выбрать только те, которые позволят установить, является ли некоторое решение корректным. Однако согласно теореме Райса [20], данная задача в общем случае является алгоритмически неразрешимой. Следовательно, возможно выбрать набор тестов, который лишь с некоторой долей уверенности позволяет утверждать о корректности решения. Цель подготовки тестов состоит в том, чтобы сделать эту уверенность как можно большей.

1.1.4. Подготовка тестов

Подготовка тестов к олимпиадной задаче является творческим процессом. Для каждой задачи пишется несколько решений, называемых решениями жюри. Среди них должны быть как корректные, так и неверные и неэффективные решения. Цель решений жюри — проверка тестов: любое корректное решение должно пройти все тесты, для любого некорректного решения должен существовать минимум один тест, который оно не проходит.

Часть тестов пишется вручную. Такие тесты проверяют решения на корректность разбора случаев, встречающихся в задаче. К таким тестам

относятся и так называемые «минимальные» тесты, которые проверяют корректность работы решений около нижних границ ограничений.

Тесты большого размера, как правило, генерируются согласно некоторому шаблону. Так, например, если в условии задачи фигурирует некоторый граф, то можно сгенерировать двоичное дерево, полный двудольный граф и другие виды графов [21]. Некоторые шаблоны подразумевают множество вариантов реализации: так, например, дерево с N пронумерованными вершинами можно сгенерировать N^{N-2} различными способами [21]. В таком случае при генерации теста некоторые действия выполняются случайным образом.

Для покрытия тех ошибок, которые могут быть не найдены с помощью описанных выше тестов, создаются «случайные» тесты. В общем случае большие тесты создаются с помощью программ, написанных членами жюри.

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. Жюри стремится предотвратить такое развитие событий, для чего реализуются эвристические решения и генерируются тесты против них. Однако не все идеи эвристических решений могут быть найдены или реализованы, а в отдельных случаях поиск тестов против них может затянуться на неопределенное время. По указанным причинам, на олимпиадах иногда встречаются задачи со «слабым» набором тестов — таким, который пропускает некорректные решения. Это приводит к тому, что наиболее подготовленные участники олимпиад, ищущие корректные решения, не решают такие задачи, в то время как менее подготовленные участники успешно сдают некорректные решения, что противоречит самой идее олимпиады.

1.2. Необходимые элементы теории графов

В данном разделе будут изложены элементы теории графов, используемые в работе. Все понятия будут проиллюстрированы примерами из практики олимпиадного программирования.

1.2.1. Виды графов с точки зрения олимпиадных задач

В литературе по теории графов часто встречаются разночтения даже в таких основных понятиях, как, собственно, «граф» [21]. Опишем, что чаще всего понимается под графом в олимпиадных задачах и в данной работе.

Определение 1. *Графом* $G = (V, E)$ называется пара из множества V и мультимножества E. Множество V — это множество *вершин* данного графа. Мультимножество E — это совокупность *ребер* данного графа. Каждый элемент мультимножества E имеет вид (s, t), где $s \in E, t \in E$. В дополнение к этому, каждая вершина и каждое ребро могут иметь дополнительные данные (свойства), присоединенные к ним.

Данное определение графа является достаточно общим. В частности, под него подпадают бесконечные графы, которые возникают в теории игр. В данной работе, посвященной генерации тестов в виде графов, бесконечные графы рассматриваться не будут, таким образом, мы полагаем V и E конечными.

В конкретных олимпиадных задачах на графы накладываются ограничения, а также указываются конкретные типы свойств вершин и ребер. Существуют «тривиальные» ограничения, такие как максимальное число вершин и максимальное число ребер. Такие ограничения всегда либо явно указываются в условии задачи, либо непосредственно из него вытекают.

Перечислим другие ограничения, часто применяемые в олимпиадных задачах.

- Граф является *неориентированным*. Это означает, что ребра рас-

сматриваются как неупорядоченные пары вершин, то есть, ребра (u, v) и (v, u) — это одно и то же ребро. В противном случае, граф является *ориентированным*, и у каждого ребра есть *начало*, *конец* и, следовательно, *направление*. В условии задач тип графа всегда указывается, часто неявно.

- В графе нет петель. *Петля* — это ребро из вершины в себя, то есть, имеющее вид (v, v).
- В графе нет кратных ребер. При этом условии мультимножество E может рассматриваться как множество (при этом, в зависимости от того, является ли граф ориентированным, элементы этого множества могут быть как упорядоченными парами, так и неупорядоченными).
- Граф является *связным*. В применении к неориентированному графу это означает, что между любыми двумя вершинами существует путь. В случае ориентированного графа можно ввести понятие *сильной связности* — когда из любой вершины в любую другую можно добраться по ребрам, двигаясь по каждому из них по направлению ребра. Также иногда вводят условие, что между любыми двумя вершинами в ориентированном графе существует путь, игнорирующий направления ребер.
- Граф является *ациклическим*. В применении к неориентированному графу это требование эквивалентно тому, что граф является или *деревом*, или *лесом* — совокупностью деревьев [21]. Для ориентированных графов это условие является менее строгим.
- С каждым ребром графа может быть ассоциирована его *длина* или *вес*. Помимо тривиальных ограничений на длину, таких как максимальная длина или неотрицательность, может быть наложено следующее ограничение: суммарная длина любого цикла не должна быть отрицательной.

Практически все способы задания графа включают в себя какое-

либо обозначение вершин. Наиболее простой способ — дать каждой вершине номер в непрерывном интервале, например, от 1 до $|V|$. Вершина может задаваться и номером из более широкого диапазона, и даже строкой (в условии задачи эта строка может фигурировать как название города, имя человека и т.д.), но для удобства работы с графом вершины, как правило, в любом случае нумеруют числами либо от 1 до $|V|$, либо от 0 до $|V| - 1$. Поэтому можно считать, что граф всегда является *перенумерованным*, или *помеченным* [21].

Большинство способов задания графа также нумеруют ребра. Чаще всего ребра нумеруются в порядке их описания. В условии некоторых задач требуется вывести в выходной файл какие-либо данные о ребрах (например, поток, текущий по ним, при каком-либо максимальном потоке в сети [21]) именно в порядке появления ребер во входном файле.

Каждое ребро может обладать множеством свойств. Наиболее часто встречающиеся свойства — это *длина* (или *вес*, или *стоимость*) ребра, а также его *пропускная способность*. В некоторых задачах, использующих термины теории строк или автоматов, на ребре может быть «написана» строка.

1.2.2. Представление графов в памяти компьютера

Среди разнообразных способов представления графа в памяти компьютера при решении олимпиадных задач чаще всего используются два: *матрица смежности* и *списки смежности*. При реализации решения задачи тот или иной способ представления выбирается, исходя из следующих критериев:

- требуемое множество операций и необходимое для них быстродействие;
- объем памяти, необходимый для размещения графа;
- наличие или отсутствие ориентации ребер, петель, кратных ребер.

Далее рассмотрим основные способы представления, их характеристики, преимущества и недостатки.

1.2.2.1. Матрица смежности

Основой данного представления является матрица размером $|V| \times |V|$, где V — множество вершин графа. В i-той строке и j-том столбце этой матрицы находится информация о ребре графа из вершины i в вершину j. В случае графа без пометок эта информация состоит из одного бита: существует ли указанное ребро или нет. Матрица смежности, как правило, строится для графа без кратных ребер, однако в нем могут быть петли, не более одной для каждой вершины.

Преимущество данного представления заключается в том, что по двум вершинам за время $O(1)$ можно определить, существует ли ребро из одной вершины в другую, а также свойства этого ребра. Недостатки его становятся очевидными для разреженных графов:

- Матрица смежности занимает $O(|V|^2)$ памяти, что может быть неприемлемо много;
- Для получения всех соседей данной вершины требуется проверить $|V|$ кандидатов независимо от степени вершины.

По этой причине матрицы смежности используются для работы с небольшими или плотными графами.

1.2.2.2. Матрица смежности двудольного графа

Если известно, что граф, который требуется хранить, является двудольным, то матрицу смежности можно упростить. Пусть в левой доле графа L вершин, а в правой — R вершин. Тогда вся информация о ребрах графа может быть размещена в матрице размером $L \times R$. При этом экономится (в случае неориентированного или ориентированного из левой в правую долю графа) $L \cdot L + R \cdot R + L \cdot R$ ячеек матрицы.

Все доводы о преимуществах и недостатках матрицы смежности переносятся и на матрицу смежности двудольного графа.

1.2.2.3. Списки смежности

При данном представлении графа имеется $|V|$ списков ребер. В i-том списке перечислены в произвольном порядке ребра, выходящие из вершины с номерм i. В конкретной реализации, в зависимости от решаемой задачи, могут храниться только номера вершин, в которые ведут эти ребра, а также может храниться иная информация, такая как вес ребра, его пропускная способность и т.д.

Преимущество описанной структуры данных заключается в том, что она занимает $O(|V| + |E|)$ памяти, таким образом, она способна эффективно хранить разреженные графы. Кроме того, по номеру вершины v за $O(deg(v))$ возможно перечислить ее соседей (в случае неориентированного графа каждое ребро содержится в списках обеих смежных вершин).

Недостатком списков смежности является то, что по двум вершинам, в отличие от матрицы смежности, нельзя «быстро» определить, существует ли инцидентное им обеим ребро — для этого необходимо проверить списки смежности этих вершин, таким образом, проверка существования ребра uv происходит за время $O(deg(u) + deg(v))$. Кроме того, удаление ребра, в отличие от добавления, также требует времени $O(deg(v))$.

Еще одним недостатком является то, что в ориентированном графе для данной вершины v нельзя быстро перечислить вершины, из которых в v ведет ребро. Для обеспечения этой функциональности требуется построить списки смежности для развернутого графа, а также, при необходимости, обеспечить соответствие ребер в обеих структурах. Для плотного ориентированного графа в этом случае может потребоваться в два раза больше памяти, чем при использовании матрицы смежности, за счет того, что требуется хранить номер вершин, в которые ведут ребра.

