УДК 001.103:002

СИСТЕМА МОРФОЛОГИЧЕСКОГО АНАЛИЗА РУССКИХ СЛОВ

А. А. Седунов

Воронежский государственный университет

В данной работе рассматривается разработка структур данных, алгоритмов и объектно-ориентированных моделей, обеспечивающих реализацию процесса автоматического морфологического анализа, как одного из важных этапов комплексного анализа текста на естественном языке. Практическим результатом данной работы является программная система, в которой указанные модели реализованы для слов русского языка.

1. ВВЕДЕНИЕ

В данной работе рассматриваются основные алгоритмы морфологического анализа текстов, составленных на естественном языке. Задача морфологического анализа возникает в качестве одного из этапов обработки естественно-языковых данных в таких приложениях, как информационно-поисковые системы, машинный перевод, классификация документов.

Морфологический анализ — процесс, в ходе которого формы слова, обнаруженные в исходном тексте, ассоциируются с определенным набором грамматических атрибутов, частью речи и основной формой этого слова (т. н. леммой). Морфологический анализ не учитывает контекст и значение слова, поэтому на дано этапе некоторые разновидности неоднозначностей (например, омонимию) разрешить нельзя. Подобная неоднозначность устраняется более высокими уровнями анализа.

Данная работа основана на материалах разработки компьютерной системы морфологического анализа русских слов. Рассматриваются ключевые алгоритмы, составляющие основу информационной модели морфологического анализа. При описании алгоритмов применяется псевдокод, сопровождаемый текстовыми комментариями.

2. РЕАЛИЗАЦИЯ КОНЕЧНЫХ АВТОМАТОВ

Конечные автоматы представляют собой основную структуру данных, применяемую в процессе анализа [2]. Конечный автомат может быть представлен в виде ориентированного графа, вершинами которого являются элементы некоторого конечного множества состояний, а дуги имеют символьные метки и соответствуют переходу из одного состояния в другое при получении на входе символа, соответствующего метке дуги. Дуга <u, v> с меткой с будет обозначаться как <u|c|v>. В конечном автомате хранятся аннотированные слова, каждое из которых представляет собой сцепление некоторого слова, взятого из словаря, со строковым представление аннотации. Аннотация — это набор атрибутов, поставленных в соответствие данному слову. Подобный подход позволяет хранить атрибуты вместе с самими словами.

При реализации процедуры построения конечных автоматов за основу был взят инкрементальный алгоритм, описанный в [3, 4].

Собственно в процессе поиска наиболее важным является алгоритм обхода автомата. Данный алгоритм основан на модифицированном алгоритме обхода графа переходов в глубину. Используются два стека, обозначаемые как S и Path соответственно, а также буфер buffer, в котором накапливаются символы кодированной аннотации. Кроме того, для хранения получаемых в результате обхода аннотаций используется список ids. В стеке S хранятся пары < состояние, символ>, которые должны быть просмотрены на следующих шагах, а в стеке Path — состояния, из которых состоит текущий просматриваемый путь в графе переходов. Перед началом обхода в стек S помещается пара <начальное состояние, начальный символ>. Далее, пока стек не пуст, выполняются следующие действия:

1. Из стека S выталкивается очередное состояние и символ, после чего символ добавляется в конец буфера, а состояние — на вершину стека Path.

2. Далее рассматривается множество дуг, выходящих из текущего состояния.

а. Если это множество не пусто, то пары <конечное состояние, символ>, соответствую-

щие каждой исходящей дуге, добавляются в стек S

b. Если же множество исходящих дуг пусто, то обход текущего пути в графе переходов завершается. При этом в буфере находится строковое представление аннотации. Содержимое буфера декодируется, и полученные идентификаторы добавляются в список ids. Далее выполняется откат по текущему пути до ближайшего (в порядке обхода) разветвления, в котором есть еще не просмотренные пути. Для этого используется состояние u и символ k на вершине стека S. Из стека Path последовательно выталкиваются состояния с одновременным удалением символа из конца буфера, пока не будет получено такое состояние v, для которого существует дуга <v|k|u>, либо стек Path не будет исчерпан. В псевдокоде для этого используется условие v == ε, где ε — псевдосостояние, обозначающее фактическое отсутствие состояния и любых возможных переходов, связанных с ним (при реализации в качестве псевдосостояния используется число 0). Если v не совпадает с (т. е. разветвление найдено раньше, чем закончился маршрут), то состояние v возвращается в стек Path, так как новый путь также содержит это состояние

