МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина»

Уральский региональный центр
образования и разработок

УДК
Код ГРНТИ

УТВЕРЖДАЮ

Директор Уральского регионального центра образования и разработок кандидат физико-математических наук, доцент

_____ Асанов М.О.
«___» ноября 2011 г.

ОТЧЁТ
О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

по проекту № ООК 6 «Алгоритмические свойства сжатых текстов»
в рамках Открытого окружного конкурса студенческих инициативных научных исследований в области информатики и информационных технологий

вид отчета: заключительный

Руководитель НИР,
младший науный сотрудник НИИ ФПМ _____ Хворост А.А.

г. Екатеринбург, 2011

Список исполнителей

Научный руководитель, _____ *Хворост А.А.*
младший науный сотрудник НИИ ФПМ подпись, дата (раздел(ы)_____)

Исполнители
студентка КН-402, института _____ *Козлова А.В.*
математики и компьютерных наук УрФУ подпись, дата (раздел(ы)_____)

студент КН-402, института _____ *Курпилянский Е.Б.*
математики и компьютерных наук УрФУ подпись, дата (раздел(ы)_____)

Нормоконтролер _____
подпись, дата

Реферат

СЖАТИЕ ДАННЫХ БЕЗ ПОТЕРЬ, КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ, ПРЯМОЛИНЕЙНЫЕ ПРОГРАММЫ, AVL-ДЕРЕВЬЯ, ДЕКАРТОВЫ ДЕРЕВЬЯ, СЕМЕЙСТВО АЛГОРИТМОВ ЛЕМПЕЛЯ-ЗИВА, LZ-ФАКТОРИЗАЦИЯ

Объектом исследования являются алгоритмы сжатия данных без потерь.

Цель работы — разработка эффективных алгоритмов сжатия данных с помощью контекстно-свободных грамматик.

В рамках научно-исследовательской работы были спроектированы два алгоритма построения контекстно-свободных грамматик. Первый алгоритм строит контекстно-свободную грамматику на основе AVL-деревьев. Второй алгоритм строит контекстно-свободную грамматику на основе декартовых деревьев. Получена теоретическая оценка сложности обоих алгоритмов. Поскольку оба алгоритма принадлежат одному классу сложности, то целесообразно сравнить их эффективность на практике.

Полученные алгоритмы впервые были реализованы на языке программирования Java SE.

В процессе работы проводились экспериментальные исследования эффективности полученных алгоритмов сжатия на текстах различной природы. В качестве входных данных использовались тексты трех видов: строки Фибоначчи, последовательности ДНК, произвольные строки над конечным алфавитом.

Основные конструктивные показатели алгоритма на основе AVL-деревьев: удовлетворительная скорость построения сжатого представления, высокая степень сжатия данных.

Основные конструктивные показатели алгоритма на основе декартовых деревьев: высокая скорость построения сжатого представления, удовлетворительная степень сжатия данных.

Степень внедрения — сравнительный анализ полученных алгоритмов сжатия и классических алгоритмов сжатия был выложен в открытый доступ по адресу `http://overclocking.usu.edu.ru`.

Оба алгоритма могут применяться для сжатия текстовых данных.

Содержание

Нормативные ссылки

Определения

В настоящем отчете о НИР применяют следующие термины с соответствующими определениями.

Строкой называется последовательность символов. В работе рассматриваются строки над конечным алфавитом Σ. *Длина* строки S равна числу символов в S и обозначается $|S|$. *Конкатенация* двух строк S_1 и S_2 обозначается $S_1 \cdot S_2$. *Позицией* в строке S называется место между соседними символами. Мы нумеруем позиции произвольной строки S слева направо, начиная с 1 и заканчивая $|S| - 1$. Дополнительно удобно ввести позиции 0, которая предшествует строке S, и $|S|$, которая следует за S. Для строки S и числа $0 \leqslant i < |S|$ обозначим через $S[i]$ символ, расположенный между позициями i и $i+1$. Например, $S[0]$ является первым символом строки S. Обозначим через $S[\ell \dots r], (0 \leqslant \ell < r \leqslant |S|)$ подстроку S, которая начинается с позиции ℓ и заканчивается в позиции r. Таким образом $S[\ell \dots r] = S[\ell] \cdot S[\ell + 1] \cdot \ldots \cdot S[r - 1]$.

Прямолинейная программа $\mathbb{S}$ – последовательность правил вывода следующего вида:

$$\mathbb{S}_1 = expr_1, \mathbb{S}_2 = expr_2, \dots, \mathbb{S}_n = expr_n,$$

где $\mathbb{S}_i$ называются *правилами*, а $expr_i$ называются *выражениями*. Выражения бывают двух видов:

- $expr_i$ является символом Σ (такие правила называются *терминальными*),
- $expr_i = \mathbb{S}_\ell \cdot \mathbb{S}_r$ $(\ell, r < i)$ (такие правила называются *нетерминальными*).

Формально, ПП это контекстно-свободная грамматика в нормальной форме Хомского, порождающая в точности одну строку над Σ^+.

Пример. Рассмотрим ПП $\mathbb{F}_7$, которая порождает 7-е слово Фибоначчи F_7= *a b a a b a b a a b a a b*:

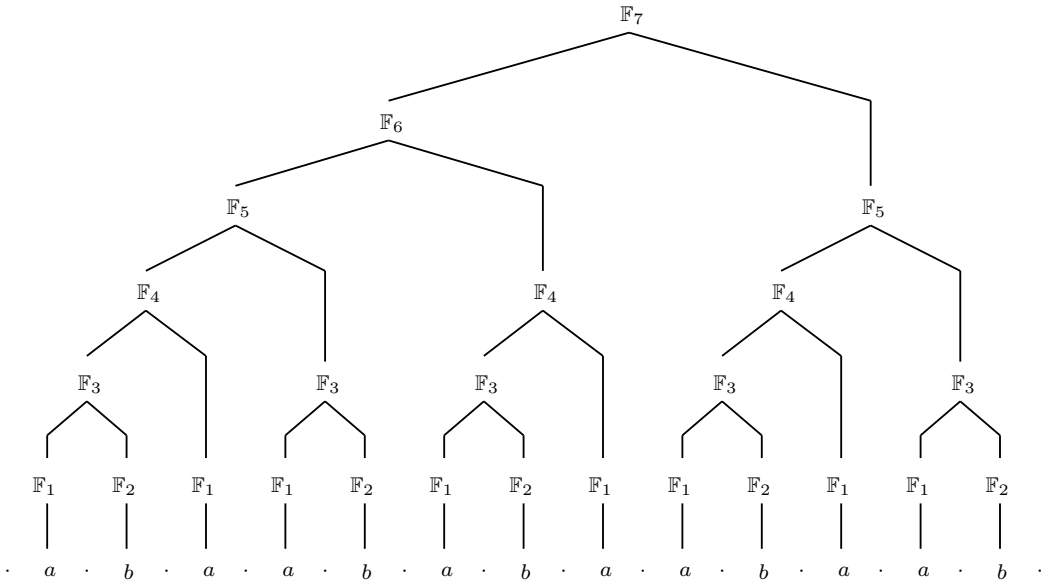
$$\mathbb{F}_1 = a, \mathbb{F}_2 = b, \mathbb{F}_3 = \mathbb{F}_1 \cdot \mathbb{F}_2, \mathbb{F}_4 = \mathbb{F}_3 \cdot \mathbb{F}_1,$$
$$\mathbb{F}_5 = \mathbb{F}_4 \cdot \mathbb{F}_3, \mathbb{F}_6 = \mathbb{F}_5 \cdot \mathbb{F}_4, \mathbb{F}_7 = \mathbb{F}_6 \cdot \mathbb{F}_5;$$

На рисунке 0.1 представлено дерево вывода ПП $\mathbb{F}_7$.

Строку S, выводимую из ПП $\mathbb{S}$, мы будем называть *текстом*. Для ПП $\mathbb{S}$, выводящей текст S, определим *дерево вывода* S через дерево вывода грамматики $\mathbb{S}$. В дереве вывода мы отождествляем терминальные узлы с их родителями,

поэтому оно является бинарным. В работе приняты следующие договоренности: все ПП обозначаются с помощью заглавных фигурных букв, например, $\mathbb{S}$. Каждое правило ПП $\mathbb{S}$ (и каждый внутренний узел ее дерева вывода) обозначается аналогичной буквой с порядковым номером, например, $\mathbb{S}_i$. *Размер* ПП $\mathbb{S}$ полагается равным числу правил и обозначается через $|\mathbb{S}|$. *Конкатенацией* ПП $\mathbb{S}_1$ и $\mathbb{S}_2$ называется прямолинейная программа, которая выводит текст $S_1 \cdot S_2$, и обозначается $\mathbb{S}_1 \cdot \mathbb{S}_2$. Для удобства прямолинейная программа отождествляется с ее деревом вывода.

Рис. 0.1. Дерево вывода ПП, которая выводит 7-е слово Фибоначчи.



Высота узла бинарного дерева определяется рекурсивно. Высота терминального узла (листа) полагается равной 0. Высота нетерминального узла полагается равной 1 + максимальная из высот его детей. Обозначим через $h(\mathbb{S}_i)$ высоту правила $\mathbb{S}_i$. *Глубиной k-го узла* будем называть количество узлов, которые являются предками этого узла и обозначать через $depth_k$. *AVL-дерево* – бинарное дерево такое, что высоты детей каждого его внутреннего узла отличаются не более чем на 1. *Декартово дерево* – это бинарное дерево такое, что в каждом узле хранится пара ключей x и y. При этом оно является бинарным деревом поиска по ключам x и кучей по ключам y. *Декартовым деревом с неявным ключом* называется декартово дерево, в котором не хранится информация о ключах x.

LZ-факторизацией текста S называется последовательность строк $F_1, F_2, \ldots, F_k$ такая, что $S = F_1 \cdot F_2 \cdot \ldots \cdot F_k$, $F_1 = S[0]$, F_i – наибольший префикс $S[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |S|]$, который входит в качестве подстроки в $F_1 \cdot \ldots \cdot F_{i-1}$ или $S[|F_1 \cdot \ldots \cdot F_{i-1}|]$, если префикс пуст.

Обозначения и сокращения

ПП — прямолинейная программа

Введение

В настоящее время алгоритмы обрабатывающие большие объемы данных привлекают все больше внимания. Причина не только в том, что с каждым днем растет объем доступных данных. Так, например, в области машинного обучения привлечение большего объема данных позволяет увеличить точность получаемой модели. Один из подходов позволяющих снизить скорость роста размера входных данных заключается в работе со сжатыми представлениями.

Известны различные сжатые представления данных: прямолинейные программы [1] (кратко ПП), коллаж системы [2], представления данных с помощью антисловарей [3] и т.д. В настоящее время сжатие текста с помощью контекстно-свободных грамматик (таких как ПП) является активно развивающимся направлением научных исследований. Причина этого не только в том, что грамматики обеспечивают хорошо-структурированное сжатие данных, но и в том, что сжатие с помощью ПП в некотором смысле полиномиально эквивалентно сжатию данных с помощью алгоритма Лемпеля-Зива. Другими словами, существует полиномиальная зависимость между размером ПП, выводящей заданный текст S, и размером словаря, построенным алгоритмом Лемпеля-Зива для текста S (подробнее см. [1]). Заметим, что классические алгоритмы сжатия LZ78 [4] и LZW [5] являются частными случаями грамматического сжатия. При этом другие алгоритмы из семейства алгоритмов Лемпеля-Зива, например LZ77 [6] и кодирование повторов, не укладываются в модель грамматического сжатия.

