МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет компьютерных технологий и прикладной математики
Кафедра информационных технологий

КУРСОВАЯ РАБОТА

ОПТИМИЗАЦИЯ МЕТОДОВ ОПРЕДЕЛЕНИЯ СЛОЖНОСТИ АЛГОРИТМА. ВНЕДРЕНИЕ СРЕДСТВ ПАРАЛЛЕЛЬНОГО ВЫЧИСЛЕНИЯ

Работу выполнил _____ Ф. Р. Петренко
(подпись)
Направление подготовки 02.03.03 — «Математическое обеспечение и
(код, наименование)
администрирование информационных систем»

Направленность (профиль) Технологии программирования

Научный руководитель
д-р физ.-мат. наук, проф. _____ А. И. Миков
(подпись)
Нормоконтролер
ст. преп. _____ А. В. Харченко
(подпись)

Краснодар
2022

РЕФЕРАТ

Курсовая работа 54 стр., 15 рис., 8 источников, 7 приложений.

ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМА, ПОСТРОЕНИЕ ГРАФИКА СЛОЖНОСТИ, ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ, ЛИНЕЙНАЯ РЕГРЕССИЯ, ПОЛИНОМИАЛЬНАЯ РЕГРЕССИЯ

Объектом исследования в данной работе является процесс оптимизации и модификации разработанной программы оценки сложности алгоритмов с помощью программных средств.

Цель работы: разработка новых методов и механизмов определения сложности алгоритмов, оптимизирующих и расширяющих возможности процесса поиска класса сложности.

Методологическая основа исследования включает в себя элементы математической статистики, метод гипотез, метод построения параллельных программ, графический метод, анализ полученных зависимостей, классификацию.

В результате работы в приложение, организующее транскомпиляцию кода на языке описания алгоритмов Algolite в код на языке C++ с дальнейшим определением класса сложности рассматриваемого алгоритма и построением графика сложности, были введены новые функциональные элементы языка описания алгоритмов, методы определения асимптотического и точного вида функции сложности и механизмы параллельного вычисления.

Научная новизна и уникальность работы заключается в том, что предлагается оригинальный способ программного анализа сложности различных алгоритмов на основе числа элементарных операций, которые они содержат, реализованный с помощью средств параллельного вычисления и математических методов, оптимизирующих процесс определения класса сложности.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Несмотря на сегодняшние тенденции к упрощению процесса разработки программного обеспечения благодаря введению различных вспомогательных средств (библиотеки, фреймворки, средства визуального программирования и т.д.), создание огромного числа программных продуктов всё ещё неразрывно связано с самостоятельной реализацией и внедрением алгоритмов, решающих набор определённых проблем. Отдельные проблемы пришли из классической математики, к примеру, численные методы решения дифференциальных и иных уравнений.

Актуальность данной работы объясняется тем, что каждый разработчик, находясь на этапе внедрения в систему некоторого конкретного алгоритма и задумываясь над резонностью данного действия, тратит собственные силы на ручное определение сложности рассматриваемого алгоритма. Подобные затраты могут быть легко исключены использованием программы, автоматизирующей данный процесс.

Основной целью данной работы является расширение и оптимизация функционала программы для автоматизированного определение класса сложности некоторого алгоритма с дальнейшим построением графика сложности, которая была разработана в рамках предыдущей курсовой работы [1].

Для добавления новых возможностей и оптимизации работы программного комплекса предполагается решить следующие задачи:

– расширить функционал языка описания алгоритмов Algolite;

– усовершенствование имеющихся методов определения класса сложности рассматриваемого алгоритма;

– оптимизировать процесс вставки счётчиков в конечный программный код для подсчёта числа элементарных операций, выполняемых при запусках с различными значениями некоторого параметра, путём внедрения средств параллельного вычисления.

Объектом исследования в данной работе является процесс оптимизации и модификации разработанной программы оценки сложности алгоритмов с помощью программных средств.

Предметом исследования является качество работы разработанной программы, характеризующееся точностью и быстротой определения сложности рассматриваемых алгоритмов.

Методологическая основа исследования включает в себя элементы математической статистики, метод гипотез, метод построения параллельных программ, графический метод, анализ полученных зависимостей, классификацию.

Научная новизна и уникальность работы заключается в том, что предлагается оригинальный способ программного анализа сложности различных алгоритмов на основе числа элементарных операций, которые они содержат, реализованный с помощью средств параллельного вычисления и математических методов, оптимизирующих процесс определения класса сложности.

Теоретическая и практическая значимость работы характеризуется пользой, которую может привнести программа, позволяющая разработчикам автоматизированным способом определять сложность рассматриваемых алгоритмов при проектировании некоторой системы, тем самым исключая излишние затраты сил на ручное определение сложности.

1 Постановка задачи

Традиционно в программировании понятие сложности алгоритма связано с использованием ресурсов компьютера. Учет памяти обычно ведется по объему данных. Время рассчитывается в относительных единицах так, чтобы эта оценка, по возможности, была одинаковой для различных машин [2].

Будем рассматривать временную сложность алгоритмов, которая подсчитывается в исполняемых командах: количество арифметических операций, сравнений, пересылок.

Временная сложность алгоритма обычно выражается с использованием нотации «О» большое (Big O), которая учитывает только слагаемое самого высокого порядка (т.е. константы, входящие в функцию сложности, не учитываются) [3]. Если сложность выражена таким способом, говорят об асимптотическом описании временной сложности, то есть при стремлении размера входа к бесконечности. Для лучшего понимания сначала опишем данный процесс с формальной стороны, приведя ряд определений.

Проколотой окрестностью точки называется окрестность точки, из которой исключена эта точка.

Пусть $f(x)$ и $g(x)$ — две функции, определенные в некоторой проколотой окрестности точки x_0, причем в этой окрестности g не обращается в ноль. Говорят, что f является «О» большим от g при $x \to x_0$, если существует такая константа $C > 0$, что для всех x из некоторой окрестности точки x_0 имеет место неравенство:

$$|f(x)| \leq C|g(x)| \tag{1}$$

Иначе говоря, в первом случае отношение модулей $f(x)$ и $g(x)$ ограничено сверху некоторой константой C в окрестностях x_0. Нотация «О» большое описывает количество операций при наихудшем сценарии.

В рамках рассмотрения сложности алгоритмов Big O по своей сути позволяет разделять различные алгоритмы по классам сложности [4]. К таким классам относятся: $O(1)$ (константное время), $O(n)$ (линейное время), $O(n^{\alpha})$ (полиномиальное время, где $\alpha > 1$), $O(\log n)$ (логарифмическое время), $O(n * \log n)$ (линейно-логарифмическое время), $O(2^n)$ (экспоненциальное время) и т.д.

Задача курсовой работы состоит в расширении и оптимизации функционала программы для автоматизированного оценивания сложности произвольного алгоритма с отображением графика его сложности.

После завершения разработки и внедрения нововведений в приложение будет проведено некоторое число тестирований, направленных на выявление качества работы программы, проявляющегося в точности и быстродействии определения класса сложности данного алгоритма. Тестирования будут проводиться при разных значениях некоторого параметра, влияющего на число выполняемых операций, на основе набора общеизвестных алгоритмов для лучшего понимания того, насколько близок результат работы программы к настоящему классу сложности того или иного алгоритма. Помимо этого, одной из задач является повышение информативности конечного результата работы программы, что может быть обеспечено дополнительным определением точного вида функции сложности рассматриваемого алгоритма.

Преобладание самостоятельности в работоспособности программы (без вмешательства человека) рассматривалось как один из наиболее важных аспектов во время её разработки. В рамках данной курсовой работы основной задачей является расширение и оптимизация имеющегося функционала, что позволит значительно сократить время ожидания до получения конечного результата при анализе сложности некоторого алгоритма. Именно такой подход может нести наибольшую пользу для разработчиков, уменьшая затраты сил на ручное определение сложности интересующих их алгоритмов.

2 Математические методы решения задачи

В целях решения задачи расширения и оптимизации функционала языка описания алгоритмов Algolite, представляющего собой небольшой набор команд из языка C++ с соответствующей семантикой и функционалом, в раннее представленную форму Бэкуса-Наура (БНФ, BNF) с периодическим использованием форм записи из регулярных выражений были добавлены новые элементы. Актуальный набор элементов языка описывается с помощью синтаксиса расширенной формы Бэкуса-Наура (РБНФ, EBNF). Ниже представлены измененные и добавленные участки описания:

<отключение анализа в блоке> ::= '# ANALYSIS_DISABLE #' (<блок> - <отключение анализа в блоке>)

<case> ::= 'case' <выражение> ':' [{<команда>}]

<default> ::= 'default' ':' [{<команда>}]

<switch> ::= 'switch' '(' <выражение> ')' '{' [{<case>}] [<default>] '}'

<печать> ::= 'cout' '<<' (<выражение> | <строковый литерал>) [{'<<' (<выражение> | <строковый литерал>)}]

<блок> ::= '{' [{<команда>}] '}'

<простая команда> ::= <вызов функции> | <объявление переменной> | <присваивание значения переменной> | <печать> | <инкремент> | <декремент> | <уменьшение значения> | <увеличение значения> | <break> | <continue> | <return>

<команда> ::= (<простая команда> ';') | (<do while> ';') | <отключение анализа в блоке> | <if> | <for> | <switch> | <while> | <блок>

<программа> ::= <объявление параметра сложности> [{<объявление функции>}]

В процессе анализа сложности алгоритма путём подсчёта числа элементарных операций, выполняемых на некотором значении рассматриваемого параметра, перехода к новым переменным и выбора класса сложности на основе средней ошибки могут возникать неточности, выдающие

ошибочный результат за верный. Решить данную проблему можно с помощью улучшения алгоритма определения класса сложности на основе средней ошибки, который будет анализировать изменения ошибки и определяемого параметра степени на некотором значении интервалов, получаемых из разбиения отрезка между минимальным и максимальным значением параметра. В случае ошибочного определения класса на этапе линейной регрессии результат останется неизвестным, что привнесёт в общий алгоритм элемент эвристики [5].

Для расширения набора классов сложности, которые программа может определить асимптотически, можно ввести анализ на принадлежность функции сложности к экспоненциальному классу, а для повышения информативности выдаваемого результата можно одновременно представлять асимптотическое и точное значение функции сложности. Точное значение можно получить в случае принадлежности функции полиномиальному классу путём проведения анализа методом полиномиальной регрессии [6].

Оптимизировать этап подсчета числа выполняемых операций можно с помощью выполнения заданного алгоритма и получения числа операций в режиме параллельного вычисления [7].

Рассмотрим следующий набор методов и решений, необходимых для выполнения поставленных задач.

2.1 Расширение функционала языка описания алгоритмов

Ввиду необходимости добавления нового функционала в язык описания алгоритмов Algolite было принято решение о введение элементов, которые смогли бы позволить пользователю самостоятельно оптимизировать рассматриваемый алгоритм.

К уже имеющемуся условному оператору switch была добавлена конструкция default, позволяющая определять случай выполнения по умолчанию. В рамках РБНФ связь этих элементов описывается так: <default>

::= 'default' ':' [{<команда>}] , <switch> ::= 'switch' '(' <выражение> ')' '{' [{<case>}] [<default>] '}'. Данный механизм позволяет пользователю определять более комплексные условные операторы, уменьшая общее количество текста описания алгоритма.

Ещё одним решением было принято введение уникального механизма, позволяющего пользователю самостоятельно определять участки кода, которые не будут подвержены анализу, то есть вставке счётчиков на этапе транскомпиляции и подсчета числа операций на этапе анализа сложности. Это было сделано для того, чтобы в общем оптимизировать процесс работы программного комплекса. Связь данной конструкции и имеющегося синтаксиса языка описывается так: <отключение анализа в блоке> ::= '# ANALYSIS_DISABLE #' (<блок> - <отключение анализа в блоке>), <блок> ::= '{' [{<команда>}] '}', <команда> ::= (<простая команда> ';') | (<do while> ';') | <отключение анализа в блоке> | <if> | <for> | <switch> | <while> | <блок>.

