Содержание

1 Введение

С каждым годом растет объем доступных данных. Как следствие, все чаще возникает задача по обработке большого объема данных. Классические строковые алгоритмы напрямую зависят от размера входных данных. Более того, начиная с некоторого размера входа они просто не применимы для решения задач, так как ориентированы в основном на работу в оперативной памяти. Для того, чтобы справиться с растущим объемом входных данных, возникают новые классы алгоритмов, которые принципиально отличаются от классических строковых алгоритмов. Так, например, есть класс алгоритмов, которые хранят и обрабатывают входные данные с помощью файловой системы. Минусом этих алгоритмов является то, что они существенно медленней, чем классические, однако позволяют обработать несравнимо больший объем данных. Другим классом алгоритмов является класс алгоритмов обработки сжатых представлений.

Грамматическое сжатие является одним из применяемых методов сжатия данных. Его основной принцип – построение по строке некоторой грамматики, из которой эта строка выводится единственным образом.

Главным преимуществом грамматического сжатия является то, что оно позволяет производить операции со сжатой строкой без предварительной распаковки. Примером такой операции может служить использование строки в сжатом виде во время поиска подстроки в строке.

Грамматическое сжатие может быть применено в биологии для хранения ДНК-последовательностей. ДНК-последовательности занимают большие объемы, поэтому обычно они хранятся в сжатом виде. Возможность работать со строкой, не разжимая ее, является преимуществом грамматического сжатия. Примером использования такого представления может служить извлечение информации об образцах в исходной строке, а конкретно, грамматическое сжатие используется для распознавания образцов в ДНК-последовательностях [1].

Грамматическое сжатие хорошо работает и в случаях, когда строка представляет из себя одно слово, тогда как традиционные методы поиска в больших объемах данных подразумевают, что текст разбит на слова (например, построение обратного индекса).

Задача построения минимальной грамматики является NP-трудной [3], поэтому существенный интерес представляют эвристики. В данной работе предложен алгоритм приближенного построения прямолинейных программ, который ориентирован на параллельное построение.

2 Условные обозначения

В данной работе рассматриваются строки над конечным алфавитом Σ. Длина строки $S \in \Sigma^*$ равна количеству символов из Σ в S и будет обозначаться как $|S|$. Позицией в строке S называется место между соседними символами. Позиция 0 предшествует строке S, а позиция $|S|$ следует за строкой S. Для строки S и числа i такого, что $0 \leq i < |S|$, через символ $S[i]$ обозначается символ, расположенный между позициями i и $i+1$. Через $S[l \dots r]$, где $0 \leq l < r \leq |S| - 1$, обозначается подстрока строки S, начинающаяся в позиции l и заканчивающаяся в позиции r.

В качестве представления строки S будут использоваться контекстно-свободные грамматики. Контекстно-свободная грамматика (КС-грамматика) – это четверка вида $G = (\Sigma, \Gamma, P, S)$, где

- Σ – конечный терминальный алфавит;
- Γ – конечный нетерминальный алфавит ($\Sigma \bigcap \Gamma = \oslash$);
- P – множество правил вывода вида $A \to \alpha,\ A \in \Gamma,\ \alpha \in (\Sigma \bigcup \Gamma)^*$;
- $S \in \Gamma$ – начальный символ (аксиома).

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

$X_1 = expr_1; X_2 = expr_2; \dots; X_m = expr_m$, где X_i – нетерминальный символ, а $expr_i$ – это:

- $expr_i$ – символ из алфавита Σ (терминал), или
- $expr_i = X_l \cdot X_r$ $(l, r < i)$, где «$\cdot$» – конкатенация правил X_l и X_r.

ПП является грамматикой в нормальной форме Хомского, выводящая единственную строку.

Пример 1.
ПП, выводящая строку "aaaababaabaaaabab":

$$X_1 = a;\ X_2 = b;\ X_3 = X_1 \cdot X_1;\ X_4 = X_3 \cdot X_3;$$
$$X_5 = X_1 \cdot X_2;\ X_6 = X_2 \cdot X_5;\ X_7 = X_4 \cdot X_6;$$
$$X_8 = X_2 \cdot X_1;\ X_9 = X_3 \cdot X_8;\ X_{10} = X_7 \cdot X_9;$$
$$X_{11} = X_1 \cdot X_3;\ X_{12} = X_{11} \cdot X_6;\ X_{13} = X_{10} \cdot X_{12};$$

Каждой ПП можно сопоставить дерево вывода – некоторое упорядоченное корневое дерево, вершины которого помечены символами из $\Sigma \bigcup \Gamma$.

На рисунке 1 представлено дерево вывода ПП, выводящей строку "*aaaababaabaaaabab*".

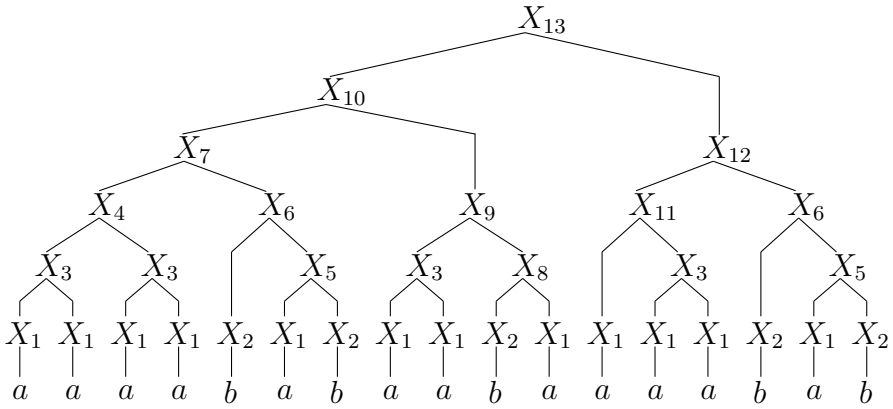

Рис. 1: ПП, выводящая строку “*aaaababaabaaaabab*”

Для представления дерева вывода ПП будет использоваться AVL-дерево. AVL-дерево – это двоичное дерево, у каждого внутреннего узла которого высоты сыновей отличаются не более чем на 1.

AVL-грамматиками называются ПП, деревья вывода которых являются AVL-деревьями.

Пусть $A \in \Gamma$. Тогда $val(A)$ – строка из терминалов, выводимая из A.

3 Постановка задачи

Задачу построения ПП можно сформулировать следующим образом:

Вход: Строка $S \in \Sigma^*$.

Выход: ПП, выводящая S.

Критерий оценивания: Размер получившейся ПП и скорость её построения.

Известно, что задача построения грамматики минимального размера для заданной строки является NP-трудной [3]. Поэтому для построения ПП используются эффективные приближенные алгоритмы. В данной работе для построения ПП будет использоваться подход, предложенный Риттером [5]. Этот подход предполагает представление строки в виде LZ-факторизации.

LZ-факторизацией строки S ($LZ(S)$) называется представление S как $f_1 \cdot f_2 \cdot \ldots \cdot f_k$, где $f_1 = S[1]$, и для любого $1 < l \leq k, f_l$ - это наибольший префикс $f_l \cdot \ldots \cdot f_k$, который встречается в $f_1 \cdot \ldots \cdot f_{l-1}$. Каждый f_l называется фактором. Размером $|LZ(S)|$ называется количество факторов в $LZ(S)$.

