Минобрнауки России

Федеральное государственное автономное образовательное учреждение высшего образования   
«Национальный исследовательский университет   
«Московский институт электронной техники»

Институт cистемной и программной инженерии и информационных технологий

(СПИНТех)

Миронов Николай Сергеевич

Магистерская диссертация   
по направлению 09.04.04 «Программная инженерия»

Разработка и программная реализация методики оценки временных затрат в системах учета энергопотребления

Студент Миронов Н.С.

Руководитель,   
{д.т.н., профессор } Портнов Е.М.

Москва 2019

## Содержание

Оглавление

[Содержание 2](#_Toc11863195)

[Список сокращений 4](#_Toc11863196)

[Введение 5](#_Toc11863197)

[Актуальность исследования 5](#_Toc11863198)

[Объект исследования 6](#_Toc11863199)

[Предмет исследования 6](#_Toc11863200)

[Цели и задачи исследования 6](#_Toc11863201)

[Положения, выносимые на защиту 7](#_Toc11863202)

[Глава 1. Обзор методов оценки временных затрат исполнения программ 9](#_Toc11863203)

[1.1 Общие положения ОВЗИП 9](#_Toc11863204)

[1.2 Динамическая ОВЗИП 9](#_Toc11863205)

[1.3 Статическая ОВЗИП 13](#_Toc11863206)

[1.4 Преобразование ДКА в РВ 26](#_Toc11863207)

[Выводы по главе 1 35](#_Toc11863208)

[Глава 2. Система статической ОВЗИП на основе вероятностного подхода 37](#_Toc11863209)

[2.1 Формальное описание метода ОВЗИП на основе вероятностного подхода 37](#_Toc11863210)

[2.2 Архитектура системы 49](#_Toc11863211)

[Выводы по главе 2 56](#_Toc11863212)

[Глава 3. Верификация результатов исследования и практические рекомендации 57](#_Toc11863213)

[3.1 Особенности динамической оценки времени исполнения программ для JVM 57](#_Toc11863214)

[3.2 Экспериментальная проверка точности 62](#_Toc11863215)

[3.3 Экспериментальная проверка производительности 68](#_Toc11863216)

[3.4 Экспериментальная оценка потребления памяти 77](#_Toc11863217)

[3.5 Практические рекомендации 81](#_Toc11863218)

[Выводы по главе 3 83](#_Toc11863219)

[Заключение 85](#_Toc11863220)

[Список литературы 86](#_Toc11863221)

[Приложение 91](#_Toc11863222)

# Список сокращений

ДКА – детерминированный конечный автомат

ДОВИ – динамическая оценка времени исполнения (программы или участка кода).

ОВЗИП – оценка временных затрат на исполнение программы.

СОВИ – статическая оценка времени исполнения (программы или участка кода).

ГПУ – граф потока управления. РВ – регулярное выражение.

WCET – Worst Case Execution Time (время исполнения в худшем случае)

# Введение

### Актуальность исследования

Анализ алгоритмов и их программных реализаций имеет очень большое значение в программировании, поскольку, как правило, существует несколько различных способов (алгоритмов) решения конкретной задачи, и требуется знать, какой из них является более приемлемым по времени вы- полнения, по затратам памяти или сложности реализации.

Особый интерес вызывает именно оценка затрат времени на исполне- ние алгоритма, одной из причин тому является широкое применение распре- деленных вычислений. В подобных распределенных системах крайне важно наличие эффективного способа планирования внутри сети. Например, в сети добровольных вычислений, построенных на базе платформы BOINC, от того, насколько быстро выполняются задания, зависит их количество, которое бу- дет выслано при следующем обращении к серверу [1]. Также возможной об- ластью применения результатов прогнозирования времени выполнения про- грамм являются вычислительные системы реального времени. Такие систе- мы, как правило, участвуют в управлении некоторым реальным процессом и должны обладать одним чрезвычайно важным свойством – вычислитель дол- жен не только правильно реализовывать алгоритм, но обязан делать это в за- данный отрезок времени. Система, которая выполняет задачу позже указанно- го директивного времени, считается работающей неверно. Поэтому для раз- работчиков таких систем возможность получения данных о времени выпол- нения программ является необходимым условием на этапе проектирования.

Существует подход к оценке времени исполнения, подразумевающий наблюдение за выполнением программы, сбор трасс, построение программ- ных и аппаратных эмуляторов. Таким образом, он всегда связан с исследова- нием динамики исполнения программы. Однако его применение не всегда приемлемо из-за возможного отсутствия реального вычислителя, дорого- визны и трудоемкости процесса реализации эмулятора исследуемого вычис-

лителя и значительного замедления (10-1000 раз) выполнения программы из- за накладных расходов при измерении.

Существует также немало методов и алгоритмов статической (основы- вающейся на данных, которые могут быть получены из исходного кода про- граммы) оценки времени выполнения. Некоторые из них дают недостаточно точные результаты, так как основываются на приближенном моделировании работы программы. Другие же ориентированы на получение оценки для наи- худшего случая и заточены под конкретную аппаратную платформу (см. например, [2], [3]), что позволяет получать более точные результаты, учиты- вая особенности конкретной архитектуры (например, наличие конвейера, кэш-памяти), однако влечет за собой необходимость добавления в систему модулей, отвечающих за моделирование работы аппаратной составляющей [3][4]. Указанные особенности необходимы при построении систем жесткого реального времени. Однако такие ограничения могут оказаться чрезмерными для систем мягкого реального времени или при планировании внутри сетей распределенных вычислений. В таком случае требуется оценивать среднее (ожидаемое) время исполнения программы.

### Объект исследования

Объектом исследования диссертационной работы является оценка вре- менных затрат на исполнение программ.

### Предмет исследования

Предметом исследования диссертационной работы является метод ста- тической оценки временных затрат на исполнение программ на основе вероятностного подхода.

### Цели и задачи исследования

Целью данной работы является получение оценки математического ожидания и дисперсии временных затрат на исполнение программы без необходимости непосредственного исполнения программы на вычислителе.

В соответствии с целью и предметом исследования в диссертации необ- ходимо решить следующие задачи:

* детально исследовать существующие методы и алгоритмы оценки

времени исполнения программ с целью выявления их преимуществ и недо- статков;

* произвести формализованную постановку задачи ОВЗИП;
* разработать метод статической оценки ожидаемого времени исполне- ния программ;
* разработать архитектуру системы оценки времени исполнения про-

грамм с ориентацией на слабую связность компонент системы, что позволит повысить применимость модуля, отвечающего за непосредственное вычисле- ние времени выполнения программы по ее абстрактному представлению;

* программно реализовать модуль вычисления ожидаемого времени

выполнения программ в соответствии с разработанным методом;

* верифицировать результаты (оценки времени исполнения программ),

получаемые при применении разработанного метода, на предмет корректно- сти и приемлемости.

### Положения, выносимые на защиту

1. Формализованное представление задачи статической ОВЗИП на основе вероятностного подхода и графового представления программы определяет выбор алгоритма оценки времени исполнения программ.
2. Алгоритм построения регулярного выражения по графу потока управления программы позволяет сохранить необходимую информацию о по- токе.
3. Оценка математического ожидания и дисперсии программы могут быть получены по регулярному выражению, описывающему все возможные пути из начальной вершины ГПУ этой программы в конечную.
4. Использование транзитивного замыкания и запоминание промежу- точных результатов позволяют ускорить процесс построения РВ по ГПУ в 5,7

и 2155,5 раз соответственно.

1. Экспериментальная проверка показывает, что отклонение получае- мой оценки математического ожидания от эталонного значения составляет от 1,3% до 6,2%. Отклонение же оценки дисперсии составляет 5%-110%.

## Глава 1. Обзор методов оценки временных затрат исполнения программ

### Общие положения ОВЗИП

Под оценкой времени выполнения обычно понимается одна из альтер- натив:

* + - измерение определенных параметров функционирования вычисли-

тельной системы на этапе ее работы;

* + - предсказание времени выполнения программ без непосредственного исполнения на исследуемом вычислителе.

Первый подход обычно подразумевает наблюдение за выполнением программы, сбор трасс, построение программных и аппаратных эмуляторов [5][6]. Таким образом, он всегда связан с исследованием динамики исполне- ния программы. Однако его применение не всегда возможно по следующим причинам:

* + - отсутствие реального вычислителя;
    - реализация эмулятора исследуемого вычислителя является дорого- стоящим и трудоемким процессом;
    - значительное замедление (10-1000 раз) выполнения программы из-за накладных расходов при измерении.

Предсказание же (или прогнозирование) времени выполнения програм- мы чаще всего предполагает использование разнообразных статических мето- дов анализа кода.

### Динамическая ОВЗИП

Под динамической ОВЗИП мы будем понимать измерение времени затрат выполнения структурных единиц программы (функций, многократно выполняемых циклов и т. д.), или всей программы целиком путем непосредственного выполнения на вычислителе.

Динамической ОВЗИП присущи свои проблемы: непостоянство времени выполнения, проблема «второго прохода», необходимость учета наведенных эффектов [7].

* + 1. Непостоянства времени выполнения

При динамической ОВЗИП, результаты измерений времени выполнения варьируются от прогона к прогону, порой серьезно отличаясь один от другого.

Причин такого непостоянства существует, по меньшей мере, две:

* + - * программное непостоянство, связанное с тем, что в многозадачных

операционных системах (в частности в Windows) исследуемая программа по- падает под влияние чрезвычайно изменчивой окружающей среды;

* + - * аппаратное непостоянство, вызванное внутренней «многозадачно-

стью» аппаратной составляющей.

*Программное непостоянство*

В многозадачной среде, например в операционной системе Windows, никакая программа не владеет всеми ресурсами системы единолично, а разде- ляет их с остальными задачами. Это значит, что скорость выполнения профи- лируемой программы не постоянна и находится в тесной зависимости от

«окружающей среды». На практике разброс результатов измерений может до- стигать 10%-15% [7], особенно, если параллельно с профилировкой исполня- ются интенсивно нагружающие систему задачи.

Однако данная проблема практически не присуща для систем реального времени, где исследуемая программа выполняется с максимальным приорите- том.

*Аппаратное непостоянство*

На аппаратном уровне время выполнения одних и тех же операций так- же не всегда постоянно и подвержено определенным разбросам, под час очень большим и значительно превосходящим программную погрешность. И если последнюю возможно ликвидировать (например, запустить программу в однозадачном режиме или на ОС реального времени), то аппаратное непосто- янство неустранимо принципиально.

Причин аппаратного непостоянства может быть множество, например, одна из них: если частота системной шины не совпадает с частотой модулей оперативной памяти, то чипсету придется каждый раз выжидать случайный промежуток времени до прихода следующего фронта тактового импульса. Ис- ходя из того, что один цикл пакетного обмена в зависимости от типа установ- ленных микросхем памяти занимает от 5 до 9 тактов, а синхронизировать приходится и его начало, и его конец, то нетрудно подсчитать, что в худшем случае мы получим разброс в 25%-40% [7].

* + 1. Обработка результатов измерений

Непосредственные результаты измерений времени исполнения про- граммы в «сыром» виде малопригодны. Очевидно, перед использованием их следует обработать.

Один из возможных методов обработки приведен в [7]. Фактически, предлагается отталкиваться от среднеминимального времени исполнения. Сам алгоритм имеет следующий вид:

* + - * производится *N* повторов программы;
      * затем отбрасывается *N* /3 максимальных и *N* /3 минимальных ре- зультатов замеров;
      * для оставшихся *N* /3 замеров находится среднее арифметическое, ко- торое и является основным результатом.

Величина *N* варьируется в зависимости от конкретной ситуации, но обычно выбирается из интервала 9-12 повторов.

Однако подобный подход больше подходит для оценки времени выпол- нения программы с целью ее дальнейшей оптимизации, нежели для оценки времени выполнения в реальных условиях.

* + 1. Проблема «второго прохода»

Итак, для достижения приемлемой точности измерений исследуемое приложение следует выполнить, по крайней мере, 9-12 раз, причем каждый

прогон должен осуществляться в идентичных условиях окружения. Однако без написания полноценного эмулятора всей системы это требование практи- чески невыполнимо. Дисковый кэш, сверхоперативная память, буферы физи- ческих страниц и история переходов чрезвычайно затрудняют измерение вре- мени выполнения программ, поскольку при повторных прогонах время вы- полнения программы значительно сокращается.

Если исследуется многократно выполняемый цикл, то этим обстоятель- ством можно пренебречь, поскольку время загрузки данных и/или кода в кэш практически не сказывается на общем времени выполнения цикла [7]. Однако так бывает далеко не всегда.

Для очистки кэша данных достаточно считать очень большой блок па- мяти, намного превышающий емкость кэша. Помимо этого можно также и за- писать большой блок данных для выгрузки всех буферов записи. Данное дей- ствие также очистит и TLB (Translate Look aside Buffer) – буфер, хранящий атрибуты страниц памяти для быстрого обращения к ним. Аналогичным об- разом очищается и кэш/TLB команд. Достаточно сгенерировать очень большую функцию, имеющую размер порядка 1-4 Мбайт, выполняющую ма- шинные операции NOP. Таким образом удастся уменьшить влияние всех, перечисленных выше эффектов, хотя и не устранить его полностью.

С другой стороны, если исследуется, отдельно взятая функция, то ее первый прогон нам ничего не даст, поскольку в данном случае основным фактором, определяющим производительность, окажется не эффективность кода/алгоритма самой функции, а накладные расходы на загрузку машинных инструкций в кэш, выделение и проецирование страниц операционной си- стемой, загрузку обрабатываемых функцией данных в сверхоперативную па- мять и т. д. В реальной же программе эти накладные расходы, как правило, уже устранены (даже если эта функция вызывается однократно).

В ходе некоторых экспериментов было показано, что устойчивая повто- ряемость результатов начинается лишь с третьего прогона [7]. Причина

медленного выполнения первого прогона ясна – загрузка данных в кэш и про- чее. Причина же медленного выполнения второго прогона – ветвления. За первый прогон алгоритм динамического предсказания ветвлений еще не успевает накопить достаточное количество информации и потому во втором прогоне еще работает достаточно неэффективно.

Таким образом, при профилировании многократно выполняющихся функций, результаты первых двух или трех прогонов стоит откинуть [7], и ка- тегорически не следует их арифметически усреднять.

Напротив, при профилировании функций, исполняющихся в реальной программе всего один раз, следует обращать внимание лишь на время перво- го прогона и отбрасывать все остальные, причем, при последующих запусках программы необходимо каким-либо образом очистить все кэши и все буферы, искажающие реальную производительность.

### Статическая ОВЗИП

* + 1. Этапы статической ОВЗИП

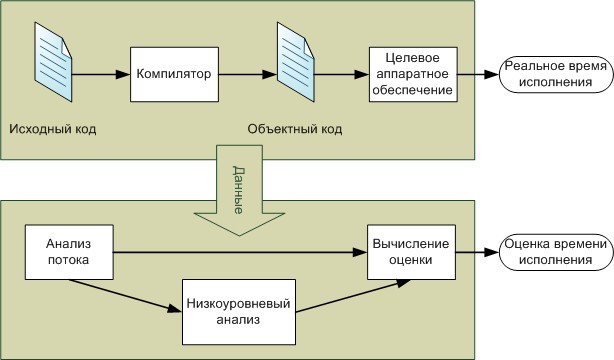
Время исполнения программы зависит от целого ряда факторов, как продемонстрировано на рисунке [1.1](#_bookmark10).

Рисунок 1.1 - Компоненты статического анализа временных затрат на исполнение программного кода

Программный код определяет набор инструкций и путей выполнения, компилятор преобразует высокоуровневый исходный код в семантически эк- вивалентный объектный код. В свою очередь объектный код выполняет на це- левом аппаратном обеспечении.

Статическая ОВЗИП обычно подразделяется на три этапа, каждый из которых тесно связан с отдельными факторами, влияющими на время исполнения программы:

* *анализ потока управления* производит анализ исходного, промежу-

точного или объектного кода программы и выявляет возможные потоки вы- полнения в программе, т. е. возможные последовательности инструкций, ко- торые могут быть выполнены;

* *низкоуровневый анализ* на основе объектного кода и параметров целе-

вого аппаратного обеспечения определяет временные характеристики выпол- няемых инструкций; для современных процессоров крайне важно учитывать эффекты, оказываемые различными методиками увеличения производитель- ности, такими как кэширование и конвейеризация;

* *этап расчетов* комбинирует информацию, полученную на предыду-

щих шагах и на ее основе производит непосредственную оценку времени ис- полнения программы.

Приведенное разделение на этапы является концептуальным и многие группы исследователей придерживаются его. Некоторые исследователи сов- мещают несколько этапов в один. Иногда же напротив некоторые этапы до- полнительно подразделяются на шаги, например, низкоуровневый анализ мо- жет быть разделен на шаги, рассматривающие каждую из особенностей аппа- ратного обеспечения в отдельности.

* + 1. Анализ потока управления

Целью данного этапа является выявление возможных потоков выполне- ния инструкций в программе, т. е. динамического поведения программы. Ре- зультатом его является информация о том, какие функции вызываются, сколь-

ко происходит итераций циклов, существуют ли зависимости между условны- ми операторами и так далее. Поскольку данная задача неразрешима в общем случае (она эквивалентна широко известной проблеме останова, которая за- ключается в том, что невозможно построить алгоритм, который бы для любой программы смог определить, остановится она или нет), обычно ограничива- ются более простым приближенным анализом. В случае оценки времени ис- полнения в худшем случае (WCET-анализ) накладывается дополнительное ограничение: получаемая информация должна быть безопасна, т. е. все воз- можные исполнения программы должны гарантированно покрываться полу- чаемыми оценками. Но в любом случае, чтобы быть пригодной на практике, получаемая информация должна быть достаточно точной.

Информация о потоке управления может быть извлечена из исходного или объектного кода программы и может быть дополнена данными, собран- ными в ходе компиляции. Этап анализа потока управления может быть подразделен на три фазы:

* + - 1. *Извлечение потока*. Получение информации о потоке управления из вручную добавленных в код аннотаций или же посредством методов автома- тического анализа потока управления.
      2. *Представление потока*. Представление результатов извлечения ин- формации о потоке управления (возможно также комбинация результатов нескольких различных методов) в виде, пригодном для дальнейшей обра- ботки.
      3. *Преобразование для этапа расчетов*. Преобразование полученной информации к виду, необходимому для последнего этапа ОВЗИП.

*Извлечение потока*

Автоматический анализ потока управления – это совокупность методов, служащих для получения информации о потоке управления из кода програм- мы без необходимости вмешательства человека. Различные подходы имеют различную сложность, позволяют получить различное количество информа-

ции и предназначены для обработки программ различного уровня сложности. Для сложных программ иногда очень непросто (или даже невозможно) полу- чить требуемую информацию о потоке, поэтому большинство методов авто- матического анализа потока необходимо дополнять аннотациями, добавляе- мыми в код вручную.

Исследователи разработали методы автоматического анализа для выяв- ления участков мертвого кода (участки кода, которые допустимы в статиче- ской структуре программы, но не являются возможными, если в учет берется семантика кода), а также верхних границ количества итераций циклов.

Чапман и др. использовали символьное выполнение, т. е. выполнение программы с использованием символьных выражений вдобавок к конкрет- ным значениям, для SPARK Ada с целью извлечения информации о потоке программы [8]. Данный подход позволяет выявить некоторые участки мертво- го кода, но для циклов требуется предоставление аннотаций вручную.

Альтенбенд и Стапперт предложили использовать символьное выполне- ние на уровне исходного кода для получения информации о потоке [9]. Ука- занный исходный код – подмножество языка C. Данный подход, также как и предыдущий способен выявлять некоторые участки мертвого кода.

Колин и др. использовали символьное выполнение для вычисления ко- личества итераций внутренних циклов, которое зависит от переменных-счет- чиков внешних циклов [10]. Однако исходные символьные формулы должны быть указаны вручную. Лю и Гомез применили символьные вычисления для функционального языка с целью нахождения исполняемых путей [11].

Хили и др. использовали анализ потока данных и специальные алгорит- мы для того, чтобы автоматически определять верхнюю и нижнюю границы циклов нескольких типов [12]. Посредством предоставляемой пользователем информации об инвариантах циклов эти границы могли быть значительно су- жены. Они также представили метод, который позволяет находить информа- цию о путях выполнения внутри циклов.

Холсти и др. использовали арифметику Пресбургера для вычисления границ циклов со счетчиками, анализируя программу на уровне объектного кода [13]. Данный подход позволяет также использовать пользовательски ан- нотации в целях улучшения результатов автоматического анализа.

Жерлек и др. представили метод для синтаксической идентификации определенных классов индукционных переменных в циклах [14]. Подобная классификация может служить для выявления нижних и верхних границ цик- лов.

*Представление потока*

Извлеченная информация должна быть перенесена на некоторое пред- ставление программы. Существует несколько таких представлений:

* графовое;
* в виде синтаксического дерева;
* в виде системы линейных ограничений.

Эти представления могут соответствовать исходному, промежуточному или объектному коду программы.

Использование представления в виде синтаксического дерева довольно интуитивно, однако связано со сложностями при работе с неструктуриро- ванным, оптимизированным кодом. В то же время, представление в виде гра- фа или системы линейных ограничений являются достаточно мощными и гибкими инструментами для работы с информацией о потоке.

