Содержание

1 Постановка задачи и структура работы

Очевидна потребность в алгоритмах, способных эффективно обрабатывать большие объемы данных. Поскольку для хранения и передачи больших объемов данных часто используются различные сжатые представления, один из возможных подходов к указанной проблеме состоит в разработке алгоритмов, оперирующих непосредственно со сжатыми представлениями.

Ясно, что алгоритмы, работающие со сжатыми представлениями, существенным образом зависят от механизма сжатия. Имеется много разных методов сжатого представления данных: коллаж-системы [6], представления с помощью антисловарей [7], прямолинейные программы (кратко ПП) [8], групповое кодирование [5] и т. д. Сжатие текста с помощью контекстно свободных грамматик (таких, как ПП) выделяется среди прочих методов двумя обстоятельствами. Во-первых, грамматики обеспечивают хорошо структурированное сжатое представление, что удобно для последующей алгоритмической обработки. Во-вторых, сжатие с помощью ПП полиномиально эквивалентно сжатию данных с помощью широко применяемых на практике алгоритмов из семейства алгоритмов Лемпеля-Зива (таких, как, например, LZ77 [14], LZ78 [15], LZW [16]). Полиномиальная эквивалентность здесь понимается в следующем смысле: существует полиномиальная зависимость между размером ПП, выводящей данный текст S, и размером словаря, построенным алгоритмом Лемпеля-Зива для S, см. [8].

Существует довольно большой класс задач, для которых разработаны алгоритмы со временем работы, полиномиальным относительно размера сжатого представления текста с помощью ПП. К этому классу относятся, например, задачи **Поиск образца в тексте** [11], **Наибольшая общая подстрока** [12], считающая версия задачи **Поиск всех палиндромов** [12], некоторые версии задачи **Наибольшая общая подпоследовательность** [13]. В то же время константы, которые скрываются за «О большим» в имеющихся оценках сложности таких алгоритмов, как правило, очень велики. Кроме того, упомянутая выше полиномиальная связь между размером ПП, выводящей данный текст S, и размером LZ77-словаря для

S еще не гарантирует, что ПП на практике обеспечивает достаточно высокую степень сжатия. Поэтому вопрос о том, существуют ли методы сжатия, основанные на ПП и подходящие для практического применения, требует дополнительного исследования.

Риттер в [8] предложил алгоритм построения ПП, в котором деревья вывода ПП являются AVL-деревьями. AVL-деревья имеют логарифмическую относительно количества вершин высоту, за счет этого размер получающейся ПП достаточно мал. В данной работе предложен алгоритм построения ПП, использующий декартовы деревья. Декартовы деревья также являются сбалансированными деревьями, но балансировка достигается не так как в AVL-деревьях. По сути балансировка происходит сама собой за счет случайно выбранных приоритетов, тогда как в AVL-деревьях выполняются различные повороты, направленные на балансировку дерева. Таким образом, используя другую структуру данных, производительность алгоритма может вырасти.

Структура работы такова. Во второй главе вводятся основные определения о строках и ПП. В третьей главе описываются два алгоритма построения ПП, использующие AVL-деревья – *алгоритм Риттера* [8] и *модифицированный алгоритм Риттера* [9]. В в четвертой проведен обзор структуры данных *декартово дерево*, а также трех ее модификаций таких как *декартовы деревья по неявному ключу*, *рандомизированные бинарные деревья поиска* и *персистентные декартовы деревья*. В пятой главе описывается алгоритм построения ПП, основанный на декартовых деревьях. В шестой главе приводятся экспериментальные результаты по сравнению эффективности алгоритмов построения ПП, а также по сравнению степени сжатия, достигаемой с помощью алгоритмов построения ПП и с помощью классических алгоритмов сжатия. Итоги подводятся в седьмой главе.

2 Обозначения и основные понятия

В работе рассматриваются строки над конечным алфавитом Σ. *Длина* строки S равна числу символов из Σ в S и обозначается через $|S|$. *Конкатенация*

двух строк S и S' обозначается через $S \cdot S'$.

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

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

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

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

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*»:

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

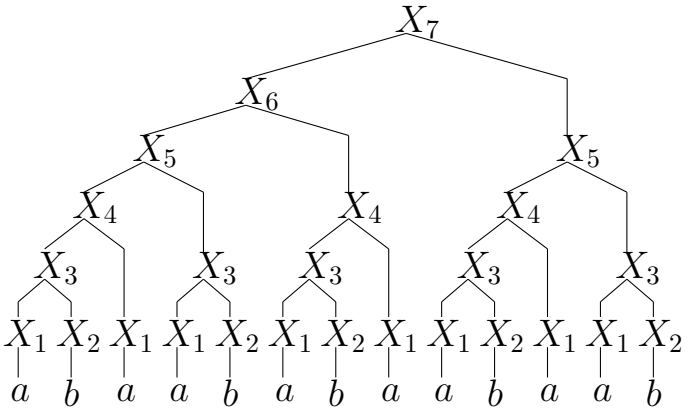

Рис. 1: Дерево вывода грамматики, порождающей «*abaababaabaab*».

Дерево вывода этой грамматики изображено на рисунке 1.

Для некоторого дерева вывода ПП T будем обозначать $S(T)$ — вывод этой ПП. Глубиной узла будем называть количество узлов на пути от этого узла до корня. Высотой дерева будем называть глубину самого глубокого листа.