Поскольку ПП обеспечивают хорошо-структурированное представление данных, то возникает идея решать классические строковые задачи в терминах ПП. Существует класс алгоритмов, которые решают следующие строковые задачи в терминах ПП: **Поиск образца в тексте** [7], **Наибольшая общая подстрока** [8], считающая версия задачи **Поиск всех палиндромов** [8], разновидность задачи **Наибольшая общая подпоследовательность** [9]. В тоже время константы, которые скрываются за определением функции O в оценке сложности таких алгоритмов, как правило очень большие. Также полиномиальная связь между размером ПП, выводящей заданный текст S, и размером LZ77-словаря для этого же текста S еще не гарантирует, что ПП обеспечивают высокую степень сжатия на практике. Поэтому актуальным становится следующий вопрос: существуют ли модели сжатия с помощью ПП эффективно работающие на практике? В ра-

боте детально рассматриваются два вопроса. Насколько трудно строить ПП на практике? Насколько высокий уровень сжатия обеспечивают ПП относительно классических алгоритмов сжатия на практике?

Алгоритмы построения ПП

Классическая формулировка задачи построения ПП выглядит следующим образом:

Задача: **Построение ПП**

Вход: текст S;

Выход: ПП $\mathbb{S}$, которая выводит S;

Однако, задача построения грамматики минимального размера для заданного текста является NP-трудной [10]. Поэтому представляют интерес приближенные алгоритмы. Один из основных подходов использует идею факторизации текста. Суть подхода заключается в том, что для фиксированного разбиения текста алгоритм последовательно просматривает факторы. Для каждого фактора алгоритм строит ПП, которая выводит этот фактор, а затем добавляет полученную программу в результирующую ПП. При таком подходе размер результирующей ПП зависит не только от размера исходного текста, но и от размера факторизации. Переформулируем задачу построения ПП следующим образом:

Задача: **Построение ПП**

Вход: текст S и его факторизация $F_1, F_2, \ldots, F_k$;

Выход: ПП $\mathbb{S}$, которая выводит S;

В качестве одного из способов факторизации в [1] рассматривается факторизация, порождаемая алгоритмом LZ77, так называемая LZ-факторизация.

Если существует полиномиальный алгоритм, решающий задачу построения ПП, то существует полиномиальная связь между размером построенной ПП и размером факторизации текста. В частности, если на вход дана LZ-факторизация текста, то существует полиномиальная связь между размером построенной ПП и размером словаря, хранимым алгоритмом LZ77.

Формулировка задачи построения ПП не накладывает никаких ограничений на дерево вывода ПП. То есть единственное ограничение – дерево вывода ПП является бинарным деревом. Существуют различные виды бинарных деревьев. Какой вид больше подходит для решения этой задачи? В классическом ал-

горитме построения ПП [1] используются сбалансированные деревья, а именно, AVL-деревья.

Особенностью AVL-деревьев является строгая логарифмическая оценка на высоту [11]. Это одна из основных причин почему этот вид деревьев используется в алгоритме Риттера. Но существуют технические трудности с построением AVL-деревьев. Они будут детально обсуждаться в этой главе. Также в этой главе рассматривается альтернативный алгоритм построения ПП на основе декартовых деревьев.

Существует вероятностная логарифмическая оценка на высоту декартова дерева (см. раздел Оценка высоты декартова дерева). Этот класс бинарных деревьев рассматривается потому, что алгоритм построения декартова дерева более простой, чем AVL-дерева и при этом с большой вероятностью получится дерево логарифмической высоты. Поэтому интересно провести сравнительный анализ этих подходов на практике.

Узкое место алгоритма Риттера

Следующая теорема дает оценку сверху для размера построенной ПП.

Теорема 0.0.1 *Существует алгоритм, который для заданного текста S длины n и его факторизации размера k строит ПП, выводящую текст S, размера $O(k \log n)$ за время $O(k \log n)$.*

Доказательство теоремы содержит алгоритм построения ПП. AVL-сбалансированными грамматиками (кратко AVL-грамматиками) будем называть такие прямолинейные программы, что их дерево вывода является AVL-деревом. Мы напомним ключевые идеи алгоритма поскольку они важны для дальнейшей дискуссии. Основной операцией используемой в алгоритме является конкатенация AVL-грамматик. Следующая лемма дает оценку сверху для этой операции:

Лемма 0.0.2 *Пусть $\mathbb{S}_1, \mathbb{S}_2$ – нетерминальные правила AVL-грамматики. Тогда мы можем построить AVL-грамматику $\mathbb{S} = \mathbb{S}_1 \cdot \mathbb{S}_2$, которая выводит текст $S_1 \cdot S_2$, за время $O(|h(\mathbb{S}_1) - h(\mathbb{S}_2)|)$ и добавив $O(|h(\mathbb{S}_1) - h(\mathbb{S}_2)|)$ новых правил.*

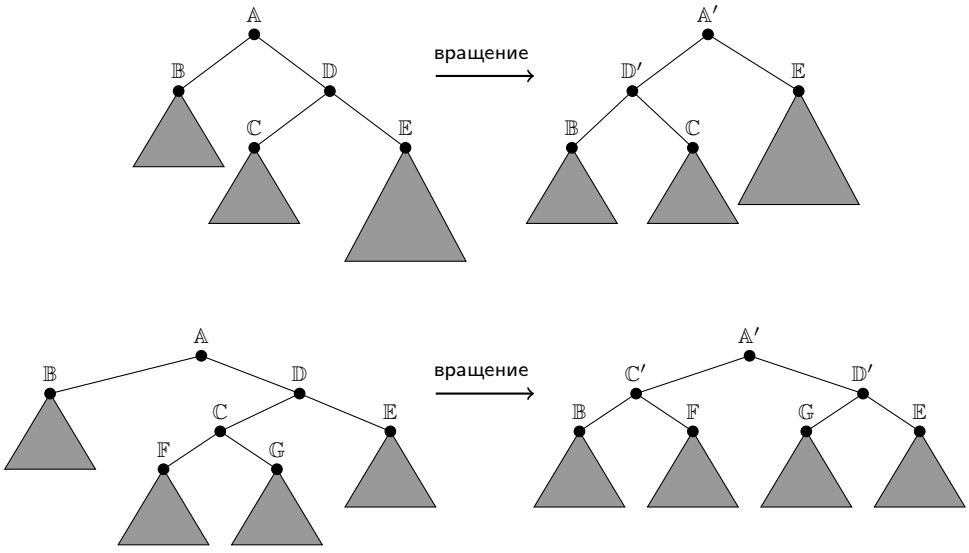
Далее мы представим алгоритм Риттера в общих чертах:

Алгоритм: Алгоритм строит ПП индуктивно по k.

База: Полагаем $\mathbb{S}$ равной терминальному правилу, выводящему $S[0]$.

Шаг: Предположим, что для фиксированного $i > 1$ уже построена ПП $\mathbb{S}$, выводящая текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Так как факторизация текста S фиксирована, то известны позиции вхождения F_{i+1} в тексте $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Используя алгоритм взятия подграмматики мы находим правила $\mathbb{S}_1, \mathbb{S}_2, \ldots, \mathbb{S}_\ell$ такие, что $F_{i+1} = S_1 \cdot S_2 \cdot \ldots \cdot S_\ell$. Поскольку $\mathbb{S}$ AVL-грамматика, то $\ell = O(\log |S|)$. С помощью Леммы 3.2 конкатенируем правила $\mathbb{S}_1, \mathbb{S}_2, \ldots, \mathbb{S}_\ell$ в некотором фиксированном порядке (подробнее см. [1]). Полагаем $\mathbb{S} = \mathbb{S} \cdot (\mathbb{S}_1 \cdot \mathbb{S}_2 \cdot \ldots \cdot \mathbb{S}_i)$.

Рис. 0.1. Типы вращений узлов AVL-дерева



При работе с AVL-деревьями самой трудной задачей является сохранение баланса в дереве. Например, при добавлении нового правила в дерево порождается последовательность вращений. Существует два типа вращений, они представлены на Рисунке 1. При этом каждое вращение может порождать не более трех новых узлов. Также каждое вращение может породить не более трех висячих (неиспользуемых) узлов. Таким образом алгоритм большую часть времени проводит за поддержанием баланса. Существует два направления оптимизации алгоритма: строить более компактные грамматики или минимизировать число запросов к грамматике. Оптимизация числа запросов к грамматике становится важной в случае, когда размер входного текста превышает объем оперативной памяти. То есть мы не можем хранить текущее состояние ПП в оперативной па-

мяти. Формально это значит, что стоимость операции доступа к ПП превышает стоимость вычислений в оперативной памяти.

Следующий пример иллюстрирует узкое место алгоритма Риттера:

Пример: Пусть S – текст длины 2^n, где n – фиксированное натуральное число. Предположим, что алгоритм Риттера уже построил ПП $\mathbb{S}$, которая выводит $S[0 \dots 2^{n-1}]$. Пусть $F_1, F_2, \dots, F_n$ оставшиеся факторы такие, что $F_1 \cdot F_2 \cdot \dots \cdot F_n = S[2^{n-1} \dots 2^n]$ и $|F_i| = 2^{n-i-1}$, где $i \in \{1, 2, \dots, n-1\}$.

Давайте оценим число вращений которое потребуется при последовательной конкатенации грамматик $(((\mathbb{S} \cdot \mathbb{F}_1) \cdot \mathbb{F}_2) \dots) \cdot \mathbb{F}_{n-1}$ в худшем случае. Чтобы получить оценку сверху, мы предполагаем, что после конкатенации двух правил высота результирующего дерева увеличивается на 1.

$$\sum_{i=1}^{n-1}(\log 2^{n-1}+(i-1)-\log 2^{n-i-1}) = \frac{(n-1)(n-2)}{2}(1+\log 2)-(n-2) = \Theta(n^2).$$

Заметим, что если бы алгоритм мог выбирать порядок конкатенации грамматик, то число вращений порождаемых алгоритмом могло быть существенно меньше. Например, если выполнить конкатенацию грамматик в обратном порядке $\mathbb{S} \cdot (\mathbb{F}_1 \dots (\mathbb{F}_{n-2} \cdot \mathbb{F}_{n-1}))$, то получится следующая оценка:

$$\sum_{i=1}^{n-1}(\log 2^{n-i} - \log 2^{n-i-1}) = (n-1)\log 2 = \Theta(n).$$

Следующий пример показывает, что несколько факторов могут быть обработаны вместе, если они входят в тексте выводимом из одной ПП.

Пример: Пусть $S = b \cdot a^{2^{n-1}} \cdot b \cdot a^{2^{n-2}} \cdot \dots \cdot b \cdot a$, где n – фиксированное натуральное число и $|S| = 2^n + n - 2$. Рассмотрим LZ-факторизацию S:

$$b \cdot a \cdot a \cdot a^2 \cdot a^4 \cdot \dots \cdot a^{2^{n-2}-1} \cdot ba^{2^{n-2}} \cdot ba^{2^{n-3}} \cdot \dots \cdot ba.$$

Пусть $\mathbb{S}_1$ – ПП, выводящая текст $b \cdot a^{2^{n-1}}$. Очевидно, что все факторы, начиная с $b \cdot a^{2^{n-2}}$, входят в S_1. Следовательно, не имеет значения в каком порядке они будут обработаны алгоритмом. Алгоритм мог бы построить ПП $\mathbb{S}_2, \mathbb{S}_3, \dots, \mathbb{S}_{n-1}$ выводящие тексты $ba^{2^{n-3}}, ba^{2^{n-4}}, \dots, ba$ соответственно. На-

конец алгоритм мог бы конкатенировать полученные ПП в следующем порядке: $\mathbb{S}_1 \cdot (\ldots (\mathbb{S}_{n-3} \cdot (\mathbb{S}_{n-2} \cdot \mathbb{S}_{n-1})))$.

Идея оптимизации заключается в том, чтобы обрабатывать факторы максимально возможными группами и в рамках каждой группы факторов выбирать оптимальный порядок конкатенации. Интуитивно, идея очень проста: если алгоритм уже построил большую ПП, то большинство факторов входит в тексте выводимом из нее, а значит факторы могут быть обработаны вместе.

