Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование
информационных систем

Системное программирование

Сусанина Юлия Алексеевна

# Оптимизация алгоритмов синтаксического анализа, основанных на матричных операциях

Дипломная работа

Научный руководитель:
к. ф.-м. н., доцент Григорьев С. В.

Рецензент:
научный координатор Центра Компьютерных Наук TUCS Бараш М. Л.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Yulia Susanina

# Optimization of parsing by matrix multiplication

Graduation Thesis

Scientific supervisor:
Assistant Professor Semyon Grigorev

Reviewer:
Ph.D., Scientific Coordinator, Turku Center for Computer Science Mikhail Barash

Saint-Petersburg
2019

# Оглавление

# Введение

Теория формальных языков активно изучается и находит широкое применение во многих областях, прежде всего, для формализации языков программирования и естественных языков. Также существует множество исследований, которые показывают эффективность использования формальных языков в биоинформатике для решения задач распознавания и классификации, некоторые из которых основаны на том, что вторичная структура геномных последовательностей содержит в себе важную информацию об организме. Характерные особенности вторичной структуры могут быть описаны с помощью некоторой контекстно-свободной (КС) грамматики, что позволяет свести проблему распознавания и классификации к задаче синтаксического анализа (определения принадлежности некоторой строки к языку, заданному грамматикой) [4, 9, 14]. Часто необходимо не просто проверить выводимость конкретной строки, но и найти все подстроки, принадлежащие некоторому формальному языку [3].

Большинство подходов к анализу биологических цепочек, которые основаны на синтаксическом анализе, сталкиваются проблемой низкой производительности. Чаще всего в этих подходах применяется алгоритм CYK [8, 16], который работает за кубическое время и неэффективен на длинных строках и для больших грамматик [10]. Необходимым требованием таких областей, как биоинформатика, является эффективная обработка больших объёмов данных, что приводит к необходимости усовершенствования существующих методов синтаксического анализа. Более того, некоторые особенности вторичной структуры не могут быть выражены с помощью КС-грамматик и требуют применения других классов грамматик [17].

На данный момент одним из самых быстрых алгоритмов, работающих с произвольной КС-грамматикой, является алгоритм Валианта [15], основанный на матричных операциях. Более того, данный алгоритм можно легко расширить для работы с конъюнктивными и булевыми грамматиками, которые обладают большей выразительностью [12].

Однако в связи с сложностью применения к выше упомянутой задаче поиска всех подстрок и отсутсвия эффективной реализации алгоритм Валианта достаточно редко используется на практике, несмотря на широкие большие потенциальные возможности.

В лаборатории языковых инструментов Jetbrains Research (СПбГУ) [7], Анной Явейн был предложен алгоритм, являющийся модификацией алгоритма Валианта [18] и обладающий определенными преимуществами, такими как легкость адаптации к задаче поиска подстрок и возможность повысить использование GPGPU и параллельных вычислений. Однако алгоритм Явейн не был должным образом изучен: для него отсутствует доказательство корректности и оценка сложности. Более того, не исследовано, как влияет на производительность данного алгоритма использование параллельных вычислений и библиотек для эффективной работы с матрицами.

# 1. Постановка задачи

Целью данной работы является исследование алгоритма Явейн и его адаптация к задаче поиска подстрок. Для её достижения были поставлены следующие задачи.

- Доказать корректность алгоритма Явейн и дать оценку его сложности.
- Проанализировать эффективность применения этого алгоритма и алгоритма Валианта к задаче поиска подстрок.
- Реализовать последовательную и параллельную версии алгоритмов Валианта и Явейн.
- Выполнить экспериментальное исследование алгоритма Явейн.

# 2. Обзор

В данном разделе мы введем основные определения из теории формальных языков и опишем алгоритмы синтаксического анализа, рассматриваемые в данной работе — алгоритмы Валианта и Явейн. Также мы отметим области применения данных алгоритмов и остановимся на биоинформатике, в частности, задаче поиска подстрок.

## 2.1. Терминология

Алфавитом $\Sigma$ будем называть некоторое конечное множество символов. Тогда $\Sigma^*$ — это множество всех конечных строк над алфавитом $\Sigma$.

Контекстно-свободная (КС) грамматика — это четверка $(\Sigma, N, R, S)$, где $\Sigma$ — конечное множетство терминальных символов, $N$ — конечное множество нетерминальных символов, $R$ — конечное множество правил вида $A \to \beta$, где $A \in N$, $\beta \in V^*$, $V = \Sigma \cup N$ и $S \in N$ — стартовый символ.

КС-грамматика $G_S = (\Sigma, N, R, S)$ находится в нормальной форме Хомского, если все ее правила имеют следующий вид: $A \to BC$, $A \to a$, или $S \to \varepsilon$, где $A, B, C \in N, a \in \Sigma$ и $\varepsilon$ — пустая строка.

$L_G(A) = \{\omega | A \overset{*}{\to} \omega\}$ — язык, порождаемый грамматикой $G_A = (\Sigma, N, R, A)$, где $A \overset{*}{\to} \omega$ означает, что $\omega$ может быть получена из нетерминала $A$ путем применения некотороый последовательности правил из $R$.

## 2.2. Алгоритм Валианта

Основной задачей синтаксического анализа является определение принадлежности некоторой строки языку, заданному грамматикой.

Алгоритм Валианта относится к табличным методам синтаксического анализа, которым на вход обычно подается грамматика в нормальной форме Хомского $G = (\Sigma, N, R, S)$ и некоторая строка $a_1 \dots a_n$, где $n + 1$ — степень двойки. Результатом работы данного алгоритма является верхнетреугольная матрица разбора $T$, элементами которой являются

подмножества нетерминалов. Каждый элемент отвечает за вывод конкретной подстроки: $T_{i,j} = \{A | A \in N, a_{i+1} \dots a_j \in L_G(A)\} \quad \forall i < j$.

Элементы матрицы инициализируются пустыми множествами. Сначала заполняется диагональ: $T_{i-1,i} = \{A | A \to a_i \in R\}$. Затем, матрица начинает последовательно заполняться по формуле $T_{i,j} = f(P_{i,j})$, где $P_{i,j} = \bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j}$ и $f(P_{i,j}) = \{A | \exists A \to BC \in R : (B,C) \in P_{i,j}\}$.

