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

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Кравцов Никита Олегович

Научный руководитель: ассистент кафедры ТП М. В. Буздалов

Санкт-Петербург
2013

Содержание

Введение

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

Автоматизация процесса построения набора тестов позволит существенно сократить затраты на тестирование. Существует множество подходов к автоматизированному построению набора тестов.

При тестировании некоторых программ, таких как решения олимпиадных задач, нет необходимости проводить модульное тестирование. Для полноценного покрытия тестами олимпиадных задач необходимо генерировать тесты, подходящие под условие задачи. При нахождении теста, на котором программа работает дольше всего, необходимо знать количество действий совершенных программой.

Для того, чтобы определить количество действий программы, написанной на Java, можно вставить счетчики, которые в конце работы программы выдадут, сколько действий было сделано в каждом участке кода программы.

Глава 1. Обзор предметной области

1.1. Задачи

Общая задача заключается в том, чтобы по имеющейся программе решения задачи, найти тест, на котором программа будет выполнять больше всего действий. С помощью эволюционных алгоритмов, мы будем выращивать такие тесты, для которых количество действий будет наибольшим. Счетчики производительности будут сообщать нашему алгоритму, количество действий, выполненных программой.

1.2. Производительность

Это количественная характеристика скорости выполнения определённых операций на компьютере. Производительность можно вычислять:

- вычисляя время программы;
- вычисляя количество действий, сделанных программой;

1.3. Счетчики производительности

Счетчики производительности — переменные в программе, в которых по завершению работы программы, будет храниться число, означающее количество действий какого либо участка кода программы.

1.4. Интеграция счетчиков производительности

Для вычисления количества действий работы программы, необходимо увеличивать счетчик производительности на единицу, после выполненного действия. Нет надобности это делать после каждого действия. Если некоторое место кода выполняется константу раз, то его можно учесть только один раз.

1.5. Java

Java (Ява) — объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Приложения Java обычно компилируются в специальный байт-код, поэтому они могут работать на любой виртуальной Java-машине (JVM) вне зависимости от компьютерной архитектуры.

1.6. Интеграция счетчиков производительности в Java

Наша задача интегрировать счетчики в те места, которые могут выполняться неопределенное количество раз.

1.7. ANTLR

ANTLR –– генератор парсеров, позволяющий автоматически создавать программу-парсер (как и лексический анализатор) на одном из целевых языков программирования (C++, Java, Python, Ruby)[1] по описанию LL(*)-грамматики. В данной работе использовался парсер, сгенерированный на языке Java.

1.8. Выводы по главе 1

Рассмотрена задача по интеграции счетчиков производительности. Описаны определения, связанные с этой областью. Кратко рассмотрен язык программирования, который покрывается счетчиками. Дано определение генератора, с помощью которого счетчики интегрируются.

Глава 2. Описание реализованного подхода

В данной главе описывается разработанный метод интеграции счетчиков производительности в программу, написанную на языке Java.

2.1. Постановка задачи

Целью данной работы является создание автоматической интеграции счетчиков в программу, написанную на языке Java, в удобном для дальнейшего использования виде.

2.2. Общая схема решения

При передаче программе кода исходной задачи, программа вставляет в код исходной программы счетчики производительноси в критические места. Далее, при запуске модифицированной программы счетчики изменяют свое значение. При необходимости их можно вывести в файл.

2.3. Критические места

Счетчики производительности стоит добавлять в места, в которых мы заранее не знаем, сколько они будут исполняться. Такие места в языке Java — это методы и циклы.

2.4. Вставка счетчиков производителности в код программы

Вставка счетчиков производительности осуществляется при помощи ANTLR и программы-парсера для языка Java. С помощью них мы можем найти критические места и добавить в них необходимые команды.

2.5. Тип счетчиков производительности

При выборе типа необходимо учитывать удобство дальнейшей работы и количество максимальных действий, произведенных программой. Так как количество действий может быть достаточно велико, то необходимо взять тип long. Сами счетчики будут представлять собой map, из названия счетчика в количество действий, подсчитанных им. В названии будет отражено в каком методе или цикле был он поставлен и символ $, для того чтобы не было коллизий с уже существующими переменными.

2.6. Выводы по главе 2

В этой главе был рассмотрен способ внедрения счетчиков производительности в программы.

Глава 3. Результаты

В данном разделе описываются результаты экспериментов, демонстрирующих работу предложенного метода интеграции счетчиков производительности.

3.1. Счетчики производительности в программе поиска Гамильтонова пути

Поставим счетчики в программе по поиску Гамильтонова пути на одном и том же тесте. Вначале рассмотрим программу, превышающую лимит по времени, а затем корректную.

3.2. Счетчики производительности

3.2.1. В некорректной программе

counter_void_solve$3 : 1
counter_for_counter_void_solve$3$0 : 101
counter_while_counter_String_nextToken$0$0 : 1042
counter_void_main$2 : 0
counter_int_nextInt$1 : 2084
counter_for_counter_void_solve$3$4 : 8
counter_for_counter_void_solve$3$3 : 100
counter_for_counter_void_solve$3$2 : 1082640
counter_for_counter_void_solve$3$1 : 1041
counter_for_counter_int_dfs$4$1 : 16656
counter_for_counter_int_dfs$4$0 : 2082
counter_for_counter_void_solve$3$5 : 8
counter_void_run$5 : 1
counter_String_nextToken$0 : 2084
counter_int_dfs$4 : 101

3.2.2. Счетчики производительности в корректной программе

counter_for_counter_void_topSort$5$0 : 1041
counter_void_solve$3 : 1
counter_for_counter_void_countPathsFrom$4$0 : 1041
counter_for_counter_void_solve$3$0 : 101
counter_while_counter_String_nextToken$0$0 : 1042
counter_void_main$2 : 0
counter_int_nextInt$1 : 2084
counter_for_counter_void_solve$3$4 : 101
counter_void_topSort$5 : 100
counter_for_counter_void_solve$3$3 : 100
counter_for_counter_void_solve$3$2 : 101
counter_for_counter_void_solve$3$1 : 1041
counter_void_run$6 : 1
counter_String_nextToken$0 : 2084
counter_void_countPathsFrom$4 : 100

3.3. Выводы по результатам

Можно заметить, что в первой программе счетчик counter_for_counter_void_solve$3$2 выполнился за 1082640, что приблизительно является квадратом от входных данных, в то время как ни один счетчик в корректной программе не подсчитал больше чем размер входных данных, умноженное на константу.

3.4. Выводы по главе 3

Были проведены эксперименты, демонстрирующие работу предложенного подхода для интеграции счетчиков производительности в программу. В результате экспериментов, проведенных на задаче, можно было

видеть, что предложенный метод успешно справляется с подсчетом количества действий.

Заключение

В данной работе создана платформа для интеграции счетчиков производительности в программы на языке Java.

Предложенный метод был успешно протестирован на задача по поиску Гамильтонова пути. Было проведено сравнение на корректной и некорректной программах.

Таким образом, данная работа полностью удовлетворяет поставленным требованиям.