Оптимизация алгоритма Риттера

Вход: текст S и его LZ-факторизация $F_1, F_2, \ldots, F_k$.
Выход: ПП $\mathbb{S}$, которая выводит текст S.
Алгоритм: Алгоритм строит ПП по индукции.

База: Полагаем $\mathbb{S}$ равным терминальному правилу, выводящему $S[0]$.

Шаг: Зафиксируем произвольное число $0 < i < k$. Пусть $\mathbb{S}$ – ПП, которая выводит $F_1, F_2, \ldots, F_i$. Пусть $1 \leqslant \ell \leqslant k - i$ – наибольшее число такое, что $F_{i+\ell}$ входит в $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Так как LZ-факторизация S фиксирована, тогда значение ℓ может быть найдено с помощью линейного поиска по факторам. ПП $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$, которые выводят тексты $F_{i+1}, F_{i+2}, \ldots, F_{i+\ell}$ соответственно, вычисляются с помощью алгоритма взятия подграмматики (аналогично алгоритму из [1]).

Алгоритм вычисляет конкатенацию ПП $F_{i+1}, F_{i+2}, \ldots, F_{i+\ell}$ с помощью динамического программирования. Положим $\varphi(p, q)$ – функция, которая вычисляет число операций вращения AVL-дерева при конкатенации $\mathbb{F}_p, \mathbb{F}_{p+1}, \ldots, \mathbb{F}_q$. Алгоритм использует следующую формулу для функции φ:

$$\varphi(p,q) = \begin{cases} 0 & \text{если } p = q, \\ \min_{r=p}^{q}(\varphi(p,r) + \varphi(r+1,q) + \\ |\log(|F_{i+p}| + \ldots + |F_{i+r}|) - \log(|F_{i+r+1}| + \ldots + |F_{i+q}|)| & \text{иначе.} \end{cases}$$

Формула корректна, так как конкатенация двух правил $\mathbb{F}_p$ и $\mathbb{F}_q$ порождает не более $2 \cdot |h(\mathbb{F}_p) - h(\mathbb{F}_q)|$ вращений AVL-дерева (см. [1]) и $h(\mathbb{F}_p) \leqslant 1.44 \cdot \log |F_p|$ (см. [11]). Для простоты общие константы были отброшены из формулы.

Вычисленные значения функции $\varphi(p,q)$ хранятся в φ-таблице. φ-таблица это квадратная таблица размера $(\ell-1)\times(\ell-1)$, которая хранит информацию о значении $\varphi(p,q)$ и информацию об оптимальном разбиении. Ячейки таблицы в p-й строке и q-м столбце по умолчанию заполнены нулями, если $p \leqslant q$. Алгоритм заполняет таблицу в следующем порядке: сначала заполняются ячейки (p,q) такие, что $q-p=1$, потом заполняются ячейки (p,q) такие, что $q-p=2$ и т.д. Используя такой порядок заполнения алгоритм избегает рекурсивного пересчета значений $\varphi(p,q)$, так как они уже содержаться в таблице.

Любое значение $\varphi(p,q)$ может быть вычислено за время $O(\ell)$ (псевдокод процедуры представлен на Рисунке 0.2). В итоге алгоритм заполнит φ-таблицу за время $O(\ell^3)$ и использует $O(\ell^2)$ памяти.

Рис. 0.2. Псевдокод функции, вычисляющей значение $\varphi(p,q)$

$result = +\infty;$
$l = 0, r = |f_{i+p}| + \cdots + |f_{i+q}|;$
for $(int\ r = p; r \leqslant q; r++)$ {
  $l += |f_{i+r}|;$
  $r -= |f_{i+r}|;$
  $tmp = \varphi(p,r) + \varphi(r+1,q) + |\log l - \log r|;$
  if $(tmp < result)$
    $result = tmp;$
}

Заметим, что значение $\varphi(1,\ell)$ оценивает сверху количество операций перебалансировки, которое необходимо выполнить для конкатенации последовательности ПП $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$. Поскольку в каждой ячейке φ-таблицы хранится индекс оптимального разбиения, то алгоритм за время $O(\ell)$ может определить оптимальный порядок конкатенации ПП $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$. Используя этот порядок алгоритм строит ПП $\mathbb{F}$, которая выводит текст $F_{i+1} \cdot F_{i+2} \cdot \ldots \cdot F_{i+\ell}$. В итоге алгоритм полагает ПП $\mathbb{S}$ равной $\mathbb{S} \cdot \mathbb{F}$.

Теорема 0.0.3 *Пусть $F_1, F_2, \ldots, F_k$ – LZ-факторизация текста w, тогда представленный выше алгоритм (оптимизированная версия алгоритма Риттера) построит ПП, выводящую текст w, размера $O(k \log n)$.*

Доказательство: Разобьем LZ-факторизацию на независимые группы с помощью жадного алгоритма. Пусть $G_1, G_2, \ldots, G_p$ – разбиение LZ-факториза-

ции, тогда $G_1 = \{F_1\}$ и для любого $1 < j \leqslant p$ $G_j = \{F_{i_j}, F_{i_j+1}, \ldots, F_{i_j+\ell_j}\}$, где F_{i_j} входит в $F_{i_{j-1}} \cdot F_{i_{j-1}+1} \cdot \ldots \cdot F_{i_{j-1}+\ell_{j-1}}$ и не входит в $F_1 \cdot F_2 \cdot \ldots \cdot F_{i_{j-2}+\ell_{j-2}}$; каждый фактор $F_{i_j+1}, F_{i_j+2}, \ldots, F_{i_j+\ell_j}$ входит в $F_1 \cdot F_2 \cdot \ldots \cdot F_{i_{j-1}+\ell_{j-1}}$. $\ell_1, \ell_2, \ldots, \ell_p$ – размеры соответствующих групп факторов и $\sum_{j=1}^{p} \ell_j = k$.

Число операций вращений необходимое при работе с j-ой группой факторов ограничено сверху $2.88 \cdot (\log|F_{i_j}| + \log|F_{i_j+1}| + \cdots + \log|F_{i_j+\ell_j}|) \cdot \ell_j \leqslant 2.88 \cdot \log n \cdot \ell_j$. Суммируем по параметру j и получаем, что общее число операций вращения необходимое для построения ПП ограничено сверху $2.88 \cdot k \cdot \log n$. Поскольку каждая операция вращения порождает не более трех новых правил, тогда размер ПП будет порядка $O(k \log n)$

Так как мы используем алгоритм динамического программирования для вычисления порядка конкатенации, то мы не можем оценить сложность алгоритма в зависимости от входных параметров. Заметим, что если размеры всех групп факторов равны 1, то мы получаем в точности алгоритм Риттера [1]. Ясно, что новый алгоритм порождает меньше вращений чем классический алгоритм, с другой стороны новый алгоритм использует дополнительные ресурсы для вычисления порядка конкатенации. В следующей главе мы сравним эффективность обоих алгоритмов на практике.

Построение ПП на основе декартовых деревьев

Алгоритмы построения ПП тесно связаны со структурой дерева вывода. Например, в случае AVL-деревьев нам приходится тратить время на поддержание баланса в дереве. Поэтому возникает идея, что со сменой структуры данных для представления дерева вывода может получиться принципиально другой алгоритм построения ПП. В качестве альтернативной структуры данных в этой работе рассматриваются декартовы деверья.

Лемма 0.0.4 *Для заданного набора ключей $(x_1, y_1), \ldots, (x_n, y_n)$ существует единственное декартово дерево, содержащее этот набор ключей при условии, что все ключи x_i попарно различны и все ключи y_i попарно различны.*

Доказательство: Докажем индукцией по параметру n.
База: Для $n = 0$ существует единственное пустое дерево.

Шаг: Зафиксируем произвольное число $0 < k < n$. Пусть для любого набора ключей мощности меньше k существует единственное декартово дерево, их содержащее. Докажем это утверждение для набора мощности k. Пусть $root$ – такой индекс, что $y_{root} = \min_{i=1}^{k} y_i$, тогда множество ключей разбивается на два множества $K_{left} = \{(x_i, y_i) | x_i < x_{root}\}$ и $K_{right} = \{(x_i, y_i) | x_i > x_{root}\}$. По свойствам кучи (x_{root}, y_{root}) – ключи корня искомого дерева. По свойствам бинарного дерева поиска левое и правое поддеревья искомого дерева построены по множествам ключей K_{left} и K_{right} соответственно. Так как ключи попарно различны, то значения (x_{root}, y_{root}), K_{left} и K_{right} определяются однозначно, по предположению индукции левое и правое поддеревья строятся однозначно.

Следствие. В силу единственности декартова дерева, содержащего данный набор ключей, все его характеристики (например, высота) однозначно определяются набором ключей и не зависит от способов построения самого декартового дерева.

Рассмотрим операции над декартовыми деревьями. Существует две стандартные операции над декартовыми деревьями: *split* и *merge*.

merge – бинарная операция над декартовыми деревьями, результатом которой является дерево, содержащее все ключи деревьев-операндов. Причем на операнды наложено условие, что любой ключ x первого дерева меньше любого ключа x второго дерева. Рассмотрим алгоритм, который выполняет эту операцию:

ОПЕРАЦИЯ: *merge*

ВХОД: Два декартовых дерева T_1, T_2.

ВЫХОД: Декартово дерево T, содержащее все ключи из деревьев T_1 и T_2.

АЛГОРИТМ:

а) Если одно из деревьев пустое, то результатом будет другое дерево.

б) Пусть оба дерева непустые и положим $T_1 = (L_1, R_1, x_1, y_1)$, $T_2 = (L_2, R_2, x_2, y_2)$. Из свойств кучи получаем, что y_1 – минимальный ключ y в дереве T_1, y_2 – в T_2. Поэтому корнем дерева-результата будет один из корней деревьев-операндов.

1) Если $y_1 < y_2$, то построим дерево $T_3 = merge(R_1, T_2)$. Тогда результатом будет дерево $T = (L_1, T_3, x_1, y_1)$.

2) Случай $y_1 \geqslant y_2$, аналогичен предыдущему.

Лемма 0.0.5 *Если h_1 и h_2 – высоты деревьев T_1 и T_2 соответственно, то операция* merge *для этих двух деревьев работает за время $O(h_1 + h_2)$.*

Доказательство: Заметим, что если деревья непустые, то алгоритм делает ровно один рекурсивный вызов, при этом высота одного из деревьев уменьшается как минимум на 1.

split – бинарная операция над декартовым деревом и числом (позиция разреза), результатом которой является упорядоченная пара декартовых деревьев, где ключи x первого дерева меньше позиции разреза, а ключи x второго дерева не меньше позиции разреза.

Рассмотрим алгоритм, который выполняет эту операцию:

ОПЕРАЦИЯ: *split*

ВХОД: Декартово дерево T и x_{split} – позиция разреза.

ВЫХОД: Упорядоченная пара декартовых деревьев (L, R) такая, что $x_i < x_{split}$ для всех $x_i \in L$ и $x_j \geqslant x_{split}$ для всех $x_j \in R$.

АЛГОРИТМ:

а) Если дерево T пустое, то результатом будет два пустых дерева.

б) Пусть T непустое дерево и положим $T =$(LT, RT, x, y).

1) Если $x_{split} < x$, то необходимо разрезать левое поддерево $(L', R') = split(LT, x_{split})$. Тогда результатом будет (L, R), где $L = L'$, $R =$(R', RT, x, y).

2) Случай $x_{split} \geqslant x$ аналогичен предыдущему.

Лемма 0.0.6 *Пусть h – высота декартова дерева T, тогда операция* split *работает за время $O(h)$ независимо от позиции разреза.*

Доказательство: Пусть T непустое дерево, тогда алгоритм делает ровно один рекурсивный вызов для текущего поддерева, при этом высота уменьшается как минимум на 1.

