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

Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

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

Yuliya Susanina

Optimization of parsing by matrix multiplication

Graduation Thesis

Admitted for defence.
Head of the chair:
Professor Andrey Terekhov

Scientific supervisor:
Assistant Professor Semyon Grigorev

Reviewer:

Saint-Petersburg
2019

Оглавление

Введение

Теория формальных языков активно изучается и находит широкое применение во многих областях [], прежде всего, для описания 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 КС-грамматики [], что позволяет свести проблему распознавания и классификации к задаче синтаксического анализа (определения принадлежности некоторой строки к языку, заданному грамматикой). Часто необходимо не просто проверить выводимость конкретной строки, но и найти всех подстроки, принадлежаних некоторому формальному языку.

Большинство подходов к анализу биологических цепочек, которые основаны на синтаксическом анализе, сталкиваются с одной и той же проблемой: низкая производительность. Чаще всего в них применяется алгоритм CYK [], который работает за кубическое время и неэффективен на длинных строках и для больших грамматик. Необходимым требованием таких областей применения, 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 дргих классов грамматик [].

На данный момент одним из самых быстрых алгоритмов, работающих с любой КС-грамматикой, является алгоритм Валианта []. Также данный алгоритм можно легко расширить для конъюнктивных и булевых грамматик, которые обладают большей выразительностью, чем КС-грамматики []. Однако в связи с сложностью применения к выше упомянутой задаче поиска всех подстрок и отсутсвие эффективной реализации алгоритм Валианта достаточно редко используется на практике несмотря на широкие теоретические возможности.

В лаборатории языковых инструментов Jetbrains, СПбГУ, предложен алгоритм, являющийся модификацией алгоритма Валианта, который обладает определенными преимуществами, способными частично решить указанные выше проблемы: легкость адаптации к задаче поиска подстрок и возможность повысить использование GPGPU и параллельных вычислений.

1. Постановка задачи

Целью данной работы является исследование алгоритма Явейн, являющегося модификацией алгоритма Валианта. Для её достижения были поставлены следующие задачи.

- Изучить алгоритм Явейн.
- Проанализировать эффективность применения этого алгоритма и алгоритма Валианта к задаче поиска подстрок.
- Доказать корректность алгоритма Явейн и дать оценку сложности.
- Реализовать последовательную и параллельную версии алгоритмов Валианта и Явейн.
- 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: алгоритм Валианта и алгоритм Явейн. Также мы отметим области применения данных алгоритмов и остановимся на биоинформатике, в частности, задаче поиска подстрок.

2.1. Терминология

Алфавитом Σ будем называть некоторое конечное множество символов. Σ^* — это множество всех конечных строк над алфавитом Σ.

Контекстно-свободная (КС) грамматика — четверка (Σ, N, R, S), где Σ — 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 $G_S = (\Sigma, N, R, S)$ называется грамматикой в нормальной форме Хомского, если все ее правила имеют одну из следующих форм: $A \to BC$, $A \to a$, или $S \to \varepsilon$, где $A, B, C \in N, a \in \Sigma$ и ε — пустая строка.

$L_G(A) = \{\omega | A \xrightarrow{*} \omega\}$ — язык, порождаемый грамматикой $G_A = (\Sigma, N, R, A)$, где $A \xrightarrow{*} \omega$ означает, что ω 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 анализ, которым на вход обычно подается грамматика в нормальной форме Хомского $G_S = (\Sigma, N, R, S)$ 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 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 $P_{i,j} = \bigcup\limits_{k=i+1}^{j-1} T_{i,k} \times T_{k,j}$ 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 $\bigcup\limits_{k=i+1}^{j-1} T_{i,k} \times T_{k,j}$ 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 $Z_{i,j} = \bigcup\limits_{k=1}^{l} X_{i,k} \times Y_{k,j}$.

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 $O(|G| BMM(n) log(n))$ для строки длины n, где $BMM(n)$ is количество операций, необходимое для перемножения двух булевых матриц размера $n \times n$.

Все элементы матриц T и P заполняются с помощью двух рекурсивных процедур. Процедура $compute(l, m)$ корректно заполняет $T_{i,j}$ для всех $l \leq i < j < m$. Процедура $complete(l, m, l', m')$ вычисляет $T_{i,j}$ для всех $l \leq i < m$, $l' \leq j < m'$. Предполагается, что $T_{i,j}$ для всех $l \leq i < j < m, l' \leq i < j < m'$ уже корректно заполнены и $P_{i,j} = \{(B, C) | \exists k, (m \leq k < l'), a_{i+1} \dots a_k \in L(B), a_{k+1} \dots a_j \in L(C)\}$ 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 $n + 1 = 2^k$