*Проблема отображения*

Непростым вопросом является выбор уровня кода, для которого будет производиться представление и анализ потока. Информация о потоке может быть представлена для исходного, промежуточного или объектного кода. Ав- томатический анализ кода, вероятно, проще проводить на уровне исходного или промежуточного кода, поскольку переменные и иные исследуемые сущ- ности сложно идентифицировать в объектном коде. Кроме того, для конечно-

го пользователя инструмента ОВЗИП ручные аннотации определенно проще предоставлять для исходного кода программы.

Однако если информация предоставлена для уровня исходного кода, то она должна каким-то образом быть отображена на объектный код. В присут- ствии оптимизирующих компиляторов эта задача становится нетривиальной. Причиной тому служат преобразования, такие как развертывание циклов, встраивание функций и другие, производимые такими компиляторами.

В случае WCET-анализа данная проблема является принципиальной, однако при оценке среднего времени ею можно пренебречь.

* + 1. Низкоуровневый анализ

Целью низкоуровневого анализа является определение времени выпол- нения для каждой атомарной составляющей потока (в качестве таких состав- ляющих могут выступать, например, инструкции, базовые блоки или даже бо- лее длинные пути исполнения) с учетом архитектурных особенностей целе- вого аппаратного обеспечения. Для получения реальных временных характе- ристик программы необходимо проводить анализ на уровне объектного кода.

Низкоуровневый анализ может быть подразделен на две фазы:

* глобальный низкоуровневый анализ, определяющий влияние ма-

шинно-зависимых факторов, которые требуют моделирования на уровне всей программы (примерами могут служить кэши команд, кэши данных, предска- затели ветвей);

* локальный низкоуровневый анализ, определяющий влияние ма-

шинно-зависимых факторов, которые могут быть рассмотрены в масштабах нескольких соседних инструкций (примером может служить скорость досту- па к памяти).

Некоторые исследователи объединяют эти фазы в одну.

* + 1. Методы оценки временных затрат исполнения

Существует три основных класса методов, описываемых в литературе

[15]:

* структурный метод,
* метод явного перечисления путей,
* метод неявного перечисления путей (IPET, англ. Implicit Path

Enumeration Technique).

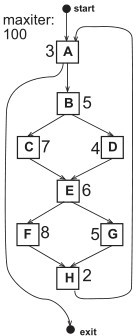
На рисунке [1.2](#_bookmark14) приведен пример графа потока управления с временны- ми метками, указанными для узлов, и верхней границей цикла. Далее на при- мере данного графа будут описаны принципы работы перечисленных мето- дов.

Рисунок 1.2 - Граф потока управления с метками времени

*Структурный метод*

В структурном подходе оценка времени исполнения получается при об- ходе снизу-вверх дерева, соответствующего дереву синтаксического разбора программы. Синтаксическое дерево – это представление программы, его узлы представляют из себя структуры программы (такие как последовательность инструкций, циклы и операции ветвления), а листья – базовые блоки. Суще- ствуют правила, которые применяются при обходе каждого узла дерева и поз- воляют составить уравнение, выражающее временные характеристики данно-

го узла на основе характеристик его дочерних узлов. Данный метод концеп- туально прост и довольно дешев в плане вычислений. Однако он не позволяет обрабатывать сложную информацию о потоке, поскольку все вычисления производятся в рамках одного программного блока и не дают возможности учитывать зависимости между различными блоками. Кроме того, данный ме- тод не применим для неструктурированного и оптимизированного кода, по- скольку в таком случае построение синтаксического дерева и правил преоб- разования становится сложной задачей.

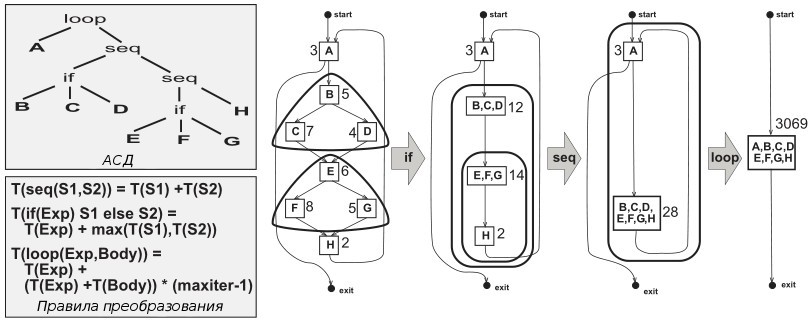
Рисунок [1.3](#_bookmark15) иллюстрирует, как структурный метод применялся бы для указанного на нем синтаксического дерева и правил преобразования. Несколько узлов «схлопываются» в один, давая при этом оценку времени для него.

Рисунок 1.3 - Структурный метод вычисления WCET

Впервые структурный метод был описан Парком и Шоу [16]. Предло- женный метод работал на уровне исходного кода программы. Лим и др. позже расширили данный метод путем учета конвейеризации и кэширования [17]. Метод, который предложил Чапманом, также строится на основе расчетов по синтаксическому дереву [18].

*Метод явного перечисления путей*

В методе явного перечисления путей верхняя граница для всей задачи определяется посредством вычисления верхних границ для всевозможных пу-

тей в этой задаче, после чего из них выбирается максимальная. Определяю- щей особенностью этого метода является то, что возможные пути исполнения представляются явно. Такой подход естественен в пределах одной итерации цикла, но имеет проблемы с увеличением информации о потоке с ростом уровня вложенности циклов. Количество путей растет экспоненциально с ро- стом количества точек ветвления, так что может потребоваться применение эвристических методов поиска.

На рисунке [1.4](#_bookmark16) иллюстрируется, как метод явного перечисления путей применялся бы для графа, приведенного на рисунке [1.2](#_bookmark14). Вначале распознает- ся цикл в графе, после чего находится самый длинный путь в пределах цикла. Время исполнения самого длинного пути в сочетании с информацией о мак- симальном количестве итераций цикла дает верхнюю границу для задачи в целом. Метод явного перечисления путей способен учитывать некоторые за- висимости между инструкциями в пределах одного пути. Однако он плохо применим для неструктурированного кода, поскольку в таком случае очень сложно выявить участки, представляющие из себя итерацию цикла.

Хили и др. выявляют самый длинный путь в циклах и функциях, рассматривая каждый уровень вложенности циклов в отдельности [19]. При этом информация о потоке может использоваться для урезания количества возможных путей, а также для ограничения количества возможных выполне- ний отдельного пути.

Эрмедал представил эффективный метод, основанный на методе явного перечисления путей и способный учитывать комплексную информацию о по- токе и временных эффектах между нелокальными частями программы [15].

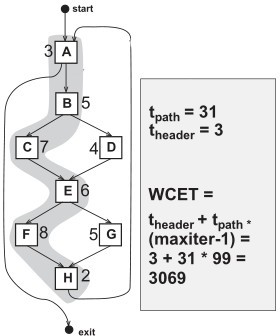


Рисунок 1.4 - Метод явного перечисления путей

*Метод неявного перечисления путей*

В случае IPET поток программы и временные метки базовых блоков объединяются в набор арифметических и/или логических ограничений. Каж- дому базовому блоку и дуге потока управления присваивается временной ко- эффициент, выражающий верхнюю границу вклада данной сущности в общее время исполнения, и переменная-счетчик, соответствующая количеству ис- полнений данной сущности. Верхняя граница определяется путем максимиза- ции суммы произведений количества исполнений и времени одного исполне- ния для каждой сущности, причем переменные-счетчики являются предметом ограничений, отражающих структуру задачи и возможных потоков. Результа- том работы IPET является верхняя граница времени исполнения и наихудшее значение для каждой переменной-счетчика исполнений.

На рисунке [1.5](#_bookmark17) иллюстрируется, как метод неявного перечисления пу- тей применялся бы для графа, приведенного на рисунке [1.2](#_bookmark14).

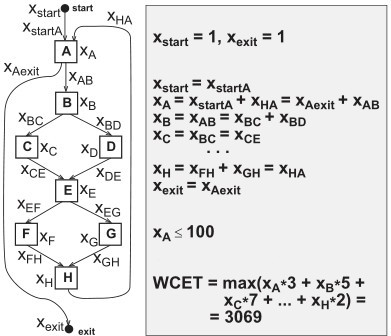


Рисунок 1.5 - Метод неявного перечисления путей (IPET)

Ограничения *start* и *exit* указывают, что вход и выход в программу вы- полняется только один раз. Структурные ограничения отражают возможный поток выполнения, они указывают, что для всякого базового блока количество раз, когда поток выполнения входит и выходит из него, должно совпадать. Граница цикла накладывает ограничение на то, сколько раз может быть вы- полнен узел *A*.

Система ограничений IPET может быть решена либо при помощи инструментальных средств решения задач с ограничениями, либо посред- ством техники целочисленного линейного программирования (ЦЛП). Причем применение ЦЛП более популярно ввиду наличия эффективных алгоритмов решения. Использование же средств решения задач с ограничениями позволя- ет выражать более сложные ограничения ценой потенциально большего вре- мени нахождения результата.

Термин «неявное перечисление путей» был впервые применен Мали- ком и др. [20]. Данный термин отражает то, что самый длинный путь выража- ется неявно. Позднее данный подход был расширен путем добавления учета

кэша команд. Фердинанд и др. также использовали IPET, но при этом в вы- числениях учитывали эффекты и от конвейеризации, и он кэширования [21].

Оттоссон и Сьедин представили подход, базирующийся на IPET и про- граммировании в ограничениях и позволяющий смоделировать возможные потоки выполнения, эффекты от конвейеризации и кэширования в рамках единого каркаса [22]. Вульф и Эрнст предложили применять IPET на про- граммных сегментах, потенциально содержащих несколько базовых блоков [23]. Таким образом, предложенный ими подход являлся гибридом явного и неявного перечисления путей.

*Метод оценки по регулярному выражению*

В работе [24] предлагается методика аналитических вычислений в ими- тационном моделировании, которая включает три этапа.

Этап 1 – представление объекта моделирования в виде графа переходов. Вершины графа соответствуют состояниям системы, а дуги всем возможным изменениям состояний. Каждая итерация моделирования заключается в выбо- ре случайного пути по графу от начальной до конечной вершины. Вес дуги определяет, насколько изменяется тот или иной параметр системы при воз- никновении события, которое моделируется переходом по данной дуге. То есть вес дуги – это случайная величина, добавляемая к расчетному значению при прохождении по этой дуге. Результатом моделирования является значе- ние сумм случайных величин, усредненное по всем возможным путям в гра- фе переходов.

Этап 2 – вывод регулярного выражения, описывающего множество переходов из начальной в конечную вершину графа.

Этап 3 – построение по регулярному выражению формулы для расчета необходимых значений. Для построения формулы регулярного выражения определяются характеристики весов дуг графа: математическое ожидание, дисперсию и вероятности прохождения дуг графа в точках разветвления.

Описанная методика может быть применена для статической оценки

времени исполнения программ. При этом объектом моделирования будет яв- ляться исследуемая программа. Вершины графа будут соответствовать точкам ветвления, а ребра – базовым блокам программы. В качестве же весов ребер будет выступать время выполнения соответствующих базовых блоков.

В данной работе, в главе 2, описан метод статической ОВЗИП, основывающийся на вышеприведенном принципе. Однако в нем предлагается базовым блокам программы сопоставлять не дуги графа, а его вершины, другими словами, предлагается в качестве модели программы использовать ГПУ, в классическом для статического анализа понимании.

* + 1. Инструменты статической ОВЗИП

Перечисленные в предыдущих разделах работы в той или иной мере яв- ляются частями некоторых утилит статической ОВЗИП. Однако не все иссле- довательские группы в итоге представляли завершенный инструмент для оценки времени исполнения программ, большинство из них – это нишевые продукты, либо исследовательские прототипы.

Группе из Университета Флориды удалось решить практически все проблемы статической ОВЗИП (в частности – WCET-анализа) и разработать соответствующую утилиту [25]. Эта утилита тесно связана с компилятором и потому может обрабатывать оптимизированный им код, а также оперировать временными характеристиками как на уровне исходного, так и на уровне объектного кода. Утилита способна автоматически определять некоторую ин- формацию о потоке, границах циклов и выявлять участки мертвого кода. В ней также применены алгоритмы, учитывающие различные типы кэша, вклю- чая ассоциативный кэш команд, многоуровневый кэш и кэш данных. Однако данная утилита является лишь исследовательским прототипом и не доступна для коммерческого применения.

Утилита Bound-T финской компании SSF была разработана при под- держке Европейского Космического Агентства для нужд космоса. Она под- держивает несколько целевых процессоров, включая Intel 8051, Analog

Devices ADSP-21020 и другие. Она работает на уровне объектного кода про- граммы, используя модули для чтения и декодирования бинарных файлов и извлечения из них ГПУ. Некоторая часть информации о потоке выполнения находится автоматически с помощью арифметики Пресбургера. Данный инструмент доступен для коммерческого использования и применялся для анализа реально существующих систем [26].

Компания AbsInt, образованная в ходе исследований в Университете Са- арланда (Германия), разработала набор инструментов, специализирующийся на технологиях компиляции для микроконтроллеров и DSP, оптимизации кода и WCET-анализе. Основой этого инструментария является PAG (Program Analysis Generator) – утилита для генерации статических анализаторов кода. Целевыми процессорами являются Coldfire MCF 5307 и PowerPC 750/755. Анализ проводится на уровне машинных команд с учетом некоторых особен- ностей аппаратного обеспечения, при этом ГПУ реконструируется из объект- ного кода. Анализ потока проводится автоматически на уровне объектного кода и основывается на значениях регистров, однако границы циклов требует- ся предоставлять вручную. Набор инструментов доступен для коммерческого использования.

### Преобразование ДКА в РВ

В разделе [1.3.4](#_bookmark13) был описан метод оценки времени исполнения про- грамм, основывающийся на преобразовании графа, представляющего про- грамму, в регулярное выражение, по которому и производится дальнейшая оценка. Стоит отметить, что существует несколько способов подобного преобразования.

В статье [27] Клини дал определение регулярных выражений и показал их эквивалентность с конечными автоматами. Также он предоставил метод преобразования ДКА в регулярное выражение – метод транзитивного замыка- ния. Позже Бжозовский расширил метод Клини, введя понятие производной регулярного выражения [28]. Еще один метод преобразования автомата в ре-

гулярное выражение – метод удаления состояний – был описан в работе [29], однако Линц позже описал более простой его вариант [30].

Далее приведено описание и анализ этих трех наиболее известных сре- ди существующих методов преобразования конечного автомата в регулярное выражение.

* + 1. Определения

Мы будем использовать модель Мура для представления конечного ав- томата [31]. Пусть дан автомат *M* с входным алфавитом *Ak*={0, 1, …*,k* −1 }. *M* имеет *m* состояний: *Ms*={*q*1,*q*2,…*, qm* }, где *q*λ – его начальное состояние.

*Mf* ={*qf ,qf ,*…*, qf* } – конечные состояния автомата, причем *n*≤*m*. Для ясно-

1 2 *n*

сти будем использовать буквы вместо чисел для представления значений из алфавита *A* (*a*=0, *b*=1 и т. д.). Также будем считать состояние *q*1 начальным, если не указано иное.

Дадим определение регулярным выражениям, предложенное Клини [27]. Регулярные выражения определяются набором рекурсивных правил:

1. Символы 0, 1,…*, k* −1, λ *,* φ являются регулярными выражениями, где

λ – пустая строка, а φ – пустое множество.

1. Объединение множеств: если даны регулярные выражения *x* и *y* , то их объединение, обозначаемое как *x* | *y*, – тоже регулярное выражение.
2. Конкатенация: если даны регулярные выражения *x* и *y* , то их конка- тенация, обозначаемая как *xy*, – тоже регулярное выражение.
3. Итерация: если дано регулярное выражение *x*, то его итерация, обозначаемая как *x*\*, – тоже регулярное выражение.
4. Если дано регулярное выражение *x*, то ( *x* ) – тоже регулярное выраже-

ние.

1. Все регулярные выражения могут быть получены путем применения

правил 1-5.

Регулярные выражения, описанные в правиле 1 называются терминала-

ми. Конкатенация терминалов – это строка. Для регулярного выражения *x* су- ществует множество *X*, которое содержит все строки, описываемые *x*. Далее мы будем считать *x* и *X* взаимозаменяемыми.

Также нам потребуются следующие равенства (поскольку операция объединения коммутативна, то также будут верны и равенства, в которых опе- ранды объединения поменяны местами):

(*ab*) *c*=*a* (*bc*)=*abc*

λ *x* =*x* λ=*x*

φ *x* =*x* φ=φ φ | *x*= *x*

λ | *x*\*=*x*\*

(λ | *x* )\*= *x*\*

(1.1)

* + 1. Метод транзитивного замыкания

Рассмотрим автомат, представленный на рисунке [1.6](#_bookmark23).

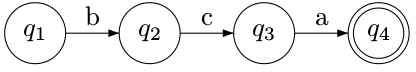


Рисунок 1.6 - Простой ДКА

Переходы между состояниями автомата могут быть описаны регуляр- ными выражениями. Так, регулярное выражение, соответствующее переходу из состояния *q*1 в *q*2 – *b*; переходу же из *q*2 в *q*3 соответствует *c* и так далее. Далее, регулярное выражение, представляющее переход из состояния *q*1 в *q*3 , – это конкатенация двух предыдущих регулярных выражений, т. е. *bc*. Та- ким образом, используя конкатенацию для всех переходов на пути из началь-

ного состояния *q*λ в конечное *qf*, мы можем построить регулярное выражение

для всего автомата. Оно будет иметь вид *bca*.

Рассмотрим более общий случай. Пусть дан ДКА, приведенный на ри- сунке [1.7](#_bookmark24).

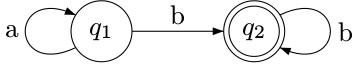


Рисунок 1.7 - Более сложный ДКА

Ясно, что существует множество различных путей из *q*1 в *q*2. Но в этом случае мы уже не можем построить регулярное выражение, используя лишь конкатенацию. Однако с помощью двух других операций это становится воз- можным [27].

Пусть регулярное выражение *Rij* представляет такое же множество

строк, что и переход автомата *M* из *q*

в *q* . Далее, пусть *Rk* представляет та-

*i j ij*

кое же множество строк, что и переход автомата *M* из *qi* в *q j* без посещения состояний выше, чем *qk* . Мы можем построить *Rij*, последовательно находя

*R*1, *R*2, …*, Rm*. Рекурсивное определение *Rk* приведено ниже.

*ij ij*

*ij ij*

База рекурсии:

{

*R* =

*r* если *i*≠ *j* и *r* переводит *M* из *qi* в *q j,*

0

*ij*

*r* | λ если *i*= *j* и *r* переводит *M* из *qi*

φ иначе.

в *q*

*j*

*,*

(1.2)

Шаг рекурсии:

*Rk* =*Rk* −1 (*Rk* −1 )\* *R*(*k*−1) | *Rk*− 1 (1.3)

*ij*

*ik*

*kk*

*kj*

*ij*

Таким образом мы можем построить регулярное выражение *Rij*. Для по-

лучения же регулярного выражения, соответствующего автомату *M* необходи- мо применить операцию объединения ко всем *R*λ *f* , где *f* ∈ *Mf* .

Этот метод схож с методом нахождения кратчайших путей между всеми

парами вершин в графе. Единственное отличие заключается в том, что в нем производится объединение и конкатенация регулярных выражений вместо суммирования расстояний.

Метод транзитивного замыкания довольно прост для понимания и реа- лизации, однако главная его проблема заключается в том, что он производит

чрезвычайно длинные регулярные выражения. Если рассмотреть формулу [(1.3)](#_bookmark25), становится ясно, что основная причина кроется в повторяющемся объединении конкатенаций термов.

* + 1. Алгебраический метод Бжозовского

Метод Бжозовского [28] заключается в следующем. Составляется систе- ма регулярных выражений с одним неизвестным для каждого состояния авто- мата *M* , после чего находится решение для *R*λ, где *R*λ – это регулярное вы-

ражение, соответствующее начальному состоянию *q*λ. Эти уравнения называ-

ются характеристическими уравнениями автомата *M* .

Характеристическое уравнение для каждого состояния *qi* автомата *M* представляет собой объединение термов. Для каждого перехода *a* из *qi* в *q j* терм имеет вид *a R j*. Причем, если *Ri* – конечное состояние, то к объединению

добавляется дополнительный терм – λ. Таким образом получаем систему

уравнений в канонической форме:

*R*1=*a*1,1 *R*1 |*a*1,2 *R*2|…| *a*1,*m Rm* |*b*1 *, R*2=*a*2,1 *R*1 | *a*2,2 *R*2 |…| *a*2,*m Rm* |*b*2 *,*

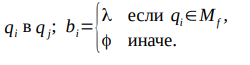
⋮

*Rm*=*am*,1 *R*1| *am*,2 *R*2 |…| *am ,m Rm* | *bm ,*

(1.4)

где *a*

*i , j*

=φ если не существует перехода из 

Данная система может быть решена простой подстановкой за исключе- нием случаев, когда неизвестное находится одновременно и в левой, и в пра- вой части уравнения. Возникновение таких ситуаций возможно, когда для со- стояния *qi* существует петля. В таком случае необходимо воспользоваться леммой Ардена [32]. Эта лемма формулируется следующим образом: для

*i*

уравнений вида *X* = *AX* | *B*, где λ∉ *A* решение имеет вид *X* = *A*\* *B*.

