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

Институт математики и компьютерных наук
Департамент математики, механики и компьютерных наук

Кафедра алгебры и дискретной математики

Классификация алгоритмов построения прямолинейных программ

Допустить к защите:

Зав. кафедрой
д. ф.-м. н.,
профессор Волков М. В.

Магистерская диссертация
студента 2 курса
Курпилянского Е. Б.

Научный руководитель:
Хворост А. А.

Екатеринбург
2014 год

Содержание

1 Введение

Представление данных в сжатом виде позволяет экономить место, требуемое для их хранения. Тем не менее для какой-либо обработки сжатых данных приходится тратить время на их распаковку, а затем работать с несжатым представлением. Один из возможных подходов к решению указанной проблемы состоит в разработке алгоритмов, оперирующих непосредственно со сжатыми представлениями. Имеется много разных методов сжатого представления данных, для которых уже разработаны такие алгоритмы: коллаж-системы [1], представления с помощью антисловарей [2], прямолинейные программы [3] и др.

Ясно, что алгоритмы, работающие со сжатыми представлениями, существенным образом зависят от механизма сжатия. Сжатие текста с помощью контекстно свободных грамматик (таких, как прямолинейные программы) выделяется среди прочих методов двумя обстоятельствами. Во-первых, грамматики обеспечивают хорошо структурированное сжатое представление, что удобно для последующей алгоритмической обработки. Во-вторых, сжатие в виде прямолинейных программ полиномиально эквивалентно сжатию данных с помощью широко применяемых на практике алгоритмов из семейства алгоритмов Лемпеля-Зива (таких, как LZ77 [4], LZ78 [5], LZW [6]). Полиномиальная эквивалентность здесь понимается в следующем смысле: существует полиномиальная зависимость между размером прямолинейной программы, выводящей данный текст S, и размером словаря, построенным алгоритмом Лемпеля-Зива для S, см. [3].

Существует довольно большой класс задач, для которых разработаны алгоритмы со временем работы, полиномиальным относительно размера прямолинейной программы. К этому классу относятся, например, задачи **Поиск образца в тексте** [7], **Наибольшая общая подстрока** [8], считающая версия задачи **Поиск всех палиндромов** [8], некоторые версии задачи **Наибольшая общая подпоследовательность** [9]. В то же время константы, которые скрываются за «О большим» в имеющихся оценках сложности таких алгоритмов, как правило, очень велики. Кроме того, упо-

мянутая выше полиномиальная связь между размером прямолинейной программы, выводящей данный текст S, и размером LZ77-словаря для S еще не гарантирует, что прямолинейная программа на практике обеспечивает достаточно высокую степень сжатия. Поэтому вопрос о том, существуют ли методы сжатия, основанные на прямолинейных программах и подходящие для практического применения, требует дополнительного исследования.

Известно, что задача построения прямолинейной программы минимального размера NP-полна [10]. Именно поэтому любой из практически применимых алгоритмов позволяет построить прямолинейную программу, являющую лишь неким приближением к минимальной.

При выполнении данной работы были поставлены следующие цели:

- представить обзор алгоритмов построения прямолинейной программы;
- сравнить рассмотренные алгоритмы по двум основным параметрам (время сжатия и коэффициент сжатия) на текстах разной природы;
- по результатам сравнения провести классификацию алгоритмов.

В данной работе представлен обзор следующих алгоритмов построения прямолинейной программы:

- классический алгоритм, предложенный Риттером [3];
- пара эвристических модификаций алгоритма Риттера, направленных на ускорение работы алгоритма [11];
- алгоритм Риттера, в котором в качестве основной структуры данных вместо AVL-деревьев используются рандомизированные деревья [12];
- многопоточный алгоритм, основанный на идеях алгоритма Риттера;
- LCA-online[1] алгоритм, предложенный в [13].

В первых четырех из перечисленных алгоритмов в качестве основной структуры данных используются сбалансированные бинарные деревья, а

[1]Автору неизвестно общепринятое название этого алгоритма в русскоязычной литературе. LCA (сокращение от Lowest common ancestor) переводится как «наименьший общий предок». Слово online в названии алгоритма означает, что символы обрабатываются последовательно слева направо и не требуется хранение всего текста в оперативной памяти.

размер построенной прямолинейной программы гарантированно не превосходит размер минимальной в $O(\log n)$, где n — длина текста. Последний алгоритм позволяет построить лишь $O(\log^2 n)$-приближение, зато не требует использования дополнительных структур данных и работает за линейное время от размера текста.

В работе предложены две новые идеи: вторая эвристическая оптимизация алгоритма Риттера (см. §3.2) и многопоточный алгоритм построения прямолинейной программы (см. §5). Также в §8 приведены результаты сравнения перечисленных алгоритмов построения прямолинейной программы по двум основным (время сжатия и коэффициент сжатия), а также еще нескольким вспомогательным параметрам. Практические эксперименты показали неожиданный результат: эффективность сжатия LCA-online алгоритма на практике оказалась выше, чем у алгоритмов из класса, использующих сбалансированные бинарные деревья.

2 Обозначения и основные понятия

Через $P\{A\}$ обозначается вероятность наступления события A, а через $P\{A \mid B\}$ — вероятность наступления события A при условии наступления события B.

В работе рассматриваются строки над конечным алфавитом Σ. *Длина* строки S равна числу символов из Σ в S и обозначается через $|S|$. Через $S{\cdot}S'$ обозначается *конкатенация* двух строк S и S', а через $S[i..j]$ — подстрока строки S с позиции i до позиции j включительно.

LZ-факторизация строки S — это факторизация $S = w_1 \cdot w_2 \cdot \ldots \cdot w_k$ такая, что для любого $j \in 1..k$

- w_j состоит из одной буквы, не встречающейся в $w_1 \cdot w_2 \cdot \ldots \cdot w_{j-1}$; или
- w_j — наибольший префикс $w_j \cdot w_{j+1} \cdot \ldots \cdot w_k$, встречающийся в $w_1 \cdot w_2 \cdot \ldots \cdot w_{j-1}$.

Прямолинейная программа (кратко ПП) – это последовательность правил вывода вида:

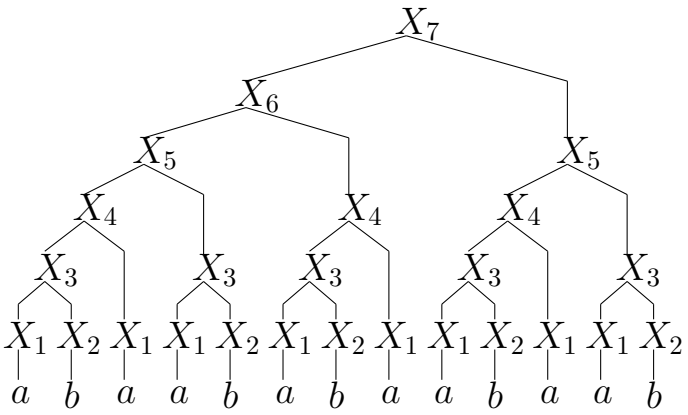

Рис. 1: Дерево вывода грамматики, порождающей «*abaababaabaab*».

$$X_1 \to expr_1, \quad X_2 \to expr_2, \quad \ldots, \quad X_n \to expr_n,$$

где X_i – это переменные, а $expr_i$ – выражения вида:

- $expr_i$ – символ из алфавита Σ (такие правила будем называть терминальными), или
- $expr_i \to X_l \cdot X_r (l, r < i)$ (такие правила будем называть нетерминальными), где «$\cdot$» обозначает конкатенацию правил X_l и X_r.

Таким образом, ПП – это контекстно свободная грамматика в нормальной форме Хомского, порождающая в точности одну строку над алфавитом Σ.

Пример: Рассмотрим ПП $\mathcal{X}$, которая порождает текст «*abaababaabaab*»:

$$X_1 \to a, \quad X_2 \to b, \quad X_3 \to X_1 \cdot X_2, \quad X_4 \to X_3 \cdot X_1,$$
$$X_5 \to X_4 \cdot X_3, \quad X_6 \to X_5 \cdot X_4, \quad X_7 \to X_6 \cdot X_5$$

Дерево вывода этой грамматики изображено на рисунке 1.

Для некоторого дерева вывода ПП T будем обозначать $S(T)$ — вывод этой ПП. Высотой дерева будем называть длину самого длинного пути от корня до какого-то листа.

Замечание. Так как прямолинейные программы и их деревья разбора взаимно однозначно соответствуют друг другу, то в описании алгоритмов позволим себе отождествлять соответствующие понятия. Например, будем отождествлять понятия «деревья разбора» и «грамматики ПП», «узлы дерева» и «правила грамматики ПП» и т.п.

3 Построение ПП с помощью AVL-деревьев

3.1 Алгоритм Риттера

Риттер [3] сформулировал алгоритм построения ПП, выводящую данный текст, по LZ-факторизации этого текста. При этом в качестве основной структурой данных используется AVL-дерево. AVL-дерево –это двоичное дерево, у каждого внутреннего узла которого высоты сыновей отличаются не более чем на 1.

Утверждение 3.1 (нижняя оценка на размер ПП)**.** *Размер любой ПП, выводящей заданный текст, не меньше размера LZ-факторизации этого текста.*

Доказательство этого факта можно найти в [3].

Утверждение 3.2. *Существует алгоритм, который по данному тексту S длины n и по его LZ-факторизации размера k за время $O(k \log n)$ строит ПП, выводящую S и имеющую размер $O(k \log n)$, то есть являющуюся $O(\log n)$-приближением к минимальной.*

Доказательство утверждения изначально является конструктивным. Опишем ключевые идеи алгоритма Риттера, поскольку они важны для дальнейшего изложения. Подробное описание и доказательства можно найти в [3].

AVL-грамматиками называются ПП, деревья вывода которых являются AVL-деревьями. Основными операциями, используемыми в алгоритме, являются конкатенация AVL-грамматик и взятие подграмматики AVL-грамматики. Следующие леммы оценивают сверху трудоемкость этих операций.

Лемма 3.1 (о конкатенации AVL-грамматик)**.** *Пусть T, T' – деревья вывода AVL-грамматик, высота которых равна h и h' соответственно. Тогда, добавив не более чем $O(|h - h'|)$ новых правил, за время $O(|h - h'|)$ можно построить AVL-грамматику $T \cdot T'$, которая выводит текст $S(T) \cdot S(T')$.*

Лемма 3.2 (о взятии подграмматики AVL-грамматики)**.** *Пусть T – дерево вывода AVL-грамматики, высота которой равна h. Тогда для любых $1 \leq$*

$i \le j \le |S(T)|$, добавив не более $O(h)$ новых правил, за время $O(h)$ можно построить AVL-грамматику T', которая выводит текст $S(T)[i..j]$.

Алгоритм Риттера получает на вход LZ-факторизацию $w_1, w_2, \ldots, w_k$ данного текста S и индуктивно строит ПП T, выводящую текст $w_1 \cdot w_2 \cdot \ldots \cdot w_i$ для $i = 1, 2, \ldots, k$.

Инициализация: Полагаем T равной грамматике, состоящей из терминального правила, выводящего w_1, равный первому символу строки S.

Основной цикл: Предположим, что для фиксированного $i > 1$ уже построена ПП T, выводящая текст $w_1 \cdot w_2 \cdot \ldots \cdot w_i$. По определению LZ-факторизации фактор w_{i+1} является подстрокой в тексте $w_1 \cdot w_2 \cdot \ldots \cdot w_i$. Фиксируем позиции вхождения фактора w_{i+1} в текст $w_1 \cdot w_2 \cdot \ldots \cdot w_i = S(T)$ и, используя алгоритм взятия подграмматики, находим ПП T_{i+1} такую, что $S(T_{i+1}) = w_{i+1}$. Полагаем $T := T \cdot T_{i+1}$.

3.2 Модернизированный алгоритм Риттера

Как видно из леммы 3.1, при конкатенации AVL-грамматик существенно разной высоты появляется много новых правил, при добавлении которых может возникнуть необходимость в большом числе вращений. Именно в этом состоит узкое место алгоритма Риттера – когда его основной цикл повторится достаточное число раз, высота текущей AVL-грамматики T становится большой, а при каждом последующем выполнении основного цикла с T конкатенируется AVL-грамматика относительно небольшой высоты.

Идея оптимизации [12] состоит в том, чтобы откладывать конкатенацию деревьев до тех пор, пока следующий фактор целиком содержится в T, и затем, пользуясь свойством ассоциативности операции конкатенации, выбрать более эффективный порядок конкатенаций накопившихся деревьев. Интуитивное обоснование этой идеи таково: если уже построена «большая» ПП, то большинство последующих факторов входит в текст, выводимый из нее, а значит, эти факторы могут быть обработаны вместе.

В работе [12] оптимальный порядок конкатенаций предлагается находить решением задачи динамического программирования.

Пусть $F_1, F_2, \ldots, F_\ell$ — деревья, которые необходимо конкатенировать оптимальным способом. Обозначим $\varphi(p, q)$ — число операций вращения AVL-дерева при оптимальном порядке конкатенаций $F_p, F_{p+1}, \ldots, F_q$. Алгоритм использует следующую формулу для вычисления значения φ:

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

Формула корректна, так как конкатенация двух деревьев F_p и F_q порождает не более $2 \cdot |h(F_p) - h(F_q)|$ вращений AVL-дерева [3] и $h(F_p) \leq 1.44 \cdot \log(|F_p|)$ (см. [14]). Для простоты общие константы были отброшены из формулы. Заметим, что данная рекуррентная формула позволяет не только вычислить количество операций вращения при оптимальном порядке конкатенаций, но и найти сам оптимальный порядок.

Тогда значение $\varphi(1, \ell)$ соответствует оптимальному порядку конкатенаций всех деревьев, а для вычисления значений φ потребуется $O(\ell^2)$ памяти и $O(\ell^3)$ времени. Таким образом, уменьшив количество поворотов при конкатенации деревьев, мы получаем выигрыш по времени. Правда, это получается ценой дополнительных ресурсов, необходимых для вычисления φ.

Очевидно, что при больших ℓ поиск оптимального порядка таким методом не оправдан, так как сам поиск потребует больше ресурсов, чем его результат позволит оптимизировать. При больших значениях ℓ для поиска порядка конкатенаций можно применить следующую эвристику. Разобьем деревья на группы из k подряд идущих, тогда общее количество групп будет равно $g = \left[\frac{\ell}{k}\right]$. Внутри каждой из групп за $O(k^3)$ найдем и применим оптимальный порядок конкатенаций. Для получившихся g деревьев опять же найдем оптимальный порядок конкатенаций за $O(g^3)$.

Пример 3.1. *Пусть нам необходимо найти конкатенацию деревьев F_1, F_2, …, F_ℓ.*

Зафиксируем некоторое число k и $g = \left[\frac{\ell}{k}\right]$. Найдем

$$H_1 = concat(F_1, \ldots, F_k)$$

$$H_2 = concat(F_{k+1}, \ldots, F_{2k})$$

$$\ldots$$

$$H_g = concat(F_{(g-1)k+1}, \ldots, F_\ell)$$

concat обозначает применение алгоритма оптимальной конкатенации с вычисление φ по формуле выше. Отметим, что в последней группе, в отличие от остальных, вероятно, окажется меньше, чем k деревьев.

В конце вычислим $T = concat(H_1, \ldots, H_g)$, и дерево T является искомой конкатенацией деревьев $F_1, \ldots F_\ell$.

Итого, мы сконкатенировали все ℓ деревьев, потратив $O(k^3 \cdot g + g^3)$ времени и $O(k^3 + g^3)$ памяти на оптимизацию количества поворотов AVL-деревьев. Следующее утверждение говорит о том, как правильно выбрать значение для k.

Утверждение 3.3. *Оптимальное время работы последней эвристики равно $O(\ell^{1.8})$ и достигается при $k = \Theta(\ell^{0.4})$.*

Доказательство:

$O(k^3 \cdot g + g^3) = O(k^2 \cdot \ell + \frac{\ell^3}{k^3})$. Из неравенства Коши-Буняковского получаем, что при $k = \Theta(\ell^{0.4})$ достигается минимум, равный $O(\ell^{1.8})$. $\square$

Понятно, что предложенная эвристика не гарантирует абсолютную минимальность количества выполненных поворотов, зато требует меньше дополнительных ресурсов для поиска порядка конкатенаций.

Замечание. Предложенные эвристики по выбору порядка конкатенаций для оптимизации количества поворотов не ухудшают размер результирующей ПП. Также как и в алгоритме Риттера будет выполнено не более k операций взятия подграмматики и ровно $k - 1$ операций конкатенации. Применение операции увеличивает размер грамматики на $O(\log n)$. Поэтому размер результирующей грамматики равен $O(k \log n)$.

4 Построение ПП с помощью рандомизированных деревьев

Результаты этой главы были получены автором и опубликованы в [12].

Двоичное дерево T является *рандомизированным* тогда и только тогда, когда

- T – пустое дерево; или
- оба его поддерева L и R являются независимыми рандомизированными двоичными деревьями, и $P\{size(L) = i\} = \frac{1}{n}$ для любого i от 0 до $n-1$, где $size(X)$ – количество вершин в дереве X, а $n = size(T)$.

Для рандомизированных деревьев имеется вероятностная логарифмическая от числа узлов оценка высоты, а именно, ожидаемая высота рандомизированного дерева с n узлами есть $O(\log n)$ [15]. Более того, для любой константы $c > 1$ вероятность того, что высота рандомизированного дерева с n узлами больше $2c \ln n$, ограничена величиной $n\left(\frac{n}{e}\right)^{-c\ln(c/e)}$.

Следовательно, рандомизированное дерево балансируется само собой за счет своей случайной природы, в отличии от других сбалансированных двоичных деревьев, которые выполняют дополнительные действия, направленные на балансировку.

