УДК 519.256

Эффективное сжатие данных с помощью прямолинейных программ*

И. С. Бурмистров А. В. Козлова Е. Б. Курпилянский
А. А. Хворост

Аннотация

Изучаются два алгоритма построения контекстно свободных грамматик, выводящих заданный текст. Первый алгоритм является модификацией известного алгоритма Риттера и строит грамматику на основе AVL-деревьев, второй алгоритм использует декартовы деревья. Описываются результаты экспериментов по сравнению эффективности этих двух алгоритмов и алгоритма Риттера на различных наборах данных и по сравнению алгоритмы построения грамматик с алгоритмами из семейства алгоритмов Лемпеля-Зива по степени сжатия.

§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 [11], прямолинейные программы (кратко ПП) [9], групповое кодирование [3] 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, LZ78 [15], LZW [13], LZ77 [14]). 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. [9].

*Работа выполнена при поддержке гранта РФФИ № 10-01-00793.

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** [7], **Наибольшая общая подстрока** [8], считающая версия задачи **Поиск всех палиндромов** [8], некоторые версии задачи **Наибольшая общая подпоследовательность** [12]. В то же время константы, которые скрываются за «O 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. Данная работа задумана как шаг именно в этом направлении. В ней рассматриваются два вопроса: насколько трудоемким является процесс построения ПП по данному тексту и насколько высокий уровень сжатия может обеспечить ПП по сравнению с классическими алгоритмами.

Структура работы такова. В §2 вводятся основные определения о строках и ПП. В §3 представлены два алгоритма построения ПП. Первый алгоритм является модифицированной версией алгоритма Риттера [9]. Второй алгоритм основан на новой структуре данных и строит ПП, являющиеся декартовыми деревьями. В §4 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 §5.

Часть результатов данной работы, относящаяся к модификации алгоритма Риттера, была представлена на конференции по сжатию, передаче и обработке данных, проходившей в Палинуро, Италия, в июне 2011 г. (CCP 2011, см. `http://ccp2011.dia.unisa.it/CCP_2011/Home.html`), и анонсирована в [4].

§2. Обозначения

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 $S \cdot S'$. *Позицией* в строке S называется место между соседними символами. Мы нумеруем позиции произвольной строки S слева направо, начиная с 1 и заканчивая $|S| - 1$. Удобно ввести также позицию 0, которая предшествует строке S, и позицию $|S|$, которая следует за S. Для строки S и числа i такого, что $0 \le i < |S|$, обозначим через $S[i]$ символ, расположенный между позициями i и $i+1$. Например, $S[0]$ – это первый символ строки S. Обозначим через $S[\ell \dots r]$, где $0 \le \ell < r \le |S|$, подстроку строки S, которая начинается с позиции ℓ и заканчивается в позиции r. Таким образом, $S[\ell \dots r] = S[\ell] \cdot S[\ell + 1] \cdot \ldots \cdot S[r - 1]$.

Прямолинейная программа (ПП) $\mathbb{S}$ – это последовательность присваи-

ваний следующего вида:

$$\mathbb{S}_1 \to expr_1,\ \mathbb{S}_2 \to expr_2,\ \ldots,\ \mathbb{S}_n \to expr_n,$$

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

- $expr_i$ есть символ из Σ (тогда правило $\mathbb{S}_i$ называется *терминальным*),
- $expr_i$ есть $\mathbb{S}_\ell \cdot \mathbb{S}_r$, где $\ell, r < i$ (тогда правило $\mathbb{S}_i$ называется *нетерминальным*).

Таким образом, ПП – это контекстно свободная грамматика в нормальной форме Хомского, порождающая в точности одну строку над алфавитом Σ. Строку S, выводимую из ПП $\mathbb{S}$, будем называть *текстом*. Для ПП $\mathbb{S}$, выводящей текст S, определим *дерево вывода* текста S как дерево вывода грамматики $\mathbb{S}$. В дереве вывода мы отождествляем терминальные узлы с их родителями, поэтому оно является двоичным. На рис. 1 представлено дерево вывода ПП

$$\mathbb{F}_0 \to b,\ \mathbb{F}_1 \to a,\ \mathbb{F}_2 \to \mathbb{F}_1 \cdot \mathbb{F}_0,\ \mathbb{F}_3 \to \mathbb{F}_2 \cdot \mathbb{F}_1,\ \mathbb{F}_4 \to \mathbb{F}_3 \cdot \mathbb{F}_2,\ \ \mathbb{F}_5 \to \mathbb{F}_4 \cdot \mathbb{F}_3,\ \ \mathbb{F}_6 \to \mathbb{F}_5 \cdot \mathbb{F}_4,$$

выводящей 6-е слово Фибоначчи *abaababaabaab*. В этом примере слово дли-

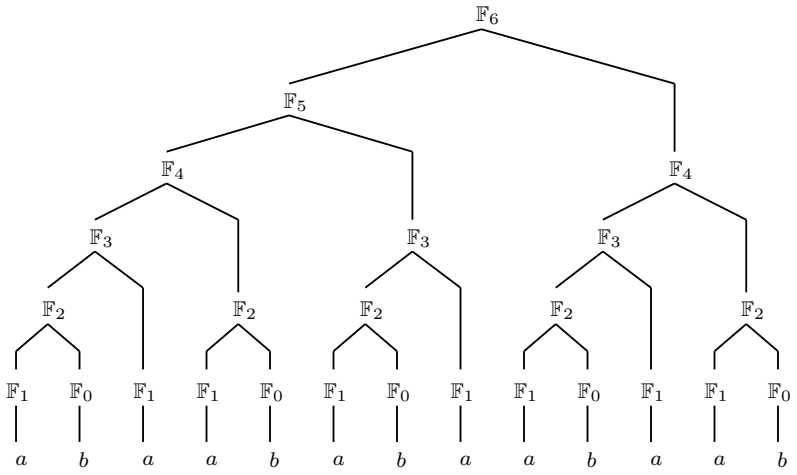


Рис. 1: Прямолинейная программа, выводящая слово *abaababaabaab*

ны 13 порождается ПП из 7 правил. Аналогично, в общем случае n-е слово Фибоначчи может быть порождено ПП

$$\mathbb{F}_0 \to b,\ \mathbb{F}_1 \to a,\ \mathbb{F}_2 \to \mathbb{F}_1 \cdot \mathbb{F}_0,\ \mathbb{F}_3 \to \mathbb{F}_2 \cdot \mathbb{F}_1,\ \ldots,\ \mathbb{F}_n \to \mathbb{F}_{n-1} \cdot \mathbb{F}_{n-2}$$

из $n+1$ правил. Поскольку длина n-го слова Фибоначчи есть $(n+1)$-е число Фибоначчи, т. е. ближайшее целое к $\frac{\varphi^{n+1}}{\sqrt{5}}$, где $\varphi = \frac{1+\sqrt{5}}{2}$ (золотое сечение), мы видим, что для некоторых текстов их сжатое представление с помощью ПП может обеспечить экспоненциальный выигрыш в пространстве.