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

3 Построение прямолинейных программ с помощью AVL-деревьев

3.1 Алгоритм Риттера

Риттер [8] сформулировал алгоритм построения ПП, выводящий данный текст, по LZ-факторизации этого текста. При этом основной структурой данных используется AVL-дерево. AVL-дерево –это двоичное дерево, у каждого внутреннего узла которого высоты сыновей отличаются не более чем на 1.

Теорема 3.1. *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 $O(k \log n)$.*

Доказательство теоремы является конструктивным. Опишем ключевые идеи алгоритма Риттера, поскольку они важны для дальнейшей дискуссии.

AVL-грамматиками называются ПП, деревья вывода которых являются AVL-деревьями. Основной операцией, используемой в алгоритме, является конкатенация AVL-грамматик. Следующая лемма из [8] оценивает сверху трудоемкость этой операции.

Лемма 3.1. *Пусть T, T' – деревъя 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* $S(T) \cdot S(T')$.

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

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

Основной цикл: Предположим, что для фиксированного $i > 1$ уже построена ПП T, выводящая текст $w_1 \cdot w_2 \cdots w_i$. По определению LZ-факторизации фактор w_{i+1} является подстрокой в тексте $w_1 \cdot w_2 \cdots w_i$. Фиксируем позиции вхождения фактора w_{i+1} в текст $w_1 \cdot w_2 \cdots w_i$ и, используя алгоритм взятия подграмматики, находим правила $T_1, T_2, \ldots, T_\ell$ такие, что $w_{i+1} = S(T_1 \cdot T_2 \cdots T_\ell)$. Поскольку T – AVL-грамматика, то $\ell = O(\log |S|)$. С помощью леммы конкатенируем правила $T_1, T_2, \ldots, T_\ell$ в некотором фиксированном порядке (подробнее см. [8]). Полагаем $T := T \cdot (T_1 \cdot T_2 \cdots T_\ell)$.

3.2 Модернизированный алгоритм Риттера

В работе [9] предлагается следующая оптимизация алгоритма 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 состоит в том, чтобы обрабатывать факторы максимально возможными группами и в рамках каждой группы факторов выбирать оптимальный порядок конкатенации. 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.

4 Декартово дерево как структура данных

4.1 Классическое декартово дерево

Определение. *Декартово дерево* — это двоичное дерево, в узлах которого хранятся ключ x и приоритет y. При этом дерево является двоичным деревом поиска по ключу x и двоичной кучей по приоритетам y. Напомним, что в двоичном дереве поиска на ключи накладывается ограничение, что ключи в узлах из левого поддерева меньше ключа в корне, а в узлах из правого поддерева – больше. В двоичной куче на приоритеты накладывается ограничение, что приоритет во всех потомках меньше приоритета в корне.

Например, декартово дерево, изображенное на рисунке 2, будем обозначать как $T_4 = (T_1, T_8, x_4, y_4)$. Из свойств кучи и двоичного дерева поиска вытекают неравенства $x_0 < x_1 < \ldots < x_9$, $y_0 < y_1$, $y_1 < y_4$, $y_2 < y_3$, $y_3 < y_1$, $y_5 < y_6$, $y_6 < y_8$, $y_7 < y_6$, $y_8 < y_4$, $y_9 < y_8$.

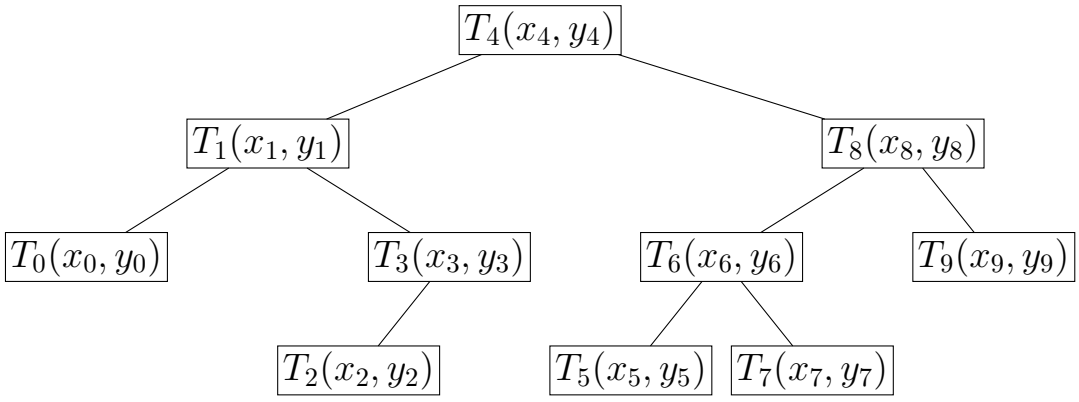

Рис. 2: Пример декартова дерева.

Для декартовых деревьев имеется вероятностная логарифмическая от числа узлов оценка высоты, см. [2, п. 4.1]. А именно, ожидаемая высота декартова дерева с n узлами, приоритеты которых выбраны случайно, независимо и имеют одинаковое распределение, есть $O(\log n)$. Более того, для любой константы $c > 1$ вероятность того, что высота декартова дерева с n узлами больше $2c \ln n$, ограничена величиной $n \left(\frac{n}{e}\right)^{-c \ln(c/e)}$.

Следовательно, декартово дерево балансируется само собой за счет случайно выбранных приоритетов y, в отличии от других сбалансированных

двоичных деревьев поиска, которые выполняют дополнительные действия, направленные на балансировку. Другим преимуществом декартовых деревьев является простота их реализации.

Помимо значения ключа и приоритета, ссылок на левое и правое поддеревья в каждом узле можно хранить дополнительную информацию различного характера. Например, можно хранить какую-то важную информацию, соответствующую ключу x, а также некую служебную информацию — например, количество узлов в поддереве. Заметим, что служебная информация меняется со временем, поэтому мы обязаны пересчитывать ее при каждом изменении узла на основе уже пересчитанной информаций в узлах детей.

