УДК 004.891

ПРОЦЕСС ТРАНСЛЯЦИИ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ РАЗНЫХ ДИАЛЕКТОВ С ОПТИМИЗАЦИЕЙ ПРОМЕЖУТОЧНЫХ ПРЕДСТАВЛЕНИЙ

Морошкин Н.А., Демидова Л.А.

МИРЭА – Российский технологический университет, 119454, г. Москва, пр-т Вернадского, 78, e-mail: seed.suboty@gmail.com, demidova.liliya@gmail.com

В статье рассматриваются аспекты трансляции регулярных выражений из исходного в целевой диалект как способа решения задачи сопоставления строки с образом. Целью данной работы является разработка и обоснование архитектуры транслятора регулярных выражений в разные диалекты с учетом оптимизации промежуточных представлений в процессе трансляции. Представлены классификация диалектов регулярных выражений и классификация программных реализаций исполнителей конечных автоматов, описываемых регулярными выражениями. Сформулированы рекомендации по выбору конкретной реализации регулярных выражений в аспекте задач обработки текста. Описан алгоритм оптимизации регулярных выражений с помощью популяционных алгоритмов. Представлены результаты эксперимента по оптимизации промежуточных представлений проверочных регулярных выражений с использованием алгоритма дифференциальной эволюции и алгоритма роя частиц.

Ключевые слова: регулярные выражения, популяционные алгоритмы оптимизации, конечные автоматы, диалекты регулярных выражений, оптимизация регулярных выражений, списки инцидентности

THE PROCESS OF TRANSLATION OF REGULAR EXPRESSIONS OF DIFFERENT DIALECTS WITH OPTIMIZATION OF INTERMEDIATE STATES

Moroshkin N.A., Demidova L.A.

MIREA – Russian Technological University, 119454, Russia, Moscow, Vernadsky Avenue, 78, e-mail: seed.suboty@gmail.com, demidova.liliya@gmail.com

The article considers aspects of translation of regular expressions from the source to the target dialect as a way to solve the problem of matching a string with an image. The purpose of this work is to develop and justify the architecture of a translator of regular expressions into different dialects taking into account the optimization of intermediate representations in the process of translation. The classification of dialects of regular expressions and the classification of software implementations of executors of finite automata described by regular expressions are presented. Recommendations are formulated for choosing a specific implementation of regular expressions in the aspect of text processing problems. An algorithm for optimizing regular expressions using population algorithms is described. The results of an experiment on optimizing intermediate representations of verification regular expressions using the differential evolution algorithm and the particle swarm algorithm are presented.

Keywords: regular expressions, population optimization algorithms, finite automata, regular expression dialects, regular expression optimization, incidence lists

Введение

Развитие сферы текстового анализа позволило исследователям обрабатывать текст с помощью самых разных инструментов – классических алгоритмов машинного обучения (например, «мешок слов», логистическая регрессия вместе с TF-IDF, статистическая мера, используемая для оценки важности слова в контексте документа, являющегося частью коллекции документов или корпуса, векторизацией), нейронных сетей с архитектурой «трансформер», больших языковых моделей и т.д. [1-2]. Однако, почти все перечисленные инструменты не способны решать задачу сопоставления с образом. Решение данной задачи позволяет сделать конкретное, основанное на образе заключение о схожести двух строк. В этом и заключается главное отличие алгоритмов для сопоставления строки с образом и методов, основанных на моделях машинного обучения – в четкости и интерпретируемости результата.

Задача сопоставления с образом используется во многих сферах анализа текстовой информации. Например, в поиске по конкретному запросу, в мониторинге социальных сетей, в лексическом анализе как этапе компиляции программного кода, в анализе поискового запроса в браузере и во многих других [3]. Наиболее распространенным решением задачи являются регулярные выражения (РВ). РВ – инструмент для поиска слов в текстовом корпусе по некоторому шаблону. Соответствующий этому инструменту процесс поиска можно также назвать сопоставлением со строкой. РВ представлены во многих языках программирования в разных реализациях.

Идея регулярных выражений, сформулированная в конце 1960-х годов, предполагала использование детерминированных конечных автоматов в качестве механизма для анализа строки [4]. Затем были предложены РВ на основе недетерминированных конечных автоматов, существенно улучшивших и расширивших возможности регулярных выражений (например, посредством добавления нечетких границ и возможности реализации обратной ссылки) [4]. Также была предложена реализация механизма регулярных выражений на основе автомата Томпсона, позволившая кратно ускорить поиск заданного паттерна в тексте [5].

Однако, при развитии языков программирования было реализовано множество различных диалектов регулярных выражений со своими особенностями и ограничениями. В результате возникла проблема перевода или «трансляции» регулярных выражений из одного диалекта в другой. Например, при решении задачи поиска паттерна в большом корпусе текста может быть использован диалект регулярных выражений языка Python. Данный диалект расширен по сравнению с другими диалектами посредством добавления ряда сложных расширенных конструкций поиска паттерна. При изменении задачи с поиска в корпусе текста на задачу поиска тех же паттернов на потоке данных разработчикам придется применять другие механизмы обработки регулярных выражений, требующие использования другого диалекта. Возможна ситуация, когда некоторый диалект не обладает расширенными конструкциями, что создает проблему обратной совместимости уже написанных и протестированных паттернов. В этом случае для решения такой проблемы необходим механизм трансляции между несколькими диалектами.

Задача сопоставления с образом заключается в построении модели, способной обобщать закономерности и паттерны в данных, чтобы правильно идентифицировать или классифицировать новые образы, которые не были частью обучающего набора данных. Задачи сопоставления с образом решаются в разных областях анализа данных – в компьютерном зрении, в машинном обучении или в обработке естественного языка. В каждой из сфер используются собственные алгоритмы и методы, что говорит об отсутствии мультимодальных алгоритмов (алгоритмов, применимых одновременно к разным типам данных, например, к изображениям и тексту).

В общем виде задача сопоставления с образом включает в себя изучение и анализ различных признаков объектов для последующего принятия решений он соответствии или классификации объектов в исходных данных по заданным образам.

Процесс трансляции регулярного выражения представляет собой представление регулярного выражения входного диалекта в целевом диалекте. После проведения трансляции транслированное регулярное выражение должно быть проверено строками, на которых исходное регулярное выражение находило соответствие. Такие строки можно получить путем генерации по структуре исходного регулярного выражения.

Пусть S – множество сгенерированных строк, которые используются для проверки регулярного выражения на целевом диалекте после процесса трансляции. Пусть A – алфавит, из символов которого состоит множество сгенерированных строк S, а S_g – g-я строка из множества проверочных строк S ($g = \overline{1, G}$). Пусть образ p – множество метасимволов $m_1, m_2..., m_e$, где e – число элементов в множестве.

Существует несколько подходов к задаче сопоставления текстовых образов – сопоставление с константным значением, сопоставление с древовидной структурой и сопоставление со строкой или некоторым образом. В данной работе рассматривается только задача сопоставление входного текста с некоторым паттерном. Существует несколько механизмов для реализации поиска по паттерну в тексте, например, можно использовать символы-джокеры, реализация которых присутствует в языке запросов к базам данных SQL, а именно – в операторе «like», и РВ. Наиболее распространенным способом поиска по паттерну в тексте на данный момент являются РВ.

Разнообразие диалектов регулярных выражений

На данный момент каждый высокоуровневый язык программирования 4-го поколения (Python, Java, PHP, Ruby и т.д.) имеют свои варианты реализации регулярных выражений со своим синтаксисом и математической базой. Из-за этого обратная совместимость различных реализаций программ становится сложной задачей для исследователей, зачастую синтаксис и особенности конкретной реализации полностью игнорируются, что приводит к возможным утечкам памяти, аварийному завершению работы программы и медлительности кода.

В качестве решения этой проблемы предлагается рассмотреть процесс трансляции регулярных выражений из одного синтаксиса в другой. Здесь под синтаксисом понимается способ задания конечного автомата текстом. По сути, РВ – это способ задания некоторого конечного автомата, который может быть использован для прохода по тексту для определения соответствия входного текста и заданного паттерна.

Существует множество различных форм регулярных выражений и множество расширений базовых реализаций, как описано в [7]. РВ в Perl считаются [5, 6, 7] каноническим или начальным вариантом для других работ. Их расширения получили наибольшую известность, среди них можно отметить PCRE/PCRE2 [8], которые часто используют в PHP, библиотека re2 от Google [9], синтаксис которой можно увидеть во многих NoSQL базах данных, например, в ClickHouse (в библиотеке hyperscan) [10]. Все перечисленные и множество других библиотек или синтаксисов имеют общую математическую базу – детерминированный конечный автомат.