Исходя из этого, можно переформулировать исходную задачу в следующем виде:

Вход: Строка S и ее факторизация $LZ(S)$.

Выход: ПП, соответствующая S.

При использовании данного подхода для заданной строки S можно построить ПП за время $O(k \log n)$, размер которой будет $O(k \log n)$, где n - длина S, k - размер $LZ(S)$ [5].

4 Обзор уже существующих алгоритмов построения ПП

В рамках проблемы построения минимальной ПП проводились исследования, в ходе которых были предложены алгоритмы приближенного решения данной проблемы. В работе будет приведен обзор двух таких алгоритмов.

4.1 Алгоритм Риттера

В 2003 году Риттером [5] был представлен алгоритм грамматического сжатия, использующий представление строки в виде LZ-факторизации и основывающийся на AVL-грамматике.

На вход алгоритму передается строка S, $|S| = n$. После чего происходит вычисление LZ-факторизации $f_1 \cdot f_2 \cdot f_3 \cdot \ldots \cdot f_k$ строки S, которая может быть вычислена за время $O(n\ log|\Sigma|)$, используя суффиксные деревья. Если LZ-факторизация достаточно большая (превышает $n/log(n)$), тогда можно пренебречь алгоритмом и сгенерировать тривиальную грамматику размера n, соответствующую данной строке. Иначе происходит обработка LZ-факторизации слева-направо по следующему алгоритму. Предположим, что для некоторого префикса $f_1 \cdot f_2 \cdot f_3 \cdot \ldots \cdot f_{i-1}$ уже построена "хорошая" (AVL-сбалансированная и размера $O(i \log n)$) грамматика G. Если f_i - терминальный символ, то генерируется подходящий нетерминал A и $G := Concat(G, A)$. Иначе в префиксе $f_1 \cdot f_2 \cdot f_3 \ldots f_{i-1}$ ищется сегмент, который соответствует f_i. Используя тот факт, что G сбалансированная, можно найти логарифмическое число нетерминалов $S_1, S_2, \ldots, S_{t(i)}$ из G, таких что $f_i = val(S_1) \cdot val(S_2) \cdot \ldots val(S_{t(i)})$. Последовательность $S_1, S_2, \ldots, S_{t(i)}$ назовем грамматическим представлением фактора f_i.

После этого производится конкатенация частей грамматики, соответствующих нетерминалам G, используя операцию $Concat$. Можно заметить, что первые $|\Sigma|$ нетерминалов соответствуют буквам алфавита, которые встретились в начале. Инициализация G производится первым символом S и содержит все нетерминалы для терминальных символов.

На рисунке 2 представлен псевдокод описанного алгоритма.

Немного подробнее остановимся на операции конкатенации двух грамматик $Concat(A, B)$. Для этого рассмотрим операцию конкатенации двух деревьев вывода этих грамматик, $T1$ и $T2$ соответственно. Каждое из них является AVL-деревом, но в данном случае случае каждое дерево грамматики содержит ключи (символы) только в листьях, поэтому конкатенируя два дерева грамматик не нужно удалять корень одного из них (это подразумевает дорогостоящую перестройку дерева). Пусть $height(T1) \geq height(T2)$, другой случай является симметричным. Идя по правой ветви $T1$ можно заметить, что высота узлов уменьшается каждый раз не более чем на 2. Остановка происходит на некотором узле v таком, что $height(v) - height(T2) \in \{1, 0\}$. Затем создается новый

узел v', отец которого является отцом v и сыновья которого v и $root(T2)$. Получившееся дерево может быть несбалансированным по правой ветви. В этом случае происходит перебалансировка получившегося AVL-дерева.

Алгоритм вычисления ПП

```
1  Вычисляем LZ-факторизацию f_1 f_2 f_3 ... f_k
2  if k > n/log(n)
3      then
4          return тривиальная грамматика размера O(n);
5      else
6          for i = 1 to k
7              do
8                  (1) Пусть S_1, S_2, ..., S_{t(i)} -
   грамматическое представление f_i;
9                  (2) H := Concat(S_1, S_2, ..., S_{t(i)});
10                 (3) G := Concat(G, H);
11 return G;
```

Рис. 2: Псевдокод алгоритма Риттера

Пример 2.

Рассмотрим пример работы данного алгоритма для строки "aaaababaabaaaabab". LZ-факторизацией данной строки будет являться следующее представление:

$$f_1 = a,\ f_2 = a,\ f_3 = aa,\ f_4 = b,\ f_5 = ab,\ f_6 = aaba,\ f_7 = aaabab$$

Дерево вывода грамматики, построенное для данной строки, изображено на рисунке 3 (здесь приводится вариант без перебалансировок, чтобы можно было видеть различия при сравнении с другими алгоритмами).

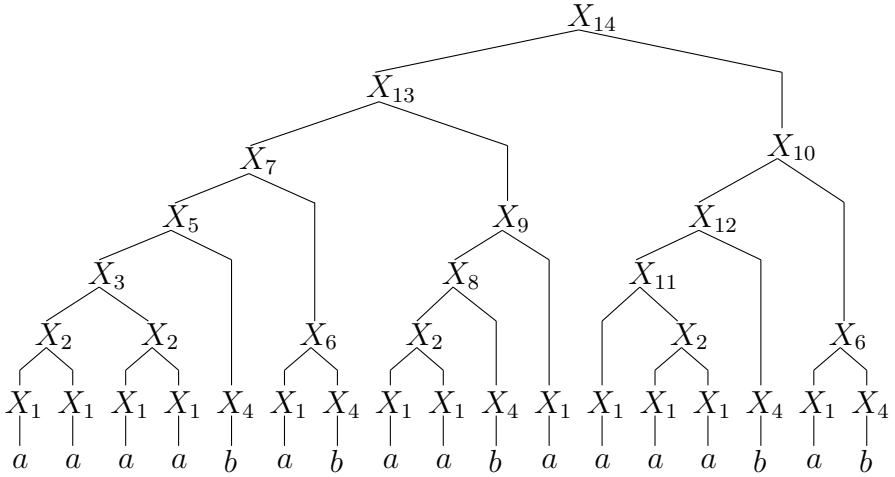

Рис. 3: ПП, выводящая строку "*aaaababaabaaaabab*"

Теорема 1.
За время $O(n \log |\Sigma|)$ можно построить приближение минимальному грамматическому сжатию с коэффициентом $O(\log n)$. Используя LZ-факторизация длины k, можно построить соответствующую грамматику размера $O(k \log n)$ за время $O(k \log n)$ [5].

Узким местом данного алгоритма является использование AVL-деревьев, а именно работа с ними затрудняется поддержанием 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, каждое из которых может порождать до трех новых узлов, а так же сделать до трех узлов неиспользуемыми.

Лемма 1.
Пусть A, B - два нетерминала AVL-сбалансированной грамматики. Тогда за время $O(|height(A) - height(B)|)$ можно построить AVL-сбалансированную грамматику $G = Concat(A, B)$, где $val(G) = val(A) \cdot val(B)$, добавлением не более $O(|height(A) - height(B)|)$ нетерминалов [5].

Из леммы можно видеть, что при конкатенации AVL-грамматик существенно разной высоты появляется много новых правил, при добавлении которых может возникнуть необходимость в большом числе вращений. При повторении основного цикла достаточное число раз, высота текущей AVL-грамматики становится большой, а при каждом последующем выполнении основного цикла с G конкатенируется AVL-грамматика относительно небольшой высоты.