Входная строка $a_1 a_2 \dots a_n$ принадлежит языку $L_G(S)$ тогда и только тогда, когда $S \in T_{0,n}$.

Если все элементы данной матрицы заполнять последовательно, то вычислительная сложность данного алгоритма будет составлять $O(n^3)$. Валиант немного изменил порядок вычисления элементов, за счет чего смог свести самую затратную по времени операцию $\bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j}$ к задаче умножения некоторого количества булевых матриц.

Определим перемножение двух подматриц матрицы разбора $T$ следующим образом: пусть $X \in (2^N)^{m \times l}$ и $Y \in (2^N)^{l \times n}$ — подматрицы $T$, тогда $X \times Y = Z$, где $Z \in (2^{N \times N})^{m \times n}$ и $Z_{i,j} = \bigcup_{k=1}^{l} X_{i,k} \times Y_{k,j}$.

Теперь можно представить вычисление $X \times Y$ как перемножение $|N|^2$ булевых матриц (для каждой пары нетерминалов). Определим матрицу, соответствующую паре $(B,C) \in N \times N$, как $Z^{(B,C)}$, тогда $Z^{(B,C)}_{i,j} = 1$ тогда и только тогда, когда $(B,C) \in Z_{i,j}$. Заметим также, что $Z^{(B,C)} = X^B \times Y^C$. Каждое такое перемножение может совершаться абсолютно независимо. С этими изменениями сложность алгоритма будет составлять $\mathcal{O}(|G| BMM(n) log(n))$ для строки длины $n$, где $BMM(n)$ — количество операций, необходимое для перемножения двух булевых матриц размера $n \times n$, а $|G|$ — длина описания грамматики.

Детально опишем алгоритм Валинта (см. алгоритм 1). Все элементы матриц $T$ и $P$ заполняются с помощью двух рекурсивных процедур. Процедура $compute(l, m)$ корректно заполняет $T_{i,j}$ для всех $l \le i < j < m$. Процедура $complete(l, m, l', m')$ вычисляет $T_{i,j}$ для всех $l \le i < m, l' \le j < m'$. Предполагается, что $T_{i,j}$ для всех $l \le i < j < m, l' \le i < j < m'$ уже корректно заполнены и $P_{i,j} = \{(B,C) | \exists k, (m \le k < l'), a_{i+1} \dots a_k \in$

$L(B), a_{k+1} \dots a_j \in L(C)\}$ для всех $l \le i < m, l' \le j < m'$.

Алгоритм 1: Алгоритм Валианта

Input: Грамматика $G = (\Sigma, N, R, S), w = a_1 \dots a_n, n \ge 1, a_i \in \Sigma$, где $n + 1 = 2^k$

```
1  main():
2  compute(0, n + 1);
3  accept if and only if S ∈ T_{0,n}

4  compute(l, m):
5  if m − l ≥ 4 then
6      compute(l, (l+m)/2);
7      compute((l+m)/2, m)
8  end
9  complete(l, (l+m)/2, (l+m)/2, m)

10 complete(l, m, l', m'):
11 if m − l = 4 and m = l' then T_{l,l+1} = {A | A → a_{l+1} ∈ R};
12 else if m − l = 1 and m < l' then  T_{l,l'} = f(P_{l,l'});
13 else if m − l > 1 then
14     leftgrounded = (l, (l+m)/2, (l+m)/2, m), rightgrounded = (l', (l'+m')/2, (l'+m')/2, m'),
15     bottom = ((l+m)/2, m, l', (l'+m')/2), left = (l, (l+m)/2, l', (l'+m')/2),
16     right = ((l+m)/2, m, (l'+m')/2, m'), top = (l, (l+m)/2, (l'+m')/2, m');
17     complete(bottom);
18     P_left = P_left ∪ (T_leftgrounded × T_bottom);
19     complete(left);
20     P_right = P_right ∪ (T_bottom × T_rightgrounded);
21     complete(right);
22     P_top = P_top ∪ (T_leftgrounded × T_right);
23     P_top = P_top ∪ (T_left × T_rightgrounded);
24     complete(top)
25 end
```

## 2.3. Алгоритм Явейн

Теперь рассмотрим алгоритм Явейн, являющийся модификацией алгоритма Валианта. Его главным отличием является возможность разбиения матрицы разбора на слои непересекающихся подматриц.

Заполнение слоев производится последовательно, снизу вверх. Слой состоит из квадратных подматриц размера $2^n$, $n > 0$. На момент начала заполнения слоя нижняя часть матриц ($bottom$) уже заполнена, так как принадлежит предыдущему слою, поэтому эти слои мы также будем называть V-образными. Пример разбиения матрицы разбора

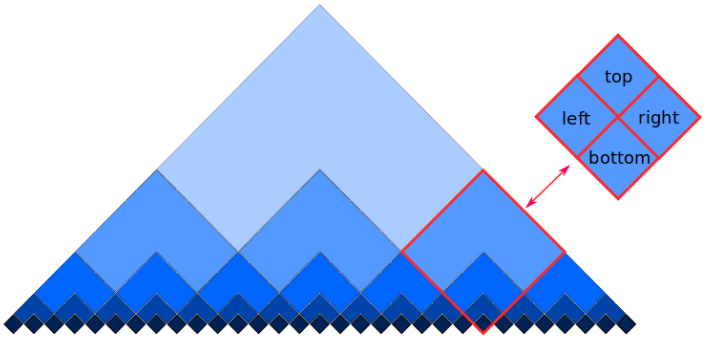



Рис. 1: Деление матриц на V-образные слои

на слои показан на рис. 1. Заметим, что каждую матрицу слоя можно обрабатывать параллельно.

Пример работы алгоритма показан на рис. 2. Нижний слой, состоящий из подматриц размера 1, вычисляется заранее, а заполнение матрицы начинается со второго слоя. (Здесь и далее, под слоем матриц будем понимать некоторое множество её подматриц разбора.) Более того, на рис. 4 на каждом шаге изображены операции, которые могут быть выполненные независимо, что позволяет значительно упростить разработку параллельной версии алгоритма.