В работе приняты следующие договоренности: все ПП обозначаются с помощью заглавных ажурных букв, например, $\mathbb{S}$. Каждое правило ПП $\mathbb{S}$ (и каждый внутренний узел ее дерева вывода) обозначается аналогичной буквой с порядковым номером, например, $\mathbb{S}_i$. *Размер* ПП $\mathbb{S}$ полагается равным числу правил и обозначается через $|\mathbb{S}|$. *Высота* узла двоичного дерева определяется рекурсивно: высота листа полагается равной 0, а высота внутреннего узла полагается равной 1 + максимум из высот его сыновей. Обозначим через $h(\mathbb{S}_i)$ высоту правила $\mathbb{S}_i$.

Конкатенацией ПП $\mathbb{S}$ и $\mathbb{S}'$, выводящих тексты S и S' соответственно, называется ПП $\mathbb{S} \cdot \mathbb{S}'$, которая выводит текст $S \cdot S'$. Подчеркнем, что конкатенация ПП не есть однозначно определенная операция (в отличие от конкатенации строк): сконструировать из $\mathbb{S}$ и $\mathbb{S}'$ ПП, выводящую $S \cdot S'$, можно разными способами, и выбор конкретного способа обычно определяется спецификой решаемой задачи.

§3. Алгоритмы построения прямолинейных программ

3.1. Прямолинейные программы, факторизации и деревья. Классическая формулировка задачи построения ПП такова:

Задача: **Построение ПП по данному тексту**
Вход: текст S;
Выход: ПП $\mathbb{S}$, которая выводит S.

Известно, что задача построения грамматики минимального размера для заданного текста является NP-трудной [5]. Поэтому представляют интерес эффективные приближенные алгоритмы. Один из подходов к быстрому построению ПП отталкивается от факторизации текста. Если зафиксирована такая факторизация, то на каждом шаге алгоритм строит ПП, которая выводит очередной фактор, а затем конкатенирует ПП, построенную на предыдущих шагах, с ПП, выводящей текущий фактор. При таком подходе размер результирующей ПП зависит не только от размера исходного текста, но и от способа факторизации. Следовательно, задача построения ПП может быть уточнена таким образом:

Задача: **Построение ПП по данной факторизации текста**
Вход: текст S и его факторизация $F_1, F_2, \ldots, F_k$;
Выход: ПП $\mathbb{S}$, которая выводит S.

Риттер [9] в качестве исходной факторизации выбрал факторизацию, порождаемую алгоритмом LZ77. Это позволило установить связь между размером ПП, выводящей данный текст S, и размером LZ77-словаря для S. Кроме того, благодаря свойствам LZ-факторизаций, ПП для текущего фактора всегда является частью уже построенной на предыдущих шагах ПП, что радикально ускоряет вычисления.

Определение 3.1. *LZ-факторизация* текста S – это последовательность строк $F_1, F_2, \ldots, F_k$ такая, что $S = F_1 \cdot F_2 \cdot \ldots \cdot F_k$, где $F_1 = S[0]$ и F_i – наибольший непустой префикс подстроки $S[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |S|]$, который входит в качестве подстроки в $F_1 \cdot \ldots \cdot F_{i-1}$, а если указанный префикс пуст, то $F_i = S[|F_1 \cdot \ldots \cdot F_{i-1}|]$. Число k называется *размером факторизации*.

Формулировка задачи построения ПП не накладывает строгих ограничений на дерево вывода ПП, кроме того, что оно является двоичным и *максимальным*. Последнее свойство означает, что любой внутренний узел имеет ровно двух сыновей (термин подсказан теорией кодирования – понятно, что двоичный префиксный код будет максимальным по включению тогда и только тогда, когда его дерево максимально в указанном смысле). Существуют различные виды двоичных деревьев. Какой из них больше подходит для решения рассматриваемой задачи? В алгоритме построения ПП из [9] используются сбалансированные деревья, а именно, AVL-деревья.

Определение 3.2. *AVL-дерево* – это двоичное дерево, у каждого внутреннего узла которого высоты сыновей отличаются не более чем на 1.

Для AVL-деревьев имеется логарифмическая от числа узлов оценка высоты, см. [1, пп. 6.2.3]. Именно поэтому данный вид деревьев используется в алгоритме Риттера. Однако алгоритм построения AVL-деревьев нетривиален и требователен к ресурсам. В качестве альтернативы в п. 3.4 мы рассматриваем алгоритм построения ПП на основе декартовых деревьев. Напомним соответствующие определения.

Определение 3.3. *Двоичное дерево поиска* – это двоичное дерево, в каждом узле которого хранится число, называемое *ключом*. При этом для произвольного внутреннего узла X ключи всех узлов левого поддерева меньше ключа X, а ключи всех узлов правого поддерева больше ключа X.

Куча – это двоичное дерево, в каждом узле которого хранится число, называемое *приоритетом*. При этом для произвольного внутреннего узла X приоритеты его сыновей меньше приоритета X.

Декартово дерево – это двоичное дерево, в каждом узле которого хранится пара чисел: *ключ* и *приоритет*. При этом оно является двоичным деревом поиска по ключам и кучей по приоритетам.

Имеется вероятностная логарифмическая от числа узлов оценка высоты декартова дерева ([10], см. п. 3.4 ниже). При этом алгоритм построения декартова дерева тратит существенно меньше времени на поддержание баланса узлов. Одной из основных целей данной работы является сравнительный анализ алгоритмов построения ПП, основанных на AVL-деревьях и на декартовых деревьях.

3.2. Узкое место алгоритма Риттера. Риттер [9] доказал следующий результат.

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 [9] оценивает сверху трудоемкость этой операции.

Лемма 3.2. *Пусть* $\mathbb{S}$, $\mathbb{S}'$ – *AVL-грамматики. Тогда, добавив не более чем* $O(|h(\mathbb{S}) - h(\mathbb{S}')|)$ *новых правил, за время* $O(|h(\mathbb{S}) - h(\mathbb{S}'|)$ *можно построить AVL-грамматику* $\mathbb{S} \cdot \mathbb{S}'$, *которая выводит текст* $S \cdot S'$.

Алгоритм Риттера получает на вход LZ-факторизацию F_1, F_2, ..., F_k данного текста S и индуктивно строит ПП $\mathbb{S}$, выводящую текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ для $i = 1, 2, \ldots, k$.

Инициализация: Полагаем $\mathbb{S}$ равной грамматике, состоящей из терминального правила, выводящего $F_1 = S[0]$.