Нередко для того, чтобы проверить работоспособность рассматриваемого алгоритма, необходимо организовывать предварительную инициализацию данных. Данный процесс может включать большое количество элементарных операций и при этом не отображать конечную суть рассматриваемого алгоритма. Благодаря введённому механизму можно избежать подсчёта лишнего числа операций и возможного получения ошибочного класса сложности.

2.2 Расширение и оптимизация процесса определения сложности

Имеющийся алгоритм, подразумевающий применение метода линейной регрессии, можно применять постепенно на всём интервале вычисленных точек, представляющих собой пары значений параметра и числа элементарных операций, поделённом на некоторое число делений. Данное число было принято за 10.

На каждом из интервалов, отсчитываемых от начала, значение шага будет вычисляться как общее число точек, делённое на число интервалов целочисленным образом. При этом к первому интервалу будет добавлено значение равное общему числу точек минус шаг, умноженный на число интервалов. Данное значение необходимо для того, чтобы захватить весь интервал точек.

Примерную схему такого анализа вычисленного набора точек отображает рисунок 1:

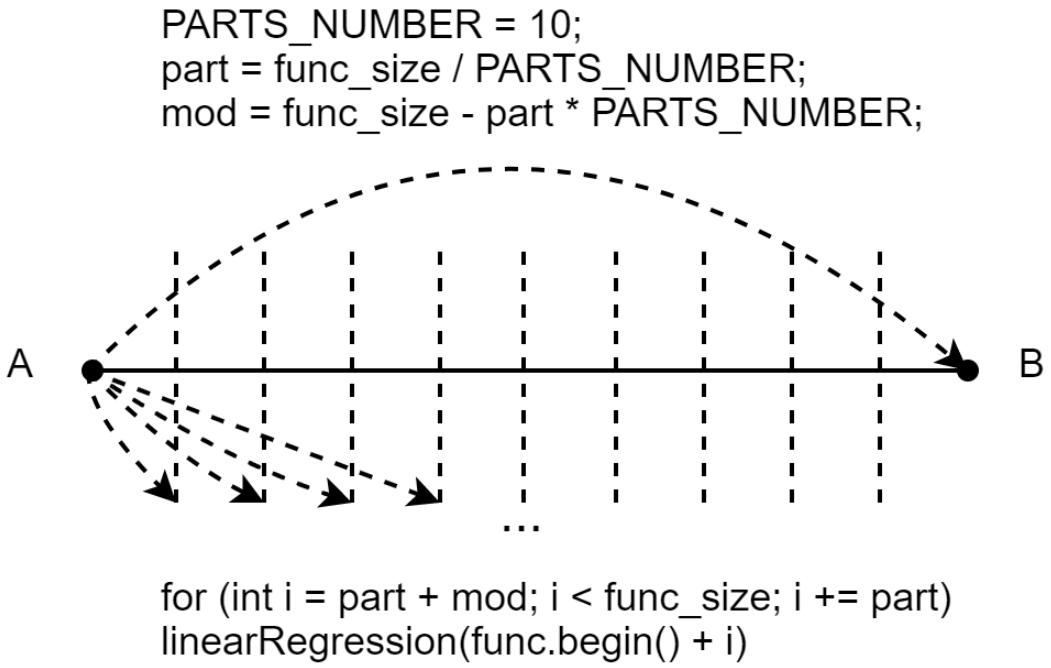

Рисунок 1 – Схематичное описание алгоритма анализа полученного набора точек, где A – начало набора, а B – конец набора

В результате будет получено число увеличений значения каждой из отслеживаемых ошибок, соответствующих предположительным классам сложности, число изменений степени для каждого из таких классов и значения последних ошибки, которые отображают вычисленные в результате применения метода линейной регрессии значения средних ошибок на всём интервале рассматриваемых точек.

В зависимости от того, какая из последних ошибок имеет наименьшее значение, выбирается предположительный класс сложности. Для окончательного определения результата были приняты следующие ограничения: если ошибка предположительного класса сложности увеличивалась менее половины раз на протяжении всего анализа или имеет конечное значение меньшее 1e-04 (0.0001) и при этом число изменений степени старшего слагаемого в конечной функции сложности происходило менее 5 раз, то предположенный класс сложности является верным, после чего может быть выдан асимптотический результат. В противном случае класс сложности будет определён как «Unknown», так как для определения класса требуются дополнительные исследования, которые программа не способна отобразить. Данное решение привносит эвристический элемент в общий алгоритм, но в общем случае такой ответ получается довольно редко (например, при выборе пользователем некорректных параметров для проведения анализа сложности) [5].

К имеющимся классам сложности, которые могут быть определены программой, был добавлен класс экспоненциальной сложности. Так как в программе применяется логарифм с основанием 10, а $10^x = e^{(x*ln10)}$, то предположим, что $y = c * 10^{kx}$ (y – число операций при данном x). Тогда $log(y) = log(c) + k * x$. Пусть $u = log(y)$, а $v = x$. Значит в новых переменных функция сложности обретает вид $u = log(c) + k * v$ (уравнение прямой). Тогда можно провести линейную регрессию и узнать значения $log(c)$ и k, где k - степень старшего члена в исходной функции (если функция сложности представляет собой постоянную, то $k = 0$) [6]. Для определения степени x в функции сложности используем следующие формулы, которые аналогичным образом применяются к полиномиальному и логарифмическому случаю:

$$b = \frac{n\sum x'y' - (\sum x')(\sum y')}{n\sum (x')^2 - (\sum (x'))^2} \quad (1)$$

$$a = \frac{\sum y\prime - b(\sum x\prime)}{n} \quad (2)$$

В этих формулах $a = log(c)$, $b = k$, $x' = v$, $y' = u$. Отсюда получим, что $u` = a + b * v = log(c) + k * v$. Теперь посчитаем среднюю ошибку, отображающую величину погрешности при попытке приближения прямой, полученной из предположения принадлежности функции к рассматриваемому классу сложности к найденным парам значений параметра и числа выполнимых элементарных операций:

$$e = \frac{\Sigma(u - u`)^2}{n} \quad (3)$$

Суть перехода к новым переменным представлена на рисунке 2.

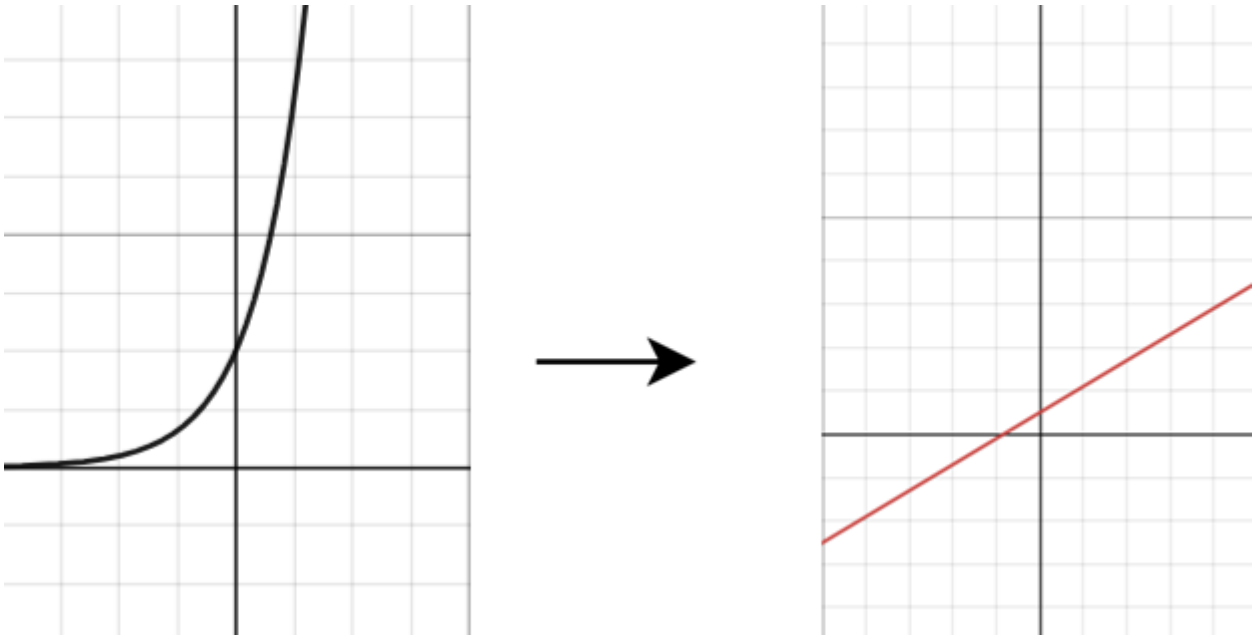

Рисунок 2 – Отображение процесса перехода к новым переменным в случае принадлежности функции сложности к экспоненциальному типу

Для того чтобы повысить информативность выдаваемого результата, можно дополнительно определять точный вид функции сложности

рассматриваемого алгоритма. В случае принадлежности такой функции к полиномиальному классу можно применить метод полиномиальной регрессии, представляющую собой в статистике форму регрессионного анализа, в которой связь между некоторыми переменными определяется как полином k-й степени. В данном случае мы не будем пренебрегать слагаемыми, которые не относятся к старшей степени.

Если после применения вышеперечисленных методов анализа программа определила полиномиальный класс как верный для функции сложности рассматриваемого алгоритма, то будут выполнены следующие действия. Пусть общий вид функции $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^k$. Воспользуемся методом полиномиальной регрессии, представимый как разновидность метода множественной регрессии, в котором роль нескольких независимых переменных выполняют одна независимая переменная и ей степени [6]. На основе n имеющихся точек составим вид системные линейных уравнений:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_2^k \\ 1 & x_3 & x_3^2 & \cdots & x_3^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^k \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_k \end{bmatrix} \quad (4)$$

При использовании матричных обозначений получим вид $\vec{y} = \mathrm{X}\vec{\beta}$, где $\vec{y} -$ вектор из значений числа операций на конкретном параметре, $\vec{\beta} -$ вектор искомых коэффициентов, а $\mathrm{X} -$ матрица Вандермонда, составленная из значений параметров для анализа. Затем можно получить $(\mathrm{X}^\mathsf{T}\mathrm{X})\vec{\beta} = \mathrm{X}^\mathsf{T}\vec{y}$, где $\mathrm{X}^\mathsf{T} -$ транспонированная матрица X. Представим это в виде системы $A\overline{x} = B$, где $A = (\mathrm{X}^\mathsf{T}\mathrm{X})$, $\overline{x} = \vec{\beta}$, а $B = \mathrm{X}^\mathsf{T}\vec{y}$. С помощью метода Гаусса можно получить значения вектора $\vec{\beta}$. Нам уже известно значений старшей степени в искомом уравнении полинома, а значит на основе полученных коэффициентов

$\beta_0, \beta_1, \dots, \beta_n$ можно составить конечный вид функции сложности рассматриваемого алгоритма.

2.3 Внедрение средств параллельных вычислений

Ввиду того, что подсчет числа элементарных операций, которые выполняются при некотором значении параметра, обеспечивается буквальным запуском программы, образованной транскомпиляцией текста рассматриваемого алгоритма, возникает необходимость ускорения данного процесса. Заметный прирост скорости можно обрести путём внедрения средств параллельных вычислений.

При параллельном подсчёте числа операций каждый из запущенных потоков будет иметь собственную переменную counter, в которую будет записывать конечный результат. В итоге все полученные пары значений параметра и числа элементарных операций будут записаны в общий набор. Сравнение подходов к подсчету числа операций отображено на рисунке 3.