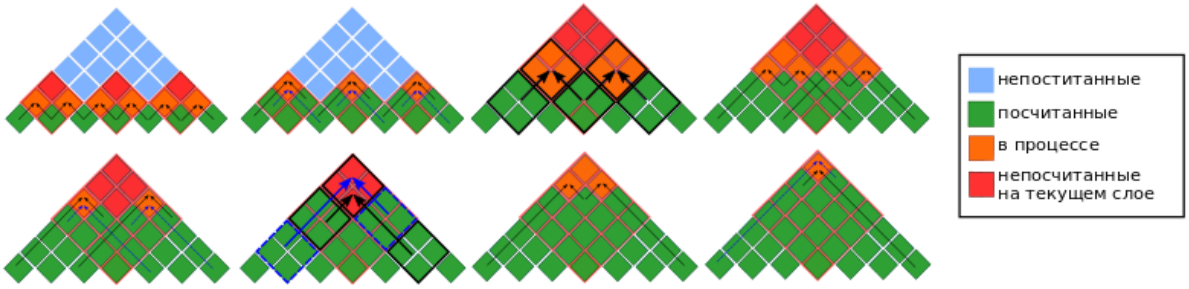



Рис. 2: Деление матриц на V-образные слои

Функция *main()* (см. алгоритм 2) заполняет диагональ: $(T_{l,l+1})$, затем делит матрицу на слои и вычисляет их с помощью процедуры *completeVLayer()*.

Дополнительные функции *left(subm)*, *right(subm)*, *top(subm)*, *bottom(subm)*, *rightgrounded(subm)* and *leftgrounded(subm)* возвращают подматрицы матрицы $subm = (l, m, l', m')$ аналогично алгоритму 1.

Процедура *completeVLayer(M)* на вход принимает слой (массив подматриц) $M$ и для каждой *subm = (l, m, l', m')* $\in M$ заполняет *left(subm), right(subm), top(subm).* Предполагается, что *bottom(subm)* и $T_{i,j}$ для всех $i$, $j$, таких что $l \leq i < j < m$, $l' \leq i < j < m'$ уже корректно вычислены и $P_{i,j} = \{(B, C) | \exists k, (m \leq k < l'), a_{i+1} \dots a_k \in L_G(B), a_{k+1} \dots a_j \in L_G(C)\}$ для всех $i$, $j$, таких что $l \leq i < m$, $l' \leq j < m'$.

---

Алгоритм 2: Алгоритм Явейн

---

Input: Грамматика $G = (\Sigma, N, R, S), w = a_1 \dots a_n, n \geq 1, a_i \in \Sigma$, где $n + 1 = 2^p$

main():
for $l \in \{1, \dots, n\}$ do $T_{l,l+1} = \{A | A \to a_{l+1} \in R\}$;
for $1 \leq i < p - 1$ do
    *layer = constructLayer(i)*;
    *completeVLayer(layer)*
end
accept if and only if $S \in T_{0,n}$

constructLayer(i):
$\{(k2^i, (k+1)2^i, (k+1)2^i, (k+2)2^i) \,|\, 0 \leq k < 2^{p-i} - 1\}$

completeLayer(M):
if $\forall (l, m, l', m') \in M \quad (m - l = 1)$ then
    for $(l, m, l', m') \in M$ do $T_{l,l'} = f(P_{l,l'})$;
end
else
    *completeLayer(*$\{$*bottom(subm)* $|$ *subm* $\in M\}$*)*;
    *completeVLayer(M)*
end

completeVLayer(M):
$multiplicationTasks_1 =$
  $\{$*left(subm), leftgrounded(subm), bottom(subm)* $|$ *subm* $\in M\} \cup$
  $\{$*right(subm), bottom(subm), rightgrounded(subm)* $|$ *subm* $\in M\}$;
$multiplicationTask_2 = \{$*top(subm), leftgrounded(subm), right(subm)* $|$ *subm* $\in M\}$;
$multiplicationTask_3 = \{$*top(subm), left(subm), rightgrounded(subm)* $|$ *subm* $\in M\}$;
*performMultiplications(multiplicationTask*$_1$*)*;
*completeLayer(*$\{$*left(subm)* $|$ *subm* $\in M\} \cup \{$*right(subm)* $|$ *subm* $\in M\}$*)*;
*performMultiplications(multiplicationTask*$_2$*)*;
*performMultiplications(multiplicationTask*$_3$*)*;
*completeLayer(*$\{$*top(subm)* $|$ *subm* $\in M\}$*)*

performMultiplication(tasks):
for $(m, m1, m2) \in$ *tasks* do $P_m = P_m \cup (T_{m1} \times T_{m2})$;

---

Процедура *completeLayer(M)* также принимает на вход массив мат-

риц $M$, но заполняет $T_{i,j}$ для всех $(i, j) \in subm$. Ограничение на $T_{i,j}$ and $P_{i,j}$ такие же, как в предыдущем случае, кроме условия на *bottom(subm)*.

Другими словами, *completeVLayer(M)* отвечает за заполнение слоя $M$, а *completeLayer($M_2$)* — вспомогательная функция для вычисления для вычисления меньших матриц внутри слоя $M$.

Теперь обратим внимание на процедуру *performMultiplication(tasks)*, где *tasks* — массив троек подматриц, реализующий основной шаг алгоритма: перемножение матриц. Здесь $|tasks| \geq 1$ и каждый $task \in tasks$ может быть выполнен параллельно, в отличие от алгоритма Валианта.

## 2.4. Применение в биоинформатике и задача поиска подстрок

Вторичная структура (определенный способ укладки биологической цепочки в сложную, упорядоченную структуру) генетических последовательностей, например, РНК, тесно связана с биологическими функциями организма, поэтому анализ таких последовательностей играет существеную роль в задачах распознавания и классификации.

Характерные черты вторичной структуры могут быть описаны с помощью КС-грамматики и, следовательно, часть подходов для анализа генетических последовательностей основаны на синтаксическом анализе. Главным недостатком таких подходов являются существенные проблемы с производительностью [3], которые можно решить с помощью алгоритма Валианта.

