Труды лаборатории языковых инструментов

Выпуск 5

Санкт-Петербург, 2017

Предисловие

Здесь будет аннотация

куратор лаборатории Д.Булычев

Оглавление

Поддержка расширенных контекстно-свободных грамматик в алгоритме синтаксического анализа Generalised LL

Горохов Артем Владимирович

Санкт-Петербургский государственный университет
gorohov.art@gmail.com

Аннотация Для автоматизации разработки синтаксических анализаторов часто используют генераторы, строящие анализаторы на основе спецификации языка. Обычно спецификация описывается неоднозначной грамматикой в расширенной форме Бэкуса-Наура (EBNF), которую не могут обрабатывать большинство инструментов без преобразования к форме Бэкуса-Наура (BNF). Существующие подходы анализа EBNF без преобразования к BNF не способны обрабатывать неоднозначные грамматики. С другой стороны, алгоритм Generalized LL (GLL) допускает произвольные контекстно-свободные грамматики и достигает высокой производительности, но не способен обрабатывать EBNF грамматики. В данной работе описана модификация GLL алгоритма для обработки расширенных контекстно-свободных грамматик, эта форма родственна EBNF. Так же показано, что предложенный подход повышает производительность анализа по сравнению с алгоритмами использующими преобразование грамматик.

Введение

Статический анализ — это анализ программного кода без его исполнения. Он производится после синтаксического анализа на основе грамматики, описывающей синтаксис языка. Общеупотребимый способ описания синтаксиса языков программирования — грамматики в расширенной форме Бэкуса-Наура (EBNF) [29]. С одной стороны, эта форма проста для понимания людей, с другой, достаточно формальна и допускает автоматизированное создание синтаксических анализаторов. Примером могут служить спецификации языков C, $C++$, *Java* и т.д.

Проблема в том, что существующие генераторы анализаторов, такие как ANTLR [1], Bison [5], преобразуют грамматики в форму EBNF для упрощения их структуры. В результате этого, представление кода строится на основе грамматики, отличной от заданной, что затрудняет обработку результатов анализа. С другой стороны, производительность таких алгоритмов анализа, как Generalised LL (GLL) [22] зависит от структуры грамматики, а её упрощение часто ведёт к избыточности представления, что отрицательно сказывается на производительности.

Алгоритм GLL показывает высокую производительность и способен работать с неоднозначными грамматиками. Предполагается, что поддержка в нём EBNF или родственных этой форме расширенных контекстно-свободных грамматик, увеличит производительность анализа для некоторых грамматик.

В данной работе предложен алгоритм синтаксического анализа, основанный на Generalised LL, для работы с расширенными контекстно-свободными грамматиками без преобразований. Показанно, что для некоторых грамматик алгоритм более производителен, чем существующие вариации GLL. В разделе 1 представлен обзор предметной области и литературы. Далее в разделе 2 представлены необходимые изменения в Generalised LL алгоритме и сопутствующих структурах данных. В разделе 3 описаны пути применения описанного ал-

горитма в задаче анализа регулярных множеств. Эксперементальное сравнение полученного алгоритма с современными модификациями Generalised LL содержится в разделе 4.

1 Обзор предметной области

Заметим, что EBNF является стандартизированной формой для *расширенных контекстно-свободных грамматик.*

Определение 1 *Расширенная контекстно-свободная грамматика (ECFG) [12] — это кортеж* (N, Σ, P, S)*, где* N *и* Σ *конечные множества нетерминалов и терминалов соответственно,* $S \in N$ *является стартовым символом, а* P *(продукция) является отображением из* N *в регулярное выражение над алфавитом* $N \cup \Sigma$*.*

Существует широкий спектр методов анализа и алгоритмов [4, 7, 9–12, 15, 17], предназначенных для обработки ECFG. Детальный обзор результатов и задач в этой области представлены в работе [12]. Следует отметить, что большинство алгоритмов основано на методах LL-анализа [4, 8, 10] и LR-анализа [7, 15, 17], но они работают только с ограниченными подклассами расширенных контекстно-свободных грамматик — LL(k), LR(k). Таким образом, нет решения для обработки произвольных (в том числе неоднозначных) ECFG-грамматик.

ECFG-грамматики широко используется в качестве входного формата для генераторов синтаксических анализаторов, но классические алгоритмы синтаксического анализа часто требуют форму Бэкуса-Наура (BNF), в продукциях которой допускаются лишь последовательности из терминалов и нетерминалов. Возможно преобразование грамматик ECFG в форму BNF [11], но оно приводит к увеличению размера грамматики и изменению её структуры: при трансформации добавляются новые нетерминалы. В результате синтаксический анализатор строит дерево вывода относительно преобразованной грамматики, что затрудняет обработку результата анализа.