Обозначения. В этой главе мы будем обозначать $size(T)$ — количество узлов в дереве T.

Существует две стандартные операции над декартовыми деревьями: *split* и *merge*. Они являются обратными друг к другу.

Операция *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.

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:

1. Если одно из деревьев пустое, то результатом будет другое дерево.
2. Иначе оба дерева непустые. Пусть $T_1 = (L_1, R_1, x_1, y_1)$, $T_2 = (L_2, R_2, x_2, y_2)$. Из свойств кучи получаем, что y_1 — минимальный приоритет y в дереве T_1, y_2 — в T_2. Поэтому корнем дерева-результата будет один из корней деревьев-операндов.
   (a) Если $y_1 < y_2$, то построим дерево $T_3 = merge(R_1, T_2)$. Тогда результатом будет дерево $T = (L_1, T_3, x_1, y_1)$.
   (b) Иначе $y_1 \geqslant y_2$, этот случай симметричен предыдущему, и мы оста-

вим его в качестве упражнения.

Операция *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.

Операция *split*

Вход: Декартово дерево T и x_{split} — позиция разреза.

Выход: Упорядоченная пара искомых декартовых деревьев (L, R).

Алгоритм:

1. Если дерево T пустое, то результатом будет пара пустых дерева.
2. Иначе предположим, что $T = (LT, RT, x, y)$.
   (a) Если $x_{split} < x$, то необходимо разрезать левое поддерево $(L', R') = split(LT, x_{split})$. Тогда результатом будет (L, R), где $L = L'$, $R = (R', RT, x, y)$.
   (b) Иначе $x_{split} \geqslant x$. Этот случай симметричен предыдущему, и мы оставим его в качестве упражнения.

Доказательство корректности приведенных алгоритмов можно прочитать [2].

Лемма 4.1 (об оценке сложности *merge*)**.** *Если h_1 и h_2 — высоты двух деревьев соответственно, то операция* merge *для этих двух деревьев работает за время $O(h_1 + h_2)$.*

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

Лемма 4.2 (об оценке сложности *split*)**.** *Если h — высота некоторого декартова дерева, то операция* split *для этого дерева работает за время $O(h)$ независимо от позиции разреза.*

Доказательство: Аналогично, заметим, что если дерево непустое, то мы делаем ровно один рекурсивный вызов от одного из поддеревьев, при этом высота уменьшается как минимум на 1. □

4.2 Рандомизированные двоичные деревья поиска

Рассмотрим следующую модификацию декартовых деревьев. Как уже говорилось приоритет y используется для автобалансировки высоты декартова дерева, если значениями этих приоритетов использовать случайные числа. Теперь вместо хранения приоритета y будем генерировать их заново по мере необходимости. То есть в алгоритме *merge* вместо приоритетов y, хранящихся в узлах дерева, сравниваются два новых случайных числа. Если правильно выбрать вероятностные распределения этих чисел, то сохранится логарифмическая оценка на высоту дерева. Получившаяся модификация декартова дерева называется *рандомизированным двоичным деревом поиска*. Более формально, двоичное дерево поиска T является *рандомизированным* тогда и только тогда, когда

- T – пустое дерево; или
- оба его поддерева L и R являются независимыми рандомизированными двоичными деревьями поиска, и $Pr\{size(L) = i\} = \frac{1}{n}$ для любого $i = 0, \ldots, n-1$, где $n = size(T)$.

Рассмотрим реализацию операций *split* и *merge* для такой модификации.

Операция *split*

Эта операция реализуется точно так же как и одноименная операция над декартовым деревом, так как она не использует информацию о приоритетах y.

Операция *merge*

Вход: Два рандомизированных двоичных дерева поиска T_1, T_2, причем ключи x из дерева T_1 меньше ключей x из дерева T_2.

Выход: Рандомизированное двоичное дерево поиска T, содержащее все узлы из деревьев T_1 и T_2.

Алгоритм:

1. Если одно из деревьев пустое, то результатом будет другое дерево.
2. Иначе оба дерева непустые. Пусть $T_1 = (L_1, R_1, x_1)$, $T_2 = (L_2, R_2, x_2)$. Пусть $k_1 = size(T_1)$, а $k_2 = size(T_2)$. Сгенерируем случайное число от

1 до $k_1 + k_2$ и обозначим его r.

(a) Если $r \leqslant k_1$, то построим дерево $R_{new} = merge(R_1, T_2)$. Тогда результатом будет дерево $T = (L_1, R_{new}, x_1)$.

(b) Иначе построим дерево $L_{new} = merge(T_1, L_2)$. Тогда результатом будет дерево $T = (L_{new}, R_2, x_2)$.

В статье [3] показано, что для рандомизированных двоичных деревьев поиска выполняется та же вероятностная логарифмическая от количества узлов оценка на высоту, а также доказана корректность описанных выше операций *split* и *merge*.

Лемма 4.3 (о сложности операций над рандомизированными двоичными деревьями поиска)**.** *Ассимптотика рассмотренных нами операций над рандомизированными двоичными деревьями поиска линейно зависит от высоты деревьев-операндов.*

Доказательство: Аналогично, доказательству соответствующего утверждения для декартовых деревьев. □

4.3 Декартово дерево по неявному ключу

Пусть мы имеем уже построенное декартово дерево. Заметим, что если мы потеряем информацию о ключах x в этом дереве, то, тем не менее, по его структуре мы сможем восстановить отношение линейного порядка определенного на ключах x. Действительно, при рекурсивном обходе двоичного дерева поиска в порядке *левое поддерево, корень, правое поддерево* мы обойдем узлы дерева в порядке возрастания ключа x и, тем самым, восстановим порядок ключей x. Таким образом, если мы не будем храним ключ x в дереве, мы сможем судить о сравнимости ключей x, соответствующим узлам по структуре дерева.

