Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Системное программирование

Ковалев Дмитрий Александрович

Синтаксический анализ данных, представленных в виде контекстно-свободной грамматики

Выпускная квалификационная работа

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

Рецензент:
программист НИУ ИТМО Авдюхин Д. А.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Dmitrii Kovalev

Parsing data represented as context-free grammar

Graduation Project

Scientific supervisor:
associate professor Semyon Grigorev

Reviewer:
Programmer at ITMO University Avdiukhin Dmitrii

Saint-Petersburg
2017

Оглавление

Введение

Контекстно-свободные грамматики, наряду с регулярными выражениями, активно используются для решения задач, связанных с разработкой формальных языков и синтаксических анализаторов. Одним из основных достоинств контекстно-свободных грамматик является возможность задания широкого класса языков при сохранении относительной компактности представления. Благодаря данному свойству, грамматики также представляют интерес в такой области информатики, как кодирование и сжатие данных. В частности, существует ряд алгоритмов, позволяющих производить сжатие текстовой информации, используя в качестве конечного [7] или промежуточного [2] представления контекстно-свободную грамматику (grammar-based compression).

Стандартной процедурой при работе с текстовыми данными является поиск в них определенных шаблонов, которые могут быть заданы строкой или регулярным выражением. В настоящее время большие объемы информации, как правило, хранятся и передаются по сети в сжатом виде, поэтому актуальной задачей становится поиск шаблонов непосредственно в компактном контекстно-свободном представлении текста. Такой подход позволяет избежать дополнительных затрат памяти на восстановление исходной формы данных и в некоторых случаях увеличивает скорость выполнения запроса. Шаблон здесь может быть, как и при поиске в обычном тексте, строкой (compressed pattern matching), сжатой строкой (fully compressed pattern matching) или регулярным выражением.

Известны ситуации, в которых для задания шаблона необходимо использовать более выразительные средства. Примером может служить одна из задач биоинформатики — поиск определенных подпоследовательностей в геноме организма. Так, для классификации и исследования образцов, полученных в результате процедуры секвенирования, в них могут искать гены, описывающие специфические рРНК. Структура таких генов, как правило, задается при помощи контекстно-свободной грамматики [8]. Для уменьшения объемов памяти, необходимых для

хранения большого количества геномов, используются различные алгоритмы сжатия, в том числе основанные на получении контекстно-свободной структуры исходных последовательностей [3].

Задача поиска КС-шаблонов при использовании КС-представления данных формулируется следующим образом: необходимо найти все строки, принадлежащие пересечению двух языков, один из которых задается грамматикой шаблона, а второй представляет собой язык всех подстрок исходного множества строк, описываемого грамматикой, полученной в результате сжатия данных. Назовем такой поиск *синтаксическим анализом данных, представленных в виде КС-грамматики.* В общем случае задача неразрешима, так как сводится к задаче о проверке пересечения двух языков, порождаемых произвольными КС-грамматиками, на пустоту [5]. Для постановки экспериментов в области биоинформатики необходимо точнее исследовать возможность проведения синтаксического анализа КС-представления и разработать прототип алгоритма, позволяющего решить данную задачу.

1. Постановка задачи

Целью данной работы является разработка алгоритма синтаксического анализа данных, представленных в виде контекстно-свободной грамматики. Для ее достижения были поставлены следующие задачи.

- Определить ограничения, при которых синтаксический анализ контекстно-свободного представления является разрешимой задачей.
- Разработать алгоритм синтаксического анализа КС-представления данных с учетом поставленных ограничений.
- Реализовать предложенный алгоритм.
- Провести экспериментальное исследование.

2. Обзор

В данной работе используется понятие *рекурсивного автомата* [11] — удобного представления произвольной контекстно-свободной грамматики. Описание этой абстракции приводится в первом параграфе обзора.

Предлагаемый в работе алгоритм основан на алгоритме синтаксического анализа регулярных множеств, который, в свою очередь, является модификацией алгоритма обобщенного синтаксического анализа Generalized LL (GLL, [9]). Об этих алгоритмах и о проекте, в рамках которого проведена разработка предложенного решения, также будет рассказано в обзоре.