В настоящее время алгоритмы на основе LL(1)-анализа представляются наиболее практичными и обеспечивают лучшую диагностику ошибок по сравнению с LR-анализом, так как являются низходящими. Но некоторые грамматики не являются LL(k) (для любого k) и не могут быть использованы в LL(k) анализаторах. Другие проблемы для инструментов на основе LL — леворекурсивные грамматики и неоднозначности в грамматике, которые, вместе с предыдущим недостатком, усложняют создание анализаторов. Алгоритм Generalised LL, предложенный в [22], решает все эти проблемы: он обрабатывает произвольные CFG, в том числе неоднозначные и леворекурсивные. В общем случае временная и пространственная сложность GLL зависит кубически от размера входа. А для LL(1) грамматик, он демонстрирует линейную временную и пространственную сложность. Но он, как и другие современные алгоритмы, не допускает использования ECFG без предварительного преобразования к форме BNF.

Для увеличения производительности Generalised LL-алгоритма, была предложена поддержка лево-факторизованных грамматик в нём [24]. Алгоритм GLL обрабатывает все возможные ветви разбора строки по заданной грамматике, эти ветви описываются так называемыми дескрипторами, которые состоят из позиций в грамматике и во входе, корня построенного леса разбора и текущего узла стека. Из этого следует, что для уменьшения времени анализа и количества используемой памяти можно снизить количество дескрипторов для обработки. Один из путей для достижения этого — уменьшение размера грамматики (снижение количества различных позиций в ней). Этого можно достичь факторизацией грамматики. Пример факторизации показан на рис. 1: из грамматики G_0 в процессе факторизации получена грамматика G'_0. Этот пример рассмотрен в работе [24], и доказано, что для некоторых грамматик факторизация существенно увеличивает производительность алгоритма GLL.

$S ::= a\ a\ b\ c\ d \mid a\ a\ c\ d \mid a\ a\ c\ e \mid a\ a$

(a) Исходная грамматика G_0

$S ::= a\ a\ (b\ c\ d \mid c\ (d \mid e) \mid \varepsilon\ )$

(b) Факторизованная грамматика G'_0

Рис. 1: Пример факторизации грамматики

Одно из возможных применений обобщённого синтаксического анализа — синтаксический анализ регулярных множеств. Синтаксическим анализом регулярных множеств называют анализ строк, заданных всеми возможными путями в конечном автомате. Такая задача возникает в различных областях одна из которых — биоинформатика. В результате экспериментов получают данные о геномах организмов, которые представлены строками в конечном автомате (геномы это строки над алфавитом $\{A; C; G; T\}$). Этот автомат называется метагеномной сборкой.

Существует множество подходов к анализу и идентификации геномов. Один из них — поиск и сравнение участков таких структур как 16s рРНК, тРНК, так как по ним можно достаточно точно классифицировать организм, которому они принадлежат. Существуют такие инструменты, как READO [20] и HMMER [28], использующие скрытые цепи Маркова для поиска, Infernal [16], использующий ковариационные модели. Но они работают лишь с линейными цепочками генома — не объединёнными в конечный автомат, такое представление требует большого количества памяти и неэффективно. С другой стороны, инструмент Xander [30] использует композицию скрытых моделей Маркова и метагеномных сборок. Изъян данного инструмента в существенно более низкой точности результата в сравнении с инструментами, использующими ковариационные модели.

Другой подход разрабатывается в лаборатории JetBrains СПбГУ. Поиск производится непосредственно в метагеномной сборке по таким структурам как тРНК, 16s рРНК. Эти

структуры имеют некоторые общие свойства в строении, которые могут быть описаны контекстно-свободной грамматикой [18]. Таким образом, можно использовать синтаксический анализ регулярных множеств для поиска. Этот подход описан в работе [19], основан на алгоритме синтаксического анализа Generalised LL и был реализован в рамках проекта YaccConstructor [31]. В нашей работе будут использоваться результаты и данные из работы [19] для сравнения производительности полученного алгоритма.

2 Использование Generalised LL для обработки ECFG

В этом разделе описываются структуры и методы необходимые для использования расширенных контекстно-свободных грамматик в алгоритме синтаксического анализа Generalised LL, а так же необходимые изменения в алгоритме анализа.

2.1 Представление ECFG

Представление ECFG, наиболее подходящее для синтаксического анализа — рекурсивные автоматы (Recursive Automaton (RA) [26].