Таким образом, если положить что *q*λ=*q*1, то система уравнений [(1.4)](#_bookmark27) имеет алгоритм решения, описанный далее.

Рассмотрим *m*-тое уравнение:

*Rm*=*am*,1 *R*1|…| *am,m Rm* | *bm*

и применим к нему лемму Ардена:

*R* =*a*\* (*a*

*m, m*

*m*

*m*,1

*R* 1 |…| *am*−1,*m* −1

*Rm*−1

| *bm* ).

Далее, произведя замену *b '*

=*a*\* *b*

и *a '*

=*a*\* *a*

, получим:

*m m, m m*

*m, i*

*m , m*

*m, i*

*Rm*=*a'm*,1 *R*1 |…| *a' m*−1, *m*−1 *Rm*−1 | *b'm* .

После чего мы можем избавиться от *Rm* во всех остальных уравнениях системы, произведя замену для всех *i , j* ∈{1,…*, m*−1}:

*b 'i*=*bi* | *ai, m b 'm ,*

*a 'i , j*=*ai.j* | *ai, m a'm , j* .

Последовательно применяя описанную процедуру ко всем уравнениям, вплоть до первого, получим выражение вида

*R*1=*b'*1 *,*

которое и является искомым решением системы.

Алгебраический метод Бжозовского предоставляет элегантное с мате- матической точки зрения решение поставленной задачи и кроме того произво- дит довольно компактные регулярные выражения. Он также довольно прост в реализации.

* + 1. Метод удаления состояний

В методе удаления состояний метки ребер графа автомата рассматрива- ются как регулярные выражения. В графе выявляются определенные шабло- ны, которые преобразуются путем удаления из них промежуточных вершин с сохранением согласованности меток ребер. Далее приведено описание алго- ритма, который был предложен Линцем [30].

На первом шаге все кратные ребра графа объединяются в одно ребро, так, что метка этого ребра представляет собой объединение меток исходных ребер. Например, если из вершины *q*2 в вершину *q*5 существует ребро *a* и ре- бро *b*, то эти два ребра будут объединены в одно, имеющее метку *a* | *b*.

Далее, в графе автомата находится подграф, имеющий структуру, пред- ставленную на рисунке [1.8](#_bookmark29). Вершина *q* может быть удалена, в результате по- лучим граф, который приведен на рисунке [1.9](#_bookmark30). Заметим, что данный шаблон может быть применен даже если некоторые из ребер отсутствуют. В таком

случае следует заменить соответствующие части регулярных выражений в метках результирующих ребер на φ, произвести упрощение по формулам [(1.1)](#_bookmark21)

и удалить ребра с меткой φ.

Этот процесс повторяется до тех пор, пока не будет получен граф, при- веденный на рисунке [1.10](#_bookmark31). Для него результирующее регулярное выражение вычисляется по формуле

*r* =*r*\* *r* (*r* | *r r*\* *r* )\* .

1 2 4 3 1 2

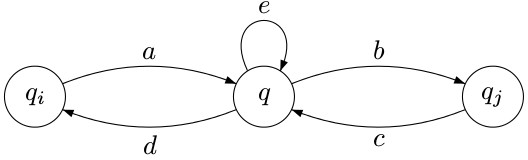


Рисунок 1.8 - Исходная форма графа

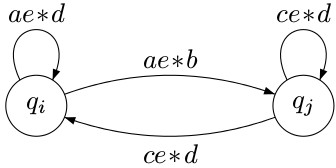


Рисунок 1.9 - Результат удаления состояния

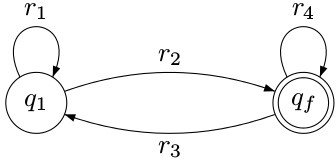


Рисунок 1.10 - Окончательная форма графа

Преимуществом такого подхода перед методом транзитивного замыка- ния является то, что он более прост для применения «вручную». Однако алго- ритмическая и программная реализация данного метода имеет ряд сложно- стей, заключающихся в необходимости рассмотрения различных особых слу- чаев. Примерами таких особых случаем может быть наличие у автомата нескольких конечных состояний, совпадение начального и конечного состоя- ний и т. д. [33]. Это делает его более сложным в реализации, а значит и более подверженным ошибкам.

* + 1. Общие недостатки

Из приведенных методов наиболее предпочтительным выглядит метод Бжозовского. Однако основной сложностью, мешающей использовать его, яв- ляется необходимость сохранения информации о потоке управления в ходе построения регулярного выражения. Речь идет, в частности, о вероятностях условных переходов и информации о количестве итераций циклов. Без этой информации построенное регулярное выражение не имеет никакой ценности для предлагаемого метода оценки времени исполнения программ. Тоже самое утверждение можно отнести и к методу транзитивного замыкания.

Другим общим недостатком для всех перечисленных методов является то, что они на вход принимают «классический» автомат, который, примени- тельно к задаче оценки времени исполнения программ, можно описать следу- ющим образом:

*M* =(*V , E , w , s ,e* )*,* (1.5)

где *V* – множество состояний, каждое из которых соответствует некоторой точке ветвления или точке, на которую передается управление после услов- ного/безусловного перехода; *E* ={(*i , j*): *i , j*∈*V* } – множество дуг, представ-

ляющих их себя упорядоченные пары состояний (не обязательно различных –

допускаются петли), каждая такая дуга по своей сути представляет базовый блок программы; *w* : *E* → *X* – функция переходов – отображение множества дуг *E* на множество случайных величин *X*, таким образом, каждому базовому

блоку (дуге) сопоставляется случайная величина, характеризующая время вы- полнения операций, входящих в него; *s* – начальное состояние, с которого на- чинаются все пути; *e* – конечное состояние, которым завершаются все пути (стоит заметить, что конечных состояний могло бы быть несколько, что соот- ветствовало бы нескольким точкам выхода из программы, однако в таком слу- чае можно искусственно добавлять еще одно состояние, принимающее поток управления от них, и считать его единственным конечным состоянием).

Однако такое определение значительно отличается от общепринятого в статическом анализе программ определения графа потока управления, в кото- ром базовые блоки программы соответствуют не ребрам, а вершинам. Из это- го следует, что на вход описанных алгоритмов построения регулярных выра- жений не получилось бы подавать графы потока управления, сформирован- ные сторонними модулями. Данную проблему можно было бы обойти, разра- ботав алгоритм преобразования графа потока управления к виду, описываемо- му формулой [(1.5)](#_bookmark33). Однако такой подход сопряжен с дополнительными на- кладными расходами вычислительных мощностей а также в большей степени подвержен ошибкам из-за излишнего усложнения процесса оценки времени исполнения программ.

### Выводы по главе 1

Проанализированы основные методы определения времени исполнения программ и современное состояние исследований в данном направлении. Вы- явлены недостатки динамической ОВЗИП, которые обуславливают необходи-

мость выбора статического подхода. Рассмотрены основные этапы, присущие статической ОВЗИП, и выделены те из них, которые удовлетворяют поставлен- ным требованиям и могут быть применены без значительных изменений (в частности, речь идет об этапах анализа потока управления и низкоуровневого анализа). Соответственно, также выявлено, что оставшийся этап – этап непо- средственного вычисления оценки – нуждается в доработке.

Кратко изложена основная идея предлагаемого метода расчета времени выполнения программ. Проанализированы существующие алгоритмы преоб- разования ДКА в регулярное выражение, которые могли бы использоваться в рамках предлагаемого метода. Выявлены две основные проблемы, препят- ствующие их применению, а именно, сложность сохранения информации о потоке управления программы в ходе преобразования, а также ожидание на входе графа, в котором базовые блоки программы соответствуют не верши- нам, как в общепринятом для статического анализа определении ГПУ, а ре- брам.

## Глава 2. Система статической ОВЗИП на основе вероятностного подхода

### Формальное описание метода ОВЗИП на основе вероятностного подхода

* + 1. Граф потока управления

Схему алгоритма всегда можно представить в виде циклического взве- шенного направленного графа, называемого графом потока управления (англ. control flow graph, CFG). Каждый узел (вершина) этого графа соответствует базовому блоку – прямолинейному участку кода, не содержащего в себе ни операций передачи управления, ни точек, на которые управление передается из других частей программы. Имеется лишь два исключения: точка, на кото- рую выполняется переход, является первой инструкцией в базовом блоке, и базовый блок завершается инструкцией перехода. Направленные дуги ис- пользуются в графе для представления инструкций перехода [34]. Также, в большинстве реализаций добавлено два специализированных блока: входной блок, через который управление входит в граф и выходной блок, который за- вершает все пути в данном графе.

Подобная форма графа потока управления используется во многих ме- тодах статического анализа кода, поэтому далее будет рассматриваться имен- но она. Формально ГПУ описывается следующим образом:

*G*=(*V , E , w , s , e*) (2.1)

где *V* – множество вершин, соответствующих базовым блокам программы; *E* ={(*i, j*): *i , j*∈*V* } – множество дуг, представляющих из себя упорядоченные пары вершин (не обязательно различных – допускаются петли); каждая дуга

по своей сути – инструкция перехода; *w* : *V* → *X* – функция, являющаяся отоб-

ражением множества вершин *V* на множество случайных величин *X* (таким образом, каждому базовому блоку (вершине) сопоставляется случайная ве- личина, характеризующая время выполнения операций (в наносекундах), входящих в этот базовый блок); *s* – начальная вершина, с которой начинаются

все пути; *e* – конечная вершина, которой завершаются все пути (за исключе- нием тех, которые приводят к зацикливанию программы).

В дальнейшем нам будет удобно считать, что функция *w* из формулы [(2.1)](#_bookmark38) не просто ставит случайную величину в соответствие вершине, но и по- мечает вершину обозначением (именем) этой случайной величины. Таким об- разом каждая вершина будет иметь собственную уникальную метку.

Пример графического представления такого ГПУ приведен на рисунке

[2.1](#_bookmark39):

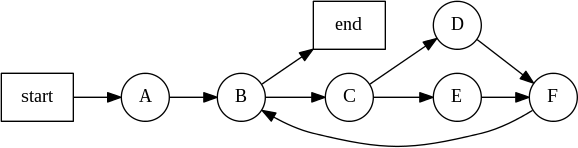


Рисунок 2.1 - Пример графа потока управления

Начальная и конечная вершины обозначаются прямоугольниками с мет- ками «start» и «end» соответственно.

* + 1. Представление ГПУ в виде регулярного выражения

Для дальнейшего анализа необходимо представить ГПУ в виде регуляр- ного выражения (РВ). Ранее, в разделе [1.4](#_bookmark19) было приведено определение РВ, данное Клини, однако нам будет удобно воспользоваться иными обозначения- ми и дать некоторые пояснения, относительно применения РВ в нашей зада- че.

Итак, РВ состоит из констант и операторных символов, которые обозна- чают соответственно множества строк и операций над ними. Следующее определение является общепринятым и может быть найдено во многих рабо-

тах по теории формальных языков [35][36]. Пусть дан алфавит Σ (в нашем

случае он состоит из меток вершин ГПУ, т. е. из наименований случайных ве- личин из множества *X*), тогда следующие константы являются РВ:

* пустое множество ∅ (в нашем случае оно означает отсутствие путей

из одной вершины ГПУ в другую);

* пустая строка (в нашем случае она соответствует пустому пути);
* литерал σ∈Σ, обозначающий множество, состоящее только из симво-

ла .

Для любых РВ *R*1 и *R*2, описывающих множества строк *S*1 и *S*2 соот-

ветственно, определены следующие операции, результаты которых также яв- ляются РВ:

* конкатенация *R*1 *R*2 описывает множество {*s*1 *s*2: *s*1∈*S*1 *, s*2∈*S*2};
* альтернация: *R*1 | *R*2 описывает множество *S*1∪*S*2;
* и замыкание Клини *R*\* описывает множество всех строк, которые мо- гут быть получены конкатенацией любого конечного числа строк из *S*1.

1

В дальнейшем нам пригодятся следующие равенства, справедливые для любого регулярного выражения *R* [36]:

*R* ∅=∅ *R* =∅ *; R* | ∅=∅| *R*=*R;*

∅\*=ε *;*

*R* ε=ε *R*=*R;*

ε\*=ε.

(2.2)

Итак, рассмотрим структуру путей из вершины *i* в вершину *j*. Любой такой путь можно разбить на две составляющие:

* + - 1. Нуль или более последовательных путей из вершины *i* в вершину *i*.
      2. Путь из вершины *i* в вершину *j*, в котором *i* не является промежуточ- ной вершиной.

Более строгое описание посредством РВ будет иметь следующий вид:

(*F* )

*Rij*

{*P*( *F* ∪{*i* })\* *P*( *F* ∪{*i* }) если *i*≠ *j ,*

ε если *i*= *j ,*

=

*ii*

*ij*

(2.3)

где *i* – начальная вершина; *j* – конечная вершина; *F* – запрещенное множе- ство (множество вершин, которые не могут быть промежуточными на пути из

*i* в *j*); *P*(*F*∪{*i*}) и *P*(*F*∪{*i*}) вычисляются по формуле [(2.4)](#_bookmark43).

*ii ij*

*P*( *F* )=σ

*ij*

(*R*(*F*) | *R*(*F*)|…| *R*(*F*))*,* (2.4)

где *i* – начальная вершина; *j* – конечная вершина; {*i*1 *,*…*,ik* } – множество вершин, смежных с *i* и не принадлежащих запрещенному множеству *F*, а так- же и вершина *j*, если она смежна с *i*; σ*i* – метка вершины *i*.

*i*

*i*1 *j*

*i* 2 *j*

*ik j*

Итоговая формула для построения РВ, описывающего все пути из на-

чальной вершины ГПУ в конечную , будет иметь вид:

*R*= *R*(∅)σ

*se*

*e*

*,* (2.5)

где *s* – начальная вершина ГПУ; *e* – конечная вершина ГПУ; σ *e* – метка конеч- ной вершины ГПУ.

Основываясь на формулах [(2.3)](#_bookmark42) и [(2.4)](#_bookmark43), заметим что множество запре- щенных вершин постоянно растет, что гарантирует конечность рекурсивного погружения.

Для упрощения выражений, получаемых при вычислении формул [(2.3)](#_bookmark42), [(2.4)](#_bookmark43) и [(2.5)](#_bookmark44), необходимо применять равенства, приведенные в [(2.2)](#_bookmark41).

Ниже приведен алгоритм построения РВ на основе ГПУ (описание конструкций, применяемых в псевдокоде, см. в приложении):

**buildFullRegex**(G)

*s*← *G.start e* ←*G.end*

σ *e* ←*G.getMark* (*e*)

*buildRegex* (*G,* ∅ *, s ,e* )σ*e*

Функция построения РВ по ГПУ *buildFullRegex* принимает в качестве параметра объект ГПУ *G*. В ней используются два поля этого объекта: *start* – начальная вершина ГПУ; *end* – конечная вершина ГПУ – а также метод *getMark*, возвращающий метку вершины, указанной в качестве параметра. *buildFullRegex* сводится к вызову с заданными параметрами функции *buildRegex*, которая приведена ниже:

**buildRegex**(G, F, i, j)

**if** *i*= *j* **then return**

**else**

*F '* ← *F*∪{*i* } *L* ←*buildPart* (*G , F ' ,i ,i* )\* *D*← *buildPart* (*G, F ' , i , j*) **return** *L D*

Функция *buildRegex* в свою очередь принимает объект ГПУ *G*, запре- щенное множество *F*, начальную вершину *i* (при рекурсивном вызове *i* не обязательно совпадает с начальной вершиной ГПУ) и конечную вершину *j* (при рекурсивном вызове *j* также не обязательно совпадает с конечной вер- шиной ГПУ). В случае, если начальная и конечная вершина различны, ре- зультирующее РВ получается конкатенацией циклической и прямой состав- ляющих. Циклическая составляющая – это нуль или более последовательных путей из вершины *i* в вершину *i*. Прямая составляющая – это непосредствен- ный путь из *i* в *j*, в котором *i* не является промежуточной вершиной. Обе эти составляющие могут быть получены при помощи функции *buildPart* **:**

**buildPart**(G, F, i, j)

*P* ←∅

**for each** *e* ∈*G* . *getIncidentEdges*(*i* ) **do**

*n*←*e* . *target*

**if** *n*= *j* **or** *n* ∉ *F* **then**

σ*i* ←*G* . *getMark* (*i*)

*P* ← *P* | σ*ibuildRegex* (*F ' , n, j*)

**return** *P*

Функция *buildPart* принимает такие же параметры, как и *buildRegex*. В ней происходит итерация по всем ребрам ГПУ, инцидентным вершине *i*. Мно- жество этих ребер возвращает метод *getIncidentEdges* объекта ГПУ. В ходе итерации происходит построение РВ путем альтернации отдельных состав- ляющих.

Используя описанный выше алгоритм для графа, изображенного на ри-

сунке [2.1](#_bookmark39), получим следующее регулярное выражение:

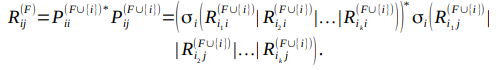
*start A* (*B*(*C D F* | *C E F* ))\* *B end.*

* + 1. Оптимизация процесса построения регулярного выражения

В алгоритме, описанном в пункте [2.1.2](#_bookmark40), несложно заметить характер- ные особенности, которые могут оказать отрицательное влияние на его производительность. Далее приведено описание этих особенностей и возмож- ных способов их устранения.

*Запоминание промежуточных результатов*

Опишем, воспользовавшись последовательно формулами [(2.3)](#_bookmark42) и [(2.4)](#_bookmark43), множество возможных путей из вершины *i* в вершину *j* (*i* и *j* различны) с учетом запрещенного множества *F*:

*R*(*F* )=*P*( *F*∪{*i*})\* *P*(*F* ∪{*i*})=(σ (*R*(*F* ∪{*i*})| *R*( *F* ∪{*i* }) |…| *R*( *F*∪{*i* })))\* σ (*R*( *F* ∪{*i* }) |

*ij*

*ii*

*ij*

*i*

*i*1 *i*

*i*2 *i*

*i k i*

*i*

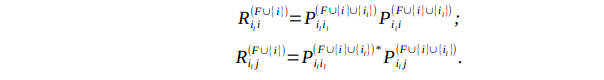
*i*1 *j*

| *R*( *F* ∪{*i* }) |…| *R*( *F*∪{*i*})).

*i*2 *j*

*ik j*

Далее, применим формул[у (2.3)](#_bookmark42) к *l*-тым элементам из обеих скобок:

*R*(*F* ∪{*i*})=*P*( *F* ∪{*i* }∪{*il* }) *P*( *F*∪{*i*}∪{*il* }) *;*

*il i*

*il il*

*il i*

*R*(*F* ∪{*i*})=*P*( *F* ∪{*i* }∪{*il* })\* *P*(*F* ∪{*i*}∪ {*il* }) .

*il j*

*il il*

*il j*

Видно, что *P*(*F*∪{*i*}∪{*il* }) входит в оба равенства. Ясно также и то, что при

*i i*

*l l*

дальнейшем преобразовании приведенных выше формул, подобные повторе- ния будут встречаться все чаще.

В «наивной» реализации алгоритма, приведенной в пункте [2.1.2](#_bookmark40), каж- дое такое повторение будет обрабатываться независимо, что может суще- ственно снизить производительность работы алгоритма. Данную проблему

можно решить, запоминая результаты построения РВ *P*(*F*) в ассоциативном массиве (кэше), ключами для которого будут выступать тройки вида (*i , j , F*). Стоит заметить, что *P*(*F*) по сути своей является частичной функцией

*ij*

*ij*

*V* ×*V* ×Φ→ *R*,

где *V* – множество вершин, Φ – множество запрещенных множеств, а *R* –

множество получаемых регулярных выражений. Таким образом, для пары вершин (начальной и конечной) и некоторого запрещенного множества суще- ствует единственный возможный результат. Отсюда можно сделать вывод, что запоминание промежуточных результатов не повлияет на корректность алго- ритма.

Для использования кэша при построении РВ необходимо внести кор- рективы в функцию *buildPart* **:**

**buildPart**(G, С, F, i, j)

*key* ←(*i , j , F* )

**if** *C* . *contains* ( *key*)

**return** *C* [ *key* ]

**else**

*P* ←∅

**for each** *e* ∈*G* . *getIncidentEdges*(*i* ) **do**

*n*←*e* . *target*

**if** *n*= *j* **or** *n* ∉ *F* **then**

σ*i* ←*G* . *getMark* (*i*)

*P* ← *P* | σ*i buildRegex* (*F ' , n, j*)

*C* [ *key* ]← *P*

**return** *P*

В новой версии *buildPart* добавлен параметр *С* – кеш, представляющий собой ассоциативный массив. При помощи него производится проверка на то, был ли ранее обработан путь с данными входными параметрами. В случае по- ложительного результата сразу возвращается результат из кэша, иначе – строится путь по обычному алгоритму, после чего результат добавляется в кэш и возвращается.

Таким образом, используя запоминание промежуточных результатов можно будет повысить производительность работы алгоритма ценой увеличе- ния потребления памяти.

*Учет достижимости вершин*

Другой причиной вероятного снижения производительности является

обработка РВ вида *R*(*F*), где вершина *i* недостижима из вершины *j*. Решение данной проблемы сводится к построению транзитивного замыкания рассмат- риваемого графа. Транзитивное замыкание графа *G* определяется как граф

*ij*

*G'*=(*V , E '* ),

где *E '* ={(*i, j*): в графе G имеется путь из вершины i в вершину j }.