В итоге мы получаем, что сложность операций с декартовым деревом линейно зависит от его высоты. В действительности высота декартова дерева может быть линейной относительно его размера. Например, высота декартова

дерева, построенного по набору ключей (1, 1), ..., (n, n), будет равна n. В следующем разделе мы докажем, что декартово дерево из n узлов, ключи y которых являются независимыми непрерывными случайными величинами с одинаковым вероятностным распределением, имеет высоту $O(\log n)$.

Оценка высоты декартова дерева

В этом разделе мы предполагаем, что все ключи x пронумерованы в соответствии с линейным порядком, то есть $x_1 < x_2 < \dots x_n$, и каждое y_i выбрано случайно и независимо с одинаковым распределением.

Следующее наблюдение является следствием упорядоченности ключей y в декартовом дереве.

Лемма 0.0.7 *i-й узел является предком k-го узла тогда и только, когда $y_i < y_j$ для любого j такого, что $i < j \leqslant k$ или $k \leqslant j < i$.*

Доказательство: Рассмотрим случай $i < k$, в случае $i > k$ доказательство аналогично.

Необходимость. Допустим, что $y_i < y_j$ для всех $i < j \leqslant k$. Тогда по свойствам кучи i-й узел не может лежать в поддереве с корнем в k-м узле. Более того, не может существовать индекс j такой, что j-й узел общий прародитель i-го и k-го узлов, но эти узлы лежат в его разных поддеревьях. Если он существует, то $x_i < x_j < x_k$, следовательно, $i < j < k$. Но тогда j-й не мог быть прародителем i-го узла. Остался последний вариант взаимного расположения i-го и k-го узлов — i-й является прародителем k-го узла.

Достаточность. Пусть i-й узел является предком k-го узла. Докажем от противного. Предположим, что существует j такое, что $i < j \leqslant k$ и $y_j < y_i$. По свойствам кучи i-й узел не может быть прародителем j-го узла. Если j-й узел является прародителем i-го узла, то из неравенства $x_i < x_j < x_k$ следует, что i-й узел содержится в левом поддереве j-го узла, а k-й — в правом. Это противоречит тому, что i-й узел прародитель k-го узла. Остается последний вариант взаимного расположения i-го и j-го узла — некоторый l-й узел является их общим прародителям, но они содержатся в его разных поддеревьях. Тогда получаем неравенство $x_i < x_l < x_j \leqslant x_k$, из которого следует, что k-й узел со-

держится в правом поддереве l-го узла. Получаем противоречие. Следовательно, наше предположение не верно.

Лемма 0.0.8 *Математическое ожидание глубины k-го узла равно $H_k + H_{n-k+1} - 1$, где $H_j = \sum_{i=1}^{j} \frac{1}{i} \approx \ln j$.*

Доказательство: Пусть A_i – случайная величина такая, что ее значение равно единице, если i-й узел является предком k-го узла, иначе равно нулю. Ясно, что A_i и A_j независимы при $i \neq j$. Поскольку y_i выбраны случайно и независимо с одинаковым распределением, тогда из предыдущего наблюдения следует, что $M(A_i) = \frac{1}{|i-k|+1}$ при $i \neq k$.

$$M(depth_k) = M(\sum_{i=1}^{n} A_i) = \sum_{i=1}^{n} M(A_i) = \sum_{i=1}^{n} \frac{1}{|i-k|+1} =$$

$$= \sum_{i=1}^{k} \frac{1}{k-i+1} + \sum_{i=k}^{n} \frac{1}{i-k+1} - 1 = H_k + H_{n-k+1} - 1.$$

Следовательно, математическое ожидание глубины конкретного узла будет порядка $O(\log n)$.

Для доказательства оценки высоты декартова дерева нам потребуется следующее вспомогательное утверждение:

Лемма 0.0.9 ([12]) *Рассмотрим независимые случайные величины A_1, $A_2 \ldots A_n$. Тогда если $A = \sum_{i=1}^{n} A_i$ и $\mu = M(A)$, то для любого $\delta > 0$*

$$P\{A > (1+\delta)\mu\} < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{\mu}$$

Лемма 0.0.10 $P\{depth_k \leqslant 7(H_k + H_{n-k+1} - 1)\} \geqslant 1 - (\frac{1}{n})n^6$.

Доказательство: Используя Лемму 3.5 получаем $M(depth_k) = H_k + H_{n-k+1} - 1$. Тогда применяя Лемму 3.6 получаем что для любого $\delta > 0$ выполнено следующее неравенство:

$$P\{A > (1+\delta)\mu\} < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{\mu}$$

где $\mu = H_k + H_{n-k+1} - 1$. Заметим, что $H_k + H_{n-k+1} - 1 \leqslant H_n \leqslant \ln n$ для любого k. Тогда подставив $\delta = e^2 - 1 > 6$ получаем следующую оценку:

$$P\{A > (1+\delta)(H_k + H_{n-k+1} - 1)\} < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{H_n}$$
$$< \left(\frac{e^{\delta+1}}{(e^2)^{(1+\delta)}}\right)^{H_n} < \left(\frac{1}{e}\right)^{\delta H_n} < \left(\frac{1}{n}\right)^6$$

Теорема 0.0.11 *Декартово дерево, построенное по набору ключей* $(x_1, y_1), \ldots, (x_n, y_n)$*, где* y_i *выбраны случайно, независимо и имеют одинаковое непрерывное распределение, имеет высоту* $O(\log n)$.

Доказательство: Пусть B_i — событие, при котором $depth_i > 7(H_k + H_{n-k+1} - 1)$. Тогда оценим вероятность объединения этих событий, то есть вероятность того, что глубина дерева будет больше чем $7(H_k + H_{n-k+1} - 1)$.

$$P\{B_1 \cup \ldots \cup B_n\} \leqslant P\{B_1\} + \ldots + P\{B_n\} \leqslant n \cdot \left(\frac{1}{n}\right)^6 = \left(\frac{1}{n}\right)^5$$

Таким образом, вероятность того, что дерево глубина дерева будет больше чем $7(H_k + H_{n-k+1} - 1)$ не превосходит $\frac{1}{n^5}$. Иначе говоря, с вероятностью $1 - \frac{1}{n^5}$ глубина декартова дерева будет $O(\log n)$. Поскольку глубина дерева ограничена сверху числом n,тогда математическое ожидание высоты декартова дерева со случайными ключами y равна $O(\log n)$.

Следствие 1. Ключи y для декартова дерева можно не хранить. Достаточно случайно их генерировать по мере необходимости. При этом логарифмическая оценка на высоту дерева сохранится.

Следствие 2. Операции *merge* и *split* для деревьев со случайными ключами y работает за $O(\log n)$.

Так как распределение является непрерывным, то ключи y являются попарно различными. Следовательно, для таких деревьев выполняется утверждение о единственности дерева.

Класс декартовых деревьев, подходящий для построения ПП

Для того, чтобы построить ПП требуются две операции над деревом вывода: операция взятия поддерева по заданным границам и операция конкатенации двух деревьев. Аналогичные операции существуют для декартовых деревьев. А именно, *split* – операция разбиения декартова дерева на два поддерева по заданной границе и *merge* – операция слияния двух декартовых деревьев. В классической реализации операции *merge* требуется дополнительное условие: ключи x первого дерева должны быть меньше ключей x второго дерева. Более того, операция *merge* не гарантирует сохранения порядка листов после ее применения. Следовательно, чтобы использовать декартовы деревья в алгоритмах построения ПП необходимо преодолеть следующие трудности:

– Корректная перегенерация ключей для деревьев, полученных после применения операции *split*. Эта задача возникает в следующем случае: пусть алгоритм уже построил некоторое дерево T, для следующего фактора алгоритм вырезает поддерево T' дерева T и необходимо выполнить операцию *merge* для T и T'. Согласно требованиям операции *merge* необходимо полностью перегенерировать ключи x для дерева T'.

– Реализовать операцию *merge* так, чтобы гарантировать порядок листов в результирующем дереве. Так как необходимо построить ПП, которая выводит в точности исходный текст.

Пусть для произвольного декартова дерева была потеряна информация о его ключах x. Тогда по структуре дерева всегда можно восстановить отношение линейного порядка на ключах x. Например, обойдем рекурсивно дерево в следующем порядке: левое поддерево, корень, правое поддерево. Дополнительно в каждом узле T_i будем хранить $count(T_i)$ – количество узлов в поддереве с корнем в T_i. Тогда мы сможем восстановить порядковый номер ключа x в данном узле среди ключей x из узлов данного поддерева (это количество узлов в левом поддереве плюс один). Таким образом можно отказаться от явного хранения значения ключей, сохраняя лишь их порядок. Рассмотрим класс декартовых деревьев по неявному ключу в класса деревьев на основе которых мы будем строить ПП. Для этого класса деревьев справедливы следующие утверждения:

Лемма 0.0.12 *Для заданного набора ключей $(y_1, y_2, \dots y_n)$ существует единственное декартово дерево по неявному ключу, содержащее его при условии, что все ключи y_i попарно различны.*

Доказательство: Докажем от противного. Предположим, что существует два различных декартовых дерева по неявному ключу, построенных по заданному кортежу ключей y. Рассмотрим естественные декартовы деревья для них. Очевидно, что эти деревья будут различными. Но они содержат один и тот же набор ключей. Это противоречит теореме о единственности декартова дерева. Значит, наше предположение неверно и существует единственное декартово дерево по неявному ключу, построенное по заданному кортежу ключей y.

Теорема 0.0.13 *Декартово дерево по неявному ключу, построенное по набору ключей $(y_1, y_2 \dots y_n)$, где y_i выбраны случайно и независимо с одинаковым вероятностным распределением, имеет высоту $O(\log n)$.*

Доказательство: Восстановим из декартова дерева по неявному ключу обычное декартово дерево. По построению их структуры совпадают. Следовательно и высоты этих деревьев совпадают. В силу теоремы для декартовых деревьев высота этих деревьев $O(\log n)$.

В дальнейшем под значением ключа x для узла $T_i \in T$ мы будем подразумевать порядковый номер T_i в отношении линейного порядка ключей x для T.

С помощью декартовых деревьев с неявным ключом решается вопрос о перегенерации ключей x. Также если в листах такого дерева хранить дополнительную информацию (например некоторый символ), то можно организовать динамический массив символов. Другими словами операция *merge* сохраняет порядок листов. Поскольку декартовы деревья с неявным ключом всего лишь подкласс всех декартовых деревьев, то для этого подкласса сохраняется логарифмическая оценка высоты.

Рассмотрим операции, которые необходимы для построения ПП:

ОПЕРАЦИЯ: *split*

ВХОД: T – декартово дерево с неявным ключом, $0 < k \leqslant |T|$ – число;

Выход: L, R – пара декартовых деревьев с неявным ключом такие, что L содержит все узлы T с ключами меньшими k, а R содержит все остальные узлы T;

Алгоритм:

Алгоритм работает рекурсивно, начиная с корня дерева T. Пусть в текущий момент алгоритм просматривает узел $T_i = (T_\ell, T_r)$. Рассмотрим следующие случаи:

– Если $k = count(T_\ell) + 1$, то $L = T_\ell, R = T_r$.

– Если $k < count(T_\ell) + 1$, то необходимо разрезать T_ℓ. Пусть $(L', R') = split(T_\ell, k)$, тогда $L = L', R = (R', T_r)$.

– Если $k > count(T_\ell) + 1$, то необходимо разрезать T_r. Пусть $(L', R') = split(T_r, k - count(T_\ell) - 1)$, тогда $L = (T_\ell, L'), R = R'$.