Определение 2 *Рекурсивный автомат R это кортеж $(\Sigma, Q, S, F, \delta)$, где Σ — конечное множество терминалов, Q - конечное множество состояний, $S \in Q$ — начальное состояние, $F \subseteq Q$ — множество конечных состояний, $\delta : Q \times (\Sigma \cup Q) \to Q$ — функция перехода.*

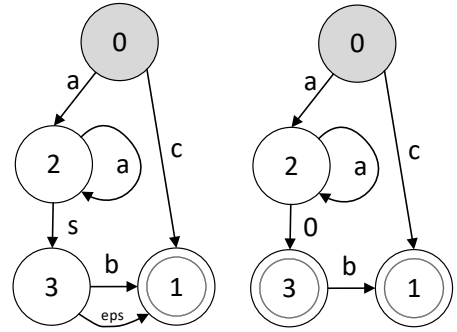
В рамках этой работы единственное различие между рекурсивным автоматом и общеизвестным конечным автоматом (FSA) состоит в том, что переходы в RA обозначаются либо терминалом (Σ), либо состоянием автомата (Q). Далее будем называть переходы помеченные элементами из Q *нетерминальными переходами*, а терминалами — *терминальными*

переходами. Нетерминальный переход в состояние q подразумевают построение вывода для некоторой подстроки начиная с текущей позиции во входе по этому нетерминалу и последующий разбор оставшейся подстроки начиная с состояния q.

Заметим, что позиции грамматики эквивалентны состояниям автомата, которые строятся из правых частей продукций грамматики. Правые части продукций ECFG являются регулярными выражениями над объединенным алфавитом терминалов и нетерминалов, поэтому построить по ним автомат можно использую общеизвестные алгоритмы. Следующий алгоритма строит RA с минимальным числом состояний для заданной ECFG.

- Построить конечный автомат, используя метод Томпсона [27] для правых частей продукций.
- Создать ассоциативный массив M из каждого нетерминала в соответствующее начальное состояние автомата. Этот массив должен оставаться консистентным на протяжение всех следующих шагов.
- Преобразовать автоматы из предыдущего шага в детерминированные без ε-переходов используя алгоритм, описанный в [3].
- Минимизировать детерминированный автомат, используя, например, алгоритм Джона Хопкрофта [13].
- Заменить нетерминальные переходы переходами по, стартовым состояниям автоматов, соответствующим данным нетерминалам, используя массив M. Результат этого шага — искомый рекурсивный автомат. Также используем M для определения функции $\Delta : Q \to N$ где N — имя нетерминала.

Пример преобразования ECFG в RA представлен на рис. 2, где состояние 0 — начальное состояние полученного RA.



$S ::= a^+ S\ b? \mid c$

(a) Грамматика G_1 (b) Конечный автомат для G_1 (c) Рекурсивный автомат R_1 для G_1

Рис. 2: Преобразование грамматики в рекурсивный автомат

2.2 Лес разбора для ECFG

Результатом синтаксического анализа является структурное представление входа — дерево или лес разбора в случае нескольких вариантов деревьев. Для начала, определим дерево вывода для рекурсивного автомата: это дерево, корень которого помечен начальным состоянием, листья терминалы или ε, а внутренние узлы нетерминалы N и их дети образуют последовательность заданную путём в автомате, который начинается в состоянии q_i, где $\Delta(q_i) = N$. Введём это определение более формально.

Определение 3 *Дерево вывода последовательности α для рекурсивного автомата $R = (\Sigma, Q, S, F, \delta)$ это дерево со следующими свойствами:*

- *корень помечен $\Delta(S)$;*
- *листья — терминалы $a \in (\Sigma \cup \varepsilon)$;*
- *остальные узлы — нетерминалы $A \in \Delta(Q)$;*
- *у узла с меткой $N_i = \Delta(q_i)$ существует:*

- *дети* $l_0 \dots l_n (l_i \in \Sigma \cup \Delta(Q))$ *тогда и только тогда, когда существует путь* p *в* R, $p = q_i \xrightarrow{l_0} q_{i+1} \xrightarrow{l_1} \dots \xrightarrow{l_n} q_m$, *где* $q_m \in F$, $l_i = \begin{cases} k_i, \ \textit{if } k_i \in \Sigma, \\ \Delta(k_i), \ \textit{if } k_i \in Q, \end{cases}$
- *только один ребенок помеченный* ε *тогда и только тогда, когда* $q_i \in F$.

Для произвольных грамматик RA может быть неоднозначным с точки зрения допустимых путей, поэтому можно получить несколько деревьев разбора для одной входной строки. Shared Packed Parse Forest (SPPF) [21] может использоваться как компактное представление всех возможных деревьев разбора. Будем использовать бинаризованную версию SPPF, предложенную в [25], для уменьшения потребления памяти и достижения кубической наихудшей временной и пространственной сложности. Бинаризованный SPPF может использоваться в GLL [23] и содержит следующие типы узлов (здесь i и j — начало и конец выведенной подстроки во входной строке):