main():
compute(0, n + 1);
accept if and only if $S \in T_{0,n}$

compute(l, m):
if $m - l \ge 4$ then
    compute(l, $\frac{l+m}{2}$);
    compute($\frac{l+m}{2}$, m)
end
complete(l, $\frac{l+m}{2}$, $\frac{l+m}{2}$, m)

complete(l, m, l', m'):
if $m - l = 4$ *and* $m = l'$ then $T_{l,l+1} = \{A | A \to a_{l+1} \in R\}$;
else if $m - l = 1$ *and* $m < l'$ then $T_{l,l'} = f(P_{l,l'})$;
else if $m - l > 1$ then
    $leftgrounded = (l, \frac{l+m}{2}, \frac{l+m}{2}, m), rightgrounded = (l', \frac{l'+m'}{2}, \frac{l'+m'}{2}, m')$,
    $bottom = (\frac{l+m}{2}, m, l', \frac{l'+m'}{2}), left = (l, \frac{l+m}{2}, l', \frac{l'+m'}{2})$,
    $right = (\frac{l+m}{2}, m, \frac{l'+m'}{2}, m'), top = (l, \frac{l+m}{2}, \frac{l'+m'}{2}, m')$;
    complete(bottom);
    $P_{left} = P_{left} \cup (T_{leftgrounded} \times T_{bottom})$;
    complete(left);
    $P_{right} = P_{right} \cup (T_{bottom} \times T_{rightgrounded})$;
    complete(right);
    $P_{top} = P_{top} \cup (T_{leftgrounded} \times T_{right})$;
    $P_{top} = P_{top} \cup (T_{left} \times T_{rightgrounded})$;
    complete(top)
end

2.3. Алгоритм Явейн

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

Заполнение слоев происходит 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

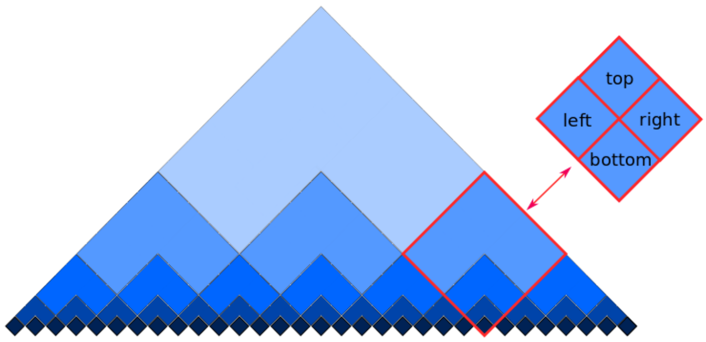

Рис. 1: Деление матриц на V-образные слои

на слои показан на рис. 1. Заметим, что каждую матрицу слоя можно обрабатывать независимо, что позволяет упростить разработку параллельной версии алгоритма.

Функция *main()* 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 $l \leq i < m$, $l' \leq j < m'$.

Процедура *completeLayer(M)* также принимает массив матриц 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*$_2$*)* — вспомагательная 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.

Алгоритм 2: Модификация алгоритма Валианта

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 $P_m = P_m \cup (T_{m1} \times T_{m2})$;

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 часть подходов для анализа генетических последовательностей основаны на синтаксическом анализе. Главным недостатком таких подходов являются существенные проблемы с производительностью [], которые можно решить с помощью алгоритма Валианта.

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

3. Доказательство корректности и оценка сложности

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

Лемма 1. *Если completeLayer(M') с выполненными 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 корректно заполненные $T_{i,j}$ *для всех* $l1 \leq i \leq m1$ *и* $l2 \leq j \leq m2$ *для всех* $(l1, m1, l2, m2) \in M'$ *для любого слоя* M' *и 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 $T_{i,j}$ *для всех* $l \leq i \leq m$ *и* $l' \leq j \leq m'$ *для всех* $(l, m, l', m') \in M$*.*

Доказательство. Сначала *performMultiplications(multiplicationTask*$_1$*)* добавит к каждому $\mathrm{P}_{i,j}$ 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 $(i, j) \in leftsublayer(M)$ and (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)* $\cup$ *rightsublayer(M))* и она вернет корректно заполненные *leftsublayer(M)* $\cup$ *rightsublayer(M)*.

Далее *performMultiplications* вызванная от аргументов *multiplicationTask*$_2$ и *multiplicationTask*$_3$ добавит все пары (B, C), такие что $\exists k$, $(\frac{l+m}{2} \leq k < m)$, $a_{i+1} \ldots a_k \in L_G(B)$, $a_{k+1} \ldots a_j \in L_G(C)$ и (B, C), такие что $\exists k$, $(l' \leq k < \frac{l'+m'}{2})$, $a_{i+1} \ldots a_k \in L_G(B)$, $a_{k+1} \ldots a_j \in L_G(C)$ к каждому $\mathrm{P}_{i,j}$ 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 $T[i, j]$ $\forall(i, j) \in M$ заполены корректно.