Как уже говорилось ранее, в основе алгоритма Риттера лежит представление деревьев вывода ПП в виде AVL-деревьев. AVL-дерево имеет логарифмическую высоту, и это позволяет выполнять операции с деревьями, добавляя лишь $O(\log n)$ новых правил. Но что будет, если вместо AVL-деревьев использовать рандомизированные деревья? Получим ли выигрыш по скорости за счет замены структуры данных на более быструю?

4.1 Рандомизированные деревья вывода ПП

Дерево вывода T некоторой ПП будем называть *рандомизированным*, если оно удовлетворяет одному из следующих условий:

- Это дерево состоит ровно из одного узла, и в нем хранится выводимый терминал.

- Оба поддерева L и R являются независимыми рандомизированными деревьями, и $P\{|S(L)| = i\} = \frac{1}{n-1}$ для любого i от 1 до $n-1$, где $n = |S(T)|$.

Лемма 4.1 (о количестве нетерминальных правил)**.** *Если ПП выводит строку длиной n, то в дереве вывода этой ПП количество узлов, соответствующих нетерминальным правилам, равно $n-1$.*

Доказательство: Пусть количество таких узлов равно k. Посчитаем количество ребер в дереве вывода. Общеизвестно, что дерево содержит $n+k-1$ ребро, а с другой стороны количество ребер равно $2k$ (из каждого нетерминального узла ровно два ребра). Приравнивая полученные два числа, получаем $k = n-1$. □

Лемма 4.2. *Если удалить из некоторой рандомизированной ПП узлы, соответствующие терминальным правилам, получится дерево являющееся рандомизированным двоичным деревом.*

Доказательство: Построим доказательство индукцией по высоте дерева.

База: Пусть высота равна 1. Тогда T состоит из одного терминального узла. После его удаления получим пустое дерево.

Шаг: Предположим, что для любого дерева высотой меньшей k ($k > 1$) это утверждение выполняется. Докажем его и для произвольного дерева высоты k. Из того, что $k > 1$ следует, что оба поддерева L и R являются независимыми рандомизированными ПП и для любого i от 1 до $n-1$ $P\{|S(L)| = i\} = \frac{1}{n-1}$, где $n = |S(T)|$. По предположению индукции после удаления терминальных правил из L и R получаются рандомизированные деревья L' и R'. Используя лемму о количестве нетерминальных правил, нетрудно из имеющегося вероятностного тождества получить тождество из определения рандомизированных двоичных деревьев $P\{size(L') = i\} = \frac{1}{n-1}$ для всех i от 0 до $n-2$.

Шаг индукции выполняется, следовательно, утверждение выполняется для любой рандомизированной ПП. □

Лемма 4.3 (о высоте рандомизированной ПП)**.** *Пусть T – рандомизированная ПП, тогда матожидание высоты дерева T равно $O(\log|S(T)|)$.*

Доказательство: Из предыдущих лемм следует, что средняя высота дерева из нетерминальных правил в среднем равна $O(\log|S(T)|-1)$. Так как добавление терминальных узлов увеличит высоту дерева на 1, то и высота дерева T удовлетворяет требуемой оценке. □

4.2 Операции над рандомизированными ПП

Рассмотрим модификации стандартных операций над рандомизированными деревьями: операция слияния и операция разрезания. В английской литературе эти операции принято называть *merge* и *split* соответственно.

Операция разрезания (*split*)

Вход: Рандомизированная ПП T, число pos такое, что $0 < pos < |S(T)|$.

Выход: Упорядоченная пара рандомизированных ПП (L, R) такая, что $S(T) = S(L) \cdot S(R)$ и $|S(L)| = pos$.

Алгоритм: Пусть $T = (LT, RT)$.

1. Если $|S(LT)| = pos$, то результатом будет пара (LT, RT).
2. Если $|S(LT)| < pos$, то сделаем рекурсивный вызов $(L', R') = split(RT, pos - |S(LT)|)$. Результатом будет пара (L, R), где $L = (LT, L')$, $R = R'$.
3. Если $|S(LT)| > pos$, то сделаем рекурсивный вызов $(L', R') = split(LT, pos)$. Результатом будет пара (L, R), где $L = L'$, $R = (R', RT)$.

Утверждение 4.1 (о корректности алгоритма разрезания)**.** *Алгоритм выполнения операции разрезания корректен.*

Доказательство: При доказательстве будем использовать обозначения, принятые в алгоритме. Определимся с тем, что нам нужно доказать. Во-первых, для всех рекурсивных вызовов должно выполняться неравенство для pos. Во-вторых, L и R – рандомизированные ПП. В-третьих, $S(T) = S(L) \cdot S(R)$ и $|S(L)| = pos$.

Сразу заметим, что если мы попадаем в первый случай алгоритма, то алгоритм возвращает правильный результат.

Будем доказывать индукцией по $|S(T)|$.

База: Пусть из T выводится строка длины 2, тогда $pos = 1$. Понятно, что из LT и RT выводятся строки длины 1. Следовательно, это первый случай алгоритма, и алгоритм вернул правильный результат.

Шаг: Пусть для всех деревьев с длиной выводимой строки меньше k алгоритм возвращает правильный результат. Докажем, что алгоритм правильно работает и дерева с выводом длины ровно k.

1. Пусть $|S(LT)| = pos$. Как уже отмечалось, в этом случае алгоритм возвращает правильный результат.

2. Пусть $|S(LT)| < pos$. Ясно, что $0 < pos - |S(LT)| < |S(RT)|$. Также длина вывода RT меньше k, следовательно, по предположению индукции, алгоритм разрезания возвращает верный результат, то есть L' и R' – рандомизированные ПП, $S(RT) = S(L') \cdot S(R')$, $|S(L')| = pos - |S(LT)|$.

   Очевидно, что $R = R'$ – рандомизированная ПП. Докажем, что $L = (LT, L')$ также является рандомизированной ПП. Рассмотрим произвольное i от 1 до $pos - 1$.

$$P\{|S(LT)| = i \mid |S(LT)| < pos\} = \frac{P\{|S(LT)| = i\}}{P\{|S(LT)| < pos\}}$$

$$= \frac{1/(k-1)}{(pos-1)/(k-1)} = \frac{1}{pos-1}$$

   Следовательно, L является рандомизированной ПП.

   Ясно, что $S(L) = S(LT) \cdot S(L')$, отсюда $|S(L)| = |S(LT)| + |S(L')| = pos$ и $S(T) = S(LT) \cdot S(RT) = S(LT) \cdot (S(L') \cdot S(R')) = (S(LT) \cdot S(L')) \cdot S(R') = S(L) \cdot S(R)$. Значит, алгоритм возвращает правильный результат.

3. Пусть $|S(LT)| > pos$. Этот случай аналогичен случаю $|S(LT)| < pos$.

Шаг индукции выполняется, значит, описанный алгоритм корректен. □

Лемма 4.4 (об оценке сложности алгоритма разрезания)**.** *Если h — высота дерева, то операция разрезания для него с произвольным pos работает за время $O(h)$.*

Доказательство: При каждом рекурсивном вызове высота дерева уменьшается хотя бы на 1. Отсюда немедленно следует требуемая асимптотика. □

Операция слияния (*merge*)

Вход: Рандомизированные ПП T_1 и T_2.

Выход: Рандомизированная ПП T, такая что $S(T) = S(T_1) \cdot S(T_2)$.

Алгоритм: Пусть $T_1 = (L_1, R_1), T_2 = (L_2, R_2)$. Обозначим $n_1 = |S(T_1)|$ и $n_2 = |S(T_2)|$. Сгенерируем случайное число от 1 до $(n_1 + n_2 - 1)$ и обозначим его как r.

1. Если $0 < r < n_1$, то построим дерево $R_{new} = merge(R_1, T_2)$ и вернем дерево $T = (L_1, R_{new})$.
2. Если $r = n_1$, то вернем дерево $T = (T_1, T_2)$.
3. Если $n_1 < r < n_1 + n_2$, то построим дерево $L_{new} = merge(T_1, L_2)$ и вернем дерево $T = (L_{new}, R_1)$.

Утверждение 4.2 (о корректности алгоритма слияния)**.** *Алгоритм выполнения операции слияния корректен.*

Доказательство: В доказательстве будем использовать обозначения принятые в описании алгоритма.

Будем доказывать индукцией по суммарному количествам узлов в деревьях T_1 и T_2.

База: Пусть оба дерева содержат по одному узлу, тогда оба эти узла являются терминальными правилами, то есть $n_1 = n_2 = 1$, и дерево (T_1, T_2), очевидно, искомая рандомизированная ПП.

Шаг: Пусть для всех деревьев с суммарным количеством узлов меньше k утверждение выполняется. Докажем, что выполняется и для деревьев ровно с k узлами.

Докажем, что $S(T) = S(T_1) \cdot S(T_2)$. Рассмотрим все возможные ветки исполнения алгоритма.

1. Пусть $0 < r < n_1$. Следовательно, в дереве T_1 есть хотя бы один нетерминальный узел, значит, L_1 и R_1 непустые деревья. В деревьях R_1 и T_2 узлов меньше k, значит, по предположению индукции R_{new} является рандомизированной ПП и $S(R_{new}) = S(R_1) \cdot S(T_2)$.

   Тогда $S(T) = S(L_1) \cdot S(R_{new}) = S(L_1) \cdot S(R_1) \cdot S(T_2) = S(T_1) \cdot S(T_2)$.

2. Пусть $r = n_1$. Очевидно, что $S(T) = S(T_1) \cdot S(T_2)$.

3. Пусть $n_1 < r < n_1 + n_2$. Этот случай аналогичен случаю $0 < r < n_1$.

Пусть $T = (L,\, R)$. Осталось доказать, что для произвольного i от 1 до $n_1 + n_2 - 1$ выполняется $P\{|S(L)| = i\} = \dfrac{1}{n_1 + n_2 - 1}$.

$$\begin{aligned} P\{|S(L)| = i\} = P\{|S(L)| = i \mid 0 < r < n_1\} \cdot P\{0 < r < n_1\} \\ + P\{|S(L)| = i \mid r = n_1\} \cdot P\{r = n_1\} \\ + P\{|S(L)| = i \mid n_1 < r < n_1 + n_2\} \cdot P\{n_1 < r < n_1 + n_2\} \end{aligned}$$

Заметим, что:

- если $0 < r < n_1$, то $0 < |S(L)| < n_1$;
- если $r = n_1$, то $|S(L)| = n_1$;
- если $n_1 < r < n_1 + n_2$, то $n_1 < |S(L)| < n_1 + n_2$.

Тогда при фиксированном i две из трех условных вероятностей равны нулю.

Для любого i от 1 $n_1 - 1$ имеем

$$\begin{aligned} P\{|S(L)| = i\} = P\{|S(L_1)| = i\} \cdot P\{0 < r < n_1\} \\ = \frac{1}{n_1 - 1} \cdot \frac{n_1 - 1}{n_1 + n_2 - 1} = \frac{1}{n_1 + n_2 - 1} \end{aligned}$$

.

Для $i = n_1$ имеем $P\{|S(L)| = i\} = 1 \cdot P\{r = n_1\} = \dfrac{1}{n_1 + n_2 - 1}$.

Для любого i от $n_1 + 1$ до $n_1 + n_2 - 1$ имеем

$$P\{|S(L)| = i\} = P\{|S(L_{new})| = i\} \cdot P\{n_1 < r < n_1 + n_2\}$$

$$= P\{|S(L_2)| = i - n_1\} \cdot P\{n_1 < r < n_1 + n_2\}$$
$$= \frac{1}{n_2 - 1} \cdot \frac{n_2 - 1}{n_1 + n_2 - 1} = \frac{1}{n_1 + n_2 - 1}$$

.

Следовательно, T является рандомизированной ПП с нужным выводом. Шаг индукции выполняется, значит, описанный алгоритм корректен. □

Лемма 4.5 (об оценке сложности алгоритма слияния)**.** *Если h_1 и h_2 — высоты двух деревьев соответственно, то операция слияния для этих двух деревьев работает за время $O(h_1 + h_2)$.*

Доказательство: При каждом рекурсивном вызове высота одного из деревьев уменьшается хотя бы на 1. Отсюда немедленно следует требуемая асимптотика. □

4.3 Алгоритм построения рандомизированной ПП

Сформулируем алгоритм построения ПП с использованием рандомизированных деревьев.

Алгоритм построения рандомизированной ПП

Вход: LZ-факторизация строки $S = w_1 \cdot w_2 \cdot \ldots \cdot w_k$.

Выход: Рандомизированная ПП T такая, что $S(T) = S$.

Алгоритм:

1. Присвоим в T_1 дерево, выводящее слово из буквы w_1.
2. Для всех $i \in 2..k$
   (a) Построим F_i, выводящую фактор w_i.
   - В случае, если фактор w_i соответствует новой букве, то создадим и положим в F_i новую рандомизированную ПП из одного узла, выводящую эту букву.
   - В противном случае w_i является подстрокой вывода $S(T_{i-1})$, и, применив не более двух операций разрезания к дереву T_{i-1}, получим искомую F_i.

   (b) Положим в $T_i = merge(T_{i-1}, F_i)$.

3. T_k — искомая рандомизированная ПП.

Для обоснования корректности этого алгоритма достаточно доказать, что $S(T_i) = w_1 \cdot w_2 \cdot \ldots \cdot w_i$ для любого $i = 1..k$. Это можно сделать методом математической индукции. Доказательство тривиально, и напрямую следует из корректности алгоритмов разрезания и слияния, поэтому оставлено читателю в качестве упражнения.

Утверждение 4.3. *Математическое ожидание размера результирующей ПП равно в среднем $O(k \log n)$, а среднее время работы алгоритма $O(k \log n)$.*

Доказательство: Вывод любой рандомизированной ПП, используемой по ходу выполнения алгоритма, не больше n, значит, матожидание высоты любого дерева в среднем $O(\log n)$. При обработке каждого из k факторов выполняется не больше двух операций разрезания и не более одной операции слияния. Каждая из этих операций создает в среднем не более $O(\log n)$ новых узлов и выполняется в среднем за время $O(\log n)$. Следовательно, размер результирующей ПП равен в среднем $O(k \log n)$, а среднее время работы равно $O(k \log n)$. □

5 Многопоточный алгоритм построения ПП

В предыдущих главах были рассмотрены алгоритм Риттера и различные его модификации, направленные на улучшение времени выполнения и/или качества сжатия. Все эти модификации носили алгоритмический характер. Но ускорения алгоритма на практике можно добиться и с помощью технических улучшений. Например, оптимизировать конкретную реализацию алгоритма, реализовать алгоритм на более эффективном языке программирования, купить более мощный компьютер, в конце концов. Среди прочих, наиболее интересным техническим улучшением кажется использование многопоточных вычислений. Именно этому вопросу посвящена данная глава.

5.1 Основная идея

Основная идея алгоритма Риттера представлять ПП в виде сбалансированного бинарного дерева и *последовательно* вырезать из уже построенного дерева следующий фактор и конкатенировать вырезанное дерево с уже имеющимся. А можно ли эти *последовательные* действия выполнять параллельно? Нельзя, потому что каждый следующий фактор мы вырезаем из дерева, построенного к моменту обработки фактора.

Наша цель — сохранить оценку на размер результирующей ПП, но сделать возможным выполнение некоторых действий параллельно. Сформулируем общий алгоритм построения ПП с использованием бинарных деревьев.

Абстрактный алгоритм построения ПП с помощью бинарных деревьев

Вход: Текст S и его LZ-факторизация.

Выход: ПП, выводящая текст S.

Алгоритм:

1. Завести множество бинарных деревьев A, изначально оно пусто.
2. Выполнять любое из следующих действий, пока множество A не будет содержать единственное дерево, выводящее текст S.
   - Заменить любые два дерева в множестве A на их конкатенацию.
   - Выбрать любой неиспользованный ранее фактор из факторизации и добавить дерево, выводящее этот фактор, в множество A. Причем это дерево должно быть получено в результате взятия подстроки некоторого дерева из A или должно состоять из одной вершины, то есть выводить однобуквенное слово.
3. Построить и вернуть ПП по единственному дереву в A.

Замечание. Для каждого дерева из множества A в процессе выполнения алгоритма можно вычислить и запомнить, где именно в тексте S встречается вывод этого дерева. Тогда границы этих вхождений можно использовать в алгоритме для понимания, из какого дерева можно вырезать фактор и какие деревья можно конкатенировать.

Теорема 5.1 (о размере ПП в общем алгоритме)**.** *Пусть длина текста S равна n, а размер LZ-факторизации равен k. Если выполнить описанный выше алгоритм, используя в качестве бинарных деревьев AVL-деревья, а для выполнения операций конкатенации и взятия подстроки алгоритмы из [3], то размер результирующей ПП будет $O(k \log n)$.*

Доказательство:

За все время выполнения алгоритма в множество может быть добавлено только k новых деревьев (по одному для каждого фактора). При этом будет выполнено не больше k операций взятия подстроки. Также очевидно, что за все время выполнения алгоритма операция конкатенации будет выполнена только $k-1$ раз. При выполнении каждой из операций добавляется $O(\log n)$ новых правил. Следовательно, общее количество правил к концу алгоритма $O(k \log n)$. □

Рассмотрим следующий алгоритм. В нем однозначно определен порядок действий, но выбран он так, что некоторые последовательные действия можно выполнять параллельно.

Многопоточный алгоритм построения ПП с помощью бинарных деревьев

Вход: Текст S и его LZ-факторизация.

Выход: ПП, выводящая текст S.

Алгоритм:

1. Завести множество бинарных деревьев A, изначально оно пусто.
2. Найти все еще неиспользованные факторы, для которых можно получить дерево с помощью не более одной операции взятия подстроки дерева из A.
3. Выполнить все операции взятия подстроки параллельно и добавить их результаты в A.
4. Заменить «соседние по S» деревья в множестве на их конкатенацию. Это можно также попытаться выполнить параллельно настолько, насколько это возможно.