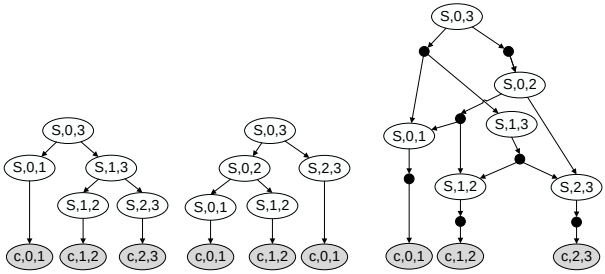
– упакованные узлы вида (S, k), где S состояние автомата, k — начало выведенной подстроки правого ребёнка; у упакованных узлов обязательно существует правый ребёнок — символьный узел, и опциональный левый — символьный или промежуточный узел;
– символьный узел помечен (X, i, j) где $X \in \Sigma \cup \Delta(Q) \cup \{\varepsilon\}$; терминальные символьные узлы $(X \in \Sigma \cup \{\varepsilon\})$ — листья; нетерминальные символьные узлы $(X \in \Delta(Q))$ могут иметь несколько упакованных детей;
– промежуточные узлы помечены (S, i, j), где S состояние в автомате, могут иметь несколько упакованных детей.

Дети символьных и промежуточных узлов — упакованные. Различные упакованные дети — различные варианты поддеревьев. Если у узла или его потомков более одного упакованного ребёнка, то он содержит несколько вариантов разбора

для подстроки. Промежуточные и упакованные узлы необходимы для бинаризации SPPF, что обеспечивает большее переиспользование узлов. Так, деревья, представленные на рис. 4a, объединяются в SPPF показанный на рис. 4b. Опишем мо-

$$S ::= S\ S \mid c$$

Рис. 3: Грамматика G_0



(a) Возможные деревья вывода (b) SPPF

Рис. 4: Пример для входа *ccc* и грамматики G_0

дификации исходных функций построения SPPF. Функция **getNodeT**(x, i), которая создает терминальные узлы, повторно используется без каких-либо модификаций из базового алгоритма. Чтобы обрабатывать недетерминизм в состояниях, определим функцию **getNodes**, которая проверяет, является ли следующее состояние RA финальным и в этом случае строит нетерминальный узел в дополнение к промежуточному. Она использует изменённую функцию **getNodeP**: вместо позиции в грамматики он принимает в качестве входных дан-

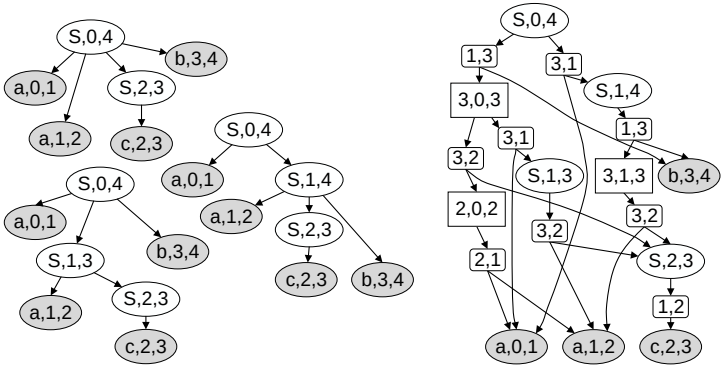
ных отдельно состояние RA и символ для нового узла SPPF: текущий нетерминал или следующее состояние RA.

function GETNODES(S, A, w, z)
  if (S is final state) **then**
    $x \leftarrow$ **getNodeP**(S, A, w, z)
  else $x \leftarrow \$$
  if ($w = \$$)& not (z is nonterminal node and its extents are equal) **then**
    $y \leftarrow z$
  else $y \leftarrow$ **getNodeP**(S, S, w, z)
  return (y, x)

function GETNODEP(S, L, w, z)
  $(\_, k, i) \leftarrow z$
  if $(w \neq \$)$ **then**
    $(\_, j, k) \leftarrow w$
    $y \leftarrow$ find or create SPPF node labelled (L, j, i)
    if ($\nexists$ child of y labelled (S, k)) **then**
      $y\prime \leftarrow$ **new** $packedNode(S, k)$
      $y\prime.addLeftChild(w)$
      $y\prime.addRightChild(z)$
      $y.addChild(y\prime)$
  else
    $y \leftarrow$ find or create SPPF node labelled (L, k, i)
    if ($\nexists$ child of y labelled (S, k)) **then**
      $y\prime \leftarrow$ **new** $packedNode(S, k)$
      $y\prime.addRightChild(z)$
      $y.addChild(y\prime)$
  return y

Рассмотрим пример SPPF для ECFG G_1, показанные на рис. 2a. Эта грамматика содержит конструкции (условное вхождение (?) и повторение (+)), которые должны быть преобразованы с использованием дополнительных нетерминалов для создания обычного GLL-анализатора. Предложенный генератор строит рекурсивный автомат R_1 (рис. 2c) и анализа-

тор для него. Возможные деревья ввода последовательности *aacb* показаны на рис. 5a. SPPF, созданный синтаксическим анализатором (рис. 5b), содержит в себе все три дерева.