1.2.2.4. Хеш-таблица

Граф можно также представить в виде ассоциативного массива $g\colon V \times V \to E$, а в случае графа с кратными ребрами — как $g\colon V \times V \to [E]$. Эффективной реализацией ассоциативного массива является хеш-таблица. Данное представление графа обладает всеми плюсами матрицы смежности, но занимает $O(|V|+|E|)$ памяти. Возможно также обеспечить эффективное перечисление соседей данной вершины, интерпретировав $g\colon V \times V \to E$ как $g\colon V \to (l\colon V \to E)$ и реализовав хранение графа в виде ассоциативных массивов, содержащих ассоциативные массивы.

Главным недостатком данного представления является большая скрытая константа реализации как в оценке времени работы операций, так и в оценке используемой памяти. По этой причине хеш-таблицы крайне редко применяются для хранения графов при решении олимпиадных задач.

1.2.3. Тематика олимпиадных задач по теории графов

Приведем основные темы олимпиадных задач по теории графов. Для каждой темы, с целью оценки ее популярности, указано число задач по этой теме в интернет-архиве задач *acm.timus.ru* [22], отмеченных тегом «графы»:

- обход в ширину или в глубину: 29 задач;
- кратчайшие расстояния (пути) из одной вершины: 9 задач;
- кратчайшие расстояния (пути) между всеми парами вершин: 8 задач;
- паросочетания (включая паросочетания минимального веса): 6 задач;
- потоки в сети (включая потоки минимальной стоимости): 6 задач;
- построение минимальных остовных деревьев: 5 задач;
- эйлеров обход графа, эйлеровы пути и циклы: 4 задачи;

- динамическое программирование на ациклическом графе: 4 задачи;
- топологическая сортировка графа: 3 задачи;
- покрытие графа минимальным числом путей: 2 задачи;
- раскраска графа: 2 задачи;
- игры на конечных графах: 2 задачи;
- клики и независимые множества: 2 задачи;
- поиск отрицательных циклов: 1 задача;
- компоненты сильной связности и конденсация: 1 задача;
- поиск ближайшего общего предка в дереве: 1 задача;
- компоненты двусвязности: 1 задача;
- другие задачи: 15 задач.

Необходимо учитывать, что если задача отнесена к определенной тематике, например, «обход в глубину», это не означает, что решение задачи состоит в том, чтобы реализовать лишь обход в глубину. Например, в процессе обхода в глубину каждой посещенной вершине графа может приписываться некоторое значение, которое вычисляется при обходе с помощью динамического программирования. Граф, с которым требуется провести определенные манипуляции, также может быть не задан явно во входном файле, а должен быть вычислен на основе входных данных, возможно, нетривиальным образом (например, с помощью решения некоторых задач вычислительной геометрии).

1.3. Эволюционные алгоритмы

1.3.1. Генетические алгоритмы

Основа концепции генетических алгоритмов была заложена в опубликованной Дж. Холландом (J. P. Holland) в 1975 году книге «Адаптация в естественных и искусственных системах» [5]. 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 [6]:

- 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 [23]. 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 [23]. 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 по-

лучения 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 оптимизации функций, имеющих разрывы, а также дискретных функций.
- Использование на каждой итерации алгоритма множества рассматриваемых значений. В отличие от более «локальных» методов оптимизации (метод имитации отжига [25], эволюционные стратегии [8]), 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.
- Простота распараллеливания. Генетические алгоритмы легко модифицируются для выполнения на многопроцессорных и распределенных системах.

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

- Высокая трудоемкость. Для поиска решения, близкого к оптимальному, нередко приходится рассматривать большое число особей. Хо-

тя это число, как правило, значительно меньше размера допустимого пространства поиска, для многих задач оно все же остается большим [26].

- 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 [6], 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.

1.3.1.1. 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 выбираются 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 слу-

чайным образом выбирается индекс $k \in [1; L-1]$ и формируются две новые особи $AB = a_1 \dots a_k b_{k+1} \dots b_L$ и $BA = b_1 \dots b_k a_{k+1} \dots a_L$. 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).

1.3.1.2. 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. Для корректной работы этого метода любая особь должна иметь неотрицательное, а лучше положительное значение приспособленности. Кроме того, значение приспособленности должно расти с ростом приспособленности (чем больше значение, тем лучше особь).
- 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 особи, подлежащей скрещиванию, из исходной популяции выбирается случайным образом группа небольшого размера m, в которой путем проведения «турниров» определяется лидер. Каждый «турнир» проводится между двумя особями и приводит к тому, что менее приспособленная особь с большой ве-

роятностью (устанавливаемой параметром отбора) выбывает из отбора. Победитель отбора отбирается для скрещивания.

1.3.1.3. Операторы скрещивания

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

- Одноточечный кроссовер. Его действие описано в разд. 1.3.1.1.
- 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.

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

1.3.1.5. Генетическое программирование

Особый вид генетических алгоритмов — *генетическое программирование* — использует в качестве хромосомы некоторое представление программы, например, синтаксическое дерево разбора программы [12] или конечный автомат [26]. Основы генетического программирования заложил Дж. Коза (Koza J. R.) в 1992 году [10]. 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.

1.3.2. Эволюционные стратегии

Еще одним семейством алгоритмов оптимизации, использующим идеи биологической эволюции, являются эволюционные стратегии [7–9, 27, 28]. Эволюционные стратегии были впервые описаны в работе [7]. Вначале они использовались для сложных задач оптимизации с непрерывными параметрами, таких как оптимизация формы крыла самолета [27], а затем область их применения была расширена и на дискретные задачи оптимизации [28].

1.3.2.1. $(1 + 1)$-эволюционная стратегия

Простейшей формой эволюционной стратегии является $(1 + 1)$-эволюционная стратегия [8]. «Поколение» этой эволюционной стратегии состоит из одной особи. Изначально эта особь создается случайным образом. На каждой итерации из текущей особи путем мутации — случайного, как правило, небольшого изменения — создается новая особь. Если новая особь представляет собой не менее оптимальное значение, чем ее «родитель», то

именно она выбирается алгоритмом и проходит в следующее поколение. В противном случае, все остается неизменным.

Данный эволюционный алгоритм также называют «метод спуска со случайными мутациями» (англ. Random Mutation Hill Climbing) [6, 9]. Главным его преимуществом перед иными эволюционными алгоритмами является простота реализации — для любой оптимизируемой сущности достаточно реализовать лишь оператор мутации.

Недостатком этого алгоритма является его высокая локализованность. Если оператор мутации может делать лишь небольшие изменения, то алгоритм никогда не уйдет далеко от начального значения. Если же этот оператор делает, как правило, большие изменения, то алгоритм мало чем отличается от случайного поиска. Решением этой проблемы может быть аккуратный выбор оператора мутации. Например, для особей, являющихся вектором вещественных чисел, вектор мутации можно генерировать с помощью гауссова случайного распределения [9]. Еще один недостаток — принципиальная однопоточность вычислений.

Для множества задач $(1+1)$-эволюционная стратегия является наиболее предпочтительным выбором в качестве эволюционного алгоритма оптимизации. В дополнение к простоте реализации, этот алгоритм, как правило, достаточно быстро находит приемлемое по оптимальности решение, а число вызовов функции приспособленности при этом может быть в 5–10 раз меньше, чем у генетического алгоритма с тем же оператором мутации [6].

Дальнейшим развитием идеи $(1+1)$-эволюционной стратегии являются $(\mu+\lambda)$ и (μ,λ)-эволюционные стратегии.

1.3.2.2. (μ,λ)-эволюционная стратегия

В данном виде эволюционной стратегии поколение состоит из μ особей. На каждом шаге каждая из этих особей порождает λ/μ потомков с помощью оператора мутации (λ должно делиться на μ нацело). Среди по-

лучившихся λ потомков выбираются μ наиболее приспособленных, которые и составляют следующее поколение. Особи из предыдущего поколения в следующее поколение не проходят.

(μ, λ)-эволюционная стратегия призвана решить некоторые проблемы, описанные ранее для $(1 + 1)$-эволюционной стратегии. В частности, так как в работу на каждой итерации вовлечены уже несколько особей, вероятность преждевременной сходимости к локальному, но не глобальному оптимуму становится меньше. Кроме того, получение нескольких потомков одновременно позволяет распараллелить процесс вычисления функции приспособленности.

1.3.2.3. $(\mu + \lambda)$-эволюционная стратегия

Данный вид эволюционной стратегии является, в некотором смысле, компромиссом между (μ, λ) и $(1 + 1)$-эволюционными стратегиями. Единственное отличие от первой из них состоит в том, что в следующее поколение проходят μ лучших особей из всех имеющихся на данный момент особей: μ родителей и λ потомков. При таком решении функция приспособленности наилучшей особи не ухудшается в процессе эволюции. С другой стороны, описанный алгоритм в большей степени подвержен преждевременной сходимости, чем (μ, λ)-эволюционная стратегия. Выбор эволюционной стратегии для решения конкретной задачи производится на основе предварительного анализа пространства поиска и экспериментальных исследований того или иного варианта в условиях этой задачи.

(1+1) и (1+λ)-эволюционная стратегия во многом являются частными случаями $(\mu + \lambda)$- стратегии, за исключением того, что в общем случае может быть введен оператор рекомбинации (см. разд. 1.3.2.5).

1.3.2.4. Выбор степени мутации и правило «одной пятой»

Во всех вариантах эволюционной стратегии имеется один параметр, значение которого способно изменить картину сходимости алгоритма к оп-

тимуму — это степень мутации. Для особей-вещественных векторов эта величина обычно обозначается как σ^2 и описывает, насколько сильно позволяется изменить ту или иную компоненту вектора. Например, в качестве оператора мутации может использоваться свертка с равномерным распределением $U(-\sigma^2, \sigma^2)$ или с нормальным (гауссовым) распределением $N(0, \sigma^2)$ [9].

Существует несколько подходов к тому, какую степень мутации использовать. Наиболее простой из них — использовать единую для всех мутаций величину, которую, в свою очередь, можно подстраивать в процессе работы алгоритма. Для некоторых эталонных задач в работе [7] выведено следующее эвристическое правило «одной пятой»:

- если доля потомков с лучшим значением приспособленности больше $1/5$, то степень мутации необходимо повысить;
- если эта доля меньше $1/5$, то степень мутации необходимо понизить;
- иначе, степень мутации изменять не требуется.