5. Если множество A содержит единственное дерево, выводящее текст S, то построить и вернуть ПП по этому дереву.

6. Иначе вернуться на второй шаг.

Теорема 5.2 (о размере ПП в многопоточном алгоритме)**.** *Пусть длина текста S равна n, а размер LZ-факторизации равен k. Если выполнить описанный выше алгоритм, используя в качестве бинарных деревьев AVL-деревья, а для выполнения операций конкатенации и взятия подстроки алгоритмы из [3], то размер результирующей ПП будет $O(k \log n)$.*

Доказательство:

Эта теорема является прямым следствием теоремы об общем алгоритме. □

5.2 Оценка количества итераций

В этой главе мы поговорим о количестве итераций, за которое предложенный многопоточный алгоритм закончит свое выполнение.

Будем говорить, что фактор w_i зависит от фактора w_j, если раннее вхождение w_i пересекается непосредственно с w_j. Построим следующий ориентированный граф. Каждому фактору w_i в графе соответствует вершина $v(w_i)$. Добавим дугу из вершины $v(w_i)$ в вершины $v(w_j)$ тогда и только тогда, когда w_i зависит от w_j. Будем называть такой граф графом зависимостей факторов.

Лемма 5.1. *Граф зависимостей факторов всегда ацикличен.*

Доказательство:

Если фактор w_i зависит от фактора w_j, то $i > j$. Следовательно, граф ацикличен. □

Обозначим $h(w_i)$ — длина максимальной цепи, начинающейся из $v(w_i)$.

Лемма 5.2. *На i-й итерации второго шага многопоточного алгоритма будут найдены факторы $\{w_j | h(w_j) = i - 1\}$.*

Доказательство:

Докажем индукцией по i.

База индукции: $i = 1$. Очевидно, что на первой итерации будут обработаны все факторы из одной буквы и только они. А так как эти факторы не зависят от других факторов, то для них значение h равно 0.

Шаг индукции: $i = k + 1 > 1$. На первых k итерациях были обработаны все факторы, для которых значение h меньше k.

Рассмотрим любой фактор w_j такой, что $h(w_j) = k$. Из определения h и предположения индукции следует, что все факторы, от которых зависит w_j, уже обработаны. Следовательно, соответствующие им деревья уже были добавлены в множество A, а так как все они «соседние в S», то были сконкатенированы в одно дерево. Следовательно, фактор w_j может быть получен одной операцией взятия подстроки дерева из A и будет обработан на текущей итерации.

Рассмотрим любой фактор w_j такой, что $h(w_j) > k$. Существует фактор w_l такой, что фактор w_j зависит от фактора w_l и $h(w_l) = h(w_j) - 1 \geq k$. Следовательно, фактор w_l еще не был обработан, и пока невозможно вырезать фактор w_j из деревьев в A. Значит, фактор w_j не может быть обработан на текущей итерации. □

Следствие. В процессе выполнения многопоточного алгоритма будет сделано $max_{i=1}^{k}(h(w_i)) + 1$ итераций.

Исследуем насколько большим может быть количество итераций.

Пример 5.1. *Пусть дан текст* $S = a^{2^x}$, $n = 2^x$.

Построим LZ-факторизацию $F = (a, a, a^2, a^4, \ldots, a^{2^{x-1}})$, $k = x + 1$.

Посчитаем значения h: $h(w_1) = h(w_2) = 0$, $h(w_i) = i - 1$ *для* $i > 2$.

Количество итераций равно $x = k - 1 = \log n$.

В этом примере k факторов будут обработаны предложенным алгоритмом в $k-1$ итерацию, что ничем не отличается от классического алгоритма Риттера. Правда, в данном примере факторизация настолько мала относительно размера текста, что оба алгоритма будут работать быстро, а именно $O(\log^2 n)$.

Приведем пример с большим размером LZ-факторизации и большим количеством итераций.

Пример 5.2 (с бесконечным алфавитом)**.** *Рассмотрим бесконечный алфавит* $\Sigma = \{x_1, x_2, x_3 \ldots\}$.

Последовательность текстов $\{S_m\}$*, где* $S_1 = x_1x_2$*, и для любого* $m > 1$ *задается* $S_m = S_{m-1} \cdot x_{m-1}x_mx_{m+1}$*. Тогда* $n_m = 3m - 1$.

Построим LZ-факторизацию $F_m = (x_1, x_2, x_1x_2, x_3, \ldots, x_{m-1}x_m, x_{m+1})$. $k_m = 2m$.

Посчитаем значения h*:* $h(w_1) = h(w_2) = 0$*,* $h(w_{2i-1}) = i - 1, h(w_{2i}) = 0$ *для* $i > 1$.

Количество итераций равно $m = \Theta(k_m) = \Theta(n_m)$.

Сформулируем теорему, в доказательстве которой конструктивно строится последовательность «плохих» слов над конечным алфавитом.

Теорема 5.3. *Существует последовательность текстов* $\{T_p\}$ *над конечным алфавитом такая, что* $\lim_{p\to\infty} |T_p| = \infty$ *и размер факторизации* $k_p = \Omega\left(\frac{|T_p|}{\log|T_p|}\right)$ *и* $c_p = \Omega\left(\frac{|T_p|}{\log|T_p|}\right)$*, где* k_p *и* c_p *— размер LZ-факторизации для* T_p *и количество итераций многопоточного алгоритма, запущенного на* T_p*, соответственно.*

Доказательство:

Зафиксируем конечный алфавит $\Sigma = \{A, B, C, D, E\}$.

Возьмем произвольное число $p > 1$ и зафиксируем y_i произвольным образом так, что $\{y_1, y_2, \ldots, y_{2^p}\} = \{A, B\}^p$.

Зафиксируем набор слов $x_1, x_2, \ldots, x_{2^p}$ так, что $x_i = y_iC$.

Зафиксируем $S_1 = x_1x_2$, $S_{m+1} = S_m \cdot x_{m-1}x_mx_{m+1}$ для $m > 1$.

Зафиксируем $T_p = x_1Dx_2D \cdot \ldots \cdot x_{2^p-1}Dx_{2^p}ES_{2^p-1}$.

Например, для $p = 2$ получится следующее:

$$y_1 = AA, y_2 = AB, y_3 = BA, y_4 = BB$$

$$x_1 = AAC, x_2 = ABC, x_3 = BAC, x_4 = BBC$$

$$T_2 = x_1Dx_2Dx_3Dx_4Ex_1x_2x_1x_2x_3x_2x_3x_4$$

$$= AAC \cdot D \cdot ABC \cdot D \cdot BAC \cdot D \cdot BBC \cdot E \cdot$$

$$\cdot AAC \cdot ABC \cdot AAC \cdot ABC \cdot BAC \cdot ABC \cdot BAC \cdot BBC$$

Докажем, что $\{T_p\}$ — искомая последовательность.

$$|T_p| = 2^p \cdot (p+2) + (3 \cdot 2^p - 4) \cdot (p+1) = 4 \cdot (p + \frac{5}{4}) \cdot (2^p - 1) + 1 = \Theta(p2^p)$$

$$\lim_{p \to \infty} |T_p| = \infty$$

Исследуем размер LZ-факторизации текста T_p. Так как буква E встречается в тексте ровно один раз, то LZ-факторизация имеет вид $(w_1, w_2, \ldots, w_k, E, v_1, v_2, \ldots, v_l)$, где $v1 \cdot v2 \cdot \ldots \cdot v_l = S_{2^p-1}$. Докажем, что $l = 2^{p+1} - 2$ и $(v_1, v_2, \ldots v_l) = (x_1, x_2, x_1x_2, x_3, x_2x_3, x_4, \ldots, x_{2^p-2}x_{2^p-1}, x_{2^p})$.

Необходимо доказать, что каждое из предложенных v_i является максимальным префиксом $v_i \cdot \ldots \cdot v_l$, содержащемся в $w_1 \cdot \ldots \cdot w_k E v_1 \cdot \ldots \cdot v_{i-1}$. То, что v_i действительно содержится в соответствующей строке не вызывает сомнений. Остается доказать максимальность этого префикса.

Заметим, что $|v_i| \geq p+1$ и $(p+1)$-й символ v_i совпадает с C, а каждый из первых p символов либо A, либо B. Вспомним, какой вид имеют каждое из x_j, и сделаем вывод, что любое вхождение v_i в T_p может быть только в начале некоторого слова x_j в разложении T_p на факторы $\{D, E, x_1, x_2, \ldots, x_{2^p}\}$. Остается заметить, что каждое из v_i входит в соответствующий префикс либо в первой половине строки (до буквы E), либо в самом конце этого префикса. Очевидно, что ни то, ни другое вхождение не может быть увеличено ни на один символ.

Таким образом, размер LZ-факторизации для T_p не меньше, чем $2^{p+1} - 2$.

$$|T_p| = \Theta(p2^p) \Rightarrow C_1 p 2^p \leq |T_p| \leq C_2 p 2^p$$

$$\log_2 |T_p| \geq p + \log_2 p + \log_2 C_1 \Rightarrow p = O(\log |T_p|)$$

$$2^p \geq \frac{|T_p|}{C_2 p} = \Omega\left(\frac{|T_p|}{\log |T_p|}\right)$$

Тогда $k_p \geq 2^{p+1} - 2 = \Omega\left(\frac{|T_p|}{\log |T_p|}\right)$

Остается исследовать количество итераций, за которое выполнится многопоточный алгоритм на тексте T_p. Заметим, что фактор v_3 зависит от фак-

торов v_1 и v_2, фактор v_5 от факторов v_3 и v_4 и т.д. Тогда

$$h(v_{l-1}) = h(v_{2^{p+1}-3}) \geq h(v_{2^{p+1}-5}) + 1 \geq h(v_{2^{p+1}-7}) + 2 \geq \dots$$

$$\geq h(v_3) + 2^p - 3 \geq h(v_1) + 2^p - 2 \geq 2^p - 2$$

$$c_p = \max_{w \in LZ(T_p)} h(w) \geq h(v_{2^{p+1}-3}) \geq 2^p - 2$$

Следовательно, $c_p = \Omega\left(\frac{|T_p|}{\log |T_p|}\right)$. $\square$

6 LCA-online алгоритм

В этой главе рассмотрим алгоритм построения ПП, основанный на другой идее. Далее дается краткое описание данного алгоритма, а более подробное его описание можно найти в [13].

6.1 LCA-offline алгоритм

Для начала опишем идеи более простого алгоритма. Этот алгоритм будет лишен свойства «онлайновости», то есть для его выполнения требуется хранить весь текст целиком, а не читать его символ за символом, храня в памяти лишь некоторый прочитанный «хвост».

Основная идея алгоритма заключается в следующем. Согласно некоторому правилу выберем пары последовательных символов в тексте, для каждой выбранной пары XY добавим правило вывода $Z \to XY$ и заменим данное вхождение XY на Z. Причем если мы уже добавляли правило вывода с такой же правой частью, то мы не создаем новое правило, а используем то самое правило с тем же нетерминалом в левой части. После замены всех выбранных пар длина текста уменьшается. Повторяем подобные операции выбора и замены пар соседних символов, пока текст не станет длиной 1. Нетрудно понять, что в итоге мы получим набор правил ПП, выводящей начальный текст.

А теперь обсудим, как именно нужно выбирать пары символов так, чтобы

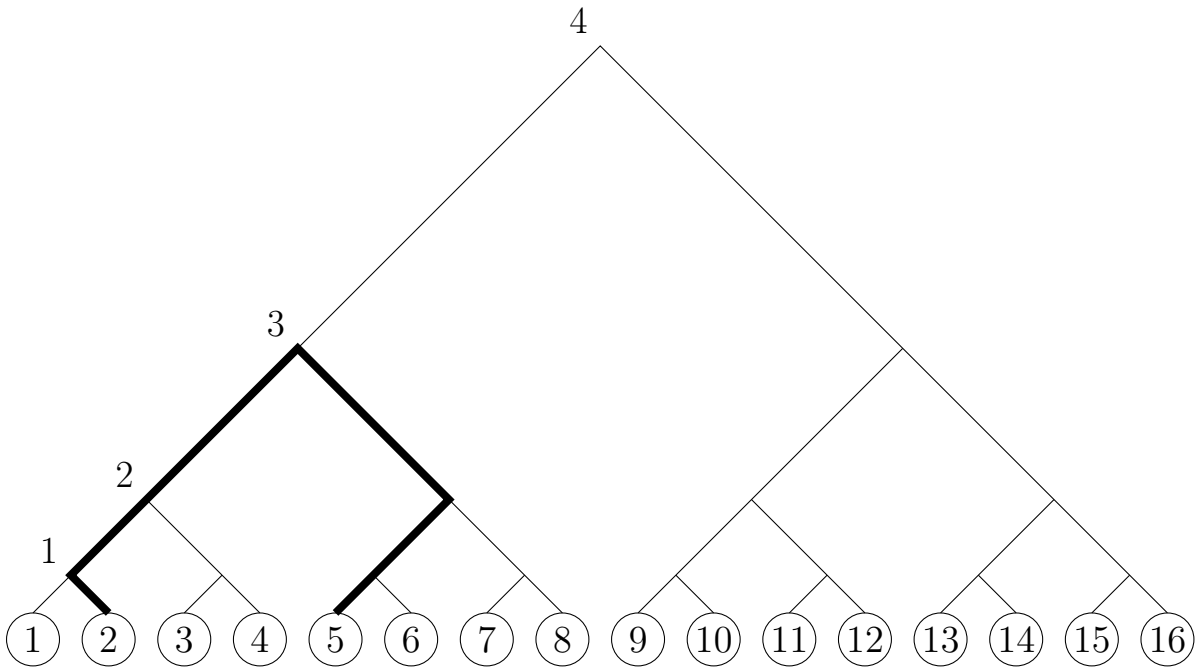

Рис. 2: Дерево T_4: $lca(2,5) = lca(5,2) = 3$.

размер получающейся ПП был небольшим.

Зафиксируем некоторый линейных порядок на множестве терминалов и нетерминалов. Будем обозначать через A_i символ, являющийся i-м в порядке возрастания.

Далее введем несколько определений.

Повторной парой будем называем вхождение A_iA_j, если $i = j$.

Если текст содержит подстроку $A_iA_jA_k$ такую, что $j < i$ и $j < k$, то будем называть пару A_jA_k **минимальной**.

Зафиксируем некоторое положительное число k. Рассмотрим корневое, полное бинарное дерево дерево T_k, листья которого пронумерованы числами от 1 до 2^k слева направо. Высотой вершины v будем называть длину пути от вершины v до листа в поддереве v. Тогда будем обозначать $lca(i,j)$ высоту наименьшего общего предка i-го и j-го листьев.

Зафиксируем некоторую подстроку $A_iA_jA_kA_l$ текста такую, что $i < j < k < l$ или $i > j > k > l$. Если $lca(j,k) > lca(i,j)$ и $lca(j,k) > lca(k,l)$, то будем называть пару A_jA_k **максимальной**.

Теперь мы можем сформулировать алгоритм выбора пар для замены

нетерминалами. Для начала сформулируем вспомогательный алгоритм по принятию решения об одной конкретной паре, а затем и сам алгоритм выбора пар.

Алгоритм $replaced\_pair$**, решающий выбирать ли пару** $S[i..i+1]$

Вход: Текст S и позиция i.

Выход: $true$, если пара $S[i..i+1]$ должна быть выбрана, и $false$ иначе.

Алгоритм:

1. Если $i = |S|$, то вернуть $false$.
2. Иначе если $i + 4 > |S|$, то вернуть $true$.
3. Иначе если пара $S[i..i+1]$ – повторная, то вернуть $true$.
4. Иначе если пара $S[i+1..i+2]$ – повторная, то вернуть $false$.
5. Иначе если пара $S[i+2..i+3]$ – повторная, то вернуть $true$.
6. Иначе если пара $S[i..i+1]$ – минимальная или максимальная, то вернуть $true$.
7. Иначе если пара $S[i+1..i+2]$ – минимальная или максимальная, то вернуть $false$.
8. Иначе вернуть $true$.

Алгоритм выбора пар

Вход: Текст S.

Выход: Множество выбранных пар.

Алгоритм:

1. Присвоить $R := \emptyset$.
2. Присвоить $i := 1$.
3. Пока $i + 1 \leq |S|$ выполнять
   (a) Если $replaced\_pair(S, i) = true$, то добавить в R пару $S[i..i+1]$ и увеличить i на 2.
   (b) Иначе увеличить i на 1.

4. Вернуть множество R.

Далее будем использовать обозначения: $S_0 = S$, S_k — текст, получившийся после k-й серии замен.

Лемма 6.1. *Если алгоритм $i < |S|$ и $replaced\_pair(S, i)$ возвращает $false$, то $replaced\_pair(S, i + 1)$ обязан вернуть $true$.*

Доказательство этой леммы довольно простое, необходимо аккуратно разобрать все случаи.

Следствие 6.1. *Для любого $k > 0$ выполняется $\frac{1}{2}|S_k| \leq |S_{k+1} \leq \frac{2}{3}|S_k|$.*

Доказательство: Это напрямую следует из леммы, так как для любого i будет выбрана либо пара $S_k[i..i + 1]$, либо пара $S_k[i + 1..i + 2]$. □

Следствие 6.2. *Общее время работы алгоритма LCA-offline $O(n)$, где n — размер сжимаемого текста.*

Доказательство: На k-й итерации выбора и замены пар символов необходимо $O(|S_k|)$ времени. Так как каждый раз длина S_{k+1} меньше длины S_k минимум в 1.5 раза, то получаем общее время работы $O(n)$. □

Утверждение 6.1. *LCA-offline алгоритм строит $O(\log^2 n)$-приближение к минимальной ПП.*

Полное доказательство можно найти в [13], здесь будут изложены лишь основные моменты.