Основной цикл: Предположим, что для фиксированного $i > 1$ уже построена ПП $\mathbb{S}$, выводящая текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. По определению LZ-факторизации фактор F_{i+1} является подстрокой в тексте $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Фиксируем позиции вхождения фактора F_{i+1} в текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ и, используя алгоритм взятия подграмматики, находим правила $\mathbb{S}_1, \mathbb{S}_2, \ldots, \mathbb{S}_\ell$ такие, что $F_{i+1} = S_1 \cdot S_2 \cdot \ldots \cdot S_\ell$. Поскольку $\mathbb{S}$ – AVL-грамматика, то $\ell = O(\log |S|)$. С помощью леммы 3.2 конкатенируем правила $\mathbb{S}_1, \mathbb{S}_2, \ldots, \mathbb{S}_\ell$ в некотором фиксированном порядке (подробнее см. [9]). Полагаем $\mathbb{S} := \mathbb{S} \cdot (\mathbb{S}_1 \cdot \mathbb{S}_2 \cdot \ldots \cdot \mathbb{S}_\ell)$.

При работе с AVL-деревьями самой трудной задачей является поддержание баланса. Если после добавления нового правила в дерево оно перестает быть сбалансированным, то полученное дерево приходится перебалансировать с помощью локального преобразования, называемого *вращением*. Существует два типа вращений, они схематически показаны на рис. 2. Каждое вращение может порождать до трех новых узлов (отмечены штрихами на рис. 2), а также сделать до трех узлов висячими (неиспользуемыми).

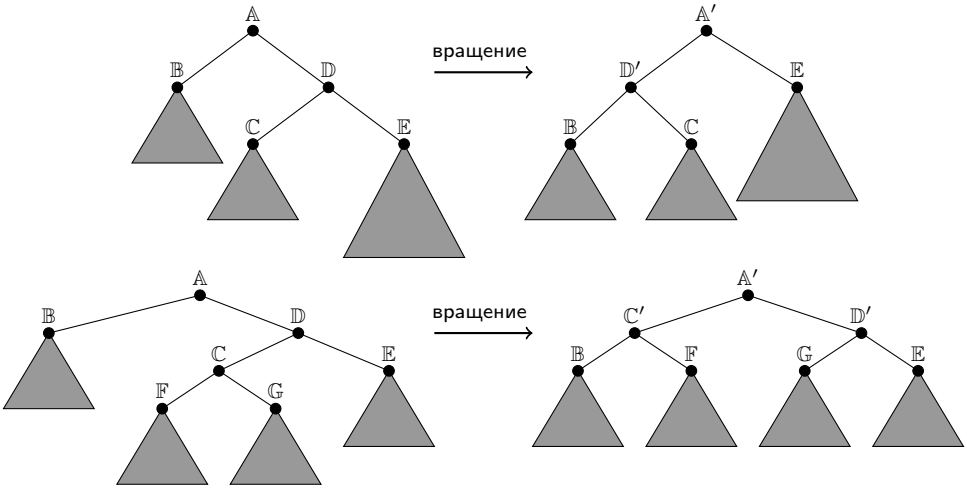


Рис. 2: Вращения после вставки узла в AVL-дерево

Как видно из леммы 3.2, 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 $\mathbb{S}$ становится большой, а при каждом последующем выполнении основного цикла с $\mathbb{S}$ конкатенируется AVL-грамматика относительно небольшой высоты. Следующий простой пример демонстрирует, что число вращений при исполнении алгоритма Риттера действительно может оказаться значительным.

Пример 1. Пусть $S = a^{2^n}bc^{2^n}$, где n – фиксированное натуральное число. LZ-факторизация строки S имеет вид:

$$S = a \cdot a \cdot a^2 \cdot a^4 \cdot \ldots \cdot a^{2^{n-1}-1} \cdot b \cdot c \cdot c \cdot c^2 \cdot c^4 \cdot \ldots \cdot c^{2^{n-1}-1}.$$

Обозначим факторы (в том порядке, в котором они следуют в этой факторизации) через $F_1, F_2, \ldots, F_{2n+3}$, а соответствующие им ПП (деревья которых, как легко понять, будут полными двоичными деревьями, в частности, AVL-деревьями) – через $\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_{2n+3}$.

Оценим число вращений, которое потребуется при последовательной конкатенации AVL-грамматик $(\ldots((\mathbb{F}_1 \cdot \mathbb{F}_2) \cdot \mathbb{F}_3)\ldots) \cdot \mathbb{F}_{2n+3}$. При последовательная конкатенации грамматик $\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_{n+1}$ вращения не понадобятся, поскольку на каждом шаге будут конкатенироваться полные двоичные деревья одинаковой высоты. Дерево грамматики $\mathbb{F}_1 \cdot \mathbb{F}_2 \cdot \ldots \cdot \mathbb{F}_{n+1}$ будет полным двоичным деревом высоты n, а конкатенация $(\mathbb{F}_1 \cdot \mathbb{F}_2 \cdot \ldots \cdot \mathbb{F}_{n+1}) \cdot \mathbb{F}_{n+1}$ даст AVL-дерево высоты $n+1$. Нетрудно проверить, что каждая из последующих конкатенаций будет нарушать баланс и приводить как минимум к одному вращению. Всего произойдет от $n+1$ до $\Theta(n^2)$ вращений (верхняя оценка вытекает из оценки для числа новых правил, содержащейся в лемме 3.2).

Заметим, что если бы алгоритм мог выбрать «правильный» порядок конкатенации, а именно

$$((\ldots((\mathbb{F}_1 \cdot \mathbb{F}_2) \cdot \mathbb{F}_3)\ldots) \cdot \mathbb{F}_{n+1}) \cdot ((\ldots((\mathbb{F}_{n+2} \cdot \mathbb{F}_{n+3}) \cdot \mathbb{F}_{n+4})\ldots) \cdot \mathbb{F}_{2n+3}),$$

то не понадобилось бы ни одного вращения. Итак, одно из направлений оптимизации алгоритма Риттера состоит в определении «удачного» порядка конкатенации. Другое направление состоит в минимизации числа запросов к грамматике. Оптимизация числа запросов к грамматике становится важной в случае, когда размер входного текста очень велик и мы не можем хранить текущее состояние ПП в оперативной памяти. Тогда временная стоимость операции доступа к ПП превышает стоимость вычислений в оперативной памяти. Следующий пример показывает, что число запросов к ПП можно уменьшить за счет того, что несколько факторов могут быть обработаны вместе.

Пример 2. Пусть $S = ba^{2^{n-1}}ba^{2^{n-2}} \cdots ba$, где n – фиксированное натуральное число. LZ-факторизация строки S имеет вид:

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

Пусть $\mathbb{S}_1$ – ПП, выводящая текст $S_1 = ba^{2^{n-1}}$. Очевидно, что все факторы указанной LZ-факторизации, начиная с $ba^{2^{n-2}}$, входят в строку S_1.

Следовательно, алгоритм мог бы за одно обращение к ПП $\mathbb{S}_1$ вычленить из нее ПП $\mathbb{S}_2$, $\mathbb{S}_3$, …, $\mathbb{S}_{n-1}$, выводящие тексты $ba^{2^{n-3}}$, $ba^{2^{n-4}}$, …, ba соответственно. После этого алгоритм мог бы конкатенировать полученные ПП в следующем порядке: $\mathbb{S}_1 \cdot (\dots (\mathbb{S}_{n-3} \cdot (\mathbb{S}_{n-2} \cdot \mathbb{S}_{n-1})) \dots)$.