Однако часто задачей является нахождение не одной, а всех подпоследовательностей, обладающих данными чертами. Для этой задачи алгоритм Валианта плохо применим, так как его трудно остановить на определенном этапе заполнения матрицы разбора и это потребует много лишних перемножений матриц. Предполагается, что алгоритм Явейн должен решить эту проблему, но сначала надо показать, что он не утратил преимущества исходного алгоритма.

# 3. Доказательство корректности и оценка сложности алгоритма Явейн

В данном разделе мы приведем доказательство корректности алгоритма Явейн и дадим оценку его вычислительной сложности.

***Лемма 1.*** *Пусть процедура completeLayer(M') с выполненными следующими ограничениями:*

1. $T_{i,j} = \{A | a_{i+1} \dots a_j \in L_G(A)\}$ *для всех* $i$ *и* $j$, *таких что* $l1 \leq i < j < m1$ *и* $l2 \leq i < j < m2$;

2. $P_{i,j} = \{(B,C) | \exists k, (m1 \leq k < l2) : a_{i+1} \dots a_k \in L_G(B), a_{k+1} \dots a_j \in L_G(C)\}$ *для всех* $l1 \leq i < m1$ *и* $l2 \leq j < m2$.

*возвращает для любого слоя* $M'$ *корректно заполненные* $T_{i,j}$ *для всех* $l1 \leq i \leq m1$ *и* $l2 \leq j \leq m2$ *при этом* $(l1, m1, l2, m2) \in M'$ *и пусть для слоя* $M$ *выполняется, что:*

1. $T_{i,j} = \{A | a_{i+1} \dots a_j \in L_G(A)\}$ *для всех* $i$ *и* $j$, *таких что* $l \leq i < j < m$ *и* $l' \leq i < j < m'$ *и для* $(i,j) \in bottom(M)$;

2. $P_{i,j} = \{(B,C) | \exists k, (m \leq k < l') : a_{i+1} \dots a_k \in L_G(B), a_{k+1} \dots a_j \in L_G(C)\}$ *для всех* $l \leq i < m$ *и* $l' \leq j < m'$.

*Тогда процедура completeVLayer(M) возвращает корректно заполненные* $T_{i,j}$ *при этом* $l \leq i \leq m$ *и* $l' \leq j \leq m'$ *для всех* $(l, m, l', m') \in M$.

*Доказательство.* Сначала *performMultiplications(multiplicationTask*$_1$*)* добавит к каждому $P_{i,j}$ все пары $(B,C)$, такие что $\exists k$, $(\frac{l+m}{2} \leq k < l')$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ для всех $(i,j) \in leftsublayer(M)$ и $(B,C)$, такие что $\exists k$, $(m \leq k < \frac{l'+m'}{2})$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ для всех $(i,j) \in rightsublayer(M)$. Теперь, так как все ограничения соблюдены, можно вызвать *completeLayer(leftsublayer(M) ∪ rightsublayer(M))* и она вернет корректно заполненные *leftsublayer(M) ∪ rightsublayer(M)*.

Далее функция *performMultiplications*, вызванная от аргументов *multiplicationTask*$_2$ и *multiplicationTask*$_3$, к каждому $\mathrm{P}_{i,j}$ добавит все пары $(B, C)$, такие что $\exists k$, $(\frac{l+m}{2} \leq k < m)$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ и все пары $(B, C)$, такие что $\exists k$, $(l' \leq k < \frac{l'+m'}{2})$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ для всех $(i, j) \in topsublayer(M)$. Так как $m = l'$ (из построения слоя), ограничения на элементы $P$ выполнены. И процедура *completeLayer(topsublayer(M))* может быть вызвана и она вернет корректно заполненные *topsublayer(M)*.

Теперь $\forall (i, j) \in M$ $T_{i,j}$ заполены корректно.

□

***Теорема 1.*** *Пусть $M$ — слой, и для всех $(l, m, l', m') \in M$ справедливо:*

1. *$T_{i,j} = \{A | a_{i+1} \dots a_j \in L_G(A)\}$ для всех $i$ и $j$, таких что $l \leq i < j < m$ и $l' \leq i < j < m'$;*

2. *$P_{i,j} = \{(B, C) | \exists k, (m \leq k < l') : a_{i+1} \dots a_k \in L_G(B), a_{k+1} \dots a_j \in L_G(C)\}$ для всех $l \leq i < m$ и $l' \leq j < m'$.*

*Тогда процедура completeLayer(M) возвращает корректно заполненные $T_{i,j}$ для всех $l \leq i \leq m$ и $l' \leq j \leq m'$ при этом $(l, m, l', m') \in M$.*

*Доказательство.* (Индукция по $m - l$.)

Будем рассматривать одну матрицу $(l, m, l', m')$, так как для остальных матриц процесс заполнения производится аналогично.

База индукции: $m - l = 1$. Необходимо вычислить всего один элемент и $P_{l,l'} = \{(B, C) | a_{l+1} \dots a_{l'} \in L(B)L(C)\}$. Алгоритм вычисляет $f(P_{l,l'}) = \{A | a_{l+1} \dots a_{l'} \in L(A)\}$ и $T_{l,l'}$ теперь заполнена корректно.

Индукционный переход: Предположим, что $(l_1, m_1, l_2, m_2)$ корректно заполняются для всех $m_2 - l_2 = m_1 - l_1 < m - l$.

Рассмотрим вызов процедуры *completeLayer(M)*, где $m - l > 1$.

Все ограничения на вызов *completeLayer(bottomsublayer(M))* выполнены и $T_{i,j}$ будут корректно заполнены для всех $(i, j) \in bottomsublayer(M)$. Стоит отдельно упомянуть, что условия теоремы позволяют корректно заполнить самый нижний элемент $T_{m,l'}$

(аналогично тому, как это сделано в базе индукции). Слой *bottomsublayer(M)* теперь заполнен и можно вызвать процедуру *completeVLayer(M)*.

