Алгоритмические свойства сжатых текстов

Анна Козлова, Евгений Курпилянский, Алексей Хворост

28 ноября 2011 года

С ростом размера входных данных для классических задач меняются алгоритмы, способные их эффективно решать.

Существует несколько подходов, например:

- Алгоритмы эффективного ввода-вывода – алгоритмы, минимизирующие чтение данных с жесткого диска.
- Сокращение размера входа за счет предварительной обработки.

С ростом размера входных данных для классических задач меняются алгоритмы, способные их эффективно решать.

Существует несколько подходов, например:

- Алгоритмы эффективного ввода-вывода – алгоритмы, минимизирующие чтение данных с жесткого диска.
- Сокращение размера входа за счет предварительной обработки.

С ростом размера входных данных для классических задач меняются алгоритмы, способные их эффективно решать.

Существует несколько подходов, например:

- Алгоритмы эффективного ввода-вывода – алгоритмы, минимизирующие чтение данных с жесткого диска.
- Сокращение размера входа за счет предварительной обработки.

Определение прямолинейной программы

Определение

Прямолинейная программа (ПП) $\mathcal{X}$ размера n – это последовательность правил вывода

$$\mathcal{X}_1 = expr_1, \mathcal{X}_2 = expr_2, \ldots, \mathcal{X}_n = expr_n,$$

где $\mathcal{X}_i$ – это **переменные**, а $expr_i$ – это **выражения** вида:

- $expr_i$ – символ из алфавита Σ (**терминальные** правила).
- $expr_i = \mathcal{X}_l \cdot \mathcal{X}_r$ $(l, r < i)$ (**нетерминальные** правила).

Прямолинейная программа – это грамматика, выводящая в точности одно слово.

Определение прямолинейной программы

Определение

Прямолинейная программа (ПП) $\mathcal{X}$ размера n – это последовательность правил вывода

$$\mathcal{X}_1 = expr_1, \mathcal{X}_2 = expr_2, \ldots, \mathcal{X}_n = expr_n,$$

где $\mathcal{X}_i$ – это **переменные**, а $expr_i$ – это **выражения** вида:

- $expr_i$ – символ из алфавита Σ (**терминальные** правила).
- $expr_i = \mathcal{X}_l \cdot \mathcal{X}_r$ $(l, r < i)$ (**нетерминальные** правила).

Прямолинейная программа – это грамматика, выводящая в точности одно слово.

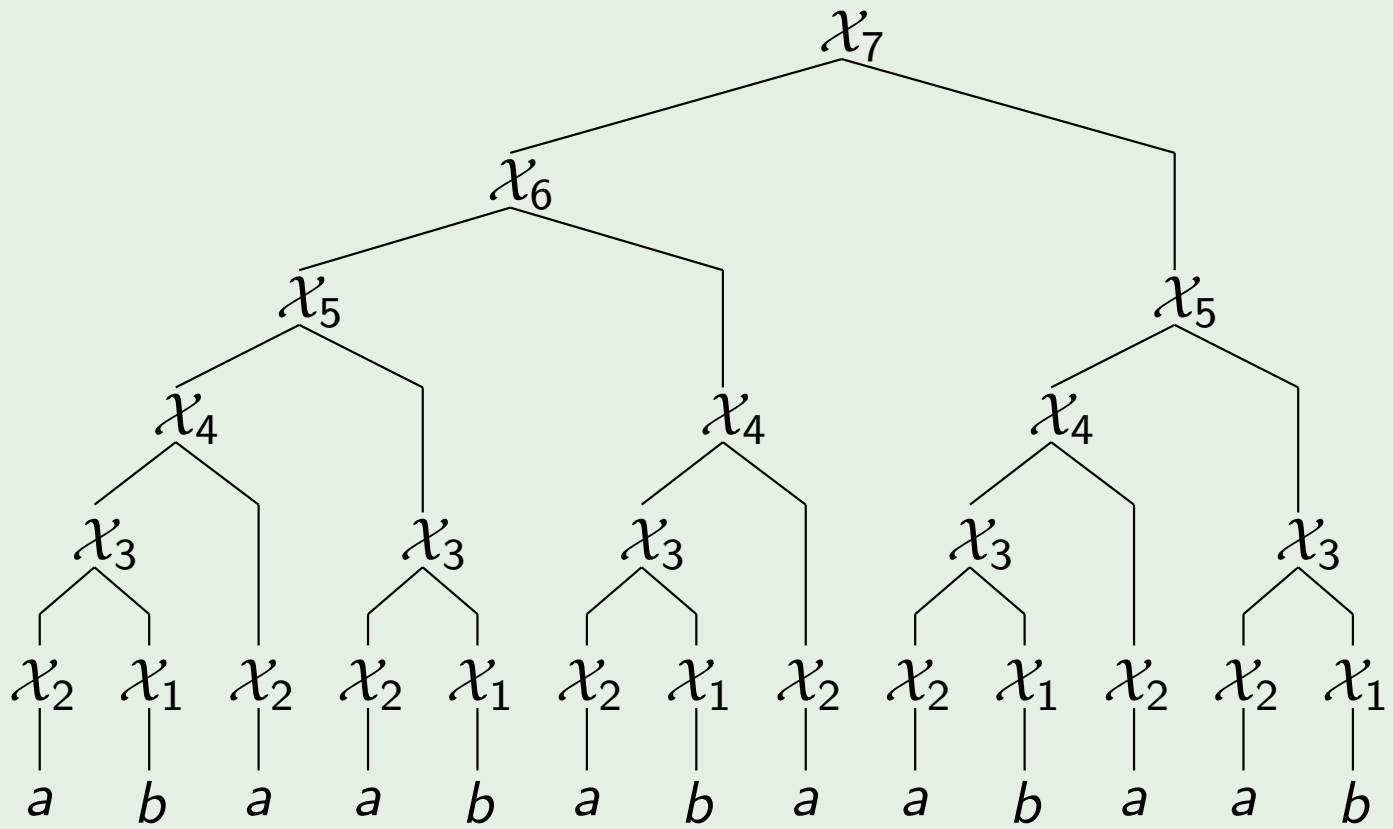
Пример

Рассмотрим ПП $\mathcal{X}$, выводящую строку $X = «abaababaabaab»$.