Сложность:

На каждом шаге рекурсии алгоритм либо завершает работу, либо выполняет рекурсивный вызов с узлом меньшей высоты. Следовательно, сложность алгоритма $O(\log |T|)$.

Операция: *merge*

Вход: T_1, T_2 – декартовы деревья с неявным ключом;

Выход: T – декартово дерево с неявным ключом, содержащее все узлы из T_1 и T_2;

Алгоритм:

Алгоритм работает рекурсивно, начиная с корней деревьев T_1 и T_2. Пусть в текущий момент алгоритм просматривает узлы $T_i = (T_{\ell_1}, T_{r_1}) \in T_1, T_j = (T_{\ell_2}, T_{r_2}) \in T_2$. Сгенерируем произвольное число y^* от 1 до $count(T_i)+count(T_j)+1$.

– Если $0 < y^* \leqslant count(T_i)$, то рекурсивно построим $T' = merge(T_{r_1}, T_2)$ и возвращаем (T_{ℓ_1}, T').

– Если $count(T_i) < y^* \leqslant count(T_j)$, то рекурсивно строим $T' = merge(T_1, T_{\ell_2})$ и возвращаем (T', T_{r_2}).

– Если $y^* = count(T_i) + count(T_j) + 1$, то возвращаем (T_i, T_j).

Заметим, что результатом слияния декартова дерева T и пустого дерева является дерево T.

Сложность:

На каждом шаге рекурсии алгоритм спускается либо внутри первого дерева, либо внутри второго дерева. Следовательно, сложность алгоритма $O(\log |T_1| + \log |T_2|)$.

В итоге для реализации всех необходимых операций нам потребовалось лишь дополнительно хранить информацию о количестве узлов в декартовом дереве.

Алгоритм построения ПП на основе декартовых деревьев

В этом разделе под ПП мы будем понимать прямолинейную программу такую, что ее дерево вывода является декартовым деревом по неявному ключу.

Переформулируем операции $split$ и $merge$ в терминах ПП и докажем их корректность.

Операция: *split*

Вход: ПП $\mathbb{S}$, которая выводит текст S, $0 < k < |S|$ – число;

Выход: ПП $\mathbb{L}, \mathbb{R}$ такие, что $\mathbb{L}$ выводит $S[0 \dots k]$, $\mathbb{R}$ выводит $S[k+1 \dots |S|]$;

Алгоритм: Аналогичен алгоритму для декартовых деревьев по неявному ключу.

Корректность:

Докажем корректность индукцией по $|S|$.

База: Пусть $\mathbb{S} = \mathbb{S}_\ell \cdot \mathbb{S}_r$ и $|S| = 2$, тогда $k = 1$. Очевидно, что правила $\mathbb{S}_\ell, \mathbb{S}_r$ являются терминальными. Следовательно, алгоритм вернет корректный результат.

Шаг: Пусть для всех $\mathbb{S}_i$ таких, что $|S_i| < k$ алгоритм корректен. Докажем, что алгоритм корректно работает для любого правила $\mathbb{S}' = \mathbb{S}_\ell \cdot \mathbb{S}_r$, где $|S_\ell| < k, |S_r| < k$ и $|S'| > k$.

а) Пусть $k = |S_\ell|$, тогда алгоритм корректен так как $\mathbb{S}_\ell, \mathbb{S}_r$ – ПП такие, что их деревья вывода являются декартовыми деревьями по неявному ключу и $|S_\ell| = k, |S_r| = |S'| - k$.

б) Пусть $k > |S_\ell|$, тогда по предположению индукции вызов $split$ с параметрами $\mathbb{S}_r$ и $k - |S_\ell|$ вернет корректный результат. Пусть $(\mathbb{L}', \mathbb{R}') = split(\mathbb{S}_r, k - |S_\ell|)$, где $\mathbb{L}'$ выводит текст $S_r[0 \dots k - |S_\ell|]$, а $\mathbb{R}'$ выводит текст $S_r[k - |S_\ell| \dots |S_r|]$, тогда

ПП $\mathbb{S}_\ell \cdot \mathbb{L}'$ выводит текст $S'[0 \dots |S_\ell|] \cdot S'[|S_\ell| \dots k] = S'[0 \dots k]$, а $\mathbb{R}'$ выводит текст $S'[k \dots |S'|]$. Отсюда следует корректность алгоритма.

в) Случай $k < |S_\ell|$ аналогичен предыдущему случаю.

ОПЕРАЦИЯ: *merge*

ВХОД: ПП $\mathbb{S}_1, \mathbb{S}_2$, которые выводят тексты S_1 и S_2 соответственно;

ВЫХОД: ПП $\mathbb{S}$, которая выводит текст $S_1 \cdot S_2$;

АЛГОРИТМ: Аналогичен алгоритму для декартовых деревьев по неявному ключу.

КОРРЕКТНОСТЬ:

Докажем корректность индукцией по суммарному количеству узлов в деревьях вывода $\mathbb{S}_1$ и $\mathbb{S}_2$.

База: Пусть $|\mathbb{S}_1| = |\mathbb{S}_2| = 1$, тогда оба корня являются терминальными правилами и $count(\mathbb{S}_1) = count(\mathbb{S}_2) = 1$. Следовательно, $\mathbb{S}_1 \cdot \mathbb{S}_2$ искомая ПП.

Шаг: Зафиксируем числа $1 < k_1 < |\mathbb{S}_1|, 1 < k_2 < |\mathbb{S}_2|$. Пусть для всех ПП $\mathbb{S}_i, \mathbb{S}_j$ таких, что $|\mathbb{S}_i| < k_1, |\mathbb{S}_j| < k_2$ алгоритм корректно выполняет операцию $merge$. Докажем корректность операции $merge$ для двух ПП $\mathbb{S}_p = \mathbb{S}_{\ell_1} \cdot \mathbb{S}_{r_1}, \mathbb{S}_q = \mathbb{S}_{\ell_2} \cdot \mathbb{S}_{r_2}$, где $|\mathbb{S}_{\ell_1}| < k_1, |\mathbb{S}_{r_1}| < k_1, |\mathbb{S}_{\ell_2}| < k_2, |\mathbb{S}_{r_2}| < k_2$ и $|\mathbb{S}_p| > k_1, |\mathbb{S}_q| > k_2$. Рассмотрим следующие возможные случаи:

а) Пусть $0 < r \leqslant k_1$. Предположению индукции все ПП $\mathbb{S}_{\ell_1}, \mathbb{S}_{r_1}, \mathbb{S}_{\ell_2}, \mathbb{S}_{r_2}$ являются непустыми и вызов функции $merge(\mathbb{S}_{r_1}, \mathbb{S}_q)$ корректен. То есть $\mathbb{S}^* = merge(\mathbb{S}_{r_1}, \mathbb{S}_q)$ является ПП, которая выводит текст $S^* = S_{r_1} \cdot S_q$ и ее дерево вывода является декартовой ПП с неявным ключом. Следовательно, ПП $\mathbb{S}_{\ell_1} \cdot \mathbb{S}^*$ является ПП, которая выводит текст $S_{\ell_1} \cdot S^* = S_p \cdot S_q$ и ее дерево вывода является декартовой ПП с неявным ключом.

б) Случай $k_1 < r \leqslant k_1 + k_2$ аналогичен предыдущему.

в) Пусть $r = k_1 + k_2 + 1$, тогда очевидно, что $\mathbb{S}_p \cdot \mathbb{S}_q$ выводит текст $S_p \cdot S_q$ и дерево вывода этой ПП является декартовым деревом с неявным ключом.

В итоге получаем следующий алгоритм построения ПП:

ВХОД: текст S и его LZ-факторизация $F_1, F_2, \dots, F_k$.

ВЫХОД: ПП $\mathbb{S}$, которая выводит текст S.

АЛГОРИТМ:

Алгоритм строит ПП по индукции.

База: Полагаем $\mathbb{S}$ равным терминальному правилу, выводящему $S[0]$.

Шаг: Зафиксируем произвольное число $0 < i < k$. Пусть $\mathbb{S}$ – ПП, которая выводит $F_1, F_2, \ldots, F_i$. Так как LZ-факторизация S фиксирована, тогда мы знаем позиции ℓ, r начала и конца вхождения F_{i+1} внутри $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ соответственно. Построим $\mathbb{F}_{i+1}$ следующим образом: $\mathbb{F}_{i+1} = split(split(\mathbb{S}, \ell)[1], r - \ell)[0]$. Наконец положим $\mathbb{S} = merge(\mathbb{S}, \mathbb{F}_{i+1})$.

Сложность: Всего в алгоритме k шагов. На каждом шаге запускается не более двух раз операция $split$ и не более одного раза операция $merge$. Следовательно, сложность каждого шага алгоритма $O(\log n)$. В итоге получаем, что сложность алгоритма построения ПП на основе декартовых деревьев $O(k \log n)$. Размер декартова дерева будет $O(k \log n)$.

Практические результаты

Условия проведения практических исследований

Очевидно, что природа исходного текста сильно влияет на время и степень сжатия. Поэтому в работе проводятся исследования на следующих типах текста:

– последовательности ДНК, взятые из открытого банка ДНК Японии (`http://www.ddbj.nig.ac.jp/`);

– строки Фибоначчи;

– произвольные строки над четырехбуквенным алфавитом.

Эти типы текстов были выбраны не случайно. Строки Фибоначчи являются одним из лучших примеров входных данных для задачи построения ПП. На этом классе строк мы можем оценить практический потенциал ПП как модели сжатого представления. Произвольные строки считаются не сжимаемыми и потенциально они являются наихудшим входом для задачи построения ПП. Последовательности ДНК это класс хорошо сжимаемых строк, используемых на практике.

В работе сравниваются алгоритмы построения ПП с классическими алгоритмами сжатия из семейства Лемпеля-Зива. Наш тестовый набор алгоритмов содержит классическую версию алгоритма Лемпеля-Зива из [6]. В дальнейшем для этого алгоритма мы используем следующее обозначение: **lz77**. Размер окна сжатия равен 32Kb. Читателю может показаться, что размер 32Kb слишком мал

для практических целей, но это значение было выбрано с целью подчеркнуть влияние размера окна на степень сжатия.

Очевидно, размер окна сжатия влияет как на длины отдельных факторов, так и на размер факторизации в целом. Поэтому если мы хотим получать более компактную факторизацию, то необходимо тратить больше ресурсов на подготовительном этапе. Иначе говоря, размер окна сжатия напрямую влияет на размер факторизации. В тоже время не существует прямой взаимосвязи между устройством ПП и размером текста выводимым из нее. Поэтому использование компактных LZ-факторизаций влечет построение более компактных ПП. Следовательно, интересно рассмотреть версию алгоритма Лемпеля-Зива с бесконечным окном сжатия. В дальнейшем для этого алгоритма мы используем следующее обозначение: **lzma**.

Дополнительно тестовый набор алгоритмов содержит алгоритм Лемпеля-Зива-Велча (краткое обозначение **lzw**).

Для удобства читателя были приняты общие обозначения для алгоритмов сжатия, которые представлены на рисунке 0.3.

Рис. 0.3. Обозначения алгоритмов сжатия

- ▫ – **lz77** (алгоритм Лемпеля-Зива с 32Kb окном сжатия)
- ▪ – **lzma** (алгоритм Лемпеля-Зива с бесконечным окном сжатия)
- △ – **lzw** (алгоритм Лемпеля-Зива-Велча)
- ○ – **AVLclassic** (алгоритм построения ПП на основе AVL-грамматик, представленный в [1])
- • – **AVLoptimal** (алгоритм представленный в главе 3.2)
- ▲ – **Cartesian** (алгоритм представленный в главе 3.3)

Эффективность алгоритмов оценивается относительно двух параметров: время работы и степень сжатия. Для алгоритмов построения ПП с помощью AVL-деревьев дополнительно считается количество операций вращения. Степень сжатия определяется как отношение между размером сжатого представления и размером исходного представления текста. Степень сжатия измеряется в процентах. Например, формула для степени сжатия с помощью ПП выглядит следующим образом: $\frac{|\mathbb{S}|}{|S|} \cdot 100$. Поскольку отношение между бинарным представлением правила ПП и бинарным представлением LZ-фактора превосходит 1 (но фиксирован), то может возникнуть следующая ситуация: степень сжатия с по-

мощью ПП меньше чем степень сжатия с помощью алгоритма Лемпеля-Зива, но бинарное представление ПП превосходит бинарное представление LZ-словаря.