Все $T_{i,j}$ уже заполнены для всех $i$ и $j$, таких что $l \leq i < j < m$ и $l' \leq i < j < m'$ из условий теоремы, следовательно теперь мы можем применить лемму 1. Это значит, что $T_{i,j}$ $\forall (i,j) \in M$ будут заполнены корректно. □

***Теорема 2.*** *Алгоритм Явейн (см. алгоритм 2) корректно заполняет $T_{i,j}$ для всех $i$ и $j$, и входная строка $a = a_1 a_2 \dots a_n \in L_G(S)$ тогда и только тогда, когда $S \in T_{0,n}$.*

*Доказательство.* Докажем по индукции, что все слои матрицы разбора $T$ вычисляются корректно.

База индукции: Слой размера $1 \times 1$ корректно заполняется в строках 2-3 листинга 2.

Индукционный переход: Предположим, что все слои размера $\leq 2^{p-2} \times 2^{p-2}$ вычислены корректно.

Обозначим слой размера $2^{p-1} \times 2^{p-1}$ как $M$. Будем рассматривать одну матрицу слоя *subm = (l, m, l', m')*, так как для остальных подматриц их заполнение будет проходить аналогично.

Рассмотрим вызов процедуры *completeVLayer(M)* call. $T_{i,j}$ для всех $i$ и $j$ таких, что $l \leq i < j < m$ и $l' \leq i < j < m'$, уже корректно заполнены, так как эти элементы лежат в слоях, которые уже вычислены по индукционному предположению.

Все условия леммы 1 и теоремы 1 выплнены. Следовательно, *completeVLayer(M)* возвращает корректно заполненные $T_{i,j}$ для всех $(i,j) \in M$ для каждого слоя $M$ матрицы разбора $T$ и строки 4-6 листинга 2 возвращают все $T_{i,j} = \{A | A \in N, a_{i+1} \dots a_j \in L_G(A)\}$.

□

***Лемма 2.*** *Пусть $calls_i$ — это количество вызовов процедуры completeVLayer(M), где для всех $(l, m, l', m') \in M$ выполняется $m - l = 2^{p-i}$. Тогда истинны следующие утверждения:*

- *для всех* $i \in \{1, .., p-1\}$ $\sum_{n=1}^{calls_i} |M| = 2^{2i-1} - 2^{i-1}$;
- *для всех* $i \in \{1, .., p-1\}$ *матрицы размера* $2^{p-i} \times 2^{p-i}$ *перемножаются ровно* $2^{2i-1} - 2^i$ *раз.*

*Доказательство.* Сначала докажем первое утверждение индукцией по $i$.

База индукции: i = 1. $calls_1$ и $|M| = 1$. So, $2^{2i-1} - 2^{i-1} = 2^1 - 2^0 = 1$.

Индукционный переход: предположим, что $\sum_{n=1}^{calls_i} |M| = 2^{2i-1} - 2^{i-1}$ для всех $i \in \{1, .., j\}$.

Пусть $i = j + 1$.

Заметим, что функция *costructLayer(i)* возвращает $2^{p-i} - 1$ матриц размера $2^i$, то есть в вызове процедуры *completeVLayer(costructLayer(k - i)) costructLayer(k - i)* вернет $2^i - 1$ матриц размера $2^{p-i}$. Также процедура *completeVLayer(M)* будет вызвано 3 раза для левых, правых и верхних подматриц матриц размера $2^{p-(i-1)}$. Кроме того, *completeVLayer(M)* вызывается 4 раза для нижних, левых, правых и верхних подматриц матриц размера $2^{p-(i-2)}$, за исключением левых, правых и верхних подматриц матриц размера $2^{i-2} - 1$, которые к этому моменту уже были посчитаны.

Таким образом, $\sum_{n=1}^{calls_i} |M| = 2^i - 1 + 3 \times (2^{2(i-1)-1} - 2^{(i-1)-1}) + 4 \times (2^{2(i-2)-1} - 2^{(i-2)-1}) - (2^{i-2} - 1) = 2^{2i-1} - 2^{i-1}$.

Теперь мы знаем, что $\sum_{n=1}^{calls_{i-1}} |M| = 2^{2(i-1)-1} - 2^{(i-1)-1}$, и можем доказать второе утверждение. Посчитаем количество перемножений матриц размера $2^{p-i} \times 2^{p-i}$. Процедура *performMultiplications* вызывается 3 раза, $|multiplicationTask1| = 2 \times 2^{2(i-1)-1} - 2^{(i-1)-1}$ и $|multiplicationTask2| = |multiplicationTask3| = 2^{2(i-1)-1} - 2^{(i-1)-1}$. То есть, количество перемножений подматриц размера $2^{p-i} \times 2^{p-i}$ равно $4 \times (2^{2(i-1)-1} - 2^{(i-1)-1}) = 2^{2i-1} - 2^i$. □

***Теорема 3.*** *Пусть* $|G|$ *— длина описания грамматики* $G$, $n$ *— длина входной строки. Тогда алгоритм из листинга 2 заполняет матрицу разбора* $T$ *за* $\mathcal{O}(|G| BMM(n) \log n)$, *где* $BMM(n)$ *— количество операций, необходимое для перемножения двух булевых матриц размера* $n \times n$.

*Доказательство.* Так как в лемме 2 было показано, что количество перемножений матриц не изменилось по сравнению с исходной версией алгоритма Валианта, то доказательство будет идентично доказательству теоремы 1 [12]. □

Таким образом, мы доказали корректность алгоритма Явейн, а также показали, что его сложность осталась такой же, как и сложность исходного алгоритма Валианта.

# 4. Анализ эффективности применения к задаче поиска подстрок алгоритмов Валианта и Явейн

В данном разделе мы продемонстрируем, как алгоритм Явейн может быть применен к задаче поиска подстрок. Пусть мы хотим для входной строки размера $n = 2^p$ найти все подстроки размера $s$, которые принадлежат языку, заданному грамматикой $G$. Тогда мы должны посчитать слои подматриц, размер которых не превышает $2^r$, где $2^{r-2} < s \leq 2^{r-1}$.