2.1. Рекурсивные автоматы и КС-грамматики

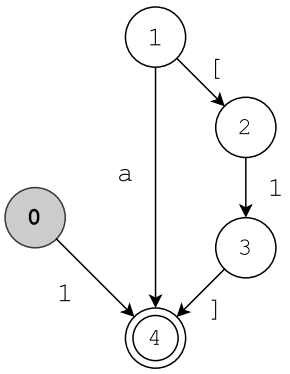
Введем понятие рекурсивного автомата, которое потребуется для дальнейшего изложения.

Определение 1. Рекурсивный автомат R — это кортеж $(\Sigma, Q, \delta, q_0, q_f)$, где Σ — конечное множество терминальных символов, Q — конечное множество состояний автомата, $\delta : Q \times (\Sigma \cup Q) \to 2^Q$ — функция переходов, $q_0 \in Q$ — начальное состояние, q_f — конечное состояние.

Можно заметить, что данное определение практически идентично определению стандартного конечного автомата. Единственное отличие состоит в том, что метками на ребрах рекурсивного автомата могут быть как терминальные символы (терминальные переходы), так и состояния (нетерминальные переходы). Класс рекурсивных автоматов обладает такой же выразительностью, как и контекстно-свободные грамматики, т.е. позволяет описать любой контекстно-свободный язык. Более того, грамматика тривиальным образом может быть преобразована в рекурсивный автомат (обратное тоже верно) [11]. Пример рекурсивного автомата, построенного по грамматике, можно увидеть на рис. 1.

$$
\begin{aligned}
S' &::= S \\
S &::= \texttt{[}\ S\ \texttt{]} \\
S &::= \texttt{a}
\end{aligned}
$$

(a) Грамматика G_1



(b) Рекурсивный автомат для G_1

Рис. 1: КС-грамматика и эквивалентный ей рекурсивный автомат

2.2. GLL-алгоритм и его модификации

Классические алгоритмы нисходящего и восходящего синтаксического анализа предполагают использование грамматики, которая является в достаточной мере однозначной. В противном случае, управляющие таблицы анализаторов содержат конфликты, из-за чего нельзя гарантировать корректное поведение на любых входных данных. Для работы с сильно неоднозначными грамматиками используются алгоритмы *обобщенного синтаксического анализа*, которые позволяют рассмотреть все возможные пути разбора строки и построить соответствующие деревья вывода. Поиск шаблонов не требует наличия деревьев вывода, поэтому в дальнейшем алгоритмы синтаксического анализа рассматриваются только как механизм, позволяющий определить принадлежность строки языку.

2.2.1. Оригинальный GLL-алгоритм

Generalized LL (GLL) — алгоритм, обобщающий идеи нисходящего синтаксического анализа. GLL, в отличие от стандартных алгоритмов LL-класса, позволяет использовать для анализа произвольную контекстно-свободную грамматику, в том числе содержащую леворекурсивные правила. Вместе с тем, GLL наследует такие полезные свойства алгоритмов нисходящего анализа, как непосредственная связь с грамматикой и простота отладки и диагностики ошибок.

Для обработки неоднозначностей GLL разделяет стек анализатора на несколько ветвей, каждая из которых соответствует возможному пути разбора. При таком подходе необходимо компактное представление множества стеков, в качестве которого выступает Graph Structured Stack (GSS). В работе [1] была представлена модификация GSS, которая позволяет увеличить эффективность GLL-анализа. Вершины такого представления хранят в себе номер нетерминала и позицию в строке, с которой начался разбор подстроки, соответствующей ему. На ребрах хранятся позиции в грамматике (вида $X \to \alpha A \cdot \beta$), на которые необходимо вернуться после завершения разбора нетерминала.

Основной идеей GLL является использование *дескрипторов*, позволяющих полностью описывать состояние анализатора в текущий момент времени.

Определение 2. Дескриптор — это тройка (L, u, i), где:

- L — текущая позиция в грамматике вида $A \to \alpha \cdot \beta$;
- u — текущая вершина GSS;
- i — позиция во входном потоке.