Эта проблема была рассмотрена в [2], где был предложен вариант ее решения.

4.2 Простой онлайн алгоритм грамматического сжатия

Данный алгоритм рассматривается в рамках [4]. Он не основывается на идее использования представления строки в виде факторизации. Однако этот алгоритм интересен тем, что представляет собой существенно новый подход к данной проблеме.

Основываясь на идее LCA, производится замена пары XY, встречающейся в данной строке, на новый символ Z и генерируется правило $Z \to XY$, после чего все вхождения пары XY в строку заменяются на Z. Таким образом, целью этого алгоритма является сведение к минимуму количества различных нетерминалов. Замена осуществляется использованием одного из трех правил.

Повторение – это строка x^k для некоторого символа x и целого $k \geq 2$. повторение $S[i,j] = x^k$ называется максимальным, если $S[i-1] \neq x$ и $S[j+1] \neq x$.

Первое правило (повторяющаяся пара): Пусть данная строка S содержит максимальное повторение $S[i,j] = a^k$. Тогда генерируется новое правило $A \to aa$, для соответствующего нетерминала A и производится замена $S[i,j] = A^{k/2}$, если k – четное, либо $S[i,j-1] = A^{(k-1)/2}$ для k нечетного.

Второе правило (минимальная пара): Введем общий порядок над $\Sigma \bigcup \Gamma$, с помощью которого каждый символ может быть представлен целым числом. Если данная строка содержит подстроку $A_iA_jA_k$, причем $j < i,k$, тогда вхождение A_j назовем минимальным. Второе правило заключается в замене всех таких пар A_jA_k в $A_iA_jA_k$ на соответствующий нетерминал.

Пусть d – положительное целое число и $k = \lceil log\ d \rceil$. Индексное дерево T_d – это корневое, упорядоченное, полное бинарное дерево, чьи листья помечены $1, \ldots 2^k$. Высотой узла v назовем количество ребер в наидлиннейшем пути от v до потомка v. Тогда высоту наименьшего общего предка листьев листьев i,j обозначим как $lca(i,j)$.

Третье правило (максимальная пара): При фиксированном порядке алфавита, если текущая строка содержит подстроку $A_iA_jA_kA_l$, такую, что числа i,j,k,l возрастают либо убывают и $lca(j,k) > lca(i,j),\ lca(k,l)$, тогда вхождение средней пары A_jA_k называется максимальным. По третьему правилу такая пара заменяются на соответствующий нетерминал.

Пару, заменяемую по одному из вышеуказанных правил, назовем специальной.

Онлайн алгоритм облегченного грамматического сжатия использует h очередей $q_1, q_2, \ldots, q_h$, где q_i представляет собой кольцевой буфер. Каждая q_i играет роль буфера хранения части строки S. Число h ограничено $O(log\ n)$ [4]. Для каждой очереди введем следующие операции:

- $enque(q_i, x)$: добавить символ x в хвост очереди q_i.
- $deque(q_i)$: вернуть голову очереди q_i и удалить ее.
- $head(q_i)$: вернуть голову очереди q_i.

- $len(q_i)$: вернуть длину очереди q_i.

Длину каждой очереди можно ограничить константой [4].

Для линейного времени сжатия будет использоваться обратный словарь $D^R(x, y)$, возвращающий нетерминал z, такой, что $z \to xy \in D$. Если $z \to xy \notin D$, тогда в $D^R(x, y)$ создается новое правило $z \to xy$ и возвращается только что созданный нетерминал z.

Псевдокод алгоритма приведен на рисунке 6. Помимо этого, так же приведены функции $IsPair$ и $InsertSymbol$, необходимые в ходе работы алгоритма.

```
IsPair(s, i)
1  if s[i, i+1] or s[i+2, i+3] – повторяющаяся пара
2      then return true;
3  if s[i+1, i+2] – повторяющаяся пара
4      then return false;
5  if s[i, i+1] – минимальная или максимальная пара
6      then return true;
7  if s[i+1, i+2] – минимальная или максимальная пара
8      then return false;
9  return true;
```

Рис. 4: Псевдокод функции $IsPair$

На вход функции $IsPair$ передается некоторая последовательность символов s и позиция, начиная с которой ищется заменяемая пара. Процедура выбирает одну из пар $s[i, i+1]$ или $s[i+1, i+2]$, которая будет заменена в соответствии с вышеуказанными правилами, причем $s[i, i+1]$ выбирается если пара $s[i+1, i+2]$ не является специальной.

Функция $InsertSymbol$, псевдокод которой приведен на рисунке 5, определяет, какая пара будет заменена на данном шаге. Если длина переданной очереди q_i больше 5, то определяется, какая именно пара $q_i[1, 2]$ либо $q_i[2, 3]$ будет заменена. В случае $q_i[1, 2]$ данная пара заменятся на соответствующий нетерминал z, $q_i[0, 1]$ извлекаются из очереди и z помещается в q_{i+1}. В случае замены $q_i[2, 3]$ на z, $q_i[0, 2]$ извлекается и $q_i[1]z$ помещается в q_{i+1}. Символ $q_i[2]$ в первом случае и $q_i[3]$ во втором запоминается в q_j для определения следующей заменяемой пары после добавления нового символа в q_{i+1}.

В начале работы алгоритма все очереди инициализируются фиктивным символом $d \notin \Sigma \bigcup \Gamma$, который необходим для вычисления первой пары в каждой очереди. В строках 7-10 алгоритма входные символы помещаются в очередь q_1 друг за другом и вызывается функция $InsertSymbol$, которая рекурсивно обрабатывает помещаемые символы. Алгоритм продолжает работу, пока все входные символы не будут помещены в очередь. В конце происходит обработка символов, оставшихся в очередях путем замены их на соответствующие нетерминалы в порядке слева-направо. Результатом работы алгоритма является получившийся словарь правил.

```
InsertSymbol(q_i, x)
 1  enque(q_i, x);
 2  if len(q_i) ≥ 5
 3     then
 4          if IsPair(q_i, 1) = true
 5             then
 6                  ▷ случай 1: q_i[1, 2] заменяемая пара
 7                  deque(q_i);
 8                  y_1 = deque(q_i);
 9                  y_2 = head(q_i);
10                  z := D^R(y_1, y_2), updateD;
11                  InsertSymbol(q_{i+1}, z);
12             else
13                  ▷ случай 2: q_i[2, 3] заменяемая пара
14                  deque(q_i);
15                  y_1 = deque(q_i);
16                  InsertSymbol(q_{i+1}, y_1);
17                  y_2 = deque(q_i);
18                  y_2 = head(q_i);
19                  z := D^R(y_2, y_3), updateD;
20                  InsertSymbol(q_{i+1}, z);
```

Рис. 5: Псевдокод функции $InsertSymbol$

```
Online LCA
 1  Создать пустые очереди q_1, q_2, ... q_h;
 2  i := 1;
 3  while i ≤ h
 4        do
 5           enque(q_i, d);
 6           i := i + 1;
 7  repeat
 8           считать символ c;
 9           InsertSymbol(q_1, c);
10     until c не является символом конца.
11  Обработать символы, оставшиеся в очередях;
12  return D;
```

Рис. 6: Псевдокод функции $OnlineLCA$

Пример 3.