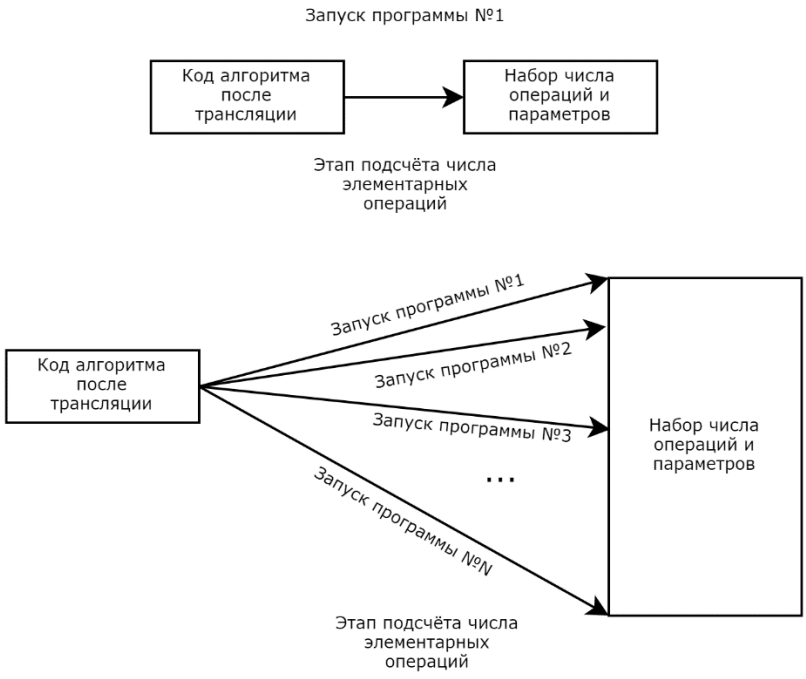

Рисунок 3 – Сравнение разных подходов к процессу подсчёта операций

При этом стоит учесть, что от общего количества потоков и номера конкретного поток зависит начальное значение параметра для потока и то, с каким шагом ему необходимо увеличить значение рассматриваемого параметра. Начальное значение вычисляется как par = par_min + par_step * thread_number, а изменение как par += par_step * thread_amount (здесь par – имя параметра, задаваемое пользователем, par_min – минимальное значение параметра, par_step – шаг, задаваемый пользователем, thread_number – номер потока, thread_amount – общее число потоков). Значение параметра для каждого из потоков будет увеличиваться при условии, что оно не превосходит par_max (где par_max – максимальное значение параметра).

3 Программная реализация

В качестве среды разработки была выбрана Microsoft Visual Studio Community 2022, версия 17.1.4. Программа создавалась на языке программирования C++. Программный комплекс состоит из двух основных .cpp и двух заголовочных файлов, трёх текстовых файлов, трёх .cpp и четырёх вспомогательных заголовочных файлов для автоматической вставки блоков кода анализа сложности, подключения стороннего компилятора, подключения сторонней реализации thread для использующегося компилятора и построения графика, а также одного текстового файла для описания анализируемого алгоритма.

Для автоматизированного отображения графика сложности в сформированную программу будет включена часть кода, отвечающая за его построение на основе функционала библиотеки pbPlots. Осям графика были даны наименования, отображающие значение параметра сложности (complexity parameter) и число выполняемых элементарных операций (operations).

В данном программном комплексе используется компилятор g++ из набора инструментов MinGW. При его установке была выбрана thread-модель Win32, что не поддерживает конструкцию std::thread в естественном виде. Для реализации работы с потоками была подключена библиотека mingw-std-threads, решающая данную проблему. Во время установки указанного инструментария можно выбрать thread-модель POSIX-threads, представляющую реализацию POSIX-совместимых потоков (нитей), но в данном случае пришлось бы использовать конструкции вида pthread из заголовочного файла pthread.h. В основном они выбираются для организации портирования многопоточных программ из UNIX-подобных систем [8].

Для дополнительного форматирования кода используется утилита Uncrustify. Она применяется путём использования команды system("uncrustify -c msvc.cfg -f finalOutput.cpp --no-backup -o finalOutput.cpp"), где заданы

следующие параметры: uncrustify -c msvc.cfg – запуск утилиты с конфигом msvc.cfg (конфиг для форматирования аналогично IDE Microsoft Visual Studio Code), -f finalOutput.cpp --no-backup -o finalOutput.cpp – форматирование файла finalOutput.cpp без создания бэкапа и конечная запись в этот же файл.

Процесс работы компилятора будет запущен после успешного формирования конечного файла в командах языка C++. Сама транскомпиляция будет запущена командой system("g++ -O3 -std=c++0x -o output_program.exe finalOutput.cpp pbPlots.cpp supportLib.cpp -lm -D _WIN32_WINNT=0x0A00"). Указанные при вызове компилятора параметры описываются следующим образом: g++ – вызов компилятора, -O3 – включение оптимизации компилятора 3-го уровня, -std=c++0x – включение стандарта C++0x, -o output_program.exe – определение будущего исполняемого файла, final_output.cpp pbPlots.cpp supportLib.cpp – выбор сформированного файла и файлов библиотеки pbPlots для компиляции, -lm – подключение математической библиотеки, -D _WIN32_WINNT=0x0A00 – выбор целевой версии Windows (выбрана Windows 10) для работы сторонней библиотеки потоков (коды версий указаны на сайте Microsoft). Запуск исполняемого файла осуществляется командой system("start output_program.exe").

3.1 Описание структур и функций

Программа разбивается на несколько файлов. В каждом из файлов выделены и указаны ниже только те части, в которых произошли значимые изменения:

1) AlgoliteParser.cpp – основной файл, содержащий функционал лексического анализа, синтаксического анализа и генерации синтаксического дерева;

   a) TokenType – тип перечисления, необходимого для определения принадлежности лексемы допустимому алфавиту

(добавлены токены для конструкции default и блока для отключения анализа);

б) accept – функция, принимающая токен перечисления (тип ожидаемой лексемы), для сравнения с токеном лексемы, рассматриваемой в данный момент (добавлены элементы для конструкции default и блока для отключения анализа);

в) command, switchCommand – функции для проверки последовательности лексем на соответствие одной из возможных конструкций анализируемого языка;

2) Nodes.h – заголовочный файл, содержащий описания классов узлов, используемых для заполнения дерева;

а) SwitchNode, DefaultNode, NonAnalysisBlockNode – классы узлов, образующих синтаксическое дерево;

б) Visitor – абстрактный класс, предназначенный для прохождения сформированного синтаксического дерева (добавлены элементы для конструкции default и блока для отключения анализа);

3) TranslatingVisitor.h – заголовочный файл, содержащий объявление методов класса TranslatingVisitor, являющегося наследников абстрактного класса Visitor (добавлены элементы для конструкции default и блока для отключения анализа);

4) TranslatingVisitor.cpp – файл, содержащий описание работы объекта класса TranslatingVisitor при обходе синтаксического дерева с последующей генерацией кода конечной программы;

а) handle(SwitchNode& n), handle(DefaultNode& n), handle(NonAnalysisBlockNode& n) (или TranslatingVisitor::handle) – методы уникальной обработки объектом-посетителем каждого из возможных узлов, формирующих синтаксическое дерево, с последующей генерацией кода конечной программы;

б) handle(Program& n) (или TranslatingVisitor::handle) – метод, символизирующий старт обработки синтаксического дерева и

включающий в текст сформированного файла в командах языка C++ блоки кода для анализа сложности и отображение графика из заранее заготовленных файлов (добавлен этап разбиения вычисления на потоки);

5) ProgramInitialization.txt – файл, содержащий заготовленный блок кода для инициализации необходимого для работы функционала (подключения библиотек, введения макросов и т.д.) и содержащий блок для проведения регрессионного анализа;

а) linearRegression – функция, принимающая вектор пар как некоторый набор значений функции сложности рассматриваемого алгоритма, требующий анализа методом линейной регрессии, и возвращающая значение вычисленной средней ошибки и степени x (добавлены аргументы, позволяющие отключать дополнительное логарифмирование переданных значений);

б) polynomialRegression – функция, принимающая вектор пар как некоторый набор значений функции сложности рассматриваемого алгоритма, требующий анализа методом линейной регрессии, и степень старшего члена функции, при этом возвращающая набор коэффициентов как результата применения полиномиальной регрессии;

в) polynomialComplexityFunction – функция для образования точного вида функции сложности рассматриваемого алгоритма;

6) ComplexityClassSearching.txt – файл, содержащий заготовленный блок кода для поиска класса сложности рассматриваемого алгоритма (добавлены элементы для определения класса экспоненциальной сложности и алгоритм разбиения на интервалы);

7) MainFunction.txt – файл, содержащий заготовленный блок кода для запуска поиска сложности алгоритма и построения графика функции (изменены параметры отрисовки графика);

3.2 Описание работы программы

Точкой входа программы является файл sourceAlgolite. В нём пользователю предлагается с помощью разработанного языка Algolite описать алгоритм, анализ сложности которого н хочет провести. Для успешного запуска процесса трансляции и последующего анализа сложности ему необходимо описать граничные значения и шаг изменения параметра сложности и ввести функцию main (без аргументов), откуда будет происходить вызов функций, составляющих рассматриваемый алгоритм. Ввиду максимальной автоматизации приложения пользователь заканчивает своё вмешательство в процесс работы программы.

Далее начинается сканирование файла sourceAlgolite, перевод текста из него в строчный формат и анализ данной строки синтаксическим и лексическим анализаторами. В зависимости от указанных конструкций, формируется синтаксическое дерево.

Затем проводится обход дерева с помощью программной реализации паттерна проектирования Посетитель (Visitor). Объект-посетитель просматривает каждый узел организованного дерева и в зависимости от типа узла формирует часть текста конечной программы на языке C++ (он вводит команды конечного языка и добавляет счётчики для вычисления числа элементарных операций, выполняемых в описанном алгоритме).

Если хотя бы на одном из этапов произошла некоторая ошибка, программа выводит текст ошибки пользователю и прекращает процесс работы.

После успешной транскомпиляции программа создаёт конечный .cpp файл путём соединения 3-х заранее описанных текстовых файлов (ComplexityClassSearching, MainFunction и ProgramInitialization), содержащих блоки для анализа сложности алгоритма и построение графика, и текста, сформированного в процессе обхода синтаксического дерева. Затем выполняется работа форматтера кода Uncrustify, после чего программа

передаётся компилятору g++ с последующим запуском исполняемого файла output_program.

После запуска исполняемого файла описанный алгоритм выполняет свою работу на различных значениях, находящихся в области граничных значений, что задал пользователь (подсчёт числа операций происходит в параллельных потоках). Данный процесс необходим для выявления зависимости числа выполняемых операций от значения параметра сложности.

На основе полученных пар значений проводится регрессионный анализ (на некотором числе интервалов), приводящий к получению параметров, в зависимости от которых и определяется класс сложности рассматриваемого алгоритма. Далее рассматриваются дополнительные условия, подтверждающие или опровергающие достоверность определённого класса. Если же функция сложности принадлежит полиномиальному класса, применяется метод полиномиальной регрессии, после чего будет получен её точный вид.

Также на основе этих пар значений проводится построение графика функции сложности алгоритма с помощью библиотеки pbPlots и его дальнейшее отображение пользователю.

На рисунках 4-7 отображён пример того, что может отобразить пользователю программа.