В процессе работы поддерживается глобальная очередь дескрипторов. В начале каждого шага исполнения алгоритм берет следующий в очереди дескриптор и производит действия в зависимости от позиции в грамматике и текущего входного символа, передвигая соответствующие указатели. При наличии конфликтов в грамматике алгоритм добавляет дескрипторы для каждого возможного пути анализа в конец очереди.

2.2.2. Поддержка грамматик в EBNF

В работе [4] была описана модификация GLL, которая позволяет использовать грамматики, записанные в расширенной форме Бэкуса-Наура (EBNF). Грамматика такого вида трансформируется в соответствующий рекурсивный автомат, в котором затем минимизируется количество состояний. Синтаксический анализ производится без построения управляющих таблиц: алгоритм обходит рекурсивный автомат в

соответствии со входным потоком символов. При обработке текущего дескриптора (C_S, C_U, i), где C_S — вершина автомата (эквивалент позиции в грамматике), C_U — вершина GSS, i — позиция в строке, могут возникать следующие ситуации.

- C_S — финальное состояние. Показывает, что разбор текущего нетерминала завершен. Необходимо осуществить возврат из C_U по меткам на исходящих из нее ребрах.
- Присутствует нетерминальный переход из C_S. В данном случае необходимо начать разбор указанного нетерминала X. Для этого в GSS должна быть создана новая вершина (X, i), если таковой не было ранее, а текущая вершина автомата изменена на стартовую для X.
- Присутствует терминальный переход из C_S. Необходимо сравнить терминал на ребре автомата с текущим входным символом. Если они совпадают, то осуществить переход в вершину автомата, на которую указывает ребро, и передвинуть указатель в строке.

За счет уменьшения количества состояний в автомате удается достичь прироста в производительности по сравнению со стандартным GLL-алгоритмом.

2.2.3. Синтаксический анализ графов

Стандартными входными данными для алгоритмов синтаксического анализа являются линейные последовательности токенов. На основе GLL был разработан алгоритм, который позволяет производить синтаксический анализ регулярных множеств строк, представленных в виде конечного автомата (который, в свою очередь, является ориентированным графом с токенами на ребрах).

Поддержка нелинейного входа не потребовала существенных изменений в оригинальном алгоритме. Дескрипторы модифицированного алгоритма хранят номер вершины входного графа вместо позиции в

строке. Также, на шаге исполнения просматривается не единственный текущий символ, а множество символов на ребрах, исходящих из текущей вершины.

Производительность данного алгоритма, как и обычного GLL, может быть увеличена при помощи представления входной грамматики в виде рекурсивного автомата. В таком случае, алгоритм будет производить обход двух автоматов — рекурсивного и конечного. Ситуации, возникающие при обработке дескрипторов, не отличаются от описанных ранее ситуаций для линейного входа. Псевдокод данной модификации приведен в приложении. Рассматривается вариант без построения деревьев вывода, алгоритм возвращает длины корректных цепочек, порождаемых автоматом.

2.3. Проект YaccConstructor

YaccConstructor [13] — исследовательский проект лаборатории языковых инструментов JetBrains на математико-механическом факультете СПбГУ, направленный на исследования в области лексического и синтаксического анализа. Проект включает в себя одноименную модульную платформу для разработки лексических и синтаксических анализаторов, содержащую большое количество компонент: язык описания грамматик YARD [12], преобразования над грамматиками и др. Основным языком разработки является F#.

Ранее в рамках YaccConstructor были реализованы генераторы GLL-анализаторов, описание которых было приведено в данном обзоре.

3. Разрешимость задачи синтаксического анализа контекстно-свободного представления

Как было сказано ранее, задачу поиска шаблона, при условии, что и шаблон, и данные, в которых осуществляется поиск, представлены контекстно-свободными грамматиками, мы назовем синтаксическим анализом контекстно-свободного представления. В данном разделе определяются ограничения, при которых подобная задача разрешима.

Для доказательства предложений, сформулированных далее, используется следующая теорема [6].