Таким образом, в ходе построения РВ *R*(*F*), по предварительно по- строенному транзитивному замыканию можно будет определить, достижима

*ij*

ли вершина *i* из вершины *j*, и в случае отрицательного результата прекратить дальнейшую обработку данного пути.

Один из способов найти транзитивное замыкание графа за время Θ(*n*3)

– присвоить каждому его ребру вес 1 и выполнить алгоритм Флойда-Варшал- ла. Если путь из вершины *i* в вершину *j* существует, то мы получим длину пути *dij*<*n*; в противном случае *dij*=∞. При этом следует заметить, что рассматриваемые нами графы потока управления удовлетворяют ограниче-

ниям, накладываемым алгоритмом Флойда-Варшалла: в них отсутствуют циклы отрицательного веса.

Внеся некоторые модификации в алгоритм Флойда-Варшалла, можно добиться экономии времени (однако, асимптотически ничего не изменится) и памяти. Для этого необходимо заменить арифметические операции *min* и + на логические операции дизъюнкции и конъюнкции соответственно [37].

Итак, опишем алгоритм построения транзитивного замыкания графа *G*=(*V , E*), где *V* ={1,2 *,*…*,n* } – множество вершин, а *E* ={(*i, j*): *i , j*∈*V* } – мно- жество ребер. Определим значение *t*(*k*) при *i , j , k* =1,2 *,*…*, n* равным 1, если в графе G существует путь из *i* в *j*, все промежуточные вершины которого при- надлежат множеству {1,2 *,*…*, k* }; в противном случае эта величина равна 0.

*ij*

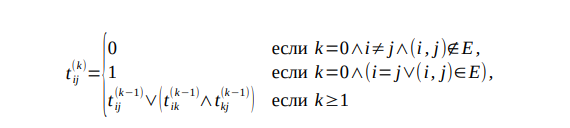
При построении транзитивного замыкания *G'*=(*V , E '* ), будем помещать ре-

бро (*i , j* ) в *E '* тогда и только тогда, когда *t*(*k*)=1. Рекурсивное определение ве-

*ij*

личины *t*(*k*) имеет вид:

*ij*



0 если *k* =0∧*i*≠ *j*∧(*i , j* )∉ *E ,*

{

*t*(*k*)= 1 если *k* =0∧(*i*= *j*∨(*i, j*)∈ *E*)*,*

*ij*

*t* (*k* −1)∨(*t*( *k* −1)∧*t*(*k* −1)) если *k* ≥1

*ij*

*ik*

*kj*

В таком случае искомое транзитивное замыкание будет представлять из себя матрицу  *T* (*n*)=(*t*(*n*)) и находиться следующим образом:

*ij*

**buildTransitiveClosure**(G)

*n* ←|*G* . *vertices*|

**for each** *i* ∈{1,…*, n* } **do for each** *j* ∈{1,…*, n* } **do**

**if** *i*= *j*∨(*i, j*)∈*G*. *edges* **then**

*t*( 0)←1

*ij*

**else**

*t*( 0)← 0

*ij*

**for each** *k* ∈{1,…*, n* } **do for each** *i* ∈{1,…*, n* } **do**

**for each** *j* ∈{1,…*, n* } **do**

*t*( *k* )←*t* (*k* −1)∨(*t*( *k* −1)∧*t*(*k* −1))

*ij ij*

**return** *T* (*n*)

*ik kj*

Для использования транзитивного замыкания в ходе построения РВ необходимо внести коррективы в функцию *buildRegex*:

**buildRegex**(G, T, F, i, j)

**if** *i*= *j* **then return**

**else if** T.contains(i, j)

*F '* ← *F*∪{*i* } *L* ←*buildPart* (*G, F ' ,i ,i* )\* *D*← *buildPart* (*G , F ' , i , j*) **return** *L D*

**else**

**return** ∅

В новой версии *buildRegex* добавлен параметр *T* – объект транзитивно- го замыкания, который предоставляет метод *contains*. Посредством этого ме- тода проверяется наличие пути из вершины *i* в вершину *j*.

Естественно, не стоит забывать, что построение транзитивного замыка-

ния по алгоритму Флойда-Варшалла занимает время Θ( *n*3), поэтому особый

интерес представляет экспериментальная проверка того, даст ли положитель- ный эффект предварительная обработка ГПУ подобным способом.

*Использование параллельных вычислений*

Для больших ГПУ этап построения регулярного выражения может ока- заться очень трудоемким. Однако, стоит заметить, что описанный алгоритм прекрасно вписывается в функциональную парадигму: он не требует измене- ния состояний и не основывается на побочных эффектах. А это значит, что он легко может быть распараллелен для выполнения на многопроцессорных си- стемах или на высокопроизводительных кластерах.

В частности, стоит обратить внимание, что каждый операнд конкатена- ции в формуле [(2.3)](#_bookmark42) и каждый операнд альтернации в формуле [(2.4)](#_bookmark43) может быть вычислен абсолютно независимо в отдельном потоке или же на отдель- ном ноде системы. Помимо этого, вычисление каждого из операндов в свою очередь также может быть распараллелено (это объясняется рекурсивной природой алгоритма: на каждом шаге выполняются одинаковые действия, из- меняются лишь начальная и конечная вершины и запрещенное множество). Поэтому, если какие-то из потоков завершили необходимые вычисления раньше, то им можно делегировать часть вычислений, которые еще предстоит выполнить остальным потокам.

Стоит также заметить, что в ходе вычисления возникает потребность неоднократно вычислять одинаковые выражения с одинаковыми параметра- ми. Естественно, получаемые результаты всегда совпадают. Как было указано в первом пункте раздела [2.1.3](#_bookmark45) данная особенность легко решается посред- ством мемоизации получаемых результатов (некоторые среды исполнения чи- сто функциональных языков сами мемоизируют результаты вызова функций, что снимает необходимость внесения изменений в исходный алгоритм). Это дает значительный прирост производительности. Однако в случае параллель- ного выполнения алгоритма следует дополнительно озаботиться синхрониза-

цией доступа к кэшу из разных потоков или разных нодов распределенной вычислительной системы.

Предположительно, для реализации «параллельной» версии алгоритма хорошо мог бы подойти Erlang – функциональный язык, предназначенный для создания распределенных вычислительных систем.

*Упрощение РВ*

Помимо оптимизации времени выполнения алгоритма также важно учитывать и объем потребляемой в ходе его работы памяти.

Например, можно заметить, что при построении регулярного выраже- ния часто будут возникать промежуточные результаты вида

*R* ∅, (2.6)

а также

*R*1 | *R*2 |…| ∅|…| *Rn*. (2.7)

Такие выражения характерны при построении регулярного выражения для вершин, между которыми отсутствует путь в ГПУ. Такие выражения мо- гут быть упрощены с использованием равенств, приведенных в [(2.2)](#_bookmark41). Так, вы- ражение [(2.6)](#_bookmark46) примет вид *R*, а выражение [(2.7)](#_bookmark47) – вид *R*1 | *R*2 |…| *Rn*. Для ГПУ с

большим количеством вершин подобные упрощения могут привести к значи-

тельной экономии памяти.

Возможные стратегии упрощения РВ:

* в ходе построения РВ, при добавлении каждой новой операции;
* по окончании построения РВ; данный подход имеет мало смысла, по- скольку не исключает чрезмерного потребления памяти в ходе построения.

Практическая ценность и эффективность данной оптимизации, опреде- ленно нуждаются в экспериментальной проверке.

* + 1. ОВЗИП по регулярному выражению

Для дальнейшего анализа сформированного на предыдущем этапе регу- лярного выражения введем операцию итерации для регулярного выражения *R*

, описывающего множество строк *S*. Итак, итерацией *RN* назовем регулярное выражение, описывающее множество строк, полученных конкатенацией *N* строк из *S*. В таком случае, если в качестве *N* выбрать случайную величину, характеризующую количество повторений, то можно от замыкания Клини

перейти к итерации, т. е. все вхождения вида *R*\* заменить на *RN*.

Напомним, что метки вершин, из которых состоит полученное РВ – это обозначения (имена) случайных величин, характеризующих время выполне- ния соответствующих базовых блоков (функция *w* из формулы [(2.1)](#_bookmark38)). Тогда, на основе сформированного на предыдущем этапе РВ, информации о вероят- ностях условных переходов, информации о количестве итераций циклов и описанных ниже рекурсивных правил можно оценить математическое ожида- ние и дисперсию времени исполнения программы, описываемой ГПУ [24].

*База рекурсии*

Для пустого пути математическое ожидание и дисперсия равны нулю:

*M*ε=0, *D*ε=0.

(2.8)

Для литерала *L* математическое ожидание и дисперсия равны математи- ческому ожиданию и дисперсии одноименной случайной величины:

*ML*=*M* [ *L* ] *,*

*DL*=*D* [ *L* ]*.*

(2.9)

*Шаг рекурсии*

Для конкатенации регулярных выражений *R*= *R*1 *R*2

*M R*=*MR* + *MR , DR*= *DR* + *DR* .

1 2

(2.10)

1 2

Для альтернации регулярных выражений *R*= *R*1| *R*2|…| *Rn*

*MR*=*P*1 *MR* + *P*2 *MR* +…+ *Pn MR ,*

1 2 *n*

*D* = *P* ( *D* + *M*2 )+ *P* (*D* + *M*2 )+…+*P* ( *D* + *M*2 )− *M*2 *,*

(2.11)

*R* 1 *R* 1 *R*1

2 *R*2 *R*2

*n Rn Rn R*

где *P*1 *, P*2 *,*…*, Pn* – вероятности переходов по соответствующим ветвям.

В случае итерации *R*= *RN*

1

*MR*=*MR M* [ *N* ] *,*

1

(2.12)

2

*DR*= *DR M* [ *N* ]+ *M D* [ *N* ] *,*

*R* 1

1

где *N* – случайная величина, характеризующая количество итераций соответ- ствующего цикла.

### Архитектура системы

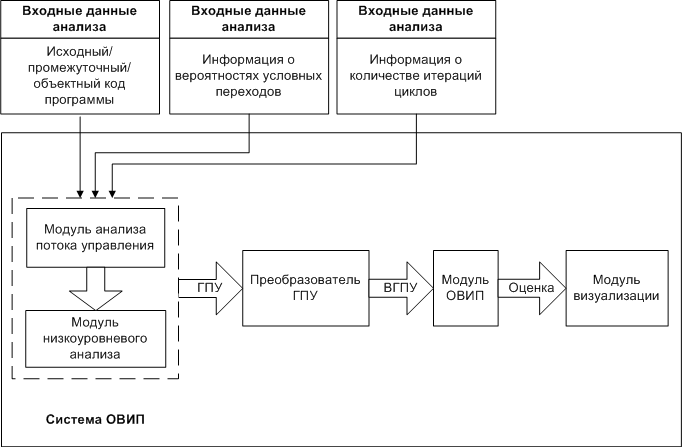
Система ОВЗИП представляет собой набор модулей, выполняющих по- следовательные шаги анализа (см. рисунок [2.2](#_bookmark55)).

Рисунок 2.2 - Диаграмма потока данных в системе ОВЗИП

Первый шаг анализа производится двумя модулями: модулем анализа потока управления и модулем низкоуровневого анализа. В результате их рабо- ты получается ГПУ, в который интегрирована такая информация, как вероят- ности условных переходов, оценки количества итераций циклов циклов, вре- мя исполнения базовых блоков. Модуль анализа потока управления произво- дит анализ исходного, промежуточного или объектного кода программы и вы- являет возможные потоки выполнения в программе, т. е. возможные последо-

вательности инструкций, которые могут быть выполнены. Модуль низкоуров- невого анализа на основе объектного кода и параметров целевого аппаратного обеспечения определяет временные характеристики выполняемых инструк- ций; для современных процессоров крайне важно учитывать эффекты, оказы- ваемые различными методиками увеличения производительности, такими как кэширование и конвейеризация. Эти два модуля логически могут быть объединены в один модуль, называемый фронтендом.

Модуль преобразования ГПУ является адаптером между внешним и внутренним представлением ГПУ. Он необходим в тех случаях, когда необхо- димо использовать стороннюю реализацию фронтенда, которая дает на выхо- де ГПУ в несовместимом формате. Данный модуль позволяет интегрировать в систему уже существующие (сторонние) фронтенды.

Модуль ОВЗИП занимается непосредственной оценкой временнных затрат на выполнения программы, представленной в виде ГПУ, в который интегрирована вся необходимая информация о потоке управления. Данный модуль включает в себя реализации алгоритмов, описанных в разделе [2.1](#_bookmark36).

Модуль визуализации отвечает за графическое отображение результатов ОВЗИП.

Архитектура системы ОВЗИП нацелена в первую очередь на гибкость и возможность «прозрачной» замены модулей. Так, изменение формата вход- ных данных (например, языка, на котором написана анализируемая про- грамма) потребует лишь замены фронтенда, остальные модули при этом не потребуют модификации. Аналогично, при необходимости использования стороннего фронтенда, возвращающего граф потока управления в собствен- ном формате, достаточно будет разработать лишь модуль преобразования ГПУ. Таким образом достигается слабая связность модуля, непосредственно проводящего оценку времени исполнения программ, с представлением вход- ных данных анализа.

На рисунке [2.3](#_bookmark56) представлена диаграмма основных компонентов систе-

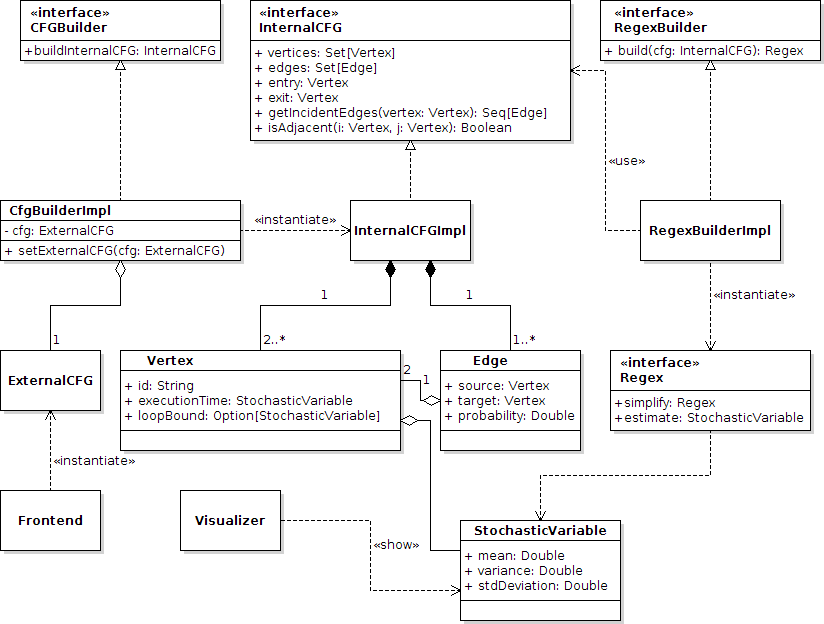
мы. Приведенные на ней классы и интерфейсы предоставляют поля и методы, необходимые для реализации алгоритмов, описанных в разделе [2.1](#_bookmark36).

Рисунок 2.3 - Диаграмма основных компонентов системы

Заметим, что на рисунке [2.3](#_bookmark56) модули анализа потока управления и низко- уровневого анализа представлены как один класс – Frontend. Это объясняется тем, что реализация этих модулей крайне нетривиальна и выходит за рамки данной работы.

* + 1. Реализация внутреннего представления ГПУ

Для представления графов в программировании зачастую используется структура данных, называемая матрицей смежности – матрица, индексы строк и столбцов которой соответствуют номерам вершин, а в ячейках хра- нятся значения, указывающие на то, существует ли ребро между соответству- ющей парой вершин. Другим возможным вариантом является использование списка смежности. Реализация графа в виде списка смежности предполагает

связь каждой вершины в графе с коллекцией соседних вершин или ребер. Для разреженных графов, в которых большинство пар вершин не связаны между собой, (именно к таким графам можно отнести ГПУ) список смежности зна- чительно более эффективен по потреблению памяти: память занимаемая им пропорциональна количеству ребер и вершин в графе, в то время, как матри- ца смежности (реализованная при помощи массива) занимает объем, пропор- циональный квадрату количества вершин. Однако существует более эффек- тивная по памяти реализация матрицы смежности, которая использует не массив, а хэш-таблицу, индексированную по всем парам вершин.

Но более значительным для нас является различие в эффективности вы- полнения операции получения списка смежных ребер, поскольку в алгорит- ме, описанном в пункте [2.1.2](#_bookmark40), эта операция используется на каждом шаге ре- курсии. Список смежности позволяет выполнить эту операцию за *O* (1), в то

время как для матрицы смежности она занимает время *O* (| *V* |), где *V* – мно-

жество ребер графа. Таким образом мы приходим к выводу, что использова- ние списка смежности является для нас более предпочтительным.

Существует множество вариантов реализации списка смежности, отли- чающихся тем, как представлены связи между вершинами и коллекциями, как реализуются сами коллекции и т. д.

В реализации, предложенной Гвидо ван Россом, используется хэш- таблица для связи вершин графа с массивами смежных им вершин [38]. В этой реализации вершиной может быть любой объект, для которого можно вычислить хэш. Однако ребра графа не имеют явного представления. Нам же требуется хранить информацию о вероятностях переходов по ребрам ГПУ, поэтому данная реализация не является подходящей.

Кормен и др. предложили реализацию, в которой вершины графа про- нумерованы, а ссылки на односвязные списки смежных им вершин хранятся в массиве, в ячейках с соответствующими индексами [37]. В такой реализа- ции узлы односвязного списка могут быть интерпретированы как ребра гра-

фа, однако в них отсутствует ссылка на вершину, являющуюся концом ребра, поэтому данная реализация также не походит для нашего алгоритма.

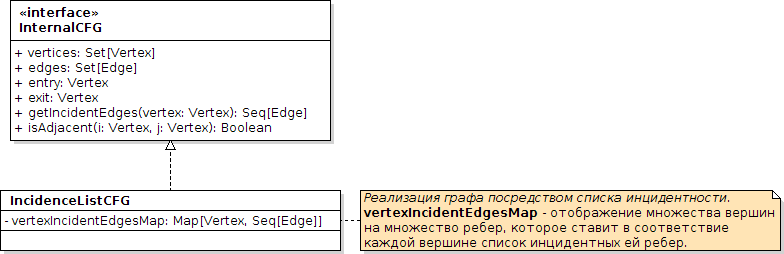
Гудрич и Тамассия предложили структуру, называемую объектно-ори- ентированным списком инцидентности [39]. В их реализации существуют отдельные классы для вершин и ребер. Каждая вершина имеет ссылку на кол- лекцию ребер. В свою очередь каждое ребро имеет ссылки на две вершины: начальную и конечную. Такой вариант списка смежности требует несколько больше памяти, чем предыдущие, однако существование явных экземпляров ребер дает дополнительную гибкость в хранении информации, относящейся к ним. По этой причине данный вариант представления графа был выбран за основу при реализации алгоритма (см. приложение, класс IncidenceListCfg и рисунок [2.4](#_bookmark58)).

Рисунок 2.4 - Реализация внутреннего представления ГПУ

* + 1. Реализация регулярных выражений

В соответствии с определением регулярного выражения, указанного в пункте [2.1.2](#_bookmark40), и с требованиями реализуемого алгоритма была спроектирована иерархия классов. Она приведена на рисунке [2.5](#_bookmark60).

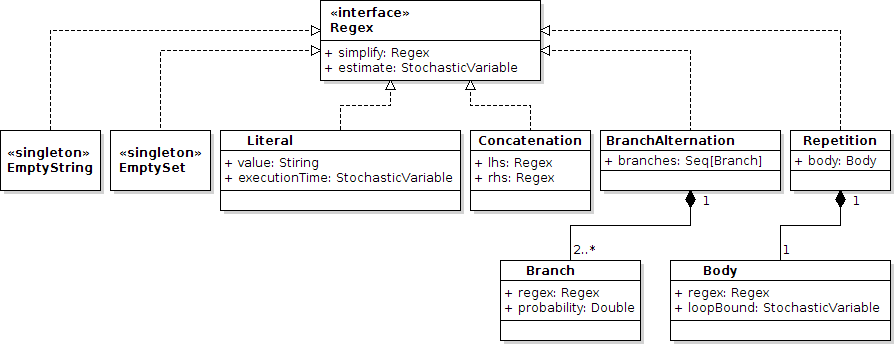


Рисунок 2.5 - Диаграмма классов, реализующих интерфейс Regex

Стоит заметить, что вместо обычной операции альтернации было введе- но понятие «пакетной» альтернации (batch alternation). Отличие ее заключает-

ся лишь в том, то она *n*-арная, где *n*≥2, в то время, как обычная альтернация –

бинарная операция. Необходимость введения новой операции связана с тем, что при вычислении по формуле [(2.11)](#_bookmark52) математического ожидания и диспер- сии для альтернации требуется информация о вероятности выбора каждой отдельной ветви в точке ветвления.