3.3. Оптимизация алгоритма Риттера 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.

Модифицированный алгоритм Риттера получает на вход LZ-факторизацию F_1, F_2, …, F_k данного текста S и строит ПП $\mathbb{S}$, выводящую S.

Инициализация: Полагаем $\mathbb{S}$ равной грамматике, состоящей из терминального правила, выводящего $F_1 = S[0]$.

Основной цикл: Пусть $\mathbb{S}$ – ПП, которая выводит текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$, где $0 < i < k$. Пусть $\ell \in \{1, \dots, k-i\}$ – наибольшее число такое, что каждый из факторов F_{i+1}, …, $F_{i+\ell}$ входит в $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. При фиксированной LZ-факторизации значение ℓ может быть найдено с помощью линейного поиска по факторам. ПП $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \dots, \mathbb{F}_{i+\ell}$, которые выводят тексты $F_{i+1}, F_{i+2}, \dots, F_{i+\ell}$ соответственно, вычисляются с помощью алгоритма взятия подграмматики (аналогично алгоритму из [9]).

Затем алгоритм вычисляет конкатенацию ПП $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \dots, \mathbb{F}_{i+\ell}$, при этом он пытается оптимизировать порядок, в котором осуществляется конкатенация, с помощью динамического программирования. Введем функцию $\varphi(p, q)$, значения которой при $1 \le p, q \le \ell$ вычисляются по формуле:

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

Значение $\varphi(p, q)$ пропорционально верхней оценке для минимального числа вращений, необходимых для конкатенации ПП $\mathbb{F}_{i+p}, \mathbb{F}_{i+p+1}, \dots, \mathbb{F}_{i+q}$, которую можно извлечь из леммы 3.2 и из оценки для высоты AVL-дерева [1, теорема 6.2.3A]. Конечно, эта верхняя оценка, как правило, оказывается сильно завышенной, так что в действительности $\varphi(p, q)$ следует рассматривать как эвристику, с помощью которой выбирается «хорошая» группировка факторов при конкатенации.

Заполним $\ell \times \ell$-таблицу значениями функции $\varphi(p, q)$, $1 \le p, q \le \ell$; при этом для случая, когда $p < q$, вместе с $\varphi(p, q)$ будем хранить то значение $r \in \{p, p+1, \dots, q-1\}$, при котором достигается минимум выражения

$$\varphi(p,r) + \varphi(r+1,q) + \big|\log(|F_{i+p}| + \ldots + |F_{i+r}|) - \log(|F_{i+r+1}| + \ldots + |F_{i+q}|)\big|.$$

Порядок заполнения таков: заполняем (нулями) все ячейки (p, q) с $p \ge q$, затем заполняем ячейки с $q - p = 1$, затем – ячейки с $q - p = 2$ и т. д. Таким

образом, к моменту подсчета очередного значения $\varphi(p,q)$ все необходимые для этого значения $\varphi(p,r)$ и $\varphi(r+1,q)$ уже содержатся в таблице. Любое значение $\varphi(p,q)$ может быть вычислено за время $O(\ell)$ (псевдокод процедуры представлен на рис. 3). Поэтому таблица будет заполнена за время $O(\ell^3)$ с использованием $O(\ell^2)$ памяти.

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

Рис. 3: Псевдокод функции, вычисляющей значение $\varphi(p,q)$

Теперь, отправляясь от значения параметра r, записанного в ячейке $(1,\ell)$, можно за время $O(\ell)$ восстановить тот порядок конкатенации ПП $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$, которому соответствует значение $\varphi(1,\ell)$. Используя этот порядок, строим ПП $\mathbb{F}$, которая выводит текст $F_{i+1} \cdot F_{i+2} \cdot \ldots \cdot F_{i+\ell}$, и полагаем $\mathbb{S} := \mathbb{S} \cdot \mathbb{F}$.

Теорема 3.3. *Размер ПП, которую модифицированный алгоритм Риттера строит по данной факторизации $F_1, F_2, \ldots, F_k$ текста длины n, равен $O(k \log n)$.*

Доказательство по существу повторяет соответствующий фрагмент доказательства теоремы 3.1, но мы воспроизведем его для полноты изложения.

Проведем индукцию по числу факторов. База индукции очевидна.

Пусть уже построена ПП $\mathbb{S}$, которая выводит текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$, где $0 < i < k$, и имеет размер $O(i \log |F_1 \cdot F_2 \cdot \ldots \cdot F_i|) = O(i \log n)$. Далее, пусть $F_{i+1}, \ldots, F_{i+\ell}$ – все те последующие факторы, которые входят в $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Рассмотрим подграмматики $\mathbb{F}_{i+1}, \mathbb{F}_{i+2}, \ldots, \mathbb{F}_{i+\ell}$ грамматики $\mathbb{S}$, которые выводят тексты $F_{i+1}, F_{i+2}, \ldots, F_{i+\ell}$ соответственно. Высота ПП $\mathbb{F}_{i+j}$ не превосходит $1.4404 \log |F_{i+j}|$ [1, теорема 6.2.3A], и потому по лемме 3.2 число новых правил, которые придется добавить на каждом этапе вычисления ПП $\mathbb{F}$, которая выводит текст $F_{i+1} \cdot F_{i+2} \cdot \ldots \cdot F_{i+\ell}$, есть $O\left(\log |F_{i+1}| + \log |F_{i+2}| + \cdots + \log |F_{i+\ell}|\right) = O(\log n)$. При балансировке грамматик каждая операция вращения порождает не более трех новых правил. Всего общее число тех правил в ПП $\mathbb{F}$, которые отсутствуют в ПП $\mathbb{S}$, есть $O(\ell \log n)$. Аналогично, число новых правил, которые придется добавить при конкатенации $\mathbb{S}$ и $\mathbb{F}$, есть $O(\log n)$, поэтому размер ПП $\mathbb{S} \cdot \mathbb{F}$, выводящей текст $F_1 \cdot F_2 \cdot \ldots \cdot F_{i+\ell}$, есть $O((i+\ell) \log n)$. □

Что касается временной сложности модифицированного алгоритма Риттера, то она не может быть меньше чем сложность исходного алгоритма

из [9], поскольку последний является специальным случаем предлагаемой модификации, когда размеры всех групп факторов равны 1. С неформальной точки зрения ясно, что новый алгоритм порождает меньше вращений, но, с другой стороны, тратит дополнительное время на вычисление оптимального порядка конкатенации. Суммарное влияние этих обстоятельств на время работы алгоритма неочевидно. В следующем параграфе представлены результаты экспериментального сравнения обсуждаемых алгоритмов.

3.4. Построение ПП на основе декартовых деревьев Выше уже обсуждалось, что алгоритмы построения ПП, использующие AVL-деревья, должны тратить время на поддержание баланса в дереве. В связи с этим возникает идея применить для представления дерева вывода другую структуру данных. В этом разделе мы опишем алгоритм, который строит ПП, деревья вывода которых являются декартовыми деревьями.