Данная эвристика обеспечила на эталонной задаче компромисс между «исследованием новых областей» и «эксплуатацией имеющихся», что привело к наилучшей сходимости алгоритма к глобальному оптимуму. Похожая эвристика была разработана для (μ, λ)-эволюционных стратегий: для них на этой же задаче оптимальным соотношением оказалось $5\mu \leq \lambda \leq 6\mu$.

Другой способ подстроить степень мутации — включить вектор степеней мутации, равный по длине вектору значений особи, в саму особь и позволять ему мутировать по аналогичным правилам [28]. В этом случае оператор мутации выглядит следующим образом:

$$\sigma_i^{t+1} = \sigma_i^t \cdot \exp^{N(0,\Delta\sigma)};$$
$$x_i^{t+1} = x_i^t + N(0, \sigma_i^{t+1}).$$

1.3.2.5. Операторы рекомбинации

В вариантах эволюционной стратегии, в которых число особей в поколении превышает единицу, возможно, по аналогии с генетическими алгоритмами, вводить операторы рекомбинации (скрещивания). Эксперименты над особями-векторами вещественных чисел показали, что одним из наиболее целесообразных операторов рекомбинации для таких особей является «среднее арифметическое» [8]. Однако ничто не мешает использовать для этой цели, например, операторы скрещивания, используемые в генетических алгоритмах. При этом грань между эволюционными стратегиями и генетическими алгоритмами стирается. Так, например, классический генетический алгоритм с фиксированным размером поколения может быть рассмотрен как (λ, λ)-эволюционная стратегия, а генетический алгоритм из работы [29], использующий сильный элитизм, напоминает $(\lambda + \lambda)$-эволюционную стратегию.

Выводы по главе 1

Олимпиадное движение в области информатики и программирования является развитым движением в России и в мире, способствует выявлению талантливых и способных программистов.

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.

Эволюционные алгоритмы, такие как генетические алгоритмы и эволюционные стратегии, являются современным семейством алгоритмов, решающих сложные задачи оптимизации. Среди их преимуществ — возможность поиска оптимумов в задачах с дискретной областью определения функции и трудноопределимым локальным поведением. Предлагается

использовать указанные эволюционные алгоритмы для поиска тестов для олимпиадных задач, демонстрирующих неэффективность (по времени или по памяти) определенных решений.

Глава 2. Описание предлагаемого подхода

2.1. Поиск теста как задача оптимизации

Тесты к каждой конкретной олимпиадной задаче образуют конечномерное, как правило дискретное пространство. Размерность пространства задается ограничениями на основные параметры теста. В применении к задачам по теории графов, этими параметрами являются:

- максимальное число вершин в графе;
- максимальное число ребер в графе;
- максимальный вес ребра;
- максимальная пропускная способность ребра;
- и т.д.

Поиск теста, удовлетворяющего тому или иному критерию, ведется в пространстве тестов. Как правило, число возможных тестов, а следовательно, размер пространства, является слишком большим, чтобы можно было перебрать все тесты. К примеру, если число вершин в графе не превышает V, число ребер не превышает E, каждое ребро имеет целочисленную пропускную способность, не превышающую C, и в графе могут быть петли и кратные ребра, то число различных графов равно $(C \cdot V^2)^E$. При весьма небольших ограничениях $V \leq 100$, $E \leq 5000$, $C \leq 10000$ число различных графов превышает 10^{40000}, а число посимвольно различных тестов, допускающих перестановку описаний ребер, еще больше.

С другой стороны, критерий выбора теста чаще всего не только формализуем, но и может быть выражен в виде числа или объекта Q с введенным на множестве таких объектов отношением полного порядка $<: Q \times Q \to \{0, 1\}$. В случае поиска тестов против неэффективных решений этой величиной может быть время, требуемое рассматриваемому

решению на работу с тестом, или объем используемой при этом памяти, или эквивалентная величина, показывающая, насколько тот или иной тест сложен для решения. По указанной причине, задачу поиска теста против неэффективного решения возможно сформулировать в виде задачи оптимизации, например, «максимизировать время работы решения при выполнении ограничений на тест, указанных в условии задачи».

Конкретный вид ограничений на тест зависит от условия задачи — как от формата входных данных, так и от накладываемых дополнительно ограничений. К примеру, в задаче может быть указано не только максимальное число вершин и ребер в графе, но и то, что граф должен быть планарным [21], или то, что требуется найти по условию задачи (например, кратчайший путь от одной вершины до другой), существует.

Применение стандартных техник теории оптимизации встречает множество препятствий. Во-первых, пространство оптимизации, как правило, дискретно, что препятствует применению методов первого и более высоких порядков, опирающихся на существование производных. Во-вторых, ограничения на тесты могут быть столь нетривиальны, что даже их проверка может быть не проще, чем решение задачи. В-третьих, значение оптимизируемой величины зачастую можно получить только путем запуска решения на тесте, при этом зависимость получаемых значений от входных данных может быть совершенно непредсказуема. В частности, редко можно сделать вывод о сложности теста для некоторого решения по сложности составных частей этого теста для рассматриваемого решения. По этой причине не представляется возможным применение динамического программирования, перебора с отсечениями или сведение, например, к задаче о выполнимости булевой формулы.

По указанным причинам, для решения задачи оптимизации выбраны эволюционные алгоритмы, так как они часто применяются в тех случаях, где более традиционные подходы не позволяют получить результаты приемлемой оптимальности.

2.2. Выбор оптимизируемой функции

Одна из наиболее сложных проблем при генерации тестов для олимпиадных задач с использованием методов оптимизации заключается в выборе оптимизируемой функции, или, в терминах эволюционных алгоритмов, *функции 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 тестовой программы, 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

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

№ теста	Время исполнения тестовой программы, 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

Описанные недостатки могут привести к выделению «ложного лидера» — ничем не выделяющаяся особь получит преимущество над остальными, что приводит к стагнации уже на начальной стадии или к существенному ухудшению производительности алгоритма оптимизации.

Для исправления этих недостатков предлагается использовать иной подход. Для этого в тестируемом решении необходимо вычислять величину, характеризующую сложность теста для данного решения. В простейшем случае эта величина может быть равна числу элементарных действий, например, вызовов определенной процедуры. Однако она может быть устроена гораздо сложнее. В конце работы решение возвращает значение этой величины вместе с ответом, а проверяющая система читает это значение и на его основе формирует значение функции приспособленности.

Хотя описанный подход требует некоторой предварительной работы с тестируемыми решениями, преимущества, которые он дает, перекрывают его недостатки. Показатель сложности теста, при должной реализации, коррелирует и с временем работы программы, и с реальной сложностью теста для решения. В то же время, он не имеет флуктуаций во времени и имеет гораздо больший диапазон значений, что стабилизирует работу эволюционных алгоритмов.

На рис. 2.1 приведены графики роста функций приспособленности для четырех различных функций приспособленности при генерации теста для решения задачи о поиске максимального потока в сети (использовался генетический алгоритм). Каждый из графиков отражает усреднение функции приспособленности по пяти запускам. Две из этих функций основаны на непосредственном измерении времени работы решения (астрономическое и процессорное время), в то время как две других — на показателях сложности теста (число запусков обхода в глубину и число посещенных этим обходом вершин). Из рисунка видно, что функции приспособленности, основанные на подсчете сложности теста, существенно выигрывают у функций, основанных на непосредственном измерении времени.

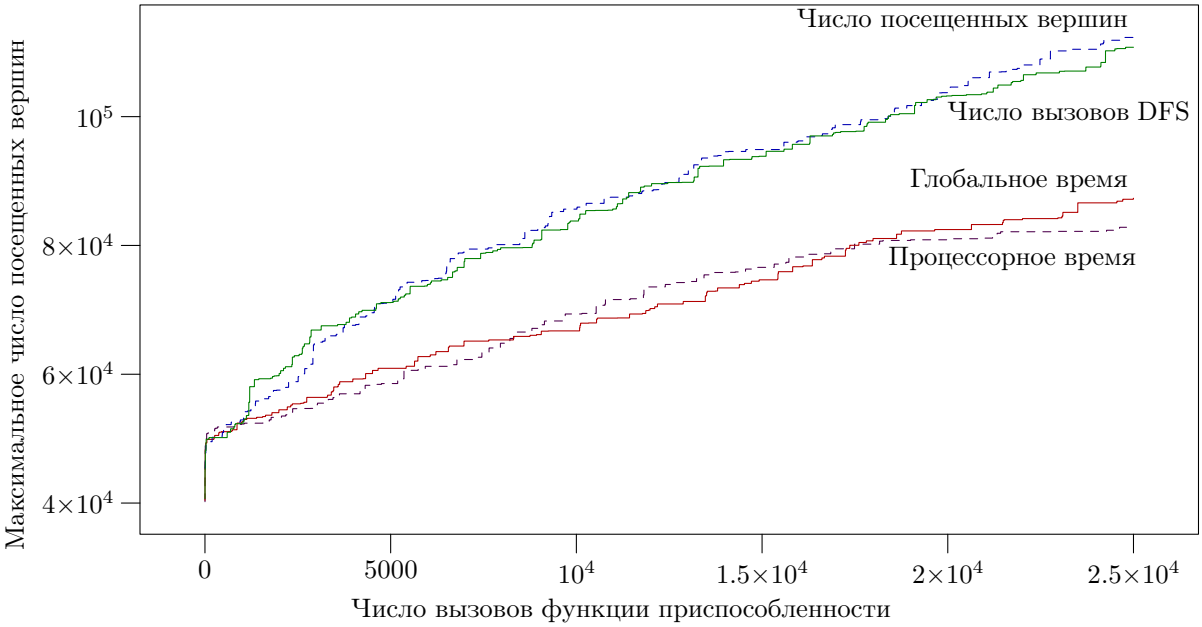

Рис. 2.1. Сравнение функций приспособленности — пример 1

Существуют решения, для которых функции приспособленности, основанные на времени, вообще не приводят к сколь-нибудь существенным результатам. На рис. 2.2 приведен пример графиков времени работы решения при оптимизации по двум различным функциям приспособленности.