Как видно из рисунка [2.5](#_bookmark60), интерфейс Regex предоставляет только два метода: *simplify* – упрощение регулярного выражения и *estimate* – расчет оценки времени исполнения участка кода, представляемого данными регуляр- ным выражением. Каждый из классов имеет собственную реализацию в соот- ветствии с формулами [(2.2)](#_bookmark41) (для метода *simplify*), [(2.8)](#_bookmark49), [(2.9)](#_bookmark50), [(2.10)](#_bookmark51), [(2.11)](#_bookmark52) и [(2.12)](#_bookmark53) (для метода *estimate*).

Отметим также, что регулярное выражение можно рассматривать как алгебраический тип данных. По определению алгебраический тип данных – размеченное объединение декартовых произведений множеств или, другими словами, размеченная сумма прямых произведений множеств [40].

С теоретической точки зрения алгебраическим типом данных является размеченное объединение множеств (иначе называемое «дизъюнктным объединением»), под которым понимается видоизмененная классическая опе- рация объединения – такая операция приписывает каждому элементу нового

множества метку (или индекс), по которой можно понять, из какого конкретно множества элемент попал в объединение. Соответственно, каждый из элемен- тов размеченного объединения в свою очередь является декартовым произве- дением некоторых иных множеств.

С практической же точки зрения это означает, что регулярные выраже- ния могут быть реализованы естественным образом на функциональных языках, таких как Haskell или Scala, поддерживающих алгебраические типы данных. Из этого в свою очередь следует, что в сравнении с ООП подходом при реализации можно будет значительно сократить количество кода для про- верки целостности и непротиворечивости состояния объектов, избавиться от проверок и приведений типов [40]. Кроме того для регулярных выражений бу- дет доступен такой механизм, как сопоставление с образцом (pattern matching) (в случае приведенных выше языков), что может существенно об- легчить реализацию необходимых алгоритмов.

* + 1. Реализация алгоритмов построения РВ

На рисунке [2.6](#_bookmark62) приведена иерархия классов, которые непосредственно реализуют тот или иной алгоритм построения РВ:

– класс SimpleRegexBuilder: исходный алгоритм, описанный в разделе

[2.1.2](#_bookmark40);

* класс PackratRegexBuilder: алгоритм с запоминанием промежуточных

результатов;

* + класс RegexBuilderWithTransitiveClosure: алгоритм с использование транзитивного замыкания ГПУ;
  + класс PackratRegexBuilderWithTransitiveClosure: алгоритм, сочетаю-

щий запоминание промежуточных результатов и использование транзитивно- го замыкания ГПУ.

Все приведенные классы реализуют общий интерфейс RegexBuilder, это позволяет легко заменять одну реализацию на другую.

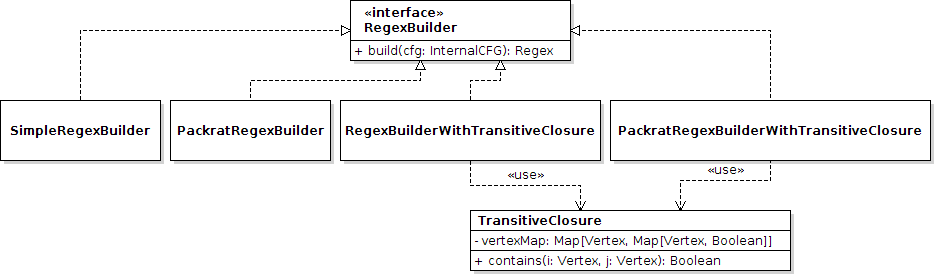


Рисунок 2.6 - Диаграмма классов, реализующих интерфейс RegexBuilder

### Выводы по главе 2

Приведено формальное описание метода статической оценки времени исполнения программ на основе вероятностного подхода:

* + описан алгоритм преобразования ГПУ в регулярное выражение;
  + описан алгоритм вычисления математического ожидания и диспер- сии времени выполнения программы по регулярному выражению.

Помимо этого предложено несколько возможных способов оптимиза- ции процесса построения регулярного выражения по ГПУ:

* + запоминание промежуточных результатов в ходе построения;
  + использование транзитивного замыкания ГПУ для учета достижимо- сти вершин;
  + использование параллельных вычислений;
  + упрощение регулярных выражений в ходе построения.

Также описана архитектура системы ОВЗИП, нацеленная в первую оче- редь на гибкость и возможность «прозрачной» замены модулей, что теорети- чески может позволить интегрировать в систему сторонние модули.

## Глава 3. Верификация результатов исследования и практические рекомендации

При верификации разработанного метода статической ОВЗИП произво- дилась оценка его показателей по трем направлениям:

* + точность получаемых результатов;
  + производительности алгоритма построения РВ,
  + объем потребляемой памяти.

### Особенности динамической оценки времени исполнения программ для JVM

Для оценки показателей разработанного метода нам потребуется поми- мо прочего применять метод динамической оценки времени исполнения про- грамм. Может показаться, что динамическое измерение времени исполнения программы – тривиальная задача, которая имеет следующий алгоритм реше- ния:

* + 1. Запомнить время начала исполнения.
    2. Выполнить исследуемый код.
    3. Запомнить время окончания исполнения.
    4. Вычислить разность значений, полученных на шаге 3 и 1.

Однако такой алгоритм не учитывает множество особенностей выпол- нения кода современными JVM. Далее приводится описание основных осо- бенностей и возможных вариантов их решения. Более полное описание и об- основание можно найти в работе Брэнта Боера [41][42].

* + 1. «Разогрев» JVM

Важно, чтобы измерения производились в стационарном состоянии JVM, а не сразу после ее запуска, поскольку в общем случае начальная произ- водительность довольно невелика и растет дискретно до тех пор, пока не до- стигнет стационарного состояния. Таким образом, необходимо учесть те факторы, которые оказывают влияние на достижения этого самого стационар-

ного состояния.

*Загрузка классов*

Загрузка классов в большинстве JVM производится лениво. Таким об- разом, первое выполнение исследуемой задачи повлечет за собой загрузку классов, которые в ней используются (если они ранее не были загружены). Загрузка классов включает в себя обращения к внешней памяти ЭВМ, синтак- сический анализ и верификацию загруженных данных, что в совокупности может значительно завысить время первого исполнения задачи. Данная проблема может быть обычно решена многократным исполнением задачи. Исключением являются задачи, имеющие сложную структуру ветвления, в которой некоторые ветви выполняются относительно редко, что может приве- сти к тому, что даже после многократного исполнения некоторые классы не будут загружены. Однако, многократное исполнение по крайней мере мини- мизирует вероятность такой ситуации.

Имеет смысл определять требуемое для «прогревания» JVM количество выполнений задачи динамически. Интерфейс *ClassLoadingMXBean* предо- ставляет методы для определения количества загруженных и выгруженных классов [43], можно считать, что стационарное состояние (по загрузке классов) достигнуто, когда эти показатели после очередного исполнения перестанут изменяться.

*Смешанный режим выполнения*

Современные JVM обычно некоторое время выполняют код в режиме интерпретации с целью сбора профилировочной информации, прежде чем выполнить just-in-time (JIT) компиляцию [44]. Для динамической оценки вре- мени исполнения это означает, что задача должна выполниться много раз, прежде чем она будет скомпилирована и достигнет стационарного состояния.

Таким образом, для измерения времени исполнения задачи в стационар- ном состоянии необходимо выполнить следующую последовательность ша- гов:

* + - 1. Выполнить задачу один раз, чтобы загрузить используемые в ней классы.
      2. Выполнить задачу достаточное количество раз, чтобы убедиться, что стационарное состояние достигнуто.
      3. Выполнить задачу еще несколько раз, чтобы получить оценочное вре- мя ее исполнения.
      4. Используя результат шага 3, вычислить *n* – количество выполнений задачи, суммарное время которых достаточно велико. Цель этого шага заклю- чается в том, чтобы избежать замеров очень малых отрезков времени, по- скольку это может привести к появлению ошибок измерений при использова- нии стандартных библиотечных функций, таких как *System.currentTimeMillis* и *System.nanoTime*.
      5. Измерить суммарное время *t* , затрачиваемое на выполнение задачи *n*

раз.

* + - 1. Вычислить искомую величину как *t* / *n*.

Однако выполнение шага 2 – нетривиальная задача, поскольку не суще-

ствует абсолютно надежного способа определения того, что этап профилиро- вания в JVM завершился. Возможный подход к решению этой задачи – испол- нение задачи в течение предопределенного достаточно продолжительного от- резка времени. 10 секунд для «прогрева» должно быть достаточно [41]. Дан- ный подход, естественно, не является абсолютно надежным, однако он прост в реализации и легко параметризуем: увеличивая время «прогрева» можно добиться большей точности, затрачивая, однако, больше времени на подго- товку к измерениям.

* + 1. Динамическая оптимизация

Помимо проблемы «разогрева», динамическая компиляция, производи- мая JVM, создает еще несколько особенностей, которые оказывают влияние на динамическую оценку времени исполнения. И, что хуже, некоторые из этих особенностей не могут быть решены средствами самой утилиты измере-

ния времени, они требуют решения со стороны прикладного программиста.

*Деоптимизация*

Одна из упомянутых выше особенностей – это деоптимизация [45]: JVM может перестать использовать скомпилированный ранее метод и вновь вернуться к интерпретации его кода, прежде чем произойдет повторная компиляция. Это может произойти, когда предположения, сделанные динами- ческим оптимизирующим компилятором стали неактуальными. Примером может служить ситуация, когда в скомпилированном блоке есть путь, вероят- ность выполнения которого на момент компиляции была оценена как низкая. Такие ветви не компилируются, а продолжают исполняться в режиме интер- претации. И если такая ветвь начинает выполняться часто, то это приводит к перекомпиляции блока.

*Устранение «мертвого» кода*

Другая возможная проблема кроется в устранении «мертвого» кода (англ. **Dead-code elimination**, DCE). При определенных обстоятельствах DCE может быть произведено статически, на этапе компиляции. Однако динамиче- ские оптимизирующие компиляторы имеют значительно больше возможно- стей для анализа и определения никогда не исполняющихся участков кода. Таким образом прикладному программисту необходимо избегать статическо- го и динамического DCE того участка кода, который исследуется на предмет времени его исполнения. Единственный гарантированный способ избежать DCE – это сделать так, чтобы исследуемый участок кода генерировал некото- рые результаты, которые бы впоследствии могли бы быть использованы (это следует из определения DCE). В случае Java или Scala этого можно достичь, если заставить исследуемый код, некоторым образом вычислять и сохранять результат в поле содержащего его объекта, при условии, что это поле исполь- зуется в методе *toString*.

* + 1. Утилизация ресурсов

Обычно JVM выполняет два типа утилизации ресурсов:

* сбор мусора (англ. garbage collection, GC);
* финализация объектов (англ. object finalization, OF).

С точки зрения прикладного программиста оба эти процесса практиче- ски недерерминированы и не поддаются прямому контролю.

При динамическом измерении времени исполнения должны быть учте- ны накладные расходы на GC/OF, если они вызваны самим исследуемым ко- дом. Например, некорректно утверждать, что задача выполняется быстро, только на основе того, что ее первое выполнение заняло мало времени, по- скольку при последующих выполнениях может запуститься сборка мусора, которая внесет значительные корректировки в итоговое время выполнения.

С другой стороны необходим изолировать сборки мусора, относящиеся к исследуемой задаче от сборок мусора, связанных с другим кодом в пределах той же сессии JVM. Для достижения этой цели можно «очищать» JVM пред выполнением замеров.

* + 1. Выводы

Таким образом, для получения достоверных измерений необходимо учитывать особенности JVM, описанные в этом разделе. Для этого была ис- пользована библиотека динамической оценки времени исполнения кода на JVM, описание принципов работы которой, а также обоснование их коррект- ности приведено в работах [41] и [42]. Помимо непосредственной оценки времени исполнения программы данная библиотека также позволяет на осно- ве данных, полученных при многократном исполнении программы, опреде- лить и другие статистические показатели, например, стандартное отклонение. Стоит также заметить, что при написании кода, который являлся объектом исследования, были приняты меры, описанные в пункте [3.1.2](#_bookmark67), в целях предот- вращений DCE.

### Экспериментальная проверка точности

* + 1. Планирование эксперимента

В целях верификации точности метода был проведен модельный экспе- римент. Модель простой программы, состоящей из инициализации, за кото- рой следует цикл со вложенной конструкцией ветвления, приведена ниже:

**class** ProgramModel**(**Configuration config**) extends** Runnable **{ private val** loopBoundDistribution **=**

config**.**createLoopBoundDistribution

**private val** trueBranchProbability **=** config**.**trueBranchProbability

**private var** v **= 0**D

**def** run**(): Unit = { val** rand **= new** Random

**val** lb **=** loopBoundDistribution**.**sample**()** v **= 1.1**

**var** i **= 0**

**while (**i **<** lb**) {**

**if (**rand**.**nextDouble **<=** trueBranchProbability**) {**

v **+=** Math**.**pow**(**v**, 2)**

**}**

**else {**

v **+=** Math**.**log**(**v**)**

**}**

i **+= 1**

**}**

**}**

**override def** toString**: String =** "v=" **+** v

**}**

Выбор структуры моделируемой программы основывался на следую-

щих критериях:

* необходимость включить в нее все три простейшие конструкции

управления: последовательное исполнение, ветвление и цикл – которые соот- ветствуют трем операциям над РВ: конкатенации, альтернации и итерации;

* сохранение простоты модели в целях снижения вероятности появле-

ния в ней ошибок;

* близость структуры моделируемой программы к простейшим реаль- ным программам.

Граф потока управления моделируемой программы представлен на ри- сунке [3.1](#_bookmark72).

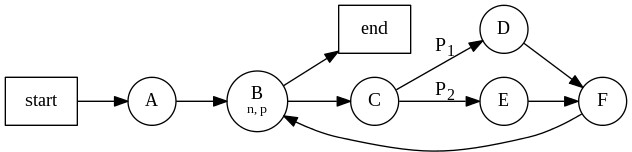


Рисунок 3.1 - ГПУ моделируемой программы

В реальных программах количество итераций цикла, а также выбор вет- ви условного оператора, в большинстве случаев зависит от набора входных данных. Для моделирования данного поведения был использован генератор псевдослучайных чисел.

Значение переменной *lb*, определяющей количество итераций цикла, генерировалось, исходя из предположения о том, что соответствующая слу- чайная величина распределена по биномиальному закону. Биномиальное рас- пределение – дискретное распределение вероятностей случайной величины *X*

, принимающей целочисленные значения *k* =0, 1,…*, n* с вероятностями

*P*( *X*=*k* )=*Ck pk* (1− *p* )*n*−*k*.

*n*

Данное распределение характеризуется двумя параметрами: целым чис-

лом *n* >0, называемым числом испытаний, и вещественным числом *p*, 0≤ *p*≤1, называемом вероятностью успеха в одном испытании. Биномиаль-

ное распределение – одно из основных распределений вероятностей, связан- ных с последовательностью независимых испытаний [46]. Если проводится серия из *n* независимых испытаний, в каждом из которых может произойти

«успех» с вероятностью *p*, то случайная величина, равная числу успехов во всей серии, имеет указанное распределение. Именно поэтому для количества итераций было выбрано данное распределение: программный цикл можно рассматривать как последовательность независимых (в первом приближении) проверок условия выхода из цикла, каждая из которых может закончится либо

«успехом», либо «неудачей».

В рассматриваемой модели выбор положительной ветви условного опе-

ратора осуществлялся с некоторой вероятностью *P*1. Данная величина, как и параметры *n* и *p* задавались в объекте конфигурации *config*, передаваемом в конструктор класса.

В рамках эксперимента были проведены две серии испытаний, в кото- рых варьировались параметры *n* и *p*. Параметры испытаний приведены в та- б[лице 1](#_bookmark73).

Таблица 1 - Параметры испытаний

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Серия испытаний | Испытание 1 | | Испытание 2 | | Испытание 3 | |
| n | p | n | p | n | p |
| 1 | 103 | 0,3 | 103 | 0,5 | 103 | 0,7 |
| 2 | 106 | 0,3 | 106 | 0,5 | 106 | 0,7 |

Для оценки точности получаемых результатов в качестве эталона ис- пользовались значения, полученные в ходе динамической оценки времени ис- полнения указанного фрагмента кода. Стоит заметить, что в целях приближе- ния экспериментальных результатов к реальным, для статической оценки вре- мени исполнения использовались не точные значения математического ожи- дания и дисперсии количества итераций цикла, которые можно было бы по- лучить, исходя из заданных параметров биномиального распределения по формулам

*M* =*np , D* =*npq ,*

а их статистические приближения.

Данные приближения получались на основе первых 60 сгенерирован- ных значений переменной *lb*, которая характеризует количество итераций цикла. Такое значение (60) обусловлено принципом работы библиотеки, при помощи которой производилась динамическая оценка времени исполнения: она делает замеры как минимум для 60 «прогонов» программы, а для каждого из них генерируется собственное значение переменной *lb*.

Таблица 2 - Характеристики системы

|  |  |  |
| --- | --- | --- |
| Характеристика | | Значение |
| Аппаратная конфигурация | CPU | Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz |
| RAM | 2x2GiB, 1333 MHz, DDR3 |
| Операционная система | | GNU/Linux 3.10.32-2-MANJARO |
| JRE | Версия | 1.7.0\_51 with Java HotSpot(TM) 64-Bit Server VM |
| Опции | -Xmx1024m (максимальный размер кучи)  -Xms256m (начальный размер кучи) |

* + 1. Результаты эксперимента

В таблице [3](#_bookmark75) и таблице [4](#_bookmark76) приведены результаты эксперимента.

Таблица 3 - Оценки среднего времени исполнения программы

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Серия испытаний | Испытание 1 | | Испытание 2 | | Испытание 3 | |
| СОВИ | ДОВИ | СОВИ | ДОВИ | СОВИ | ДОВИ |
| 1 (мкс) | 175,496 | 165,217 | 284,592 | 273,787 | 393,633 | 380,459 |
| 2 (мс) | 163,623 | 161,561 | 272,693 | 268,843 | 381,760 | 376,545 |

Таблица 4 - Оценки стандартного отклонения времени исполнения программы

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Серия испытаний | Испытание 1 | | Испытание 2 | | Испытание 3 | |
| СОВИ | ДОВИ | СОВИ | ДОВИ | СОВИ | ДОВИ |
| 1 (мкс) | 24,092 | 23,424 | 30,298 | 29,563 | 35,090 | 33,367 |
| 2 (мс) | 0,740 | 1,099 | 0,940 | 0,648 | 1,095 | 1,135 |

Для большей наглядности результаты эксперимента приведены на ри- сунка[х 3.2](#_bookmark77), [3.3](#_bookmark78), [3.4](#_bookmark79) и [3.5](#_bookmark80) в виде гистограмм:

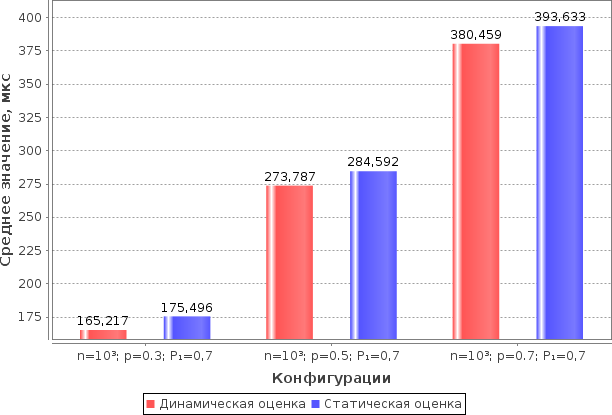


Рисунок 3.2 - Среднее время исполнения исследуемой программы (1-ая серия испытаний)

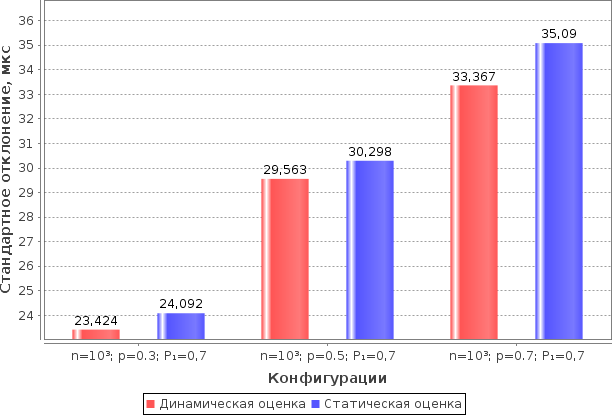


Рисунок 3.3 - Стандартное отклонение времени исполнения исследуемой программы (1-ая серия испытаний)

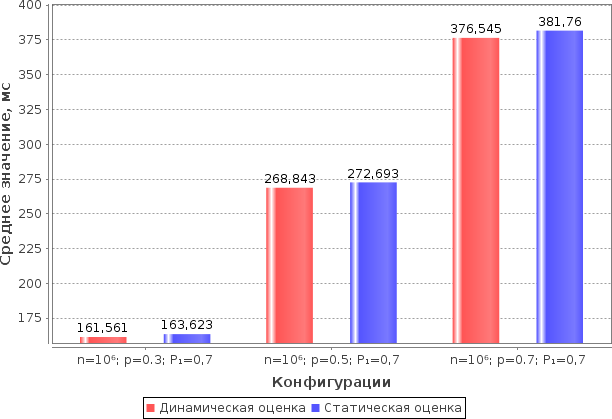


Рисунок 3.4 - Среднее время исполнения исследуемой программы (2-ая серия испытаний)

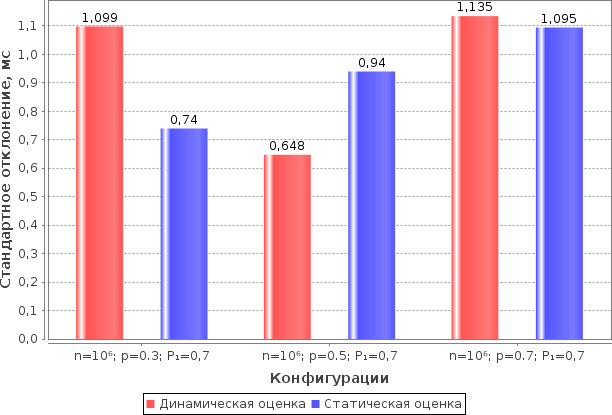


Рисунок 3.5 - Стандартное отклонение времени исполнения исследуемой программы (2-ая серия испытаний)

На основе полученных результатов можно заключить, что предлагае- мый метод позволяет получить достаточно точную оценку сверху среднего времени исполнения программы. Оценки же стандартного отклонения не все- гда ограничивают сверху значения, получаемые при динамической оценке. Однако этот факт можно объяснить тем, что существует большое количество факторов, значительно влияющих на получаемую при динамической оценке величину стандартного отклонения и, соответственно, вносящих серьезные погрешности в этот показатель [41].