В основе доказательства лежит следующее утверждение. Пусть существуют индексы такие, что $i_1 \leq j_1 < i_2 \leq j_2$ и $S[i_1..j_1] = S[i_2..j_2]$, тогда за одну серию замен на отрезке от i_2 до j_2 будет добавлено лишь $O(\log n)$ новых нетерминалов, а остальные нетерминалы уже были добавлены при замене на отрезке от i_1 до j_1. Доказательство этого факта основано на логике алгоритма *replaced_pair* и будет опущено в виду своей громоздкости.

Так как каждый фактор LZ-факторизации уже встречался в строке ранее, то при применении замен внутри этого фактора будет добавлено не более $O(\log n)$ новых правил. Так как всего выполняется $O(\log n)$ серий

замен, то общее количество правил равно $O(k \log^2 n)$, где k — размер LZ-факторизации. Вспомнив утверждение о нижней оценке на размер ПП, получаем, что построено $O(\log^2 n)$-приближение.

6.2 Преобразование LCA-offline алгоритма в LCA-online алгоритм

Идея преобразования заключается в том, что можно моделировать все итерации выбора и замены пар одновременно. Причем для моделирования каждой итерации достаточно хранить лишь некоторую конечную окрестность символов текущей позиции в тексте.

Заметим, что выполнение алгоритма $replaced\_pair(S, i)$ зависит только от подстроки $S[i-1..i+4]$. Таким образом, нам *уже* неважны символы до $(i-1)$-го и *еще* не важны символы после $(i+4)$-го.

Заведем очереди $q_0, q_1, \ldots q_h$. Каждая очередь будет моделировать итерацию выбора и замены пар соответствующего текста. Символы сжимаемого текста один за другим добавляются в конец очереди q_0. *В это время* из начала очереди q_0 извлекаются символы, принимается решение о замене пары, и получающийся после замены текст символ за символом добавляется в конец очереди q_1. И, таким образом, текст перемещается из очереди q_i в очередь q_{i+1} с заменой некоторых пар на нетерминалы. Очередь q_h будет содержать лишь один символ.

Фраза «в это время» означает, что удаление из начала очереди происходит параллельно с добавлением в конец. Как было замечено для принятия решения о паре $S[i..i+1]$ нужно знать лишь пять символов. Поэтому если некоторая очередь q_i содержит 5 элементов, мы уже можем принять решение о замене или не замене следующей пары и после этого выкинуть несколько элементов из начала очереди, добавив их в следующую.

Опишем более формально алгоритм добавления символа в конец очереди q_i.

Алгоритм $insert\_symbol$ **добавления символа в конец очереди**

Вход: Очередь q_i и добавляемый символ x.

Алгоритм:

- Добавить в конец очереди q_i символ x.
- Если q_i содержит 5 элементов, то
  1. Если $replaced\_pair(q_i, 1) = true$, то
     (a) Удалить символ из начала очереди q_i.
     (b) Присвоить $X := q_i[0]$ и $Y := q_i[1]$.
     (c) Добавить правило с правой частью XY и получить нетерминал Z из его левой части.
     (d) Удалить символ из начала очереди q_i.
     (e) Вызвать $insert\_symbol(q_{i+1}, Z)$.
  2. иначе
     (a) Удалить символ из начала очереди q_i.
     (b) Вызвать $insert\_symbol(q_{i+1}, q_i[0])$.

Тогда алгоритм LCA-online выглядит следующим образом.

Алгоритм LCA-online

Вход: Текст S.

Вход: ПП, выводящая текст S.

Алгоритм:

1. Завести пустое множество правил вывода.
2. Проинициализировать очереди q_0, q_1, ..., добавив в каждую из них фиктивный символ (для принятия решения о первой паре).
3. Для каждого i от 1 до $|S|$
   вызвать $insert\_symbol(q_0, S[i])$.
4. Для каждой очереди q_0, q_1, ...
   выполнить пост-обработку элементов в очереди.
5. Вернуть ПП с построенным множеством правил вывода.

Пост-обработка оставшихся в очередях символов заключается в опустошении очереди, замене пары символов слева направо нетерминалами и добавлением получившего текста в следующую очередь.

Утверждение 6.2. *LCA-online алгоритм строит $O(\log^2 n)$-приближение к минимальной ПП.*

Доказательство: Результат LCA-online алгоритма совпадает с результатом LCA-offline алгоритма. □

Утверждение 6.3. *LCA-online алгоритм использует $O(g_* \log^2 n)$ памяти, где n — длина текста, g_* — размер минимальной ПП, выводящей данный текст.*

Доказательство: Для хранения каждой очереди требуется $O(1)$, а всего очередей $O(\log n)$. Тогда как для хранения правил результирующей ПП необходимо $O(g_* \log^2 n)$. □

7 Сериализация прямолинейных программ

До этого момента обсуждался вопрос: как именно найти набор правил прямолинейной программы. Но для практического применения рассмотренных алгоритмов сжатия, безусловно, нужно уметь представлять построенные прямолинейные программы в виде последовательности байтов. Процесс перевода объектов в последовательность байтов принято называть сериализацией.

В [13] предлагается эффективный алгоритм сериализации прямолинейных программ.

7.1 Процесс сериализации

В начале сериализации прямолинейной программы G построим ее *частичное дерево вывода* $PTree(G)$ (концепция была предложена Риттером [3]). Пусть $Tree(G)$ – дерево вывода G. Тогда $PTree(G)$ получается из $Tree(G)$ следующим образом. Если $Tree(G)$ содержит некоторое максимальное по включению поддерево несколько раз, то заменим все его вхождения, за исключением самого левого, на единственную вершину с соответствующим нетерминалом. После всех таких возможных замен мы получим $PTree(G)$.

Таким образом, $PTree(G)$ содержит g внутренних вершин и $g+1$ лист, причем g совпадает с количеством нетерминальных правил в G.

Замечание. $PTree(G)$ можно построить за время $O(g)$ и память $O(g)$ без промежуточного построения $Tree(G)$.

Структуру двоичного дерева $PTree(G)$ можно представить в виду последовательности из $2g+1$ открывающихся и закрывающихся скобок. Необходимо обойти дерево в обратном обходе (post-order traversal), то есть в порядке *левое поддерево, правое поддерево, корень*, и при посещении листа вывести `'('`, а внутренней вершины — `')'`.

Для сохранения информации о данных в листьях дерева $PTree(G)$ достаточно запомнить $g+1$ число, каждое из них соответствует листу в обратном порядке обхода дерева.

- Если лист соответствует терминальному правилу, то этим числом является номер символа в алфавите.
- Если лист соответствует нетерминальному правилу, то этим числом является $|\Sigma|+x$, где x – номер внутренней вершины $PTree(G)$ в обратном

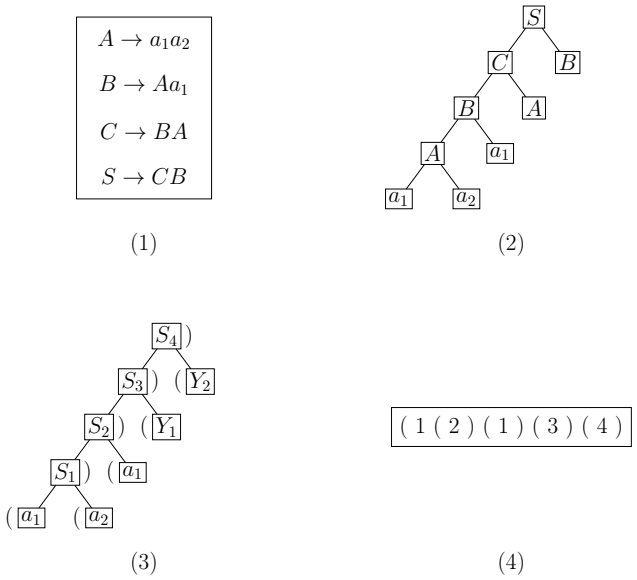

Рис. 3: Пример сериализации ПП. **(1)** правила вывода ПП. **(2)** частичное дерево вывода ПП. **(3)** частичное дерево после переобозначения нетерминалов. **(4)** результат сериализации.

порядке обхода дерева, содержащей тот же нетерминал, что и данный лист.

Оценим количество бит, которое требуется для кодирования последовательности из $2g + 1$ скобок и последовательности из $g + 1$ числа, от 1 до $|\Sigma|+g$ каждое. Последовательность скобок можно закодировать с помощью $2g+1$ битов. Последовательность из чисел можно закодировать с помощью $(g + 1)\lceil\log(|\Sigma| + g)\rceil$ битов.

Заметим, что при наивном подходе кодирования ПП нам бы потребовалось $2g\lceil\log(|\Sigma|+g)\rceil$ битов. Таким образом, данный подход при кодировании позволяет сократить затраты почти в два раза.

7.2 Процесс десериализации

По алфавиту $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$, последовательности из $2g + 1$ скобок B и последовательности из $g + 1$ числа L можно восстановить ПП. Ниже приведен алгоритм десериализации.

Алгоритм:

1. Завести пустое множество R с правилами вывода грамматики.
2. Для всех $i \in 1..\sigma$ добавить в R правило $X_i \to a_i$.
3. Создать стек ST, изначально пустой.
4. Положить $j = 1$ и $k = 1$.
5. Для всех $i \in 1..2g + 1$
   - Если $B[i] =$ '('
     (a) Положить на вершину стека ST число $L[j]$
     (b) Увеличить j на единицу
   - Иначе
     (a) Забрать с вершины стека ST число y_{right}
     (b) Забрать с вершины стека ST число y_{left}
     (c) Добавить в R правило $X_{k+\sigma} \to X_{y_{left}}X_{y_{right}}$.

(d) Положить на вершину стека ST число $k + \sigma$

(e) Увеличить k на единицу

6. ПП с правилами вывода из R – искомая ПП.

Доказательство корректности данного алгоритма можно посмотреть в [13].

8 Экспериментальные результаты

8.1 Условия экспериментов

Очевидно, что природа исходного текста влияет как на скорость, так и на коэффициент сжатия. В экспериментах использовались тексты следующих трех типов:

- последовательность плохо сжимаемых слов над пятибуквенным алфавитом, используемую в теореме 5.3;
- случайные строки над четырехбуквенным алфавитом;
- последовательности ДНК, взятые из открытого банка ДНК Японии (`http://www.ddbj.nig.ac.jp/`).

Выбор этих типов текстов был обусловлен такими соображениями. Слова из представленной последовательности имеют искусственную структуру, и размер LZ-факторизаций этих слов максимален. Следовательно, они, в каком-то смысле, являются наихудшим входом для задачи построения ПП. Случайные строки также сжимаются плохо, но случайные строки на практике встречаются чаще, чем искусственно построенные тексты. Наконец, последовательности ДНК – это практически важный класс хорошо сжимаемых строк.

Мы сравниваем рассмотренные алгоритмы построения ПП между собой. Исходный код проекта доступен по адресу `http://code.google.com/p/overclocking/`. Все алгоритмы запускались в одинаковых условиях на компьютере со следующими параметрами: процессор Intel Core i7-2600 с

тактовой частотой 3.4GHz, 8ГБ оперативной памяти, операционная система Windows 7 x64.

8.2 Результаты экспериментов

Основные результаты экспериментов для случайных строк и последовательностей ДНК представлены графиками на рис. 4–20. На этих графиках приняты следующие обозначения для данных, относящихся к различным алгоритмам:

- ▫ – алгоритм Лемпеля-Зива с окном сжатия 32КБ;
- ■ – алгоритм Лемпеля-Зива с бесконечным окном сжатия;
- ▪ – алгоритм Лемпеля-Зива-Велча;
- ○ – алгоритм Риттера;
- ◍ – алгоритм Риттера с эвристикой по оптимизации порядка конкатенаций;
- ● – алгоритм Риттера с улучшенной эвристикой по оптимизации порядка конкатенаций;
- ▲ – многопоточный алгоритм построения ПП (четыре потока);
- △ – алгоритм построения рандомизированной ПП;
- ▲ – LCA-online алгоритм.

По оси абсцисс на всех графиках откладывается длина сжимаемого текста.

Также стоит зафиксировать, как именно были проинтерпретированы предложенные в параграфе 3.2 эвристические оптимизации алгоритма Риттера. Напомним, что в описании эвристик говорилось, что оптимизировать порядок вращений очень большого количества деревьев неоправданно. Так как выигрыш от оптимизации будет меньше, чем время уже потраченное на поиск этой оптимизации.

Для «простой» эвристики был зафиксирован параметр $\ell = 10$ – максимальный размер группы, допускаемой до поиска оптимизации. Таким образом, как только накопится хотя бы 10 деревьев, незамедлительно ищется и выполняется оптимальный порядок их конкатенации. Для «улучшенной» эвристики было зафиксировано два параметра $\ell = 100$ и $k = 10$ – максимальный размер группы, допускаемой до поиска оптимизации, и размер промежуточного деления при поиске порядка конкатенации (см. описание данной эвристики). Данные коэффициенты были подобраны эмпирическим путем.

Основными анализируемыми параметрами алгоритмов были время работы и коэффициент сжатия. Коэффициент сжатия текста можно определять двумя способами: отношение размеров сжатого и несжатого текста, выраженного в байтах, или отношение размеров сжатого и несжатого представления текста, выраженного в соответствующих единицах (то есть символы — для несжатого текста, правила вывода — для ПП и факторы — для LZ-факторизации).

Дополнительно предоставлена статистика еще по двум параметрам: высота дерева вывода ПП и количество поворотов AVL-дерева, выполненных в процессе построения ПП. Высота дерева вывода ПП является важным параметром, так как участвует в оценках сложности алгоритмов, работающих в ПП. Также оба эти параметра позволяют получить обоснованное объяснение, почему один алгоритм работает медленнее другого.

На рис. 4, 5 и 6 представлены полученные зависимости времени сжатия от размера текста. Видно, что эвристики по оптимизации количества вращений AVL-дерева ускорили алгоритм Риттера: на сжатие потребовалось в среднем в два раза меньше времени. На рис. 9, 10 и 11 показана зависимость количества поворотов AVL-дерева, выполненных при построении ПП. Этот график показывает, что примененные эвристики действительно в 3-4 раза сократили количество поворотов AVL-дерева. А небольшой выигрыш по времени одной эвристики над другой объясняется тем, что нам удалось найти оптимальный порядок конкатенации, потратив меньше времени.

Алгоритм, строящий рандомизированные ПП, почти всегда уступает по скорости всем алгоритмам. Причина здесь, по-видимому, связана с тем, что высота рандомизированного дерева, возвращаемого алгоритмом, существенно больше высоты соответствующих AVL-деревьев. На рис. 12, 13 и 14 видно, что высота AVL-деревьев в экспериментах варьируются от 20 до 30, а высота рандомизированного дерева составляет 50-70. Из-за большей высоты дерева при построении рандомизированных ПП приходится обрабатывать намного больше правил, что сводит на нет весь выигрыш, который возникает за счет простоты поддержания баланса в рандомизированных деревьях.

Многопоточный алгоритм немного обогнал по скорости все алгоритмы, использующие деревья. К сожалению, в рамках данной работы не удалось полноценно реализовать данный алгоритм — пятый шаг алгоритма практически не ускоряется при увеличении количества потоков. Тем не менее остальные шаги были успешно распараллелены, и именно это обеспечило этот выигрыш по времени. Для того, чтобы оценить потенциал алгоритма, посмотрим на зависимость количества итераций многопоточного алгоритма (рис. 7 и 8). На любом из рассматриваемых текстов количество итераций мало (от 34 до 56), при этом факторы почти равномерно распределяются по итерациям (за исключением первых и последних итераций). Из чего можно сделать вывод, что третий и четвертый шаги алгоритма замечательно выполняются в несколько потоков.

Также отметим, что многопоточный алгоритм работал безобразно медленно на «плохих» строках. Но это и неудивительно, ведь в теореме 5.3 доказано, что алгоритм выполняет очень много итераций на этих текстах. Таким образом, десятки и сотни тысяч раз происходит вызов единственной асинхронной операции и ожидание ее выполнения, именно это и объясняет столь долгое выполнение алгоритма.

Лидером по скорости работы оказался LCA-online алгоритм. Отметим также, что остальные алгоритмы на вход получают не текст, а его LZ-факторизацию, которую еще надо построить. Время на построение LZ-факторизации не было учтено при построении данного графика. Таким образом, LCA-online действительно превосходит остальные алгоритмы по скорости.

На рис. 15, 16 и 17 коэффициенты сжатия, достигаемый алгоритмами построения ПП, сравниваются с коэффициентами сжатия алгоритмов из семейства Лемпеля-Зива. Интересно, что отношение коэффициента сжатия алгоритмов, строящих ПП, к коэффициенту сжатия алгоритмов из семейства Лемпеля-Зива практически не зависит ни от типа сжимаемого текста, ни от его длины.

Алгоритмы, строящие AVL-деревья, обеспечивают практически одинаковый коэффициент сжатия (отличия составляют доли процента), поэтому на

рисунках точки, соответствующие этим алгоритмам слились в одну. Размер полученных ПП примерно в два раза больше размера LZ-факторизаций, полученных алгоритмом Лемпеля-Зива с бесконечным окном.

У алгоритма, строящего рандомизированные ПП, коэффициент сжатия заметно хуже, чем у алгоритмов, строящих AVL-деревья. Это также можно объяснить большой высотой рандомизированных деревьев, ведь во всех оценках на размер результирующей ПП участвует именно высота дерева.

LCA-online алгоритм показал лучший результат на последовательностях ДНК и случайных строках. Результат его работы на 2-3% лучше, чем результат алгоритма Риттера. А вот на «плохих» строках данный алгоритм чуть-чуть проиграл алгоритму Риттера.