Дерево вывода грамматики, построенное для строки "aaaababaabaaaabab" с помощью данного алгоритма, изображено на рисунке 7.

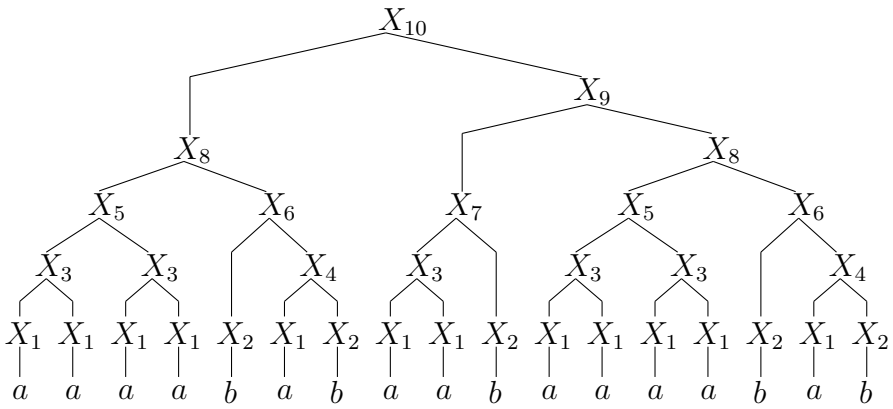

Рис. 7: ПП, выводящая строку "*aaaababaabaaaabab*"

Теорема 2.

Время работы предложенного алгоритма можно оценить $O(n)$, где n - длина исходной строки [4].

Теорема 3.

Приближенное оценка отношения $\frac{g}{g_m}$ для предложенного алгоритма $O(log^2\ n)$, где g - размер получающейся грамматики, g_m - размер минимальной грамматики, n - длина исходной строки [4].

В данном алгоритме, в отличии от алгоритма, предложенного Риттером, дерево вывода получающейся грамматики не является сбалансированным, из-за чего трудно оценить его высоту. Из-за этого возникает трудность использования данного алгоритма для дальнейшего применения к построенной грамматике алгоритмов поиска.

5 Алгоритм ленивого построения ПП

5.1 Идея

Первоначальной идеей для данного алгоритма был анализ LZ-факторизации текста с точки зрения использования символов. Под использованием в данном случае понимается частота употребления каждого символа текста в факторах (сколько раз конкретный символ используется при построении фактора). Для того, чтобы было удобней воспринимать частоты с точки зрения факторизации без привязки к конкретным символам, был осуществлен пересчет этих частот на факторы. В этом случае, у каждого фактора из факторизации появляется вес, который определяется отношением суммы соответствующих ему частот символов к длине подстроки, соответствующей данному фактору. Благодаря этому расширению понятия фактора, дальнейшую работу можно производить с точки зрения фактора и его веса, абстрагируясь от частот конкретных символов строки. Узнав вес каждого фактора, появилась возможность проанализировать частоту использования каждого фактора. Для факторов с большим весом количество использований подстроки, ему соответствующей, было большим, чем у фактора с меньшим весом. Поэтому вполне естественным будет предположение о порядке обработки каждого фактора. В алгоритме Риттера факторы обрабатывались один за другим в соответствии с их порядком в LZ-факторизации. В данном случае можно обрабатывать факторы в зависимости от их веса. Это позволит в начале построить факторы, которым соответствуют самые часто используемые подстроки, а значит когда будут строиться факторы с меньшим весом, то скорее всего для подстроки, которой соответствует данный фактор, деревья вывода уже будут построены и останется только сконкатенировать их.

Далее будет рассмотрен алгоритм, основывающийся на предложенной идее. Аналогично алгоритму Риттера, для построения ПП будут использоваться AVL-грамматика.

5.2 Структуры данных, которые понадобятся в ходе работы алгоритма

Для эффективной работы алгоритма будут применены следующие структуры.

- Зависимости факторов **association(factor)**

  Данная структура позволяет по конкретному символу узнать какой фактор ему соответствует. Использование этой структуры позволяет найти декомпозицию факторов, соответствующую данному. Ее можно мыслить как отображение $f_i \to f_{i_1} \cdot f_{i_2} \cdot \ldots \cdot f_{i_l}$ и пару чисел, указывающих где начало f_i в f_{i_1} и конец в f_{i_l}.

- Отображение фактора в дерево **treesMap(factor)**

Отображение, которое по фактору возвращает дерево, соответствующее этому фактору.

5.3 Функции, которые понадобятся при работе алгоритма

- Функция веса фактора **weight(factor)**

  Функция веса фактора вычисляется как отношение суммы частот всех символов, которые соответствуют данному фактору в исходной строке, к длине подстроки, соответствующей данному фактору.

- Нахождение последовательности поддеревьев, соответствующей заданным началу и концу **split(tree, begin, end)**

  Данная функция аналогична нахождению грамматического представления фактора в алгоритме Риттера. Результатом работы данной функции будет являться последовательность поддеревьев, соответствующих подстроке $S[begin \dots end]$. Функция *split* выполняется за время, не превышающее $O(\log n)$.

- Объединение последовательности поддеревьев **concat**(t_1, t_2, $\dots t_k$)

  На вход данной функции подается последовательность деревьев, на выходе - дерево, соответствующее объединению исходных деревьев в первоначальном порядке. Так же аналогично алгоритму Риттера.

- Создание дерева **CreateTree(factor)**

  Данная функция вызывается для каждого фактора для создания дерева вывода, соответствующего данному фактору. Псевдокод данной функции приведен на рисунке 8.

- Поиск дерева **Search(factor)**

  Эта функция используется в функции **CreateTree(factor)**, если для данного фактора дерево еще не создано. Псевдокод данной функции приведен на рисунке 9.

Лемма 2.
Для произвольного фактора f строки S процедура $CreateTree(f)$ создает дерево, соответствующее этому фактору, за время $O(k^2 \log n)$, где n - длина строки S, k - размер $LZ(S)$.

Доказательство: Рассмотрим работу функции $CreateTree(f)$ для некоторого фактора f.

В начале работы функции проверяется не было ли построено дерево для f ранее. Если дерево уже построено, а значит, содержится в $treesMap$, то оно достается из $treesMap$ и является результатом работы. Данный шаг требует $O(1)$ операций.

В противном случае для построения дерева для фактора f возможны два случая:

- $|f| = 1$.
  В этом случае фактор f является терминальным и для f создается дерево t с единственным узлом, выводящее f, которое сохраняется в $treesMap(f)$. После чего t возвращается как результат работы. Этот случай требует так же $O(1)$ операций.