***Теорема* 1** (Nederhof, Satta)**.** *Пусть G_1 — произвольная контекстно-свободная грамматика, G_2 — грамматика, которая не содержит непосредственной или скрытой рекурсии. Тогда проблема проверки пустоты пересечения языков, порождаемых данными грамматиками, относится к классу PSPACE-complete.*

Рассмотрим случай, когда грамматика данных задает ровно одну строку. Пусть G_t — произвольная КС-грамматика, задающая шаблоны для поиска, а G_d — КС-грамматика, которая не содержит непосредственной или скрытой рекурсии. $L(G_t)$ и $L(G_d)$ — языки, порождаемые грамматиками, при этом $L(G_d) = \{\omega\}$, где ω — исходные данные, к которым был применен алгоритм сжатия. Необходимо определить, существуют ли такие строки ω', что $\omega' \in L(G_t)$ и ω' — подстрока ω.

***Предложение* 1.** *При выполнении описанных условий задача синтаксического анализа КС-представления разрешима.*

Доказательство. Пользуясь эквивалентностью представлений, можно записать грамматику G_d в виде рекурсивного автомата R_d. Рассмотрим рекурсивный автомат $R_{i,\,j}$, полученный из R_d путем замены стартового состояния на $i \in Q(R_d)$ и назначения терминирующего (финального, из которого не могут быть совершены переходы) состояния $j \in Q(R_d)$. Такой автомат описывает грамматику, которая является представлением

некоторой подстроки ω. Рассмотрев все возможные пары i и j, получаем конечное множество грамматик, для каждой из которых необходимо проверить, содержится ли строка, порождаемая ей, в языке $L(G_t)$. Согласно теореме 1, такая проверка является разрешимой задачей и принадлежит к классу PSPACE-complete. □

Отдельно отметим, что для описанных процедур используется лишь исходный автомат, эквивалентный грамматике G_d. Условия задачи поиска шаблонов непосредственно в контекстно-свободном представлении, таким образом, выполняются. Верна также разрешимость более общей задачи.

Предложение 2. *Пусть грамматика G_d задает конечное множество строк $L(G_d) = \{\omega_1, \ldots, \omega_n\}$. Необходимо определить, существуют ли строки ω', для которых верно: $\omega' \in L(G_t)$ и ω' — подстрока одной из строк $\omega_i \in L(G_d)$. Данная задача разрешима и принадлежит классу PSPACE-complete.*

Доказательство. Как и в предыдущем доказательстве, используем запись грамматики в виде рекурсивного автомата R_d и рассмотрим автоматы $R_{i,j}$. В данном случае каждый из этих автоматов представляет собой грамматику, которая порождает некоторое конечное множество подстрок исходных строк из $L(G_d)$. Проверка пустоты пересечения такой грамматики с G_t также соответствует условиям теоремы 1. □

В случае, когда грамматика G_d представляет собой бесконечный регулярный язык (т.е. содержит левую и/или правую рекурсию), разрешимость задачи поиска шаблонов установить не удается. Подход, использованный ранее в доказательстве предложений, не может быть применен, так как части рекурсивного автомата, представляющего грамматику G_d, также могут содержать рекурсивные переходы, что выходит за рамки условия теоремы 1. Проверка разрешимости и определение класса сложности задачи проверки пустоты пересечения произвольной и регулярной КС-грамматик в настоящее время остаются открытыми проблемами [6].

4. Алгоритм синтаксического анализа контекстно-свободного представления

Разработанный алгоритм расширяет подход к синтаксическому анализу графов на основе GLL. На вход алгоритм принимает два рекурсивных автомата, RA_1 и RA_2, построенных по исходным грамматикам G_1 и G_2 соответственно, где G_1 — произвольная контекстно-свободная грамматика, а G_2, исходя из ограничений разрешимости задачи, — грамматика, не содержащая непосредственной или скрытой рекурсии.

Алгоритм производит обход автоматов, рассматривая три типа ситуаций (финальное состояние, терминальный/нетерминальный переходы) для каждого из их состояний. Автомат, состояние которого обрабатывается в данный момент, будем называть *основным*; другой автомат из пары назовем *побочным*.