Для декартовых деревьев имеется вероятностная логарифмическая от числа узлов оценка высоты, см. [10, 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 $n \left(\frac{n}{e}\right)^{-c \ln(c/e)}$.

Для построения ПП по заданной факторизации исходного текста требуются две операции над деревьями вывода: операция взятия поддерева по заданным границам и операция конкатенации двух деревьев. Для декартовых деревьев легко реализуются операции *split* – операция разбиения декартова дерева на два поддерева по заданной границе – и *merge* – операция слияния двух декартовых деревьев. Однако при стандартной реализации операции *merge* требуется дополнительное условие: каждый из ключей первого дерева должен быть меньше каждого из ключей второго дерева. Следовательно, чтобы использовать декартовы деревья в алгоритме построения ПП необходимо уметь корректно перегенерировать ключи для деревьев, полученных после применения операции *split*. Эта задача возникает в основном цикле алгоритма, когда алгоритм уже построил некоторое дерево T; для обработки следующего фактора алгоритм должен вырезать отвечающее этому фактору поддерево T' из T и затем должен выполнить операцию *merge* для T и T'. Перед слиянием необходимо полностью перегенерировать ключи для дерева T'. Оказывается, что для упрощения этой процедуры выгодно отказаться от явного хранения ключей. Поясним, почему это возможно[1].

Пусть T – произвольное декартово дерево и пусть информация о его ключах была утеряна. Тогда по структуре дерева всегда можно восстановить отношение линейного порядка на ключах. Для этого обойдем дерево рекурсивно в следующем порядке: левое поддерево, корень, правое поддерево. Тогда порядковый номер ключа данного узла среди всех ключей узлов

[1] К сожалению, изящная идея декартова дерева без явного хранения ключей, насколько нам известно, пока не освещалась в академической литературе. Достаточно полно эта идея представлена в интернет-публикации [2]. Достоверно известно, что декартовы деревья с неявными ключами возникли в 2002 г. в олимпиадном программировании, а авторами идеи были Н. В. Дуров (член студенческой сборной СПбГУ) и А. С. Лопатин (?)

дерева на единицу больше числа узлов в той части дерева, которое мы обошли до того, как попали в данный узел. Таким образом, можно отказаться от явного хранения значений ключей.

Определение 3.4. *Декартовым деревом с неявным ключом* называется декартово дерево, в узлах которого не хранятся значения ключей.

В дальнейшем под значением ключа узла декартова дерева T с неявным ключом мы будем подразумевать порядковый номер ключа этого узла в линейном упорядочении ключей для T. Поддерево с корнем в узле T_i обозначим через $\overline{T_i}$, а число узлов в этом поддереве – через $\mathrm{count}(T_i)$. Запись вида $T_i = (T_\ell, T_r)$ будем использовать для обозначения того факта, что узлы T_ℓ и T_r суть соответственно левый и правый сыновья узла T_i; при этом в правой части такой записи узлы T_ℓ и/или T_r могут быть пустыми (например, если T_i – лист, то и T_ℓ, и T_r пусты). Аналогично, запись вида $T = (L, R)$ означает, что L и R суть соответственно левое и правое поддеревья дерева T; при этом возможно, что L и/или R пусты.

Опишем, как реализуются операции *split* и *merge* для декартовых деревьев с неявным ключом.

Операция *split*. Эта операция получает на вход декартово дерево T с неявным ключом и целое положительное число $k \le |T| + 1$ и возвращает такую пару декартовых деревьев L и R с неявным ключом, что L содержит все узлы дерева T со значениями ключа, меньшими k, а R содержит все остальные узлы дерева T. Условимся, что на входе (пустое дерево,1) операция *split* возвращает пару пустых деревьев.

Алгоритм, реализующий *split*, работает рекурсивно, начиная с корня T_0 дерева T. Пусть $T_0 = (T_\ell, T_r)$. Возможны два случая.

(S1) Если $k \le \mathrm{count}(T_\ell) + 1$, то корень T_0 должен попасть в дерево R и нужно разрезать поддерево $\overline{T}_\ell$. Если *split* на входе $(\overline{T}_\ell, k)$ возвращает пару деревьев L' и R', то $L = L'$, а $R = (R', \overline{T}_r)$.

(S2) Если $k > \mathrm{count}(T_\ell) + 1$, то корень T_0 должен попасть в дерево L и нужно разрезать поддерево T_r. Если *split* на входе $(\overline{T}_r, k - \mathrm{count}(T_\ell) - 1)$ возвращает пару деревьев L' и R', то $L = (\overline{T}_\ell, L')$, а $R = R'$.

Поскольку на каждом шаге алгоритм либо завершает работу, либо выполняет рекурсивный вызов с узлом меньшей высоты, ожидаемое время работы алгоритма есть $O(\log |T|)$, если хранить в каждом узле T_i число $\mathrm{count}(T_i)$.

Мы будем использовать *split* при построении ПП, а по определению ПП как грамматики в нормальной форме Хомского дерево вывода ПП должно быть максимальным. Поэтому нужно немного модифицировать операцию, чтобы гарантировать, что возвращаемые ей декартовы деревья максимальны. Сделать это очень просто – достаточно удалить из каждого из деревьев, возвращаемых *split*, все узлы, имеющие ровно одного сына. Более формально, если узел T_j – отец единственного сына T_k, то узел T_j удаляется из дерева. В новом дереве узел T_k объявляется корнем, если T_j был корнем, и

сыном того узла, сыном которого был T_j, если T_j не был корнем. Приоритеты узлов при этом не изменяются.

Ясно, что если исходное дерево T максимально, то на каждом шаге алгоритма, реализующего *split*, в каждом из деревьев L и R возникает максимум один узел с единственным сыном. Поэтому ожидаемое время выполнения описанной только что процедуры «максимизации» деревьев L и R есть $O(\log|T|)$. В действительности, при программной реализации эта процедура не требует отдельного прохода, так как она может быть просто встроена в алгоритм. В дальнейшем, говоря об операции *split*, мы всегда будем иметь в виду ее модификацию, возвращающую максимальные деревья.

Операция *merge*. Эта операция получает на вход пару декартовых деревьев T' и T'' с неявным ключом и возвращает декартово дерево T с неявным ключом, содержащее все узлы из T' и T''. Условимся, что если дерево T' пусто, то операция возвращает дерево T'', и наоборот, если T'' пусто, то возвращается T'.

Алгоритм, реализующий *merge*, работает рекурсивно, начиная с корней T'_0 и T''_0 деревьев T' и T''. Пусть $T'_0 = (T'_\ell, T'_r)$ и $T''_0 = (T''_k, T''_q)$. Поскольку приоритеты узлов выбираются случайно и независимо, можно предполагать, что все они различны. Тогда возможны два случая.