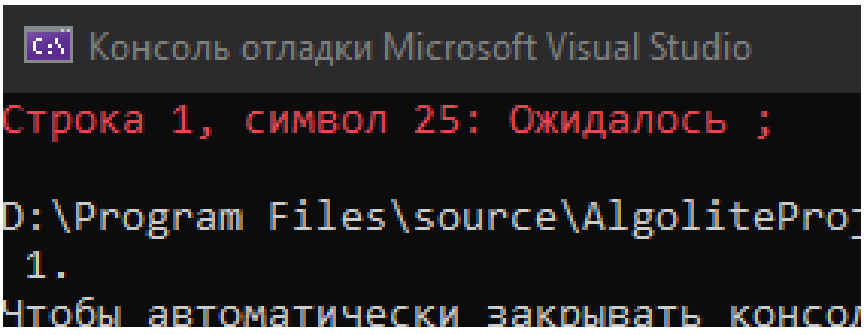

Рисунок 4 – Экстренное прекращение работы в случае ошибки

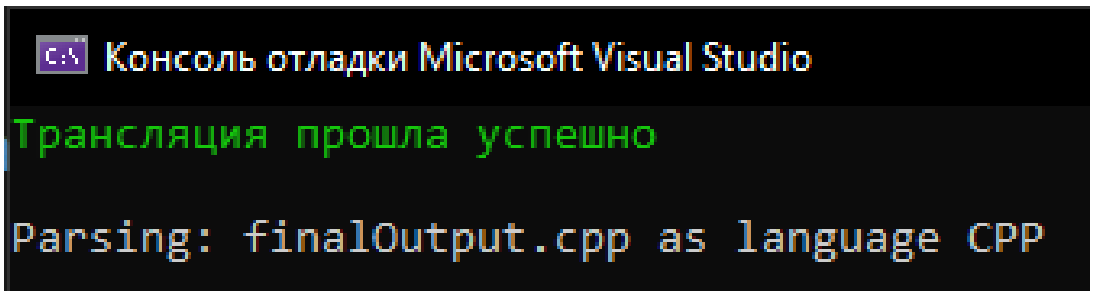

Рисунок 5 – Успешное выполнение процесса транскомпиляции

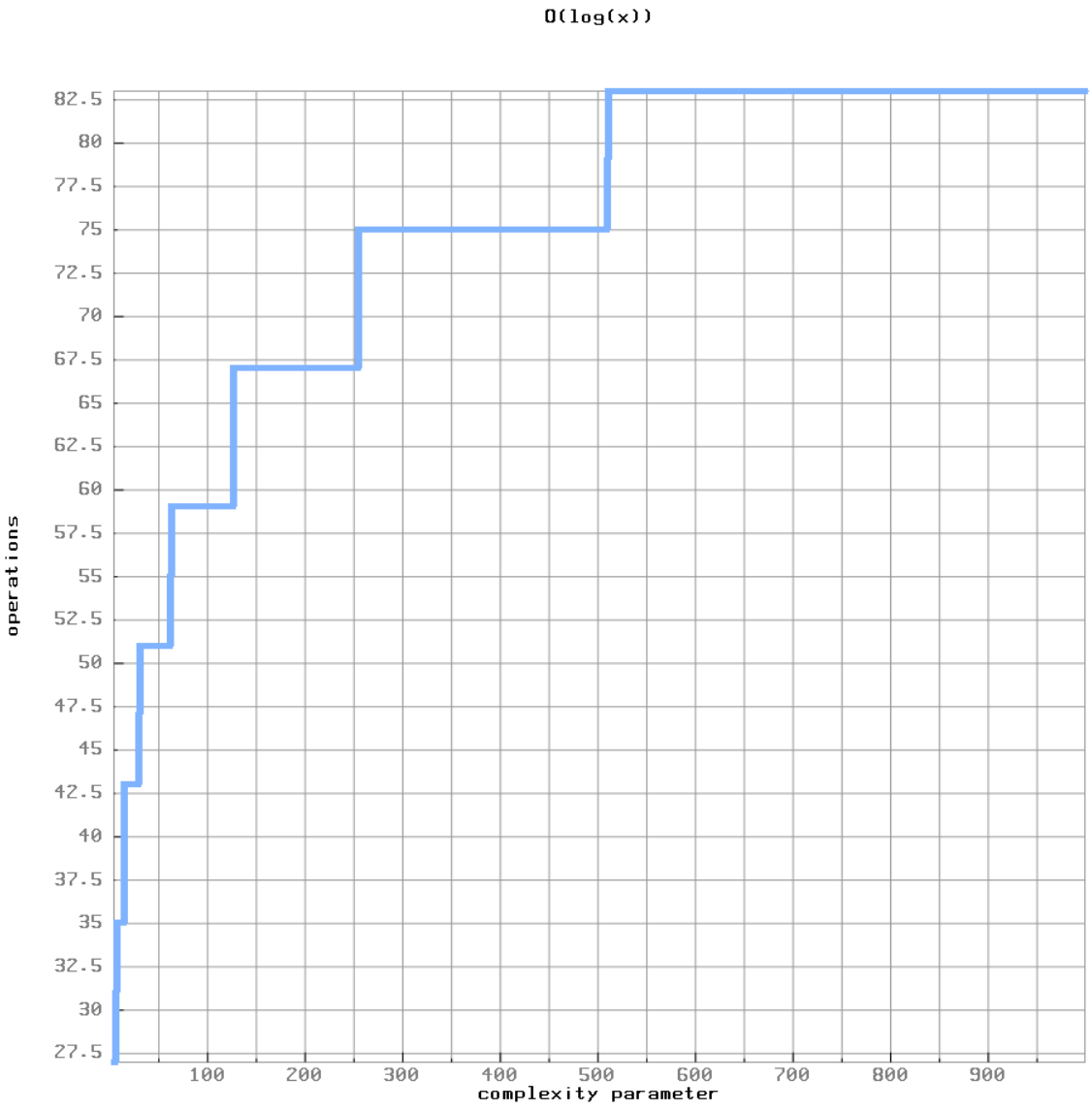

Рисунок 6 – Пример определения асимптотического класса сложности некоторого алгоритма

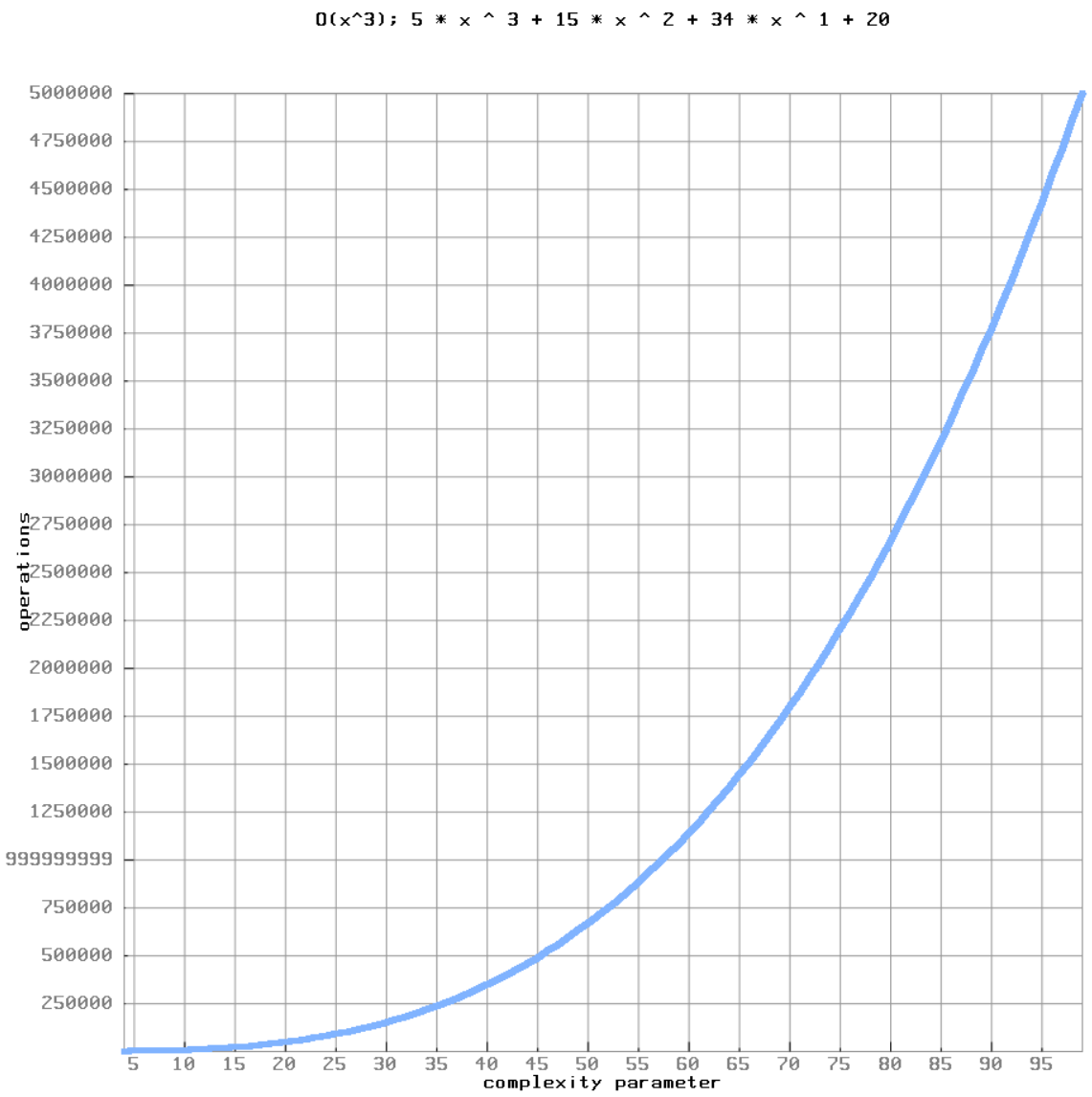

Рисунок 7 – Пример определения асимптотического и точного класса сложности некоторого алгоритма

Инструкция по работе с разработанной программой:

– открыть файл sourceAlgolite для описания алгоритма;

– объявить команду инициализации параметра сложности и ввести идентификатор параметра, граничные значения и величину шага изменения параметра при анализе;

– описать исследуемый алгоритм через набор функций;

– определить функцию main (без параметров), откуда будет запускаться весь процесс выполнения программных блоков, образующих рассматриваемый алгоритм;

– запустить процесс выполнения основной программы, описанный в файле AlgoliteParser;

Пример корректной программы на языке Algolite, описанной в соответствии с инструкцией:

```
parameter par (4, 45, 1);
long long factorial (int n) {
   long long f =1;
   for (long long i=2; i<=n;i++)
   f*=i;
   return f;
}
void main () {
   # ANALYSIS_DISABLE # { int a = 5; }
   factorial(par);
}
```

4 Результаты исследований

Так как все введённые методы расширяют или оптимизируют разработанную программу характерным именно им образом, то рассмотрим эффективность каждого из нововведений отдельно.

Расширение функционала языка описания алгоритмов позволяет создавать более гибкие конструкции, а также выделять те области алгоритма, которые пользователь не хочет учитывать при анализе. Данный подход реализует оптимизацию на уровне пользовательского контроля.

```
void binarySearch(int border, int key) {

  int x[1000];

  x[0] = 0; x[1] = 1; x[2] = 2; x[3] = 3; x[4] = 4; x[5] = 5; x[6] = 6; x[7] = 7; x[8] = 8; x[9] = 9; x[10] = 10;

  int left = 0;
  int right = border;

  int midd = 0;

  while( true ) {

  midd = (left + right) / 2;

  if  (key < x[midd])
    right = midd - 1;
  else if  (key > x[midd])
  left = midd + 1;

  else break;

  if(left > right)
  break;

  }

}

void main() {
    binarySearch(par, 1500);
}
```