### Экспериментальная проверка производительности

* + 1. Планирование эксперимента

В ходе эксперимента проводилось сравнение производительности сле- дующих алгоритмов построения РВ по ГПУ:

* + - * «стандартного» алгоритма, описанного в разделе [2.1.2](#_bookmark40), далее будем

для краткости называть его АС;

* + - * модификации, в которой на основе транзитивного замыкания графа

отсеиваются несуществующие пути, далее будем для краткости называть его АТЗ;

* + - * модификации, в которой производится запоминание промежуточных

результатов, далее будем для краткости называть его АЗ;

* + - * модификации, включающей одновременно и использование транзи-

тивного замыкания и запоминание промежуточных результатов, далее будем для краткости называть его АЗТЗ.

Построение транзитивного замыкания осуществлялось посредством ал- горитма Флойда-Варшалла.

Для оценки производительности алгоритмов построения РВ был разра- ботан и реализован алгоритм генерации ГПУ структурных программ. Под структурностью в данном случае понимается то, что программа соответству- ет основному принципу структурного программирования [47], то есть по- строена путем вложения друг в друга конструкций трех типов:

* + - * последовательное исполнение – однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
      * ветвление – однократное выполнение одной из двух или более опе- раций, в зависимости от выполнения некоторого заданного условия;
      * цикл – многократное исполнение одной и той же операции до тех

пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

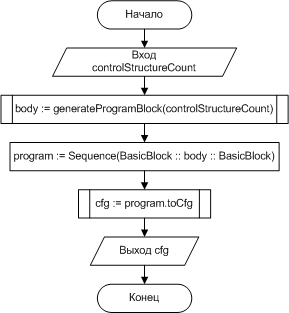
Схема алгоритма генерации ГПУ для структурных программ приведена на рисунке [3.6](#_bookmark83).

Рисунок 3.6 - Схема алгоритма генерации «структурных» ГПУ

Входным параметром для алгоритма является количество управляющих конструкций в генерируемом ГПУ. Помимо этого можно было бы также ва- рьировать количество ветвей в условном операторе и количество блоков в операторе последовательного выполнения, но здесь для простоты они рассматриваются как глобальные константы *branchCount* и *seqLength* Генери- руемая программа по сути представляет последовательность из трех блоков: вход, тело программы и выход.

Схемы функций, участвующих в алгоритме приведены на рисунках [3.7](#_bookmark84), [3.8](#_bookmark85) и [3.9](#_bookmark86).

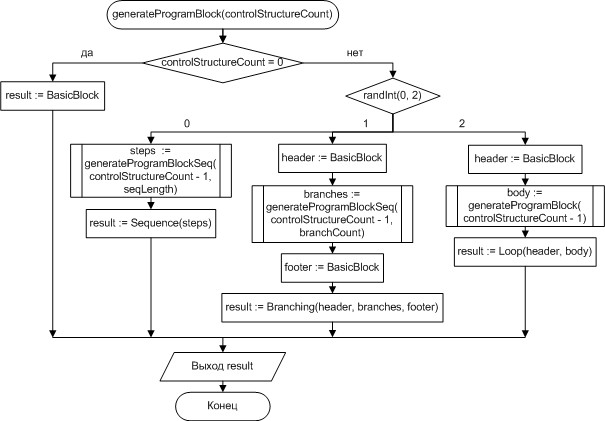


Рисунок 3.7 - Схема алгоритма генерации программного блока

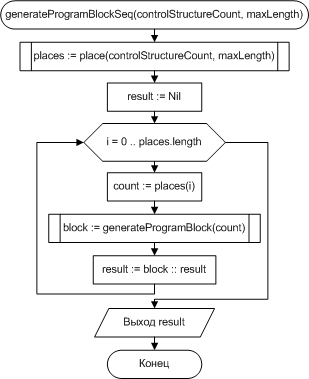


Рисунок 3.8 - Схема алгоритма генерации последовательности программных блоков

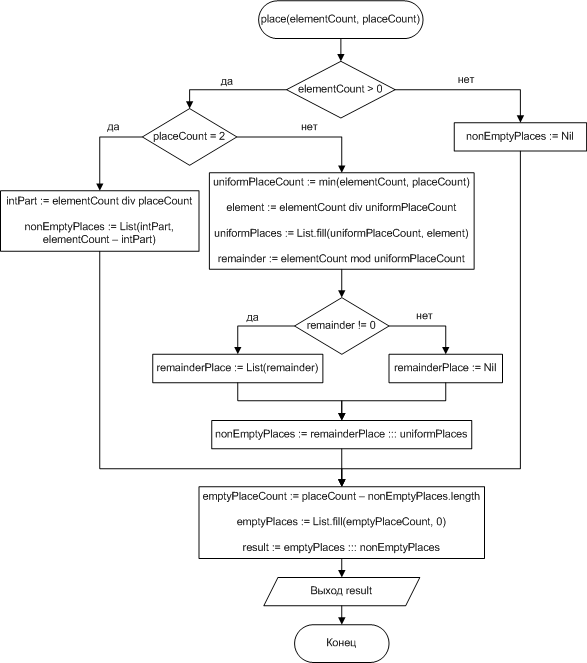


Рисунок 3.9 - Схема алгоритма размещения n элементов по m позициям

В схемах на рисунках [3.6](#_bookmark83), [3.7](#_bookmark84), [3.8](#_bookmark85) и [3.9](#_bookmark86) использовались следующие обозначения:

* + - * div – операция целочисленного деления;
      * *mod* – операция взятия остатка от деления;
      * *Nil* – пустой список;
      * *List* ( *x*1, *x*2,…*, xn*) – список из элементов *x*1 *, x*2*,*…*, xn*;
      * *List* . *fill* (*n , v* ) – список из *n* элементов, имеющих значение *v*;
      * *a* : : *B* – добавление элемента *a* в начало списка *B*;
      * *A* : : : *B* – конкатенация списков *A* и *B*.

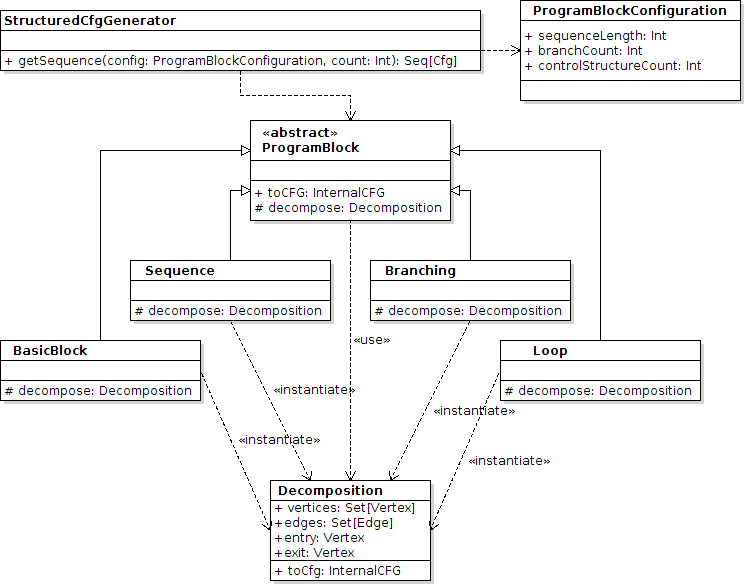
На рисунке [3.10](#_bookmark87) приведена диаграмма классов, участвующих в генера- ции ГПУ. Для каждого типа управляющих конструкций существует отдель- ный класс, наследующий абстрактному классу ProgramBlock. Объект каждого из этих классов может декомпозировать себя, т. е. представлять соответствую- щий программный блок в виде подграфа ГПУ.

Рисунок 3.10 - Диаграмма классов, используемых алгоритмом генерации ГПУ

В качестве характеристики генерируемых структурных ГПУ можно рассматривать количество базовых управляющих конструкций, из которых они состоят (входной параметр алгоритма генерации). Именно эта характери- стика была выбрана в качестве варьируемого параметра при оценке произво- дительности. При этом для всех генерируемых ГПУ количество ветвей в конструкциях ветвления равнялось 2, а количество программных блоков в

конструкции последовательного выполнения равнялось 4.

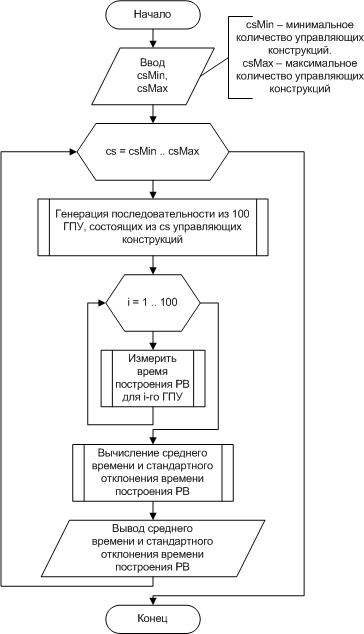
Очевидно, что для различных ГПУ, состоящих при этом из одинакового количества управляющих конструкций, время, необходимое для построения РВ, может значительно различаться. С учетом данной особенности алгоритм проведения испытаний имел вид, представленный на рисунке [3.11](#_bookmark88).

Рисунок 3.11 - Схема алгоритма измерения времени построения РВ

* + 1. Результаты эксперимента

В таблице [5](#_bookmark90) приведены результаты эксперимента.

Таблица 5 - Время построения РВ по ГПУ с различным количеством управляющих конструкций

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во управляющих конструкций | АС, мс | АТЗ, мс | АЗ, мс | АЗТЗ, мс |
| 5 | 12,484 | 2,263 | 0,458 | 1,307 |
| 6 | 39,701 | 3,627 | 0,600 | 1,843 |
| 7 | 491,260 | 22,701 | 1,252 | 2,976 |
| 8 | 1413,358 | 132,738 | 1,506 | 4,206 |
| 9 | 2881,361 | 727,526 | 2,837 | 5,761 |
| 10 | 6881,361 | 1202,333 | 3,192 | 7,583 |
| 11 | - | - | 3,835 | 9,295 |
| 12 | - | - | 5,084 | 11,314 |
| 13 | - | - | 5,942 | 14,584 |
| 14 | - | - | 9,527 | 17,723 |
| 15 | - | - | 11,217 | 19,666 |
| 16 | - | - | 11,542 | 25,796 |
| 17 | - | - | 17,353 | 31,502 |
| 18 | - | - | 22,967 | 34,135 |
| 19 | - | - | 35,777 | 46,279 |
| 20 | - | - | 45,918 | 56,930 |

На рисунках [3.12](#_bookmark91) и [3.14](#_bookmark93) приведены зависимости времени построения РВ по ГПУ от количества управляющих конструкций, из которых состоит этот ГПУ. На рисунках же [3.13](#_bookmark92) и [3.15](#_bookmark94) приведены зависимости стандартного отклонения времени построения РВ по ГПУ от количества управляющих конструкций.

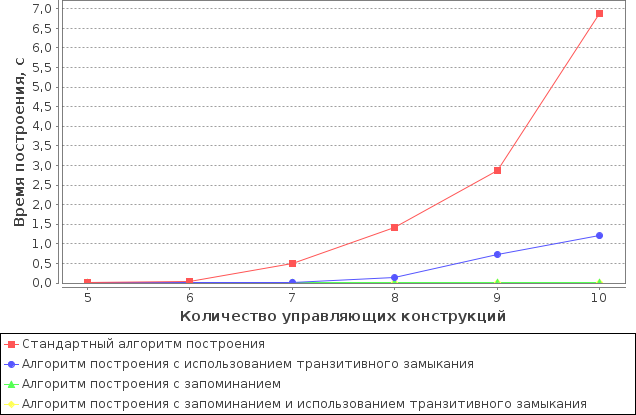


Рисунок 3.12 - Время построения РВ по ГПУ различными вариациями алгоритма

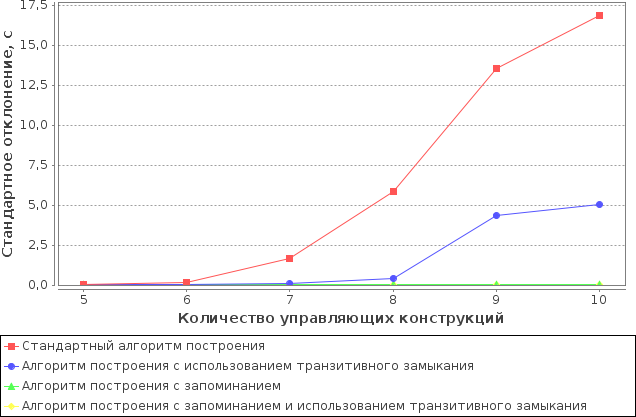


Рисунок 3.13 - Стандартное отклонение времени построения РВ по ГПУ различными вариациями алгоритма

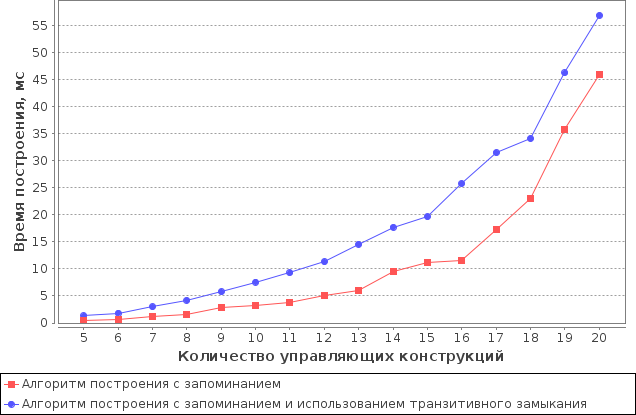


Рисунок 3.14 - Время построения РВ по ГПУ алгоритмами с запоминанием



Рисунок 3.15 - Стандартное отклонение времени построения РВ по ГПУ алгоритмами с запоминанием

На основе полученных результатов можно заключить, что для структур- ных графов с увеличением количества управляющих конструкций наблюдает- ся рост преимущества алгоритма, использующего транзитивное замыкание, перед стандартным алгоритмом в скорости построения РВ. Таким образом, оправдывается предположение о том, что для разреженных графов (к кото- рым можно отнести структурные) накладные расходы на построение транзи- тивного замыкания, в значительной степени окупаются снижением затрат времени на обработку несуществующих путей.

Однако более значительный прирост производительности дает запоми- нание промежуточных результатов. Так, для ГПУ из 10 управляющих конструкций алгоритм с запоминанием работает примерно в 2155,5 раз бы- стрее чем стандартный алгоритм, в то время, как алгоритм с транзитивным замыканием дает прирост всего в 5,7 раз.

По результатам исследования также видно, что для алгоритма с запоми- нанием дополнительное использование транзитивного замыкания не дает по- ложительного эффекта с точки зрения среднего времени выполнения (см. ри- сунок [3.14](#_bookmark93)): для количества управляющих конструкций в ГПУ из отрезка

[5,…*,*20] алгоритм с одновременным использованием запоминания и транзи-

тивного замыкания незначительно медленнее алгоритма, использующего лишь запоминание. Однако стоит также обратить внимание на стандартное отклонение (см. рисунок [3.15](#_bookmark94)). Видно, что с ростом количества управляющих конструкций, значительно растет и разброс времени построения регулярного выражения. В большей степени это отражается на алгоритме с запоминанием, который не использует транзитивное замыкание. Таким образом можно за- ключить, что использование транзитивного замыкания несколько увеличивает среднее время работы алгоритма, однако делает его менее зависимым от структуры входного ГПУ.

### Экспериментальная оценка потребления памяти

Как было сказано в разделе [2.1.3](#_bookmark45), упрощение РВ в ходе построения мо-

жет значительно снизить потребление памяти. В этом разделе приведена экс- периментальная проверка данного утверждения.

* + 1. Планирование эксперимента

Очевидно, что эффект от упрощения РВ «на лету» будет особенно хоро- шо заметен для больших ГПУ. По этой причине измерения проводились для ГПУ, состоящего из 1000 управляющих конструкций (описание алгоритма ге- нерации структурных ГПУ приведено в разделе [3.3.1](#_bookmark82)).

Построение РВ по ГПУ производилось при помощи алгоритма, исполь- зующего запоминание промежуточных результатов. По результатам экспери- ментальной проверки производительности данный алгоритм оказался наибо- лее эффективным (см. раздел [3.3](#_bookmark81)).

Программа построения РВ запускалась со следующими опциями JVM:

-Xmx1024m -Xms256m. Соответственно, максимально возможный объем ис- пользуемой программой кучи был ограничен 1 Гб, а объем первоначально вы- деленной памяти в куче составлял 256 Мб.

Для измерения объема потребляемой памяти использовалась утилита VisualVM, входящая в комплект поставки JDK и предназначенная для сбора подробной информации о приложениях, выполняющихся JVM [48].

Стоит отметить, что в данном случае под «потребляемой памятью» по- нимается не весь объем, занимаемый программой в оперативной памяти, а только та его часть, что относится к области кучи.

* + 1. Результаты эксперимента

Результаты измерения потребляемой программой памяти, а также ак- тивности ЦПУ и сборщика мусора в ходе ее выполнения приведены ниже на рисунках [3.16](#_bookmark98) и [3.17](#_bookmark99) (для реализации без упрощения) и на рисунках [3.18](#_bookmark100) и

[3.19](#_bookmark101) (для реализации с упрощением).

Видно, что в реализации без упрощения объем потребляемой памяти растет вплоть до 800 Мб, после чего программа аварийно завершается с ошибкой JVM java.lang.OutOfMemoryError: GC overhead limit exceeded. Дан-

ная ошибка связана с тем, что в условиях нехватки памяти сборщик мусора начинает активно работать, освобождая место, однако это не дает должного результата (на рисунке [3.17](#_bookmark99) отчетливо видно значительное увеличение актив- ности сборщика мусора после 4:38:30 PM).

В тоже время стоит отметить, что реализации с упрощением завершает- ся корректно (без исчерпания памяти). При этом объем потребляемой памяти большую часть времени не превышает 100 Мб. Активность сборщика мусора незначительна, следствием этого является меньшая нагрузка на ЦПУ, что в целом положительно влияет на производительность программы.

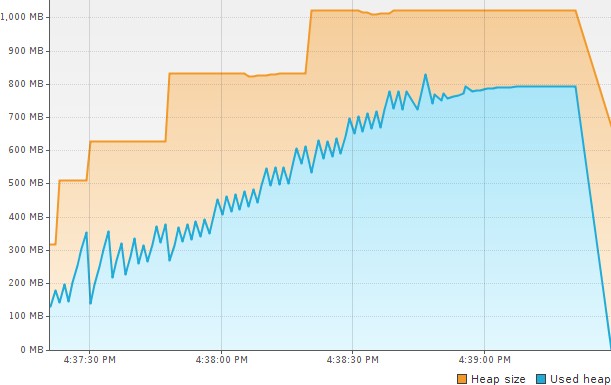
Итак, в итоге экспериментальной проверки было подтверждено предпо- ложение о том, что упрощение промежуточных выражений в ходе построения РВ дает положительный эффект в виде значительного снижения потребляе- мой памяти, а также в виде снижения нагрузки на ЦПУ.