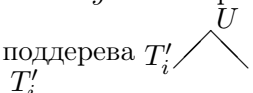
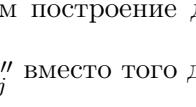
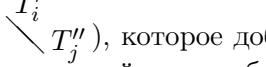
(M1) Если приоритет узла T'_0 больше приоритета узла T''_0, то дерево T имеет в качестве корня узел T'_0. Левым поддеревом для корня служит дерево $\overline{T}'_\ell$, а правым – то дерево, которое *merge* возвращает на входе $(\overline{T}'_r, T'')$.

(M2) Если приоритет узла T'_0 меньше приоритета узла T''_0, то дерево T имеет в качестве корня узел T''_0. Правым поддеревом для него служит дерево $\overline{T}''_q$, а левым – то дерево, которое *merge* возвращает на входе $(T', \overline{T}''_k)$.

Так как на каждом шаге рекурсии алгоритм спускается либо внутри первого дерева, либо внутри второго дерева, математическое ожидание времени работы алгоритма есть $O(\log|T'| + \log|T''|)$.

Так же, как и в случае *split*, придется немного модифицировать операцию *merge*, чтобы приспособить ее к построению ПП. Здесь возникают две проблемы. Во-первых, мы должны гарантировать максимальность возвращаемого декартова дерева. Во-вторых, поскольку операция *merge* будет использоваться для конкатенации ПП, необходимо, чтобы массив листьев дерева T получался приписыванием к массиву листьев дерева T' массива листьев дерева T''. Оказывается, что обе эти проблемы одновременно решаются следующей простой модификацией стандартного алгоритма.

Пусть T'_i – крайний правый лист дерева T', а T''_j – крайний левый лист дерева T'' (заметим, что в декартовых деревьях с неявным ключом такие «крайние» листья определяются однозначно), и пусть y' и соответственно y'' – приоритеты этих листьев. Положим $y_* = \min(y', y'')$, $y^* = \max(y', y'')$. Переопределим приоритеты узлов T'_i и T''_j, полагая их равными y_*. Тогда алгоритм, выполняя правила (M1) и (M2), необходимо придет к конфигурации, в которой роль текущих корней T'_0 и T''_0 играют листья T'_i и T''_j

соответственно. На этом шаге введем новый узел $U = (T'_i, T''_j)$ с приоритетом y^* и завершим построение дерева T добавлением трехэлементного поддерева  вместо того двухэлементного поддерева ( или ), которое добавил бы стандартный алгоритм. Ясно, что модифицированный таким образом алгоритм выполняется за то же ожидаемое время $O(\log |T'| + \log |T''|)$, и нетрудно проверить, что возвращаемое им дерево T максимально, если максимальны деревья T' и T''. Более того, T является конкатенацией T' и T'' в смысле ПП, см. §2. В дальнейшем, говоря об операции *merge*, мы всегда будем иметь в виду ее описанную модификацию.

Теперь приведем алгоритм построения «декартовой» ПП, т. е. ПП, дерево вывода которой является декартовым деревом с неявным ключом.

Алгоритм построения декартовой ПП получает на вход LZ-факторизацию F_1, F_2, ..., F_k данного текста S и индуктивно строит декартову ПП $\mathbb{S}$, выводящую текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$ для $i = 1, 2, \ldots, k$.

Инициализация: Полагаем $\mathbb{S}$ равной грамматике, состоящей из терминального правила, выводящего $F_1 = S[0]$.

Основной цикл: Предположим, что для фиксированного $i > 1$ уже построена декартова ПП $\mathbb{S}$, выводящая текст $F_1 \cdot F_2 \cdot \ldots \cdot F_i$. По определению LZ-факторизации фактор F_{i+1} является подстрокой в тексте $S = F_1 \cdot F_2 \cdot \ldots \cdot F_i$. Фиксируем такие позиции ℓ и r в тексте S, что $F_{i+1} = S[\ell \ldots r]$. Пусть ℓ^* и r^* – значения ключей для листьев $S[\ell]$ и соответственно $S[r]$ дерева $\mathbb{S}$. Заметим, что значения ℓ^* и r^* легко вычисляются по ℓ и соответственно r, если хранить в каждом узле $\mathbb{S}_i$ число $\mathrm{count}(\mathbb{S}_i)$.

Применим операцию *split* к входу $(\mathbb{S}, \ell^*)$ и пусть R – правое из деревьев, возвращаемых этой операцией. Теперь применим *split* к входу $(R, r^* - \ell^*)$. Левое из возвращаемых деревьев даст декартову ПП $\mathbb{F}$, которая выводит текст F_{i+1}. Применение операции *merge* к $\mathbb{S}$ и $\mathbb{F}$ дает декартову ПП, которая выводит текст $F_1 \cdot F_2 \cdot \ldots \cdot F_{i+1}$.

Теорема 3.4. *Математическое ожидание времени работы описанного алгоритма на тексте S длины n и его LZ-факторизации размера k равно $O(k \log n)$. Математическое ожидание размера ПП, возвращаемой алгоритмом, также равно $O(k \log n)$.*

Доказательство. На каждом шаге алгоритма запускается не более двух раз операция *split* и не более одного раза операция *merge*. Из описания этих операций следует, что математическое ожидание времени работы каждого шага алгоритма есть $O(\log n)$, а так как алгоритм делает ровно k шагов, математическое ожидание всего времени работы алгоритма есть $O(k \log n)$.

Для того, чтобы оценить размер декартовой ПП, возвращаемой алгоритмом, заметим, что на каждом шаге каждой из операций *split* и *merge* возникает не более одного нового нетерминального правила. Ожидаемое число шагов каждой операции есть $O(\log n)$, а в сумме обе операции выполняются $3k$ раз. Поэтому математическое ожидание размера ПП, возвращаемой алгоритмом, есть $O(k \log n)$. □

§4. Экспериментальные результаты

4.1. 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 алгоритмы построения ПП из §3 с классическими алгоритмами сжатия из семейства Лемпеля-Зива. Наш тестовый набор алгоритмов содержит два варианта алгоритма Лемпеля-Зива [14]: алгоритм с малым (32КБ) размером окна сжатия и алгоритм с бесконечным окном сжатия, а также алгоритм Лемпеля-Зива-Велча [13]. 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.

4.2. 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. 4–7. 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 [9];
- • – модифицированный алгоритм Риттера, см. п. 3.3;
- ▲ – алгоритм построения декартовой ПП, см. п. 3.4.

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 алгоритмов построения ПП с помощью AVL-деревьев подсчитывалось также число произведенных операций вращения.