$$\begin{aligned}
\mathcal{X}_1 &= b \\
\mathcal{X}_2 &= a \\
\mathcal{X}_3 &= \mathcal{X}_2 \cdot \mathcal{X}_1 \\
\mathcal{X}_4 &= \mathcal{X}_3 \cdot \mathcal{X}_2 \\
\mathcal{X}_5 &= \mathcal{X}_4 \cdot \mathcal{X}_3 \\
\mathcal{X}_6 &= \mathcal{X}_5 \cdot \mathcal{X}_4 \\
\mathcal{X}_7 &= \mathcal{X}_6 \cdot \mathcal{X}_5
\end{aligned}$$

Пример

Графическое изображение ПП:



ПП – это способ сжатия данных

Длина строки, выводимой из ПП, может быть экспоненциальной относительно размеров ПП.

Недостатки

Сжатие с помощью ПП значительно уступает стандартным алгоритмам сжатия.

Преимущества

Есть возможность осуществлять различные алгоритмы над ПП, не распаковывая данные.

ПП – это способ сжатия данных

Длина строки, выводимой из ПП, может быть экспоненциальной относительно размеров ПП.

Недостатки

Сжатие с помощью ПП значительно уступает стандартным алгоритмам сжатия.

Преимущества

Есть возможность осуществлять различные алгоритмы над ПП, не распаковывая данные.

ПП – это способ сжатия данных

Длина строки, выводимой из ПП, может быть экспоненциальной относительно размеров ПП.

Недостатки

Сжатие с помощью ПП значительно уступает стандартным алгоритмам сжатия.

Преимущества

Есть возможность осуществлять различные алгоритмы над ПП, не распаковывая данные.

Алгоритмы над ПП

Существует ряд классических задач, сформулированных в терминах ПП и разрешимых за полиномиальное время от размера ПП, например:

- Поиск сжатого образца в сжатом тексте
- Поиск наибольшей общей подстроки двух сжатых строк
- Поиск палиндромов в сжатой строке
- Поиск квадратов в сжатой строке

Алгоритмы над ПП

Существует ряд классических задач, сформулированных в терминах ПП и разрешимых за полиномиальное время от размера ПП, например:

- Поиск сжатого образца в сжатом тексте
- Поиск наибольшей общей подстроки двух сжатых строк
- Поиск палиндромов в сжатой строке
- Поиск квадратов в сжатой строке

Главный вопрос

Осмысленно ли использование прямолинейных программ на практике?

- Насколько сложна задача построения ПП?
- Насколько сжатие с использованием ПП уступает классическим алгоритмам сжатия?

Главный вопрос

Осмысленно ли использование прямолинейных программ на практике?

- Насколько сложна задача построения ПП?
- Насколько сжатие с использованием ПП уступает классическим алгоритмам сжатия?

Главный вопрос

Осмысленно ли использование прямолинейных программ на практике?

- Насколько сложна задача построения ПП?
- Насколько сжатие с использованием ПП уступает классическим алгоритмам сжатия?

Как строить ПП?

Утверждение. W.Rytter et al. 2003

Задача построения минимальной ПП, выводящей заданную строку T – NP-полная.

Для построения ПП требуется использовать приближенные алгоритмы.

Как строить ПП?

Утверждение. W.Rytter et al. 2003

Задача построения минимальной ПП, выводящей заданную строку T – NP-полная.

$\Downarrow$

Для построения ПП требуется использовать приближенные алгоритмы.

Идея

Для построения ПП в качестве отправной точки использовать классические алгоритмы сжатия.

Оказалось, что классический **алгоритм Лемпеля-Зива** хорошо подходит для этой задачи.

Идея

Для построения ПП в качестве отправной точки использовать классические алгоритмы сжатия.

Оказалось, что классический **алгоритм Лемпеля-Зива** хорошо подходит для этой задачи.

Определение

Факторизация строки T – это набор строк $F_1, F_2, \ldots, F_k$ такой, что $T = F_1 \cdot F_2 \cdot \ldots \cdot F_k$.

Определение

***LZ*-факторизация** строки T – это набор строк $F_1, F_2, \ldots, F_k$, где

- $F_1 = T[0]$;
- F_i равен наибольшему префиксу $T[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |T| - 1]$, который встречался в тексте ранее **или** $T[|F_1 \cdot \ldots \cdot F_{i-1}|]$ в случае, если этот префикс пустой.

Факторизации строки «*abaababaabaab*»

- $a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot a \cdot b$;
- $a \cdot b \cdot a \cdot aba \cdot baaba \cdot ab$;

Определение

Факторизация строки T – это набор строк $F_1, F_2, \ldots, F_k$ такой, что $T = F_1 \cdot F_2 \cdot \ldots \cdot F_k$.

Определение

***LZ*-факторизация** строки T – это набор строк $F_1, F_2, \ldots, F_k$, где

- $F_1 = T[0]$;
- F_i равен наибольшему префиксу $T[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |T| - 1]$, который встречался в тексте ранее **или** $T[|F_1 \cdot \ldots \cdot F_{i-1}|]$ в случае, если этот префикс пустой.