- $|f| > 1$.
  В этом случае дерево строится с помощью процедуры $Search(f)$. В начале, для фактора f строится декомпозиция $factors$, используя структуру $association$. Такое представление является единственным по определению LZ-факторизации. Для каждого фактора f_l из декомпозиции $factors$ создается дерево t_l с помощью функции $CreateTree$. Если для некоторого f_l дерева еще построено не было, то он так же представляется в виде декомпозиции факторов $factors_l$ и для каждого f_{l_j} из декомпозиции строится дерево t_{l_j}. В худшем случае на некотором шаге получается декомпозиция терминальных факторов, для которых время построения дерева равно $O(1)$. Количество факторов в декомпозиции $factors$ не более чем $|f|$, тогда, с точки зрения исходной последовательности факторов в факторизации, для любого фактора f_i, количество факторов в декомпозиции не более чем i. Пусть $h_{max} = max\{height(t_l)\}$, $h_{min} = min\{height(t_l)\}$ и в исходной последовательности факторов $f = f_i$, тогда для конкатенации всех деревьев t_l необходимое количество операций можно оценить, используя **Лемму 1**, как $i * (h_{max} - h_{min}) \leq i * h_{max} \leq i * \log|f_i| = i * \log|f|$. В худшем случае $\log|f|$ оценивается $\log n$ и общее число операций, необходимое для построения дерева t для фактора f, равно $O(\sum_{i=1}^{k} i \log n)$ или $O(k^2 \log n)$. Дерево t является результатом работы функции $Search$, которое затем сохраняется в $treesMap(f)$ и функция $CreateTree$ завершает работу, вернув t.

Таким образом, время работы функции $CreateTree$ равно $O(k^2 \log n)$. □

```
CreateTree(f)
 1  Tree t = treesMap(f);
 2  if t == null
 3      then
 4           if f.isTerminal
 5              then
 6                   t = newTree(f.symbol);
 7              else
 8                   tree = Search(f);
 9           treesMap(f) = tree;
10  return tree;
```

Рис. 8: Псевдокод функции $CreateTree$

```
SEARCH(f)
 1  factors = association(f);
 2  trees = {};
 3  for Factorfactor : factors
 4        do
 5           trees.add(createTree(factor));
 6  Tree firstTree = trees.item(0);
 7  if firstTree.absoluteBeginPosition ≠ f.begin
 8     then
 9           trees.delete(0);
10           trees.addToHead(split(firstTree, f.begin, firstTree.end));
11  Tree lastTree = trees.item(trees.length − 1);
12  if lastTree.absoluteEndPosition ≠ f.end
13     then
14           trees.delete(trees.length − 1);
15           trees.add(split(lastTree, lastTree.begin, f.end));
16  tree = concat(trees);
17  return tree;
```

Рис. 9: Псевдокод функции *Search*

5.4 Алгоритм

1. На вход алгоритму подается уже построенная факторизация $factorization$ строки S. По ней вычисляются частоты использования символов в факторизации.

2. Для каждого фактора f_i из факторизации вычисляется его вес $w_i = weight$.

3. Все факторы сортируются в зависимости от веса для получения новой последовательности $f_{j_1},\ f_{j_2},\ \ldots f_{j_k}$.

4. Для каждого фактора в новой последовательности строится соответствующее ему дерево
   $for\ (f : f_{j_1}, f_{j_2}, \ldots f_{j_k})\ \{\ createTree(f);\ \}$

5. В соответствии с первоначальной последовательностью факторов в факторизации происходит объединение всех факторов в одно дерево $concat(treesMap(factorization))$.

Пример 4.
Рассмотрим пример работы данного алгоритма для строки "aaaababaabaaaabab". LZ-факторизацией этой строки является следующее представление:

$$f_1 = a,\ f_2 = a,\ f_3 = aa,\ f_4 = b,\ f_5 = ab,\ f_6 = aaba,\ f_7 = aaabab$$

Для каждого фактора f_i его вес равен:

$$weight(f_1) = 3,\ weight(f_2) = 2,\ weight(f_3) = 2.5,\ weight(f_4) = 4,\ weight(f_5) = 1.5,\ weight(f_6) = 0,\ weight(f_7) = 0$$

Тогда факторы будут обрабатываться в следующем порядке: $f_4,\ f_1,\ f_3,\ f_2,\ f_5,\ f_6,\ f_7$.

Дерево вывода грамматики, построенное для данной строки, изображено на рисунке 10 (здесь так же приводится вариант без перебалансировок, чтобы можно было видеть различия при сравнении с другими алгоритмами).

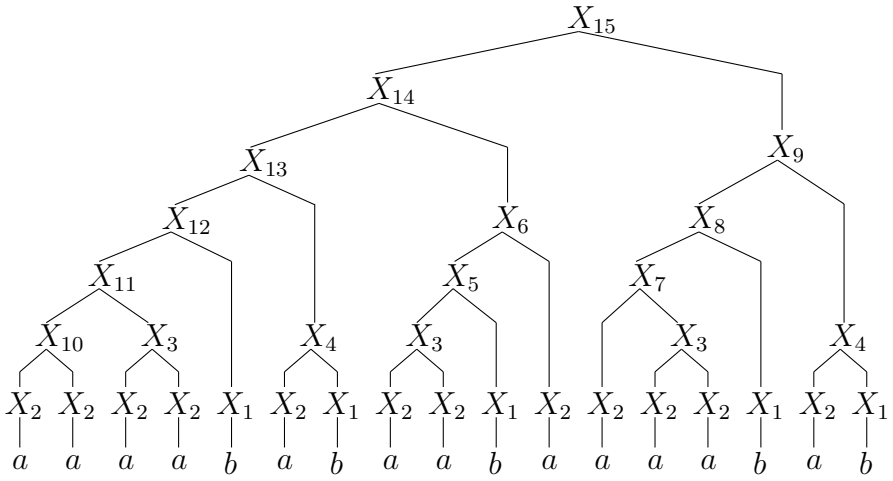

Рис. 10: ПП, выводящая строку “*aaaababaabaaaabab*”

5.5 Анализ предложенного алгоритма

Теорема 4.
Время работы данного алгоритма $max\{O(n),\ O(k^3 \log n)\}$*, где* k *- размер* LZ*-факторизации исходной строки* S*,* n *- длина строки* S*.*

Доказательство: Оценим каждый шаг работы предложенного алгоритма.

1. Время выполнения этого шага алгоритма оценивается $O(n)$, при реализации его следующим образом: рассматривать каждый фактор f_i из факторизации и для каждого символа из подстроки, соответствующей этому фактору, увеличивать счетчик использования этого символа.

2. Данный шаг так же оценивается $O(n)$ для каждого фактора вычислять вес по имеющимся частотам.

3. Здесь предполагается сортировка факторов в зависимости от веса, которую можно выполнить за время $O(k \log k)$ в среднем.

4. Исходя из **Леммы 2** данный шаг осуществим за время $O(k^3 \log n)$.

5. Объединение всех факторов в одно дерево в исходной последовательности может быть реализовано за время $O(k \log n)$.

Таким образом, время работы предложенного алгоритма $max\{O(n),\ O(k^3 \log n)\}$. □

Теорема 5.
Для AVL*-сбалансированной грамматики* G*, соответствующей строке* S*,* $|S| = n$*, высота дерева вывода, соответствующего данной грамматике* G*, равна* $O(log\ n)$ *[5].*

Теорема 6.
Предложенный алгоритм строит грамматику, высота дерева вывода которой равна $O(log\ n)$*, где* n *- длина строки* S*.*

Доказательство: Так как в качестве грамматики использовалась AVL-сбалансированная грамматика, то по Теореме 5, высота дерева вывода такой грамматики равна $O(log\ n)$. □

Теорема 7.
Размер памяти, необходимой для работы данного алгоритма, равен $O(n)$*, где* S *- исходная строка,* n *- длина строки* S*.*

Доказательство: Рассмотрим структуры, используемые в данном алгоритме.