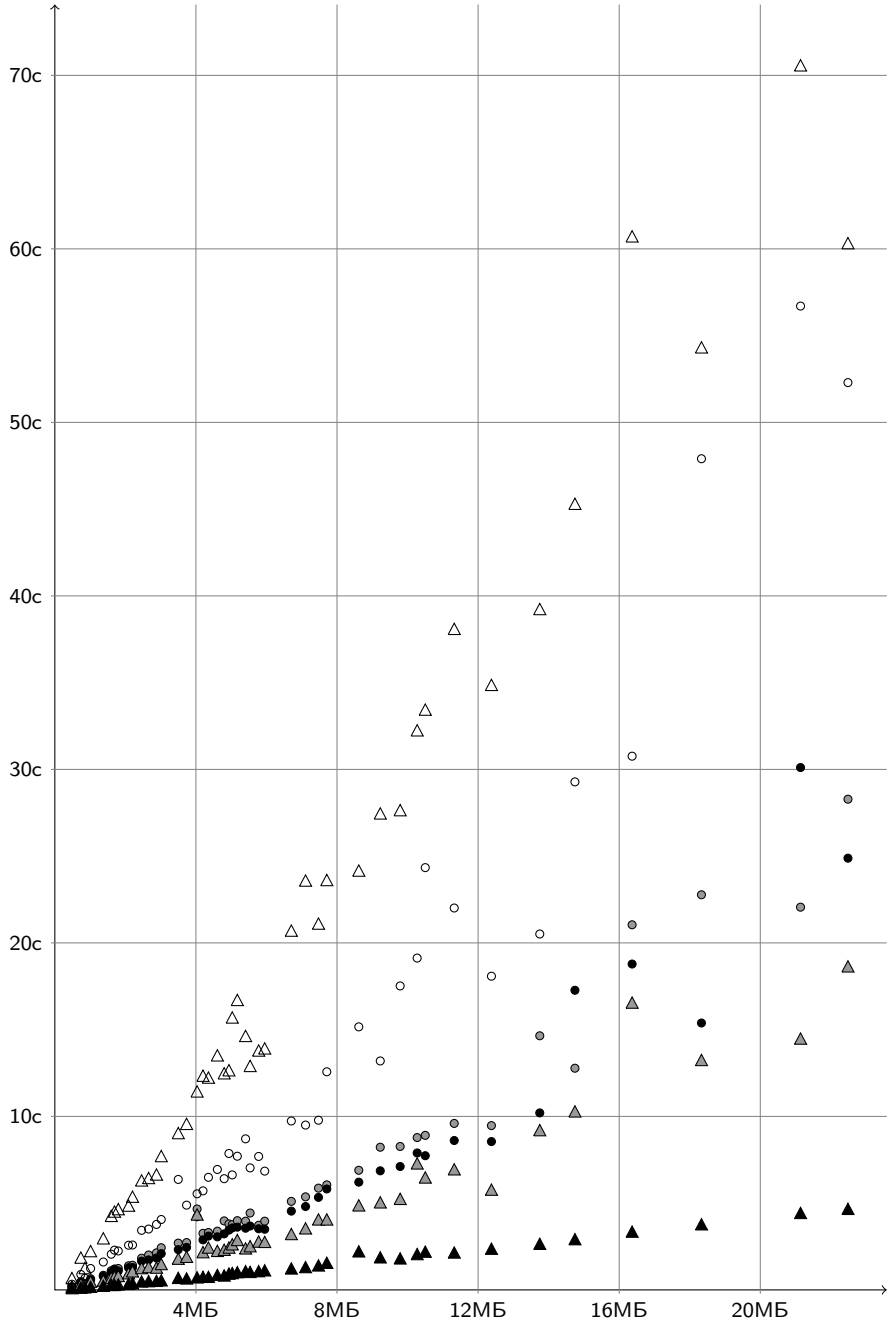

Рис. 4: Время работы алгоритмов построения ПП на последовательностях ДНК

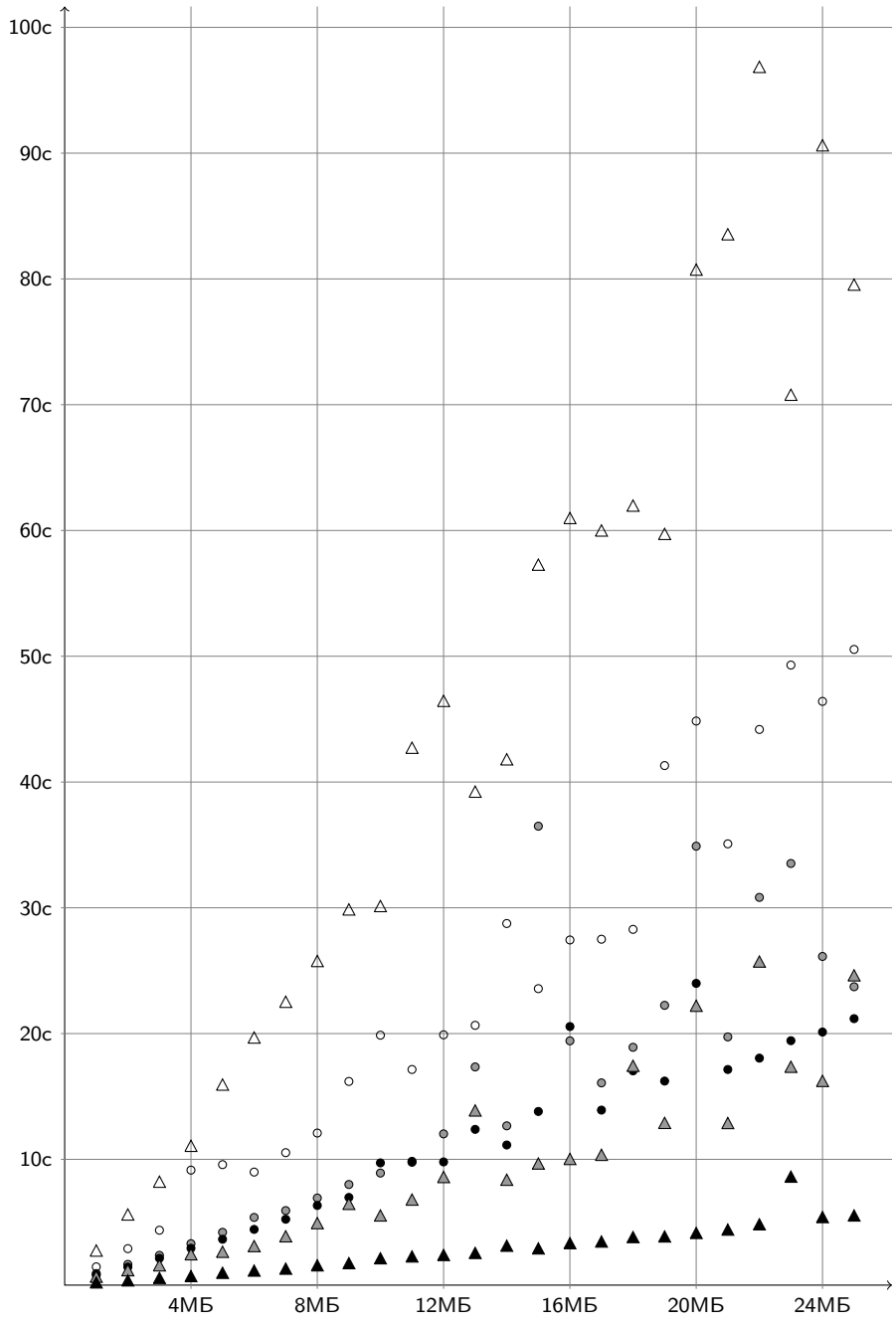

Рис. 5: Время работы алгоритмов построения ПП на случайных строках

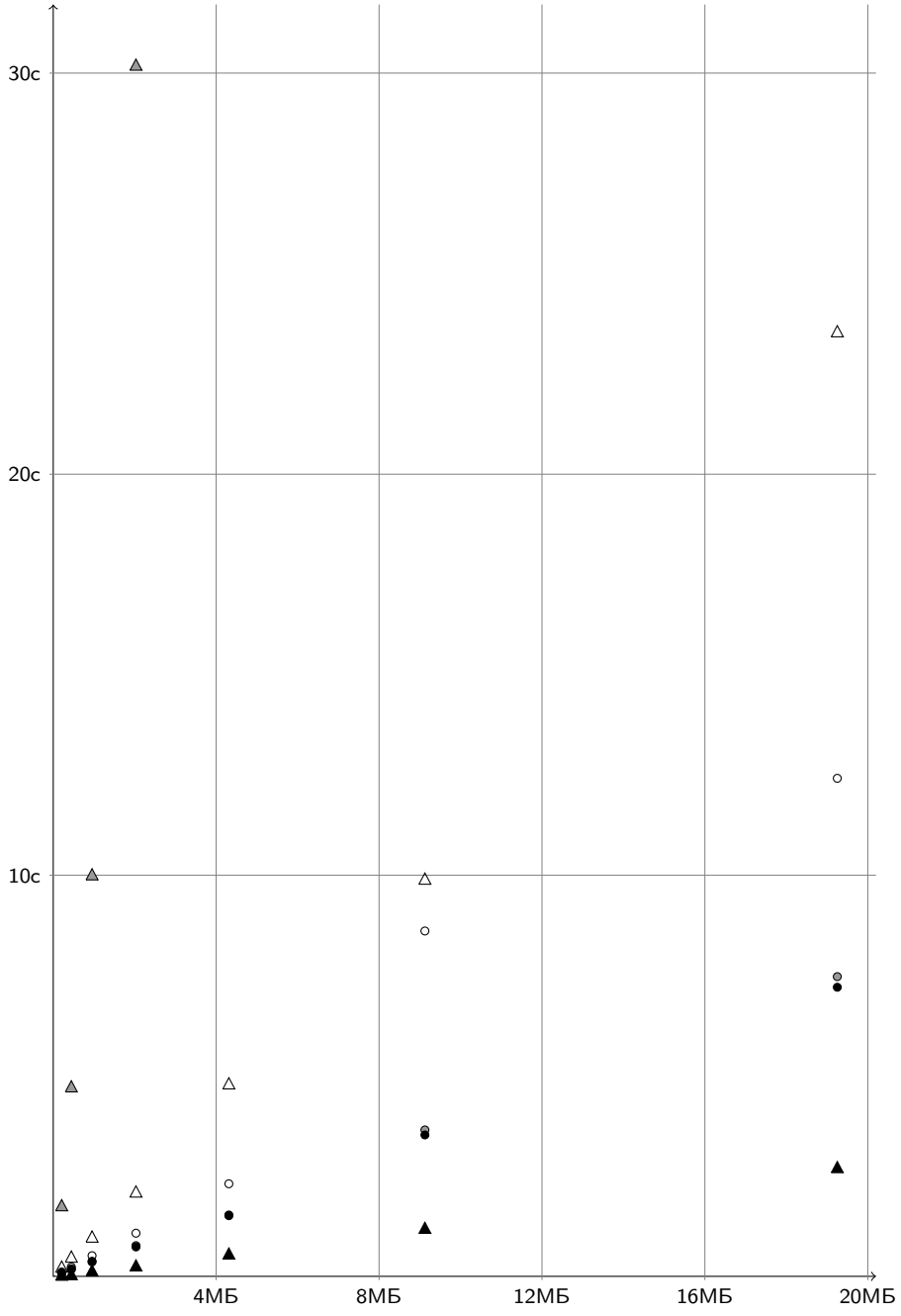

Рис. 6: Время работы алгоритмов построения ПП на «плохих» строках

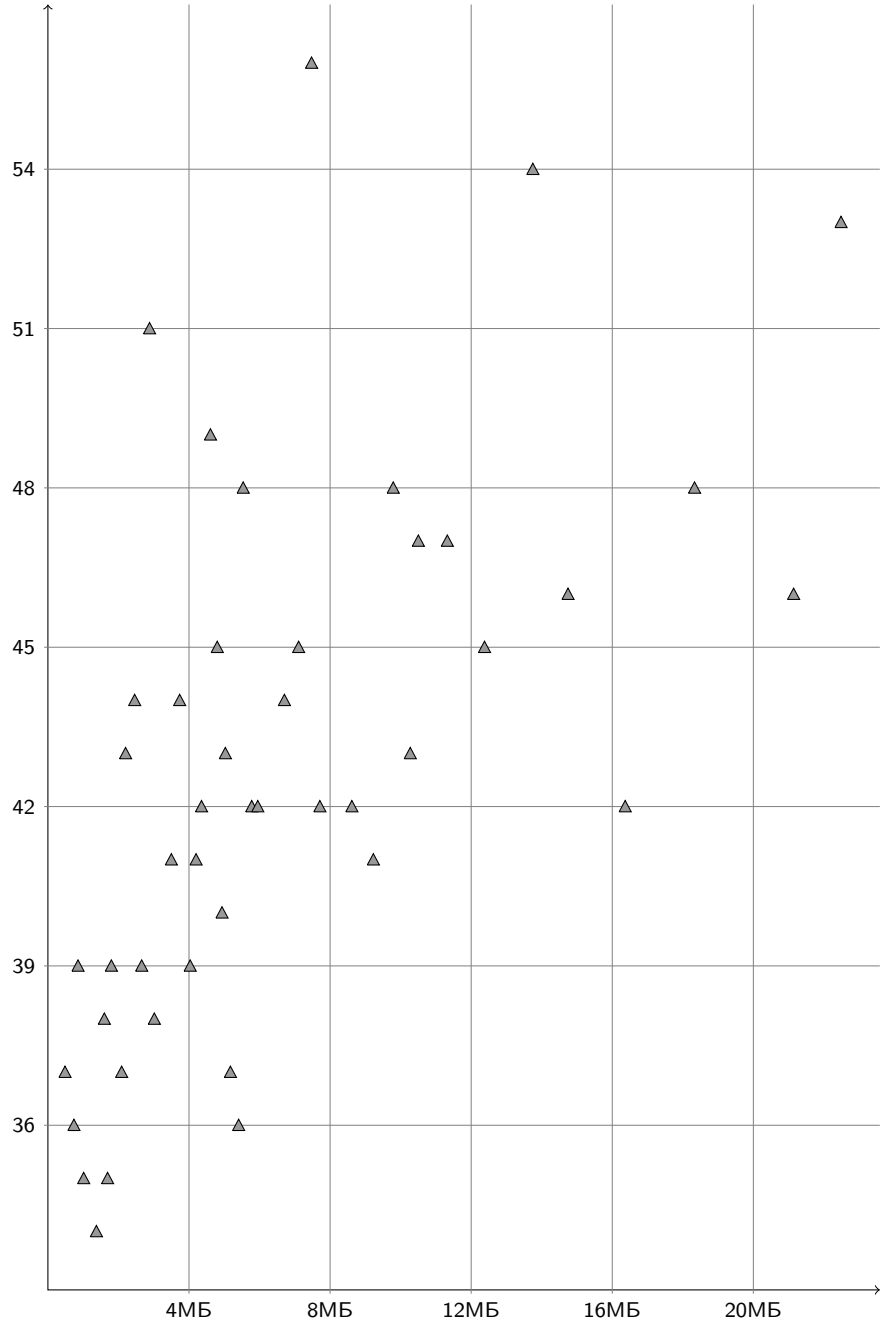

Рис. 7: Количество итераций многопоточного алгоритма построения ПП на последовательностях ДНК

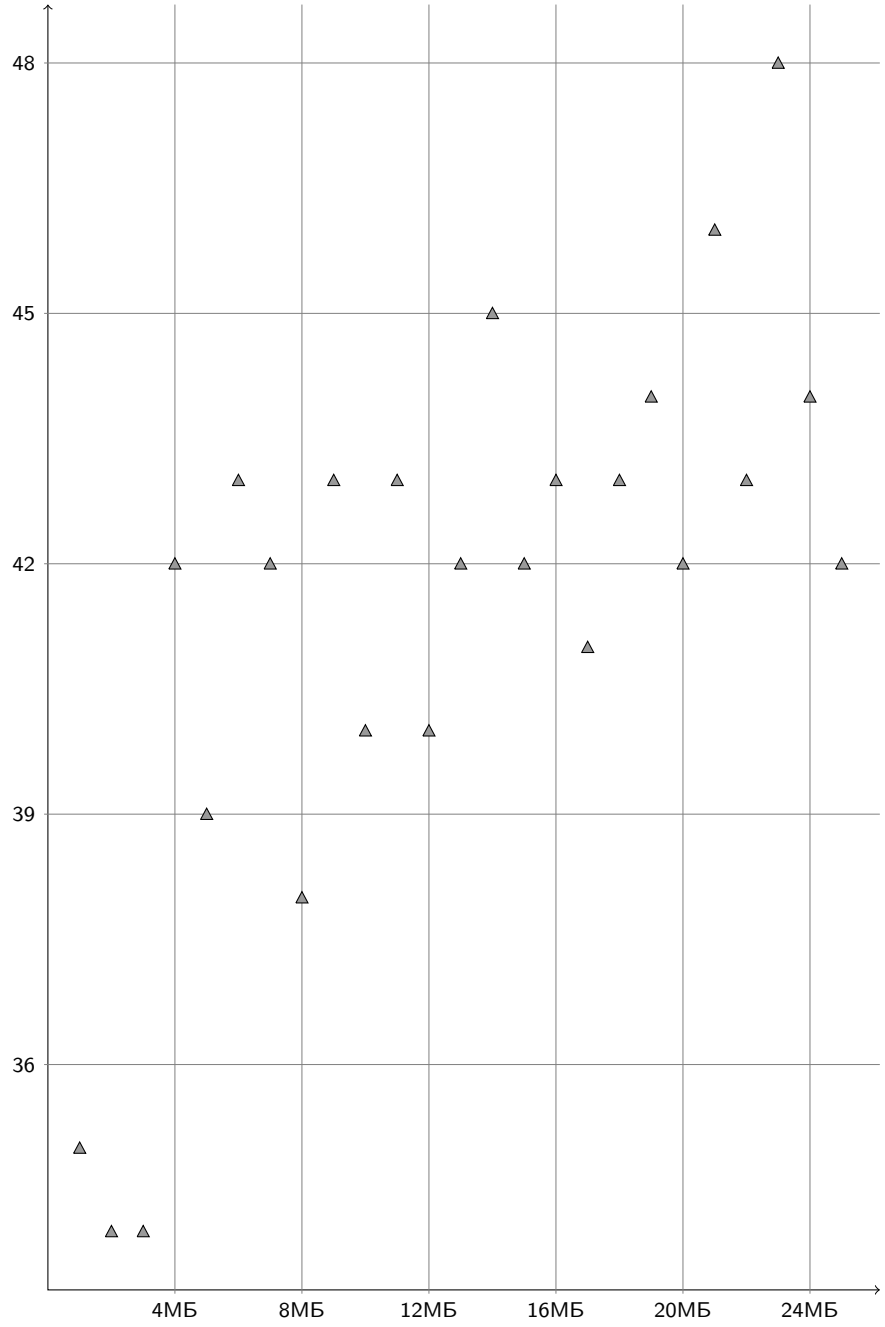

Рис. 8: Количество итераций многопоточного алгоритма построения ПП на случайных строках