Факторизации строки «*abaababaabaab*»

- $a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot a \cdot b$;
- $a \cdot b \cdot a \cdot aba \cdot baaba \cdot ab$;

Определение

Факторизация строки T – это набор строк $F_1, F_2, \ldots, F_k$ такой, что $T = F_1 \cdot F_2 \cdot \ldots \cdot F_k$.

Определение

***LZ*-факторизация** строки T – это набор строк $F_1, F_2, \ldots, F_k$, где

- $F_1 = T[0]$;
- F_i равен наибольшему префиксу $T[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |T| - 1]$, который встречался в тексте ранее **или** $T[|F_1 \cdot \ldots \cdot F_{i-1}|]$ в случае, если этот префикс пустой.

Факторизации строки «*abaababaabaab*»

- $a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot b \cdot a \cdot a \cdot b \cdot a \cdot a \cdot b$;
- $a \cdot b \cdot a \cdot aba \cdot baaba \cdot ab$;

LZ-факторизацию строки T длины n можно построить как минимум двумя способами:

- За $O(n)$ с использованием суффиксного дерева. Тяжело адаптируется на случай, когда заканчивается оперативная память.
- За $O(n \log n)$ с использованием суффиксного массива. Проще в реализации, легко адаптируется на использование внешней памяти.

LZ-факторизацию строки T длины n можно построить как минимум двумя способами:

- За $O(n)$ с использованием **суффиксного дерева**. Тяжело адаптируется на случай, когда заканчивается оперативная память.
- За $O(n \log n)$ с использованием **суффиксного массива**. Проще в реализации, легко адаптируется на использование внешней памяти.

LZ-факторизацию строки T длины n можно построить как минимум двумя способами:

- За $O(n)$ с использованием **суффиксного дерева**. Тяжело адаптируется на случай, когда заканчивается оперативная память.
- За $O(n \log n)$ с использованием **суффиксного массива**. Проще в реализации, легко адаптируется на использование внешней памяти.

LZ77-факторизация

Определение

***LZ77*-факторизация** строки T – это набор строк $F_1, F_2, \ldots, F_k$, где

- $F_1 = T[0]$;
- F_i равен наибольшему префиксу $T[|F_1 \cdot \ldots \cdot F_{i-1}| \ldots |T| - 1]$, который встречался **в суффиксе** уже просмотренного текста **или** $T[|F_1 \cdot \ldots \cdot F_{i-1}|]$ в случае, если этот префикс пустой.

Недостатки

LZ77-факторизация содержит больше факторов, чем *LZ*-факторизация.

Преимущества

Алгоритм построения факторизации может работать в оперативной памяти.

Недостатки

LZ77-факторизация содержит больше факторов, чем *LZ*-факторизация.

Преимущества

Алгоритм построения факторизации может работать в оперативной памяти.

Постановка задачи

Вход: Строка T и ее факторизация $F_1, F_2, \ldots, F_k$.
Выход: ПП, выводящая строку T.

Алгоритм Риттера

Описание алгоритма

- Дерево разбора ПП представляется в виде *AVL*-дерева.
- Факторы обрабатываются последовательно.
- Необходимо уметь реализовывать операцию конкатенации двух ПП.
- Для эффективности реализации структура данных должна обладать свойством *confluently persistent*.
- Все вершины дерева должны быть неизменяемыми. Если требуется изменить вершину, создаем и изменяем копию этой вершины.
- Появляются много неиспользуемых вершин. $\Rightarrow$ Необходим менеджер памяти, освобождающий ресурсы.

Алгоритм Риттера

Описание алгоритма

- Дерево разбора ПП представляется в виде *AVL*-дерева.
- Факторы обрабатываются последовательно.
- Необходимо уметь реализовывать операцию конкатенации двух ПП.
- Для эффективности реализации структура данных должна обладать свойством *confluently persistent*.
- Все вершины дерева должны быть неизменяемыми. Если требуется изменить вершину, создаем и изменяем копию этой вершины.
- Появляются много неиспользуемых вершин. $\Rightarrow$ Необходим менеджер памяти, освобождающий ресурсы.

Алгоритм Риттера

Описание алгоритма

- Дерево разбора ПП представляется в виде *AVL*-дерева.
- Факторы обрабатываются последовательно.
- Необходимо уметь реализовывать операцию конкатенации двух ПП.

- Для эффективности реализации структура данных должна обладать свойством *confluently persistent*.
- Все вершины дерева должны быть неизменяемыми. Если требуется изменить вершину, создаем и изменяем копию этой вершины.
- Появляются много неиспользуемых вершин. $\Rightarrow$ Необходим менеджер памяти, освобождающий ресурсы.

Алгоритм Риттера

Описание алгоритма

- Дерево разбора ПП представляется в виде *AVL*-дерева.
- Факторы обрабатываются последовательно.
- Необходимо уметь реализовывать операцию конкатенации двух ПП.

- Для эффективности реализации структура данных должна обладать свойством *confluently persistent*.
- Все вершины дерева должны быть неизменяемыми. Если требуется изменить вершину, создаем и изменяем копию этой вершины.
- Появляются много неиспользуемых вершин. $\Rightarrow$ Необходим менеджер памяти, освобождающий ресурсы.

Алгоритм Риттера

Описание алгоритма

- Дерево разбора ПП представляется в виде *AVL*-дерева.
- Факторы обрабатываются последовательно.
- Необходимо уметь реализовывать операцию конкатенации двух ПП.