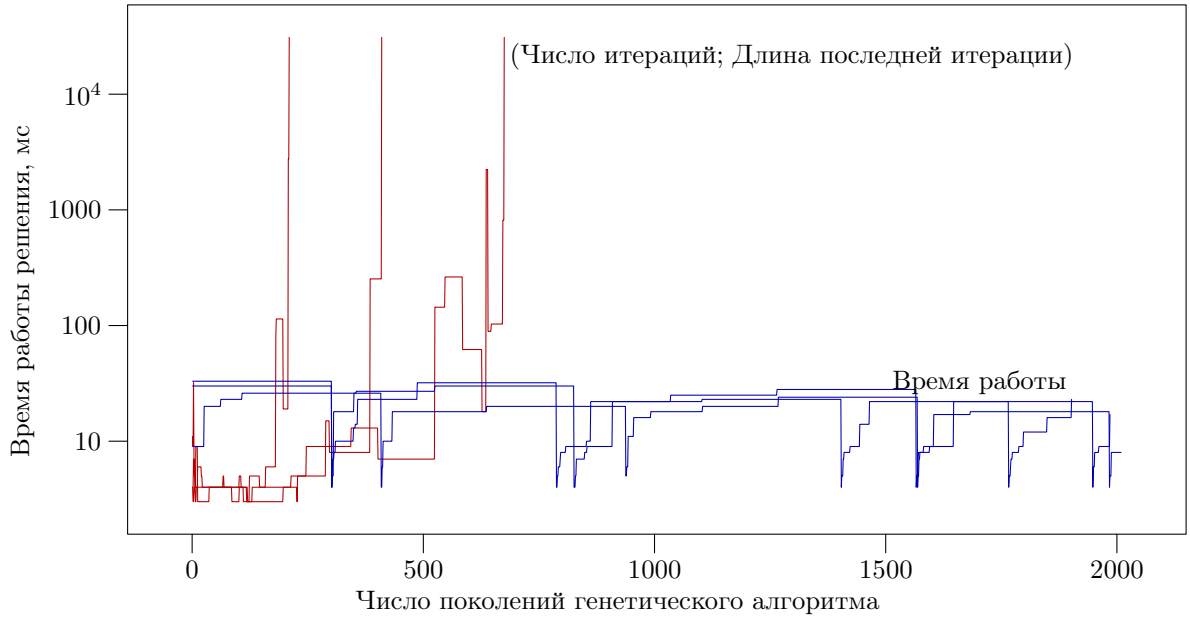

Рис. 2.2. Сравнение функций приспособленности — пример 2

Из рисунка видно, что при выборе функции приспособленности, основанной на показателе сложности теста, цель генерации теста достигается

(время работы решения превышает 30 секунд), в то время как при использовании времени работы как функции приспособленности время работы решения остается очень малым в течение длительного времени.

Выводы по главе 2

Описаны принципы сведения генерации тестов для олимпиадной задачи к решению задачи оптимизации. Приведено обоснование того, что полученную задачу оптимизации крайне трудно или невозможно решить с помощью стандартных методов теории оптимизации. Обоснован выбор эволюционных алгоритмов в качестве алгоритмов оптимизации.

Приведены теоретические и практические обоснования того, что непосредственно измеренное время работы решения и используемая им память недостаточно эффективны в качестве оптимизируемой функции (функции приспособленности). Предложен подход к разработке функций приспособленности, основанный на подсчете величин, характеризующих сложность теста для рассматриваемого решения, и продемонстрирована его эффективность на нескольких примерах.

Глава 3. Генерация тестов для задачи «Work for Robots»

В данной главе описывается применение подхода, описываемого в данной работе, к задаче «Work for Robots». С текстом условия этой задачи можно ознакомиться на сайте Timus Online Judge [22], где она размещена под номером 1695 [13].

3.1. Условие задачи

Дан неориентированный граф без петель и кратных ребер. Число вершин графа N находится в диапазоне от одной до 50. Необходимо найти число клик в этом графе. Ограничение по времени — две секунды, ограничение по памяти — 64 мегабайта.

3.2. Решения задачи

3.2.1. Авторское решение

Авторское решение этой задачи основано на известном среди участников олимпиад по программированию приеме «meet-in-the-middle».

Разделим вершины графа на два множества L и H, так чтобы $|L| \leq 23$ и $\big||H| - |L|\big|$ было минимальным. Занумеруем вершины множества L числами от 0 до $|L| - 1$. Любое подмножество множества L может быть выражено в виде битовой маски — неотрицательного целого числа, не превосходящего $2^{|L|} - 1$, в двоичном представлении которого единицы в разрядах $i_1, \ldots, i_k$ соответствуют входящим в это подмножество вершинам с индексами соответственно $i_1, \ldots, i_k$. Аналогично можно задавать подмножества множества H.

Пусть E — множество ребер в данном графе. Введем функцию покрытия e, которая сопоставляет множеству вершин V множество вершин,

для которых каждая вершина из V является смежной:

$$e(V) = \{\, u \mid \forall v \in V uv \in E \,\}.$$

Число клик $c(A)$ для каждого подмножества A множества L может быть вычислено по следующим формулам:

$$c(\emptyset) = 1;$$
$$c(A \cup \{v\}) = c(A) + c(A \cap e(\{v\})).$$

При использовании кодирования подмножеств множества L, описанного выше, и хранения значения $c(A)$ в массиве `cliquesL`, индексируемом целочисленным представлением подмножества A, вычисление всех $c(A)$ можно выполнить за $O(2^{|L|})$ времени и памяти с помощью динамического программирования. При указанном ограничении на размер L массив `cliquesL` занимает не более 32 мегабайт оперативной памяти.

Рассмотрим произвольную клику C и два ее подмножества $C_L = C \cap L$, $C_H = C \cap H$. Очевидно, что $C_L \subset e(C_H) \cap L$, иначе C не было бы кликой. Следовательно, если бы мы посчитали для C_H число клик в $e(C_H) \cap L$, то рассматриваемая клика была бы посчитана ровно один раз.

Отсюда следует идея второй половины решения. Для каждого подмножества C множества H, являющегося кликой, вычислим $L_C = e(C_H) \cap L$ и найдем число клик в L_C в массиве `cliquesL`. Сумма всех таких значений и будет ответом, так как каждая клика в исходном графе будет учтена ровно один раз.

Для эффективной реализации этой идеи требуется быстро выяснять, является ли данное подмножество C множества H кликой, а также быстро находить $e(C) \cap L$. Для этого достаточно разбить множество H на две приблизительно равных половины H_L и H_H, подсчитать для каждого подмножества из этих половин функции покрытия и вычислять требуемое по следующим формулам:

$$isClique(C) = (e(H_L) \cap e(H_H) \cap H = C);$$
$$e(C) \cap L = e(H_L) \cap e(H_H) \cap L.$$

Вычисление указанных значений для каждой из половин требует $O(2^{\lceil |H|/2 \rceil})$ времени и памяти. Вычисление итоговой суммы, а, следовательно, и ответа, требует $O(2^{|H|})$ времени и $O(1)$ дополнительной памяти.

Итоговое время работы оценивается как $O(2^{N-\min(23,\lfloor N/2 \rfloor)})$ с достаточно небольшой скрытой константой. Фактическое время работы находится в пределах от $0,5$ до $1,5$ секунд при реализации на языках *C++* и *Java*. Используется $4 \cdot 2^{\min(23,\lfloor N/2 \rfloor)} + 16 \cdot 2^{\lceil (N-\min(23,\lfloor N/2 \rfloor))/2 \rceil} + O(N)$ байт памяти, что составляет приблизительно 33 мегабайта.

Автору работы также известно о существовании решения с оценкой на время работы $O(2^{3N/7})$.

3.2.2. Неэффективные решения

По состоянию на 1 августа 2009 года по этой задаче было засчитано множество решений, основанных на неэффективных алгоритмах. К таким алгоритмам относится, например, перебор с запоминанием — число состояний, которые требуется запомнить, в худшем случае чрезвычайно велико, но на всех имевшихся тестах это число было небольшим. Были также сданы некоторые решения, эффективные по времени, но потенциально использующие больше памяти, чем разрешено.

3.2.2.1. Перебор с запоминанием

Решения, использующие перебор с запоминанием, реализованы, как правило, в виде рекурсивной процедуры, которая вычисляет ответ для подграфа исходного графа, построенного на некотором подмножестве его вершин. Если для данного подграфа A ответ еще не вычислялся, то выбирается некоторая вершина v этого подграфа и производится вычисление ответа по формуле:

$$ans(A) = ans(A \setminus v) + ans(A \setminus v \cap e(v)).$$

При этом возможно использование некоторых эвристик, ускоряющих работу программы. Например, если $A \setminus v \subset e(v)$, то

$ans(A) = 2 \cdot ans(A \setminus v)$, при этом удается избежать повторного вызова процедуры на том же подмножестве.

После вычисления полученный ответ, если возможно, сохраняется. Если же для данного подграфа сохранен ранее вычисленный ответ, то он и возвращается. При каждом вызове такой процедуры выполняется $O(1)$, $O(\log N)$ или $O(N)$ действий (где N — число вершин в графе) в зависимости от структуры данных, используемой для хранения подсчитанной информации, и от стратегии выбора удаляемой вершины.

3.2.2.2. Другие способы решения

Помимо описанных способов решения, известны решения, основанные на следующих принципах:

- поиск наибольшего независимого подмножества и динамическое программирование;
- предварительное переупорядочивание вершин и перебор с запоминанием;
- случайная перестановка вершин и перебор с запоминанием.

С помощью переупорядочивания вершин можно добиться уменьшения асимптотической оценки на время работы и потребляемую память до величины $O(2^{3N/7})$. Остальные описанные способы не дают асимптотического улучшения, но, за счет использования генератора случайных чисел, могут в константное число раз снизить математическое ожидание времени работы.

3.3. Эволюционная стратегия

Для генерации тестов к этой задаче была использована $(1 + 1)$-эволюционная стратегия.

3.3.1. Кодирование теста

Тесты для данной задачи предлагается кодировать в виде матрицы булевых значений размера $N \times N$, являющейся матрицей смежности графа, данного в тесте. При этом на диагонали матрицы всегда стоят нули. Такой способ кодирования является достаточно простым. Кроме того, попытки закодировать тест другим способом оказались недостаточно эффективными. Чтобы получающиеся тесты были как можно более трудными, для их генерации взято максимально возможное число вершин в графе ($N = 50$).