(a) Возможные деревья вывода (b) SPPF

Рис. 5: Пример для входа *aacb* и автомата R_1

2.3 Алгоритм построения леса разбора для ECFG

В этом разделе описываются изменения в управляющих функциях базового алгоритма Generalised LL, необходимые для обработки ECFG. Основной цикл представленного алгоритма аналогичен базовому GLL: на каждом шаге основная функция **parse** извлекает из очереди так называемый дескриптор R — кортеж описывающий текущую ветку разбора. Пусть текущий дескриптор (C_S, C_U, i, C_N), где C_S — состояние RA, C_U — узел GSS, i — позицию во входной строке ω, C_N — узел SPPF. В ходе обработки дескриптора могут возникнуть следующие не исключающие друг друга ситуации.

- C_S — финальное состояние. Это возможно только если C_S — стартовое состояние текущего нетерминала. Следует построить нетерминальный узел с ребёнком (ε, i, i) и вызвать функцию **pop**, так как разбор нетерминала окончен.
- Существует терминальный переход $C_S \xrightarrow{\omega.[i]} q$. Во-первых, построить терминальный узел $t = (\omega.[i], i, i+1)$, далее вызвать функцию **getNodes** чтобы построить родителя для C_N и t. Функция **getNodes** возвращает кортеж (y, N), где N — опциональный нетерминальный узел. Создать дескриптор $(q, C_U, i+1, y)$ и, если в q ведёт несколько переходов, вызвать функцию **add** для этого дескриптора. Иначе поместить его в очередь вне зависимости от того был ли он создан до этого. Если $N \neq \$$, вызвать функцию **pop** для этого узла, состояния q и позиции во входе $i+1$.
- Существуют нетерминальные переходы из C_S. Это значит, что следует начать разбор нового нетерминала, поэтому должен быть создан новый узел GSS, если такового ещё нет. Для этого нужно вызвать функцию **create** для каждого такого перехода. Она осуществляет необходимые операции с GSS и проверяет наличие узла GSS для текущих нетерминала и позиции во входе.

Псевдокод для необходимых функций представлен ниже.

Функция **add** помещает в очередь дескриптор, если он не был создан до этого; эта функция не изменилась.

```
function CREATE(S_call, S_next, u, i, w)
  A ← Δ(S_call)
  if (∃ GSS node labeled (A, i)) then
    v ← GSS node labeled (A, i)
    if (there is no GSS edge from v to u labeled (S_next, w))
then
      add GSS edge from v to u labeled (S_next, w)
      for ((v, z) ∈ P) do
        (y, N) ← getNodes(S_next, u.nonterm, w, z)
        (_, _, h) ← y
        add(S_next, u, h, y)
```

```
        if N ≠ $ then
          (_, _, h) ← N; pop(u, h, N)
    else
      v ← new GSS node labeled (A, i)
      create GSS edge from v to u labeled (S_next, w)
      add(S_call, v, i, $)
    return v
  function POP(u, i, z)
    if ((u, z) ∉ P) then
      P.add(u, z)
      for all GSS edges (u, S, w, v) do
        (y, N) ← getNodes(S, v.nonterm, w, z)
        add(S, v, i, y)
        if N ≠ $ then pop(v, i, N)
  function PARSE
    GSSroot ← newGSSnode(StartNonterminal, 0)
    R.enqueue(StartState, GSSroot, 0, $)
    while R ≠ ∅ do
      (C_S, C_U, i, C_N) ← R.dequeue()
      if (C_N = $) and (C_S is final state) then
        eps ← getNodeT(ε, i)
        (_, N) ← getNodes(C_S, C_U.nonterm, $, eps)
        pop(C_U, i, N)
      for each transition(C_S, label, S_next) do
        switch label do
          case Terminal(x) where (x = input[i])
            T ← getNodeT(x, i)
            (y, N) ← getNodes(S_next, C_U.nonterm, C_N, T)
            if N ≠ $ then pop(C_U, i + 1, N)
            if S_next has multiple ingoing transitions then
              add(S_next, C_U, i + 1, y)
            else
              R.enqueue(S_next, C_U, i + 1, y)
          case Nonterminal(S_call)
```

create$(S_{call}, S_{next}, C_U, i, C_N)$
if SPPF node $(StartNonterminal, 0, input.length)$ exists **then**
return this node
else report failure