- Для эффективности реализации структура данных должна обладать свойством *confluently persistent*.
- Все вершины дерева должны быть неизменяемыми. Если требуется изменить вершину, создаем и изменяем копию этой вершины.
- Появляются много неиспользуемых вершин. $\Rightarrow$ Необходим менеджер памяти, освобождающий ресурсы.

Алгоритм Риттера

Описание алгоритма

- Дерево разбора ПП представляется в виде *AVL*-дерева.
- Факторы обрабатываются последовательно.
- Необходимо уметь реализовывать операцию конкатенации двух ПП.

- Для эффективности реализации структура данных должна обладать свойством *confluently persistent*.
- Все вершины дерева должны быть неизменяемыми. Если требуется изменить вершину, создаем и изменяем копию этой вершины.
- Появляются много неиспользуемых вершин. $\Rightarrow$ Необходим менеджер памяти, освобождающий ресурсы.

Теорема. W.Rytter et al. 2003

Пусть дан текст T и его факторизация размера k. Тогда **алгоритм Риттера** позволяет построить ПП $\mathcal{T}$ размером $O(k \cdot \log |T|)$ за время $O(k \cdot \log |T|)$.

Преимущества

- Размер полученной ПП всего в $O(\log n)$ больше размера минимальной ПП
- *AVL*-дерево «сильносбалансированное» дерево, его высота ограничена сверху $\frac{1+\sqrt{5}}{2} \log n$.

Недостатки

- Операция перебалансировки узла является достаточно дорогой операцией, обновляется три вершины.

Преимущества

- Размер полученной ПП всего в $O(\log n)$ больше размера минимальной ПП
- AVL-дерево «сильносбалансированное» дерево, его высота ограничена сверху $\frac{1+\sqrt{5}}{2} \log n$.

Недостатки

- Операция перебалансировки узла является достаточно дорогой операцией, обновляется три вершины.

Преимущества

- Размер полученной ПП всего в $O(\log n)$ больше размера минимальной ПП
- AVL-дерево «сильносбалансированное» дерево, его высота ограничена сверху $\frac{1+\sqrt{5}}{2} \log n$.

Недостатки

- Операция перебалансировки узла является достаточно дорогой операцией, обновляется три вершины.

Модернизация алгоритма Риттера

Проблема

На каждой итерации алгоритм выполняется конкатенацию потенциально огромного дерева с маленьким.

Основная идея

Можно соптимизировать порядок конкатенаций, уменьшив количество поворотов дерева.

Решение

Найти оптимальный порядок конкатенаций k деревьев можно с помощью динамического программирования за $O(k^3)$.

Модернизация алгоритма Риттера

Проблема

На каждой итерации алгоритм выполняется конкатенацию потенциально огромного дерева с маленьким.

Основная идея

Можно соптимизировать порядок конкатенаций, уменьшив количество поворотов дерева.

Решение

Найти оптимальный порядок конкатенаций k деревьев можно с помощью динамического программирования за $O(k^3)$.

Модернизация алгоритма Риттера

Проблема

На каждой итерации алгоритм выполняется конкатенацию потенциально огромного дерева с маленьким.

Основная идея

Можно соптимизировать порядок конкатенаций, уменьшив количество поворотов дерева.

Решение

Найти оптимальный порядок конкатенаций k деревьев можно с помощью динамического программирования за $O(k^3)$.

Построение ПП с помощью декартовых деревьев

Мысль

Почему бы для построения ПП не использовать другое сбалансированное двоичное дерево?

- Декартово дерево – двоичное дерево, в каждой вершине которого хранится ее приоритет. При этом всегда выполняется свойство: приоритет в потомках меньше, чем приоритет самой вершины.
- Доказано, что если приоритеты выбираются случайно, высота декартова дерева $O(\log n)$.
- Над декартовым деревом определены две стандартные операции: конкатенация и разрезание.

Построение ПП с помощью декартовых деревьев

Мысль

Почему бы для построения ПП не использовать другое сбалансированное двоичное дерево?

- Декартово дерево – двоичное дерево, в каждой вершине которого хранится ее приоритет. При этом всегда выполняется свойство: приоритет в потомках меньше, чем приоритет самой вершины.
- Доказано, что если приоритеты выбираются случайно, высота декартова дерева $O(\log n)$.
- Над декартовым деревом определены две стандартные операции: конкатенация и разрезание.

Построение ПП с помощью декартовых деревьев

Мысль

Почему бы для построения ПП не использовать другое сбалансированное двоичное дерево?

- Декартово дерево – двоичное дерево, в каждой вершине которого хранится ее приоритет. При этом всегда выполняется свойство: приоритет в потомках меньше, чем приоритет самой вершины.
- Доказано, что если приоритеты выбираются случайно, высота декартова дерева $O(\log n)$.
- Над декартовым деревом определены две стандартные операции: конкатенация и разрезание.

Построение ПП с помощью декартовых деревьев

Мысль

Почему бы для построения ПП не использовать другое сбалансированное двоичное дерево?

- Декартово дерево – двоичное дерево, в каждой вершине которого хранится ее приоритет. При этом всегда выполняется свойство: приоритет в потомках меньше, чем приоритет самой вершины.
- Доказано, что если приоритеты выбираются случайно, высота декартова дерева $O(\log n)$.
- Над декартовым деревом определены две стандартные операции: конкатенация и разрезание.

Алгоритм

Идея алгоритма совпадает с идеей алгоритма Риттера.

Преимущества

- Простота реализации.
- С помощью двух операций разрезания можно получить подстроку, соответсвующую следующему фактору, одним деревом целиком.