3.3.2. Операторы мутации

Для описанного выше способа кодирования разработано два оператора мутации. Первый оператор инвертирует один элемент матрицы, не стоящий на диагонали, и симметричный ему. Второй оператор симметрично инвертирует случайно выбранные элементы матрицы в количестве, не превышающем 1/10 от числа всех элементов матрицы. Данные операторы применяются поочередно.

Целью первого из операторов является попытка перепробовать все возможные «единичные» изменения теста. Для этого элементы матрицы, подлежащие изменению, перебираются в некотором, заранее определенном порядке, например $A_{2,1}$, $A_{3,1}$, $A_{3,2}$ и т. д. Второй оператор делает более глобальные изменения, которые в некоторых случаях могут быстрее привести к цели.

3.4. Функции приспособленности

Для генерации тестов к данной задаче, в зависимости от свойств решения, против которого генерировались тесты, применялись различные функции приспособленности.

Решения, имеющие вид, описанный в разд. 3.2.2.1, можно условно разделить на три типа:

1. Запоминание вычисленных значений в ассоциативном массиве. Такие решения используют объем памяти, прямо пропорциональный числу подсчитанных значений. Это число может быть очень большим, кроме этого, операции добавления и поиска в таких структурах, как ассоциативный массив, имеют большой скрытый множитель в асимптотической оценке, поэтому такие решения неэффективны.
2. Запоминание вычисленных значений в массиве небольшого размера. Для решений такого типа массив должен иметь размер, равный степени двойки. Небольшим считается размер массива, не превосходящий 32 мегабайта, так как массивы размером 64 мегабайта и более не умещаются в разрешенном объеме памяти вместе с исполняемым кодом программы. В этом случае в массиве можно запомнить не более 2^{23} значений. Хотя существуют корректные решения такого типа, многие решения, следующие этой стратегии, не укладываются в ограничение по времени, поскольку вынуждены много раз вычислять одни и те же значения, которые они не способны запомнить.
3. Запоминание вычисленных значений в массиве большого размера (64 мегабайт и более). Используя тот факт, что на x86-совместимых архитектурах компьютера для статических массивов память выделяется страницами по четыре килобайта, можно определить в программе массив размером 64 и более мегабайт. При этом, если не все ячейки массива использовались в процессе работы программы, объем использованной памяти, равный суммарному объему задействованных страниц, может быть намного меньше размера массива, и, в частности, может удовлетворять ограничениям задачи. Однако, возможно существование тестов, которые приводят к более плотному заполнению массива и нарушению ограничений на используемую память.

   Для генерации тестов против решений первого и второго типа значе-

ние функции приспособленности равняется числу вызовов главной рекурсивной процедуры или числу элементов, хранящихся в структуре данных, используемой для запоминания значений. Для достаточно сложных тестов это число имеет порядок нескольких десятков или сотен миллионов, в то время как для простых тестов это число не превышает нескольких десятков.

Для генерации тестов против решений третьего типа используемый массив разбивается на участки по одному килобайту, и функция приспособленности равняется числу таких участков, использованных при работе программы. Для решений, не подходящих под вышеуказанные описания, использовался индивидуальный подход к разработке функции приспособленности.

Далее будут приведены примеры решений различных типов с указанием используемой в каждом конкретном случае функции приспособленности. В реализациях решений считается, что знаковый целочисленный тип `long` занимает 64 бит, `edgesOf` — массив 64-битных знаковых чисел, хранящий заданный на входе граф с использованием битового сжатия (`edgesOf[i]` — битовая маска, в которой установлены биты, соответствующие смежным с `i` вершинам).

Процедуры ввода-вывода, объявление глобальных массивов и некоторые другие конструкции языков программирования, не относящиеся к сути решения, в приводимых фрагментах не указываются. Фрагменты приводятся на языке *Java*.

3.4.1. Число вызовов функции

Приведем в качестве примера фрагмент самого простого из возможных решений задачи (лист. 3.1).

Это решение использует перебор с запоминанием (разд. 3.2.2.1). Тест против такого решения найти достаточно просто: на полном графе с N вершинами оно совершает 2^{N-1} вызовов функции, и при $N = 50$ время его

```
long calc(long mask) {
    if (mask == 0) {
        return 1;
    }
    if ((mask & (mask - 1)) == 0) {
        return 2;
    }
    int lastBit = Long.numberOfTrailingZeros(mask);
    long rec1 = mask ^ (1L << lastBit);
    long rec2 = rec1 & edgesOf[lastBit];
    return calc(rec1) + calc(rec2);
}
```

Листинг 3.1. Фрагмент простейшего решения

работы может составлять более 1000 часов. Это решение, однако, можно модифицировать так, чтобы оно запоминало все значения, вычисленные для $mask < 2^{23}$ (лист. 3.2).

```
int[] cache = new int[1 << 23];

long calc(long mask) {
    if (mask == 0) {
        return 1;
    }
    if ((mask & (mask - 1)) == 0) {
        return 2;
    }
    if (mask < cache.length && cache[(int) (mask)] != 0) {
        return cache[(int) (mask)];
    }
    int lastBit = Long.numberOfTrailingZeros(mask);
    long rec1 = mask ^ (1L << lastBit);
    long rec2 = rec1 & edgesOf[lastBit];
    long res = calc(rec1) + calc(rec2);
    if (mask < cache.length) {
        cache[(int) (mask)] = (int) (res);
    }
    return res;
}
```

Листинг 3.2. Фрагмент решения с кешированием младших бит

Назовем это решение *решением с кешированием младших бит*. Для теста в виде полного графа при $N = 50$ число вызовов функции будет приблизительно равно $2^{26} \approx 6,7 \times 10^7$, что при использовании процессора с тактовой частотой порядка 2–3 ГГц занимает менее половины секунды.

Для выращивания теста против описанного решения функция приспособленности принимается равной числу вызовов функции `calc`. На рис. 3.1 показан пример графика такой функции приспособленности. Время

работы решения, соответствующее максимальному значению функции на этом графике, равно 5,25 секунд.

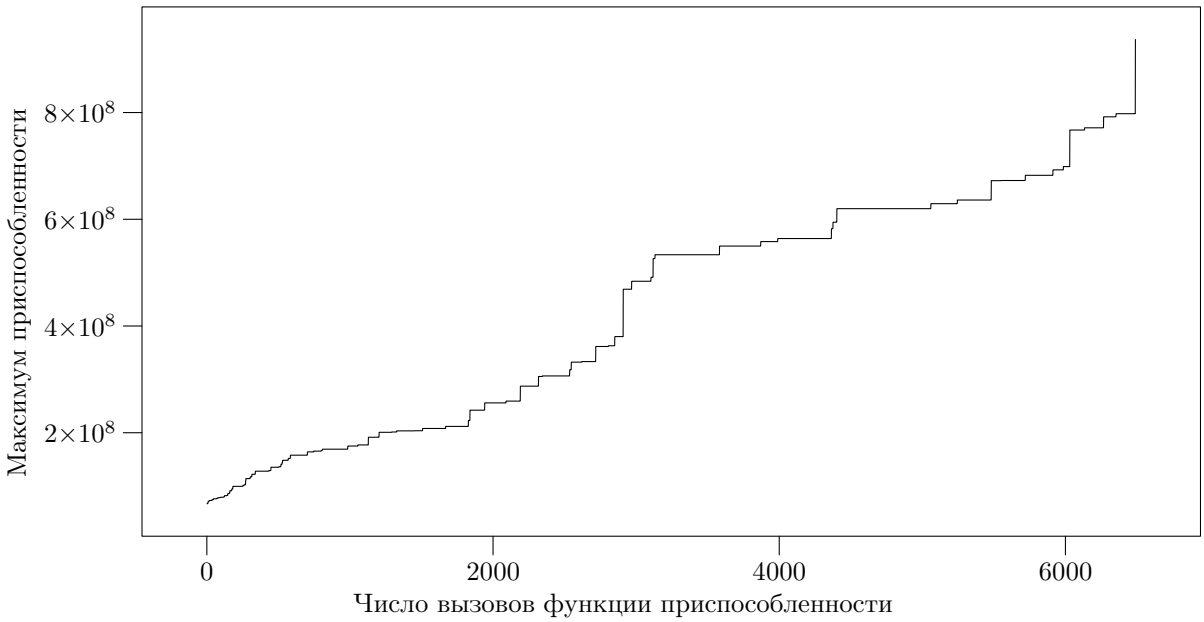

Рис. 3.1. График функции приспособленности для решения с кешированием младших бит

Решение с кешированием младших бит допускает еще одну оптимизацию. А именно, если переменные `rec1` и `rec2` равны, то соответствующее значение можно посчитать лишь один раз (лист. 3.3). В этом случае при максимальном возможном размере кеша (2^{23} ячеек) решение работает меньше двух секунд, тем самым становясь корректным.

```
/* ... */
int lastBit = Long.numberOfTrailingZeros(mask);
long rec1 = mask ^ (1L << lastBit);
long rec2 = rec1 & edgesOf[lastBit];
long res = rec1 == rec2 ? 2 * calc(rec1) : calc(rec1) + calc(rec2);
/* ... */
```

Листинг 3.3. Оптимизация решения с кешированием младших бит

3.4.2. Заполненность структуры данных

Проблема, состоящая в том, что число клик одного и того же подграфа может вычисляться много раз, может быть решена иначе. Будем использовать ассоциативный массив, в котором будем хранить соответствие подграфу (битовой маске) вычисленного для него ответа. При этом мо-

жет использоваться как реализация ассоциативного массива из стандартной библиотеки языка (`std::map` в *C++*, `java.util.HashMap` в *Java*), так и более экономная и быстрая реализация, выполненная автором решения. На лист. 3.4 приведен фрагмент такой программы.

```
/* Open hashing implemented by the participant */
LLOpenHashMap hash = new LLOpenHashMap(4000037);

long calc(long mask) {
    if (mask == 0) {
        return 1;
    }
    if ((mask & (mask - 1)) == 0) {
        return 2;
    }
    long hv = hash.get(mask);
    if (hv != -1) {
        return hv;
    }
    int lastBit = Long.numberOfTrailingZeros(mask);
    long rec1 = mask ^ (1L << lastBit);
    long rec2 = rec1 & edgesOf[lastBit];
    return hash.put(mask, calc(rec1) + calc(rec2));
}
```

Листинг 3.4. Фрагмент решения с использованием ассоциативного массива

Решению такого типа присущ следующий недостаток: так как каждая пара ключ-значение занимает 16 байт (два 64-битных целых числа), то, не превышая ограничения по памяти, возможно запомнить не более четырех миллионов таких пар. В предположении того, что существуют тесты, где этому решению придется использовать больше значений, в качестве функции приспособленности выбирается число пар, хранящихся по окончании работы решения в ассоциативном массиве.

