Содержание

1 Введение

Представление данных в сжатом виде позволяет экономить место, требуемое для их хранения. Тем не менее для какой-либо обработки сжатых данных приходится тратить время на их распаковку, а затем работать с несжатым представлением. Один из возможных подходов к решению указанной проблемы состоит в разработке алгоритмов, оперирующих непосредственно со сжатыми представлениями. Имеется много разных методов сжатого представления данных, для которых уже разработаны такие алгоритмы: коллаж-системы [2], представления с помощью антисловарей [3], прямолинейные программы [4] и др.

Ясно, что алгоритмы, работающие со сжатыми представлениями, существенным образом зависят от механизма сжатия. Сжатие текста с помощью контекстно свободных грамматик (таких, как прямолинейные программы) выделяется среди прочих методов двумя обстоятельствами. Во-первых, грамматики обеспечивают хорошо структурированное сжатое представление, что удобно для последующей алгоритмической обработки. Во-вторых, сжатие в виде прямолинейных программ полиномиально эквивалентно сжатию данных с помощью широко применяемых на практике алгоритмов из семейства алгоритмов Лемпеля-Зива (таких, как LZ77 [5], LZ78 [6], LZW [7]). Полиномиальная эквивалентность здесь понимается в следующем смысле: существует полиномиальная зависимость между размером прямолинейной программы, выводящей данный текст S, и размером словаря, построенным алгоритмом Лемпеля-Зива для S, см. [4].

Существует довольно большой класс задач, для которых разработаны алгоритмы со временем работы, полиномиальным относительно размера прямолинейной программы. К этому классу относятся, например, задачи **Поиск образца в тексте** [8], **Наибольшая общая подстрока** [9], считающая версия задачи **Поиск всех палиндромов** [9], некоторые версии задачи **Наибольшая общая подпоследовательность** [10]. В то же время константы, которые скрываются за «О большим» в имеющихся оценках сложности таких алгоритмов, как правило, очень велики. Кроме того, упо-

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

Известно, что задача построения прямолинейной программы минимального размера NP-полна [11]. Именно поэтому любой из практически применимых алгоритмов позволяет построить прямолинейную программу, являющую лишь неким приближением к минимальной.

В данной работе представлен обзор следующих алгоритмов построения прямолинейной программы:

- классический алгоритм, предложенный Риттером [4];
- эвристические модификации алгоритма Риттера, направленных на ускорение работы алгоритма [12];
- алгоритм Риттера, в котором в качестве основной структуры данных вместо AVL-деревьев используются рандомизированные деревья [13];
- многопоточный алгоритм, основанный на идеях алгоритма Риттера;
- LCA-online[1] алгоритм, предложенный в [14].

В первых четырех из перечисленных алгоритмов в качестве основной структуры данных используются сбалансированные бинарные деревья, а размер построенной прямолинейной программы гарантированно не превосходит размер минимальной в $O(\log n)$, где n — длина текста. Последний алгоритм позволяет построить лишь $O(\log^2 n)$-приближение, зато не требует использования дополнительных структур данных и работает за линейное время от размера текста.

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

2 Обозначения и основные понятия

Через $P\{A\}$ обозначается вероятность наступления события A.

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

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

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

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

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

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

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

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

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

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

Дерево вывода этой грамматики изображено на рисунке 1.

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

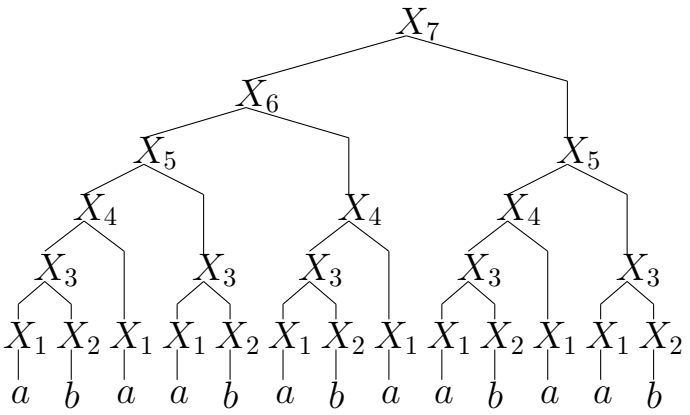

Рис. 1: Дерево вывода грамматики, порождающей «*abaababaabaab*».