3 Синтаксический анализ регулярных множеств

Описанный в данной работе алгоритм можно применять для анализа регулярных множеств. При работе с конечным автоматом в качестве входных данных необходимо обрабатывать все переходы из текущей позиции (состояния) в автомате. Так, основная функция приобретает следующий вид:

function parseRegularSet
$GSSroot \leftarrow newGSSnode(StartNonterminal, StartState)$
$R.enqueue(StartState, GSSroot, StartState, \$)$
while $R \neq \varnothing$ **do**
$(C_S, C_U, i, C_N) \leftarrow R.dequeue()$
if $(C_N = \$)$ and (C_S is final state) **then**
$eps \leftarrow$ **getNodeT**(ε, i)
$(\_, N) \leftarrow$ **getNodes**$(C_S, C_U.nonterm, \$, eps)$
pop(C_U, i, N)
for each $transition(C_S, label, S_{next})$ **do**
switch *label* **do**
case $Terminal(x)$
for each (input[i] $\xrightarrow{x}$ input[k]) **do**
$T \leftarrow$ **getNodeT**(x, i)
$(y, N) \leftarrow$ **getNodes**$(S_{next}, C_U.nonterm, C_N, T)$
if $N \neq \$$ **then** **pop**(C_U, k, N)
add(S_{next}, C_U, k, y)
case $Nonterminal(S_{call})$
create$(S_{call}, S_{next}, C_U, i, C_N)$

if SPPF node $(StartNonterminal, StartState, \_)$ exists **then**
    return this node
else report failure

Позициями во входе для автомата становятся номера состояний и обрабатываются все исходящие переходы во входе. Кроме того, Функция **add** вызывается при обработке терминального перехода всегда, чтобы поддержать возможные циклы во входе. Например, для грамматики $S ::= a*$ и входного автомата на рис. 6 дескриптор будет создаваться бесконечно, если не добавить его в множество созданных, и алгоритм не остановится. Данное изменение не меняет теоретическую сложность алгоритма, но может сказаться на производительности в худшую сторону. Поэтому этот подход можно применять лишь только в случае присутствия циклов во ходе, иначе вызывать функцию **add** только при наличии нескольких входящих переходов в текущее состояние.

Рис. 6: Пример входа для грамматики $S ::= a^*$.

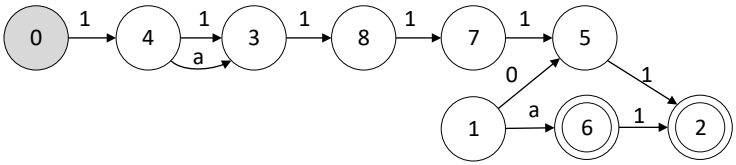
4 Эксперименты

В работе [24] было проведено экспериментальное сравнение алгоритма для факторизованных грамматик (Factorised GLL, FGLL) и базового GLL-алгоритма, продемонстрировавшее, что FGLL показывает большую производительность для грамматик в форме Бэкуса-Наура, которые могут быть факторизованы. Предполагается, что предложенная в данной работе

версия алгоритма продемонстрирует большую производительность, чем FGLL, для грамматик, имеющих эквивалентные позиции для алгоритма минимизации автомата, но различные после факторизации. Автомат, построенный для грамматики, в которой есть эквивалентные позиции, для которых алгоритм создаёт большое количество дескрипторов, объединит данные позиции, сократив тем самым множество создаваемых дескрипторов, что в свою очередь увеличит производительность. Примером такой ситуации может служить грамматика G_2 (рис. 7a), так как она содержит длинные последовательности в альтернативах, которые не сливаются при факторизации, но эквивалентны для алгоритма минимизации автомата. Рекурсивный автомат построенный для этой грамматики показан на рис. 7b.

$$S ::= K\ (K\ K\ K\ K\ K \mid a\ K\ K\ K\ K)$$
$$K ::= S\ K \mid a\ K \mid a$$

(a) Грамматика G_2



(b) Рекурсивный автомат для грамматики G_2

Рис. 7: Грамматика G_2 и RA для неё

Эксперименты проводились на входах различной длины, результаты приведены на рис. 8. Точные данные для входа a^{450} показаны в таблице 1.

Для экспериментов использовался ПК со следующими характеристиками: Microsoft Windows 10 Pro x64, Intel(R)

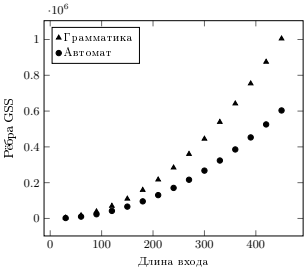
Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Cores, 4 Logical Processors, 16 GB.