Оригинальный алгоритм синтаксического анализа графов использует один GSS, необходимый для правильного обхода рекурсивного автомата, который является представлением эталонной грамматики. Для работы с двумя рекурсивными автоматами требуется поддерживать одновременно два стека, GSS_1 и GSS_2. При этом используется модификация GSS с измененной структурой вершин: здесь они представляют собой тройку (S, i, u), где:

- S — нетерминал;
- i — состояние побочного автомата;
- u — вершина стека побочного автомата.

Пара (i, u) описывает состояние обхода побочного автомата на момент начала разбора нетерминала S. В оригинальном алгоритме достаточно было сохранять в вершине GSS лишь состояние входного автомата, так как подразумевалось, что даный автомат относится к классу конечных и для его обхода не используется стек.

Дескрипторы, которые использует разработанный алгоритм, также отличаются от представленных ранее. Для описания процесса анализа в

определенный момент времени теперь необходимо указывать вершины GSS для обоих автоматов. Получаем дескриптор вида (S_1, S_2, u_1, u_2), где S_1, S_2 — состояния автоматов, а u_1, u_2 — вершины соответствующих стеков. Функции **add**, **create** и **pop** были изменены в соответствии с новыми определениями дескрипторов и вершин GSS.

Algorithm 1 Функции для работы с GSS и дескрипторами

```
function ADD(S_1, S_2, u_1, u_2)
    if ((S_1, S_2, u_1, u_2) ∉ U) then
        U.add(S_1, S_2, u_1, u_2)
        R.enqueue(S_1, S_2, u_1, u_2)
function CREATE(S_call, S_next, u, S_i, w)
    A ← Δ(S_call)
    if (∃ GSS node labeled (A, i, w)) then
        v ← GSS node labeled (A, i)
        if (there is no GSS edge from v to u labeled S_next) then
            add GSS edge from v to u labeled S_next
            for ((v, j) ∈ P) do
                if (S_next is a final state) then
                    POP(u, j, w)
                ADD(S_next, S_i, u, w)
    else
        v ← new GSS node labeled (A, i, w)
        create GSS edge from v to u labeled S_next
        ADD(S_call, S_i, v, w)
function POP(u, S_i, w)
    if ((u, i) ∉ P) then
        P.add(u, i)
        for all GSS edges (u, S, v) do
            if (S is a final state) then
                POP(v, S_i, w)
            ADD(S, S_i, v, w)
```

Обработка терминальных и нетерминальных переходов, а также финальных состояний автомата была вынесена из главного цикла алгоритма в отдельную функцию, которая вызывается для каждого из автоматов.

Algorithm 2 Функция для обработки состояния рекурсивного автомата

```
function HANDLESTATE(RA, context)
    S_1, S_2, u_1, u_2 ← get values for RA from the context
    if (S_1 is a final state) then
        POP(u_1, S_2, u_2)
    for each transition(S_1, label, S_next) do
        switch label do
            case Terminal(x)
                for each (input[i] -x-> input[k]) do
                    if (S_next is a final state) then
                        POP(u_1, k, u_2)
                    if S_next have multiple ingoing transitions then
                        ADD(S_next, k, u_1, u_2)
                    else
                        R.enqueue(S_next, k, u_1, u_2)
            case Nonterminal(S_call)
                CREATE(S_call, S_next, u1, S_2, u_2)
```

Основная управляющая функция алгоритма, содержащая цикл обработки дескрипторов, представлена в листинге 3. Необходимо отметить, что для поиска шаблонов-подстрок обход автомата RA_2 запускается из каждой его вершины. Соответствующие стартовые дескрипторы создаются перед входом в цикл.

Algorithm 3 Алгоритм синтаксического анализа КС-представления

function PARSE(RA_1, RA_2)
  for each $q \in RA_2.States$ **do**
    $v1, v2 \leftarrow (RA_1.StartState, q, \$), (q, RA_1.StartState, \$)$
    add $v1$ to GSS_1, $v2$ to GSS_2
    $R.enqueue(RA_1.StartState, q, v1, v2)$
  while $R \neq \emptyset$ **do**
    $currentContext \leftarrow R.dequeue()$
    HANDLESTATE($RA_1, currentContext$)
    HANDLESTATE($RA_2, currentContext$)
  $result \leftarrow \emptyset$
  for each $(u, i) \in P$ where $u = GSS_1.Root$ **do**
    $result.add(i)$
  if $result \neq \emptyset$ **then** return $result$
  else report failure

Результатом работы алгоритма являются пары вида (n_1, n_2), где n_1, n_2 — номера состояний автомата RA_2. Для каждой из таких пар выполняется следующее утверждение: $\exists\, \omega \in T^*$ такая, что $\omega \in L(G_1)$ и $\omega \in L(RA')$, где RA' — рекурсивный автомат, полученный из RA_2 путем замены начального и конечного состояний на n_1 и n_2 соответственно.