Все алгоритмы запускались в одинаковых условиях на компьютере со следующей спецификацией: процессов Intel(R) Core(TM) i7-2600 CPU 3.4GHz, 8Гб оперативной памяти, ОС Windows 7 x64.

Сравнительный анализ алгоритмов построения ПП

В этом разделе представлен сравнительный анализ трех алгоритмов построения ПП.

Как ожидалось, все алгоритмы работают чрезвычайно быстро на строках Фибоначчи и строят очень компактные ПП. Например, 36-е слово Фиббоначчи размера порядка 40Мб обрабатывается в течение 1мс, а на выходе получается ПП размера около 100 правил. Этот факт доказывает, что существует класс строк, на котором модель сжатия с помощью ПП чрезвычайно эффективна. В дальнейшем мы не будем обсуждать результаты полученные на строках Фибоначчи.

Рисунок 0.4 показывает как меняется количество операций вращения после применения оптимизации к алгоритму Риттера [1]. Даже для входных данных небольших размеров количество операций вращения уменьшилось около 10 раз.

Для того, чтобы оценить влияние операций вращения на скорость построения ПП были проведены два теста. В первом тесте алгоритмы хранили построенное AVL-дерево в оперативной памяти, а во втором – во внешнем файле (то есть каждое обращение к дереву является обращением к файловой системе). Мы ожидаем, что при хранении ПП в оперативной памяти не будет принципиальной разницы между алгоритмом Риттера и его оптимизированной версией, а алгоритм на основе декартовых деревьев будет быстрее. Но как алгоритмы будут себя вести, если стоимость доступа к AVL-дереву будет дороже вычислений в оперативной памяти? Такая ситуация естественным образом возникает в случае, когда размер входного текста очень велик и алгоритм не может хранить все дерево в оперативной памяти. На рисунках 0.5 и 0.6 представлены результаты обоих тестов.

Рис. 0.4. Статистика количества операций вращения AVL-дерева для ДНК)

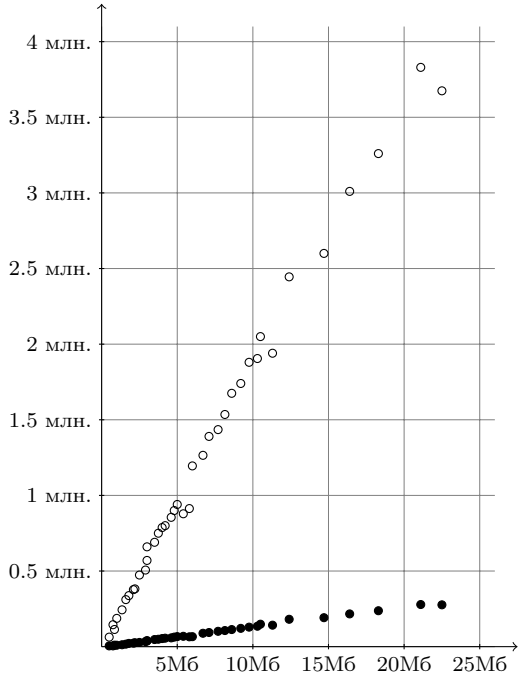
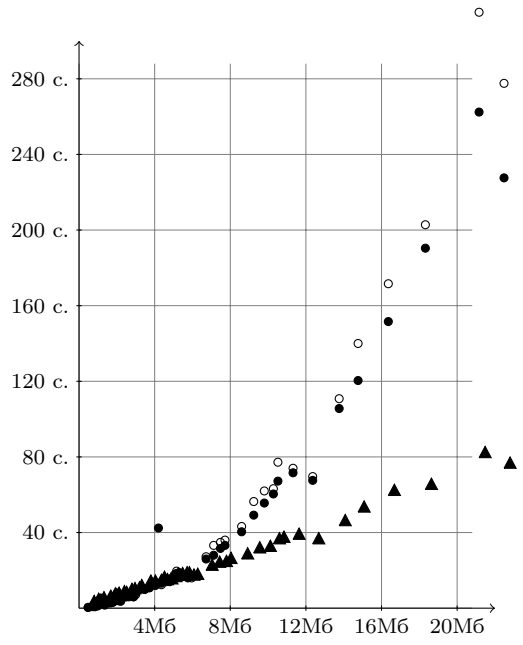
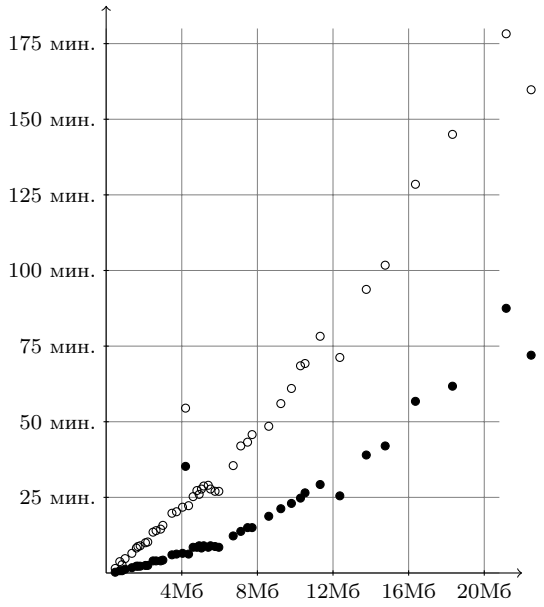


Рис. 0.5. Статистика времени работы алгоритмов построения ПП на ДНК (ПП хранится в оперативной памяти)



На рисунке 0.7 представлена статистика высот ПП полученных всеми алгоритмами на последовательностях ДНК. Высоты ПП полученные с помощью классического алгоритма Риттера и с помощью его потимизированного варианта очень близки, поэтому на рисунке предтавлены результаты только одного алгоритма.

Рис. 0.6. Статистика времени работы алгоритмов построения ПП на ДНК (ПП хранится в отдельном файле)



Аналогичные эксперименты были проведены на произвольных текстах. Результаты этих исследований приведены на рисунках 0.8, 0.9 и 0.10.

На рисунке 0.11 представлена статистика высот ПП полученных всеми алгоритмами на произвольных текстах.

Результаты исследования показывают, что полученная оптимизированная версия алгоритма Риттера работает немного быстрее при хранении ПП в оперативной памяти. А при хранении дерева в оперативной памяти тесты показывают, что новый алгоритм работает в 2-3 раза быстрее чем классическая версия алгоритма Риттера. Таким образом оптимизированный алгоритм более устойчив к росту входных данных. В тоже время алгоритм, использующий декартовы деревья, работает в 3-4 раза быстрее чем алгоритмы, которые используют AVL-деревья. А высота декартова дерева в среднем в два раза больше чем высота AVL-дерева.

Рис. 0.7. Статистика высот ПП на ДНК

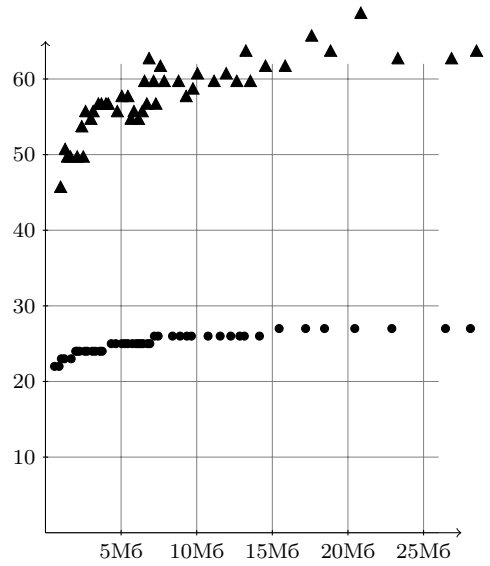


Рис. 0.8. Статистика количества операций вращения AVL-дерева для произвольных текстов)

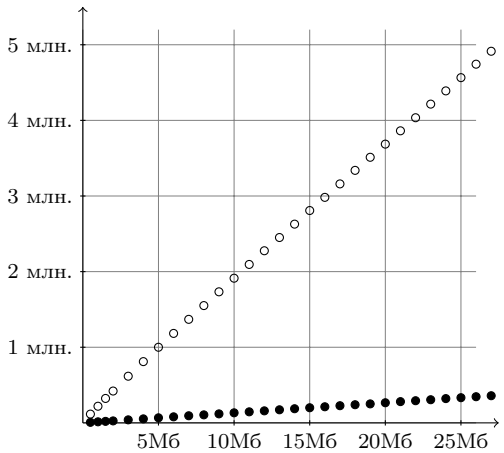


Рис. 0.9. Статистика времени работы алгоритмов построения ПП на произвольных текстах (ПП хранится в оперативной памяти)

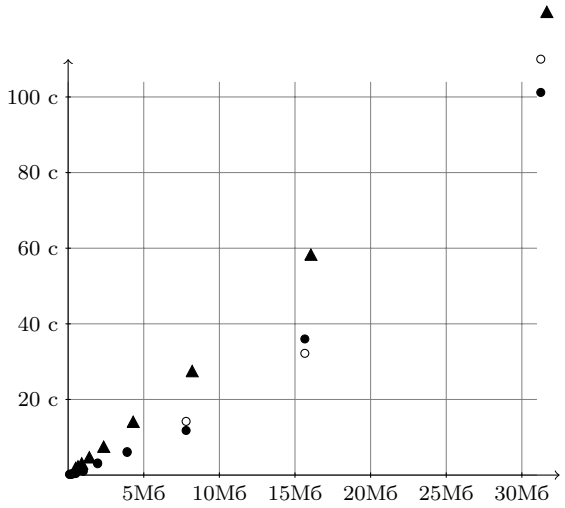
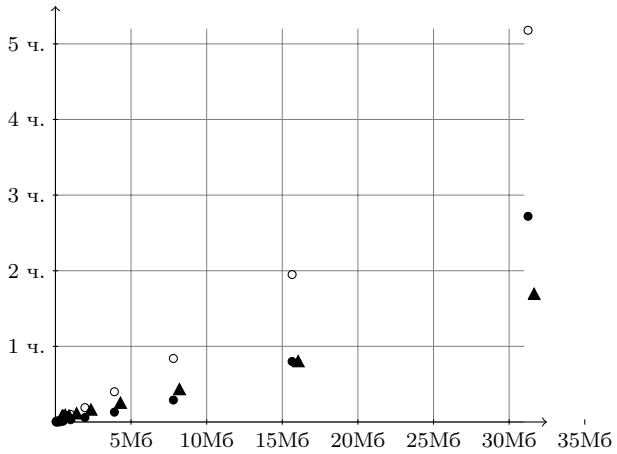


Рис. 0.10. Статистика времени работы алгоритмов построения ПП на произвольных текстах (ПП хранится в отдельном файле)



Сравнительный анализ степени сжатия

Рисунки 0.12 и 0.13 показывают, что классический алгоритм Риттера и его оптимизированная версия обеспечивают очень степень сжатия близкую к степени сжатия классических алгоритмов (LZ77 и LZW). Как и ожидалось алгоритм на декартовых деревьях обеспечивает более плохую степень сжатия. При этом алгоритмы сжатия на основе AVL-деревьев обеспечивают степень сжатия, которая не более чем в два раза хуже алгоритмов LZ77 и LZW. А в результате алгоритмы сжатия на основе ПП строят хорошо-структурированное представление данных, которое поддерживает поисковые запросы.