Пример графика функции приспособленности для этого решения приведен на рис. 3.2.

Максимальное время работы решения составило порядка 1,4 секунды, однако использование памяти существенно превысило 64 мегабайта. В зависимости от реализации ассоциативного массива, решения такого типа могут раньше нарушить ограничение по памяти (как в случае описанного решения) или по времени (если использовать, например, `java.util.HashMap<Long, Long>`), а также, в некоторых случаях, может

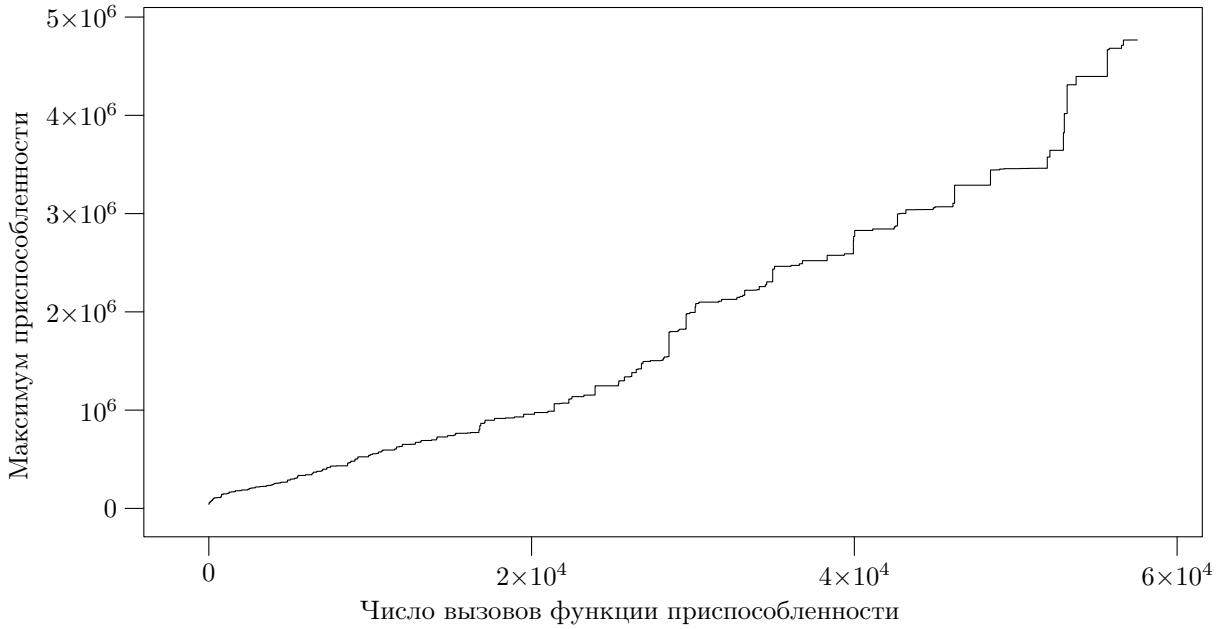

Рис. 3.2. График функции приспособленности для решения с использованием ассоциативного массива

произойти ошибка времени выполнения («падение» решения) или выдача неверного ответа.

3.4.3. Число использованных страниц памяти

Решение с кешированием младших бит (разд. 3.4.1) можно оптимизировать и другим способом. На x86-совместимых платформах (на которых, в основном, и производится тестирование олимпиадных задач) страницы памяти, которые никогда не были прочитаны или записаны, не выделяются операционной системой и поэтому не учитываются при подсчете используемой памяти. Такие страницы, как правило, имеют размер 4 килобайта. По этой причине можно статически выделить очень большой массив (например, 256 мегабайт), и, если не все составляющие его страницы будут использованы, объем используемой памяти будет гораздо меньше (в частности, решение будет укладываться в ограничение по памяти).

Отметим, что это утверждение неверно для языка *Java*, так как в нем все массивы выделяются в динамической памяти и предварительно инициализируются нулями. В целях выращивания теста против решения на языке *C++* можно использовать как оригинальное решение, так и пе-

ревод этого решения на язык *Java* с целью лучшей интеграции с кодом эволюционного алгоритма. При этом решение на *Java* будет всегда использовать больше 64 мегабайт памяти, что не мешает успешному построению теста.

Для такого решения (назовем его *решение с использованием большого массива*) можно ввести функцию приспособленности, равную числу использованных страниц памяти. С учетом возможного отсутствия выравнивания массива по границам страниц, целесообразно проверить (и учесть в функции приспособленности) решение на использование крайних элементов, таких как нулевой и с номером $2^{24} - 1$. Для вычисления функции приспособленности проще всего реализовать класс, эмулирующий массив и одновременно вычисляющий, какие страницы памяти использованы. При этом можно считать не число страниц, а число блоков по четыре килобайта (или даже по одному килобайту для «перестраховки»). Приведем код такого класса (лист. 3.5).

Максимальное значение приспособленности для кеша размером 2^{24} элемента равно 65538. Пример графика функции приспособленности для решения такого типа с кешом размера 2^{24} приведен на рис. 3.3. Максимальное значение функции приспособленности достигнуто примерно на 150 000 поколении. При этом время работы решения на лучшем тесте составило около 80 миллисекунд.

3.5. Результаты эксперимента

Всего на момент начала эксперимента было засчитано 86 решений различных авторов. Для генерации тестов были отобраны 10 решений, которые, по предположению автора работы, представляли собой достаточно репрезентативную выборку некорректных решений. Из них одно решение относилось к третьему типу, большинство остальных решений примерно поровну распределялись между первым и вторым типами. Против одного из отобранных решений не удалось сгенерировать тест, а после более

```
public class CountingArray {
    private int[] data;
    private boolean[] used;
    private boolean used0, usedL;
    public CountingArray(int ilogSize) {
        data = new int[1 << ilogSize];
        used = new boolean[1 << (ilogSize - 8)];
    }
    private void digest(int index) {
        used0 |= index == 0;
        usedL |= index == data.length - 1;
        used[index >>> 8] = true;
    }
    public int get(int index) {
        digest(index);
        return data[index];
    }
    public void set(int index, int value) {
        digest(index);
        data[index] = value;
    }
    public int fitness() {
        int rv = 0;
        rv += used0 ? 1 : 0;
        rv += usedL ? 1 : 0;
        for (boolean v : used) {
            rv += v ? 1 : 0;
        }
        return rv;
    }
}
```

Листинг 3.5. Класс, реализующий подсчет числа используемых страниц

тщательного теоретического анализа реализованного в нем алгоритма это решение было признано корректным. Против остальных решений было сгенерировано в совокупности 10 тестов, получивших номера с 43 по 52.

В табл. 3.1 для каждого теста T приведено число решений, которые прошли все тесты с номерами, меньшими T, и не прошли тест T, включая отдельную статистику для каждого возможного результата прохождения теста.

Таким образом, с помощью описанного метода сгенерированы тесты, позволившие почти полностью устранить имеющиеся на время проведения эксперимента зачтенные неэффективные решения, которых было более 50 %. По состоянию на текущий момент времени, процент зачтенных неэффективных решений составляет не более 20 %, что существенно ниже, чем до использования описанного метода.

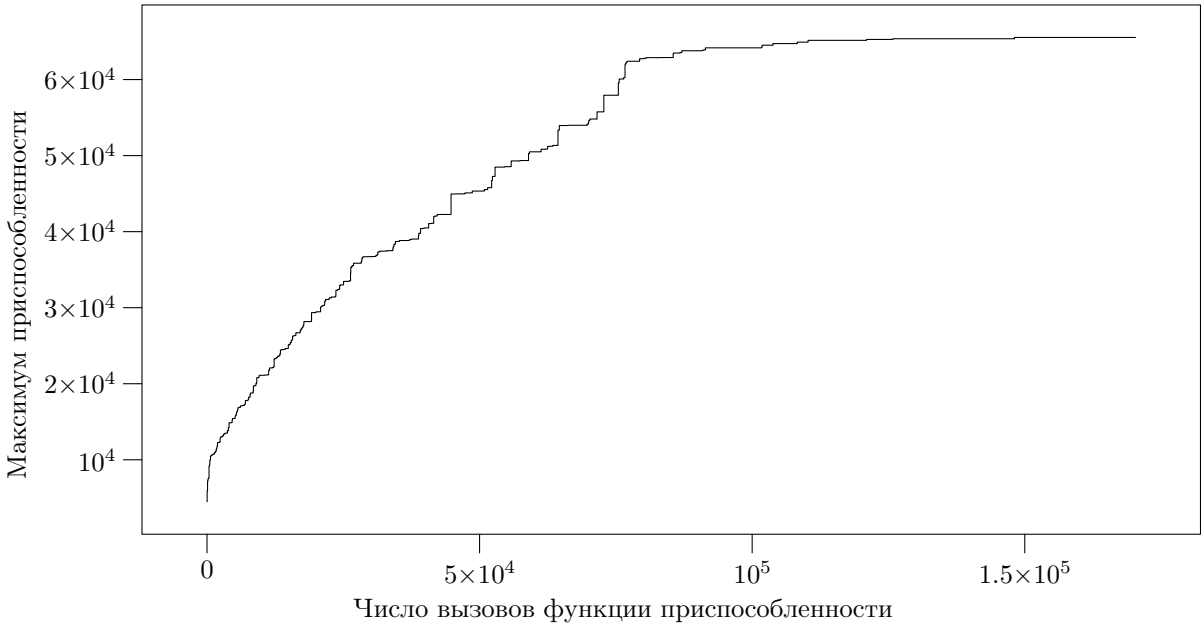

Рис. 3.3. График функции приспособленности для решения с использованием большого массива

Таблица 3.1. Эффективность сгенерированных тестов для задачи «Work for Robots»

№ теста	Число решений, не прошедших тест			
	Все	WA	TL	ML
43	1	0	1	0
44	2	0	2	0
45	2	0	2	0
46	1	0	1	0
47	0	0	0	0
48	9	0	8	1
49	13	0	13	0
50	12	0	12	0
51	2	2	0	0
52	3	0	0	3
Всего	45	2	39	4

Тем не менее, полностью устранить неэффективные решения по этой задаче не получилось. Связано это с тем, что новые решения существенно используют рандомизацию (случайную перестановку номеров вершин), которой в данной задаче пока не получается противопоставить эффективные меры. По некоторым данным, для многих таких решений возможно доказать, что математическое ожидание времени работы и используемой памяти для таких решений удовлетворяет ограничениям на время работы и объем используемой памяти соответственно.