Первыми программными реализациями, имеющими возможность практического использования регулярных выражений, были текстовые редакторы (ed, sed). Затем принципы работы с регулярными выражениями были вынесены в отдельную утилиту – grep. В последствии принципы сопоставления текста с образом с помощью регулярных выражениях мигрировали из редакторов в языки программирования. Со временем образовалась группа базовых вариантов описания синтаксиса регулярных выражений, называемая диалектами (по аналогии с языком запросов SQL). Обобщенная классификация регулярных выражений по диалекту представлена на рисунке 1.

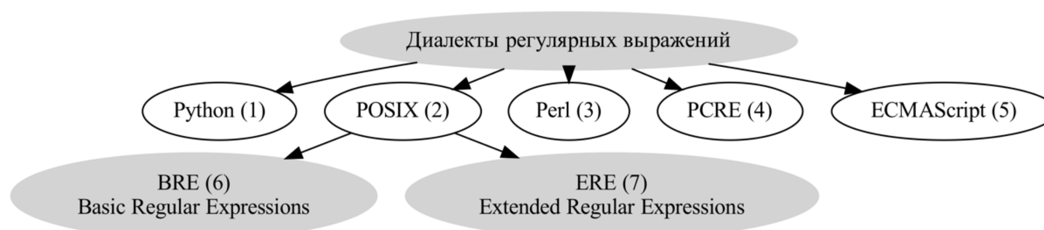

Рисунок 1 – Классификация регулярных выражений по диалекту

Под трансляцией в данной работе понимается процесс преобразования исходного входного текста в текст, написанный на целевом синтаксисе. В случае регулярных выражений – на диалекте.

Диалект Python [11] (рисунок 1) реализован в формате стандартного встроенного модуля re. Данный диалект содержит в себе синтаксис, во многом схожий со стандартными POSIX-совместимыми регулярными выражениями. Широкое распространение языка программирования Python сначала в научных кругах, а затем и в кругах веб-разработки и машинного, а также глубокого обучения, повлияло на популярность данного диалекта регулярных выражений. Существует множество самостоятельных уникальных реализаций механизмов регулярных выражений для различных систем (чаще всего это самостоятельные синтаксисы поиска паттернов в тексте в аналитических low-code платформах [12], которые выходят за рамки данной работы), имеющих синтаксис, заимствованный от модуля re в Python. Также необходимо упомянуть расширенную версию данного модуля, реализованную в виде библиотеки regex для языка Python. Данная библиотека расширяет базовые возможности модуля re, добавляя возможность заглядываний вперед и назад с неограниченной длинной и добавляя нечеткое сопоставление строк (с возможностью выбора класса и числа ошибок).

Диалект POSIX [13] (рисунок 1) состоит из двух диалектов BRE (Basic Regular Expressions, базовые регулярные выражения) и ERE (Extended Regular Expressions, расширенные регулярные выражения). Например, BRE диалект используется программой grep, а ERE программой – egrep. Данные диалекты приняты сообществом исследователей и разработчиков как канонические.

Диалект Perl [14] (рисунок 1) появился в языке программирования Perl, диалект следует синтаксису ERE, дополняя его новыми возможностями. Множество современных диалектов регулярных выражений копируют именно стиль Perl (например, многое из этого диалекта заимствованно у диалекта Python).

Диалект PCRE [15] (Perl Compatible Regular Expressions) (рисунок 1) – диалект, используемый в одноименной библиотеке. Он похож на диалект Perl, но имеет некоторые отличия. Диалект PCRE используется в языке PHP и многих современных графических текстовых редакторах. Данный диалект особенно важен с учетом того, что наиболее часто выбирается исследователями как основа для написания наиболее быстродейственных механизмов регулярных выражений (например, в библиотеке re2 от Google).

Диалект ECMAScript (изначально применяемый в скриптовом языке программирования ECMAScript) [15] (рисунок 1) используется в языках ECMA-спецификации, например в языке программирования JavaScript. Данный диалект формировался под влиянием языка Perl 5 и соответственно диалекта Perl. Он используется по умолчанию в стандартной библиотеке STL, начиная с версии стандарта C++11.

Реализации механизмов регулярных выражений

Многие компоненты электронно-вычислительных устройств, включая аппаратное и программное обеспечение, основаны на концепции конечного автомата. В программировании конечный автомат часто описывается как «чёрный ящик», который принимает сигналы входного алфавита, выдаёт сигналы выходного алфавита и находится в одном из конечного числа состояний.

РВ также могут быть классифицированы по типу программной реализации конечного автомата, описанного выражениями. В общем виде можно разделить все такие программные реализации на два типа – реализованные на основе детерминированного конечного автомата (ДКА) и реализованные на основе недетерминированного конечного автомата (НКА).

Детерминированный конечный автомат (ДКА) – это абстрактная модель, которая может распознавать последовательности символов, где каждая комбинация входных символов приводит к одному определенному состоянию автомата [16]. В ДКА каждое состояние определено однозначно, что позволяет автомату оставаться в одном состоянии при получении данных. Если использовать таблицу для представления переходов в ДКА, каждая запись в ней соответствует только одному состоянию.

Недетерминированный конечный автомат (НКА) – это абстрактная модель, которая принимает вводимые символы и принимает решение о принятии или отклонении слова. НКА может изменять своё состояние, переходя из одного состояния в другое [16]. Структуру такого автомата можно представить в виде графа переходов НКА, который также способен распознавать слова. Слово считается принятым, если после обработки автомат оказывается хотя бы в одном допустимом состоянии. Таким образом, НКА определяет заданный язык.

Любой НКА может быть преобразован в детерминированный таким образом, чтобы их языки, которые они создают, совпадали. Такие автоматы считаются эквивалентными. Однако из-за того, что число состояний в эквивалентном ДКА может экспоненциально возрасти по сравнению с исходным НКА, на практике детерминизация (перевод НКА в эквивалентный ДКА) не всегда выполнима. НКА способен находиться в нескольких состояниях одновременно, что является главным его минусом, если рассматривать его как основу для реализации механизма работы регулярных выражений.

В [17] описаны основные преимущества детерминированного автомата для реализации алгоритма работы регулярного выражения, а именно – простота интерпретации, вычислений, и, как следствие, влияние такой реализации на скорость работы механизма. Исследователи начали изучать возможность расширения базовой функциональности регулярных выражений, в результате они пришли к реализации механизма на основе НКА. Разница в работе двух конечных автоматов описана в [17-18], преимущество использования НКА описано в [18]. При сравнении двух подходов появляется проблема зацикливания в определенных состояниях недетерминированных реализаций, при которых в ряде случаев приходится разрабатывать алгоритм детерминизации НКА для минимизации исходного ДКА, пример такой разработки представлен в работе [19].

Различие между ДКА и НКА в схеме автоматов можно продемонстрировать на примере регулярного выражения «ab?c». Построенные для него НКА и ДКА показаны на рисунке 2. НКА имеет так называемый «ε-переход», по сути означающий альтернативный путь, если искомый символ в текущем состоянии не был найден.

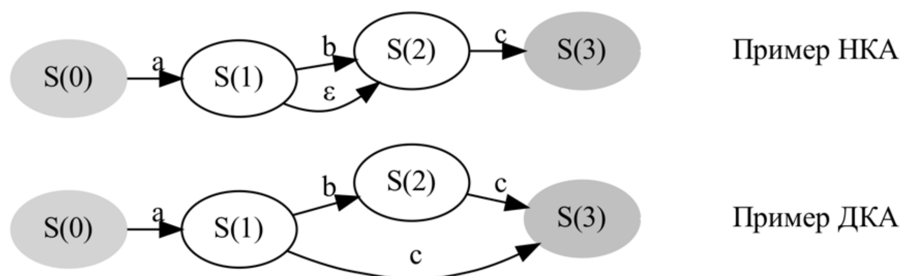

Рисунок 2 – Примеры схем ДКА и НКА для регулярного выражения

НКА используют «жадный алгоритм отката», проверяя все возможные расширения регулярного выражения в определённом порядке и выбирая первое подходящее значение из входной строки. НКА может обрабатывать подвыражения и обратные ссылки. Но из-за алгоритма отката традиционный НКА может проверять одно и то же место несколько раз, что отрицательно сказывается на скорости работы. Это главный минус такой реализации. Однако, гибкость и универсальность НКА позволяет добавлять сложные по семантике конструкции, как, например, в библиотеке regex [20], реализованной на языке Python.

Реализации на ДКА работают линейно по времени, поскольку не используют откаты и никогда не проверяют какую-либо часть текста дважды. Они могут гарантированно найти самую длинную строку из возможных. На данный момент, реализации на ДКА считаются наиболее оптимизированными по скорости работы по сравнению с НКА. Механизм регулярных выражений, основанный на ДКА, исследователи выбирают в тех случаях, когда

можно пренебречь гибкостью и функциональностью регулярных выражений, в угоду скорости работы. Ярким примером такой реализации является библиотека re2 [9] и основанная на ней библиотека hyperscan [21]. Эти две библиотеки намеренно обходят сложные конструкции классических регулярных выражений, они считаются самыми эффективными на данный момент.