	Время	Дескрипторы	Рёбра GSS	Узлы GSS	Узлы SPPF	Память, Мб
FGLL	10 мин. 13 с.	1104116	1004882	902	195 млн.	11818
RA	5 мин. 51 с.	803281	603472	902	120 млн.	8026
Ratio	43%	28%	40 %	0 %	39 %	33 %

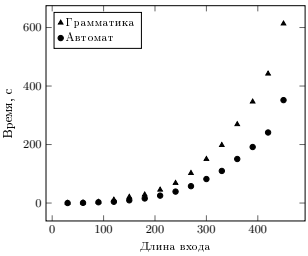
Таблица 1: Результаты экспериментов для входа a^{450}

Результаты данных экспериментов поддерживают предположение о том, что на некоторых грамматиках предложенный подход показывает результаты лучше FGLL. Для этого рекурсивного автомата анализатор создаёт меньше дескрипторов, чем для грамматики, так как цепочки нетерминалов K в альтернативах представлены единственным путём в автомате. Эта особенность ведёт к снижению количества узлов SPPF и размера GSS. В среднем, с грамматикой G_2 версия с минимизированными автоматами работает на 43% быстрее, использует на 28% меньше дескрипторов, на 40% меньше рёбер GSS, создаёт на 39% меньше узлов SPPF и использует на 33% меньше памяти.

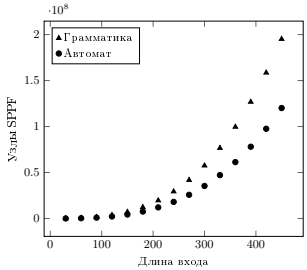
Было проведено экспериментальное сравнение разработанного алгоритма GLL с существующим в проекте YaccConstructor (основан на оригинальном алгоритме Generalised LL) в задаче поиска 16s рРНК в метагеномной сборке. Длинные рёбра сборки были предварительно отфильтрованы с помощью инструмента Infernal. В результате фильтрации сборка разбивается на компоненты, которые можно



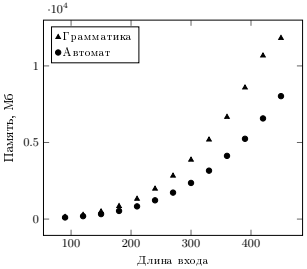
(a) Количество рёбер GSS.



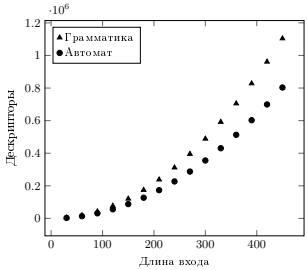
(b) Время работы.



(c) Количество узлов SPPF.



(d) Использование памяти



(e) Количество дескрипторов.

Рис. 8: Результаты экспериментов с грамматикой G_2.

анализировать независимо друг от друга. Тем не менее, предложенный ранее алгоритм не позволяет обработать некоторые компоненты, поэтому сравнение проводилось на остальных: 10 компонент с 400-100 состояний и переходов и 1118 компонент с менее чем 100 состояний и переходов. Результаты сравнения приведены в таблице 2 и показывают, что при работе с метагеномными сборками новый алгоритм, в среднем, использует на 65% меньше памяти и работает на 45% быстрее базового GLL. Сравнение с FGLL показывает на 4% меньшее использование памяти новым алгоритмом и на 10% меньшее время работы.

		GSS			
	Деск-ры	Рёбра	Узлы	Память	Время
GLL	802млн	414млн	339млн	20Гб	52 мин. 43 с.
FGLL	382млн	187млн	134млн	7Гб	29 мин. 27 с.
RA	362млн	190млн	134млн	6,8Гб	26 мин. 34 с.

Таблица 2: Результаты экспериментов с метагеномной сборкой

Заключение

В рамках данной работы была разработана и реализована модификация алгоритма GLL, работающая с расширенными контекстно-свободными грамматиками. Показано, что полученный алгоритм повышает производительность поиска структур, заданных с помощью контекстно-свободной грамматики в метагеномных сборках. Описанный алгоритм и генератор синтаксических анализаторов реализованы в расках проекта YaccConstructor на языке программирования F#. Исходный код доступен в репозитории проекта: `https://github.com/YaccConstructor/YaccConstructor`.

Одним из методов для описания семантики языка являются атрибутные грамматики, но они не поддержаны в описанном алгоритме. Опубликовано несколько работ о подклассе атрибутных ECFG (например [4]), тем не менее нет общего решения для произвольных ECFG. Таким образом, поддержка атрибутных расширенных контекстно-свободных грамматик и подсчёт семантики может быть дальнейшим развитием данной работы.