Пусть $r = p-(m-2)$ и, следовательно, $(m-2) = p-r$. Для всех $m \leq i \leq p$ перемножение матриц размера $2^{p-i}$ выполняется ровно $2^{2i-1}-2^i$ раз и каждое из них включает перемножение $\mathcal{O}(|G|)$ булевых подматриц.

$$C \sum_{i=m}^{p} 2^{2i-1} \cdot 2^{\omega(p-i)} \cdot f(2^{p-i}) = C \cdot 2^{\omega l'} \sum_{i=2}^{r} 2^{(2-\omega)i} \cdot 2^{2(p-r)-1} \cdot f(2^{r-i}) \leq$$
$$C \cdot 2^{\omega r} f(2^r) \cdot 2^{2(p-r)-1} \sum_{i=2}^{r} 2^{(2-\omega)i} = BMM(2^r) \cdot 2^{2(p-r)-1} \sum_{i=2}^{r} 2^{(2-\omega)i}$$

Временная сложность алгоритма для поиска всех подстрок длины $s$ равна $\mathcal{O}(|G|2^{2(p-r)-1}BMM(2^r)(r-1))$, где появившийся дополнительный множитель обозначает количество матриц в последнем вычисленном слое, но он, во-первых, мал относительно общей работы алгоритма, во-вторых, не существенен, так как эти матрицы могут быть обработаны параллельно.

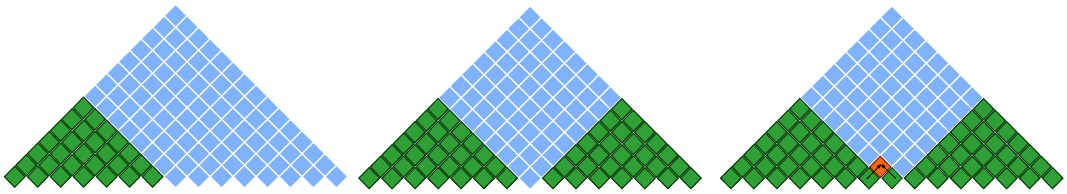


Рис. 3: Количество элементов, вычисляемых в алгоритме Валианта (2 треугольные подматрицы размера $\frac{n}{2}$), выделенные зеленым цветом.

Алгоритм Валианта, в отличие от модификации, не может так легко быть применен к данной задаче. В нем необходимо будет полностью вычислить, как минимум, две треугольные подматрицы размера $\frac{n}{2}$, как

показано на рис. 3. Это значит, что минимальная сложность, улучшить которую без дополнительных модификаций не удастся, будет составлять $\mathcal{O}(|G|BMM(2^{p-1})(p-2))$.

Таким образом, в данном разделе мы показади, что алгоритм Явейн может быть эффективно применен для строк размера $s \ll n$.

# 5. Реализация алгоритмов Валианта и Явейн

В рамках данной работы мы реализовали алгоритм Явейн несколькими способами. Мы хотели исследовать, как повлияют на производительность те или иные особенности каждой реализации. Также был реализован исходный алгоритм Валианта для сравнения и проверки эффективности модифицированного алгоритма.

## 5.1. Последовательная версия

Первая реализизация основана на использовании уже существующих библиотек. Языком программирования был выбран C++. Для перемножения матриц была использована библиотека для работы с плотними матрицами — M4RI [1]. Данная библиотека была выбрана, так как там реализован один их наиболее эффективных способов перемножения булевых матриц — метод четырех русских [2, 13].

## 5.2. Параллельная версия

Далее мы решили остановиться на использовании параллельных техник, а именно GPGPU (General-purpose computing on graphics processing units). Была создана простая реализация перемножения подматриц на языке программирования CUDA C [11]. Это расширение языка C, позволяющее создавать эффективные программы за счет использования возможностей графических процессоров. Использование параллельных вычислений происходит сразу на трех уровнях: само перемножение матриц (каждый элемент результирующий матрицы обрабатывается независимо), перемножение булевых матриц для каждой пары нетерминалов, которым соответствует хотя бы одно правило, и перемножение подматриц слоя для алгоритма Явейн.

# 6. Эксперименты

В данной секции мы приводим результаты экспериментов, целью которых было исследование производительности и практической применимости алгоритма Явейн.

Эксперименты проводились на рабочей станции со следующими характеристиками:

- операционная система: Linux Mint 19.1;
- ЦПУ: Intel i5-8250U, 1600-3400 Mhz, 4 Core(s), 8 Logical Processor(s);
- объем оперативной памяти: 8.0 GB;
- графический процессор: NVIDIA GeForce GTX 1050 MAX-Q.

Основной целью поставленных экспериментов было исследование возможностей алгоритма Явейн. Для этого были поставлены следующие вопросы.

Q1. Сравнение алгоритмов Валианта и Явейн.

Q2. Эффективность применения алгоритма Явейн к задаче поиска подстрок.

Для ответа на вопрос Q1 был проведен сравнительный анализ как последовательной, так и параллельной версий реализации алгоритмов Валианта и Явейн.

При исследовании алгоритмы были протестированы на двух грамматиках. Сначала была выбрана грамматика Дика для двух типов скобок ($D2$) [6], потому что грамматики для описания правильных скобочных последовательностей применяются при анализе строк в биоинформатике. Она представлена на листинге 1.

$$\text{s} : \text{s s} \mid (\text{s}) \mid [\text{s}] \mid \epsilon$$

Листинг 1: Грамматика $D2$

Грамматика $D2$ переводится в нормальную форму Хомского и подается на вход алгоритму со специально сгенерированными строками

различной длины (127-8191 символов). Строки составлены следующим образом: заранее создается подстрока, принадлежащая языку Дика, далее в полную строку вставляется максимально возможное количество созданных подстрок, которые можно разделить “перегородками” (терминалами, из-за которых все остальные строки, кроме вставленных, будут невыводимыми в грамматике $D2$). Строки были созданы таким образом, чтобы проверять корректность предложенного алгоритма.