Ещё одним подходом к реализации регулярных выражений стал недетерминированный автомат Томпсона [22] или автомат, полученный путем преобразования ДКА в НКА с помощью алгоритма Томпсона [22]. Основные преимущества автомата Томпсона подробно описаны в [17-19], где предложены идеи по минимизации такого автомата. Классификация регулярных выражений по программным реализациям представлена на рисунке 3.

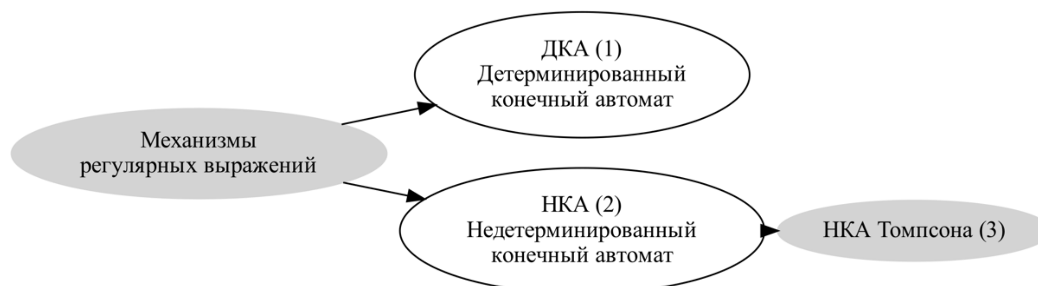

Рисунок 3 – Классификация механизмов регулярных выражений с точки зрения программной реализации

Многообразие всех реализаций показывает, насколько сложна задача обратной совместимости. Зачастую в одном проекте при разработке приходится сталкиваться с несколькими синтаксисами регулярных выражений. Связано это с разными предназначениями каждой реализации. Например, зачастую реализации на НКА используют для сложной аналитики или предобработки, тогда как, ДКА используют для фильтрации на потоке большого объема входных данных. Также отметим малую читаемость некоторых конструкций, которая ведет к ошибкам со стороны операторов информационных систем при написании запросов.

Для решения всех вышеуказанных проблем авторы предлагают транслятор регулярных выражений исходного диалекта в целевой диалект.

Реализации механизмов регулярных выражений

Предлагаемый транслятор, архитектура которого представлена на рисунке 4, будет иметь классическую архитектуру транслятора для языков программирования [23], однако он будет адаптирован под трансляцию регулярных выражений.

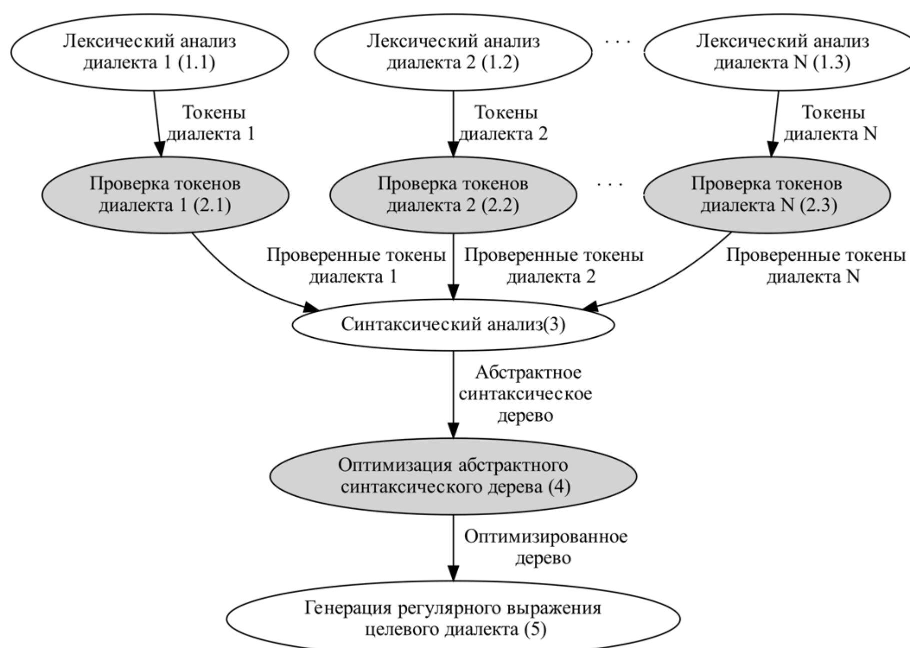

Рисунок 4 – Архитектура транслятора регулярных выражений

На первом этапе транслятору подается регулярное выражение в неизмененном виде. Далее оно разбирается с помощью лексического анализатора (блоки 1.1, 1.2 и 1.N на рисунке 4) на токены, где N – число диалектов,

поддерживаемых транслятором. В данном контексте, токен – лексема, обозначающая одну конструкцию на определенном диалекте регулярных выражений. На этом этапе алгоритм разбора входного регулярного выражения должен быть реализован отдельно для каждого диалекта, так как имеющиеся различия не позволяют реализовать это в едином анализаторе. При этом на этапе лексического анализа происходят валидации (проверки на корректность) конструкций и символов, например, проверка на баланс скобок или допустимость той или иной используемой конструкции.

В общем виде результаты работы этапа лексического анализа и проверки токенов можно представить в виде последовательности токенов $T_1, T_2, \dots T_u$, где u – число найденных токенов. Токен имеет следующий вид: $< dialect.name, param >$, где $dialect$ – идентификатор рассматриваемого диалекта (идентификатор необходим для понимания того, какие ограничения следует учитывать на этапе синтаксического анализа и какие оптимизации применять на этапе оптимизации токенов); $name$ – имя токена, а $param$ – возможный параметр, который может отсутствовать. Например, для регулярного выражения «*ab?c.|d*», написанного на диалекте языка Python, последовательность токенов будет иметь следующий вид: *«1.atom,a», «1.atom,b», «1.0_or_1», «1.atom,c», «1.any», «1.alt», «1.atom,d»*.

В таблице 1 приведены наиболее часто используемые токены для диалекта регулярных выражений языка Python.

Таблица 1 – Наиболее часто используемые [24] токены диалекта регулярных выражений языка Python

Номер токена	Символ	Имя токена	Значение
1	Все допустимые ASCII символы	***<1.atom, param>***, где *param* – это конкретный символ	ASCII символ
2	\	***<1.escape, param>*** где *param* – это следующий в строке символ	Экранирование следующего символа
3	|	***<1.alt>***	Граница двух альтернативных групп
4	.	***<1.any>***	Любой символ
5	?	***<1.0_or_1>***	Квантификатор, повторение 0 или 1 раз
6	*	***<1.0_or_more>***	Квантификатор, повторение любое число раз
7	+	***<1.1_or_more>***	Квантификатор, повторение более 1 раза
8	[	***<1.start_range>***	Начало некоторого диапазона
9	]	***<1.end_range>***	Конец некоторого диапазона
10	(	***<1.start_group>***	Начало группы
11	)	***<1.end_group>***	Конец группы
12	{	***<1.start_quantifier>***	Старт конструкции квантификатора с ручным вводом параметров повторения
13	}	***<1.end_quantifier>***	Конец конструкции квантификатора с ручным вводом параметров повторения
14	^	***<1.start_string>***	Якорь начала введенной строки
15	&	***<1.end_string>***	Якорь конца введенной строки

Таблица 1 описывает не все токены, содержащиеся в диалекте языка Python, однако приведенных токенов достаточно, чтобы провести анализ предлагаемого подхода.

После формирования набора токенов необходимо реализовать проверку токенов (блоки 2.1, 2.2 и 2.N на рисунке 4). Данный процесс подразумевает проверку всех токенов на корректность использования, учитывая особенности и ограничения выбранного синтаксиса. Например, при использовании стандартного модуля re в языке Python может быть написано регулярное выражение, содержащее метасимвол «*», который является образом любого допустимого символа. Этот метасимвол представляет собой квантификатор, означающий повторение всей предыдущей группы, вырожденным случаем которой также является один символ, повторение

которого не ограничивается (от 0 до границы, определяющейся размером выделенной оперативной памяти для хранения результата работы механизма). Метасимвол «*» определен диалектом регулярных выражений Python, однако его использование запрещено внутри паттерна в подгруппе конструкций «заглядывание назад» и «негативное заглядывание назад». В данном контексте конструкция «заглядывание» – операция в автомате регулярных выражений, где каретка автомата перемещается на H шагов вправо (индекс символ, на который указывает каретка увеличивается на H) по строке или влево на H шагов (индекс символ, на который указывает каретка уменьшается на H). Перемещение каретки вправо называется «заглядыванием вперед», а перемещение каретки влево – «заглядыванием назад».

Данные конструкции используются для поиска соответствия («положительное заглядывание») или несоответствия («негативное заглядывание») левой или правой строки с некоторым паттерном. Ограничение по использованию той или иной конструкции внутри другой конструкции связано с внутренними ограничениями автомата регулярных выражений, используемой в модуле re. Проверка на подобное ограничение должна проходить на этапе оптимизации результата лексического анализа.