3 Построение ПП с помощью AVL-деревьев

3.1 Алгоритм Риттера

Риттер [4] сформулировал алгоритм построения ПП, выводящей данный текст, по LZ-факторизации этого текста. При этом в качестве основной структуры данных используется AVL-дерево. AVL-дерево –это двоичное дерево, у каждого внутреннего узла которого высоты сыновей отличаются не более чем на 1.

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

Доказательство этого факта можно найти в [4].

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

Доказательство утверждения изначально является конструктивным. Опишем ключевые идеи алгоритма Риттера. Подробное описание и доказательства можно найти в [4].

AVL-грамматиками называются ПП, деревья вывода которых являются AVL-деревьями. Основными операциями, используемыми в алгоритме, яв-

ляются конкатенация AVL-грамматик и взятие подграмматики AVL-грамматики. Следующие леммы оценивают сверху трудоемкость этих операций.

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

Лемма 3.2 (о взятии подграмматики AVL-грамматики)**.** *Пусть T – дерево вывода AVL-грамматики, высота которой равна h. Тогда для любых $1 \leq i \leq j \leq |S(T)|$, добавив не более $O(h)$ новых правил, за время $O(h)$ можно построить AVL-грамматику T', которая выводит текст $S(T)[i..j]$.*

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

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

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

3.2 Модернизированный алгоритм Риттера

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

Идея оптимизации [13] состоит в том, чтобы откладывать конкатенацию деревьев до тех пор, пока следующий фактор целиком содержится в T, и затем, пользуясь свойством ассоциативности операции конкатенации, выбрать более эффективный порядок конкатенаций накопившихся деревьев. Интуитивное обоснование этой идеи таково: если уже построена «большая» ПП, то большинство последующих факторов входит в текст, выводимый из нее, а значит, эти факторы могут быть обработаны вместе.

В работе [13] оптимальный порядок конкатенаций предлагается находить решением задачи динамического программирования. Подробности можно найти в [1] или в оригинальной статье [13].

Таким образом, выбрав оптимальный порядок конкатенаций и уменьшив количество поворотов при конкатенации деревьев, мы получаем выигрыш по времени. Правда, это получается ценой дополнительных ресурсов, необходимых для решения задачи динамического программирования.

В [1] также предложено небольшое обобщение данной эвристики, направленное на уменьшение времени оптимизации порядка конкатенаций при большом количестве деревьев.

4 Построение ПП с помощью рандомизированных деревьев

Результаты этой главы были получены автором и опубликованы в [13]. Все доказательства были опущены для краткости изложения.

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

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

Для рандомизированных деревьев имеется вероятностная логарифмическая от числа узлов оценка высоты, а именно, ожидаемая высота рандо-

мизированного дерева с n узлами есть $O(\log n)$ [16]. Более того, для любой константы $c > 1$ вероятность того, что высота рандомизированного дерева с n узлами больше $2c \ln n$, ограничена величиной $n \left(\frac{n}{e}\right)^{-c \ln(c/e)}$.

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

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

4.1 Рандомизированные деревья вывода ПП

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

- Это дерево состоит ровно из одного узла, и в нем хранится выводимый терминал.
- Оба поддерева L и R являются независимыми рандомизированными деревьями, и $P\{|S(L)| = i\} = \frac{1}{n-1}$ для любого i от 1 до $n-1$, где $n = |S(T)|$.