В каждом узле будем хранить дополнительную информацию — количество узлов в поддереве с корнем в этом узле. Тогда мы сможем восстановить порядковый номер ключа x в корневом узле среди ключей x из узлов данного поддерева (это количество узлов в левом поддереве плюс 1, если нумеровать с единицы).

Определение. Декартово дерево, в котором не хранится информация о ключах x, называется *декартовым деревом по неявному ключу*[1].

Отметим, что высота декартова дерева по неявному ключу также удовлетворяет вероятностной логарифмической оценке на высоту. Если восстановить ключи x любым способом, сохранив при этом требуемый линейный порядок, то мы получим классическое декартово дерево, а для него оценка на высоту выполняется.

Какие же преимущества перед классическим декартовым деревом имеет декартово дерево по неявному ключу? Вообще, на декартово дерево по неявному ключу выгодно смотреть как на структуру данных, реализующую функциональность массива. Действительно, пусть в каждом узле дерева хранится некоторая однотипная информация. Как мы знаем каждый узел дерева имеет некоторый номер в линейном порядке для неявному ключа x. Будем называть i-ым элементом этого «массива» информацию, хранящуюся в узле с номером i. Тогда задача обращения к элементу массива по индексу совпадает с задачей поиска порядковой статистики в двоичном дереве поиска и реализуется за $O(\log n)$.

Рассмотрим операции *split* и *merge* для декартова дерева по неявному ключу и выясним каким операциям они соответствуют в терминах массива.

Операция *merge*

Исследуем, как должен измениться алгоритм *merge* для слияния декартовых деревьев по неявному ключу. Для обычных декартовых деревьев (по явному ключу) у операции *merge* было ограничение о сравнимости ключей x между деревьями-операндами. Расширить линейный порядок на ключи обоих деревьев так, чтобы это ограничение выполнялось, можно единственным способом. Достаточно добавить в отношение сравнимость самого большого ключа x левого поддерева с самым маленьким ключом x правого поддерева и замкнуть по транзитивности получившееся отношение. Осталось заметить, что выполнение алгоритма никак не зависит от значений x и он

[1] К сожалению, изящная идея декартова дерева без явного хранения ключей, насколько известно автору, пока не освещалась в академической литературе. Достаточно полно эта идея представлена в интернет-публикации [1], согласно которой декартовы деревья по неявному ключу возникли в 2002 г. в олимпиадном программировании, а авторами идеи были Н. В. Дуров и А. С. Лопатин, в то время – члены студенческой сборной команды СПбГУ.

не требует никаких изменений.

Из того, как мы доопределили отношение линейного порядка для неявного ключа x, следует, что эта операция соответствует операции конкатенации массивов, то есть приписыванию второго массива в конец первого.

Операция *split*

Вход: Декартово дерево по неявному ключу T и x_{split} — позиция разреза.

Выход: Упорядоченная пара декартовых деревьев по неявному ключу $(L,\ R)$. Дерево L содержит первые x_{split} узлов в линейном порядке для x, а R – остальные.

Алгоритм:

1. Если дерево T пустое, то результатом будет два пустых дерева.
2. Иначе предположим, что $T = (LT,\ RT,\ y)$.
   (a) Если $x_{split} < size(LT) + 1$, то необходимо разрезать левое поддерево $(L',\ R') = split(LT,\ x_{split})$. Тогда результатом будет $(L,\ R)$, где $L = L',\ R = (R',\ RT,\ y)$.
   (b) Иначе $x_{split} \geqslant size(LT) + 1$. В этом случае необходимо разрезать правое поддерево $(L',\ R') = split(RT,\ x_{split} - size(LT) - 1)$. Тогда результатом будет $(L,\ R)$, где $L = (LT,\ L',\ y)$, $R = R'$.

Лемма 4.4 (о сложности операций над декартовыми деревьями по неявному ключу)**.** *Ассимптотика рассмотренных нами операций над декартовыми деревьями по неявному ключу линейно зависит от высоты деревьев-операндов.*

Доказательство: Аналогично, доказательству соответствующего утверждения для декартовых деревьев. □

В итоге, мы получаем структуру данных «массив» со следующими операциями:

- Обращение по ключу, который фактически играет роль индекса в массиве (с помощью операции поиска в двоичном дереве поиска)
- Конкатенация двух таких «массивов» (с помощью операции слияния двух деревьев $merge$)

- Разрезание «массива» на два других (с помощью операции разрезания дерева *split*)

Причем каждая из этих операций будут иметь сложность выполнения в среднем $O(\log n)$, если приоритеты y декартова дерева выбираются случайно и независимо с одинаковым вероятностным распределением. Заметим, что для обычных массивов операции конкатенации и разрезания выполняются за $O(n)$.

4.4 Персистентное декартово дерево

Структура данных называется *персистентной*, если в любой момент времени доступны все версии этой структуры данных. То есть любая операция над персистентной структурой данных не только возвращает новую версию структуры данных, но еще и оставляет неизменными операнды этой операции. Предложенный в работе [4] метод копирования узлов позволяет получить персистентную структуру из эфемерной структуры, основанной на ссылках. Суть метода состоит в том, что вместо изменения узла создается его модифицированная копия. Если на этот узел ссылались какие-то другие узлы, то они также должны быть скопированы, чтобы они ссылались на новую версию этого узла. Таким образом, мы почти без изменений алгоритмов из обычных декартовых деревьев получили персистентные декартовы деревья.