На следующем этапе проверенные токены переходят в состояние обработки синтаксическим анализатором, изображенным под номером 3 на рисунке 4. На этом этапе идет формирование абстрактного синтаксического дерева (АСД) [25]. Каждый узел такого дерева, по сути, представляет конструкцию, похожую на форму Бэкуса-Наура [26]. Например, для последовательности токенов регулярного выражения «*ab?c.|d*» получившееся дерево представлено на рисунке 5.

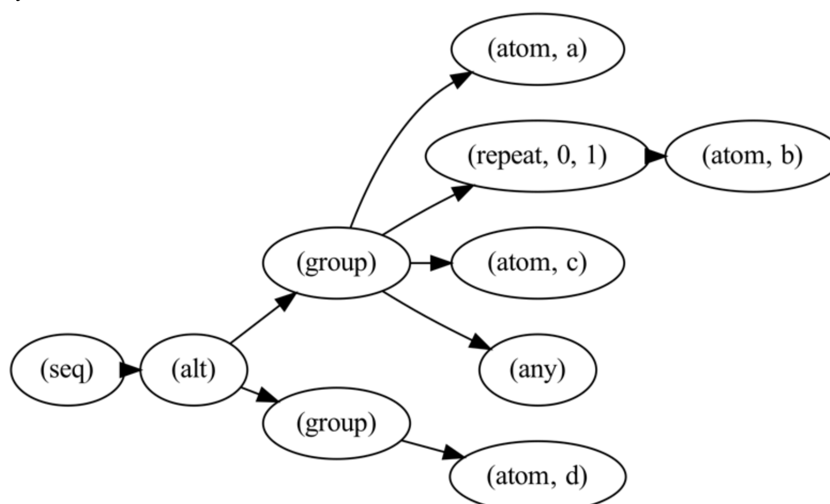

Рисунок 5 – Пример АСД для регулярного выражения

На рисунке 5 можно заметить, что в процессе построения дерева произошел ряд изменений. Например, квантификатор «?», который при лексическом анализе был преобразован в токен *«1.0_or_1»,* в процессе построения АСД преобразовался в форму *(repeat, 0, 1)*. Подобной формой описываются все квантификаторы для того, чтобы не перегружать лишними смыслами дерево. Вершина *(alt)* представленного дерева содержит в себе две группы, являющиеся альтернативой друг для друга. Эти группы будут раскрыты при генерации регулярного выражения на целевом синтаксисе, то есть будут убраны родительские скобки.

АСД в предлагаемом трансляторе является промежуточным представлением регулярного выражения, которое может быть переведено на любой диалект, описанный правилами непосредственно внутри исходного кода транслятора. После получения АСД наступает этап оптимизации промежуточного состояния.

Под оптимизацией промежуточных представлений понимается получение регулярного выражения, которое по некоторым метрикам качества будет показывать результаты лучше, чем исходное регулярное выражение. В настоящем исследовании предлагается использовать такие метрики как, метрика качества поиска паттернов регулярного выражения, метрика читаемости регулярного выражения и метрика скорости работы регулярного выражения.

Для расчета количественных характеристик регулярного выражения необходимо передать на вход функциям расчета метрик некоторое число строк из множества сгенерированных строк $\{S\}$.

Пусть $S^{`}$ ($S^{`} \subset S$) – множество проверочных строк для расчета количественных характеристик регулярного выражения. Необходимость использования двух множеств S и $S^{`}$ заключается в том, что множество строк S генерируется автоматически по структуре регулярного выражения, при этом на всех строках $s \in S$ автомат

регулярного выражения должен найти соответствие, в противном случае процесс трансляции может считаться некорректным.

Множество S является однородным и не вариативным, так как строки генерируются для всех возможных вариантов строк, на которых регулярное выражение сможет найти соответствие.

Множество $S^{`}$ выбирается случайным образом из множества S, так как при расчете количественных характеристик регулярного выражения при обходе автомата регулярного выражения не требуется проверка на минимальные изменения строк и обход всех входных строк множества S.

Метрика качества поиска паттернов регулярного выражения reg может быть определена как:

$$Acc(reg, S^{`}) = \frac{1}{G^{`}} \sum_{g=1}^{G^{`}} f(reg, S_g^{`}), \quad (1)$$

где $S_g^{`}$ – $g$-я строка из множества проверочных строк $S^{`}$ ($g = \overline{1, G^{`}}$); $G^{`}$ – число проверочных строк, на которых проверяется корректность работы регулярного выражения; f – функция прохода строки $S_g^{`}$ через автомат регулярного выражения $reg$, возвращающая 1, если регулярное выражение $reg$ полностью соответствует некоторому участку в строке $S_g^{`}$ и 0 – в противном случае.

Метрику читаемости предлагается определить через нативное выражение для подсчета сложности понимания регулярного выражения. Для этого все конструкции регулярного выражения в промежуточном представлении должны быть оценены на сложность. В таблице 2 представлены оценки сложности для наиболее встречающихся конструкций [24].

Метрика читаемости показывает сложность интерпретации регулярного выражения в некоторый конечный автомат. В таблице 2 представлены оценки сложности некоторых конструкций регулярного выражения. Эти оценки сформированы на основе исследования [24], в котором авторы рассматривают сложность тех или иных конструкций для компиляции и используют два класса – *“hard”* и *“easy”*. В настоящем исследований предлагается использовать эмпирически подобранные целочисленные эквиваленты каждой конструкции в АСД.

Таблица 2 – Сложность конструкций регулярного выражения

Номер типа конструкции	**Конструкция**	**Сложность**
1	(atom)	1
2	(any)	1
3	(escape)	1
4	(alt)	2
5	(group)	3
6	(range)	3
7	(repeat)	3

Оценки сложности, приведенные в таблице 2, подобраны эмпирическим путем, исходя из сложности интерпретации регулярных выражений.

Метрика читаемости для регулярного выражения reg с промежуточным представлением ast^{reg} может быть определена как:

$$Read(ast^{reg}) = levels(ast^{reg}) \times \sum_{l=1}^{L} diff(ast_l^{reg}), \quad (2)$$

где L – число конструкций в промежуточном представлении ast^{reg}; $diff$ – функция, возвращающая значение сложности конкретного типа конструкции ast_l^{reg}, извлекаемое из таблицы 2, где $1 \leq l \leq L$; $levels$ – функция, возвращающая число уровней вложенности.

Чтобы определить уровень вложенности дерева, необходимо осуществить прямой обход дерева. В случае, если у вершины имеется включенное или вложенное дерево, необходимо увеличить счетчик уровней вложенности на 1. В случае, если разбор поддерева завершен и требуется перейти вновь на нулевую или корневую вершину, счётчик вложенности необходимо увеличить только тогда, когда проход по поддеревьям будет больше текущего значения счетчика уровней вложенности.

Например, для регулярного выражения «*ab?c.|d*» значение метрики читаемости будет равно:

$$Read(ast^{reg}) = 3 \times (1 + 1 + 3 + 1 + 1 + 2 + 1) = 30. \quad (3)$$

Метрику скорости работы регулярного выражения reg предлагается определить как:

$$Perf(reg, S^{`}) = \frac{1}{N^{iter}} \sum_{n}^{N^{iter}} \frac{1}{G^{`}} \sum_{g=1}^{G^{`}} t(f(reg, S_g^{`})), \quad (4)$$

где $S_g^{`}$ – $g$-я строка из множества проверочных строк $S^{`}$ $(g = \overline{1, G^{`}})$; $G^{`}$ – число проверочных строк, на которых проверяется корректность работы регулярного выражения; f – функция прохода строки $S_g^{`}$ через автомат регулярного выражения $reg$, возвращающая 1, если регулярное выражение $reg$ полностью соответствует некоторому участку в строке $S_g^{`}$ и 0 – в противном случае, t – функция замера времени переданной операции; N^{iter} – число заданных итераций.

Константа N^{iter} задается при проектировании программы. В приведенных примерах и представленных расчетах $N^{iter} = 500$. Данная константа необходима для точного замера времени работы регулярного выражения, так как при различных запусках регулярного выражения затраченное время может отличаться.

Функция замера времени $t(\partial)$ осуществляет замер времени работы для некоторой операции. В настоящем исследовании функция $t(\partial)$ реализует замер времени стандартной библиотекой time языка Python. При этом время измеряется в миллисекундах с шестью знаками после запятой.

На этапе генерации регулярного выражения целевого диалекта применяются те же правила, которые ранее были представлены в этой работе для лексического анализа (некоторый аналог контекстно-свободных грамматик). Однако следует заметить, что на этом этапе выходным значением может быть не только желаемое регулярное выражение.