Рисунок 8 – Пример текста алгоритма без использования нововведений

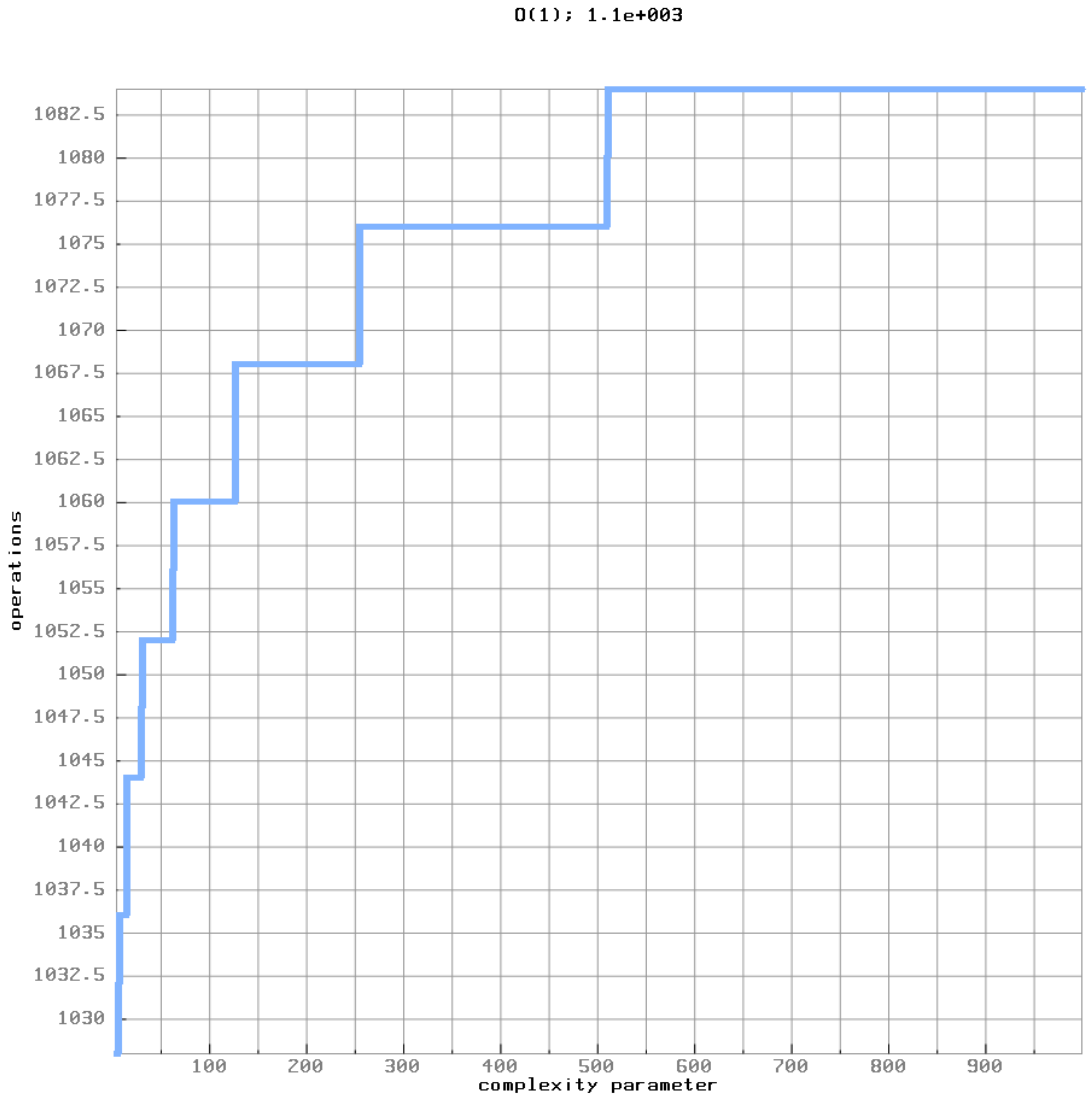

Рисунок 9 – Пример ошибочного выбора класса сложности в следствие лишних вычислений

```
void binarySearch(int border, int key) {

  int x[1000];

  # ANALYSIS_DISABLE # {
  x[0] = 0; x[1] = 1; x[2] = 2; x[3] = 3; x[4] = 4; x[5] = 5; x[6] = 6; x[7] = 7; x[8] = 8; x[9] = 9; x[10] = 10;
  }
  int left = 0;
  int right = border;

  int midd = 0;

  while( true ) {

  midd = (left + right) / 2;

  if  (key < x[midd])
    right = midd - 1;
  else if  (key > x[midd])
  left = midd + 1;

  else break;

  if(left > right)
  break;

  }

}

void main() {
    bool work = true;
    switch(work) {
        case true:
            binarySearch(par, 1500);
            break;
        default:
            break;
    }
}
```

Рисунок 10 – Пример текста алгоритма с использованием нововведений

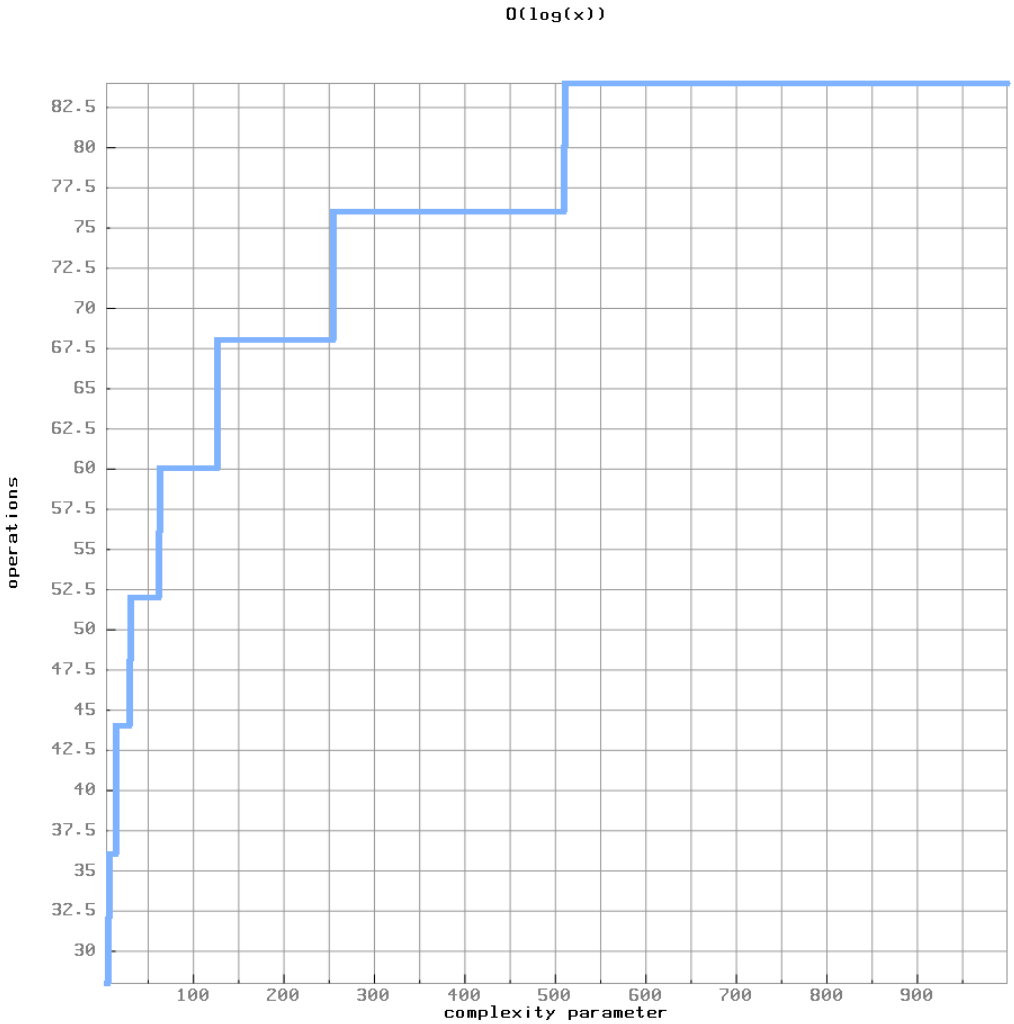

Рисунок 11 – Пример верного выбора класса сложности при использовании нововведений

В результате расширения этапа анализа алгоритма с применением линейной регрессии увеличилась точность определения асимптотического класса сложности. Пример изменений значения степени старшего члена и средней ошибки на каждом из интервалов отображен на рисунке 12.

```
polynomial_degree_changes: 1 polynomial_error_increasings: 0
polynomial_degree_changes: 1 polynomial_error_increasings: 1
polynomial_degree_changes: 1 polynomial_error_increasings: 2
polynomial_degree_changes: 1 polynomial_error_increasings: 3
polynomial_degree_changes: 1 polynomial_error_increasings: 3
polynomial_degree_changes: 1 polynomial_error_increasings: 3
polynomial_degree_changes: 1 polynomial_error_increasings: 3
polynomial_degree_changes: 1 polynomial_error_increasings: 3
polynomial_degree_changes: 1 polynomial_error_increasings: 3

logarithmic_degree_changes: 1 logarithmic_error_increasings: 0
logarithmic_degree_changes: 1 logarithmic_error_increasings: 1
logarithmic_degree_changes: 1 logarithmic_error_increasings: 2
logarithmic_degree_changes: 1 logarithmic_error_increasings: 3
logarithmic_degree_changes: 1 logarithmic_error_increasings: 3
logarithmic_degree_changes: 1 logarithmic_error_increasings: 3
logarithmic_degree_changes: 1 logarithmic_error_increasings: 3
logarithmic_degree_changes: 1 logarithmic_error_increasings: 3
logarithmic_degree_changes: 2 logarithmic_error_increasings: 3

exponential_degree_changes: 1exponential_error_increasings: 0
exponential_degree_changes: 1exponential_error_increasings: 1
exponential_degree_changes: 1exponential_error_increasings: 2
exponential_degree_changes: 1exponential_error_increasings: 3
exponential_degree_changes: 1exponential_error_increasings: 3
exponential_degree_changes: 1exponential_error_increasings: 3
exponential_degree_changes: 1exponential_error_increasings: 3
exponential_degree_changes: 1exponential_error_increasings: 3
```

Рисунок 12 – Изменения средней ошибки и старшей степени

На основании величины средней ошибки и выбранных ограничений выбирается верный класс сложности или ответ представляется как «Unknown». Данный этап позволяет точнее определять асимптотический класс сложности (в том случае, если функция принимает вид одного из классов, которые оно может определить).

Если функция сложности рассматриваемого алгоритма относится к полиномиальному классу, программа выдаст более подробный ответ. Пример такой случая отображён на рисунках 13, 14.

```
void matrixComputing() {
    double a[ 100 ][ 100 ] ; double b[ 100 ][ 100 ] ; double c[ 100 ][ 100 ] ;
        int i, j ;
        for ( i = 0 ; i < par ; ++i)
            for ( j = 0 ; j < par ; ++j) {
                a[ i ][ j ] = 1.0 ; b[ i ][ j ] = 1.0 ; c[ i ][ j ] = 0 ;
            }
        i = 4 ; j = 0 ;
        for ( int v = 0 ; v < par ; ++v) {
            a[ i ][ j ] = 2.0 ;
            i -= 1 ;
            j += 1 ;
        }
        i = 4 ; j = 4 ;
        for ( int v = 0 ; v < par ; ++v) {
            b[ i ][ j ] = 2.0 ;
            i -= 1 ;
            j -= 1 ;
        }
        a[ 2 ][ 4 ] = 3.0 ;
        for ( i = 0 ; i < par ; ++i) {
            for ( j = 0 ; j < par ; ++j)
                            cout << "";     }

                b[ 3 ][ 2 ] = 5.0 ;
        for ( i = 0 ; i < par ; ++i) {
            for ( j = 0 ; j < par ; ++j)
                                cout << ""; }
                for ( i = 0 ; i < par ; ++i)
            for ( j = 0 ; j < par ; ++j)
                for ( int k = 0 ; k < par ; ++k) c[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ] ;
        for ( i = 0 ; i < par ; ++i) {
            for ( j = 0 ; j < par  ; ++j)
                                cout << ""; }
    }

void main() {
    matrixComputing();
}
```