Лемма 4.1 (о высоте рандомизированной ПП)**.** *Пусть T – рандомизированная ПП, тогда матожидание высоты дерева T равно $O(\log |S(T)|)$.*

Лемма 4.2 (о конкатенации рандомизированных ПП)**.** *Пусть T, T' – деревья вывода рандомизированных ПП, высота которых равна h и h' соответственно. Тогда, добавив не более чем $O(|h + h'|)$ новых правил, за время $O(|h + h'|)$ можно построить рандомизированную ПП $T \cdot T'$, которая выводит текст $S(T) \cdot S(T')$.*

Лемма 4.3 (о взятии подстроки рандомизированной ПП)**.** *Пусть T – дерево вывода рандомизированной ПП, высота которой равна h. Тогда для*

любых $1 \le i \le j \le |S(T)|$, *добавив не более* $O(h)$ *новых правил, за время* $O(h)$ *можно построить рандомизированную ПП* T', *которая выводит текст* $S(T)[i..j]$.

Сам алгоритм построения ПП по LZ-факторизации полностью повторяет алгоритм Риттера.

Утверждение 4.1. *Математическое ожидание размера результирующей ПП равно в среднем* $O(k \log n)$, *а среднее время работы алгоритма* $O(k \log n)$.

5 Многопоточный алгоритм построения ПП

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

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

В [1] предложен следующий алгоритм. В нем действия в алгоритме Риттера переставлены так, что некоторые последовательные действия можно выполнять параллельно.

Многопоточный алгоритм построения ПП с помощью бинарных деревьев

Вход: Текст S и его LZ-факторизация.

Выход: ПП, выводящая текст S.
Алгоритм:

1. Завести множество бинарных деревьев A, изначально оно пусто.
2. Найти все еще неиспользованные факторы, для которых можно получить дерево с помощью не более одной операции взятия подстроки дерева из A.
3. Выполнить все операции взятия подстроки параллельно и добавить их результаты в A.
4. Заменить «соседние по S» деревья в множестве на их конкатенацию. Это можно также попытаться выполнить параллельно настолько, насколько это возможно.
5. Если множество A содержит единственное дерево, выводящее текст S, то построить и вернуть ПП по этому дереву.
6. Иначе вернуться на второй шаг.

Утверждение 5.1 (о размере ПП в многопоточном алгоритме)**.** *Пусть длина текста S равна n, а размер LZ-факторизации равен k. Если выполнить описанный выше алгоритм, используя в качестве бинарных деревьев AVL-деревья, а для выполнения операций конкатенации и взятия подстроки алгоритмы из [4], то размер результирующей ПП будет $O(k \log n)$.*

Утверждение 5.2. *Существует последовательность текстов $\{T_p\}$ над конечным алфавитом такая, что $\lim_{p\to\infty} |T_p| = \infty$ и размер факторизации $k_p = \Omega\left(\frac{|T_p|}{\log |T_p|}\right)$ и $c_p = \Omega\left(\frac{|T_p|}{\log |T_p|}\right)$, где k_p и c_p — размер LZ-факторизации для T_p и количество итераций многопоточного алгоритма, запущенного на T_p, соответственно.*

Доказательство данного утверждения является конструктивным. В результате получается последовательность слов над пятибуквенным алфавитом.

Тем не менее при проведении экспериментов было установлено, что на строках ДНК и на случайных строках количество итераций многопоточного алгоритма не очень велико.

6 LCA-online алгоритм

В этой главе рассмотрим алгоритм построения ПП, основанный на другой идее. Далее дается краткое описание данного алгоритма, а более подробное его описание можно найти в [14].

6.1 LCA-offline алгоритм

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

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

А теперь обсудим, как именно нужно выбирать пары символов так, чтобы размер получающейся ПП был небольшим.

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

Далее введем несколько определений.