Существует множество комплексных задач, решаемых сложными (с точки зрения конструкций) регулярными выражениями, которые могут содержать ошибку, пропущенную транслятором [например, если в изначальном регулярном выражении была допущена синтаксическая ошибка (ошибка семантического характера, не влияющая на процесс компиляции)]. В подобных случаях может быть полезен инструмент для генерации набора строк, которые точно будут соответствовать паттерну, описываемому регулярным выражением. Такие инструменты существуют (например, один из них рассмотрен в [20]) и применяются для отладки написанного паттерна. Такой функционал можно реализовать с помощью получения промежуточного представления выражения, и генерации из него строк. Например, для регулярного выражения «ab?c.|d» синтезированные строки будут выглядеть следующим образом – «aca», «d», «abca», и т.д. Подобный функционал будет полезен на этапе оптимизации промежуточных представлений.

Следует отметить, что, зная промежуточное представление регулярного выражения, можно генерировать другие выражения с помощью эквивалентных замен.

Эквивалентная замена представляет собой процесс изменения конструкции промежуточного представления, при которой не изменяется множество соответствующих выражению строк. Например, для конструкции

$$(group, (atom, x), (alt, (group, (atom, y)), (group, z))),$$

которая соответствует выражению $x(y|z)$, можно произвести замену на конструкцию

$$(alt, (group, (atom, x), (atom, y)), (group, (atom, x), (atom, z))),$$

которая соответствует регулярному выражению $(xy|xz)$.

Реализации механизмов регулярных выражений

В качестве инструмента оптимизации промежуточных представлений предлагается использовать популяционные алгоритмы. Популяционные алгоритмы – одни из самых широко используемых алгоритмов решения задачи глобальной оптимизации. Предлагается рассмотреть два таких алгоритма – алгоритм дифференциальной эволюции (ДЭ, DE) [27] и алгоритм роя частиц [27] (РЧ, Particle Swarm Optimization, PSO).

ДЭ – алгоритм, основанный на случайной инициализации и популяционном поиске [28]. ДЭ вдохновлена природной эволюцией и включает в себя процессы мутации, селекции и комбинации индивида. В базовом варианте алгоритма ДЭ для каждого индивида популяции создаётся новый мутантный вектор. Затем мутантный вектор комбинируют с целевым вектором для создания пробного вектора. Наконец, применяется фаза селекции для выбора индивидов следующей популяции. Итерации алгоритма продолжаются, пока не будет достигнут критерий остановки. Особенностью алгоритма является исходной целевой функции. Существует несколько видов алгоритма ДЭ, которые классифицируются по типу мутации и селекции [29]. В классическом варианте ДЭ применяется стратегия $rand/1$, мутационный вектор формируется из трех случайно выбранных векторов из всей популяции.

Алгоритм оптимизации РЧ является алгоритмом оптимизации, вдохновленным поведением стай птиц или роя частиц в пространстве поиска [27]. Суть алгоритма сводится к непрерывному перемещению частиц (индивидов) в возможном пространстве решений, при этом каждому текущему состоянию частицы присваиваются два значения: скорость перемещения и координата. РЧ представляет собой алгоритм с глобальной координацией, так как каждая частица учитывает как локальную, так и глобальную информацию при принятии решений о

движении. Каждая частица «помнит» свое текущее положение в пространстве поиска и скорость движения. Положение частицы соответствует потенциальному решению, а скорость определяет направление и скорость ее движения.

Алгоритмы, предлагаемые к использованию для оптимизации промежуточных состояний регулярных выражений, позволят одновременно проводить как исследование пространства возможных решений, так и сосредотачиваться на областях, где находится лучшее решение, учитывая многообразие синтаксисов регулярных языков.

Для реализации алгоритма оптимизации регулярных выражений необходимо описать векторное представление выражений. Для этого предлагается использовать списки инцидентности [30]. В данном случае под инцидентностью понимается бинарное отношение двух вершин в АСД, как в ориентированном графе [26]. Значения, сопоставляемые вершинам, будут определяться номерами типов вершин, представленных в таблице 3. Список инцидентности будет формироваться при рекурсивном обходе [31] АСД. При этом он будет представляться в виде упорядоченного списка кортежей, определяемых парами значений для двух вершин.

Вершины с их идентификационными номерами целесообразно хранить в таблице T (таблица 3). В таблице 3 описаны все конструкции регулярных выражений и их типы вершин, используемые для АСД. Они соответствуют выбранным токенам в таблице 1.

Конструкции <seq> и <altgroup> не рассматривались при описании АСД, так как они не несут в себе семантической нагрузки, а используются только для удобства разбора дерева.

Эти конструкции регулярных выражений (таблица 3) выбраны из-за их распространенности [24], и схожего написания в разных диалектах.

Таблица 3 – Типы вершины АСД

Номер типа конструкции регулярного выражения	**Тип вершины**
0	seq (корень дерева)
1	atom (лепестки дерева)
2	any (лепестки дерева)
3	repeat
4	alt
5	altgroup
6	group
7	range
8	escape

Для однозначной интерпретации списка инцидентности в регулярное выражение к типам вершин (таблица 3) при обработке регулярного выражения $\boldsymbol{reg}$ в таблицу T добавляются значения лепестков (вершины $\boldsymbol{atom, any}$) АСД для $\boldsymbol{reg}$.

Например, для регулярного выражения $\boldsymbol{abc(c|a)b}$ список инцидентности будет выглядеть следующим образом: [(0, 1), (1, 9), (0, 1), (1, 10), (0, 1), (1, 11), (0, 6), (6, 4), (4, 5), (5, 1), (1, 11), (4, 5), (5, 1), (1, 9), (0, 1), (1, 10), (0, 1), (1, 12)]. В таком формате и будут представляться индивиды популяции.

В общем случае задачу, которую решают популяционные алгоритмы, можно представить в виде:

$$x^* = \underset{x \in \mathbb{R}^z}{arg\ min}\big(Q(x)\big), \tag{5}$$

где $\boldsymbol{x}$ – вектор, принадлежащий пространству поиска $\mathbb{R}^{\boldsymbol{z}}$ и описывающий индивида; $\boldsymbol{z}$ – размерность вектора $\boldsymbol{x}$; $\mathbb{R}^{\boldsymbol{z}}$ – область допустимых значений вектора; $\boldsymbol{Q(x)}$ – целевая функция, определяющая значение показателя качества индивида; $\boldsymbol{x}^*$ – вектор оптимального решения.

В контексте решения задачи оптимизации список инцидентности, представляющий собой список кортежей, преобразуется в простой список $\boldsymbol{x}$ с учетом порядка конструкций в исходном списке инцидентности. Длина списка $\boldsymbol{x}$ равна числу U, которое определятся как длина списка инцидентности исходного регулярного выражения, умноженная на некоторый коэффициент. В настоящем исследовании этот коэффициент равен 1, то есть максимально допустимое число пар инцидентности будет определятся по исходному списку инцидентности. Например, для регулярного выражения $\boldsymbol{abc(c|a)b}$ длина списка пар инцидентности равна 18, следовательно, популяционные алгоритмы при коэффициенте 1 могут генерировать список (индивид) только с удвоенной длиной $\boldsymbol{U = 1 \times 18 \times 2 = 36}$.

Целевую функцию $\boldsymbol{Q(x)}$ популяционного алгоритма оптимизации можно представить в виде:

$$Q(x) = \alpha \times \left(Perf\left(greg(x), S^{`}\right)\right), \quad (6)$$

где $\boldsymbol{x}$ – список, описывающий индивида; элементы списка представляют собой упорядоченные пары вида $(\boldsymbol{Node_1}, \boldsymbol{Node_2})$, где $\boldsymbol{Node_1}$ и $\boldsymbol{Node_2}$ – номера инцидентных вершин в таблице *T*; длина списка $\boldsymbol{x}$ определяется максимально удвоенной допустимой длиной списка инцидентности $\boldsymbol{U}$ регулярного выражения; $\boldsymbol{Perf}$ – функция расчета скорости работы регулярного выражения (4); $\boldsymbol{greg}$ – функция, возвращающая регулярное выражение по входному списку инцидентности *x*; $\boldsymbol{S}^{`}$ – множество проверочных строк; $\boldsymbol{\alpha}$ – коэффициент учета точности (7) при подсчете целевой функции.

Коэффициент α можно вычислен как:

$$\alpha = \left(2 - Acc\left(greg(x)\,,\ S^{`}\right)\right), \quad (7)$$

где $\boldsymbol{Acc}$ – метрика качества регулярного выражения (1); $\boldsymbol{greg}$ – функция, возвращающая регулярное выражение по входному списку инцидентности *x*; $\boldsymbol{S}^{`}$ – множество проверочных строк.

Коэффициент α принимает значение 1 в случае, когда автомат регулярного выражения определил соответствие во всех строках множества $S^{`}$. Коэффициент $\alpha$ принимает значение из диапазона (1,2], в случае пропуска соответствия в проверочных строках $S^{`}$.