5. Экспериментальное исследование

Предложенный алгоритм был реализован на платформе .NET как часть исследовательского проекта YaccConstructor. Основным языком разработки являлся F#. Был переиспользован генератор рекурсивных автоматов, реализованный ранее в рамках работы [4]. Также были переиспользованы структуры данных для представления структурированного в виде графа стека.

Для проверки работоспособности и оценки производительности алгоритма был проведен ряд тестов. В качестве представления шаблонов для поиска была выбрана грамматика G_1, задающая язык правильных скобочных последовательностей. Входные данные, в которых осуществлялся поиск, создавались следующим образом.

1. Из символов латинского алфавита генерировалась последовательность определенной длины, содержащая ровно пять подстрок, удовлетворяющих грамматике G_1.

2. Последовательность сжималась в контекстно-свободную грамматику при помощи алгоритма Sequitur [7, 10].

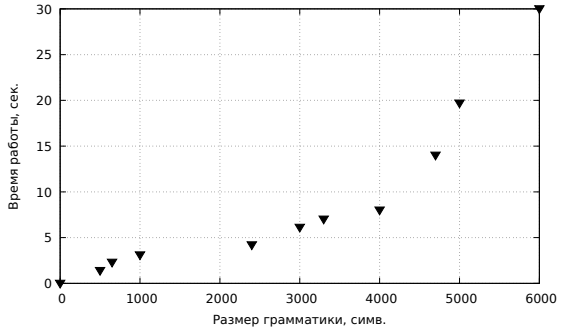
3. Описание полученной грамматики транслировалось в язык YARD, используемый в проекте YaccConstructor.

В каждом тесте из серии алгоритм корректно определял наличие шаблонов, получая пять пар вершин рекурсивного автомата, построенного по входной грамматике.

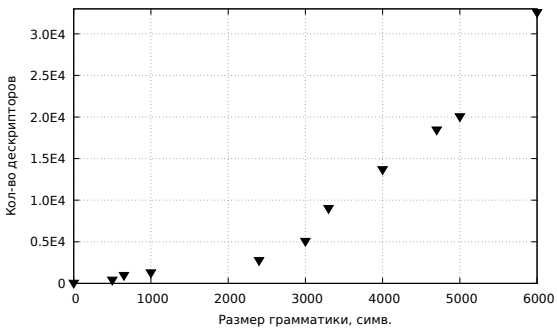
Оценка производительности заключалась в измерении времени работы алгоритма и количества создаваемых дескрипторов (рис. 2), а также объема используемой памяти, выраженного в количестве вершин и ребер GSS, построенных в процессе анализа (рис. 3). Тестирование проводилось при различных размерах входной грамматики данных. Размером грамматики $G = (\Sigma, N, P, S)$ назовем следующую величину:

$$|G| = |N| + \sum_{p \in P} length(p)$$

Для тестов использовался ПК со следующими характеристиками: MS Windows 10 x64, Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz 4 Cores, 16GB.