Рисунок 13 – Пример текста алгоритма полиномиальной сложности

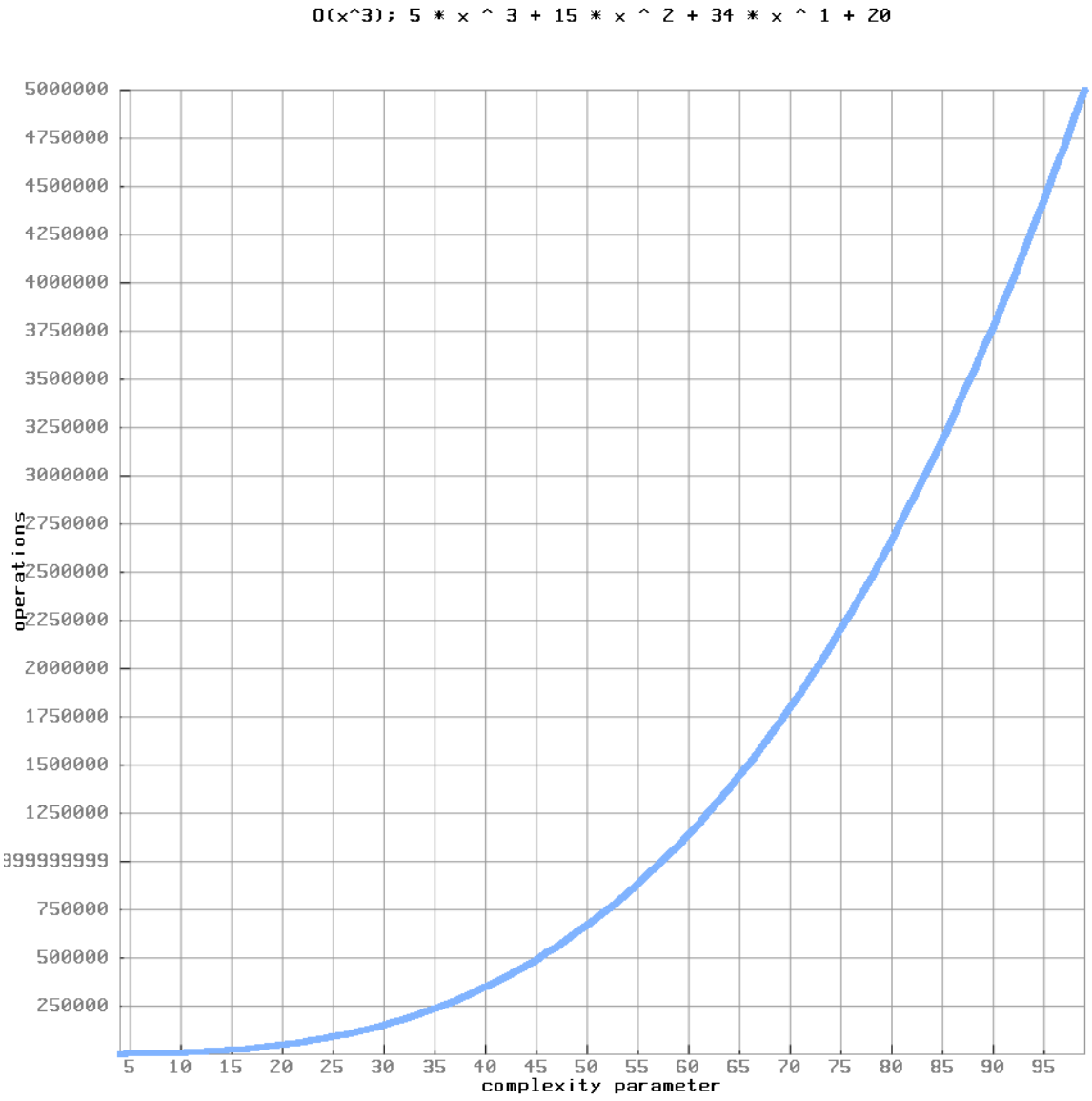

Рисунок 14 – Пример вывода асимптотического и точного значения функции сложности рассматриваемого алгоритма

Чтобы убедиться в оптимизации процесса подсчёта числа элементарных операций на некоторых значениях параметра путём введения средств параллельного вычисления достаточно запустить анализ вышеуказанного алгоритма на достаточно большом интервале параметров. На рисунке 15 отображены преимущества использования параллельного подхода (результаты приведены в количестве миллисекунд, затраченных на подсчёт числа операций на каждой из 10 итераций).

x_1	y_1	y_2
1	37200	86603
2	30152	94321
3	35768	84582
4	34679	76121
5	41563	89211
6	37547	84725
7	32643	87512
8	37251	96142
9	34412	89411
10	36157	98617

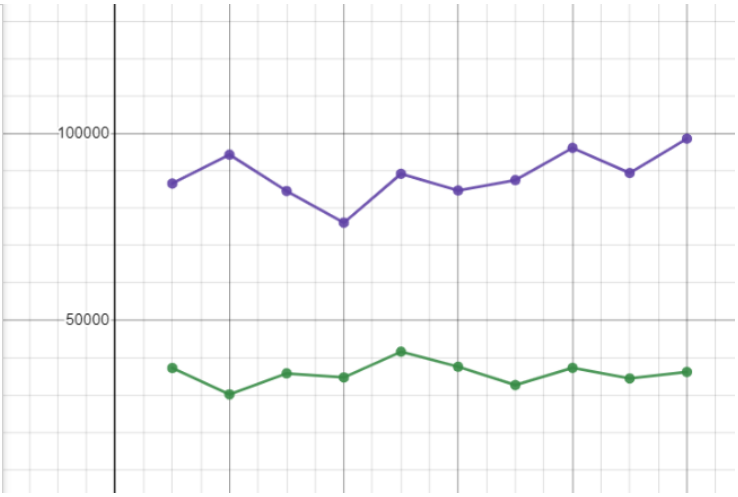

Рисунок 15 – Число затраченных миллисекунд при запуске анализа на 10 итерациях (параметр изменяется от 4 до 1000 с шагом равным 1)

ЗАКЛЮЧЕНИЕ

Анализ сложности алгоритмов приходится проводить вручную, что может потребовать значительных затрат сил вка, пытающегося внедрить данный алгоритм в проектируемую систему. Поэтому актуальность автоматизации и оптимизации данного процесса и поиска оптимальных методов его программной реализации неоспорима. Дальнейшее развитие инструментария поиска сложности алгоритмов для улучшения точности определения классов сложности является важным направлением.

В результате работы в приложение, организующее транскомпиляцию кода на языке описания алгоритмов Algolite в код на языке C++ с дальнейшим определением класса сложности рассматриваемого алгоритма и построением графика сложности, были введены новые функциональные элементы языка описания алгоритмов, методы определения асимптотического и точного вида функции сложности и механизмы параллельного вычисления.

С помощью усовершенствованной программы были проведены тестирования на некотором числе существующих алгоритмов, повсеместно применяемых разработчиками, отображающие высокую точность определения классов сложности, что позволяет ускорить процесс внедрения или отсеивания алгоритма при конструировании системы. Скорость работы программы была заметно увеличена благодаря перечисленным нововведениям.

На основании полученного опыта при разработке и результатах проведённых тестов можно сделать вывод, что использование автоматизированного и оптимизированного метода оценки сложности некоторого алгоритма на основе программной реализации с вычислениями при помощи различных приёмов регрессии способно внести ощутимый вклад в упрощение процесса написания программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Петренко, Ф. Р. Программа оценки сложности алгоритмов : специальность 02.03.03 «Математическое обеспечение и администрирование информационных систем» : курсовая работа / Петренко Филипп Романович ; Кубанский государственный университет. – Краснодар, 2021. – 88 с. – Место защиты: Кубанский государственный университет. – Библиогр.: с. 47, неопубликовано.

2 Миков, А. И. Вычислимость и сложность алгоритмов: учебное пособие / А. И. Миков, О. Н. Лапина. – Краснодар: Кубан. гос. ун-т, 2013. – 448 с.

3 Cozac, E. Big O Notation / E. Cozac // Хабр: [сайт]. – 2021. – URL: https://habr.com/ru/post/559518/ (дата обращения: 06.12.2021).

4 Conery, R. Big O / R. Conery // Хабр: [сайт]. – 2019. – URL: https://habr.com/ru/post/444594/ (дата обращения: 06.12.2021).

5 Орехов, Э.Ю. Оценка качества эвристических алгоритмов / Э.Ю. Орехов, Ю.В. Орехов. – Кишинёв : LAP Lambert Academic Publishing, 2012. – 60 с. – ISBN 978-3-65922-941-1.

6 Халафян, А. А. Теория вероятностей и математическая статистика: учеб. пособие / А. А. Халафян, Г. В. Калайдина, Е. Ю. Пелипенко. – Краснодар: Кубанский гос. университет, 2018. – 184 с. – ISBN 978-5-8209-1462-1.

7 Федотов, И.Е. Параллельное программирование. Модели и приемы / И.Е. Федотов. – Москва : Солон-пресс, 2017. – 390 с. – ISBN 978-5-91359-222-4.

8 Городничев, М.А. Параллельное программирование над общей памятью. POSIX Threads : учеб. пособие / М.А. Городничев, С.Б. Арыков, Г.А. Щукин. – Новосибирск : НГТУ, 2018. – 85 с. – ISBN 978-5-7782-3642-4.

ПРИЛОЖЕНИЕ А

Файл AlgoliteParser.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "Nodes.h"
#include "TranslatingVisitor.h"

#define SUCCESS_COLORIZED_COUT "\033[1m\033[32m"
#define ERROR_COLORIZED_COUT "\033[1m\033[31m"
#define STOP_COLORIZED_COUT "\033[0m"

using namespace std;
string text;
string current_id;
string string_literal;
int index, line, pos;
int prev_symbol_index, prev_symbol_line, prev_symbol_pos;
bool is_real_number;
bool has_main = false;
bool in_non_analysis_block = false;
double number_value;

string readSource(string file_name) {
	ifstream f(file_name);
	f.seekg(0, ios::end);
	size_t size = f.tellg();
	string s(size, ' ');
	f.seekg(0);
	f.read(&s[0], size);
	return s;
}

enum TokenType {
	LParen = '(',
	RParen = ')',
	LBrace = '{',
	RBrace = '}',
	LBracket = '[',
	RBracket = ']',
	Comma = ',',
	Plus = '+',
	Minus = '-',
	Div = '/',
	Mod = '%',
	Mult = '*',
	Equals = '=',
	Less = '<',
	Greater = '>',
	Colon = ':',
	Pragma = '#',
	Semicolon = ';',
	NewLine = '\n',
	EndOfFile = '\0',
	If = 256,
	Else,
	For,
	While,
	Do,
```

```
	Switch,
	Default,
	Case,
	AnalysisDisable,
	Cout,
	LeftLeft,
	RightRight,
	And,
	Or,
	LessEq,
	GreaterEq,
	PlusEq,
	MinusEq,
	MultEq,
	DivEq,
	ModEq,
	NotEq,
	EqualsEq,
	Inc,
	Dec,
	Break,
	Continue,
	Return,
	Void,
	Int,
	Long,
	Double,
	Bool,
	Ident,
	Number,
	LogicalValue,
	StringLiteral,
	Parameter
};

TokenType symbol;
string id();
//HashTable* hashTable;
//HashIndex hashForIdent;

...

void nextSymbol() {
	is_real_number = false;
	skipSpaces();
	savePrevSymbolData();

	if (index == text.length()) {
		symbol = EndOfFile;
		return;
	}
	switch (text[index]) {
	case '(': case ')':
	case '{': case '}':
	case '[': case ']':
	case ',': case ';':
	case ':': case '#':
		symbol = (TokenType)text[index];
		index++;
		pos++;
		break;

	case '\n':
```

```
		line++;
		symbol = NewLine;
		index++;
		pos = 1;
		break;

	case '&':
		if (text[index + 1] == '&') {
			symbol = And;
			index += 2;
			pos++;
			break;
		}
		else error("Неизвестный символ");

	case '|':
		if (text[index + 1] == '|') {
			symbol = Or;
			index += 2;
			pos++;
			break;
		}
		else error("Неизвестный символ");

	case '<': case '>':
	case '-': case '+':
	case '!': case '=':
	case '/': case '%':
	case '*':
		if (text[index + 1] != '=') {
			if (text[index] == '+' && text[index + 1] == '+') {
				symbol = Inc;
				index += 2;
				pos += 2;
				break;
			}
			else if (text[index] == '-' && text[index + 1] == '-') {
				symbol = Dec;
				index += 2;
				pos += 2;
				break;
			}
			else if (text[index] == '<' && text[index + 1] == '<') {
				symbol = LeftLeft;
				index += 2;
				pos += 2;
				break;
			}
			else if (text[index] == '>' && text[index + 1] == '>') {
				symbol = RightRight;
				index += 2;
				pos += 2;
				break;
			}
			else {
				symbol = (TokenType)text[index];
				index++;
				pos++;
			}
		}
		else {
			switch (text[index]) {
			case '<': symbol = LessEq; break;
```