Анализ целевой функции (6) позволяет заметить, что, если регулярное выражение $\boldsymbol{reg}^*$, сгенерированное по индивиду $\boldsymbol{x}$, не показывает итоговой точности $\boldsymbol{Acc}(\boldsymbol{reg}^*, \boldsymbol{S}) = \boldsymbol{1}$, то индивид $\boldsymbol{x}$ всё равно рассматривается как потенциальное оптимальное решение задачи (5).

Алгоритм расчёта значения целевой функции индивида может быть описан следующей последовательностью шагов.

1. Проверка индивида на запрещенные пары $(Node_1, Node_2)$.
2. Представление списка инцидентности в виде регулярного выражения.
3. Проверка возможности компиляции регулярного выражения.
4. Расчёт значения метрики (4) для полученного регулярного выражения.
5. Расчёт значения целевой метрики (6) для полученного регулярного выражения.

В случае, если на некотором шаге, в частности, на шагах 1 и 3, происходит ошибка, значение целевой функции (6) принимается равным значению целевой функции (6) исходного регулярного выражения, которое необходимо оптимизировать с помощью популяционных алгоритмов.

На шаге 1 возможна ошибка, вызванная использованием недопустимого значения идентификатора вершины, на шаге 3 возможна ошибка компиляции из-за запрещенной конструкции в автомате регулярных выражений.

В настоящем исследовании среди запрещенных пар есть только пара (0, 0), обозначающая вершину *seq* и вхождение в эту вершину ещё одной вершины *seq*. Так как вершина *seq* является корневой и может быть только одна в АСД некоторого РВ, конструкция считается запрещенной. Все остальные конструкции считаются допустимыми.

Реализации механизмов регулярных выражений

Апробация предлагаемого алгоритма оптимизации регулярных выражений была выполнена с применением языка программирования Python в среде разработки PyCharm. В ходе экспериментов был использован компьютер со следующими характеристиками: MacBook Air 13 2020 A2337 (процессор: Apple M1 3.2 ГГц 5 нм, ARMv8.5-A, 3.2 ГГц, 8 ядер; оперативная память: 8 Гб; 64-разрядная операционная система).

Для оценки эффективности популяционных алгоритмов было сформировано 10 регулярных выражений, представленных в таблице 4.

РВ из таблицы 4 подобраны таким образом, чтобы рассмотреть во время экспериментальных исследований выражения с разными уровнями вложенности и разными типами конструкций. При этом в таблице 4 показано число сгенерированных строк *S*, то есть число строк, на которых была выполнена проверка решений популяционных алгоритмов. В настоящем исследовании предполагается, что $S^{`} = S$.

Таблица 4 – РВ для эксперимента

Номер регулярного выражения	Регулярное выражение	Количество сгенерированных строк *S*	Используемые конструкции (таблица 3)
1	a	1	atom
2	abcde	1	atom
3	ab?c.	182	atom, any, repeat
4	ab*c.	1820	atom, any, repeat

5	ab+c.	1820	atom, any, repeat
6	ab?c.|d	183	atom, any, repeat, alt
7	abc(c|a)b	2	atom, any, repeat, alt, group
8	(a|b)c	2	atom, group, alt
9	(a|b{1,2})c	3	atom, alt, repeat, group
10	ab?c.|[0-9]\\[	192	atom, any, repeat, alt, group, range, escape

В качестве программной реализации рассматриваемых алгоритмов была использована библиотека mealpy [28], содержащая коллекцию различных популяционных алгоритмов оптимизации.

При проведении эксперимента по оптимизации списка инцидентности с использованием целевой функции (6) введенного регулярного выражения (таблица 4) эмпирически были подобраны следующие параметры для запуска популяционных алгоритмов.

1. Для алгоритма ДЭ была выбрана адаптивная реализация [27]. Параметры запуска: схема «rand/1/bin»; границы изменения переменных в пространстве поиска, определяющие значения в матрице переходов, – от 0 до максимального номера вершины в списке инцидентности входного регулярного выражения; максимальное число поколений – 100; число индивидов в популяции – 100. Значения остальных параметров заданы по умолчанию.
2. Для алгоритма РЧ была выбрана реализация алгоритма с адаптивной оптимизацией инерционного веса частиц с учетом инерционного веса (AIW-PSO) [32]. Параметры запуска: границы изменения переменных в пространстве поиска, определяющие значения в матрице переходов, – от 0 до максимального номера вершины в списке инцидентности входного регулярного выражения; максимальное число поколений – 100, число индивидов в популяции – 100, константа α – 0.4. Значения остальных параметров заданы по умолчанию (локальный коэффициент – 2.05).

Для демонстрации работы алгоритма оптимизации были построены графики изменения целевой функции для регулярного выражения $\boldsymbol{ab}?\boldsymbol{c}.|[\mathbf{0}-\mathbf{9}]\backslash[$ для алгоритмов ДЭ и РЧ. Запуск алгоритма по оптимизации проводился 10 раз. На рисунке 6 приведены графические зависимости для лучших запусков каждого из двух популяционных алгоритмов в смысле получения минимального значения целевой функции у лучшего индивида, а также – коробчатые диаграммы, реализующие визуализацию результатов расчёта целевой функции для лучшего индивида при 10 запусках соответствующих популяционных алгоритмов.

На рисунке 6 при построении графиков изменения целевой функции для лучших запусков по вертикальным осям показаны значения целевой функции, по горизонтальным осям – номера итераций.

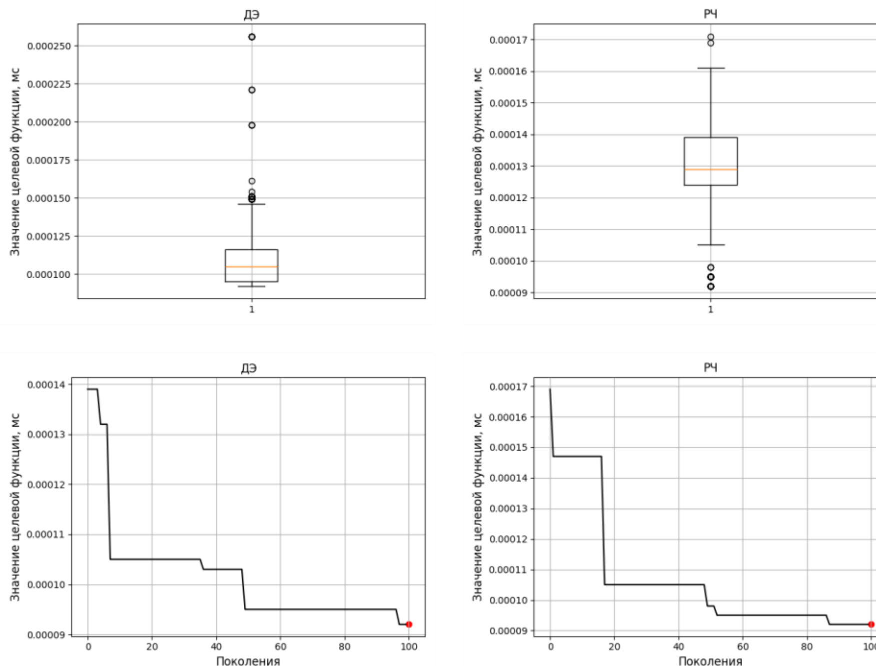

Рисунок 6 – Графики изменения значения ЦФ (6) и коробчатые диаграммы

В таблице 5 приведены значения целевой функции (6) для исходных регулярных выражений, а также средние значения целевой функции (6) и стандартные отклонения значений целевой функции (6) по результатам

оптимизации на основе 10 запусков алгоритмов ДЭ и РЧ. Жирным шрифтом в таблице 5 выделены лучшие (то есть меньшие) средние значения целевой функции (6) и меньшие значения стандартного отклонения выборки.

Из таблицы 5 видно, что в 4 случаях из 5 при применении алгоритма ДЭ были получены меньшие значения целевой функции (6). При этом во всех случаях при применении алгоритма ДЭ были получены большие значения стандартного отклонения. Таким образом, можно сделать вывод о том, что алгоритм ДЭ является более эффективным в смысле минимизации целевой функции (6), однако значения целевой функции (6) по ДЭ имеют больший разброс, чем у алгоритма РЧ.

Таблица 5 – Значение ЦФ (6) для исходных и оптимизированных РВ

Номер регулярного выражения	**Значение целевой функции (6) для исходного регулярного выражения**	**Среднее значение целевой функции (6) по результатам оптимизации на основе алгоритма ДЭ**	**Среднее значение целевой функции (6) по результатам оптимизации на основе алгоритма РЧ**	**Стандартное отклонение значений целевой функции (6) по результатам оптимизации на основе алгоритма ДЭ**	**Стандартное отклонение значений целевой функции (6) по результатам оптимизации на основе алгоритма РЧ**
1	0.0001196	0.000113	**0.000110**	0.00015	**0.00003**
2	0.0001318	**0.000098**	0.000107	0.000011	**0.000010**
3	0.0001221	**0.000098**	0.000114	0.000014	**0.000012**
4	0.0002442	**0.000097**	0.000115	0.000013	**0.000011**
5	0.0002392	**0.00098**	0.000113	0.000013	**0.000010**
6	0.0001611	**0.000110**	0.000128	0.000026	**0.000015**
7	0.0001099	**0.000126**	0.000139	0.000027	**0.000025**
8	0.0001001	**0.000114**	0.000131	0.000018	**0.000011**
9	0.0001221	**0.000122**	0.000137	0.000008	**0.000005**
10	0.0001733	**0.000109**	0.000129	0.000021	**0.000013**

В таблице 6 представлены результаты расчета метрик (1), (2) и (4) для исходного регулярного выражения $\boldsymbol{ab}?\boldsymbol{c}.|[\mathbf{0}-\mathbf{9}]\backslash[$.