Архитектура проекта

Код проекта находится в свободном доступе. Скачать исходный код проекта можно по адресу `http://overclocking.googlecode.com/svn/webService/trunk/`.

В проекте были использованы следующие внешние библиотеки:

– **log4j** (`http://logging.apache.org/`) – библиотека для логирования событий, происходящих в системе;

– **simple xml serializer** (`http://simple.sourceforge.net/`) – библиотека для сериализации/десериализации объектов в формате xml;

– **container** (`http://overclocking.googlecode.com/svn/webService/trunk/IDEContainer/`) – реализация DI контейнера;

– **cassandra** (`http://cassandra.apache.org/`) – распределенная файловая система;

– **hector** (`https://github.com/rantav/hector`) – клиент для распределенной файловой системы;

Архитектура проекта состоит из следующих компонентов:

а) Кластер распределенной файловой системы cassandra. Кластер состоит из трех узлов и обеспечивает отказоустойчивость и доступность данных;

б) Набор сервисов сжатия данных. Для каждого алгоритма сжатия данных написан свой сервис. Каждый сервис сжатия данных отслеживает изменения в распределенной файловой системе. А именно, сервис отслеживает появление

Рис. 0.11. Статистика высот ПП на произвольных текстах

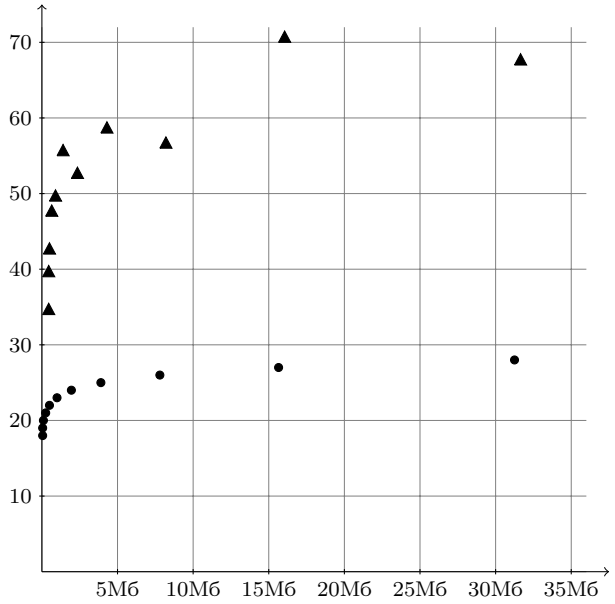


Рис. 0.12. Статистика степени сжатия на ДНК

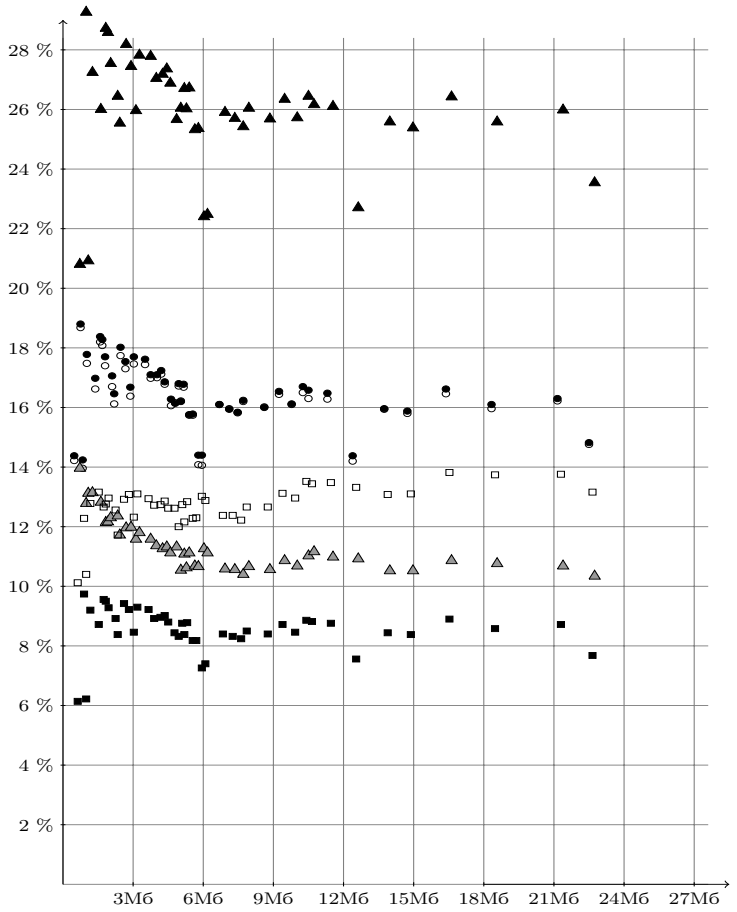
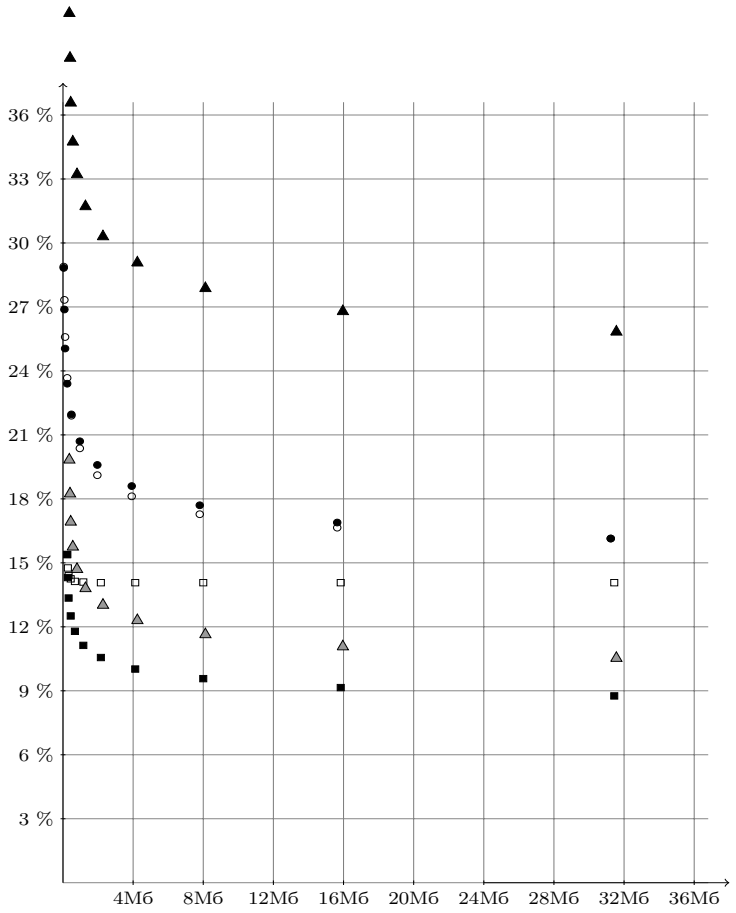


Рис. 0.13. Статистика степени сжатия на произвольных текстах



новых текстов в системе. Как только появляется новый текст запускается задача по сжатию этого текста. В процессе сжатия текста собирается статистика о результатах сжатия данных. Далее полученная статистика сохраняется в распределенной файловой системе.

в) Сервис публикации данных – сервис, который публикует свежую статистику в открытый доступ. А именно это сервис, который интегрируется с сайтом, на котором опубликована статистика по сжатию данных. Сайт расположен по адресу `http://overclocking.usu.edu.ru`.

Реализация алгоритмов сжатия

В этом разделе мы представим основные трудности с которыми пришлось столкнуться во время реализации алгоритмов сжатия.

Основным объектом с которым работают алгоритмы сжатия на основе ПП является факторизация текста. Но как получать эту факторизацию? Существует большое число способов разбить текст на части. В проекте реализованы два способа факторизации текста. Обе факторизации соответствуют естественному разбиению текста получаемому алгоритмами сжатия LZ77 и LZMA. Реализация алгоритмов факторизации различна, так как факторизация на основе LZ77 в каждый момент времени просматривает ограниченый участок текста, в то время как LZMA просматривает весь доступный текст.

В процессе факторизации необходимо быстро отвечать на поисковые запросы, а именно, содержится ли некоторая подстрока в рабочей области. Для решения таких задач существуют две похожие структуры данных: суффиксный массив и суффиксное дерево. Обе структуры данных поддерживают поисковые запросы, но отличаются по способу хранения данных. Так каждый узел суффиксного дерева хранит ссылки на своих потомков, следовательно дополнительно расходуется память на хранение ссылок. Более того, если память под данные выделяется каким-то универсальным механизмом, то потенциально ссылки на потомком могут быть раскиданы произвольно по оперативной памяти. Следовательно, при работе с деревом возникнет большее количество переходов в памяти, что негативно сказывается на работе кэша памяти. Так же потенциально число потомков некоторого узла равно размеру алфавита, следовательно надо организовывать работу с большим числом потомком. А это также требует ре-

сурсы оперативной памяти. С другой стороны работа с суффиксным деревом более удобна. Из представленных аргументов следует, что суффиксные деревья больше подходят для решения небольших (локальных) задач.

Поскольку алгоритм LZ77 использует рабочее окно фиксированного размера, то для построения факторизации используются суффиксные деревья. Алгоритм построения LZ77-факторизации состоит из следующих шагов:

– Один раз алгоритм строит суффиксное дерево на основе рабочей области;

– На основе запросов к суффиксному дереву алгоритм находит максимальный фактор, который уже содержится в рабочей области;

– После этого сдвигается рабочая область и соответственно перестраивается суффиксный массив. Необходимо подчеркнуть, что суффиксный массив не строится заново, а именно локально модифицируется. За счет локальной модификации дерева достигается большая эффективность алгоритма.

Поскольку алгоритм LZMA не накладывает никаких ограничений на рабочую область и размеры входа могут быть достаточно большими, то для построения LZMA-факторизации более удобен суффиксный массив. Алгоритм построения LZMA-факторизации содержит следующие шаги:

– Алгоритм строит один суффиксный массив для всей входной строки;

– Последовательно просматривая входную строку слева направо алгоритм разбивает строку на факторы, обращаясь при этом с запросами к суффиксному массиву;

При реализации алгоритмов сжатия на основе ПП мы столкнулись с тем, что алгоритмы построения больших деревьев очень ресурсоемки. Например, если размер исходного текста равен 10 Мб, то алгоритм сжатия на основе ПП может потребовать порядка 3-4 Гб оперативной памяти. Такое поведение связано с постоянным перестроением дерева и поддержанием его сбалансированности во время обработки очередного фактора. А именно, в процессе перебалансировки дерева могут порождаться и новые узлы, и узлы на которые больше никто не ссылается. Так как скорость построения дерева велика, то рассчитывать, что на то, что сборщик мусора (garbage collector) соберет неиспользуемые ссылки не приходится. Поэтому все алгоритмы сжатия содержат модуль управления памя-

тью. Более того, такой модуль необходим если алгоритм сжатия хранит сжатое представление сразу в файловой системе.

Для каждого узла дерева алгоритм дополнительно хранит количество внешних ссылок на этот узел. При выполнения операций, которые модифицируют дерево алгоритм изменяет число ссылок на правило. Если число ссылок на узел стало равно нулю, то он помечается как свободный. Другими словами мы не удаляем правила, которые были хотя бы раз использованы, а помечаем их как правила, которые можно полностью менять и при этом не испортится уже построенное дерево.

После реализации представленной выше модели управления узлами потребности алгоритмов сжатия в оперативной памяти сократились от трех до восьми раз.