Недостатки

- Высота декартова дерева на практике в два-три раза больше высоты *AVL*-дерева.
- Недетерминированность алгоритма.

Алгоритм

Идея алгоритма совпадает с идеей алгоритма Риттера.

Преимущества

- Простота реализации.
- С помощью двух операций разрезания можно получить подстроку, соответсвующую следующему фактору, одним деревом целиком.

Недостатки

- Высота декартова дерева на практике в два-три раза больше высоты *AVL*-дерева.
- Недетерминированность алгоритма.

Алгоритм

Идея алгоритма совпадает с идеей алгоритма Риттера.

Преимущества

- Простота реализации.
- С помощью двух операций разрезания можно получить подстроку, соответсвующую следующему фактору, одним деревом целиком.

Недостатки

- Высота декартова дерева на практике в два-три раза больше высоты *AVL*-дерева.
- Недетерминированность алгоритма.

Алгоритм

Идея алгоритма совпадает с идеей алгоритма Риттера.

Преимущества

- Простота реализации.
- С помощью двух операций разрезания можно получить подстроку, соответсвующую следующему фактору, одним деревом целиком.

Недостатки

- Высота декартова дерева на практике в два-три раза больше высоты *AVL*-дерева.
- Недетерминированность алгоритма.

Алгоритм

Идея алгоритма совпадает с идеей алгоритма Риттера.

Преимущества

- Простота реализации.
- С помощью двух операций разрезания можно получить подстроку, соответсвующую следующему фактору, одним деревом целиком.

Недостатки

- Высота декартова дерева на практике в два-три раза больше высоты *AVL*-дерева.
- Недетерминированность алгоритма.

Практические результаты

Алгоритмы были протестированы на ДНК, взятых с сайта `http://www.ddbj.nig.ac.jp/`.

Будут приведены результаты тестирования следующих алгоритмов:

- Алгоритм *LZ*-факторизации. Обозначение – lz
- Алгоритм *LZ*77-факторизации. Обозначение – **lz77**
- Алгоритм Риттера. Обозначение – **SLPClassic**
- Модернизированный алгоритм Риттера. Обозначение – **SLPNew**
- Алгоритм построения ПП с помощью декартовых деревьев. Обозначение – **SLPCartesian**

Будут приведены результаты тестирования следующих алгоритмов:

- Алгоритм *LZ*-факторизации. Обозначение – **lz**
- Алгоритм *LZ77*-факторизации. Обозначение – **lz77**
- Алгоритм Риттера. Обозначение – **SLPClassic**
- Модернизированный алгоритм Риттера. Обозначение – **SLPNew**
- Алгоритм построения ПП с помощью декартовых деревьев. Обозначение – **SLPCartesian**

Будут приведены результаты тестирования следующих алгоритмов:

- Алгоритм *LZ*-факторизации. Обозначение – **lz**
- Алгоритм *LZ*77-факторизации. Обозначение – **lz77**
- Алгоритм Риттера. Обозначение – **SLPClassic**
- Модернизированный алгоритм Риттера. Обозначение – **SLPNew**
- Алгоритм построения ПП с помощью декартовых деревьев. Обозначение – **SLPCartesian**

Будут приведены результаты тестирования следующих алгоритмов:

- Алгоритм *LZ*-факторизации. Обозначение – **lz**
- Алгоритм *LZ77*-факторизации. Обозначение – **lz77**
- Алгоритм Риттера. Обозначение – **SLPClassic**
- Модернизированный алгоритм Риттера. Обозначение – **SLPNew**
- Алгоритм построения ПП с помощью декартовых деревьев. Обозначение – **SLPCartesian**

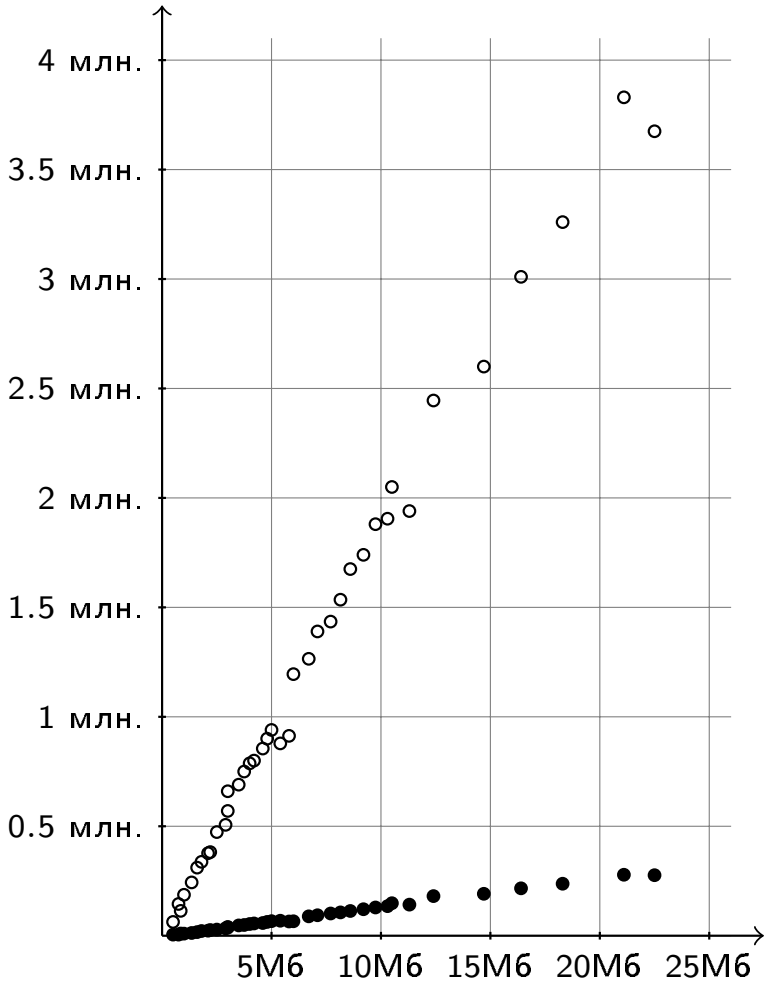
Будут приведены результаты тестирования следующих алгоритмов:

- Алгоритм *LZ*-факторизации. Обозначение – **lz**
- Алгоритм *LZ77*-факторизации. Обозначение – **lz77**
- Алгоритм Риттера. Обозначение – **SLPClassic**
- Модернизированный алгоритм Риттера. Обозначение – **SLPNew**
- Алгоритм построения ПП с помощью декартовых деревьев. Обозначение – **SLPCartesian**

Будут приведены результаты тестирования следующих алгоритмов:

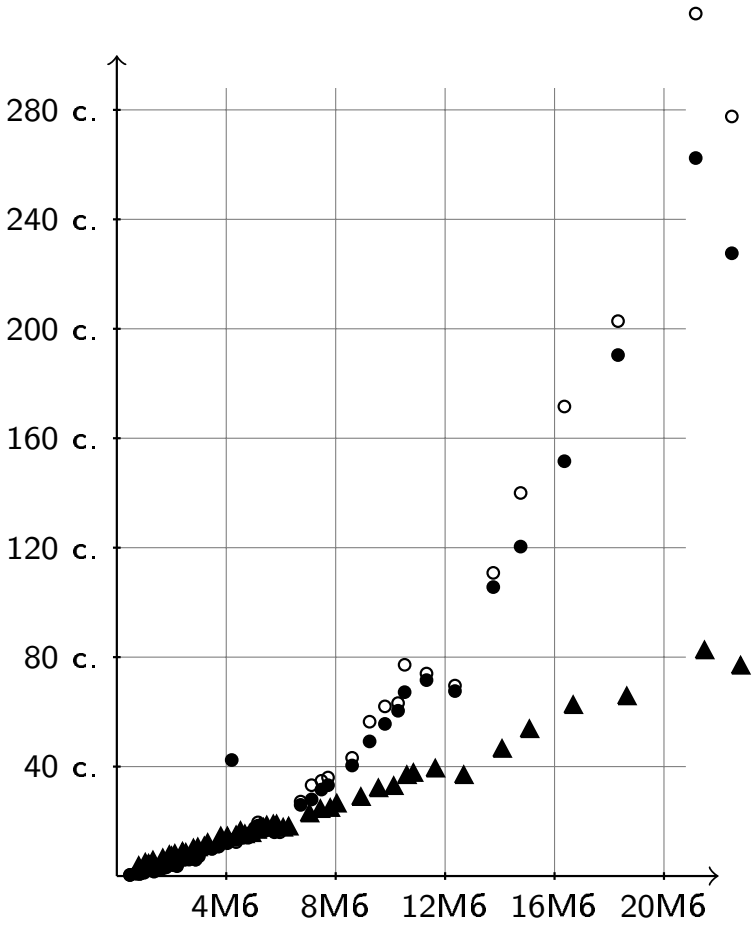
- Алгоритм *LZ*-факторизации. Обозначение – **lz**
- Алгоритм *LZ77*-факторизации. Обозначение – **lz77**
- Алгоритм Риттера. Обозначение – **SLPClassic**
- Модернизированный алгоритм Риттера. Обозначение – **SLPNew**
- Алгоритм построения ПП с помощью декартовых деревьев. Обозначение – **SLPCartesian**