В таблицах 7 и 8 для регулярного выражения $\boldsymbol{ab}?\boldsymbol{c}.|[\mathbf{0}-\mathbf{9}]\backslash[$ представлены все РВ, полученные после 10 запусков алгоритма оптимизации на основе алгоритмов ДЭ и РЧ соответственно. Кроме того, в таблицах представлены значения метрик (1), (2) и (4) для представленных регулярных выражений.

Таблица 6 – Расчет метрик (1), (2) и (4) для исходного РВ

Исходное регулярное выражение	**Значение метрики точности (1) для исходного регулярного выражения**	**Значение метрики читаемости (2) для исходного регулярного выражения**	**Значение метрики скорости работы (4) для исходного регулярного выражения, мс**
ab?c.|[0-9]\[	1.0	156	0.00021044

Таблица 7 – Оптимизированные РВ по ДЭ

Запуск алгоритма ДЭ	**Оптимизированное регулярное выражения по ДЭ**	**Значение метрики точности (1) для оптимизированного регулярного выражения после ДЭ**	**Значение метрики читаемости (2) для оптимизированного регулярного выражения после ДЭ**	**Значение метрики скорости работы (4) для оптимизированного регулярного выражения после ДЭ, мс**
1	a(){0,1}.	0.94791	40	0.00010058
2	a([0-9]){0,1}	0.94791	50	0.00010206
3	a(..{0,1}.)	0.94791	70	0.00010058
4	a..()	0.94791	30	0.00010206
5	a.()	0.94791	20	0.00010450
6	a()	0.94791	12	0.00010206
7	a.()	0.94791	20	0.00010450
8	a([0-9]){0,1}	0.94791	50	0.00010206
9	a(.).	0.94791	30	0.00010254
10	a()[0-9]{0,1}	0.94791	50	0.00010206

Проанализировав результаты работы алгоритмов, представленные в таблицах 7 и 8, можно заметить, что во всех 10 прогонах с помощью алгоритма ДЭ были получены наименьшие значения метрики скорости работы (4). При этом повторяемость оптимизированных регулярных выражений характерна для результатов работы алгоритмы РЧ (один случай встретился 6 раз, другой случай встретился 2 раза).

Таблица 8 – Оптимизированные РВ по РЧ

<table>
<tr><th>Запуск алгоритма РЧ</th><th>Оптимизированное регулярное выражения по РЧ</th><th>Значение метрики точности (1) для оптимизированного регулярного выражения после РЧ</th><th>Значение метрики читаемости (2) для оптимизированного регулярного выражения после РЧ</th><th>Значение метрики скорости работы (4) для оптимизированного регулярного выражения после РЧ, мс</th></tr>
<tr><td>1</td><td>[0-9]b{0,1}()\.|</td><td>1.0</td><td>143</td><td>0.00016064</td></tr>
<tr><td>2</td><td>a..([0-9]{0,1})</td><td>0.94791</td><td>84</td><td>0.00010206</td></tr>
<tr><td>3</td><td>[0-9]{0,1}a.()</td><td>0.94791</td><td>66</td><td>0.00011816</td></tr>
<tr><td>4</td><td>[0-9]b{0,1}()\.|</td><td>1.0</td><td>143</td><td>0.00016064</td></tr>
<tr><td>5</td><td>a..([0-9]{0,1})</td><td>0.94791</td><td>84</td><td>0.00010206</td></tr>
<tr><td>6</td><td>[0-9]b{0,1}()\.|</td><td>1.0</td><td>143</td><td>0.00016064</td></tr>
<tr><td>7</td><td>(.).</td><td>1.0</td><td>20</td><td>0.00013964</td></tr>
<tr><td>8</td><td>[0-9]b{0,1}()\.|</td><td>1.0</td><td>143</td><td>0.00016064</td></tr>
<tr><td>9</td><td>[0-9]b{0,1}()\.|</td><td>1.0</td><td>143</td><td>0.00016064</td></tr>
<tr><td>10</td><td>[0-9]b{0,1}()\.|</td><td>1.0</td><td>143</td><td>0.00016064</td></tr>
</table>

Анализ структуры оптимизированных выражений и их сложности показывает, что в большинстве случаев более легкую по сложности (2) структуру предлагается алгоритм ДЭ. В целом можно сказать, что во всех 10 прогонах алгоритмы ДЭ и РЧ смогли предложить РВ, для которых значения метрик (2) и (4) меньше, чем у исходного регулярного выражения (таблица 6). При этом меньшие значения метрик (2) и (4) удалось получить при снижении значения метрики точности (1) не более, чем на 0,05209.

Заключение

РВ представляют собой мощный инструмент для поиска слов в текстовом корпусе по определенному образу. Однако, различия в синтаксисе и математической базе могут приводить к проблемам с обратной совместимостью и оптимизацией работы выражений. Для решения этих проблем был предложен универсальный транслятор, способный транслировать один синтаксис в другой с оптимизацией промежуточных представлений в виде списка инцидентности АСД входного РВ. Подобный формат представления РВ может быть использован в качестве универсального для хранения выражений на разных диалектах.

Кроме того, предложен алгоритм по оптимизации регулярных выражений в виде списка инцидентности на основе популяционных алгоритмов, а также проведены экспериментальные исследования, демонстрирующие работоспособность предлагаемого алгоритма на примере двух алгоритмов – ДЭ и РЧ.

Целью дальнейших исследований является анализ влияния размера выборки проверочных строк на результаты оптимизации исходных РВ, кроме того, планируется исследовать аспекты формирования правил. Кроме того, в дальнейших работах планируется исследование других популяционных алгоритмов оптимизации, а также добавление других диалектов, описанных в теоретической части (POSIX, Perl, EcmaScript, PCRE), и других конструкций регулярных выражений.

Список литературы

1. Демидова Л.А., Морошкин Н.А. Аспекты разработки архитектуры вопросно-ответной системы для обработки больших данных на основе нейросетевого моделирования, Вестник Рязанского государственного радиотехнического университета, 2023, № 86, 55–69

2. Ноникашвили Г.Л. Обзор рационального применения искусственного интеллекта для повышения эффективности управления ИТ проектов в банковской отрасли, ИТ-Стандарт, 2024, №1, С. 55-62

3. M. Erwig, R. Gopinath. Explanations for Regular Expressions, 15th international conference on Fundamental Approaches to Software Engineering, 2012, 394-408

4. H. Hosoya, J. Vouillon, B.C. Pierce, Regular Expression Types for XML, The International Conf. on Functional Programming (ICFP), 2000, 11–22

5. B. W. Watson, A Taxonomy of Finite Automata Minimization Algorithms, Computing science notes, 1993, №

9343, 18-49

6. L. Colussi, Fastest Pattern Matching in Strings, Journal of Algorithms, 1994, № 16, 163-189

7. Q. Chen, X. Wang, X. Ye, G. D. Dillig, Multi-modal Synthesis of Regular Expressions, The 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2020

8. P. Hazel, PCRE2 - Perl-compatible regular expressions (revised API), University of Cambridge, 2020

9. re2: Regular expression library [Online]. — URL: https://github.com/google/re2 (дата обращения: 15.06.2024)

10. I.N. Vdovychenko, System analysis of ClickHouse - column DBMS for online processing of analytical requests, Jornal of Kryvyi Rih National University, 2020, 153-158

11. re: CPython regular expression library [Online]. — URL: https://github.com/python/cpython/tree/3.12/Lib/re/ (дата обращения: 15.06.2024)

12. PolyAnalyst: Функциональные возможности и системные требования [Online]. — URL: https://www.megaputer.ru/wp-content/uploads/-возможности-и-системные-требования-polyanalyst.pdf (дата обращения: 16.06.2024)

13. POSIX Regular Expression Syntax and Examples [Online]. — URL: https://www.ibm.com/docs/en/watson-explorer/11.0.1?topic=queries-posix-regular-expression-syntax-examples (дата обращения: 16.06.2024)

14. perlre: Perl regular expressions [Online]. — URL: https://perldoc.perl.org/perlre (дата обращения: 16.06.2024)

15. Modified ECMAScript regular expression grammar [Online]. — URL: https://en.cppreference.com/w/cpp/regex/ecmascript (дата обращения: 16.06.2024)

16. Жаркова, Г.А. Детерменированные и недетерминорованные конечные автоматы в современном программировании / Г. А. Жаркова, К. Н. Лаптев // Ученые записки УлГУ. Сер. Математика и информационные технологии. УлГУ. Электрон. журн. 2018, № 2, С. 24-27.