□

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

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

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

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

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

<u>База индукции:</u> $m - l = 1$. Необходимо вычислить всего один элемент и $P_{l,l'} = \{(B, C) | a_{l+1} \ldots a_{l'} \in L(B)L(C)\}$. Алгоритм вычисляет $f(P[l, l']) = \{A | a_{l+1} \ldots a_{l'} \in L(A)\}$ и $T[l, l']$ теперь заполнена корректно.

<u>Индукционный переход:</u> 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 *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). *bottomsublayer(M)* теперь заполнен и можно вызвать *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. $\square$

Теорема 2. *Алгоритм из листинга 2 корректно заполняет $T_{i,j}$ для всех i и j, и входная строка $a = a_1 a_2 \ldots a_n \in L_G(S)$ 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 1×1 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 в строки 4-6 листинга 2 возвращают все $T_{i,j} = \{A | A \in N, a_{i+1} \ldots a_j \in L_G(A)\}$.

$\square$

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

- *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.

Б͟а͟з͟а͟ ͟и͟н͟д͟у͟к͟ц͟и͟и͟:͟ 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 $i \in \{1, .., j\}$.

Рассмотрим $i = j + 1$.

Заметим, что функция *costructLayer(i)* возвращает $2^{p-i} - 1$ matrices of size 2^i, то есть в вызове процедуры *completeVLayer(costructLayer(k - i)) costructLayer(k - i)* вернет $2^i - 1$ матриц размера 2^{p-i}. Также, *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 $\sum_{n=1}^{calls_{i-1}} |M|$ is $2^{2(i-1)-1} - 2^{(i-1)-1}$, и можем доказать второе утверждение: посчитаем количество перемножений матриц размера $2^{p-i} \times 2^{p-i}$. *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, приведенному Охотиным []. □

Таким образом, мы доказали корректность алгоритма Явейн, а также показали, что его сложность осталась прежней.

4. Анализ эффективности применения к задаче поиска подстрок

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, $(m-2) = p-$. 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.

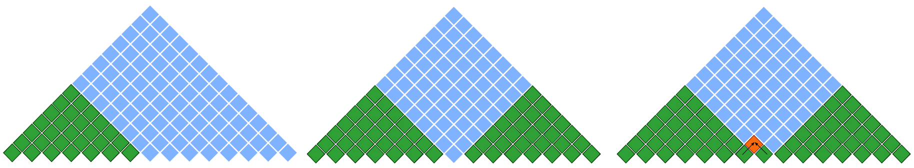

Рис. 2: Количество элементов, вычисляемых в алгоритме Валианта (2 треугольные подматрицы размера $\frac{n}{2}$).

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

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

5. Реализация

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 плотными матрицами — M4RI []. Данная библиотека была выбрана, так как там реализован один их наиболее эффективных способов перемножения булевых матриц — метод четырех русских.

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 []. 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 модификации.

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, ???? cores.

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

- Сравнение алгоритмов Валианта и Явейн.
- Эффективность применения алгоритма Явейн к задаче поиска подстрок.

Для ответа на первый вопрос был проведен сравнительный анализ как последовательной, так и параллельной версий реализации.

При исследвании последовательной версии алгоритмы были протестированы на грамматике Дика для двух типов скобок. Она представлена на листинге 1. Данная граммматика была выбрана потому, что грамматики для описания правильных скобочных последовательностей применяются при анализе строк в биоинформатике.

```
s : s s | (s) | [s] | ϵ
```

Листинг 1: Грамматика $D2$

Для случая параллельной версии алгоритма мы использовали реальную грамматику, применяющуюся в некоторых работах по биоинформатике []. Она представлена на листинге 2.

```
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*

6.1. Сранительный анализ для последоватльной версии

6.2. Сранительный анализ для параллельной версии

6.3. Примененимость к задаче поиска подстрок

s	n	Valiant(ms)	Modification(ms)
250	1023	4858	2996
	2047	19613	6647
	4095	78361	13825
	8191	315677	28904
510	2047	19613	12178
	4095	78361	26576
	8191	315677	56703
1020	4095	78361	48314
	8191	315677	108382
2040	8191	315677	197324

7. Результаты

В данной работе были получены следующие результаты.

- Изучен алгоритм Явейн, являющийся модификацией алгоритма Валинта, основной особенностью которого является ...
- Проведен анализ, который показал, что алгоритм Явейн лучше оригинальной версии алгоритма применительно к задаче поиска подстрок.
- Доказана корректность и дана оценка вычислительной сложности, которая составляет $\mathcal{O}(BMM(n)log(n))$.
- Реализованы последовательная и параллельная версии алгоритмов ..
- Проведено экспериментальное исследование алгоритма, показавшее эффективность модификации: последовательная версия реализации модификации не уступает ..; параллельная выигрывает на строка большой длины..; эффективно применима к задаче поиска подстрок ..

Список литературы