Лемма 4.5. *Выполнение одной операции* split *или* merge *увеличивает размер используемой памяти в среднем на* $O(\log n)$.

Доказательство: Этот результат напрямую следует из того, что при выполнении алгоритма слияния или разрезания будет рассмотрено в среднем $O(\log n)$ узлов дерева, каждый из которых скопирован не более одного раза. □

Замечание. Для устранения этого недостатка персистентных деревьев достаточно реализовать менеджер памяти, который считает количество входящих ребер в узел и освобождает память из-под узлов без входящих ссылок.

Хочется отметить, что персистентное рандомизированное двоичное дерево поиска по неяному ключу имеет особое преимущество: для него вполне осмыслена операция конкатенации с самим собой, с его поддеревом или с другой версией этого дерева. В действительности, это достигается за счет всех трех рассмотренных нами модификаций вместе взятых: **неявный ключ** (отказ от ключа x), **рандомизированный *merge*** (отказ от приоритета y) и **персистентность структуры данных**. Отказ от ключа x снимает ограничение на деревья-операнды (максимальный ключ x левого дерева меньше минимального ключа x правого дерева). Отказ от приоритета y предотвращает появление одинаковых приоритетов y в дереве при слиянии пересекающихся деревьев, что нарушило бы требование независимости этих случайных величин. И наконец, благодаря своему свойству персистентности оставлять неизменными старые версии операндов, мы используем ссылки на деревья-операнды только для чтения, поэтому нам не важно, как именно в памяти относительно друг друга находятся эти деревья.

Замечание. При совмещении трех модификаций декартовых деревьев в одной реализации у нас не возникнет трудностей. Как мы уже выяснили **персистентность** легко встраивается в реализацию дерева: достаточно просто создавать новые узлы, а не менять старые. Осталось понять почему **неявный ключ** и **рандомизированный *merge*** не «помешают» друг другу. В этой ситуации также все просто: достаточно вспомнить, что каждая из модификаций меняла реализацию только одной из двух основных операций *split* и *merge*.

5 Использование декартова дерева для построения прямолинейных программ

5.1 Рандомизированные ПП

5.1.1 Декартово дерево как ПП

Как мы уже знаем декартово дерево и дерево вывода ПП являются бинарными деревьями. В этом разделе мы модифицируем декартово дерево так, чтобы оно одновременно являлось деревом вывода некоторой 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 $Pr\{|S(L)| = i\} = \frac{1}{n-1}$ для любого $i = 1, \ldots, n-1$.

Лемма 5.1 (о количестве нетерминальных правил)**.** *Если ПП выводит строку длиной n, то в дереве вывода этой ПП количество узлов, соответствущих нетерминальным правилам, равно $n-1$.*

Доказательство: Пусть количество таких узлов равно k. Посчитаем количество ребер в дереве вывода. С одной стороны ребер $n+k-1$ (общеизвестный факт о деревьях), с другой стороны $2k$ (из каждого нетерминального узла ровно два ребра). Приравнивая полученные два числа, получаем $k = n-1$. $\square$

Лемма 5.2. *Если удалить из некоторой рандомизированной ПП узлы, соответствующие терминальным правилам, получится дерево являющееся рандомизированным двоичным деревом поиска по неявному ключу.*

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 меньших 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 $i = 1, \ldots, n-1$ $Pr\{|S(L)| = i\} = \frac{1}{n-1}$, где $n = |S(T)|$. По предположению индукции после удаления терминальных правил из L и R получаются рандомизированные двоичные деревья поиска по неявному ключу L' и R'. Используя лемму о количестве нетерминальных правил, нетрудно из имеющегося вероятностного тождества получить тождество из определения рандомизированных двоичных деревьев поиска $Pr\{size(L') = i\} = \frac{1}{n-1}$ для всех $i = 0, \ldots, n-2$.

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

Лемма 5.3 (о высоте рандомизированной ПП)**.** *Пусть T – рандомизированная ПП, тогда матожидание высоты дерева T равно $O(\log |S(T)|)$*

Доказательство: Из предыдущих лемм следует, что 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. □

5.1.2 Операции над рандомизированными ПП

Рассмотрим стандартные операции для декартовых деревьев: *merge* и *split*.

Операция *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 $(L', R') = split(RT, pos)$. Результатом будет пара (L, R), где $L = L'$, $R = (R', RT)$.

Теорема 5.1 (о корректности алгоритма разрезания)**.** *Алгоритм выполнения операции* split *корректен.*

Доказательство: При доказательстве будем использовать обозначения, принятые в алгоритме. Определимся с тем, что нам нужно доказать. Во-первых, для всех рекурсивных выховов 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 *split* 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 $i = 1, \ldots, pos - 1$.

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

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

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 5.4 (об оценке сложности алгоритма *split*)**.** *Если h — высота дерева, то операция* split *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 ассимптотика. □

Операция *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 $T = (L_{new},\ R_1)$.