17. A. Cataltepe, V. Kosoy, Time complexity for deterministic string machines, 2024

18. A. Currin, K. Korovin, M. Ababi, K. Roper, D. B. Kell, P. J. Day, R. D. King, Computing exponentially faster: Implementing a nondeterministic universal Turing machine using DNA, 2016

19. P. Bille, M. Thorup , Faster Regular Expression Matching, Conference Automata, Languages and Programming, 2019, 171-182

20. F. Yu, Z. Chen, Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection, Conference: Architecture for Networking and Communications systems (ANCS 2006), 2006

21. Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, Heqing Zhu, Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs, the 16th USENIX Symposium on Networked SystemsDesign and Implementation (NSDI 2019), 2019

22. G. Xing, Minimized Thompson NFA, International Journal of Computer Mathematics, № 81, 2004, 1097-1106

23. A.V. Gorchakov, L.A. Demidova, P.N. Sovietov, Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task, Future Internet, 2023, № 9, C. 314

24. Arturs Backurs, Piotr Indyk, Which Regular Expression Patterns are Hard to Match?, arXiv, 2016

25. Конаков П.О Показатели качества линии формирования данных на основе статического анализа SQL-запросов кода, ИТ-Стандарт, 2024, №2, 56-64

26. Павлов К.С., Советов П.Н. Библиотека llvm2py для анализа промежуточного представления LLVM и её применение в оценке степени распараллеливания линейных участков кода, ИТ-Стандарт, 2024, №3, 87-96

27. A.K.Qin, P.N. Suganthan, Self-adaptive differential evolution algorithm for numerical optimization., 2005 IEEE congress on evolutionary computation, 2005, № 2, 1785-1791

28. Л.А. Демидова, А.В. Горчаков, Применение биоинсперированных алгоритмов глобальной оптимизации для повышения точности прогнозов компактных машин экстремального обучения, Российский технологический журнал, 2022, № 2, 59-74

29. L.A. Demidova, A.V. Gorchakov, Research and Study of the Hybrid Algorithms Based on the Collective Behavior of Fish Schools and Classical Optimization Methods, Algorithms, 2020, № 4, 85-102

30. М. О. Асанов, В. А. Баранский, В. В. Расин: Дискретная математика: графы, матроиды, алгоритмы — НИЦ РХД, 2001, 288 с., ISBN 5-93972-076-5

31. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 3-е изд. — М.: «Вильямс», 2013, С. 440., ISBN 978-5-8459-1794-2

32. S. Kessentini, D. Barchiesi, Particle Swarm Optimization with Adaptive Inertia Weight, International Journal of Machine Learning and Computing, 2015, № 5, 368-373

References

1. Demidova L.A., Moroshkin N.A. Aspekty razrabotki arhitektury voprosno-otvetnoj sistemy dlja obrabotki bol'shih dannyh na osnove nejrosetevogo modelirovanija, Vestnik Rjazanskogo gosudarstvennogo radiotehnicheskogo universiteta, 2023, № 86, 55–69
2. Nonikashvili G.L. Obzor racional'nogo primenenija iskusstvennogo intellekta dlja povyshenija jeffektivnosti upravlenija IT proektov v bankovskoj otrasli, IT-Standart, 2024, №1, 55-62
3. M. Erwig, R. Gopinath. Explanations for Regular Expressions, 15th international conference on Fundamental Approaches to Software Engineering, 2012, 394-408
4. H. Hosoya, J. Vouillon, B.C. Pierce, Regular Expression Types for XML, The International Conf. on Functional Programming (ICFP), 2000, 11–22
5. B. W. Watson, A Taxonomy of Finite Automata Minimization Algorithms, Computing science notes, 1993, № 9343, 18-49
6. L. Colussi, Fastest Pattern Matching in Strings, Journal of Algorithms, 1994, № 16, 163-189
7. Q. Chen, X. Wang, X. Ye, G. D. Dillig, Multi-modal Synthesis of Regular Expressions, The 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2020
8. P. Hazel, PCRE2 - Perl-compatible regular expressions (revised API), University of Cambridge, 2020
9. re2: Regular expression library [Online]. — URL: https://github.com/google/re2 (data obrashhenija: 15.06.2024)
10. I.N. Vdovychenko, System analysis of ClickHouse - column DBMS for online processing of analytical requests, Jornal of Kryvyi Rih National University, 2020, 153-158
11. re: CPython regular expression library [Online]. — URL: https://github.com/python/cpython/tree/3.12/Lib/re/ (data obrashhenija: 15.06.2024)
12. PolyAnalyst: Funkcional'nye vozmozhnosti i sistemnye trebovanija [Online]. — URL: https://www.megaputer.ru/wp-content/uploads/-vozmozhnosti-i-sistemnye-trebovanija-polyanalyst.pdf (data obrashhenija: 16.06.2024)
13. POSIX Regular Expression Syntax and Examples [Online]. — URL: https://www.ibm.com/docs/en/watson-explorer/11.0.1?topic=queries-posix-regular-expression-syntax-examples (data obrashhenija: 16.06.2024)
14. perlre: Perl regular expressions [Online]. — URL: https://perldoc.perl.org/perlre (data obrashhenija: 16.06.2024)
15. Modified ECMAScript regular expression grammar [Online]. — URL: https://en.cppreference.com/w/cpp/regex/ecmascript (data obrashhenija: 16.06.2024)
16. Zharkova, G.A. Determenirovannye i nedeterminorovannye konechnye avtomaty v sovremennom programmirovanii / G. A. Zharkova, K. N. Laptev // Uchenye zapiski UlGU. Ser. Matematika i informacionnye tehnologii. UlGU. Jelektron. zhurn. 2018, № 2, S. 24-27.
17. A. Cataltepe, V. Kosoy, Time complexity for deterministic string machines, 2024
18. A. Currin, K. Korovin, M. Ababi, K. Roper, D. B. Kell, P. J. Day, R. D. King, Computing exponentially faster: Implementing a nondeterministic universal Turing machine using DNA, 2016
19. P. Bille, M. Thorup , Faster Regular Expression Matching, Conference Automata, Languages and Programming, 2019, 171-182
20. F. Yu, Z. Chen, Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection, Conference: Architecture for Networking and Communications systems (ANCS 2006), 2006
21. Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, Heqing Zhu, Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs, the 16th USENIX Symposium on Networked SystemsDesign and Implementation (NSDI 2019), 2019
22. G. Xing, Minimized Thompson NFA, International Journal of Computer Mathematics, № 81, 2004, 1097-1106
23. A.V. Gorchakov, L.A. Demidova, P.N. Sovietov, Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task, Future Internet, 2023, № 9, S. 314
24. Arturs Backurs, Piotr Indyk, Which Regular Expression Patterns are Hard to Match?, arXiv, 2016
25. Konakov P.O Pokazateli kachestva linii formirovanija dannyh na osnove staticheskogo analiza SQL-zaprosov koda, IT-Standart, 2024, №2, 56-64
26. Pavlov K.S., Sovetov P.N. Biblioteka llvm2py dlja analiza promezhutochnogo predstavlenija LLVM i ejo primenenie v ocenke stepeni rasparallelivanija linejnyh uchastkov koda, IT-Standart, 2024, №3, 87-96
27. A.K.Qin, P.N. Suganthan, Self-adaptive differential evolution algorithm for numerical optimization., 2005 IEEE congress on evolutionary computation, 2005, № 2, 1785-1791
28. L.A. Demidova, A.V. Gorchakov, Primenenie bioinsperirovannyh algoritmov global'noj optimizacii dlja povyshenija tochnosti prognozov kompaktnyh mashin jekstremal'nogo obuchenija, Rossijskij tehnologicheskij zhurnal,

2022, № 2, 59-74

29. L.A. Demidova, A.V. Gorchakov, Research and Study of the Hybrid Algorithms Based on the Collective Behavior of Fish Schools and Classical Optimization Methods, Algorithms, 2020, № 4, 85-102

30. M. O. Asanov, V. A. Baranskij, V. V. Rasin: Diskretnaja matematika: grafy, matroidy, algoritmy — NIC RHD, 2001, 288 s., ISBN 5-93972-076-5

31. Tomas H. Kormen, Charl'z I. Lejzerson, Ronal'd L. Rivest, Klifford Shtajn Algoritmy: postroenie i analiz — 3-e izd. — M.: «Vil'jams», 2013, S. 440., ISBN 978-5-8459-1794-2

32. S. Kessentini, D. Barchiesi, Particle Swarm Optimization with Adaptive Inertia Weight, International Journal of Machine Learning and Computing, 2015, № 5, 368-373