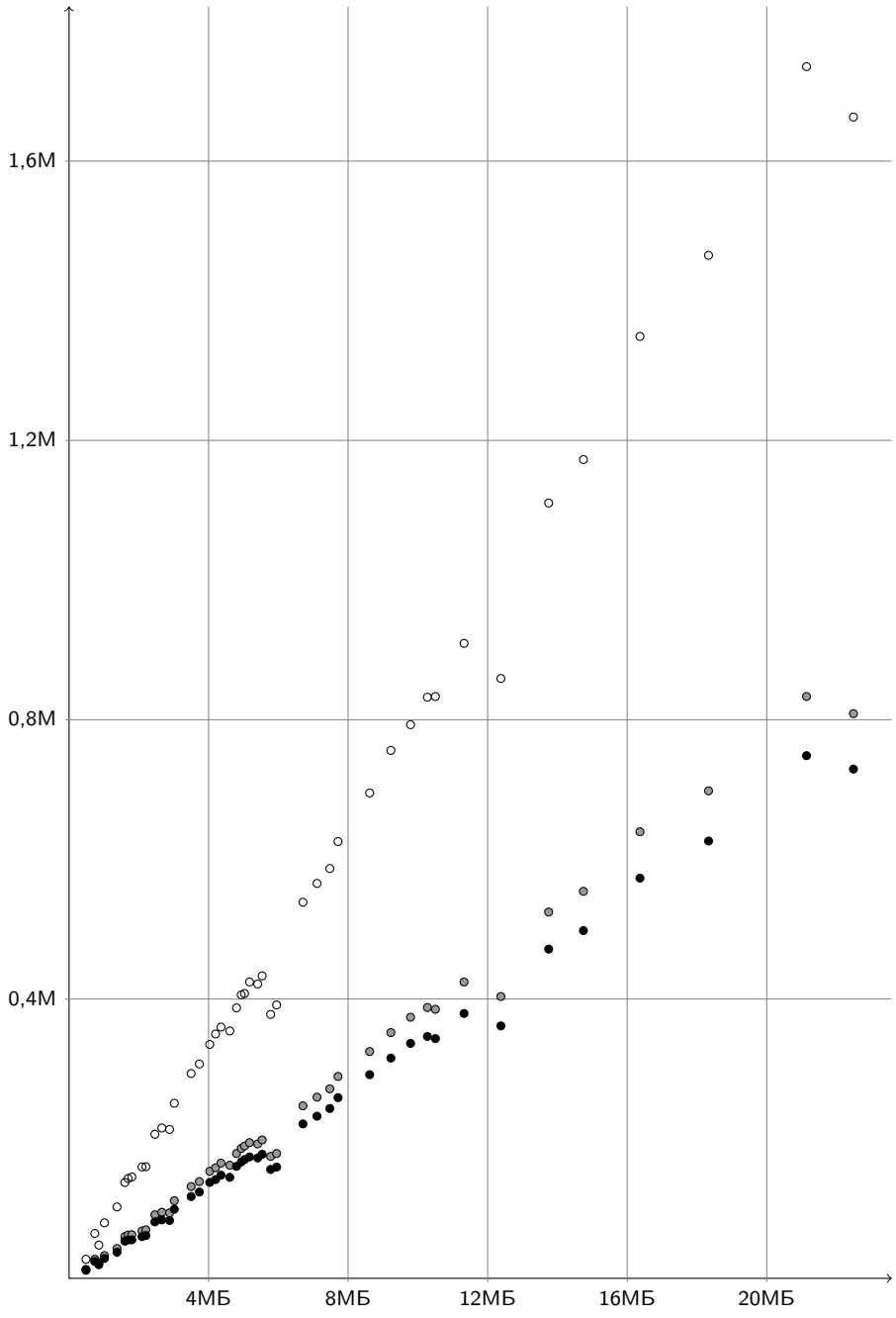

Рис. 9: Количество поворотов AVL-дерева на последовательностях ДНК

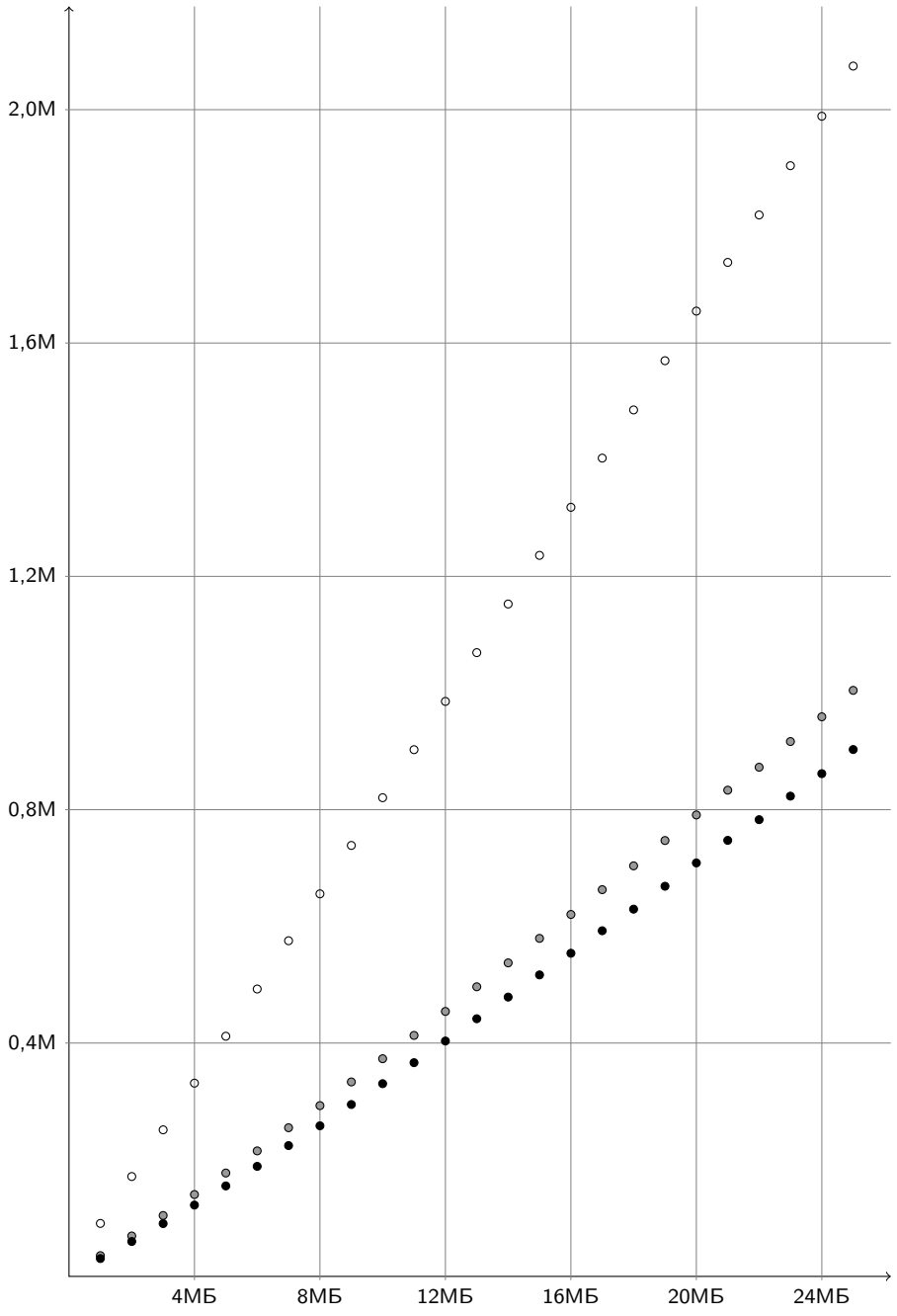

Рис. 10: Количество поворотов AVL-дерева на случайных строках

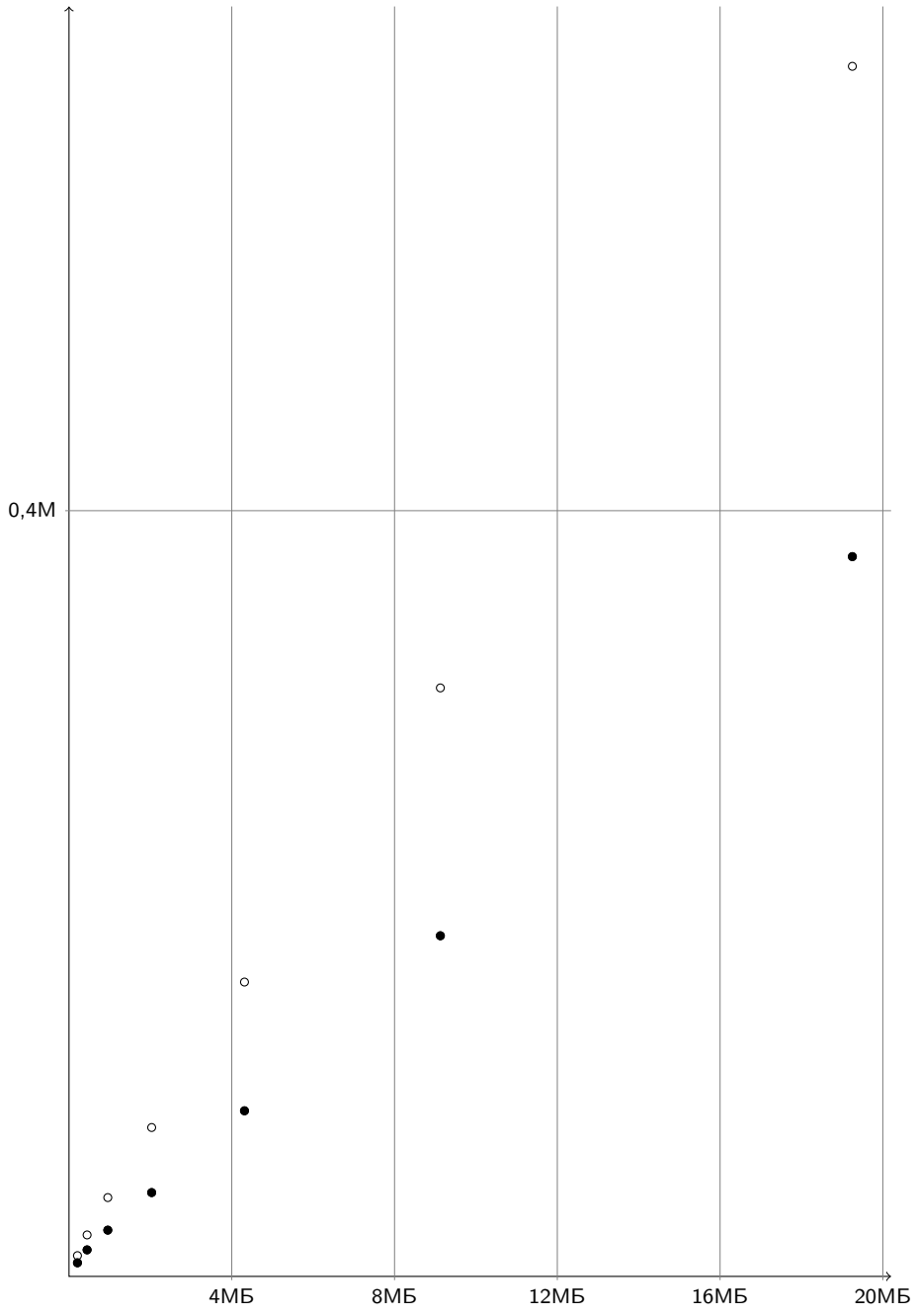

Рис. 11: Количество поворотов AVL-дерева на «плохих» текстах

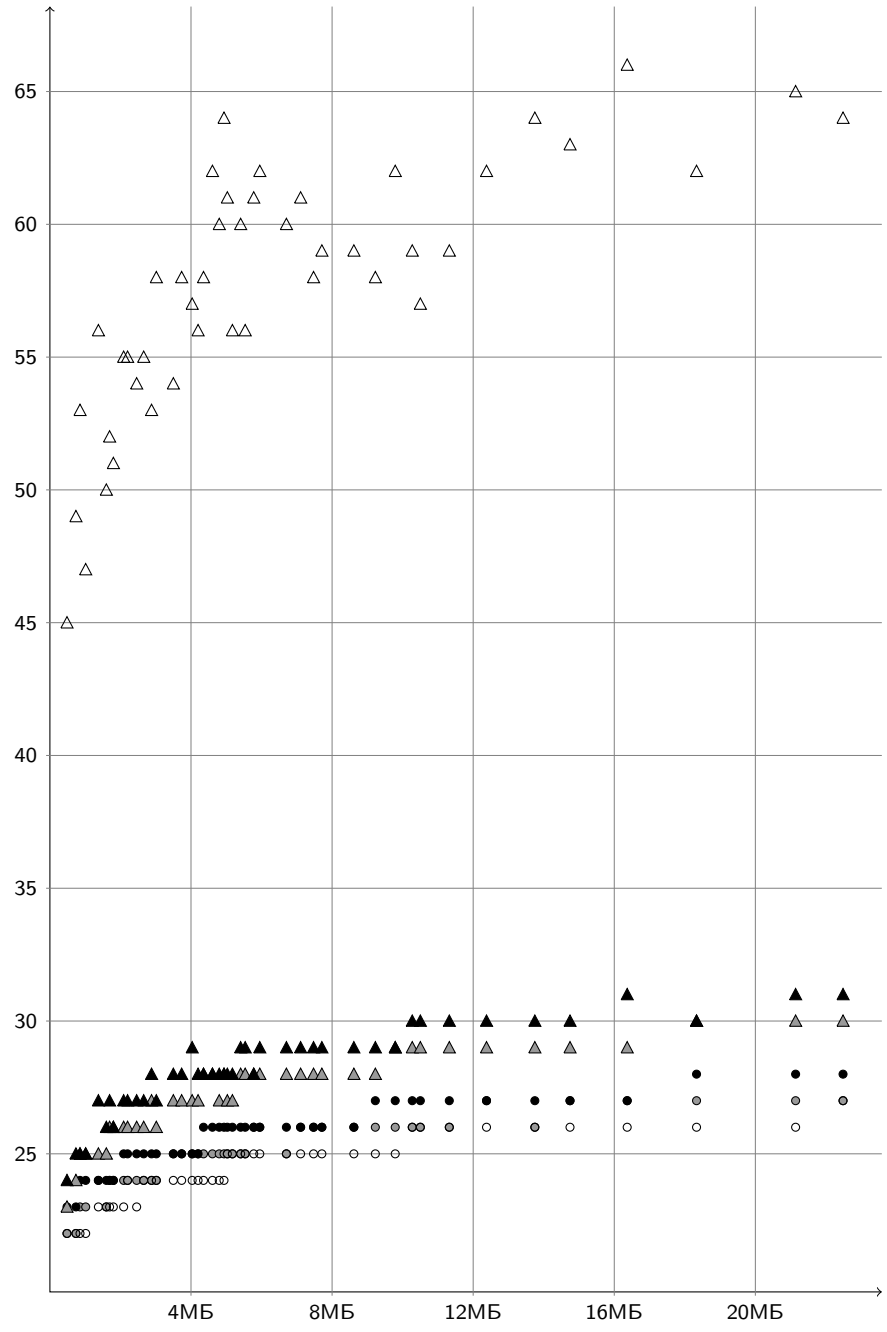

Рис. 12: Высота построенной ПП на последовательностях ДНК

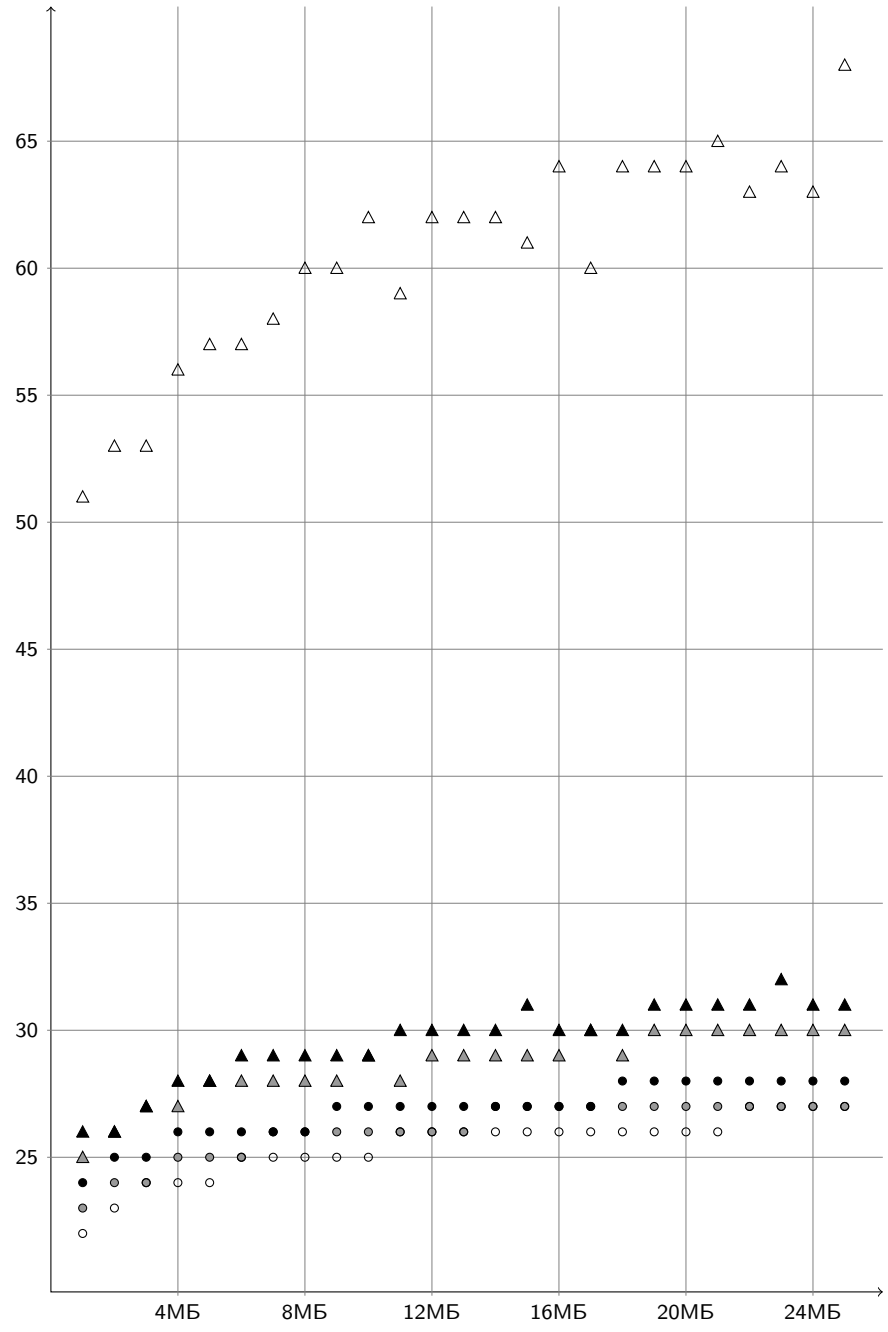

Рис. 13: Высота построенной ПП на случайных строках