- *association* позволяет получить декомпозицию заданного фактора. В качестве ее реализации можно использовать массив, в котором индекс соответствует номеру

символа в исходной строке, а значением является указатель на фактор из LZ-факторизации, которому соответствует подстрока, которой принадлежит символ с данным индексом. Для хранения данной структуры нам необходимо $O(n)$ памяти.

- $treesMap$ хранит для каждого фактора соответствующее ему дерево. Для хранения данной структуры нам необходимо $O(k)$ памяти, где k - размер LZ-факторизации.

- Для вычисления частот символов на первом шаге алгоритма необходимо для каждого символа хранить счетчик, соответствующий использованию данного символа. Это можно реализовать используя $O(n)$ памяти.

Таким образом, для работы данного алгоритма необходимо $O(n)$ памяти. □

Можно заметить преимущество алгоритма Риттера перед предложенным алгоритмом. Для этого рассмотрим два фактора f_i и f_j. Пусть f_i можно представить в виде $f_i = f_{i_1} \cdot f_{i_2} \cdot f_{i_3} \cdot \ldots \cdot f_{i_k}$, а f_j в виде $f_j = f_{j_1} \cdot f_{i_2} \cdot f_{i_3} \cdot \ldots \cdot f_{j_m}$. При построении дерева для фактора f_i будет построено дерево для $f_{i_2} \cdot f_{i_3}$, в то же время при построении фактора f_j дерево для $f_{i_2} \cdot f_{i_3}$ будет построено вновь, так как информации о том, что деревья факторов f_{i_2} и f_{i_3} были объединены. Это повторное построение дерева может замедлить как скорость работы алгоритма, так и увеличить количество правил строящейся грамматики. В противоположность этому работает алгоритм Риттера, так как в нем дерево строится по ходу и информация о том, объединялись ли данные деревья хранится непосредственно в самом дереве, то такой ситуации не возникает.

Для исправления этой ситуации рассмотрим следующее изменение.

5.6 Эвристическое улучшение алгоритма

Для того, чтобы избежать повторного объединения существующих деревьев, добавим метод $Refresh$, который будет осуществлять обновление структуры $association$. Псевдокод этого метода приведен на рисунке 11.

```
REFRESH(f, beginRefreshPosition)
1  for i = beginRefreshPosition to beginRefreshPosition + f.length
2        do
3            association(i) = f;
```

Рис. 11: Псевдокод функции $Refresh$

Параметрами данной функции являются фактор f и целое положительное число $beginRefreshPosition$, означающее, что с данной позиции и до $beginRefreshPosition + f.length$ $association$ будет возвращать фактор f.

Помимо этого, изменится функция $Search$. Изменения продемонстрированы на рисунке 12.

Как и в первоначальном варианте, на вход данной функции передается фактор f, для которого следует построить дерево. Для него с помощью $association$ находится декомпозиция, после чего осуществляется создание фактора, который будет передаваться на вход функции $Refresh$. Он формируется исходя из наибольшего дерева, которое может быть построено из следующий деревьев: "среднего" ($middleTree$) - это дерево является объединением всех деревьев, за исключением первого ($leftTree$) и последнего ($rightTree$). Далее рассматриваются 4 возможных случая размещения фактора относительно его декомпозиции:

1. Разница между левыми границами первого фактора в декомпозиции и фактора f равна 0 и разница между правыми границами последнего фактора и фактора f также равна 0. В этом случае все деревья (первое, "среднее" и последнее) объединяются в одно, которое будет результатом работы процедуры, а так же создается новый фактор ($fakeFactor$), начало которого соответствует $f.beginPosition$, длина соответствует ширине получившегося дерева, а $absoluteBeginPosition$ - это $absoluteBeginPosition$ первого фактора декомпозиции, после чего вызывается функция $Refresh(fakeFactor)$.

2. Разница между левыми границами первого фактора в декомпозиции и фактора f равна 0, при этом разница между правыми границами последнего фактора и фактора f отлична от 0. В этом случае сначала объединяются первое и "среднее" деревья, происходит создание нового фактора с параметрами: его начало соответствует $f.beginPosition$, длина соответствует ширине получившегося дерева, а

```
SEARCH(f)
 1  factors = association(f);
 2  trees = {};
 3  for Factorfactor : factors
 4        do
 5           trees.add(createTree(factor));
 6  Tree leftTree = trees.remove(0),
    middleTree = emptyTree, rightTree = trees.remove(trees.length − 1);
 7  middleTree = concat(trees);
 8  if leftTree.absoluteBeginPosition == f.begin
 9     then
10           tree = concat(leftTree, middleTree);
11           fakeFactor =
    new Factor(f.beginPosition, tree.width, factors.item(0).absoluteBeginPosition);
12           if rightTree.absoluteEndPosition == f.end
13              then
14                    tree = concat(tree, rightTree);
15                    treesMap(fakeFactor) = tree;
16                    refresh(fakeFactor, factors.item(0).absoluteBeginPosition);
17              else
18                    treesMap(fakeFactor) = tree;
19                    refresh(fakeFactor, factors.item(0).absoluteBeginPosition);
20                    tree = concat(tree, split(rightTree, rightTree.begin, f.end));
21     else
22           tree = middleTree;
23           fakeFactor =
    new Factor(f.beginPosition, tree.width, factors.item(1).absoluteBeginPosition);
24           if rightTree.absoluteEndPosition == f.end
25              then
26                    tree = concat(tree, rightTree);
27                    treesMap(fakeFactor) = tree;
28                    refresh(fakeFactor, factors.item(1).absoluteBeginPosition);
29                    tree = concat(split(leftTree, f.begin, leftTree.end), tree);
30              else
31                    treesMap(fakeFactor) = tree;
32                    refresh(fakeFactor, factors.item(1).absoluteBeginPosition);
33                    tree = concat(split(leftTree, f.begin, leftTree.end),
    tree, split(rightTree, rightTree.begin, f.end));
34  return tree;
```

Рис. 12: Псевдокод функции *Search*

$absoluteBeginPosition$ - это $absoluteBeginPosition$ первого фактора декомпозиции. После чего вызывается функция $Refresh(fakeFactor)$. Результатом работы функции является объединение полученного ранее дерева с последовательностью деревьев, соответствующих поддеревьям, получаемых из последнего дерева от его начала и до конца фактора f.

3. Разница между левыми границами первого фактора в декомпозиции и фактора f отлична от 0 и разница между правыми границами последнего фактора и фактора f равна 0. В этом случае сначала объединяются "среднее" и последнее деревья и происходит создание нового фактора с параметрами: начало соответствует $f.beginPosition$, длина соответствует ширине получившегося дерева, а $absoluteBeginPosition$ - это $absoluteBeginPosition$ второго фактора декомпозиции. После чего вызывается функция $Refresh(fakeFactor)$. Результатом работы функции является объединение последовательности деревьев, соответствующих поддеревьям первого фактора с позиции, соответствующей $f.beginPosition$, и получившегося дерева.

4. Разница между левыми границами первого фактора в декомпозиции и фактора f отлична от 0, при этом разница между правыми границами последнего фактора и фактора f отлична от 0. В этом случае просходит создание нового фактора с параметрами: начало соответствует $f.beginPosition$, длина соответствует ширине среднего дерева, а $absoluteBeginPosition$ - это $absoluteBeginPosition$ второго фактора декомпозиции. После чего происходит вызов функции $Refresh(fakeFactor)$. Результатом работы данной функции является объединение последовательности деревьев, соответствующих поддеревьям первого фактора с позиции, соответствующей $f.beginPosition$ со "средним" деревом и последовательностью деревьев, соответствующих поддеревьям, получаемых из последнего дерева от его начала и до конца фактора f.