Выводы по главе 3

Приведено формализованное условие задачи «Work for Robots» из интернет-архива *acm.timus.ru*.

Описано авторское решение указанной задачи с доказательством корректности и оценками времени работы и используемой памяти.

Описаны типы неэффективных решений указанной задачи с указанием причины неэффективности. Для каждого типа решений приведен пример такого решения, а также указана используемая для решений такого типа функция приспособленности.

Приведены результаты эксперимента по применению (1 + 1)-эволюционной стратегии для генерации тестов к указанной задаче. Из имевшихся на момент эксперимента 86 решений, проходивших имеющиеся тесты, 45 не прошли 10 вновь сгенерированных тестов. Полученный результат показывает высокую эффективность метода генерации тестов с помощью эволюционных алгоритмов для указанной задачи по теории графов.

Глава 4. Генерация тестов для задачи о поиске максимального потока

В данной главе описывается применение подхода, описанного в главе 2, к генерации тестов для задачи о поиске максимального потока в сети.

4.1. Условие задачи

Задача о поиске максимального потока в сети является классической задачей теории графов [19]. Она формулируется следующим образом:

- дан ориентированный граф с V вершинами и E ребрами;
- среди вершин графа выделены исток s и сток t;
- i-тому ребру приписана *пропускная способность* c_i;
- требуется найти максимальный *поток* — такой набор чисел f_i $(1 \leq i \leq E)$, что:
  - для каждого ребра $f_i \leq c_i$;
  - для каждой вершины, кроме истока и стока, суммарный поток по входящим ребрам равен суммарному потоку по исходящим ребрам;
  - суммарный поток, выходящий из истока, максимален.

Существует множество решений этой задачи. Наиболее известны алгоритмы Форда-Фалкерсона, Эдмондса-Карпа (время работы $O(V \cdot E^2)$), Диница ($O(V^2 \cdot E)$, $O(V\sqrt{E})$ для единичных пропускных способностей), проталкивания предпотока ($O(V^2 \cdot E)$), «поднять-и-в-начало» ($O(V^3)$) [19]. Также, часто применяются модификации имеющихся алгоритмов методом масштабирования потока.

При генерации тестов для олимпиадных задач, основанных на задаче о поиске максимального потока, главную трудность представляет то, что тесты, генерируемые случайным образом или согласно шаблонам, ред-

ко бывают сложными. Так, для алгоритма Эдмондса-Карпа не составляет особого труда пройти тест с 500 вершинами и 10000 ребрами за доли секунды, хотя из оценки времени работы следует, что алгоритм должен произвести порядка 10^{11} элементарных действий.

Далее приводятся результаты экспериментов, показывающие, что описываемый в настоящей работе подход может упростить задачу генерации качественных тестов для задачи о поиске максимального потока.

4.2. Постановка эксперимента

Для проведения экспериментов выбраны следующие ограничения на входные данные: число вершин $V \leq 100$, число ребер $E \leq 5000$, пропускные способности ребер целые и не превышают 10^4. Допускаются петли и кратные ребра. Кроме того, считается, что ребра являются двунаправленными, то есть, каждое ребро может пропускать поток, не превышающий его пропускную способность, в обе стороны. Истоком является вершина с номером 1, стоком — вершина с номером V. Такой выбор ограничений связан с тем, что задача с такими ограничениями дается в летней компьютерной школе по информатике в качестве учебного задания.

4.2.1. Особь эволюционного алгоритма

В качестве особи эволюционного алгоритма была выбрана строка длиной, не превышающей 5000. Каждый символ этой строки является тройкой (s, t, c), где $1 \leq s, t, \leq V$, $0 \leq c \leq 10^4$ и описывает ребро, ведущее из вершины s в вершину t с пропускной способностью c. Особь такого вида задает граф, удовлетворяющий ограничениям задачи.

4.2.2. Эволюционные операторы

Во всех используемых эволюционных алгоритмах применяется оператор мутации. Он состоит в том, что с вероятностью $0,05$ каждый символ

строки заменяется на сгенерированный случайным образом. Для генетических алгоритмов, кроме того, применяется двухточечный оператор скрещивания.

4.2.3. Исследуемые решения задачи о поиске потока

В рамках эксперимента исследовались следующие решения задачи о поиске максимального потока в сети:

- решение, использующее алгоритм Форда-Фалкерсона;
- решение, использующее алгоритм Форда-Фалкерсона с масштабированием.

4.3. Результаты эксперимента

На рис. 4.1 приведены графики функций приспособленности при генерации тестов против решения, использующего алгоритм Форда-Фалкерсона, с использованием различных алгоритмов оптимизации (включая случайную генерацию тестов).

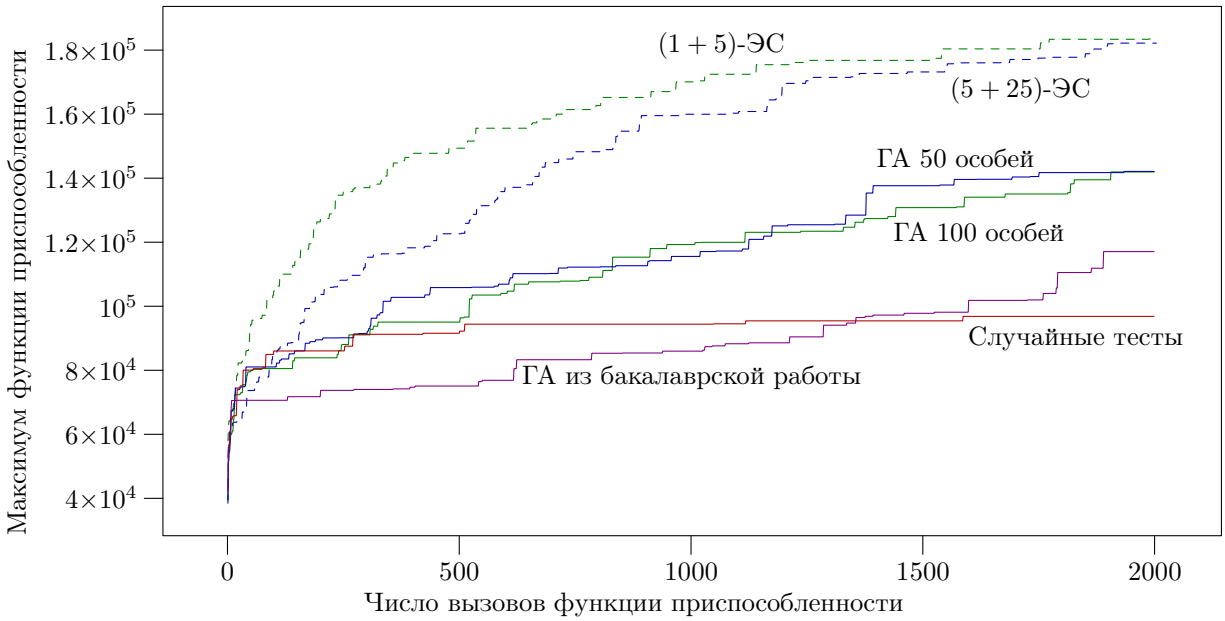

Рис. 4.1. Графики функций приспособленности при использовании различных алгоритмов оптимизации

Помимо случайного поиска использовались эволюционные страте-

гии и генетические алгоритмы, доступные в библиотеке Watchmakers [30], с двумя различными настройками. В качестве функции приспособленности использовалось число вызовов обхода в глубину. Для каждого из алгоритмов график усреднен по пяти запускам. Процесс оптимизации прекращался после 2000 вычислений функции приспособленности.

Из графиков следует, что для данного решения наиболее эффективным является использование $(1 + 5)$-эволюционной стратегии.

На рис. 4.2 приведены графики функций приспособленности при генерации тестов против решения, использующего алгоритм Форда-Фалкерсона с масштабированием. Это решение является более эффективным, чем решение, не использующее масштабирование. Помимо алгоритмов оптимизации, указанных для предыдущего решения, был также применен генетический алгоритм из работы [29]. В качестве функции приспособленности использовалось число вызовов обхода в глубину. Графики усреднялись по пяти запускам, процесс прекращался после 20000 вычислений функции приспособленности.