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

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

Зафиксируем некоторое положительное число k. Рассмотрим корневое, полное бинарное дерево дерево T_k, листья которого пронумерованы числа-

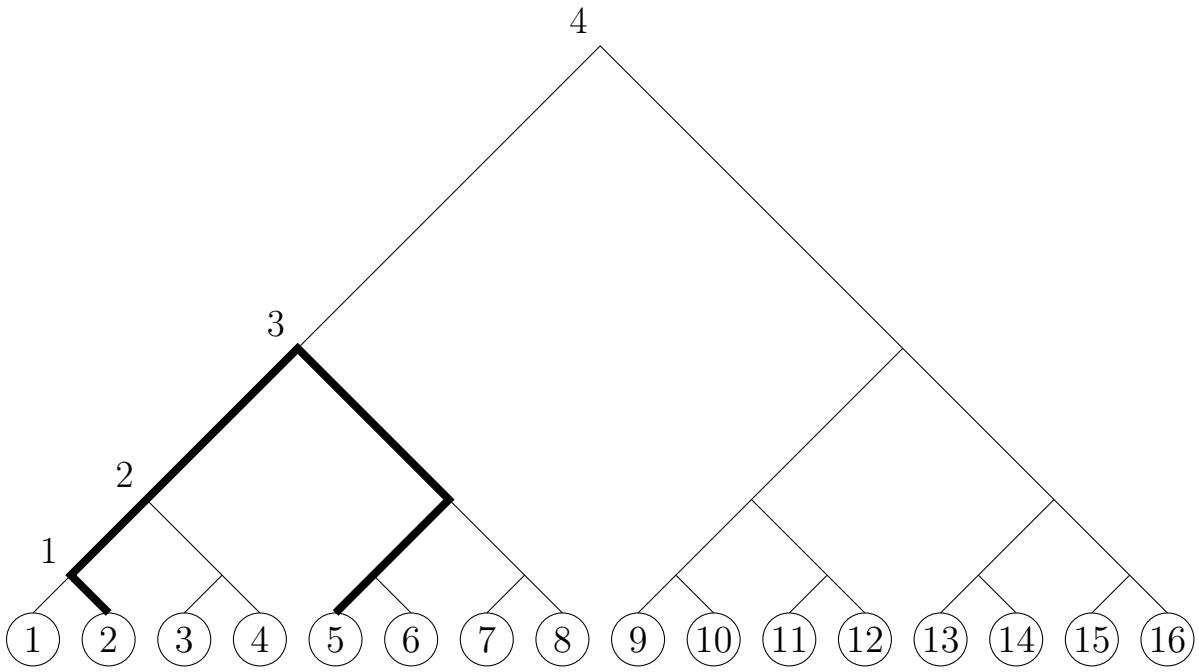

Рис. 2: Дерево T_4: $lca(2,5) = lca(5,2) = 3$.

ми от 1 до 2^k слева направо. Высотой вершины v будем называть длину пути от вершины v до листа в поддереве v. Тогда будем обозначать $lca(i,j)$ высоту наименьшего общего предка i-го и j-го листьев.

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

Теперь мы можем сформулировать алгоритм выбора пар для замены нетерминалами. Для начала сформулируем вспомогательный алгоритм по принятию решения об одной конкретной паре, а затем и сам алгоритм выбора пар.

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

Вход: Текст S и позиция i.

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

Алгоритм:

1. Если $i = |S|$, то вернуть $false$.
2. Иначе если $i + 4 > |S|$, то вернуть $true$.

3. Иначе если пара $S[i..i+1]$ – повторная, то вернуть $true$.

4. Иначе если пара $S[i+1..i+2]$ – повторная, то вернуть $false$.

5. Иначе если пара $S[i+2..i+3]$ – повторная, то вернуть $true$.

6. Иначе если пара $S[i..i+1]$ – минимальная или максимальная, то вернуть $true$.

7. Иначе если пара $S[i+1..i+2]$ – минимальная или максимальная, то вернуть $false$.

8. Иначе вернуть $true$.