Каждое из созданных в ходе данного алгоритма деревьев так же сохраняются в $treesMap$.

Это изменение помогает избежать недостатка, возникающего ранее, а именно повторного объединения деревьев, так как когда будет происходит обращение к $association$ далее, на месте факторов, чьи деревья уже объединялись, будет стоять недавно созданный фактор, которому соответствует дерево с ранее объединенными деревьями.

Пример 5.
Рассмотрим пример работы данного алгоритма для строки "aaaababaabaaaabab". LZ-факторизацией этой строки является следующее представление:

$$f_1 = a,\ f_2 = a,\ f_3 = aa,\ f_4 = b,\ f_5 = ab,\ f_6 = aaba,\ f_7 = aaabab$$

Для каждого фактора f_i его вес равен:

$$weight(f_1) = 3,\ weight(f_2) = 2,\ weight(f_3) = 2.5,\ weight(f_4) = 4,\ weight(f_5) = 1.5,\ weight(f_6) = 0,\ weight(f_7) = 0$$

Тогда факторы будут обрабатываться в следующем порядке: $f_4,\ f_1,\ f_3,\ f_2,\ f_5,\ f_6,\ f_7$.

Дерево вывода грамматики, построенное для данной строки, изображено на рисунке 13 (здесь так же приводится вариант без перебалансировок, чтобы можно было видеть различия при сравнении с другими алгоритмами).

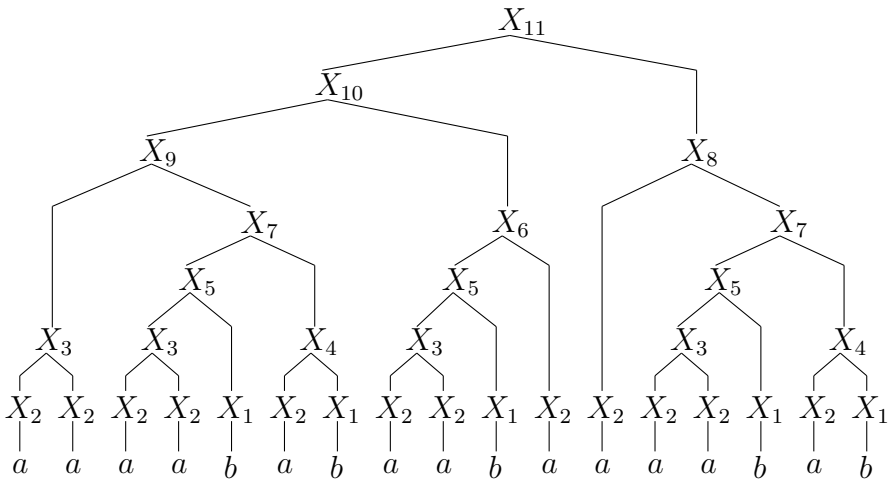

Рис. 13: ПП, выводящая строку “*aaaababaabaaaabab*”

Так как приведенные выше изменения не никак не влияют на LZ-факторизацию исходной строки и не увеличивают количество факторов в декомпозиции для некоторого фактора, оценки алгоритма, доказанные выше, остаются верными и в данном случае.

5.7 Практические результаты

На рисунках 14, 15 и 16 представлены графики работы данного алгоритма на ДНК-последовательностях.

График 14 отображает количество вращений AVL-деревьев при построении грамматики. По оси OX расположена информация о размере сжимаемого файла в МБ, а по оси OY - размер получающейся грамматики в миллионах. Как видно по данному графику, по сравнению с классическим алгоритмом Риттера, количество вращений как для ленивого алгоритма, так и для его эвристического улучшения, значительно меньше. Не смотря на это, он уступает модифицированному алгоритму Риттера [2].

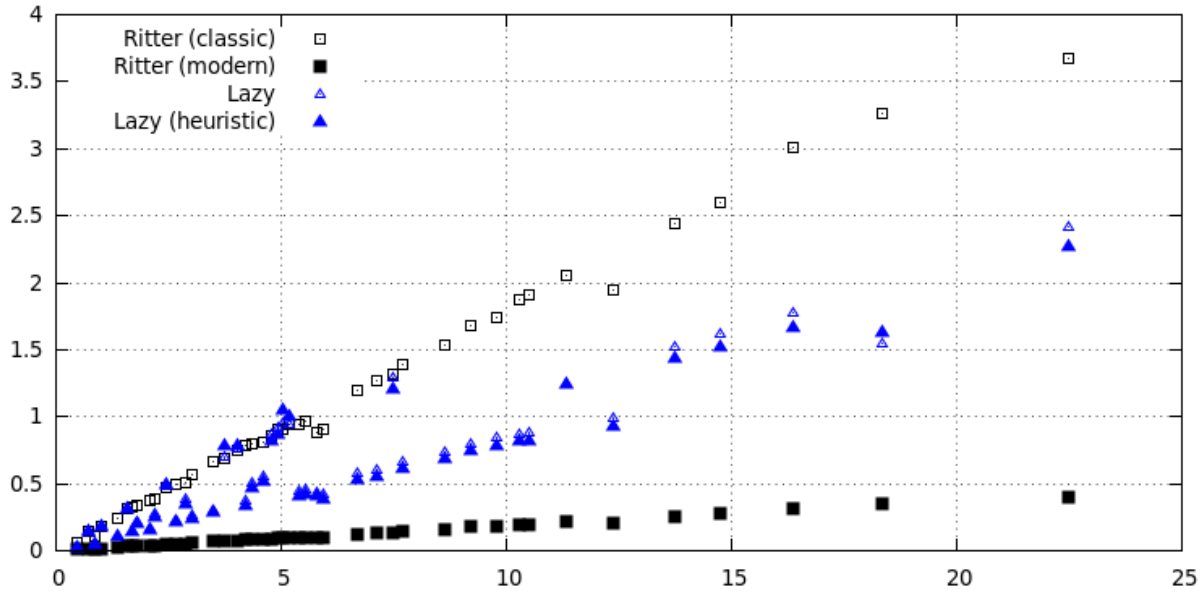

Рис. 14: Вращения для ДНК

На графике 15 по оси OX расположена информация о размере сжимаемого файла в МБ, а по оси OY указано отношение сжатого представления к размеру исходного файла в процентах. Чем меньше это отношение, тем лучше сжатие. Размер грамматики, получаемой с помощью ленивых алгоритмов хуже, чем у алгоритма Риттера.

График 16 отображает время работы алгоритмов. По оси OX расположена информация о размере сжимаемого файла в МБ, а по оси OY указано время построения ПП в минутах. По нему можно судить, что скорость работы ленивого алгоритма почти совпадает со скоростью работы его эвристического улучшения и лучше, чем скорость работы классического алгоритма Риттера. Это позволяет судить о том, что доказанная оценка работы ленивого алгоритма является оценкой худшего случая, так как на практике по полученным данным можно судить, что существует более лучшая средняя оценка.