```
                case '+': symbol = PlusEq; break;
                case '-': symbol = MinusEq; break;
                case '*': symbol = MultEq; break;
                case '/': symbol = DivEq; break;
                case '%': symbol = ModEq; break;
                case '>': symbol = GreaterEq; break;
                case '!': symbol = NotEq; break;
                case '=': symbol = EqualsEq; break;
                default: break;
                }
                index += 2;
                pos += 2;
          }
          break;

    case '"':
          symbol = StringLiteral;
          string_literal = "\"";
          index++;
          pos++;
          while (text[index] != '"') {
                string_literal += text[index];
                if (index == text.length())
                      error("Неизвестный символ");
                else {
                      index++;
                      pos++;
                }
          }
          string_literal += "\"";
          index++;
          pos++;
          break;

    default:

          if (digit()) {
                readNumber();
          }

          else if (letter()) {
                current_id = id();
                if (current_id == "ANALYSIS_DISABLE") symbol =
AnalysisDisable;
                else if (current_id == "for") symbol = For;
                else if (current_id == "while") symbol = While;
                else if (current_id == "do") symbol = Do;
                else if (current_id == "switch") symbol = Switch;
                else if (current_id == "default") symbol = Default;
                else if (current_id == "case") symbol = Case;
                else if (current_id == "if") symbol = If;
                else if (current_id == "else") symbol = Else;
                else if (current_id == "break") symbol = Break;
                else if (current_id == "continue") symbol = Continue;
                else if (current_id == "return") symbol = Return;
                else if (current_id == "void") symbol = Void;
                else if (current_id == "int") symbol = Int;
                else if (current_id == "long") symbol = Long;
                else if (current_id == "double") symbol = Double;
                else if (current_id == "bool") symbol = Bool;
                else if (current_id == "true" || current_id == "false")
symbol = LogicalValue;
                else if (current_id == "cout") symbol = Cout;
```

```
                else if (current_id == "parameter") symbol = Parameter;
                else {
                      symbol = Ident;
                }
          }
          else error("Неизвестный символ");
    }
}

...

unique_ptr<StatementNode> command() {
    unique_ptr<StatementNode> result_command;
    switch (symbol) {
    case If:
          result_command = ifCommand();
          break;

    case For:
          result_command = forCommand();
          break;

    case Switch:
          result_command = switchCommand();
          break;

    case While:
          result_command = whileCommand();
          break;

    case Pragma: {
          nextSymbol();
          accept(AnalysisDisable);
          accept(Pragma);
          accept(LBrace);

          unique_ptr<NonAnalysisBlockNode> temp_block =
make_unique<NonAnalysisBlockNode>();
          while (symbol != RBrace)
                temp_block->commands.push_back(command());
          result_command = move(temp_block);
          nextSymbol();
          break;
    }

    case LBrace: {
          unique_ptr<BlockNode> temp_block = make_unique<BlockNode>();
          nextSymbol();
          while (symbol != RBrace)
                temp_block->commands.push_back(command());
          result_command = move(temp_block);
          nextSymbol();
          break;
    }

    default:
          if (symbol == Do)
                result_command = doWhileCommand();
          else
                result_command = simpleCommand();
          accept(Semicolon);
          break;
    }
```

```
	return result_command;
}

...

int main() {
	setlocale(LC_ALL, "ru");
	initText();
	Program current_program = program();
	TranslatingVisitor visitor;
	current_program.visit(visitor);

	cout << SUCCESS_COLORIZED_COUT << "Трансляция прошла успешно" <<
STOP_COLORIZED_COUT << "\n\n";
	system("uncrustify -c msvc.cfg -f finalOutput.cpp --no-backup -o
finalOutput.cpp");
	system("g++ -O3 -std=c++0x -o output_program.exe finalOutput.cpp
pbPlots.cpp supportLib.cpp -lm -D _WIN32_WINNT=0x0A00");
	system("start output_program.exe");
}
```

ПРИЛОЖЕНИЕ Б

Файл TranslatingVisitor.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include "Nodes.h"
#include "TranslatingVisitor.h"

#define ERROR_COLORIZED_COUT "\033[1m\033[31m"
#define STOP_COLORIZED_COUT "\033[0m"

bool in_for_translating = false;
bool writing_mode = true;
bool analysis_mode = true;
stringstream analysing_program;

int _expression_counter;

...

void TranslatingVisitor::handle(DefaultNode& n) {
      if (writing_mode)
            analysing_program << "default: " << "\n";

      for (const unique_ptr<StatementNode>& command : n.commands) {
            if (writing_mode)
                  analysing_program << "    ";
            command->visit(*this);
            if (writing_mode)
                  analysing_program << "\n";
      }
}

...

void TranslatingVisitor::handle(SwitchNode& n) {
      _expression_counter = 0;
      writing_mode = false;

      n.expression_for_switch->visit(*this);

      writing_mode = true;

      if (analysis_mode)
            analysing_program << "_counter += " << _expression_counter << ";\n";

      analysing_program << "switch" << "(";
      n.expression_for_switch->visit(*this);
      analysing_program << ") ";

      analysing_program << "{ " << "\n";

      for (const unique_ptr<CaseNode>& case_command : n.cases) {
            analysing_program << "    ";
            case_command->visit(*this);
      }

      if (n.default_case != nullptr) {
            analysing_program << "    ";
```

```
            n.default_case->visit(*this);
      }

      analysing_program << " }";
}

void TranslatingVisitor::handle(NonAnalysisBlockNode& n) {
      analysis_mode = false;

      for (const unique_ptr<StatementNode>& command : n.commands) {
            if (writing_mode)
                  analysing_program << "    ";
            command->visit(*this);
            if (writing_mode)
                  analysing_program << "\n";
      }

      analysis_mode = true;
}

...

void TranslatingVisitor::handle(Program& n) {
      ifstream
initialization_file("ProgramPartFiles/ProgramInitialization.txt");

      if (initialization_file && writing_mode) {
            analysing_program << initialization_file.rdbuf();
            initialization_file.close();
      }

      if (writing_mode)
            analysing_program << "__thread long long _counter;" << "\n";
            analysing_program << "__thread long long " << n.parameter_name <<
";" << "\n";
            analysing_program << "__thread int global_thread_number;" << "\n";
            analysing_program << "vector<pair<long long, double>> func_vec(" <<
(n.parameter_max - n.parameter_min + n.parameter_step - 1) / n.parameter_step
<< ");\n\n";

      for   (const   unique_ptr<FunctionDefinitionNode>&   func_definition   :
n.functions) {
            func_definition->visit(*this);
            if (writing_mode)
                  analysing_program << "\n\n";
      }

      if (writing_mode) {
            analysing_program    <<    "void    fillComplexityFuncInThread(int
thread_number) {" << "\n";
            analysing_program  <<  "global_thread_number  =  thread_number;"  <<
"\n";
            analysing_program << "int index = 0;" << "\n";
            analysing_program  <<  "for  ("  <<  n.parameter_name  <<  "  =  "  <<
n.parameter_min << " + " << n.parameter_step << " * "  << "global_thread_number"
<<  ";  "  <<  n.parameter_name  <<  "  <  "  <<  n.parameter_max  <<  ";  "  <<
n.parameter_name << " += " << n.parameter_step << " * 4) {" << "\n";
            analysing_program << "_counter = 0; " << "\n";
            analysing_program << "_main(); " << "\n";
            analysing_program  <<  "func_vec[global_thread_number  +  index]  =
make_pair(" << n.parameter_name << ", _counter); " << "\n";
            analysing_program << "index += 4;" << "\n";
            analysing_program << "}" << "\n";
```

```
            analysing_program << "}" << "\n\n";
      }

      if (writing_mode) {
            analysing_program << "void createComplexityFunc() {" << "\n";
            analysing_program << "thread t0(fillComplexityFuncInThread, 0);" <<
"\n";
            analysing_program << "thread t1(fillComplexityFuncInThread, 1);" <<
"\n";
            analysing_program << "thread t2(fillComplexityFuncInThread, 2);" <<
"\n";
            analysing_program << "thread t3(fillComplexityFuncInThread, 3);" <<
"\n";
            analysing_program << "t0.join();" << "\n";
            analysing_program << "t1.join();" << "\n";
            analysing_program << "t2.join();" << "\n";
            analysing_program << "t3.join();" << "\n";
            analysing_program << "}" << "\n\n";
      }

      ifstream
complexity_class_file("ProgramPartFiles/ComplexityClassSearching.txt");
      if (complexity_class_file) {
            if (writing_mode)
                  analysing_program << complexity_class_file.rdbuf();
            complexity_class_file.close();
      }

      ifstream main_func_file("ProgramPartFiles/MainFunction.txt");
      if (main_func_file) {
            if (writing_mode)
                  analysing_program << main_func_file.rdbuf();
            main_func_file.close();
      }

      ofstream f("finalOutput.cpp", ios::out);
      f << analysing_program.str();
      f.close();
}
```

ПРИЛОЖЕНИЕ В

Файл Nodes.h

```
#pragma once
#include <string>
#include <vector>
using namespace std;

...

struct DefaultNode;
struct CaseNode;
struct SwitchNode;
struct NonAnalysisBlockNode;

...

struct Visitor {
	virtual void handle(ArgumentNode& n) = 0;
	virtual void handle(FunctionDefinitionNode& n) = 0;
	virtual void handle(ParenExpressionNode& n) = 0;
	virtual void handle(BinaryOperationNode& n) = 0;
	virtual void handle(UnaryOperationNode& n) = 0;
	virtual void handle(IntegerLiteralNode& n) = 0;
	virtual void handle(DoubleLiteralNode& n) = 0;
	virtual void handle(BooleanLiteralNode& n) = 0;
	virtual void handle(StringLiteralNode& n) = 0;
	virtual void handle(VariableNode& n) = 0;
	virtual void handle(IfNode& n) = 0;
	virtual void handle(WhileNode& n) = 0;
	virtual void handle(ForNode& n) = 0;
	virtual void handle(DefaultNode& n) = 0;
	virtual void handle(CaseNode& n) = 0;
	virtual void handle(SwitchNode& n) = 0;
	virtual void handle(NonAnalysisBlockNode& n) = 0;
	virtual void handle(BlockNode& n) = 0;
	virtual void handle(FunctionCallNode& n) = 0;
	virtual void handle(FunctionCallAsStatementNode& n) = 0;
	virtual void handle(CoutNode& n) = 0;
	virtual void handle(IncDecNode& n) = 0;
	virtual void handle(BreakNode& n) = 0;
	virtual void handle(ContinueNode& n) = 0;
	virtual void handle(ReturnNode& n) = 0;
	virtual void handle(VariableDeclarationNode& n) = 0;
	virtual void handle(AssignmentNode& n) = 0;
	virtual void handle(Program& n) = 0;
};

...

struct DefaultNode : public StatementNode {
	vector<unique_ptr<StatementNode>> commands;

	void visit(Visitor& v) override {
		v.handle(*this);
	}
};

struct CaseNode : public StatementNode {
	unique_ptr<ExpressionNode> condition;
	vector<unique_ptr<StatementNode>> commands;
```

```
      void visit(Visitor& v) override {
            v.handle(*this);
      }
};

struct SwitchNode : public StatementNode {
      unique_ptr<ExpressionNode> expression_for_switch;
      vector<unique_ptr<CaseNode>> cases;
      unique_ptr<DefaultNode> default_case;

      void visit(Visitor& v) override {
            v.handle(*this);
      }
};

struct NonAnalysisBlockNode : public StatementNode {
      vector<unique_ptr<StatementNode>> commands;

      void visit(Visitor& v) override {
            v.handle(*this);
      }
};

...
```