Теорема 5.2 (о корректности алгоритма слияния)**.** *Алгоритм выполнения операции* merge *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 $i = 1, \ldots, n_1 + n_2 - 1$ выполняется $Pr\{|S(L)| = i\} = \dfrac{1}{n_1 + n_2 - 1}$.

$$Pr\{|S(L)| = i\} = Pr\{|S(L)| = i \mid 0 < r < n_1\} \cdot Pr\{0 < r < n_1\}$$

$$+Pr\{|S(L)| = i \mid r = n_1\} \cdot Pr\{r = n_1\}$$

$$+Pr\{|S(L)| = i \mid n_1 < r < n_1 + n_2\} \cdot Pr\{n_1 < r < n_1 + n_2\}$$

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 $i = 1, \ldots n_1{-}1$ $Pr\{|S(L)| = i\} = Pr\{|S(L_1)| = i\} \cdot Pr\{0 < r < n_1\}$ $\frac{1}{n_1 - 1} \cdot \frac{n_1 - 1}{n_1 + n_2 - 1} = \frac{1}{n_1 + n_2 - 1}$.

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

Для любого $i = n_1{+}1, \ldots, n1{+}n2{-}1$ $Pr\{|S(L)| = i\} = Pr\{|S(L_{new})| = i\} \cdot Pr\{n_1 < r < n_1 + n_2\} = Pr\{|S(L_2)| = i - n_1\} \cdot Pr\{n_1 < r < n_1 + n_2\} = \frac{1}{n_2 - 1} \cdot \frac{n_2 - 1}{n_1 + n_2 - 1} = \frac{1}{n_1 + n_2 - 1}$.

Следовательно, T является рандомизированной ПП с нужным выводом.

Шаг индукции выполняется, значит, описанный алгоритм корректен. □

Лемма 5.5 (об оценке сложности алгоритма слияния)**.** *Если h_1 и h_2 — высоты двух деревьев соответственно, то операция* merge *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 ассимптотика. □

5.2 Алгоритм построения прямолинейных программ

Известно, что задача построения грамматики минимального размера для заданного текста является NP-трудной [10]. Поэтому представляют интерес приближенные методы решения этой задачи. Риттер в работе [8] предложил алгоритм построения ПП, выводящей заданный текст, по известной LZ-факторизации этого текста.

Алгоритм построения рандомизированной ПП

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

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

Алгоритм:

1. Положим в T_1 дерево, выводящее слово из буквы w_1.
2. Для всех $i \in 2..k$ вычислим
   (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 *split* 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.

Замечание. Так как мы применяем операцию *merge* к T_{i-1} и F_i (которое частично пересекается с T_{i-1}), то, естественно, для корректной работы алгоритма реализация рандомизированных ПП должна быть *персистентной*.

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

Теорема 5.3. *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 большех двух операций *split* и не более одной операции *merge*. Каждая из этих операций создает в срденем 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 $O(k \log n)$. □

6 Экспериментальные результаты

6.1 Условия экспериментов.

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

- строки Фибоначчи;
- 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. Строки Фибоначчи – это в определенном смысле наилучшие входные данные для задачи построения ПП, и эксперименты над ними позволяют оценить потенциал ПП как модели сжатого представления «сверху». Напротив, случайные строки сжимаются плохо и, следовательно, являются наихудшим входом для задачи построения ПП. Поэтому эксперименты со случайными строками оценивают возможности ПП «снизу». 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.

6.2 Результаты экспериментов.

Как и ожидалось, все алгоритмы построения ПП работают бесконечно быстро на строках Фибоначчи и строят очень компактные ПП. Например, 35-е слово Фибоначчи длиной порядка 40МБ обрабатывается в течение 1мс, а на выходе получается ПП, имеющая около 100 правил.

Основные результаты наших экспериментов для случайных строк и последовательностей ДНК представлены графиками на рис. 3–5. 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 из [8];
- ● – модифицированный алгоритм Риттера;
- ▲ – алгоритм построения рандомизированной ПП.

По оси абсцисс на всех графиках откладывается длина сжимаемого текста.

Основными параметрами алгоритмов, которыми мы интересовались, были время работы и степень сжатия. Степень сжатия текста определялась как отношение размера сжатого представления текста к длине текста, выраженное в процентах.

Для сравнения скорости алгоритмов были проведены тесты двух типов. В тестах первого типа алгоритмы хранили строящееся дерево в оперативной памяти, а в тестах второго типа – во внешнем файле (т. е. каждое обращение к дереву было обращением к файловой системе). На рис. 3 и 4 представлены результаты тестов обоих типов соответственно на последовательностях ДНК и на случайных строках. Видно, что при хранении ПП в оперативной памяти время работы модифицированного алгоритма в среднем в несколько раз меньше чем время работы исходного алгоритма Риттера (в два раза меньше на произвольных строках, в три раза меньше на последовательностях ДНК). Если же дерево хранится в файловой системе, то модифицированный алгоритм работает в среднем в пять раз быстрее на последовательностях ДНК и в три раза быстрее на произвольных строках. Алгоритм, строящий рандомизированные ПП, превосходит по быстродействию исходный алгоритм Риттера, однако немного уступает его модифицированной версии. Причина здесь, по-видимому, связана с тем, что высота декартова дерева, возвращаемого алгоритмом, существенно больше высоты соответствующих AVL-деревьев. (Средняя высота AVL-дерева в наших экспериментах равна 21.8, а средняя высота декартова дерева составляет 47.8.) Из-за большей высоты дерева алгоритму построения рандомизиро-