Список литературы

1. ANTLR Project website. — `http://www.antlr.org/`.
2. Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // International Conference on Compiler Construction / Springer. — 2015. — P. 89–108.
3. Aho Alfred V, Hopcroft John E. The design and analysis of computer algorithms. — Pearson Education India, 1974.
4. Alblas Henk, Schaap-Kruseman Joos. An attributed ELL (1)-parser generator // International Workshop on Compiler Construction / Springer. — 1990. — P. 208–209.
5. Bison Project website. — `https://www.gnu.org/software/bison/`.
6. Breveglieri Luca, Crespi Reghizzi Stefano, Morzenti Angelo. Shift-Reduce Parsers for Transition Networks // Language and Automata Theory and Applications: 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings / Ed. by Adrian-Horia Dediu, Carlos Martín-Vide, José-Luis Sierra-Rodríguez, Bianca Truthe. — Cham : Springer International Publishing, 2014. — P. 222–235. — ISBN: 978-3-319-04921-2. — Access mode: `http://dx.doi.org/10.1007/978-3-319-04921-2_18`.
7. Breveglieri Luca, Reghizzi Stefano Crespi, Morzenti Angelo. Shift-reduce parsers for transition networks // International Conference on Language and Automata Theory and Applications / Springer. — 2014. — P. 222–235.
8. Brüggemann-Klein Anne, Wood Derick. On predictive parsing and extended context-free grammars // International Conference on Implementation and Application of Automata / Springer. — 2002. — P. 239–247.

9. Bruggemann-Klein Anne, Wood Derick. The parsing of extended context-free grammars. — 2002.
10. Heckmann Reinhold. An efficient ELL (1)-parser generator // Acta Informatica. — 1986. — Vol. 23, no. 2. — P. 127–148.
11. Heilbrunner Stephan. On the definition of ELR (k) and ELL (k) grammars // Acta Informatica. — 1979. — Vol. 11, no. 2. — P. 169–176.
12. Hemerik Kees. Towards a Taxonomy for ECFG and RRPG Parsing // International Conference on Language and Automata Theory and Applications / Springer. — 2009. — P. 410–421.
13. An n log n algorithm for minimizing states in a finite automaton : Rep. / DTIC Document ; Executor: John Hopcroft : 1971.
14. Lee Gyung-Ok, Kim Do-Hyung. Characterization of extended LR (k) grammars // Information processing letters. — 1997. — Vol. 64, no. 2. — P. 75–82.
15. Morimoto Shin-ichi, Sassa Masataka. Yet another generation of LALR parsers for regular right part grammars // Acta informatica. — 2001. — Vol. 37, no. 9. — P. 671–697.
16. Nawrocki Eric P, Eddy Sean R. Infernal 1.1: 100-fold faster RNA homology searches // Bioinformatics. — 2013. — Vol. 29, no. 22. — P. 2933–2935.
17. Purdom Jr Paul Walton, Brown Cynthia A. Parsing extended LR (k) grammars // Acta Informatica. — 1981. — Vol. 15, no. 2. — P. 115–127.
18. 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.
19. Ragozina Anastasiya. GLL-based relaxed parsing of dynamically generated code : Master's Thesis / Anastasiya Ragozina ; SpBU. — 2016.
20. Reconstructing 16S rRNA genes in metagenomic data / Cheng Yuan, Jikai Lei, James Cole, Yanni Sun // Bioinformatics. — 2015. — Vol. 31, no. 12. — P. i35–i43.
21. Rekers Joan Gerard. Parser generation for interactive environments : Ph. D. thesis / Joan Gerard Rekers ; Universiteit van Amsterdam. — 1992.
22. Scott Elizabeth, Johnstone Adrian. GLL parsing // Electronic Notes in Theoretical Computer Science. — 2010. — Vol. 253, no. 7. — P. 177–189.

23. Scott Elizabeth, Johnstone Adrian. GLL parse-tree generation // Science of Computer Programming. — 2013. — Vol. 78, no. 10. — P. 1828–1844.
24. Scott Elizabeth, Johnstone Adrian. Structuring the GLL parsing algorithm for performance // Science of Computer Programming. — 2016. — Vol. 125. — P. 1–22.
25. Scott Elizabeth, Johnstone Adrian, Economopoulos Rob. BRNGLR: a cubic Tomita-style GLR parsing algorithm // Acta informatica. — 2007. — Vol. 44, no. 6. — P. 427–461.
26. 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.
27. Thompson Ken. Programming Techniques: Regular Expression Search Algorithm // Commun. ACM. — 1968. — Jun. — Vol. 11, no. 6. — P. 419–422. — Access mode: `http://doi.acm.org/10.1145/363347.363387`.
28. Wheeler Travis J, Eddy Sean R. nhmmer: DNA homology search with profile HMMs // Bioinformatics. — 2013. — P. btt403.
29. Wirth Niklaus. Extended Backus-Naur Form (EBNF) // ISO/IEC. — 1996. — Vol. 14977. — P. 2996.
30. Xander: employing a novel method for efficient gene-targeted metagenomic assembly / Qiong Wang, Jordan A. Fish, Mariah Gilman et al. // Microbiome. — 2015. — Vol. 3, no. 1. — P. 32. — Access mode: `http://dx.doi.org/10.1186/s40168-015-0093-6`.
31. YaccConstructor [Электронный ресурс]. — Режим доступа: `https://github.com/YaccConstructor/YaccConstructor` (дата обращения: 11.05.2015).