Второй грамматикой для тестирования производительности алгоритмов была выбрала грамматика, применяющаяся в некоторых работах по биоинформатике ($BIO$) [5]. Она представлена на листинге 2. Грамматика $BIO$ также, как и в предыдущем случае, переводится в нормальную форму Хомского. Строки для данного эксперимента были сгенерированны случайным образом.

```
s1: stem<s0>
any_str : any_smb*[2..10]
s0: any_str | any_str stem<s0> s0
any_smb: A | T | C | G
stem1<s>: A s T | G s C | T s A | C s G
stem2<s>: stem1<stem1<s>>
stem<s>:
      A stem<s> T
    | T stem<s> A
    | C stem<s> G
    | G stem<s> C
    | stem1<stem2<s>>
```

Листинг 2: Грамматика $BIO$

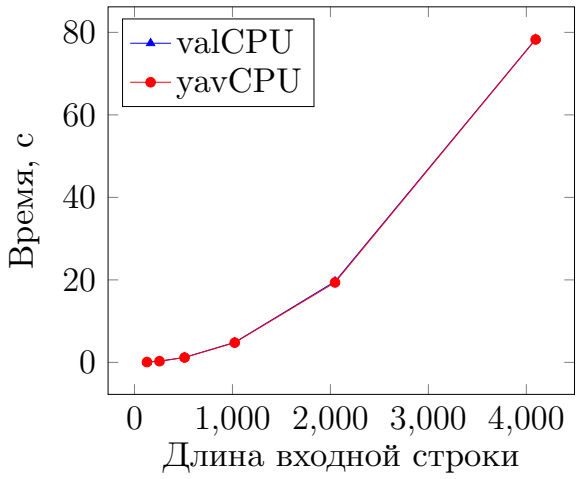
Для ответа на вопрос Q2 в алгоритме Явейн была изменена функция $main()$: теперь она принимает дополнительный аргумент s — длину максимальной искомой подстроки и вычисляет только те слои, которые содержат в себе нужные подстроки, как было показано в разделе 4. Далее версию алгоритма Явейн с измененной функцией $main()$ будем называть адаптированнной к задаче поиска подстрок. Тестирование проводилось на грамматиках $D2$ и $BIO$, строки были сгенерированы также, как и в предыдущем случае.

## 6.1. Сранительный анализ

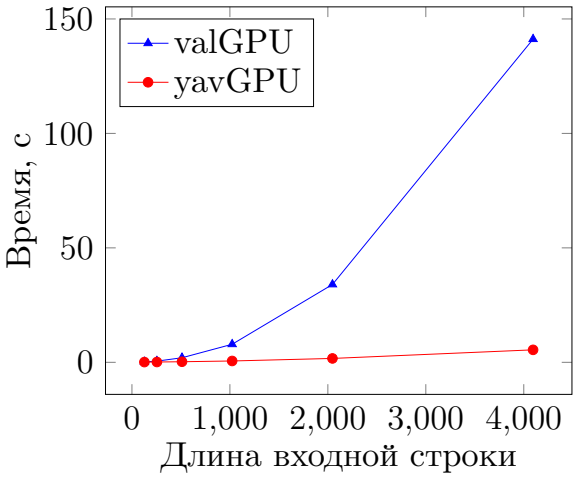
Результаты сравнительного анализа реализаций алгоритма Валианта и Явейн представлены в таблице 1 и на рис. 4. При этом $N$ — это длина сгенерированной строки, valCPU — последовательная реализация алгоритма Валианта, yavCPU — последовательная реализация алгоритма Явейн, valGPU — параллельная реализация алгоритма Валианта и yavGPU — параллельная версия алгоритма Явейн. Время работы алгоритмов представленно в таблице 1 в миллисекундах.

Таблица 1: Результаты сравнительного анализа (время в мс)

| N | Грамматика $D2$ | | | | Грамматика $BIO$ | | | |
|---|---|---|---|---|---|---|---|---|
| | valCPU | yavCPU | valGPU | yavGPU | valCPU | yavCPU | valGPU | yavGPU |
| 127 | 78 | 76 | 195 | 105 | 1345 | 1339 | 193 | 106 |
| 255 | 289 | 292 | 523 | 130 | 5408 | 5488 | 525 | 140 |
| 511 | 1212 | 1177 | 1909 | 250 | 21969 | 22347 | 1994 | 256 |
| 1023 | 4858 | 4779 | 7878 | 540 | 88698 | 90318 | 7890 | 598 |
| 2047 | 19613 | 19379 | 33508 | 1500 | 363324 | 374204 | 34010 | 1701 |
| 4095 | 78361 | 78279 | 140473 | 4453 | 1467675 | 1480594 | 141104 | 5472 |
| 8191 | 315677 | 315088 | - | 13650 | - | - | - | 18039 |




(a) Последовательная реализация




(b) Параллельная версия

Рис. 4: Результаты экспериментов с грамматикой $D2$.

Результаты сравнительного анализа для последовательной версии показывают, что алгоритмы работают практически одинаково. Однако видно, как на последовательную реализацию влияет константа $|G|$ —

длина описания грамматики $G$. В нашем случае, для грамматики $D2$ количество правил при переводе в нормальную форму Хомского составляет 7, а для $BIO$ — 106. Это означает, что скорость работы алгоритмов прямо пропорциональна количеству правил грамматики.

Параллельная версия алгоритма Валианта оказалась медленнее на грамматике $D2$. Можно предположить, что это связано с большим количеством перемножений матриц небольшого размера и невозможностью их параллельной обработки. Но на больших грамматиках ($BIO$) она демонстрирует значительное улучшение производительности по сравнению с последовательной версией за счет использования параллелизма на уровне правил грамматики (то есть независимого перемножения матриц для каждой пары нетерминалов).

Лучшее время работы показывает параллельная версия алгоритма Явейн. Это связано с тем, что параллелизм используется сразу на трех уровнях, как было замечено в предыдущем разделе.

## 6.2. Применимость к задаче поиска подстрок

Результаты работы адаптированного к задаче поиска подстрок алгоритма Явейн представлены в таблице 2. Здесь N — это длина сгенерированной строки, s — длина искомой подстроки, adpCPU — время работы последовательной реализации адаптированного алгоритма Явейн, adpGPU — время работы параллельной реализации адаптированнного алгоритма Явейн. Время работы на таблице 2 представлено в миллисекундах.