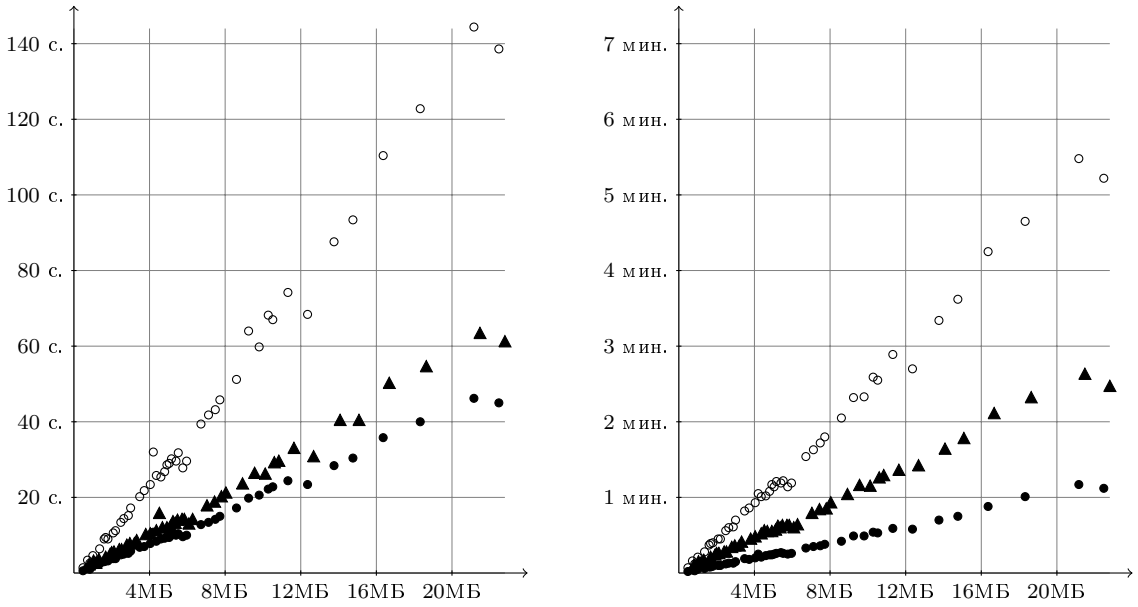

Рис. 3: Время работы алгоритмов построения ПП на последовательностях ДНК при хранении ПП в памяти (слева) и в файле (справа)

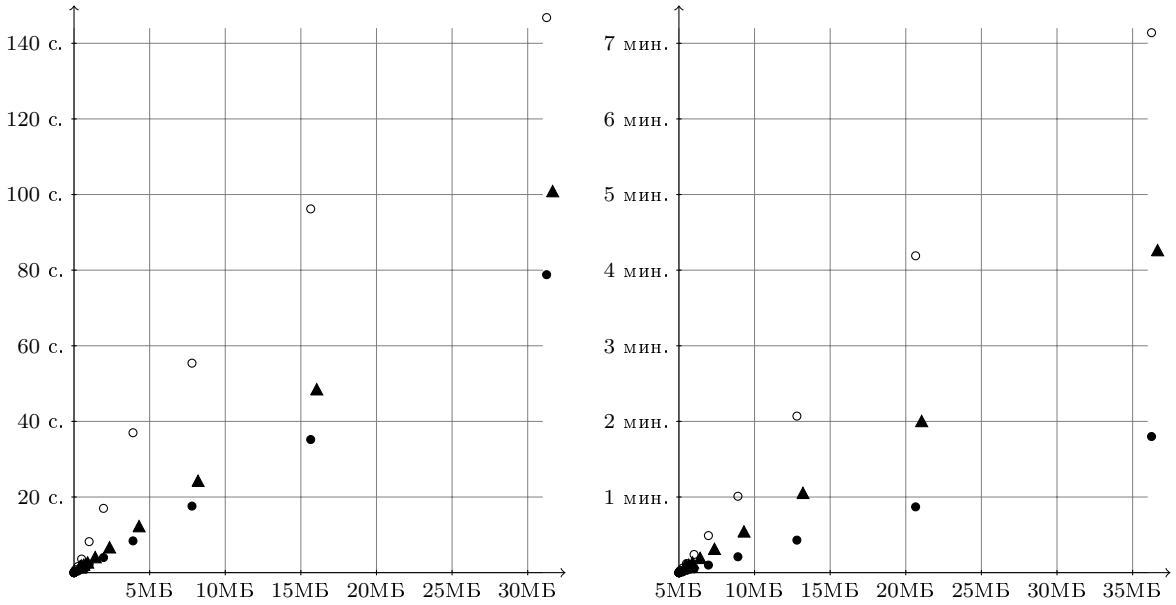

Рис. 4: Время работы алгоритмов построения ПП на случайных строках при хранении ПП в памяти (слева) и в файле (справа)

ванных ПП приходится обрабатывать намного больше правил, что сводит на нет весь выигрыш, который возникает за счет простоты поддержания баланса в декартовых деревьях.

На рис. 5 степень сжатия, достигаемая алгоритмами построения ПП, сравнивается со степенью сжатия алгоритма Лемпеля-Зива с бесконечным окном. Интересно, что отношение степени сжатия алгоритмов, строящих ПП, к степени сжатия алгоритма Лемпеля-Зива по существу не зависит ни от типа сжимаемого текста, ни от его длины. Видно, что алгоритмы, строящие AVL-деревья, обеспечивают практически одинаковую степень сжатия, которая хуже степени сжатия алгоритма Лемпеля-Зива примерно в два раза. У алгоритма, строящего рандомизированные ПП, степень сжатия заметно хуже, чем у алгоритмов, строящих AVL-деревья.