Полученные практические результаты позволяют судить о том, что данный алгоритм по некоторым параметрам лучше классического алгоритма Риттера. В частности можно заметить, что предложенная эвристика работает не хуже, чем сам алгоритм, а

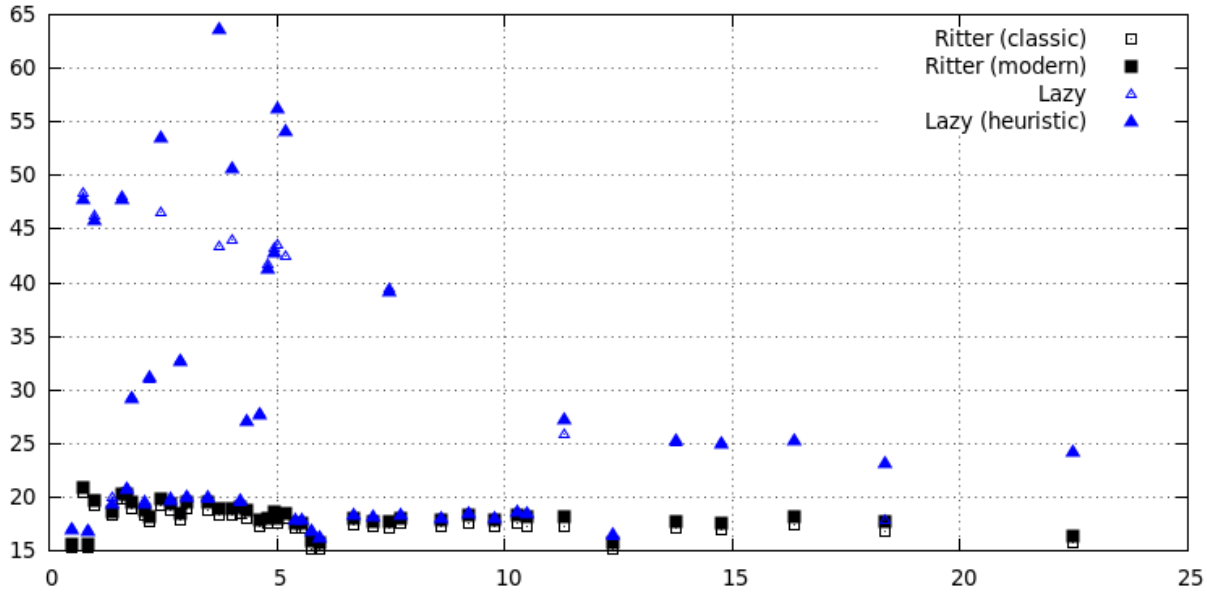

Рис. 15: Степень сжатия для ДНК

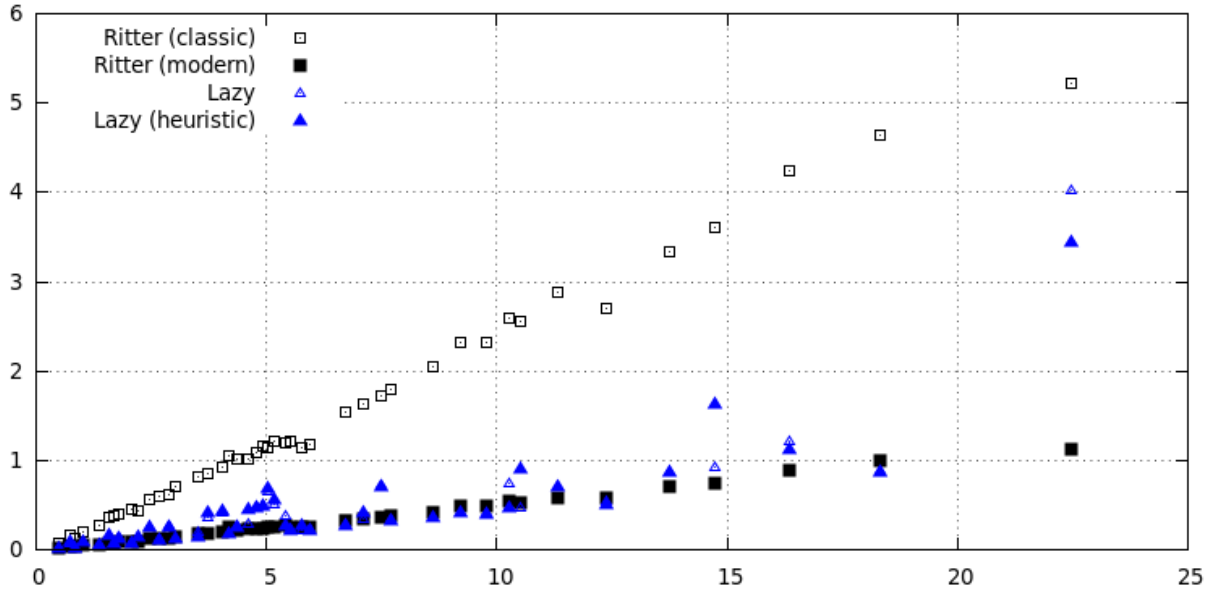

Рис. 16: Время построения для ДНК на файловой системе

значит возможно дальнейшее развитие алгоритма с помощью улучшения этой эвристики.

6 Выводы

В данной работе предложен алгоритм построения прямолинейных программ. Было предложено эвристическое улучшение данного алгоритма. Была доказана оценка его работы в худшем случае. Был проведен практический сравнительный анализ с алгоритмом Риттера.

Не смотря на то, что худшая оценка этого алгоритма $max\{O(k^3 \log n), O(n)\}$, на практике этот алгоритм показал себя лучше, чем классический алгоритм Риттера.

Предложенный в данной работе алгоритм можно рассматривать как альтернативный вариант алгоритма Риттера. По полученным практическим результатам видно, что размер грамматики, построенный с помощью него, уступает размеру грамматики, которая получается при использовании алгоритма Риттера. Однако характерной особенностью здесь является подход к построению грамматики. В алгоритме Риттера дерево грамматики строится за один проход и на каждом шаге построение некоторого фактора напрямую зависит от уже построенного дерева. В то время как алгоритм, предложенный в этой работе строит факторы таким образом, что не возникает сильной зависимости от уже построенных факторов, так как каждый фактор строится “по необходимости”, то есть лениво. Этот подход позволяет распараллелить данный алгоритм. В данном случае участком распараллеливания будет является этап построения деревьев для факторов, так как их построение слабо связано между собой. Применение этого подхода позволит увеличить скорость работы алгоритма. В противоположность этому, алгоритм Риттера распараллелить невозможно.

Список литературы

[1] E. Lehman, A. Shelat, Approximation Algorithms for Grammar-Based Compression, In Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (2002), 205-212.

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

[3] 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.

[4] S. Maruyama, M. Takeda, M. Nakahara, H. Sakamoto, An Online Algorithm for Lightweight Grammar-Based Compression, First International Conference on Data Compression, Communications and Processing (2011).

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

РЕФЕРАТ

Козлова А. В. Ленивое построение прямолинейных программ квалификационная работа на степень бакалавра наук: 28 страниц, 16 рисунков, 5 источников.

Ключевые слова: прямолинейная программа, грамматическое сжатие, ленивое построение.

Целью проделанной работы была разработка нового алгоритма построения прямолинейных программ. В результате был разработан алгоритм, доказана оценка его работы в худшем случае, предложена эвристика по улучшению этого алгоритма. Так же было проведено сравнение этого алгоритма с алгоритмом Риттера на практике.