Рисунок 3.16 - График потребления памяти при построении РВ без применения упрощения РВ

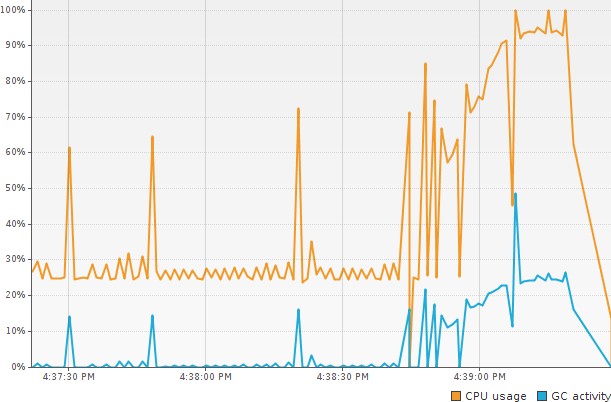


Рисунок 3.17 - График активности ЦПУ и сборщика мусора при построении РВ без применения упрощения РВ

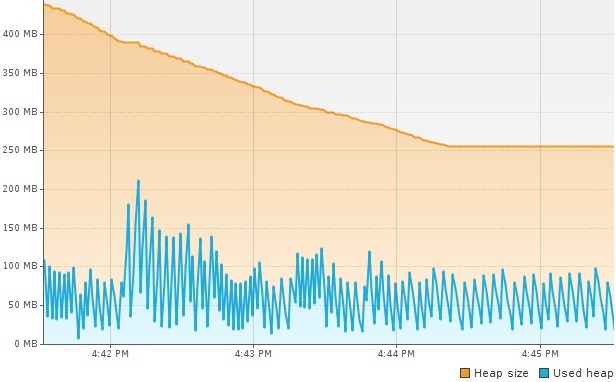


Рисунок 3.18 - График потребления памяти при построении РВ с применением

упрощения РВ

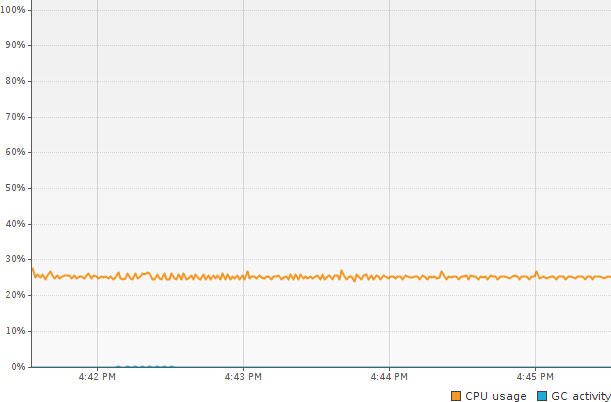


Рисунок 3.19 - График активности ЦПУ и сборщика мусора при построении РВ с применением упрощения РВ

### 3.5 Практические рекомендации

Оценка затрат времени на исполнение программы крайне важна в рабо- те систем распределенных вычислений. В подобных распределенных систе- мах крайне важно наличие эффективного способа планирования внутри сети. Например, в сети добровольных вычислений, построенных на базе платфор- мы BOINC, от того, насколько быстро выполняются задания, зависит их ко- личество, которое будет выслано при следующем обращении к серверу. Та- ким образом, предложенный в данной работе метод мог бы найти применение при определении наиболее вероятного времени выполнения узлом сети сети распределенных вычислений своей доли работы.

Также возможной областью применения результатов прогнозирования времени выполнения программ являются вычислительные системы реально- го времени. Такие системы, как правило, участвуют в управлении некоторым реальным процессом и должны обладать одним чрезвычайно важным свой- ством – вычислитель должен не только правильно реализовывать алгоритм,

но обязан делать это в заданный отрезок времени. В зависимости от допусти- мых нарушений временных ограничений системы реального времени можно поделить на системы жесткого реального времени, для которых нарушения равнозначны отказу системы, и системы мягкого реального времени, наруше- ния характеристик которых приводят лишь к снижению качества работы си- стемы. Стоит заметить, что требования к системам мягкого реального време- ни можно задать только в вероятностных терминах, например, как процент откликов, выданных в установленные временные рамки. Интересно, что при проектировании предварительные расчеты легче выполнить для системы жесткого реального времени, чем получить, например, долю выполняемых в срок задач в системе мягкого реального времени, поэтому разработчики таких систем часто пользуются инструментами и методиками для проектирования систем жесткого реального времени.

Предлагаемый в данной работе метод оценки времени исполнения про- грамм не может быть применен в системах жесткого реального времени без внесения в него определенных корректировок, позволивших бы оценивать с его помощью наихудшее время выполнения. Однако, вероятностный подход, на котором он основывается, делает его отлично подходящим для применения в рамках систем мягкого реального времени.

Таким образом, вместе с системами реального времени предлагаемый метод мог бы найти применения в самых различных областях. Так, особенно широко системы реального времени применяются в промышленности, вклю- чая системы управления технологическими процессами, системы промыш- ленной автоматики, SCADA-системы, испытательное и измерительное обо- рудование, робототехнику. Применения в медицине включают в себя томо- графию, оборудование для радиотерапии, прикроватный мониторинг. Систе- мы реального времени встроены в периферийные устройства компьютеров, телекоммуникационное оборудование и бытовую технику, такую как лазер- ные принтеры, сканеры, цифровые камеры, кабельные модемы, маршрутиза-

торы, системы для видеоконференций и интернет-телефонии, мобильные телефоны, микроволновые печи, музыкальные центры, кондиционеры, систе- мы безопасности. На транспорте системы реального времени применяются в бортовых компьютерах, системах регулирования уличного движения, управ- лении воздушного движения, аэрокосмической технике, системе бронирова- ния билетов и т. п. Помимо этого, системы реального времени находят при- менения и в военной технике: системах наведения ракет, противоракетных системах, системах спутникового слежения.

### Выводы по главе 3

Проведены эксперименты по трем направлениям:

* проверка точности;
* проверка производительности
* оценка потребления памяти.

Экспериментально подтверждено, что предлагаемый метод статической ОВЗИП позволяет получить достаточно точную оценку сверху среднего време- ни исполнения программы: отклонение полученных значений от эталонных (полученных при динамической оценке) для исследуемой модели программы составляет 1,3%-6,2%. Также выявлено, что оценки стандартного отклонения не всегда ограничивают сверху значения, получаемые при динамической оценке, что можно объяснить большим количеством факторов, вносящих се- рьезные погрешности в величину стандартного отклонения при динамиче- ской оценке.

В ходе экспериментальной проверки производительности выявлено, что для структурных графов исходная версия алгоритма построения РВ по ГПУ неприемлемо медленна: для ГПУ из 10 управляющих конструкций среднее время построения составляет 6,881 с. Однако предложенные в разделе [2.1.3](#_bookmark45) способы оптимизации дают значительный прирост производительности. В частности, для ГПУ из 10 управляющих конструкций запоминание промежу- точных результатов позволяет снизить среднее время построения примерно в

2155,5 раз, а использование транзитивного замыкания – в 5,7 раз. Помимо этого выявлено, что для алгоритма с запоминанием дополнительное исполь- зование транзитивного замыкания несколько увеличивает среднее время ра- боты алгоритма, однако делает его менее зависимым от структуры входного ГПУ (это выражается в снижении дисперсии времени построения РВ).

В результате экспериментальной оценки потребления памяти было под- тверждено предположение о том, что упрощение промежуточных выражений в ходе построения РВ дает положительный эффект в виде значительного сни- жения потребляемой памяти (примерно в 10 раз), а также в виде снижения на- грузки на ЦПУ.

Помимо всего вышеизложенного, были описаны возможные области применения описанного в данной работе метода статической ОВЗИП.

# Заключение

В ходе диссертационного исследования были получены следующие ре- зультаты:

* проведен аналитический обзор существующих методов ОВЗИП;
* произведена формализация постановки задачи статической ОВЗИП на основе вероятностного подхода;
* разработан метод статической ОВЗИП на основе вероятностного под-

хода;

* разработана архитектура системы ОВЗИП и программно реализован

ее модуль расчета времени исполнения;

* + предложены методы оптимизации процесса построения РВ по ГПУ и

экспериментально показана их эффективность: использование транзитивного замыкания и запоминание промежуточных результатов позволяют ускорить процесс построения РВ по ГПУ в 5,7 и 2155,5 раз соответственно;

* + посредством разработанного метода получены оценки математиче-

ского ожидания и дисперсии времени исполнения программы с отклонением от эталонных значений 1,3%-6,2% и 5%-110% соответственно.

## Список литературы

1. Client scheduling policies [wiki] // Официальный сайт проекта BOINC: сайт. UR[L: http://boinc.berkeley.edu/trac/wiki/ClientSched](http://boinc.berkeley.edu/trac/wiki/ClientSched)
2. Капитонова А. П. Методы и средства прогнозирования времени выполнения последовательных фрагментов программ на вычислителях с различной архитектурой: дис. канд. физ.-мат. наук. МГУ, Москва, 1997.
3. Christian Ferdinand. Worst Case Execution Time Prediction by Static Program Analysis. Paper presented at the 18th International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, USA, 2009.
4. Reinhard Wilhelm et al. The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools // ACM Transactions on Embedded Computing Systems. - 2008.
5. Смелянский P. Л. Методы анализа и оценки производительности вычислительных систем - М.: МГУ, 1990.
6. Davidson J. W., Rabung J. R., Whalley D. B.. Relating Static and Dyanamic Machine Code Measurements // IEEE Trans, on Computers. - 1992, №4. c. 444- 454.
7. Касперски, К. Техника оптимизации программ. Эффективное использование памяти - Спб.: BHV, 2003. - 560.
8. R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst- Case Timing Analysis of SPARK Ada. ACM SIGPLAN Workshop onLanguages, Compilers and Tools for Real-Time System, 1994.
9. F. Stappert and P. Altenbernd. Сomplete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs // Journal of Systems Architecture. - 2000,

№46(4). c. 339–355.

1. A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction // Journal of Real-Time Systems. - 2000, №18(2/3). c. 249– 274.
2. Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level

languages. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, 1998.

1. C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. 4th IEEE Real-Time Technology and Applications Symposium, 1998.
2. N. Holsti, T. Langbacka, and S. Saarinen. Worst-Case Execution-Time Analysisfor Digital Signal Processors. EUSIPCO2000 Conference (X European Signal Processing Conference), 2000.
3. M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables, detectingand classifying sequences using a demand-driven ssa form // AVM Transactionson Programming Languages and Systems. - 1995, №17(1). c. 85–122.
4. Ermedahl, A. A Modular Tool Architecture for Worst-Case Execution Time Analysis: PhD dissertation. Uppsala University, Dept. of Information Technology, Uppsala, Sweden, 2003.
5. C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. 11th IEEE Real-Time Systems Symposium, 1990.
6. S. S. Lim et al. Accurate Worst-Case Timing Analysis for RISC Processors // IEEE Transactions on Software Engineering. - 1995, №21(7). c. 593–604.
7. R. Chapman Static Timing Analysis and Program Proof: PhD thesis. University of York, York, England, 1995.
8. C. Healy, R. Arnold, F. Muller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance // IEEE Transactions on Computers. - 1999, №48(1).
9. Y-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. 32nd Design Automation Conference, 1995.
10. C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. ACM SIGPLAN Workshop on Languages,Compilers and Tools for Real-Time System, 1997.
11. G. Ottosson and M. Sjodin. Worst-Case Execution Time Analysis for Modern

Hardware Architectures. ACM SIGPLAN Workshop on Languages,Compilers and Tools for Real-Time System, 1997.

1. F. Wolf and R. Ernst. Execution Cost Interval Refinement in Static Software Analysis // Journal of Systems Architecture, The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design. - 2001, №47(3-4). c. 339–356.
2. Дорогов В.Г., Дорогова Е.Г., Гагарина Л.Г., Портнов Е.М., Теплова Я.О. Математическая модель пациент-ориентированной системы информационной поддержки для лиц с ограниченными возможностями здоровья // Актуальные проблемы современной науки. - 2013, №2. c. 124-127.
3. C. Healy, R. Arnold, F. Muller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance // IEEE Transactions on Computers. - 1999, №48(1).
4. N. Holsti, T. Langbacka, and S. Saarinen. Worst-Case Execution-Time Analysisfor Digital Signal Processors. EUSIPCO2000 Conference (X European Signal Processing Conference), 2000.
5. Kleene, S. C. Representation of events in nerve nets and finite automata // Automata Studies. - 1956, №34. c. 3-40.
6. Janusz A. Brzozowski. Derivatives of regular expressions // ACM. - 1964,

№11(4). c. 481–494.

1. Ding-Shu Du, Ker-I Ko. Problem Solving in Automata, Languages, and Complexity - New York: John Wiley & Sons, 2001.
2. Peter Linz. An introduction to Formal Languages and Automata - Sudbury: Jones and Bartlett Publishers, 2001.
3. E. F. Moore. Gedanken experiments on sequential machines // Automata Studies. - 1956, №34. c. 129–153.
4. Dean N. Arden. Delayed-logic and finite-state machines // Theory ofComputing Machine Design. - 1960, №. c. 1-35.
5. How to convert finite automata to regular expressions? [Q&A] // Computer

Science Stack Exchange: сайт. URL: <http://cs.stackexchange.com/questions/2016/how-to-convert-finite-automata-to-> regular-expressions

1. Tim Teitelbaum. Introduction to Compilers. Lecture 24: Control Flow Graphs. March, 2008 [эл. документ] // Cornell University, Department of Computer Science: сайт. URL: <http://www.cs.cornell.edu/courses/cs412/2008sp/lectures/lec24.pdf>
2. John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation - : Prentice Hall, 2006. - 750.
3. Sipser, Michael. Introduction to the Theory of Computation - Boston: Cengage Learning, 2012. - 482.
4. Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ - М.: «Вильямс», 2011. - 1296.
5. Guido van Rossum. Python Patterns — Implementing Graphs [статья] // Официальный сайт языка Python: сайт. URL: https[://www.python.org/doc/essays/graphs/](http://www.python.org/doc/essays/graphs/)
6. Michael T. Goodrich, Roberto Tamassia. Algorithm Design. Foundations, Analysis, and Internet Examples - New York: Wiley, 2001. - 720.
7. Душкин Р. Алгебраические типы данных и их использование в программировании // Практика функционального программирования. - 2009,

№2. c. 85.

1. Boyer B. Robust Java benchmarking (Part 1: Issues, Part 2: Statistics and solutions) [статья] // IBM developerWorks: сайт. URL: <http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html>
2. Brent Boyer. Robust Java benchmarking (supplementary material) [] // Elliptic Group, Inc.: сайт. URL: <http://www.ellipticgroup.com/html/benchmarkingArticle.html>
3. Interface ClassLoadingMXBean [документация] // Java API Documentation: сайт. URL:

<http://docs.oracle.com/javase/7/docs/api/java/lang/management/ClassLoadingMX> Bean.html

1. Just-in-time compilation [wiki] // Wikipedia: сайт. URL: <http://en.wikipedia.org/wiki/Just-in-time_compilation>
2. Brian Goetz. Java theory and practice: Dynamic compilation and performance measurement [статья] // IBM developerWorks: сайт. URL: <http://www.ibm.com/developerworks/library/j-jtp12214/>
3. Ширяев А.Н. Вероятность - M.: МЦНМО, 2004. - 520.
4. Böhm, C. and Jacopini, G. Flow diagrams, Turing machines and languages with only two formation rules // Communications of the ACM. - May 1966, №5. c. 366-371.
5. Getting Started with VisualVM [статья] // Официальный сайт VisualVM: сайт. UR[L: http://visualvm.java.net/gettingstarted.html](http://visualvm.java.net/gettingstarted.html)

## Приложение

*Файл Cfg.scala*

**package** ru.miet.xtestimator.cfg

**import** ru.miet.xtestimator.StochasticVariable

**import** ru.miet.xtestimator.cfg.Cfg.**{**Edge**,** Vertex**}**

**trait** Cfg **{**

**def** vertices**: Set[Vertex] def** edges**: Set[Edge]**

**def** entry**: Vertex def** exit**: Vertex**

**def** getIncidentEdges**(**vertex**: Vertex): Seq[Edge]**

**def** isAdjacent**(**i**: Vertex,** j**: Vertex): Boolean**

}

**object** Cfg **{**

**def** apply**(**vertices**: Set[Vertex],** edges**: Set[Edge],** entry**: Vertex,** exit**: Vertex): Cfg = new** IncidenceListCfg**(**vertices**,** edges**,** entry**,** exit**)**

**case class** Vertex**(**id**: String,** executionTime**: StochasticVariable,**

loopBound**: Option[StochasticVariable] =** None**) {**

**def this(**id**: String,** executionTime**: StochasticVariable,** loopBound**: StochasticVariable) = this(**id**,** executionTime**,** Some**(**loopBound**))**

**override def** toString**: String =** id

**}**

**object** Vertex **{**

**def** apply**(**id**: String,** executionTime**: StochasticVariable,** loopBound**: StochasticVariable): Vertex = new** Vertex**(**id**,** executionTime**,** loopBound**)**

**}**

**case class** Edge**(**source**: Vertex,** target**: Vertex,** probability**: Double**

**= 1) {**

**override def** toString**: String =** s"$source -> $target"

**}**

**}**

*Файл IncidenceListCfg.scala*

**package** ru.miet.xtestimator.cfg

**import** ru.miet.xtestimator.cfg.Cfg.**{**Edge**,** Vertex**}**

**private[cfg] final class** IncidenceListCfg**(val** vertices**: Set[Vertex], val** edges**: Set[Edge], val** entry**: Vertex, val** exit**: Vertex) extends** Cfg

**{**

**private val** vertexMap **= {**

**val** tmpMap **=** edges**.**toVector groupBy **(\_.**source**)**

vertices**.**map**(**v **=> (**v**,** tmpMap**.**getOrElse**(**v**,** Vector**.**empty**))).**toMap

**}**

**override def** getIncidentEdges**(**vertex**: Vertex): Seq[Edge] =**

vertexMap**(**vertex**)**

**override def** isAdjacent**(**i**: Vertex,** j**: Vertex): Boolean =**

vertexMap**(**i**).**exists**(\_.**target **==** j**)**

**override def** equals**(**other**: Any): Boolean =** other **match { case** that**: IncidenceListCfg =>**

vertexMap **==** that**.**vertexMap **&&**

entry **==** that**.**entry **&&**

exit **==** that**.**exit

**case \_ => false**

**}**

**override def** hashCode**(): Int = {**

**val** state **=** Seq**(**vertexMap**,** entry**,** exit**)** state**.**map**(\_.**hashCode**()).**foldLeft**(0)((**a**,** b**) => 31 \*** a **+** b**)**

**}**

**}**

*Файл regex.scala*

**package** ru.miet.xtestimator.regex

**import** ru.miet.xtestimator.cfg.Cfg.Vertex

**import** ru.miet.xtestimator.StochasticVariable

**import** ru.miet.xtestimator.regex.Repetition.Body

**import** ru.miet.xtestimator.regex.BatchAlternation.Branch

**sealed abstract class** Regex **{ def** simplify**: Regex = this**

**def** estimate**: StochasticVariable**

}

**final case class** Literal**(**value**: String,** executionTime**: StochasticVariable) extends** Regex **{**

**override def** estimate**: StochasticVariable =** executionTime

**override def** toString**: String =** value

}

**object** Literal **{**

**def** apply**(**vertex**: Vertex): Literal =** apply**(**vertex**.**id**,** vertex**.**executionTime**)**

}

**case object** EmptySet **extends** Regex **{**

**override def** estimate**: StochasticVariable =** StochasticVariable**.**Zero

**override def** toString**: String =** "Ø"

}

**case object** EmptyString **extends** Regex **{**

**override def** estimate**: StochasticVariable =** StochasticVariable**.**Zero

**override def** toString**: String =** "ε"

}

**final case class** Concatenation**(**lhs**: Regex,** rhs**: Regex) extends** Regex **{ override def** simplify**: Regex = {**

**(**lhs**,** rhs**) match {**

**case (**EmptySet**, \_) =>** EmptySet **case (\_,** EmptySet**) =>** EmptySet **case (**EmptyString**,** r**) =>** r **case (**r**,** EmptyString**) =>** r **case \_ => this**

**}**

**}**

**override def** estimate**: StochasticVariable = { val** lhsExecutionTime **=** lhs**.**estimate

**val** rhsExecutionTime **=** rhs**.**estimate

lhsExecutionTime **+** rhsExecutionTime

**}**

**override def** toString**: String =** s"($lhs) + ($rhs)"

}

**final case class** BatchAlternation**(**branches**: Seq[Branch]) extends** Regex

**{**

**override def** simplify**: Regex = {**

**val** filteredBranches **=** branches filter **(\_.**regex **!=** EmptySet**)** filteredBranches **match {**

**case** Seq**() =>** EmptySet

**case** Seq**(**head**) =>** head**.**regex

**case \_ =>** BatchAlternation**(**filteredBranches**)**

**}**

**}**

**override def** estimate**: StochasticVariable = {**

**val** tmpExecutionTime **=** branches**.**foldLeft**(**StochasticVariable**.**Zero**)**

**{**

**(**accumulator**,** branch**) => {**

**val** branchExecutionTime **=** branch**.**regex**.**estimate

**val** m **=** branch**.**probability **\*** branchExecutionTime**.**mean

**val** v **=** branch**.**probability **\* (**branchExecutionTime**.**variance **+**

Math**.**pow**(**branchExecutionTime**.**mean**, 2))**

accumulator **+** StochasticVariable**(**m**,** v**)**

**}**

**}**

tmpExecutionTime**.**copy**(**variance **=** tmpExecutionTime**.**variance **-**

Math**.**pow**(**tmpExecutionTime**.**mean**, 2))**

**}**

**override def** toString**: String =** branches**.**mkString**(**" | "**)**

}

**object** BatchAlternation **{**

**case class** Branch**(**regex**: Regex,** probability**: Double) { override def** toString**: String =** regex**.**toString