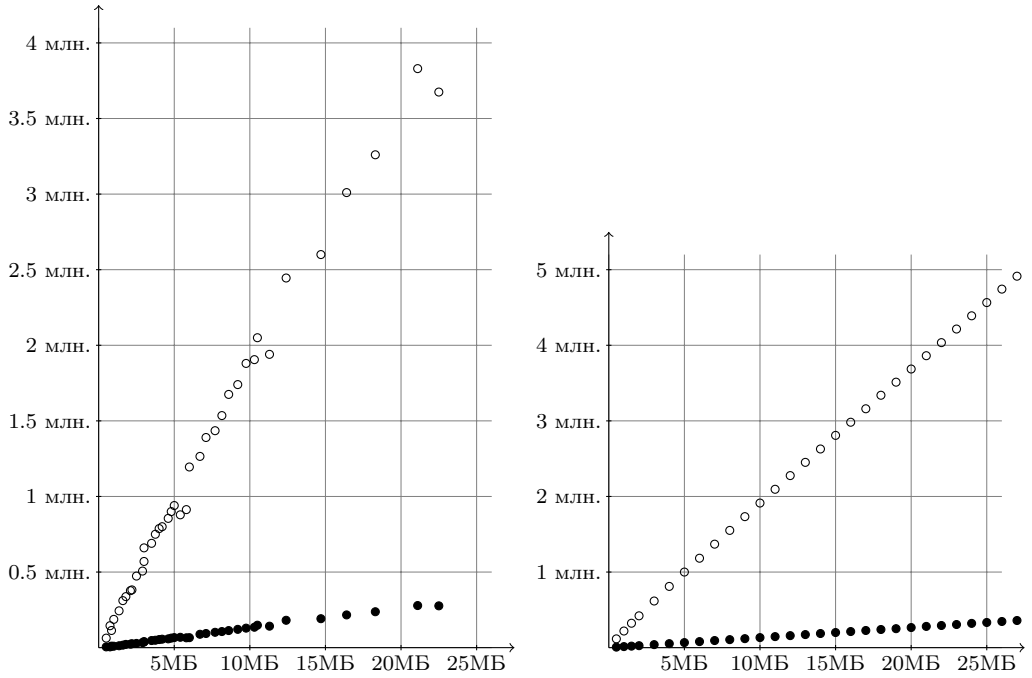


Рис. 4: Число вращений AVL-дерева (по оси ординат) при построении ПП для последовательностей ДНК (слева) и для случайных строк (справа)

Рис. 4 демонстрирует, как сказывается описанная в п. 3.3 модификация алгоритма Риттера на числе вращений. Видно, что уже для текстов длиной около 10МБ модифицированный алгоритм использует на порядок меньше вращений. Это свидетельствует об эффективности предложенной эвристики. Обращает на себя внимание то обстоятельство, что для обоих алгоритмов число вращений довольно регулярно зависит от длины входного текста и в то же время мало зависит от его природы. Мы не располагаем никаким теоретическим объяснением для этих наблюдений.

Как уже обсуждалось в п. 3.3, выигрыш в числе вращений еще не гарантирует выигрыш в скорости построения ПП, поскольку модифицированный алгоритм тратит дополнительное время на вычисление оптимального порядка конкатенации. Для сравнения скорости алгоритмов из §3 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. 5 и 6 представлены результаты тестов обоих типов соответственно на последовательностях ДНК и на случайных строках. Видно, что при хранении ПП в оперативной памяти время работы модифицирован-

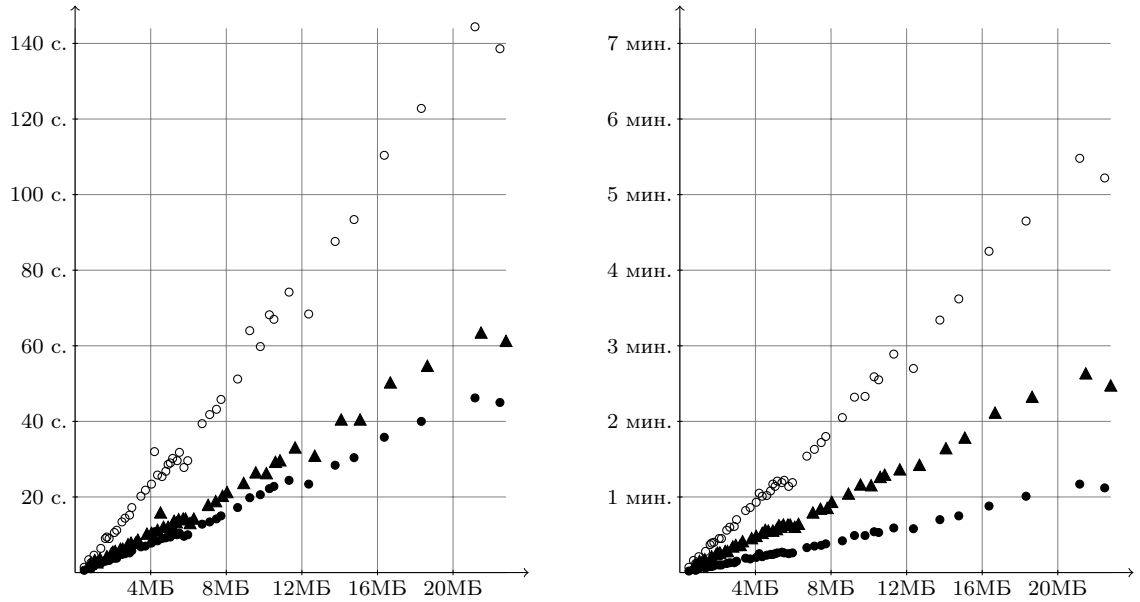


Рис. 5: Время работы алгоритмов построения ПП на последовательностях ДНК при хранении ПП в памяти (слева) и в файле (справа)

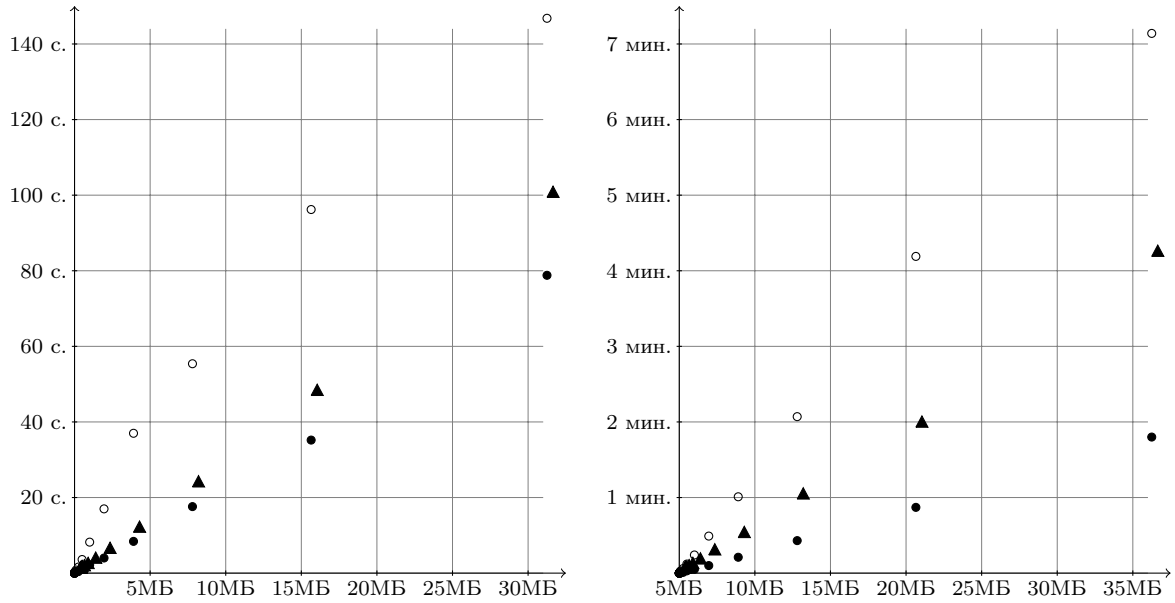


Рис. 6: Время работы алгоритмов построения ПП на случайных строках при хранении ПП в памяти (слева) и в файле (справа)

ного алгоритма, описанного в п. 3.3, 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 декартовы деревья, 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 декартовых ПП приходится обрабатывать намного больше правил, что сводит на нет весь выигрыш, который возникает за счет простоты поддержания баланса в декартовых деревьях.