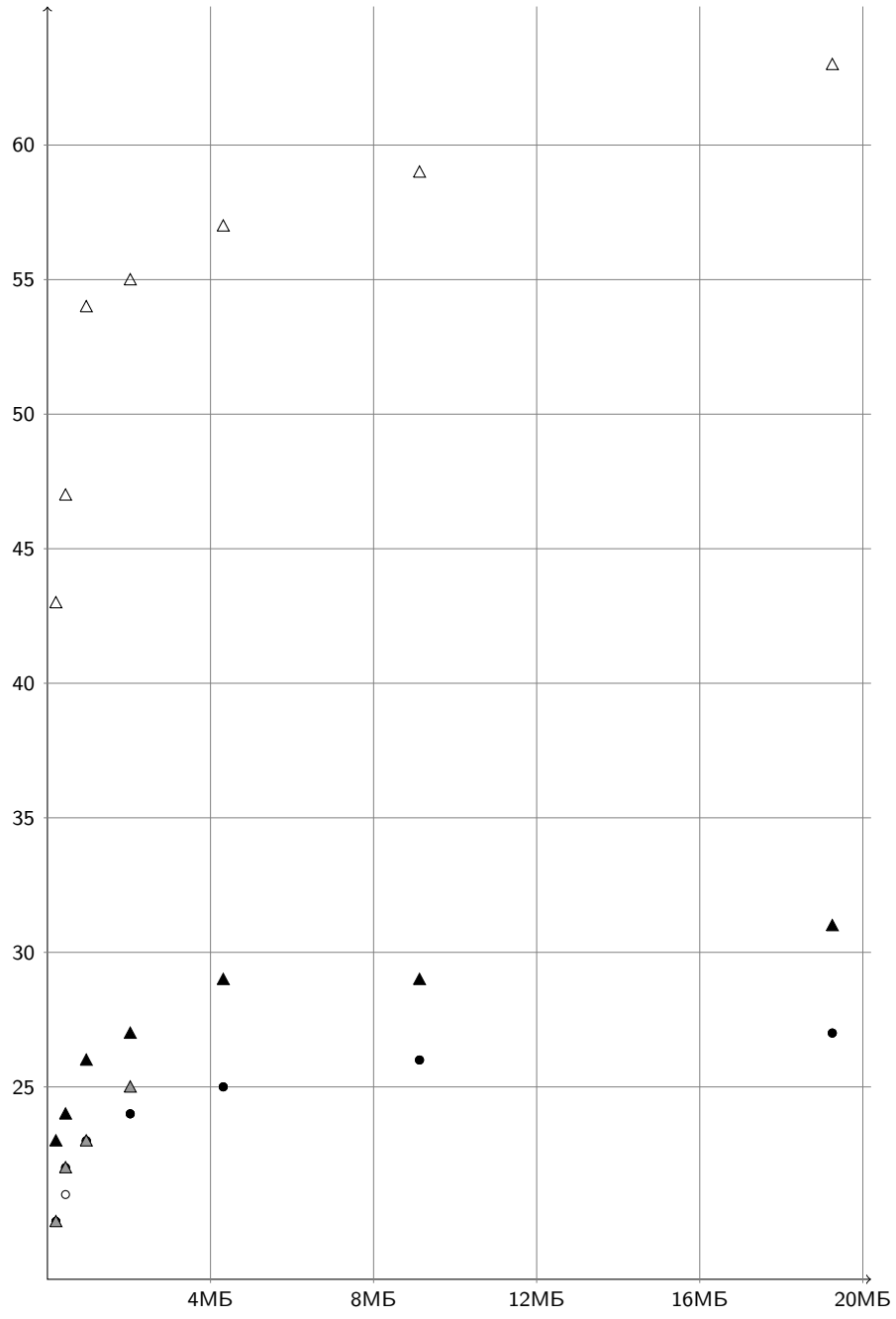

Рис. 14: Высота построенной ПП на «плохих» текстах

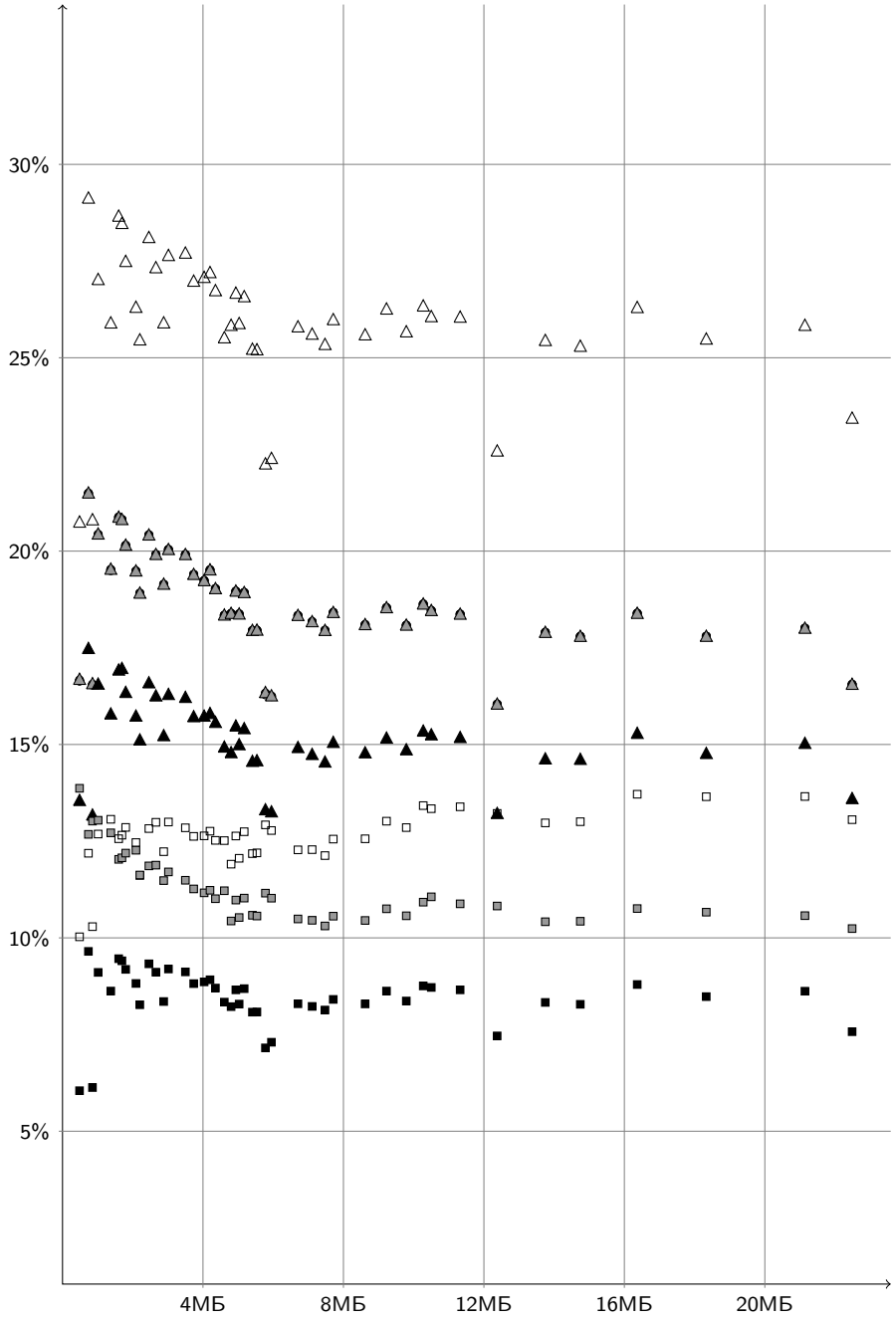

Рис. 15: Отношение размера сжатого представления к длине текста на последовательностях ДНК

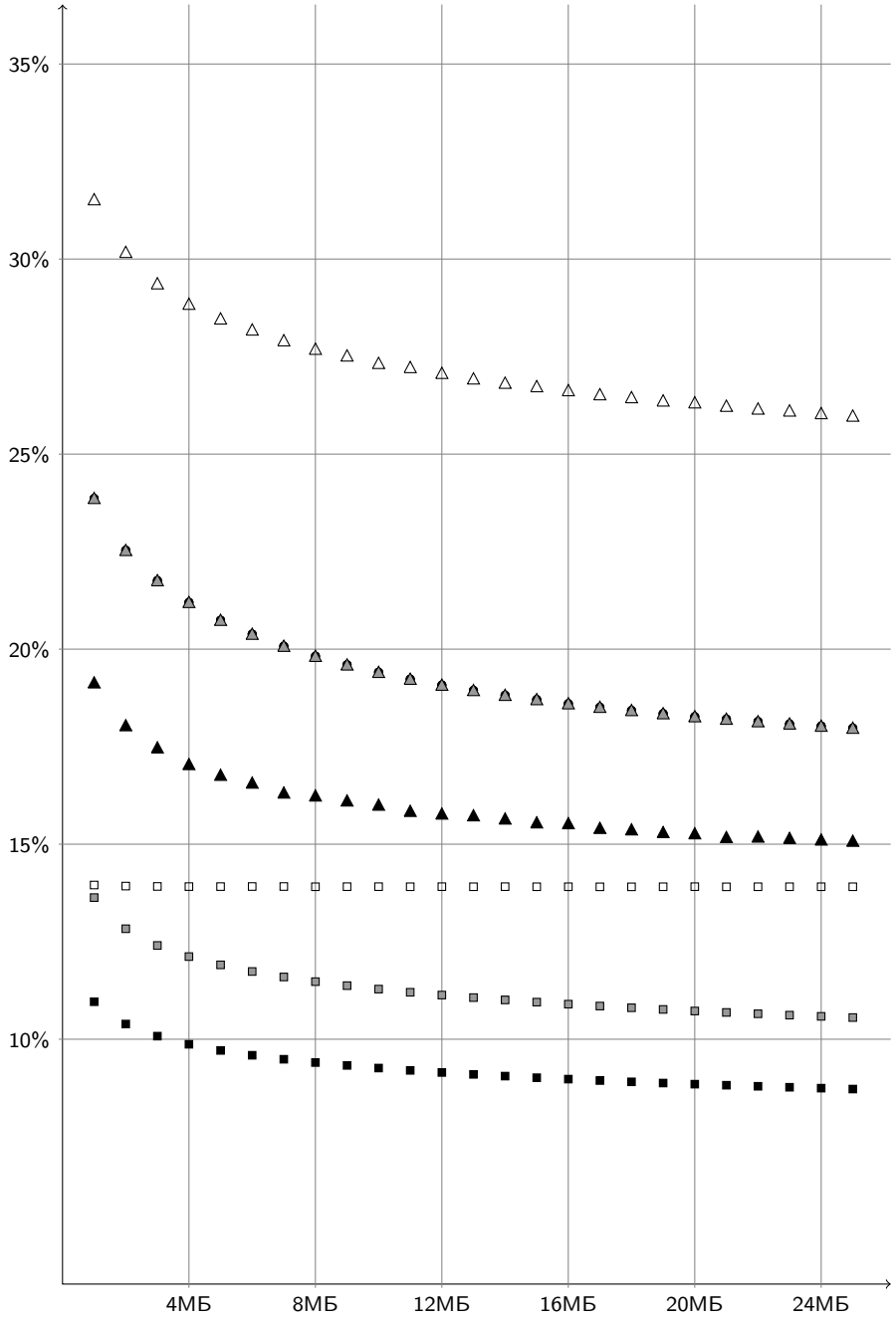

Рис. 16: Отношение размера сжатого представления к длине текста на случайных строках

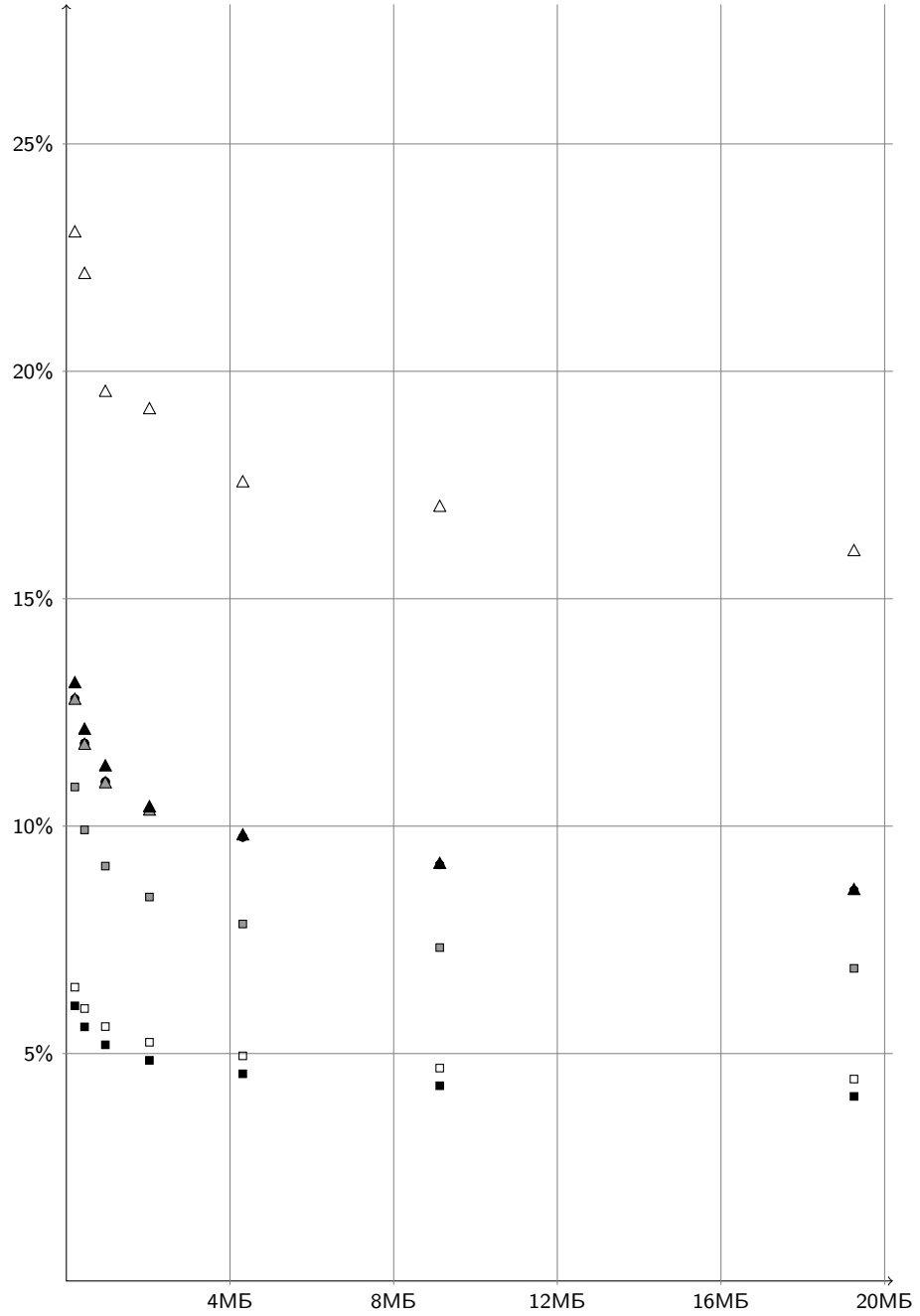

Рис. 17: Отношение размера сжатого представления к длине текста на «плохих» текстах