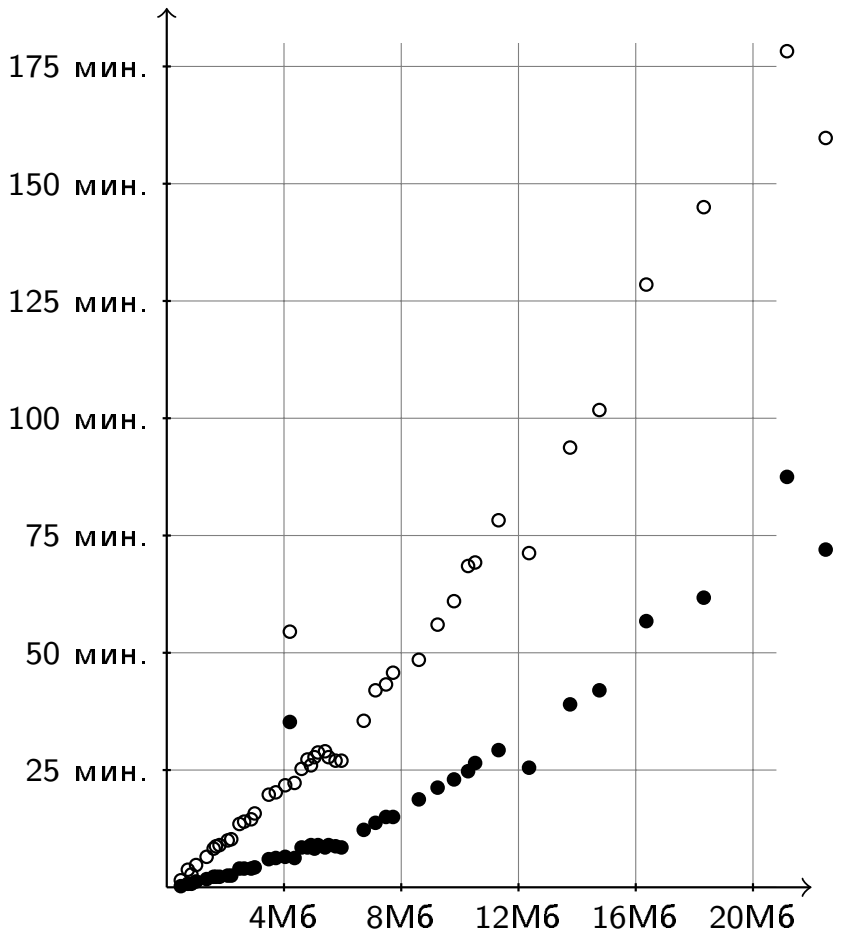
Число перебалансировок



◦ – **SLPclassic** • – **SLPnew**

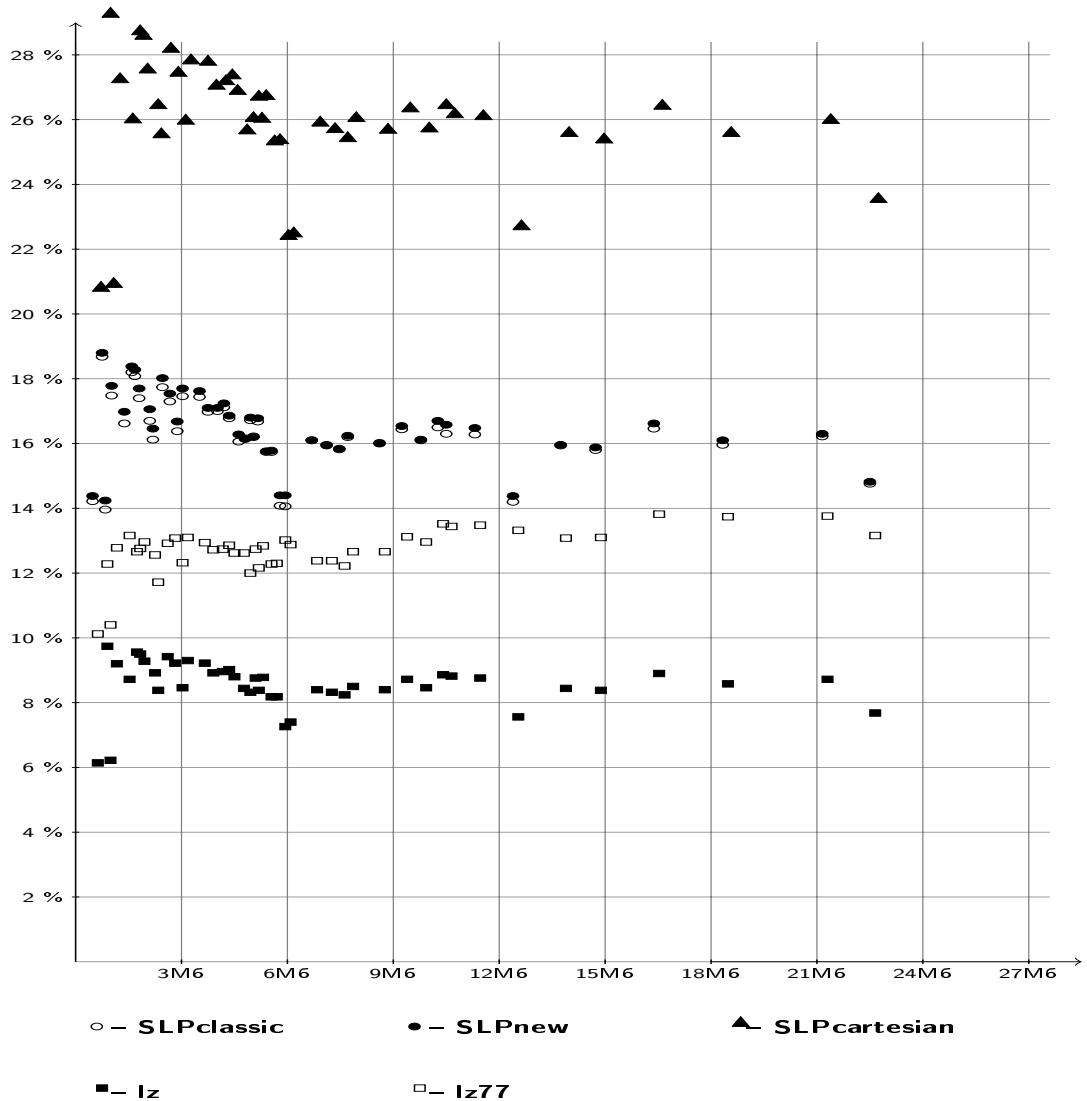
Скорость работы на строках ДНК



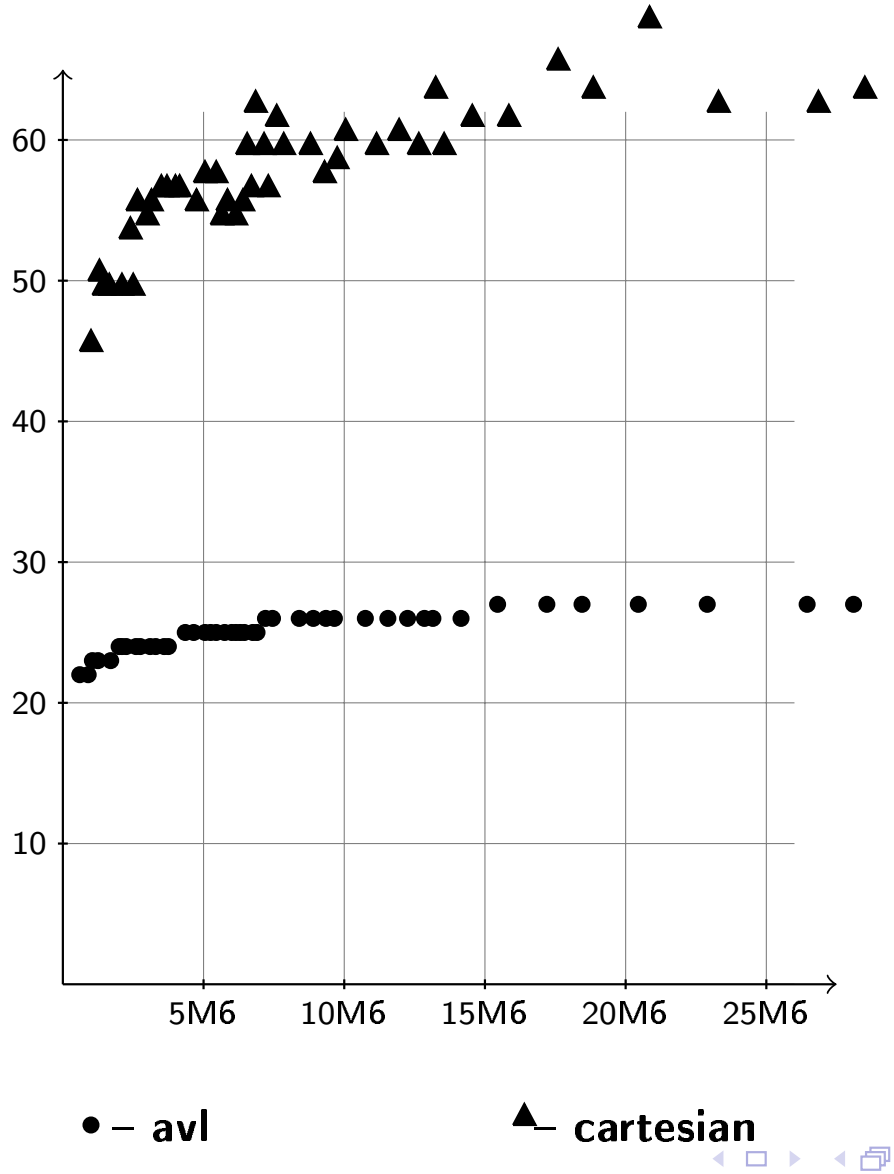


○ – **SLPclassic** • – **SLPnew** ▲– **SLPcartesian**

Коэффициенты сжатия



Высоты деревьев



Результаты

- Алгоритм *LZ*-факторизации с помощью суффиксного массива
- Алгоритм *LZ77*-факторизации с помощью суффиксного дерева
- Модернизированный алгоритм Риттера
- Алгоритм построения ПП с помощью декартовых деревьев
- Тесты

Результаты

- Алгоритм LZ-факторизации с помощью суффиксного массива
- Алгоритм $LZ77$-факторизации с помощью суффиксного дерева
- Модернизированный алгоритм Риттера
- Алгоритм построения ПП с помощью декартовых деревьев
- Тесты

Результаты

- Алгоритм LZ-факторизации с помощью суффиксного массива
- Алгоритм $LZ77$-факторизации с помощью суффиксного дерева
- Модернизированный алгоритм Риттера
- Алгоритм построения ПП с помощью декартовых деревьев
- Тесты

Результаты

- Алгоритм LZ-факторизации с помощью суффиксного массива
- Алгоритм $LZ77$-факторизации с помощью суффиксного дерева
- Модернизированный алгоритм Риттера
- Алгоритм построения ПП с помощью декартовых деревьев
- Тесты

Результаты

- Алгоритм LZ-факторизации с помощью суффиксного массива
- Алгоритм $LZ77$-факторизации с помощью суффиксного дерева
- Модернизированный алгоритм Риттера