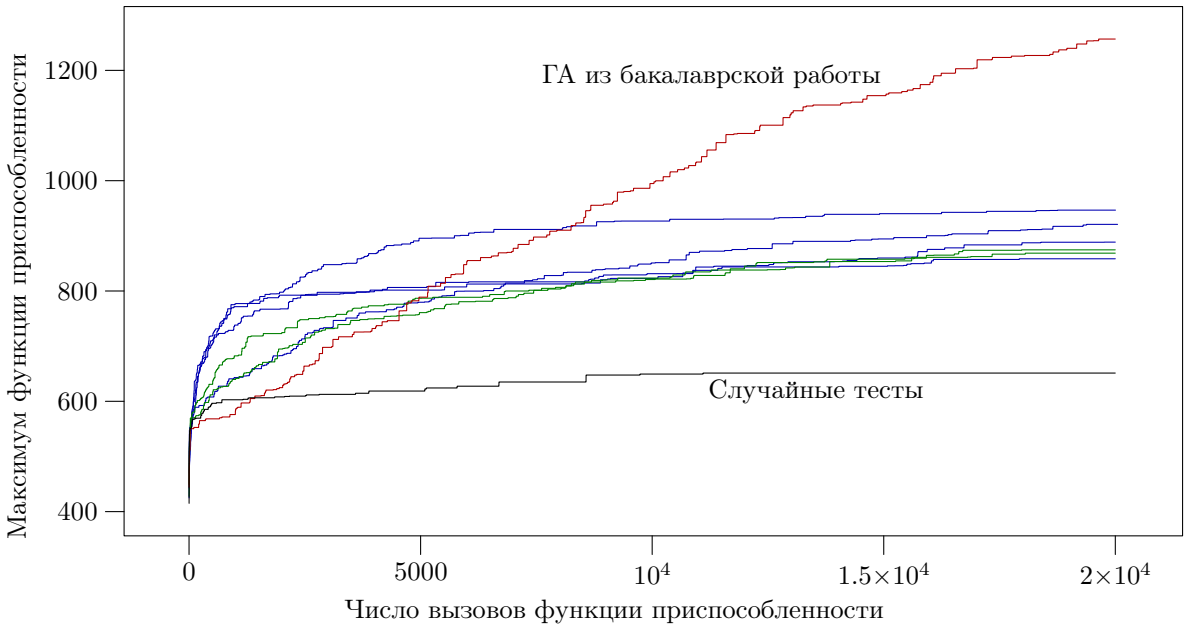

Рис. 4.2. Графики функций приспособленности при использовании различных алгоритмов оптимизации

Из графиков следует, что для решения с масштабированием наиболее эффективным оказывается генетический алгоритм из работы [29],

а генетические алгоритмы и эволюционные стратегии из библиотеки [30] работают примерно с одинаковой эффективностью. В дальнейших экспериментах использовался алгоритм из [29], как наиболее эффективный.

В главе 2 на рис. 2.1 показано, что наиболее эффективными функциями приспособленности для решений, использующих алгоритмы, основанные на поиске в глубину, являются число вызовов поиска в глубину и суммарное число вершин, посещенных всеми поисками в глубину. Однако, по причине предварительного характера экспериментов, отраженных на этом рисунке, число вызовов функции приспособленности было ограничено 25000. На рис. 4.3 приведены графики времени работы решения в процессе оптимизации указанных функций приспособленности в течение более длительного времени.

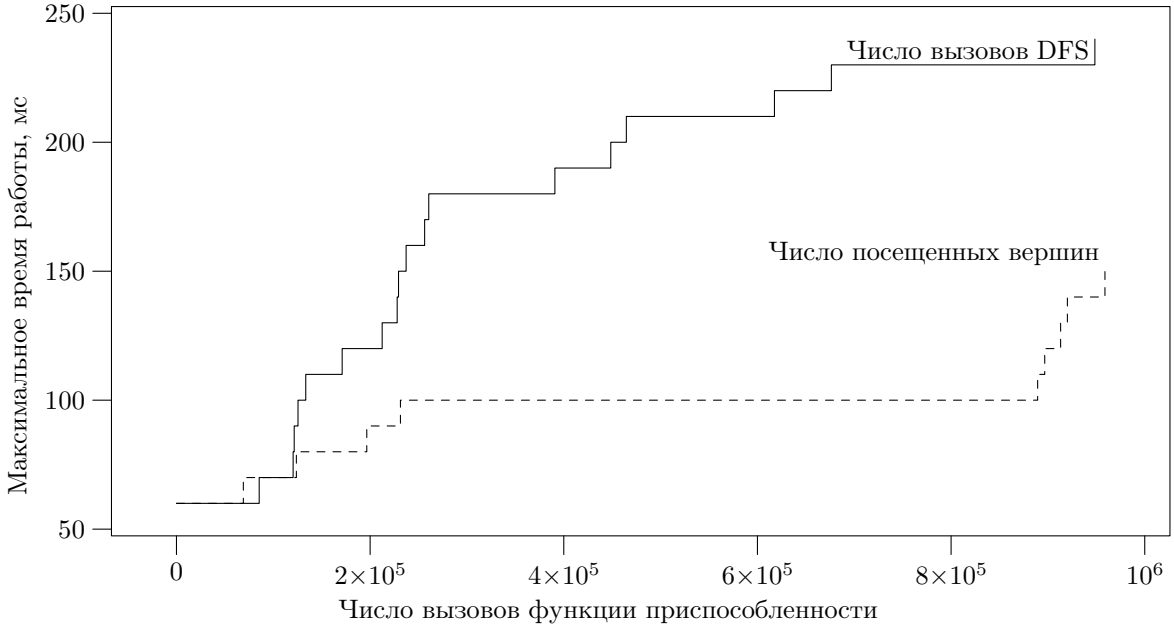

Рис. 4.3. Графики функций приспособленности при использовании различных алгоритмов оптимизации

Из графиков видно, что использование числа вызовов поиска в глубину в качестве функции приспособленности приводит к лучшим результатам. Тесты, генерируемые при использовании этой функции, имеют число вызовов поиска в глубину порядка 4500 и среднее число посещенных вершин на каждой итерации порядка 500. Эти тесты существенно труднее, чем тесты, генерируемые случайным образом, и в несколько раз лучше тестов,

имевшихся в изначальном наборе, который используется при проведении летних компьютерных школ.

Выводы по главе 4

Подход, описанный в главе 2, применен к генерации тестов для задачи о поиске максимального потока в сети. Тесты генерировались против решений на основе алгоритма Форда-Фалкерсона, как использующего масштабирование, так и не использующего его.

Эксперименты показали, что против алгоритма Форда-Фалкерсона, не использующего масштабирование, генерация тестов наиболее эффективна с использованием $(1 + 5)$-эволюционной стратегии. Против алгоритма, использующего масштабирование, наиболее эффективным оказался генетический алгоритм из работы [29]. Наилучшей функцией приспособленности оказалось число вызовов поиска в глубину, произведенных алгоритмом Форда-Фалкерсона.

В ходе экспериментов были сгенерированы тесты, существенно более сложные, чем тесты, генерируемые случайным образом, и в несколько раз более сложные, чем тесты, присутствующие в изначальном тестовом наборе.

Заключение

В работе предложен метод генерации тестов для олимпиадных задач по теории графов с использованием эволюционных алгоритмов.

Предложенным методом были сгенерированы новые тесты к реальной олимпиадной задаче. По результатам тестирования более половины решений, прошедших имеющийся набор тестов, не прошли сгенерированные таким образом тесты.

Метод был также применен к генерации тестов для задачи о поиске максимального потока в сети. При этом были построены тесты, существенно превосходящие случайно сгенерированные тесты и в несколько раз более сложные, чем тесты из набора, используемого при тестировании решений задачи о поиске потока в летней компьютерной школе.

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

- генерация тестов для других классов задач;
- генерация тестов против решений, выдающих неверный ответ;
- генерация тестов, покрывающих весь достижимый код решений жюри;
- улучшение стабильности работы генетических алгоритмов для типов функций приспособленности, время работы которых возрастает вместе со значением функции.

Литература

1. ACM International Collegiate Programming Contest. URL: http://en.wikipedia.org/wiki/ACM_ICPC.
2. International Olympiad in Informatics. URL: http://www.ioinformatics.org.
3. Интернет-олимпиады по информатике. URL: http://neerc.ifmo.ru/school/io/.
4. Правила проведения полуфинала NEERC. URL: http://neerc.ifmo.ru/information/contest-rules.html.
5. *Holland J. P.* Adaptation in Natural and Artificial Systems. University of Michigan, 1975.
6. *Mitchell M.* An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, 1996.
7. *Rechenberg I.* Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Stuttgart: Fromman-Holzboorg Verlag, 1973.
8. *Bäck T.*, *Hoffmeister F.*, *Schwefel H.-P.* A Survey of Evolution Strategies / Proceedings of the Fourth International Conference on Genetic Algorithms. Proceedings of the Fourth International Conference on Genetic Algorithms. Morgan Kauffman, 1991. С. 2 – 9.
9. *Luke S.* Essentials of Metaheuristics. Lulu, 2009.
10. *Koza J. R.* Genetic programming: on the programming of computers by means of natural selection. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-11170-5.
11. *Alander J. T.*, *Mantere T.*, *Turunen P.* Genetic Algorithm Based Software Testing / Artificial Neural Nets and Genetic Algorithms. Artificial Neural Nets and Genetic Algorithms. Wien, Austria: Springer-Verlag, 1998. С. 325 – 328.
12. *Tonella P.* Evolutionary testing of classes / ISSTA. ISSTA. 2004. С. 119 – 128.
13. Задача «Work for Robots». URL: http://acm.timus.ru/problem.aspx?num=1695.
14. TopCoder. URL: http://www.topcoder.com/tc.
15. Google Code Jam. URL: http://code.google.com/codejam.
16. *Оршанский С. А.* О решении олимпиадных задач по программированию формата ACM ICPC // Мир ПК. 2005. № 9.
17. *Акишев И. Р.* Об опыте участия в командных соревнованиях по программированию формата ACM // Методическая газета для учителей «Информатика». 2008. № 19. С. 20 – 28.
18. *Pisinger D.* Algorithms for Knapsack Problems. PhD thesis. University of Copenhagen, 1995.
19. *Т. Кормен*, *Ч. Лейзерсон*, *Р. Ривест.* Алгоритмы. Построение и анализ. Второе издание. М.: Издательский дом «Вильямс», 2005. С. 1296.
20. *Гэри М.*, *Джонсон Д.* Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
21. *Харари Ф.* Теория графов. М.: Едиториал УРСС, 2003.
22. *Timus Online Judge.* Архив задач с проверяющей системой. URL: http://acm.timus.ru.
23. *Дубинин Н. П.* Общая генетика. Наука, 1986. С. 560.
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. № 4598. С. 671 – 680.
26. *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.
27. *Höfler A.* Formoptimierung von Leichtbaufachwerken durch Einsatz einer Evolutionsstrategie. PhD thesis. Technical University of Berlin, 1976.
28. *Schwefel H.-P.* Binäre Optimierung durch somatische Mutation. Tech. rep. Technical University of Berlin и Medical University of Hannover, 1975.

29. *Буздалов М. В.* Генерация тестов для олимпиадных задач по программированию с использованием генетических алгоритмов // Научно-технический вестник СПбГУ ИТМО. 2011. № 2(72). С. 72 – 77.
30. Watchmaker Framework for Evolutionary Computation. URL: `http://watchmaker.uncommons.org`.