**}**

}

**final case class** Repetition**(**body**: Body) extends** Regex **{ override def** simplify**: Regex = {**

body**.**regex **match {**

**case** EmptySet **=>** EmptyString **case** EmptyString **=>** EmptyString **case \_ => this**

**}**

**}**

**override def** estimate**: StochasticVariable = {**

body**.**loopBound **match { case** Some**(**bound**) =>**

**val** bodyExecutionTime **=** body**.**regex**.**estimate

**val** m **=** bodyExecutionTime**.**mean **\*** bound**.**mean

**val** v **=** bodyExecutionTime**.**variance **\*** bound**.**mean **+**

Math**.**pow**(**bodyExecutionTime**.**mean**, 2) \*** bound**.**variance StochasticVariable**(**m**,** v**)**

**case** None **=> throw new** IllegalStateException**()**

**}**

**}**

**override def** toString**: String =** s"($body)\*"

}

**object** Repetition **{**

**case class** Body**(**regex**: Regex,** loopBound**: Option[StochasticVariable])**

**{**

**override def** toString**: String =** regex**.**toString

**}**

**}**

*Файл RegexImplicits.scala*

**package** ru.miet.xtestimator.regex

**import** ru.miet.xtestimator.StochasticVariable

**import** ru.miet.xtestimator.regex.Repetition.Body

**import** ru.miet.xtestimator.regex.BatchAlternation.Branch

**object** RegexImplicits **{**

**implicit final class** RegexAssoc**(**regex**: Regex) {**

**def +(**anotherRegex**: Regex): Regex =** Concatenation**(**regex**,** anotherRegex**).**simplify

**def \*(**loopBound**: Option[StochasticVariable]): Regex =**

Repetition**(**Body**(**regex**,** loopBound**)).**simplify

**def \*(**loopBound**: StochasticVariable): Regex =** regex **\***

Some**(**loopBound**)**

**}**

**implicit final class** BranchesAssoc**(**branches**: Seq[Branch]) { def** alternate**: Regex =** BatchAlternation**(**branches**).**simplify

**}**

**}**

*Файл RegexBuilder.scala*

**package** ru.miet.xtestimator.regex

**trait** RegexBuilder **{ def** build**: Regex**

**}**

Файл SimpleRegexBuilder.scala

**package** ru.miet.xtestimator.regex

**import** ru.miet.xtestimator.cfg.Cfg

**import** ru.miet.xtestimator.cfg.Cfg.Vertex

**import** ru.miet.xtestimator.regex.BatchAlternation.Branch

**import** RegexImplicits.\_

**final class** SimpleRegexBuilder**(**cfg**: Cfg) extends** RegexBuilder **{ def** build**: Regex =** buildRegex**(**Set**.**empty**,** cfg**.**entry**,** cfg**.**exit**) +**

Literal**(**cfg**.**exit**)**

**private def** buildRegex**(**forbiddenSet**: Set[Vertex],** source**: Vertex,**

target**: Vertex): Regex = {**

**val** newForbiddenSet **=** forbiddenSet **+** source

**val** loopPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** source**) \***

source**.**loopBound

**val** directPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** target**)** loopPart **+** directPart

**}**

**private def** buildRegexPart**(**forbiddenSet**: Set[Vertex],** source**: Vertex,** target**: Vertex): Regex = {**

**val** branches **= for {**

edge **<-** cfg**.**getIncidentEdges**(**source**)** neighbor **=** edge**.**target

**if (**neighbor **==** target**) || !(**forbiddenSet contains neighbor**)**

**} yield {**

**val** tail **= if (**neighbor **==** target**)** EmptyString **else**

buildRegex**(**forbiddenSet**,** neighbor**,** target**)** Branch**(**Literal**(**source**) +** tail**,** edge**.**probability**)**

**}**

**if (**branches**.**isEmpty**)** EmptySet **else** branches**.**alternate

**}**

}

*Файл RegexBuilderWithTransitiveClosure.scala*

**package** ru.miet.xtestimator.regex

**import** ru.miet.xtestimator.cfg.Cfg.Vertex

**import** ru.miet.xtestimator.regex.BatchAlternation.Branch

**import** ru.miet.xtestimator.cfg.Cfg

**import** RegexImplicits.\_

**final class** RegexBuilderWithTransitiveClosure**(**cfg**: Cfg) extends**

RegexBuilder **{**

**private val** transitiveClosure **=** TransitiveClosure**(**cfg**)**

**def** build**: Regex =** buildRegex**(**Set**.**empty**,** cfg**.**entry**,** cfg**.**exit**) +**

Literal**(**cfg**.**exit**)**

**private def** buildRegex**(**forbiddenSet**: Set[Vertex],** source**: Vertex,**

target**: Vertex): Regex =**

**if (**transitiveClosure**.**contains**(**source**,** target**)) { val** newForbiddenSet **=** forbiddenSet **+** source

**val** loopPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** source**) \***

source**.**loopBound

**val** directPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** target**)** loopPart **+** directPart

**}**

**else {**

EmptySet

**}**

**private def** buildRegexPart**(**forbiddenSet**: Set[Vertex],** source**: Vertex,** target**: Vertex): Regex = {**

**val** branches **= for {**

edge **<-** cfg**.**getIncidentEdges**(**source**)** neighbor **=** edge**.**target

**if (**neighbor **==** target**) || !(**forbiddenSet contains neighbor**)**

**} yield {**

**val** tail **= if (**neighbor **==** target**)** EmptyString **else**

buildRegex**(**forbiddenSet**,** neighbor**,** target**)**

Branch**(**Literal**(**source**) +** tail**,** edge**.**probability**)**

**}**

**if (**branches**.**isEmpty**)** EmptySet **else** branches**.**alternate

**}**

}

*Файл PackratRegexBuilder.scala*

**package** ru.miet.xtestimator.regex

**import** ru.miet.xtestimator.cfg.Cfg

**import** ru.miet.xtestimator.cfg.Cfg.Vertex

**import** ru.miet.xtestimator.regex.BatchAlternation.Branch

**import** ru.miet.xtestimator.regex.RegexImplicits.\_

**import** scala.collection.mutable

**final class** PackratRegexBuilder**(**cfg**: Cfg) extends** RegexBuilder **{ import** ru.miet.xtestimator.regex.PackratRegexBuilder.\_ **private[this] val** cache **=** mutable**.**WeakHashMap**[CacheKey**, **Regex]() def** build**: Regex =** buildRegex**(**Set**.**empty**,** cfg**.**entry**,** cfg**.**exit**) +**

Literal**(**cfg**.**exit**)**

**private def** buildRegex**(**forbiddenSet**: Set[Vertex],** source**: Vertex,**

target**: Vertex): Regex = {**

**val** newForbiddenSet **=** forbiddenSet **+** source

**val** loopPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** source**) \***

source**.**loopBound

**val** directPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** target**)** loopPart **+** directPart

**}**

**private def** buildRegexPart**(**forbiddenSet**: Set[Vertex],** source**: Vertex,** target**: Vertex): Regex = {**

**val** key **=** CacheKey**(**source**,** target**,** forbiddenSet**)** cache get key **match {**

**case** Some**(**cachedResult**) =>**

cachedResult

**case** None **=>**

**val** branches **= for {**

edge **<-** cfg**.**getIncidentEdges**(**source**)** neighbor **=** edge**.**target

**if (**neighbor **==** target**) || !(**forbiddenSet contains neighbor**)**

**} yield {**

**val** tail **= if (**neighbor **==** target**)** EmptyString **else**

buildRegex**(**forbiddenSet**,** neighbor**,** target**)** Branch**(**Literal**(**source**) +** tail**,** edge**.**probability**)**

**}**

**val** result **= if (**branches**.**isEmpty**)** EmptySet **else**

branches**.**alternate

cache**(**key**) =** result

result

**}**

**}**

}

*Файл PackratRegexBuilderWithTransitiveClosure.scala*

**package** ru.miet.xtestimator.regex

**import** ru.miet.xtestimator.cfg.Cfg

**import** ru.miet.xtestimator.cfg.Cfg.Vertex

**import** ru.miet.xtestimator.regex.BatchAlternation.Branch

**import** ru.miet.xtestimator.regex.RegexImplicits.\_

**import** scala.collection.mutable

**final class** PackratRegexBuilderWithTransitiveClosure**(**cfg**: Cfg) extends**

RegexBuilder **{**

**import**

ru.miet.xtestimator.regex.PackratRegexBuilderWithTransitiveClosure.\_

**private[this] val** transitiveClosure **=** TransitiveClosure**(**cfg**) private[this] val** cache **=** mutable**.**WeakHashMap**[CacheKey**, **Regex]()**

**def** build**: Regex =** buildRegex**(**Set**.**empty**,** cfg**.**entry**,** cfg**.**exit**) +**

Literal**(**cfg**.**exit**)**

**private def** buildRegex**(**forbiddenSet**: Set[Vertex],** source**: Vertex,**

target**: Vertex): Regex =**

**if (**transitiveClosure**.**contains**(**source**,** target**)) { val** newForbiddenSet **=** forbiddenSet **+** source

**val** loopPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** source**) \***

source**.**loopBound

**val** directPart **=** buildRegexPart**(**newForbiddenSet**,** source**,** target**)** loopPart **+** directPart

**}**

**else {**

EmptySet

**}**

**private def** buildRegexPart**(**forbiddenSet**: Set[Vertex],** source**: Vertex,** target**: Vertex): Regex = {**

**val** key **=** CacheKey**(**source**,** target**,** forbiddenSet**)** cache get key **match {**

**case** Some**(**cachedResult**) =>**

cachedResult

**case** None **=>**

**val** branches **= for {**

edge **<-** cfg**.**getIncidentEdges**(**source**)** neighbor **=** edge**.**target

**if (**neighbor **==** target**) || !(**forbiddenSet contains neighbor**)**

**} yield {**

**val** tail **= if (**neighbor **==** target**)** EmptyString **else**

buildRegex**(**forbiddenSet**,** neighbor**,** target**)** Branch**(**Literal**(**source**) +** tail**,** edge**.**probability**)**

**}**

**val** result **= if (**branches**.**isEmpty**)** EmptySet **else**

branches**.**alternate

cache**(**key**) =** result

result

**}**

**}**

}

*Файл TransitiveClosure.scala*

**package** ru.miet.xtestimator.regex

**import** ru.miet.xtestimator.cfg.Cfg

**import** ru.miet.xtestimator.cfg.Cfg.Vertex

**import** scala.collection.mutable

**final class** TransitiveClosure **private(**t**: mutable.HashMap[Vertex**, **mutable.HashMap[Vertex**, **Boolean]]) {**

**def** contains**(**source**: Vertex,** target**: Vertex): Boolean =** t**(**source**) (**target**)**

}

**object** TransitiveClosure **{**

**def** apply**(**cfg**: Cfg): TransitiveClosure = {**

**val** t **=** mutable**.**HashMap**[Vertex**, **mutable.HashMap[Vertex**, **Boolean]]**

**()**

**for (**i **<-** cfg**.**vertices**) {**

t**(**i**) =** mutable**.**HashMap**[Vertex**, **Boolean]() for (**j **<-** cfg**.**vertices**) {**

t**(**i**)(**j**) =** cfg**.**isAdjacent**(**i**,** j**)**

**}**

**}**

**for (**k **<-** cfg**.**vertices**) { for (**i **<-** cfg**.**vertices**) {**

**for (**j **<-** cfg**.**vertices**) {**

t**(**i**)(**j**) =** t**(**i**)(**j**) ||** t**(**i**)(**k**) &&** t**(**k**)(**j**)**

**}**

**}**

**}**

**new** TransitiveClosure**(**t**)**

**}**

}

*Файл StructuredCfgGenerator.scala*

**package** ru.miet.xtestimator.tests.performance.cfggeneration

**import** ru.miet.xtestimator.cfg.Cfg

**import** scala.util.Random

**import** StructuredCfgGenerator.\_

**import** java.io.\_

**import** scala.collection.mutable

**import** ru.miet.utils.Loan.\_

**class** StructuredCfgGenerator **extends** AutoCloseable **{ private val** cacheFile **= new** File**(**"program-block.cache"**) private val** cache **=** initCache**()**

**private def** initCache**() = { if (**cacheFile**.**exists**)**

loan**(new** ObjectInputStream**(new** FileInputStream**(**cacheFile**)))** to **{**

**\_.**readObject**().**asInstanceOf**[mutable.HashMap[ProgramBlockConfiguration**, **Seq[ProgramBlock]]]**

**}**

**else**

mutable**.**HashMap**[ProgramBlockConfiguration**, **Seq[ProgramBlock]]()**

**}**

**override def** close**(): Unit =** loan**(new** ObjectOutputStream**(new** FileOutputStream**(**cacheFile**)))** to **{**

**\_.**writeObject**(**cache**)**

**}**

**def** getSequence**(**config**: ProgramBlockConfiguration,** count**: Int): Seq[Cfg] = {**

**val** relevantEntries **=** cache get config **match { case** Some**(**s**) =>** s

**case** None **=>** Seq**()**

**}**

**val** programBlocks **= if (**relevantEntries**.**length **>=** count**) {**

relevantEntries**.**take**(**count**)**

**}**

**else {**

**val** missingEntriesCount **=** count **-** relevantEntries**.**length

**val** missingEntries **= for (**i **<- 1** to missingEntriesCount**) yield { val** programBlockGenerator **= new**

ProgramBlockGenerator**(**config**.**sequenceLength**,** config**.**branchCount**)** programBlockGenerator**(**config**.**controlStructureCount**)**

**}**

**val** result **=** relevantEntries **++** missingEntries cache**(**config**) =** result

result

**}**

programBlocks**.**map**(\_.**toCfg**)**

**}**

**private class** ProgramBlockGenerator**(**sequenceLength**: Int,**

branchCount**: Int) {**

**def** apply**(**controlStructureCount**: Int): ProgramBlock = { val** entry **=** BasicBlock**.**generate

**val** body **=** generateProgramBlock**(**controlStructureCount**)**

**val** exit **=** BasicBlock**.**generate Sequence**(**List**(**entry**,** body**,** exit**))**

**}**

**private def** generateProgramBlock**(**controlStructureCount**: Int): ProgramBlock =** controlStructureCount **match {**

**case 0 =>** BasicBlock**.**generate

**case \_ =>** selectAny**(**

**() =>** generateSequence**(**controlStructureCount **- 1), () =>** generateBranching**(**controlStructureCount **- 1), () =>** generateLoop**(**controlStructureCount **- 1)**

**)**

**}**

**private def** generateSequence**(**controlStructureCount**: Int) =**

Sequence**(**generateProgramBlockList**(**controlStructureCount**,** sequenceLength**))**

**private def** generateBranching**(**controlStructureCount**: Int) =**

Branching**(**BasicBlock**.**generate**,** generateProgramBlockList**(**controlStructureCount**,** branchCount**),** BasicBlock**.**generate**)**

**private def** generateProgramBlockList**(**controlStructureCount**: Int,**

maxLength**: Int) =**

**for (**count **<-** place**(**controlStructureCount**,** maxLength**)) yield**

generateProgramBlock**(**count**)**

**private def** generateLoop**(**controlStructureCount**: Int) =**

Loop**(**BasicBlock**.**generate**,** generateProgramBlock**(**controlStructureCount**))**

**}**

}

**object** StructuredCfgGenerator **{**

**private[cfggeneration] def** selectAny**(**options**: (() => ProgramBlock)\*)**

**=** options**(**Random**.**nextInt**(**options**.**length**))()**

**private[cfggeneration] def** place**(**elementsCount**: Int,** placeCount**: Int) = {**

require**(**elementsCount **>= 0)**

**val** nonEmptyPlaces **= if (**elementsCount **> 0) { if (**placeCount **== 2) {**

**val** integralPart **=** elementsCount **/** placeCount List**(**integralPart**,** elementsCount **-** integralPart**)**

**}**

**else {**

**val** uniformPlaceCount **=** Math**.**min**(**elementsCount**,** placeCount **-**

**1)**

**val** uniformPlaces **=** List**.**fill**(**uniformPlaceCount**)(**elementsCount

**/** uniformPlaceCount**)**

**val** remainderPlace **= {**

**val** remainder **=** elementsCount **%** uniformPlaceCount

**if (**remainder **!= 0)** List**(**remainder**) else** Nil

**}**

remainderPlace **:::** uniformPlaces

**}**

**}**

**else {**

Nil

**}**

List**.**fill**(**placeCount **-** nonEmptyPlaces**.**length**)(0) :::**

nonEmptyPlaces

**}**

}

*Файл ProgramBlockConfiguration.scala*

**package** ru.miet.xtestimator.tests.performance.cfggeneration

**final case class** ProgramBlockConfiguration**(**sequenceLength**: Int,**

branchCount**: Int,** controlStructureCount**: Int) {**

require**(**sequenceLength **>= 0)**

require**(**branchCount **>= 0)**

require**(**controlStructureCount **>= 0)**

}

*Файл programBlock.scala*

**package** ru.miet.xtestimator.tests.performance.cfggeneration

**import** ru.miet.xtestimator.cfg.Cfg.**{**Edge**,** Vertex**} import** ru.miet.xtestimator.StochasticVariable **import** ru.miet.xtestimator.cfg.Cfg

**import** ru.miet.xtestimator.tests.performance.cfggeneration.ProgramBlock.Decom position

**sealed abstract class** ProgramBlock **{ def** toCfg**: Cfg =** decompose**.**toCfg

**private[cfggeneration] def** decompose**: Decomposition**

}

**object** ProgramBlock **{**

**private[cfggeneration] final case class** Decomposition**(**vertices**: Set[Vertex],** edges**: Set[Edge],** entry**: Vertex,** exit**: Vertex) {**

**def** toCfg**: Cfg =** Cfg**(**vertices**,** edges**,** entry**,** exit**)**

**}**

}

**final case class** BasicBlock**(**id**: String) extends** ProgramBlock **{ private[cfggeneration] def** decompose**: Decomposition = {**

**val** v **=** Vertex**(**id**,** StochasticVariable**.**Zero**)** Decomposition**(**Set**(**v**),** Set**.**empty**,** v**,** v**)**

**}**

}

**object** BasicBlock **{**

**def** generate**: BasicBlock =** BasicBlock**(**IdGenerator**.**nextId**())**

**private object** IdGenerator **{ private var** id **= -1**

**def** nextId**(): String = {**

id **+= 1**

id**.**toString

**}**

**}**

}

**final case class** Sequence**(**steps**: List[ProgramBlock]) extends**

ProgramBlock **{**

require**(!**steps**.**isEmpty**)**

**private[cfggeneration] def** decompose**: Decomposition = { val** stepDecompositions **=** steps map **(\_.**decompose**)**

**val** stepEdges **=** stepDecompositions**.**foldLeft**(**Set**.**empty**[Edge])((**acc**,**

d**) =>** acc **++** d**.**edges**)**

**val** intermediateEdges **=** stepDecompositions**.**sliding**(2).**foldLeft**(**Set**.**empty**[Edge])((**acc**,** p**) =>** acc **+** Edge**(**p**(0).**exit**,** p**(1).**entry**))**

Decomposition**(**

vertices **=** stepDecompositions**.**foldLeft**(**Set**.**empty**[Vertex])((**acc**,**

d**) =>** acc **++** d**.**vertices**),**

edges **=** stepEdges **++** intermediateEdges**,** entry **=** stepDecompositions**.**head**.**entry**,** exit **=** stepDecompositions**.**last**.**exit

**)**

**}**

}

**final case class** Branching**(**header**: BasicBlock,** branches**: List[ProgramBlock],** footer**: BasicBlock) extends** ProgramBlock **{**

require**(**branches**.**length **> 1)**

**private[cfggeneration] def** decompose**: Decomposition = { val** headerDecomposition **=** header**.**decompose

**val** branchDecompositions **=** branches map **(\_.**decompose**) val** footerDecomposition **=** footer**.**decompose

**val** foldBranchDecompositions **=**

branchDecompositions**.**foldLeft**(**Set**.**empty**[Edge]) \_**

**val** headerEdges **=** foldBranchDecompositions**((**acc**,** d**) =>** acc **+**

Edge**(**headerDecomposition**.**exit**,** d**.**entry**))**

**val** branchEdges **=** foldBranchDecompositions**((**acc**,** d**) =>** acc **++**

d**.**edges**)**

**val** footerEdges **=** foldBranchDecompositions**((**acc**,** d**) =>** acc **+**

Edge**(**d**.**exit**,** footerDecomposition**.**entry**))** Decomposition**(**

vertices **=** headerDecomposition**.**vertices **++** branchDecompositions**.**flatMap**(\_.**vertices**) ++** footerDecomposition**.**vertices**,**

edges **=** headerEdges **++** branchEdges **++** footerEdges**,** entry **=** headerDecomposition**.**entry**,**

exit **=** footerDecomposition**.**exit

**)**

**}**

}

**final case class** Loop**(**header**: BasicBlock,** body**: ProgramBlock) extends**

ProgramBlock **{**

**private[cfggeneration] def** decompose**: Decomposition = { val** headerDecomposition **=** header**.**decompose

**val** bodyDecomposition **=** body**.**decompose

Decomposition**(**

vertices **=** headerDecomposition**.**vertices **++**

bodyDecomposition**.**vertices**,**

edges **=** bodyDecomposition**.**edges **+** Edge**(**headerDecomposition**.**exit**,** bodyDecomposition**.**entry**) +** Edge**(**bodyDecomposition**.**exit**,** headerDecomposition**.**entry**),**

entry **=** headerDecomposition**.**entry**,** exit **=** headerDecomposition**.**exit

**)**

**}**

}