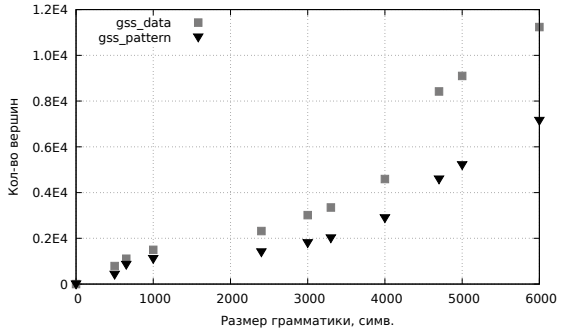


(a) Время работы

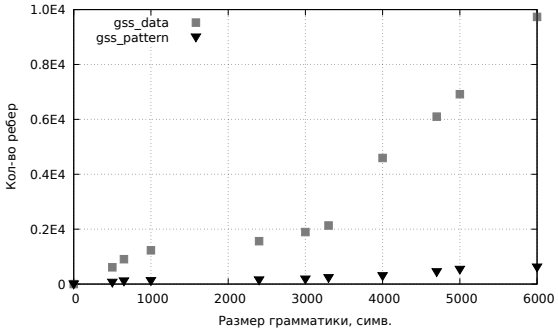


(b) Кол-во дескрипторов

Рис. 2: Зависимость времени работы и кол-ва дескрипторов от размера входной грамматики



(a) Кол-во вершин



(b) Кол-во ребер

Рис. 3: Зависимость размера построенных GSS от размера входной грамматики. Для обозначения GSS, соответствующего грамматике G_1, используется метка $gss\_pattern$, входной грамматике — gss_data

Результаты тестирования показывают, что даже в случае поиска простого шаблона время работы алгоритма быстро возрастает при увеличении размеров грамматики, представляющей данные. Предположительно, производительность может быть улучшена путем технической доработки используемых структур данных и самого алгоритма. Помимо этого, влияние на работу алгоритма может оказывать не только размер входной грамматики, но и ее структура, которая зависит от используемого алгоритма сжатия.

Заключение

В ходе данной работы получены следующие результаты.

- Определены ограничения, при которых синтаксический анализ контекстно-свободного представления является разрешимой задачей. Было показано, что грамматика, являющаяся представлением данных, должна порождать конечный язык.
- Разработан алгоритм синтаксического анализа KC-представления, учитывающий поставленные ограничения. Полученный алгоритм является модификацией алгоритма синтаксического анализа графов на основе GLL.
- Выполнена реализация алгоритма на языке программирования F# в рамках исследовательского проекта YaccConstructor.
- Проведено экспериментальное исследование: выполнено тестирование производительности на синтетических данных.

В дальнейшем планируется провести апробацию алгоритма на реальных данных — сжатом представлении ДНК организмов — и доказать теоретическую оценку сложности алгоритма.

Исходный код проекта YaccConstructor представлен на сайте `https://github.com/YaccConstructor/YaccConstructor`.

Список литературы

[1] Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // Compiler Construction: 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings / Ed. by Björn Franke. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. — P. 89–108. — ISBN: 978-3-662-46663-6. — URL: http://dx.doi.org/10.1007/978-3-662-46663-6_5.

[2] Arimura Mitsuharu. A grammar-based compression using a variation of Chomsky normal form of context free grammar // 2016 International Symposium on Information Theory and Its Applications (ISITA). — 2016.

[3] Gallé Matthias. Searching for Compact Hierarchical Structures in DNA by means of the Smallest Grammar Problem : Theses / Matthias Gallé ; Université Rennes 1. — 2011. — Feb. — URL: https://tel.archives-ouvertes.fr/tel-00595494.

[4] Gorokhov Artem, Grigorev Semyon. Extended Context-Free Grammars Parsing with Generalized LL. — 2017.

[5] Harrison M. A. Introduction to Formal Language Theory. — 1st edition. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1978. — ISBN: 0201029553.

[6] Nederhof Mark-Jan, Satta Giorgio. The Language Intersection Problem for Non-recursive Context-free Grammars // Inf. Comput. — 2004. — Aug. — Vol. 192, no. 2. — P. 172–184. — URL: http://dx.doi.org/10.1016/j.ic.2004.03.004.

[7] Nevill-Manning Craig G., Witten Ian H. Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm // J. Artif. Int. Res. — 1997. — Sep. — Vol. 7, no. 1. — P. 67–82. — URL: http://dl.acm.org/citation.cfm?id=1622776.1622780.

[8] Quantifying variances in comparative RNA secondary structure prediction / James WJ Anderson, Ádám Novák, Zsuzsanna Sükösd et al. // BMC Bioinformatics. — 2013. — Vol. 14, no. 1. — P. 149. — URL: http://dx.doi.org/10.1186/1471-2105-14-149.