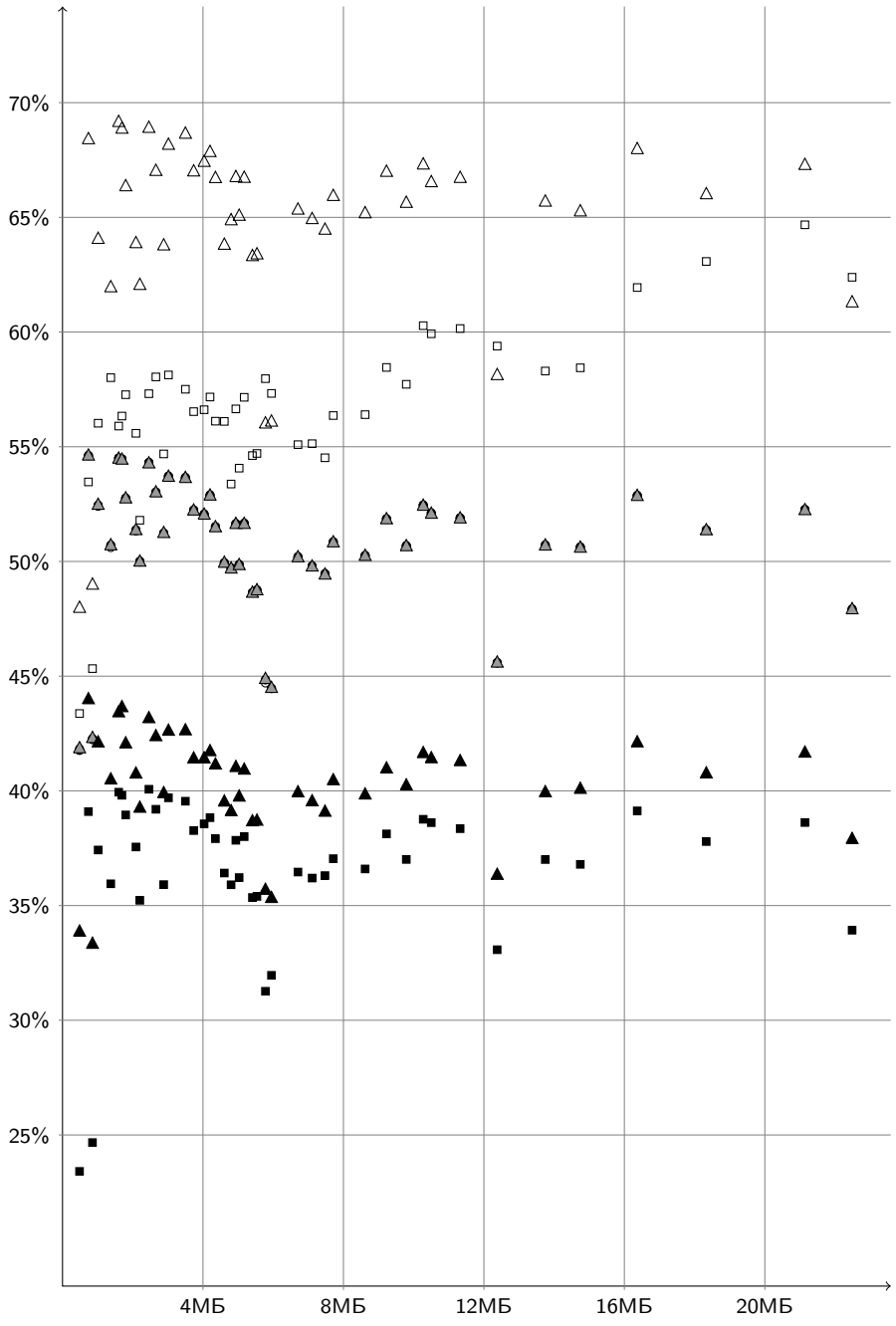

Рис. 18: Коэффициент сжатия на последовательностях ДНК

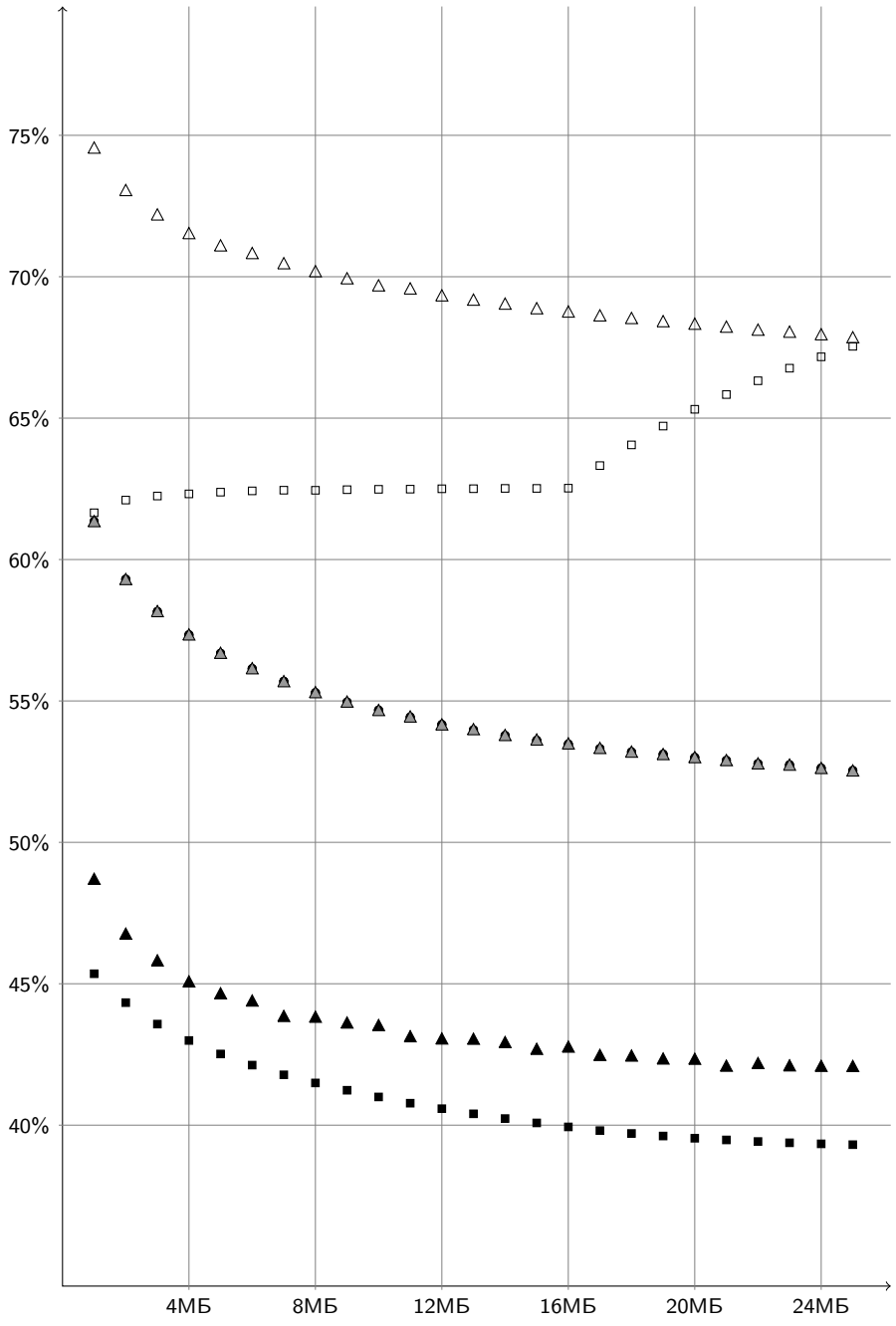

Рис. 19: Коэффициент сжатия на случайных строках

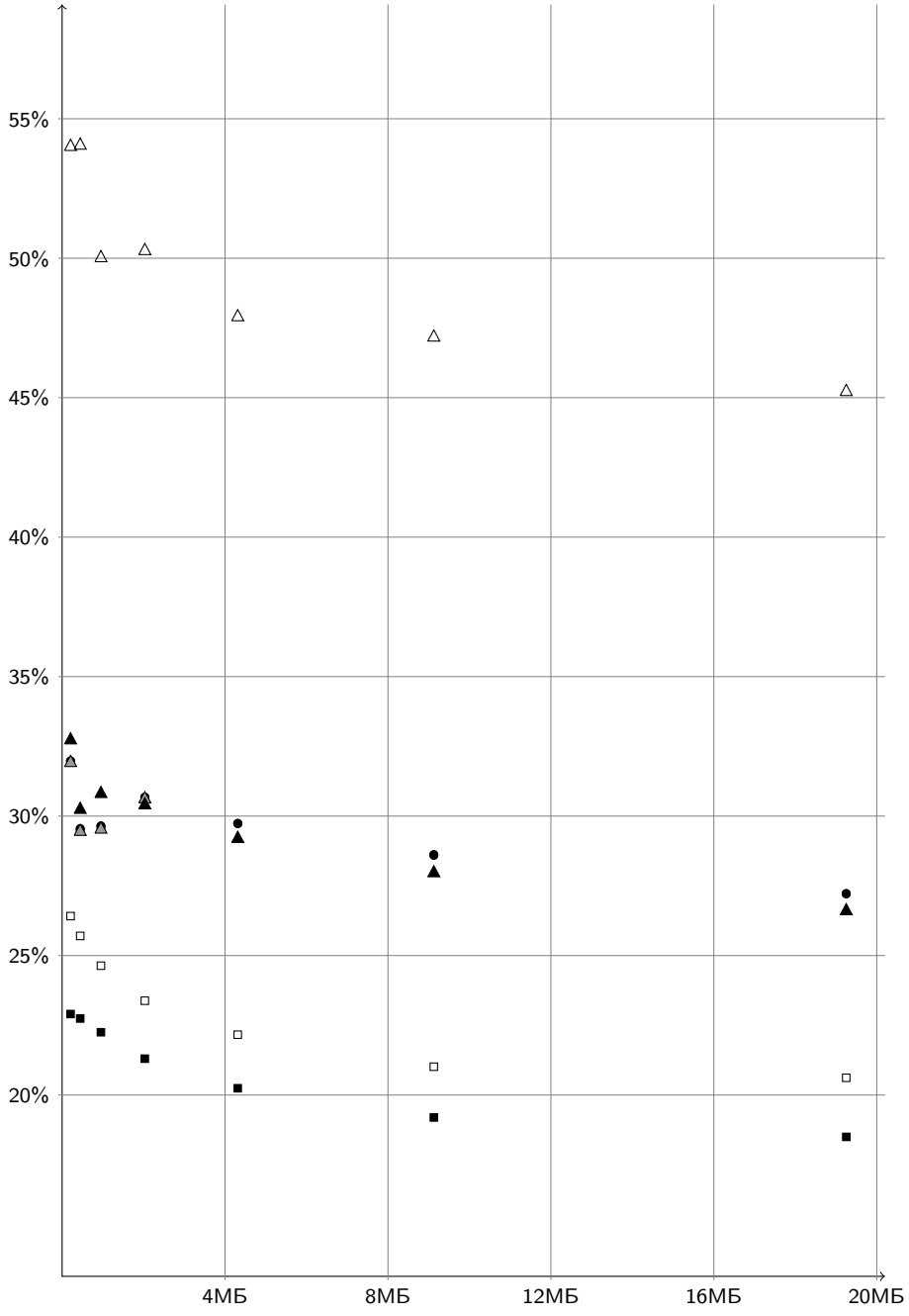

Рис. 20: Коэффициент сжатия на «плохих» текстах

9 Выводы и дальнейшие перспективы

Можно выделить два класса алгоритмов построения ПП.

- Класс алгоритмов, основанных на представлении деревьев вывода в виде сбалансированных бинарных деревьев и строящих ПП на основе LZ-факторизаций. Эти алгоритмы сложны и требовательны к ресурсам, зато гарантируют построение $O(\log n)$-приближения.
- Класс алгоритмов, строящих ПП на основе текста. Эти алгоритмы как правило просты, зато для них доказана худшая асимптотика на размер, получившей ПП.

В этой работе основное внимание было уделено алгоритмам из первого класса. Главным недостатком этих алгоритмов является то, что дерево занимает большое количество памяти. При увеличении размера сжимаемого текста возникает необходимость хранения дерева не в оперативной памяти, а на жестком диске, что, естественно, пагубно сказывается на времени работы алгоритма. В данном контексте эвристики по оптимизации количества вращений AVL-дерева сыграют свою роль и приведут алгоритм Риттера к еще большему ускорению, так как уменьшат количество обращений к жесткому диску.

Как показали эксперименты, замена AVL-дерева другим типом деревьев не увенчалась успехом. Алгоритм, использующий рандомизированные деревья, проиграл всем алгоритмам с AVL-деревом по обоим параметрам (время сжатия и коэффициент сжатия). Скорее всего, и другие типы деревьев также не приведут к улучшению параметров алгоритма, так как основное влияние на качество алгоритма имеет высота деревьев, а свойства AVL-деревьев накладывают наиболее жесткие ограничения на высоту.

Лучшие характеристики показал многопоточный алгоритм, хотя и был реализован в рамках данного исследования не в полную силу. При реализации алгоритма многие вещи пришлось реализовывать по-новому, а некоторый функционал так вообще урезать. Например, при выполнении конкатенации деревьев появляется $O(\log n)$ новых правил, тогда как старые остаются, но некоторые из них со временем становятся ненужными. Для

экономии используемой оперативной памяти был реализован собственный менеджер памяти, который отслеживает ненужные правила и переиспользуют занимаемую ими память. Для многопоточной версии алгоритма реализация подобного менеджера памяти требовала гораздо больше сил и пока не была реализована. Более того, при переходе с оперативной памяти на жесткий диск мы также столкнемся с проблемой. Совершенно очевидно, что для приемлемой скорости работы необходим некоторый кэш данных, хранящихся на жестком диске. Реализация подобного кэша для многопоточного алгоритма, скорее всего, также потребует огромных усилий и навыков. Таким образом, предложенный многопоточный алгоритм имеет свой потенциал и, скорее всего, сможет конкурировать с LCA-online алгоритмом по скорости. Но исправление всех перечисленных проблем потребует огромных усилий.

Среди алгоритмов из второго класса был выбран один с наилучшей оценкой на размер результирующей ПП. Теоретическая оценка LCA-online алгоритма была заведомо хуже оценок алгоритмов из класса, использующих сбалансированные бинарные деревья. Эксперименты же показали неожиданный результат — на практике этот алгоритм оказался лучшим по качеству сжатия. В дополнение этот алгоритм обладает рядом преимуществ: данный алгоритм имеет наибольшую скорость сжатия, в процессе построения не используется сложных структур данных, а также перед началом выполнения алгоритма не требуется построение LZ-факторизации. Безусловно, вопрос более тщательного изучения оценки качества сжатия данного алгоритма кажется весьма перспективным, как, пожалуй, единственного недостатка алгоритма.

Судя по доказательству данной оценки, при сжатии каждого фактора каждая очередь добавляет не более $\log n$ новых правил. На основании этого делается вывод, что размер построенной ПП не превышает $O(k \log^2 n)$. Вероятно, именно в этом месте доказательства происходит огрубление оценки, и, возможно, с помощью амортизационного анализа можно получить более точную оценку на общее количество правил. Альтернативой этому может оказаться, что оценка $O(log^2 n)$ верная, но достигается только на текстах

определенной структуры, в то время как для практически интересных текстов эта оценка может быть улучшена.

Таким образом, если удастся теоретически улучшить оценку для LCA-online алгоритма, то можно твердо сказать, что практическое применение алгоритмов из первого класса лишено всякого смысла.

Хотя размер прямолинейных программ больше размера LZ-факторизаций (удалось достигнуть превышения всего в 1.5-2 раза), благодаря своей структуре размер сериализованных прямолинейных программ наилучшего алгоритма очень близок к размеру сериализованных LZ-факторизаций. Таким образом, сжатие с помощью прямолинейных программ лишь немного уступает сжатию алгоритмами из семейства Лемпеля-Зива, но, в то же время, позволяет выполнять различные алгоритмы с текстами без их распаковки.

Список литературы

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

[2] *Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa*, Pattern matching in text compressed by using antidictionaries, Lect. Notes Comput. Sci., 1645 (1999), 37–49.

[3] *W. Rytter*, Application of Lempel-Ziv factorization to the approximation of grammar-based compression, Theor. Comput. Sci., 302 (2003), 211–222.

[4] *J. Ziv, A. Lempel*, A universal algorithm for sequential data compression, IEEE Trans. Information Theory, 23 (1977), 337–343.

[5] *J. Ziv, A. Lempel*, Compression of individual sequences via variable-rate coding, IEEE Trans. Information Theory, 24 (1978), 530–536.

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

[7] *Y. Lifshits*, Processing compressed texts: A tractability border, Lect. Notes Comput. Sci., 4580 (2007), 228–240.

[8] *W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, K. Hashimoto*, Computing longest common substring and all palindromes from compressed strings, Lect. Notes Comput. Sci., 4910 (2008), 364–375.

[9] *A. Tiskin*, Faster subsequence recognition in compressed strings, Journal of Mathematical Sciences, 158 (2009), 759–769.

[10] *M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat*, The smallest grammar problem, IEEE Trans. Information Theory, 51 (2005), 2554–2576.

[11] *I. Burmistrov, L. Khvorost*, Straight-line programs: a practical test, Proc. Int. Conf. Data Compression, Commun., Process., CCP (2011), 76–81.

[12] *I. Burmistrov, A. Kozlova, E. Kurpilyansky, A. Khvorost*, Straight-line programs: a practical test, Journal of Mathematical Sciences 192.3 (2013): 282-294.

[13] *S. Maruyama, H. Sakamoto, M. Takeda*, An online algorithm for lightweight grammar-based compression, Algorithms, 2012.

[14] *D. Knuth*, The Art of Computing, Vol. III Second edition, 1998.

[15] *R. Seidel, C. Aragon*, Randomized search trees, Algorithmica 16 (1996), 464–497.