Алгоритм выбора пар
Вход: Текст S.
Выход: Множество выбранных пар.
Алгоритм:

1. Присвоить $R := \emptyset$.

2. Присвоить $i := 1$.

3. Пока $i + 1 \leq |S|$ выполнять

   (a) Если $replaced\_pair(S, i) = true$, то добавить в R пару $S[i..i+1]$ и увеличить i на 2.

   (b) Иначе увеличить i на 1.

4. Вернуть множество R.

Утверждение 6.1. *Общее время работы алгоритма LCA-offline $O(n)$, где n — размер сжимаемого текста.*

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

Доказательство этого факта основано на логике алгоритма $replaced\_pair$ и будет опущено в виду своей громоздкости.

6.2 Преобразование LCA-offline алгоритма в LCA-online алгоритм

Идея преобразования заключается в том, что можно моделировать все итерации выбора и замены пар одновременно. Причем для моделирования каждой итерации достаточно хранить лишь некоторую конечную окрестность символов текущей позиции в тексте. Для этого заметим, что выполнение алгоритма $replaced\_pair(S, i)$ зависит только от подстроки $S[i-1..i+3]$. Таким образом, нам *уже* неважны символы до $(i-1)$-го и *еще* не важны символы после $(i+3)$-го.

Остальные детали данного преобразования можно посмотреть в [1] или в оригинальной статье [14].

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

7 Выводы и дальнейшие перспективы

В [1] приведены результаты сравнения всех описанных алгоритмов, запущенных на текстах разной природы. Сравнение производилось по двум основным параметрам (время сжатия и коэффициент сжатия), а также по нескольким вспомогательным.

По результатам проделанного исследования можно сделать следующие выводы.

Класс алгоритмов, основанных на представлении деревьев вывода в виде сбалансированных бинарных деревьев и строящих ПП на основе LZ-факторизаций, имеет два недостатка:

1. алгоритмы сложны и требовательны к ресурсам;
2. алгоритмы требуют предварительного построения LZ-факторизации, что также требует больших ресурсов.

Зато все эти алгоритмы гарантируют построение $O(\log n)$-приближения.

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

Эвристическая оптимизация порядка конкатенаций и многопоточный алгоритм построения привели к ускорению построения ПП. Однако при реализации многопоточного алгоритма возникло много сложностей, не все из которых удалось решить в рамках данной работы. Исправление всех этих проблем представляется возможным, но потребует огромных усилий.

Класс алгоритмов, строящих ПП на основе текста, содержит как правило простые в реализации алгоритмы, зато для них доказана худшая ассимптотика. В этой работе рассматривается алгоритм с лучшей известной доказанной ассимптотикой: LCA-online строит $O(\log^2 n)$-приближение. Теоретическая оценка LCA-online алгоритма была заведомо хуже оценок алгоритмов из класса, использующих сбалансированные бинарные деревья. Эксперименты же показали неожиданный результат — на практике этот алгоритм оказался лучшим по качеству сжатия. В дополнение этот алгоритм обладает рядом преимуществ: данный алгоритм имеет наибольшую скорость сжатия, в процессе построения не используется сложных структур данных, а также перед началом выполнения алгоритма не требуется построение LZ-факторизации. Безусловно, вопрос более тщательного изучения оценки качества сжатия данного алгоритма кажется весьма перспективным, как, пожалуй, единственного недостатка алгоритма.

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

оказаться, что оценка $O(log^2 n)$ верная, но достигается только на текстах определенной структуры, в то время как для практически интересных текстов эта оценка может быть улучшена.

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

Список литературы

[1] *Е. Курпилянский*, Классификация алгоритмов построения прямолинейных программ, Магистерская диссертация, `https://overclocking.googlecode.com/svn/texfiles/Kurpilyansky/MasterPaper/MasterPaper.pdf`.

[2] *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.

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

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

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

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

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

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

[9] *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.

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

[11] *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.

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

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

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

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

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