[9] Scott Elizabeth, Johnstone Adrian. GLL parsing // Electronic Notes in Theoretical Computer Science. — 2009. — Vol. 253, no. 7.

[10] Sequitur [Электронный ресурс]. — URL: https://github.com/craignm/sequitur (online; accessed: 25.05.2017).

[11] Tellier Isabelle. Learning recursive automata from positive examples // Revue des Sciences et Technologies de l'Information-Série RIA: Revue d'Intelligence Artificielle. — 2006. — Vol. 20, no. 6. — P. 775–804.

[12] YARD [Электронный ресурс]. — URL: http://yaccconstructor.github.io/YaccConstructor/yard.html (online; accessed: 25.05.2017).

[13] Григорьев С. В. Синтаксический анализ динамически формируемых программ : Дисс... кандидата наук / С. В. Григорьев ; Санкт-Петербургский государственный университет. — 2015.

Приложение

Псевдокод алгоритма синтаксического анализа графов

Пусть (C_S, C_U, i, l) — текущий дескриптор, где C_S — состояние рекурсивного автомата, представляющего грамматику, C_U — вершина GSS, i — вершина входного графа, l — длина разобранной части строки. Для получения имени нетерминала грамматики, соответствующего состоянию автомата используется функция $\Delta : Q \to N$.

Во время работы алгоритма поддерживаются следующие множества: R — глобальная очередь дескрипторов, U — множество созданных ранее дескрипторов, P — множество, хранящее информацию о вызовах функции **pop**.

function ADD(S, u, i, l)
    if $((S, u, i, l) \notin U)$ **then**
        $U.add(S, u, i, l)$
        $R.enqueue(S, u, i, l)$

function CREATE$(S_{call}, S_{next}, u, i, l)$
    $A \leftarrow \Delta(S_{call})$
    if ($\exists$ GSS node labeled (A, i)) **then**
        $v \leftarrow$ GSS node labeled (A, i)
        if (there is no GSS edge from v to u labeled (S_{next}, l)) **then**
            add GSS edge from v to u labeled (S_{next}, l)
            for $((v, j, m) \in P)$ **do**
                if (S_{next} is a final state) **then**
                    POP$(u, j, (l + m))$
                ADD$(S_{next}, u, j, (l + m))$
    else
        $v \leftarrow$ **new** GSS node labeled (A, i)
        create GSS edge from v to u labeled (S_{next}, l)
        ADD$(S_{call}, v, i, 0)$

function POP(u, i, l)
  if $((u, i, l) \notin P)$ **then**
    $P.add(u, i, l)$
    for all GSS edges (u, S, m, v) **do**
      if (S is a final state) **then**
        POP$(v, i, (l + m))$
      ADD$(S, v, i, (l + m))$

function PARSE(RA, input)
  $GSSroot \leftarrow newGSSnode(RA.StartState, input.StartState)$
  $R.enqueue(RA.StartState, GSSroot, input.StartState, 0)$
  while $R \neq \varnothing$ **do**
    $(C_S, C_U, i, l) \leftarrow R.dequeue()$
    if $(l = 0)$ and (C_S is a final state) **then**
      POP$(C_U, i, 0)$
    for each $transition(C_S, label, S_{next})$ **do**
      switch $label$ **do**
        case $Terminal(x)$
          for each (input[i] $\xrightarrow{x}$ input[k]) **do**
            if (S_{next} is a final state) **then**
              POP$(C_U, k, (l + 1))$
            if S_{next} have multiple ingoing transitions **then**
              ADD$(S_{next}, C_U, k, (l + 1))$
            else
              $R.enqueue(S_{next}, C_U, k, (l + 1))$
        case $Nonterminal(S_{call})$
          CREATE$(S_{call}, S_{next}, C_U, i, l)$
  $result \leftarrow \emptyset$
  for each $(u, i, l) \in P$ where $u = GSSroot$, $i = input.FinalState$ **do**
    $result.add(l)$
  if $result \neq \emptyset$ **then** return $result$
  else report failure