7 Выводы и дальнейшие перспективы

Эксперименты демонстрируют, что алгоритм, представленный в работе, обыгрывает по скорости работы исходный алгоритм Риттера и чуть уступает модифицированной версии алгоритма Риттера. Применив в алгоритме построения рандомизированных ПП, эвристику по оптимизации порядка конкатенаций, возможно, мы получим алгоритм, работающий быстрее всех трех рассмотренных. Но, к сожалению, коэффициент сжатия с помощью рандомизированных ПП почти в два раза хуже коэффициента сжатия с помощью алгоритма Риттера. Последнее обстоятельство существенно с точки зрения влияния на быстродействия алгоритмов, работающих непосредственно со сжатыми представлениями данных. Таким образом, нам удалось алгоритмически ускорить построение ПП с помощью «прогрессивной» структуры данных, но размер получающихся ПП пока не впечатляет. И более того, автор пока не знает за счет чего размер рандомизированных ПП станет меньше.

Все проанализированные алгоритмы сжатия на основе ПП уступают алгоритмам из семейства Лемпеля-Зива по степени сжатия и по времени сжатия. Теоретическое преимущество алгоритмов на основе ПП состоит в том,

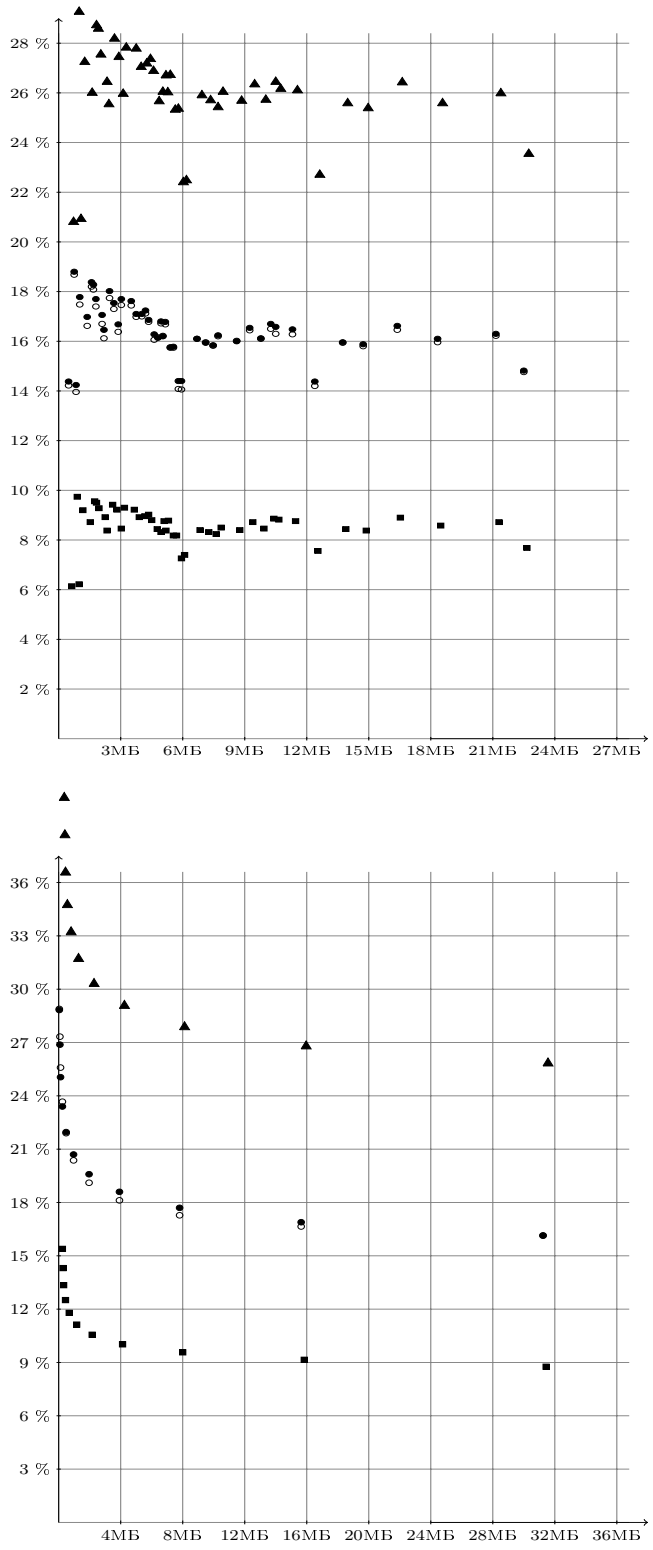

Рис. 5: Степень сжатия на последовательностях ДНК (сверху) и на случайных строках (снизу)

что они обеспечивают хорошо структурированное сжатое представление, для которого имеются алгоритмы, позволяющие решать некоторые важные задачи (типа поиска подстроки) без разархивирования. Однако вопрос о том, на каких объемах входных данных алгоритмы, работающие с ПП, смогут обогнать классические строковые алгоритмы, остается открытым и, на наш взгляд, является одним из основных направлений для дальнейших исследований в рассматриваемой области.

Список литературы

[1] *А. Полозов*, Декартово дерево: Часть 3. Декартово дерево по неявному ключу, Электронный ресурс, `http://habrahabr.ru/blogs/algorithm/102364/`.

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

[3] *C. Martinez, S. Roura*, Randomized binary search trees, Journal of the ACM (1998), 288–323.

[4] *J. Driscoll, N. Sarnak, D. Sleator, R. Tarjan*, Making data structures persistent, Journal of Computer and System Sciences, 1989, 86–124.

[5] *A. Apostolico, G. M. Landau, S. Skiena*, Matching for Run-Length Encoded Strings, J. Complexity, 15 (1999), 4–16.

[6] *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.

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

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

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

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

[12] *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.

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

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

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

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