Результаты второго эксперимента показывают, что адаптированная версия алгоритма Явейн может быть эффективно применена к задаче поиска подстрок: она корректно находит все выводимые подстроки в строке и работает существенно быстрее алгоритма Валианта (см. таблицу 1), который будет совершать большое количество лишних вычислений из-за сложности его преждевременной остановки.

Таблица 2: Результаты работы алгоритма Явейн для задачи поиска подстрок (время в мс)

| s | N | adpCPU | adpGPU |
|---|---|---|---|
| 250 | 1023 | 2996 | 242 |
| | 2047 | 6647 | 255 |
| | 4095 | 13825 | 320 |
| | 8191 | 28904 | 456 |
| 510 | 2047 | 12178 | 583 |
| | 4095 | 26576 | 653 |
| | 8191 | 56703 | 884 |
| 1020 | 4095 | 48314 | 1590 |
| | 8191 | 108382 | 1953 |
| 2040 | 4095 | 197324 | 5100 |

Таким образом, проведенные эксперименты показали практическую применимость алгоритма Явейн, возможность его существенного ускорения за счет использования параллельных вычислений и адаптации к задаче поиска подстрок.

# 7. Заключение

В данной работе были получены следующие результаты.

- Доказана корректность алгоритма Явейн и дана оценка вычислительной сложности, которая составляет $\mathcal{O}(BMM(n)log(n))$.
- Проведен анализ, который показал, что алгоритм Явейн лучше применим к задаче поиска подстрок, чем алгоритм Валианта.
- Реализованы последовательная и параллельная версии алгоритмов. Исходный код доступен в репозитории: https://github.com/SusaninaJulia/PBMM.
- Проведено экспериментальное исследование алгоритма, показавшее эффективность алгоритма Явейн: последовательные версии алгоритмов Валианта и Явейн работают одинаково; параллельная версия алгоритма Явейн показывает значительный прирост производительности на строках большей длины; показана эффективность применения алгоритма Явейн к задаче поиска подстрок.
- Результаты работы приняты к публикации в журнале «Труды ИСП РАН».

Кроме того, мы можем определить несколько направлений будущих исследований. Например, оптимизация алгоритмов перемножения матриц за счет использования разделяемой памяти позволит повысить производительность алгоритмов. Еще планируется расширить алгоритм Явейн для других классов грамматик, которые применяются в биоинформатике: конъюнктивных и булевых. Также, открытым остается вопрос, можно ли как-либо изменить порядок перемножения подматриц, чтобы полностью избавить алгоритм от рекурсивных вызовов.

# Список литературы


[1] Albrecht Martin, Bard Gregory. — The M4RI Library. — The M4RI Team, 2019. — Access mode: `https://bitbucket.org/malb/m4ri`.

[2] Albrecht MR, Bard GV, Hart W. Efficient multiplication of dense matrices over GF (2) // arXiv preprint arXiv:0811.1714. — 2008.

[3] Biological sequence analysis / Richard Durbin, Sean Eddy, Anders Krogh, G Mitchison. — Cambridge University Press, 1996.

[4] Dowell Robin D, Eddy Sean R. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction // BMC bioinformatics. — 2004. — Vol. 5, no. 1. — P. 71.

[5] Grigorev. Semyon, Lunina. Polina. The Composition of Dense Neural Networks and Formal Grammars for Secondary Structure Analysis // Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3: BIOINFORMATICS, / INSTICC. — SciTePress, 2019. — P. 234–241.

[6] Hopcroft John E, Ullman Jeffrey D. Formal languages and their relation to automata. — 1969.

[7] JetBrains Programming Languages and Tools Lab [Электронный ресурс]. — Access mode: `https://research.jetbrains.org/groups/plt_lab` (online; accessed: 05.05.2019).

[8] Kasami Tadao. An efficient recognition and syntax-analysis algorithm for context-free languages // Coordinated Science Laboratory Report no. R-257. — 1966.

[9] Knudsen Bjarne, Hein Jotun. RNA secondary structure prediction using stochastic context-free grammars and evolutionary history. // Bioinformatics (Oxford, England). — 1999. — Vol. 15, no. 6. — P. 446–454.

[10] Liu Tong, Schmidt Bertil. Parallel RNA secondary structure prediction using stochastic context-free grammars // Concurrency and Computation: Practice and Experience. — 2005. — Vol. 17, no. 14. — P. 1669–1685.

[11] Nvidia CUDA. Nvidia cuda c programming guide // Nvidia Corporation. — 2011. — Vol. 120, no. 18. — P. 8.

[12] Okhotin Alexander. Parsing by Matrix Multiplication Generalized to Boolean Grammars // Theor. Comput. Sci. — 2014. — Jan. — Vol. 516. — P. 101–120. — Access mode: http://dx.doi.org/10.1016/j.tcs.2013.09.011.

[13] On economical construction of the transitive closure of an oriented graph / Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, IgorAleksandrovich Faradzhev // Doklady Akademii Nauk / Russian Academy of Sciences. — Vol. 194. — 1970. — P. 487–488.

[14] Rivas Elena, Eddy Sean R. The language of RNA: a formal grammar that includes pseudoknots // Bioinformatics. — 2000. — Vol. 16, no. 4. — P. 334–340.

[15] Valiant Leslie G. General Context-free Recognition in Less Than Cubic Time // J. Comput. Syst. Sci. — 1975. — Apr. — Vol. 10, no. 2. — P. 308–315. — Access mode: http://dx.doi.org/10.1016/S0022-0000(75)80046-8.

[16] Younger Daniel H. Context-free Language Processing in Time N3 // Proceedings of the 7th Annual Symposium on Switching and Automata Theory (Swat 1966). — SWAT '66. — Washington, DC, USA : IEEE Computer Society, 1966. — P. 7–20. — Access mode: https://doi.org/10.1109/SWAT.1966.7.

[17] Zier-Vogel Ryan, Domaratzki Michael. RNA pseudoknot prediction through stochastic conjunctive grammars // Computability in Europe 2013. Informal Proceedings. — 2013. — P. 80–89.

[18] Явейн А. Разработка алгоритма синтаксического анализа через умножение матриц. — Access mode: https://github.com/YaccConstructor/articles/blob/master/InProgress/Yaveyn_alg/jbalg.pdf (online; accessed: 13.05.2018).