ПРИЛОЖЕНИЕ Г

Файл ComplexityClassSearching.txt

```
void findComplexityClass() {
    vector<pair<long long, double>> func;
    vector<pair<long long, double>> logarithmic_func;
    vector<pair<long long, double>> exponential_func;
    pair<double, double> params;
    int polynomial_degree, logarithmic_degree, exponential_degree, func_size,
part, mod;
    double polynomial_error, logarithmic_error, exponential_error;

    int polynomial_error_increasings = 0;
    int logarithmic_error_increasings = 0;
    int exponential_error_increasings = 0;
    int polynomial_degree_changes = 0;
    int logarithmic_degree_changes = 0;
    int exponential_degree_changes = 0;
    double last_error = INFINITY;
    int last_degree = -1;
    const int PARTS_NUMBER = 10;
    complete_complexity_function = "";

    createComplexityFunc();

    func = func_vec;

    logarithmic_func = func;
    for (pair<long long, double>& element : logarithmic_func) {
        element.second /= log10(element.first);
    }

    exponential_func = func;

    deleteOverflowing(func_vec);
    deleteOverflowing(func);

    func_size = func.size();
    part = func_size / PARTS_NUMBER;
    mod = func_size - part * PARTS_NUMBER;

    for (int i = part + mod; i < func_size; i += part) {
        vector<pair<long long, double>> current_func(func.begin(),
func.begin() + i);
        params = linearRegression(current_func, true, true);

        double current_error = params.first;
        int current_degree = round(abs(params.second));

        if (current_error > last_error * 1.1)
            polynomial_error_increasings++;
        last_error = current_error;

        if (current_degree != last_degree)
            polynomial_degree_changes++;
        last_degree = current_degree;
    }

    polynomial_error = last_error;
    polynomial_degree = last_degree;
```

```
    last_error = INFINITY;
    last_degree = -1;

    deleteOverflowing(logarithmic_func);

    func_size = logarithmic_func.size();
    part = func_size / PARTS_NUMBER;
    mod = func_size - part * PARTS_NUMBER;

    for (int i = part + mod; i < func_size; i += part) {
        vector<pair<long long, double>>
current_func(logarithmic_func.begin(), logarithmic_func.begin() + i);
        params = linearRegression(current_func, true, true);

        double current_error = params.first;
        int current_degree = round(abs(params.second));

        if (current_error > last_error * 1.1)
            logarithmic_error_increasings++;
        last_error = current_error;

        if (current_degree != last_degree)
            logarithmic_degree_changes++;
        last_degree = current_degree;
    }

    logarithmic_error = last_error;
    logarithmic_degree = last_degree;

    last_error = INFINITY;
    last_degree = -1;

    deleteOverflowing(exponential_func);

    func_size = exponential_func.size();
    part = func_size / PARTS_NUMBER;
    mod = func_size - part * PARTS_NUMBER;

    for (int i = part + mod; i < func_size; i += part) {
        vector<pair<long long, double>>
current_func(exponential_func.begin(), exponential_func.begin() + i);
        params = linearRegression(current_func, false, true);

        double current_error = params.first;
        int current_degree = round(abs(params.second));

        if (current_error > last_error * 1.1)
            exponential_error_increasings++;
        last_error = current_error;

        if (current_degree != last_degree)
            exponential_degree_changes++;
        last_degree = current_degree;
    }

    exponential_error = last_error;
    exponential_degree = last_degree;

    bool class_is_unknown = false;

    complexity_class = "O(";
```

```
    if (polynomial_error < logarithmic_error && polynomial_error <
exponential_error) {
        if ((polynomial_error_increasings <= PARTS_NUMBER / 2 ||
polynomial_error < 1e-04) && polynomial_degree_changes < 5) {
            if (polynomial_degree != 0) {
                complexity_class += "x";
                if (polynomial_degree > 1 && polynomial_degree <= 5)
                    complexity_class += "^" + to_string(polynomial_degree);
                else if (polynomial_degree > 5)
                    complexity_class = "> " + complexity_class + "^5";
            }
            else
                complexity_class += "1";
            complete_complexity_function = "; " +
polynomialComplexityFunction(polynomial_degree, func_vec);
        }
        else
            class_is_unknown = true;
    }
    else if (logarithmic_error < polynomial_error && logarithmic_error <
exponential_error) {
        if ((logarithmic_error_increasings <= PARTS_NUMBER / 2 ||
logarithmic_error < 1e-04) && logarithmic_degree_changes < 5) {
            if (logarithmic_degree != 0) {
                complexity_class += "x";
                if (logarithmic_degree > 1 && logarithmic_degree <= 5)
                    complexity_class += "^" + to_string(logarithmic_degree);
                else if (logarithmic_degree > 5)
                    complexity_class = "> " + complexity_class + "^5";
                complexity_class += " * ";
            }
            complexity_class += "log(x)";
        }
        else
            class_is_unknown = true;
    }
    else if (exponential_error <= polynomial_error && exponential_error <=
logarithmic_error) {
        if ((exponential_error_increasings <= PARTS_NUMBER / 2 ||
exponential_error < 1e-04) && exponential_degree_changes < 5) {
            if (exponential_degree != 0) {
                complexity_class += "e^";
                if (exponential_degree > 1)
                    complexity_class += "(" + to_string(exponential_degree) +
"x)";
                else
                    complexity_class += "x";
            }
            else
                complexity_class += "1";
        }
        else
            class_is_unknown = true;
    }
    else
        class_is_unknown = true;
    complexity_class += ")" + complete_complexity_function;

    if (class_is_unknown)
        complexity_class = "Unknown";
    w = wstring(complexity_class.begin(), complexity_class.end());
    graph_title = w.c_str();
}
```

ПРИЛОЖЕНИЕ Д

Файл MainFunction.txt

```
int main() {
    findComplexityClass();

    RGBABitmapImageReference* imageReference =
CreateRGBABitmapImageReference();

    for (auto points_pair = func_vec.begin(); points_pair != func_vec.end();
points_pair++) {
        xs.push_back((*points_pair).first);
        ys.push_back((*points_pair).second);
    }

    ScatterPlotSeries* series = GetDefaultScatterPlotSeriesSettings();
    series->xs = &xs;
    series->ys = &ys;
    series->linearInterpolation = true;
    series->lineThickness = 5;
    series->color = CreateRGBColor(0.5, 0.7, 1);

    ScatterPlotSettings* settings = GetDefaultScatterPlotSettings();
    settings->width = 1000;
    settings->height = 1000;
    settings->autoBoundaries = true;
    settings->autoPadding = true;
    settings->title = toVector(graph_title);
    settings->xLabel = toVector(L"complexity parameter");
    settings->yLabel = toVector(L"operations");
    settings->scatterPlotSeries->push_back(series);
    settings->gridColor = CreateRGBColor(0.6, 0.6, 0.6);
    settings->showGrid = true;

    DrawScatterPlotFromSettings(imageReference, settings);

    vector<double>* pngdata = ConvertToPNG(imageReference->image);

    WriteToFile(pngdata, "func.png");
    DeleteImage(imageReference->image);
    system("start func.png");
}
```

ПРИЛОЖЕНИЕ Е

Файл ProgramInitialization.txt

```
#include "pbPlots.hpp"
#include "supportLib.hpp"
#include "mingw.thread.h"
#include <vector>
#include <cmath>
#include <iostream>
#include <limits>
#include <string>
#include <sstream>
#define LLONG_MAX numeric_limits<long long>().max()

using namespace std;
#include "matrixOperations.cpp"

vector<double> xs;
vector<double> ys;
vector<double> xs1;
vector<double> ys1;
string complexity_class;
string complete_complexity_function;
const wchar_t* graph_title;
wstring w;

pair<double, double> linearRegression(vector<pair<long long, double>> func,
bool log_first_var, bool log_second_var) {
    double error = 0;
    long long n = func.size();
    double a;
    double b;

    double v_sum = 0;
    double u_sum = 0;
    double vu_sum = 0;
    double vv_sum = 0;

    for (long long i = 0; i < n; i++) {

        double v = log_first_var ? log10(func[i].first) : func[i].first;
        double u = log_second_var ? log10(func[i].second) : func[i].second;

        v_sum += v;
        u_sum += u;
        vu_sum += v * u;
        vv_sum += v * v;
    }

    b = (n * vu_sum - v_sum * u_sum) / (n * vv_sum - v_sum * v_sum);
    a = (u_sum - b * v_sum) / n;

    for (long long i = 0; i < n; i++) {
        double v = log10(func[i].first);
        double u = log10(func[i].second);

        double estimated_u = a + b * v;
        error += (estimated_u - u) * (estimated_u - u);
    }

    error /= n;
```

```
    return make_pair(error, b);
}

Vector polynomialRegression(int polynomial_degree, vector<pair<long long,
double>> func) {
    createXY(polynomial_degree, func);
    Matrix left_matrix(polynomial_degree + 1);
    Vector right_vector(polynomial_degree + 1);
    Vector result(polynomial_degree + 1);

    for (int i = 0; i < left_matrix.size(); i++) {
        Vector current_vector(polynomial_degree + 1);
            left_matrix[i] = current_vector;
    }

    multiplyMTM(x_matrix, left_matrix);
    multiplyMTV(x_matrix, y_vector, right_vector);

    getGaussianSolution(left_matrix, right_vector, result);

    return result;
}

string polynomialComplexityFunction(int polynomial_degree, vector<pair<long
long, double>> func) {
    vector<double> resultCoefficients =
polynomialRegression(polynomial_degree, func);
    stringstream result;

    result.precision(2);

    for (int i = polynomial_degree; i > 0; i--) {
        result << resultCoefficients[i] << " * x ^ " << i << " + ";
    }
    result << resultCoefficients[0];

    return result.str();
}

void deleteOverflowing(vector<pair<long long, double>>& func) {
    for (auto points_pair = func.begin(); points_pair != func.end();
points_pair++) {
        if ((*points_pair).second > 0.6 * LLONG_MAX || (*points_pair).second
< 0.0) {
            func.erase(points_pair, func.end());
            break;
        }
    }
}
```

ПРИЛОЖЕНИЕ Ж

Файл matrixOperations.cpp

```
using Vector = vector<double>;
using Matrix = vector<Vector>;

Matrix x_matrix;
Matrix x_matrix_tra;
Vector y_vector;

void createXY(int polynomial_degree, vector<pair<long long, double>> func) {
    Vector current_vector;
    for (const pair<long long, double>& point : func) {
        double current_x = 1;

        for (int i = 0; i <= polynomial_degree; i++) {
            current_vector.push_back(current_x);
            current_x *= point.first;
        }

        x_matrix.push_back(current_vector);
        y_vector.push_back(point.second);

        current_vector.clear();
    }
}

void multiplyMV(const Matrix& a, const Vector& f, Vector& out) {
    for (int i = 0; i < a.size(); i++) {
        out[i] = 0;
        for (int j = 0; j < a[i].size(); j++) {
            out[i] += a[i][j] * f[j];
        }
    }
}

void multiplyMM(const Matrix& a, const Matrix& b, Matrix& out) {
    for (int i = 0; i < a.size(); i++) {
        for (int j = 0; j < b[0].size(); j++) {
            out[i][j] = 0;
            for (int k = 0; k < b.size(); k++) {
                out[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

void multiplyMTM(const Matrix& a, Matrix& out) {
    for (int i = 0; i < a[0].size(); i++) {
        for (int j = 0; j < a[0].size(); j++) {
            out[i][j] = 0;
            for (int k = 0; k < a.size(); k++) {
                out[i][j] += a[k][i] * a[k][j];
            }
        }
    }
}

void multiplyMTV(const Matrix& a, const Vector& f, Vector& out) {
    for (int i = 0; i < a[0].size(); i++) {
        out[i] = 0;
```

```
        for (int j = 0; j < a.size(); j++) {
            out[i] += a[j][i] * f[j];
        }
    }
}

void getGaussianSolution(const Matrix& a, const Vector& f, Vector& out) {
    Matrix m = a;

    out = f;

    for (int i = 0; i < m.size(); i++) {
        int best_index = i;
        double best_val = abs(m[i][i]);
        for (int index = i + 1; index < m.size(); index++)
            if (best_val < abs(m[index][i])) {
                best_val = abs(m[index][i]);
                best_index = index;
            }

        if (best_index != i) {
            for (int j = i; j < m[0].size(); j++)
                swap(m[best_index][j], m[i][j]);
            swap(out[best_index], out[i]);
        }

        double val = m[i][i];
        for (int k = 0; k < m[0].size(); k++) {
            m[i][k] /= val;
        }
        out[i] /= val;

        for (int j = i + 1; j < m.size(); j++) {
            double current_column_element = m[j][i];
            for (int k = 0; k < m[0].size(); k++) {
                m[j][k] -= m[i][k] * current_column_element;
            }
            out[j] -= out[i] * current_column_element;
        }

    }

    for (int i = m.size() - 1; i >= 0; i--) {
        for (int j = i + 1; j < m[0].size(); j++) {
            out[i] -= m[i][j] * out[j];
        }
    }
}
```