- Алгоритм построения ПП с помощью декартовых деревьев
- Тесты

Результаты

- Алгоритм LZ-факторизации с помощью суффиксного массива
- Алгоритм $LZ77$-факторизации с помощью суффиксного дерева
- Модернизированный алгоритм Риттера
- Алгоритм построения ПП с помощью декартовых деревьев
- Тесты

Планы

- Алгоритм факторизации, эффективный с точки зрения операций чтения-записи
- Алгоритм построения ПП, эффективный с точки зрения операций чтения-записи
- Худший случай алгоритма Риттера или уточнение оценки размера ПП

Планы

- Алгоритм факторизации, эффективный с точки зрения операций чтения-записи
- Алгоритм построения ПП, эффективный с точки зрения операций чтения-записи
- Худший случай алгоритма Риттера или уточнение оценки размера ПП

Планы

- Алгоритм факторизации, эффективный с точки зрения операций чтения-записи
- Алгоритм построения ПП, эффективный с точки зрения операций чтения-записи
- Худший случай алгоритма Риттера или уточнение оценки размера ПП

Планы

- Алгоритм факторизации, эффективный с точки зрения операций чтения-записи
- Алгоритм построения ПП, эффективный с точки зрения операций чтения-записи
- Худший случай алгоритма Риттера или уточнение оценки размера ПП

Планы

- Алгоритм факторизации, эффективный с точки зрения операций чтения-записи
- Алгоритм построения ПП, эффективный с точки зрения операций чтения-записи
- Худший случай алгоритма Риттера или уточнение оценки размера ПП

Вопросы?