На рис. 7 степень сжатия, достигаемая алгоритмами построения ПП, сравнивается со степенью сжатия алгоритмов из семейства Лемпеля-Зива. Видно, что алгоритмы, строящие AVL-деревья, обеспечивают практически одинаковую степень сжатия, которая хуже степени сжатия алгоритмов Лемпеля-Зива(-Велча) примерно в два раза. Интересно, что отношение степени сжатия алгоритмов, строящих AVL-деревья, к степени сжатия алгоритмов Лемпеля-Зива(-Велча) по существу не зависит ни от типа сжимаемого текста, ни от его длины. У алгоритма, строящего декартовы ПП, степень сжатия заметно хуже, чем у алгоритмов, строящих AVL-деревья. И здесь можно отметить, что отношение между степенями сжатия мало изменяется при изменении типа и/или размера входных данных.

§5. Выводы и дальнейшие перспективы

Наши эксперименты демонстрируют, что предложенная модификация алгоритма Риттера не уступает исходной версии этого алгоритма по достигаемой степени сжатия и заметно превосходит ее по скорости построения ПП. Поскольку с возрастанием длины входного текста использование файловой системы становится неизбежным, можно заключить, что модифицированный алгоритм также более устойчив к росту входных данных.

Другой алгоритм, представленный в работе, использует для построения ПП декартовы деревья. Он близок по скорости работы к алгоритмам, основанным на AVL-деревьях, по скорости, но ощутимо уступает им по степени сжатия и по высоте полученного сжатого представления. Последнее обстоятельство существенно с точки зрения влияния на быстродействия алгоритмов, работающих непосредственно со сжатыми представлениями данных. Таким образом, цель алгоритмически ускорить построение ПП с помощью «прогрессивной» структуры данных нельзя считать достигнутой, более того, такая цель сейчас представляется нам труднодостижимой. По-видимому, большой отдачи можно ожидать от поиска новых эвристик, позволяющих строить более компактные ПП, основанные на AVL-деревьях.

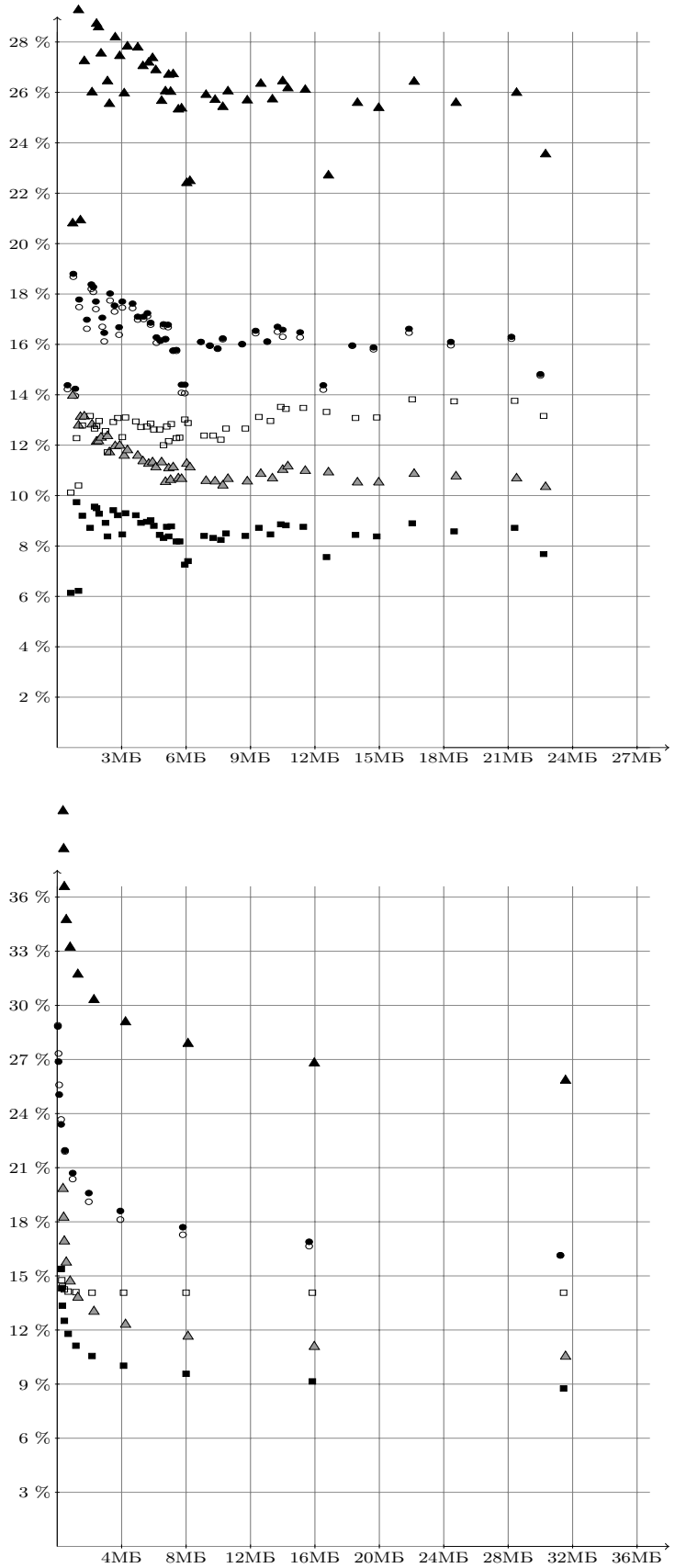


Рис. 7: Степень сжатия на последовательностях ДНК (сверху) и на случайных строках (снизу)

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, 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.

Авторы выражают свою благодарность профессору М.В. Волкову за его критические замечания и всестороннюю поддержку их деятельности. Авторы благодарны анонимному рецензенту за его замечания и предложения по улучшению исходного текста работы.

Список литературы

[1] *Д. Кнут*, Искусство программирования, том 3. Сортировка и поиск, 2-е изд. М.: «Вильямс», 2007.

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

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

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

[5] *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.

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

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

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

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

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

[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.