Методология разработки

В нашем проекте для ведения разработки используется методология Agile. Agile это методология разработки, которые основываются на последовательном (итеративном) ведении процесса разработки. Одной из целей этой методологии является минимизация рисков проекта за счет регулярного получения обратной связи и пересмотра планов. В рамках этой методологии сформулированы различные техники, которые позволяют достичь желаемого результата. Понятно, что в зависимости от проекта и от команды, которая разрабатывает проект, какие-то техники могут быть более эффективны, а какие-то менее. Нашему проекту больше подходят техники экстремального программирования, которые сформулированы в [13], [14] и [15].

В рамках нашего проекта актуальность техник экстремального программирования сложно недооценить. Потому, что допущенная на раннем этапе ошибка будет очень дорого стоить на этапе использования. Например, если запущен алгоритм сжатия данных, который хранит сжатое представление в файловой системе, то он может работать в течение нескольких часов и при этом будут записаны мегабайты логов. Пусть перед самым концом алгоритм завершился исключением, тогда уже потеряно несколько часов работы зря, получено большое количество логов, которые необходимо анализировать, чтобы понять причину ошибки. Техники экстремального программирования позволяют свести вероят-

ность такой ситуации к минимуму. Существуют различные техники экстремального программирования (> 10), в нашей команде используются следующие:

а) разработка через тестирование;

б) игра в планирование;

в) парное программирование;

г) непрерывная интеграция;

д) рефакторинг;

е) частые релизы;

ж) простой дизайн;

з) коллективное владение кодом;

и) стандарт кодирования;

Рассмотрим каждую технику в отдельности, чтобы понять какую цель преследует каждая из техника.

Разработка через тестирование

Выделяются следующие виды тестов:

– модульные тесты (фиксируют взаимодействие отдельного модуля (класса) с внешним миром):

– интеграционные (фиксируют базовые сценарии поведения некоторого блока, например, интеграционные тесты на взаимодействие с базой данных);

– функциональные (фиксируют работу функционала из пользовательского интерфейса);

– приемочные (фиксируют работоспособность бизнес процессов);

Поскольку пользовательский интерфейс проекта еще достаточно невелик, то в проекте используются только модульные и интеграционные тесты. Что дает проекту тестирование?

Разработчик не может быть уверен в правильности написанного им кода до тех пор, пока не сработают абсолютно все тесты. Модульные тесты позволяют разработчикам за очень короткое время убедиться в том, что их код работает корректно. Модульные позволяют разработчику без каких-либо опасений выполнять рефакторинг проекта. Также они помогают другим разработчикам понять,

зачем нужен тот или иной фрагмент кода и как он функционирует. Фактически модульные тексты можно использовать в качестве проектной документации.

Интеграционные тесты гарантируют, что довольно большие участки проекта работоспособны. А если интеграционный тест находит ошибку, то эта ошибка локальна и не надо тратить время на то, чтобы определить в каком блоке проекта произошел сбой.

В рамках экстремального программирования используется подход, называемый Test Driven Development – сначала пишется тест, который не проходит, затем пишется код, чтобы тест прошел, а в заключении делается рефакторинг кода.

Игра в планирование

Основная цель игры в планирование – сформировать приблизительный план работы на ближайшую итерацию и постоянно обновлять его по мере того, как условия задачи становятся все более четкими. В этой игре участники делятся на две команды. В первой команде находятся члены команды разработки, а во второй представители внешнего мира (сотрудники, которые исследуют потребности клиентов). Вначале представители внешнего мира формируют набор задач, который упорядочен по степени важности с точки зрения клиента. Потом участники второй команды подробно рассказывают о каждой задаче. А участники первой команды в процессе обсуждения формируют стоимость задачи. Если задача оказывается слишком дорогой участники второй команды могут убрать задачу из плана. Критическим фактором, благодаря которому такой стиль планирования оказывается эффективным, является то, что в данном случае внешний мир отвечает за принятие бизнес-решений, а команда разработчиков отвечает за принятие технических решений.

Парное программирование

Парное программирование предполагает, что большая часть кода создается парами программистов, работающих за одним компьютером. Один из них работает непосредственно с текстом программы, другой просматривает его работу и следит за общей картиной происходящего. При необходимости клавиатура свободно передается от одного к другому. В течение работы над проектом пары не фиксированы: рекомендуется их перемешивать, чтобы каждый программист в команде имел хорошее представление о всей системе. Таким образом, парное

программирование усиливает взаимодействие внутри команды, улучшает качество решения задачи, способствует обмену знаниями внутри команды.

Непрерывная интеграция

Если выполнять интеграцию разрабатываемой системы достаточно часто, то можно избежать большого количества проблем. В традиционных методиках интеграция, как правило, выполняется в самом конце работы над продуктом, когда считается, что все составные части разрабатываемой системы полностью готовы. В экстремального программирования интеграция кода всей системы выполняется несколько раз в день, после того, как разработчики убедились в том, что все модульные тесты корректно работают. Это практика позволяет в очень сжатые сроки собрать работающую версию продукта. Например, это актуально, когда необходимо выложить обновление проекта.

Рефакторинг

Рефакторинг —- это методика улучшения кода, без изменения его функциональности. Предполагается, что однажды написанный код в процессе работы над проектом почти наверняка будет неоднократно переделан. Разработчики переделывают написанный ранее код для того, чтобы улучшить архитектуру проекта и снизить стоимость поддержки продукта. Этот процесс называется рефакторингом. Отсутствие тестового покрытия провоцирует отказ от рефакторинга, в связи с боязнью сломать функционал. Результатом отказа от рефакторинга является постепенная деградация кода.

Частые релизы

Версии продукта должны поступать в эксплуатацию как можно чаще. Работа над каждой версией должна занимать как можно меньше времени. При этом каждая версия должна быть достаточно осмысленной с точки зрения полезности для внешнего мира.

Чем раньше мы выпустим первую рабочую версию продукта, тем раньше клиент начнет получать необходимую функциональность. Чем раньше клиент приступит к эксплуатации продукта, тем раньше разработчики получат от него информацию о том, что соответствует требованиям заказчика, а что — нет. Эта информация может оказаться чрезвычайно полезной при планировании следующего выпуска.

Простота дизайна

В концепции экстремального программирования предполагается, что в процессе работы условия задачи могут неоднократно измениться, а значит, разрабатываемый продукт не следует проектировать заблаговременно целиком и полностью. Если в самом начале работы вы пытаетесь от начала и до конца детально спроектировать систему, вы напрасно тратите время. Проектирование —- это настолько важный процесс, что его необходимо выполнять постоянно в течение всего времени работы над проектом. Проектирование должно выполняться небольшими этапами, с учетом постоянно изменяющихся требований. В каждый момент времени мы пытаемся использовать наиболее простой дизайн, который подходит для решения текущей задачи. При этом мы меняем его по мере того как условия задачи меняются.

Стандарты кодирования

Все члены команды в ходе работы должны соблюдать требования общих стандартов кодирования. Благодаря этому члены команды не тратят время на споры о вещах, которые фактически никак не влияют на скорость работы над проектом и обеспечивается эффективное выполнение остальных практик.

Если в команде не используются единые стандарты кодирования, разработчикам становится сложнее выполнять рефакторинг. При смене партнеров в парах возникает больше затруднений. Следовательно, продвижение проекта затрудняется. Поэтому необходимо добиться того, чтобы было сложно понять, кто является автором того или иного участка кода, —- вся команда работает унифицировано, как один человек. Команда должна сформировать набор правил, а затем каждый член команды должен следовать этим правилам в процессе кодирования. Перечень правил не должен быть исчерпывающим или слишком объемным. Задача состоит в том, чтобы сформулировать общие указания, благодаря которым код станет понятным для каждого из членов команды. Стандарт кодирования поначалу должен быть простым, затем он будет эволюционировать по мере того, как команда обретает опыт. Не следует тратить на предварительную разработку стандарта слишком много времени.

Коллективное владение

Коллективное владение означает, что каждый член команды несёт ответственность за весь исходный код. Таким образом, каждый вправе вносить изменения в любой участок программы. Парное программирование поддерживает

эту практику: работая в разных парах, все программисты знакомятся со всеми частями кода системы. Важное преимущество коллективного владения кодом заключается в том, что оно ускоряет процесс разработки, поскольку при появлении ошибки её может устранить любой программист.

Давая каждому программисту право изменять код, мы получаем риск появления ошибок, вносимых программистами, которые считают что знают что делают, но не рассматривают некоторые зависимости. Хорошо определённые модульные тесты решают эту проблему: если нерассмотренные зависимости порождают ошибки, то следующий запуск модульных тестов будет неудачным.

Заключение

В работе представлены два новых алгоритма сжатия на основе ПП. Первый алгоритм очень похож на классический алгоритм Риттера из [1], но за счет оптимизации более устойчив к росту входных данных. Второй алгоритм использует более простую (с точки зрения построения) структуру дерева и поэтому более эффективен по времени построения. Алгоритм на основе декартовых деревьев строит более высокие ПП, чем алгоритмы на основе AVL-деревьев.

В работе представлен сравнительный анализ алгоритмов сжатия на основе ПП и классических алгоритмов сжатия. Результаты показывают, что алгоритмы построения ПП ресурсоемки и уступают классическим алгоритмам по времени сжатия. С другой стороны степень сжатия, достигаемая алгоритмами на основе ПП, близка к степени сжатия, достигаемой алгоритмами из семейства Лемпеля-Зива. Также высота полученного сжатого представления получается небольшой.

Полученные алгоритмы сжатия могут быть использованы на практике, но для различных целей. Сравнительный анализ показал, что оптимизированный алгоритм Риттера больше подходит для построения компактных ПП, рассчитанных на дальнейший поиск. А алгоритм на основе декартовых деревьев больше подходит для практических нужд, так как демонстрирует высокую скорость работы. Нам кажется, что алгоритмически ускорить построение ПП очень трудно. Однако, на данный момент алгоритм сжатия на основе декартовых деревьев показывает очень хорошую скорость, а вычислительная техника становится мощнее с каждым годом. Больший интерес представляет вопрос о том насколько большой сжимающий потенциал у ПП. В данной работе представлена эвристика на основе которой строится ПП, но это не значит что не существует другой ПП меньшего размера, выводящий исходный текст. Также интересно сравнить эффективность поисковых алгоритмов на ПП и на строках. А именно, на каких объемах входных данных алгоритмы на ПП смогут обогнать классические строковые алгоритмы.

Статья, содержащая улучшенный алгоритм Риттера, была представлена на международной конференции 1st International Conference on Data Compression, Communication and Processing `http://ccp2011.dia.unisa.it/CCP_2011/Home.html` и принята к печати в сборник работ коференции. Статья, содержащая ал-

горитм построения ПП на основе декартовых деревьев, была отправлена на рецензию в журнал Записки научных семинаров ПОМИ http://www.pdmi.ras.ru/znsl/.

Список использованных источников

1. W. Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

2. T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 1(298):253–272, 2003.

3. Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *CPM*, volume 1645 of *Lecture Notes in Computer Science*, pages 37–49. Springer, 1999.

4. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

5. T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

6. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

7. Y. Lifshits. Processing compressed texts: A tractability border. In *CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2007.

8. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Computing longest common substring and all palindromes from compressed strings. In *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 2008.

9. A. Tiskin. Faster subsequence recognition in compressed strings. *CoRR*, abs/0707.3407, 2007.

10. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

11. D. Knuth. *The Art of Computing, Vol. III Second edition*. Addison-Wesley, 1998.

12. T. Hagerup and C. Rüb. A guided tour of chernoff bounds. *Inf. Process. Lett.*, 33(6):305–308, 1990.

13. К. Бек. *Экстремальное программирование*. Питер, 2003.

14. М. Фаулер К. Бек. *Экстремальное программирование: планирование*. Питер, 2003.

15. Р. Миллер К. Ауэр. *Экстремальное программирование: постановка процесса. С первых шагов и до победного конца*. Питер, 2004.