Псевдокод функции *traverse* приведен ниже:

```
function traverse(q, c0)
   ids := ∅; S := ∅; Path := ∅;
   buffer := "";
   S.push(<q, c0>);
   while (not S.isEmpty()) do
         <p, c> := S.pop();
         buffer := buffer + c;
         Path.push(p);
         if (∃(<p|k|s> ∈ E)) then
               for (<p|k|s> ∈ E) do
                     S.push(<s, k>);
         else
               ids.add(parseChain(buffer));
               if (not S.isEmpty()) then
                     <u, k> := S.top();
                     v := Path.pop();
                     while ((v ≠ ε) and (<v, k> ∈ D(δ))
and (δ(v, k) ≠ u)) do
                           buffer.removeLast();
                           v := Path.pop();
                     if (v ≠ ε) then
                           Path.push(v);
   return ids;
end;
```

Функция *traverse* представляет собой основу для функций более высокого уровня. Ниже представлен псевдокод функций *search* и *searchMaxPrefix*, которые используются соответственно при поиске полного слова и вторичном предсказании (в этом случае выполняется поиск всех слов, имеющих максимально возможный общий префикс с искомым словом). Именем DELIMITER назван атрибут алфавита ДКА, который обозначает разделитель, используемый при построении аннотаций.

```
function search(word)
  q := δ(δ*(q0, word), DELIMITER);
  if (q ≠ ε)
    return traverse(q, DELIMITER);
  return ∅;
end;

function searchMaxPrefix(word)
  n := max{i ∈ 1..n |
          <q0,word[1..i]> ≠ D(δ*)]};
  return traverse(δ*(q0, word[1..n]),
                  word[n]);
end;
```

3. МОРФОЛОГИЧЕСКИЙ АНАЛИЗ

Ниже рассматриваются алгоритмы поиска морфологических атрибутов, а также построение конечного автомата, позволяющего реали-

зовать предсказание структуры и грамматического значения слова, а также алгоритмы первичного и вторичного предсказания.

Функция `initPredictor` осуществляет построение конечного автомата, который используется в алгоритме вторичного предсказания, используя словарь `dic`. Построение выполняется путем перебора всех групп, хранящихся в данном словаре, имеющих счетчик использования, превышающий некоторое пороговое значение (в данной реализации 3) и относящихся к определенной части речи (в данной реализации используются существительные, прилагательные и глаголы). Для каждой такой группы выполняется построение всех возможных для нее словоформ. Далее от каждой словоформы отнимается суффикс, максимальная длина суффикса фиксирована и является параметром алгоритма предсказания (фактическая длина может быть меньше для коротких слов). Затем суффикс записывается в обратном порядке (в псевдокоде эта операция обозначена функцией `reverse`) и добавляется в конечный автомат. Аннотация формируется из номеров группы и словоформы.

Функция `firstPassPredict` осуществляет первичное предсказание, которое применяется в том случае, если точный эквивалент слова не был найден в словаре. Первичное предсказание выполняется в том предположении, что отняв от исходной словоформы некоторый префикс можно получить новую словоформу, которая (в отличие от исходной) присутствует в словаре. Максимальная длина отнимаемого префикса ограничивается соответствующим параметром алгоритма предсказания. В функции `firstPassPredict` последовательно отнимется по одному символу слева от исходной словоформы, после чего выполняется поиск полученной строки в основном автомате. Процесс останавливается, когда либо поиск строки в автомате дает положительный результат, либо превышается установленная длина префикса.

```
function firstPassPredict(s)
  forms := ∅;
  k := min{prefixLength + 1,
           |s| - suffixThreshold};
  for (i ∈ 1..k) do
    forms := mainDFA.
search(s[k..|s|]);
```

```
function initPredictor(dic)
   appended := ∅;
   for (group ∈ dic) do
        u := dic.getUsage(group.formPattern);
        if (u > usageThreshold) then
             for (form ∈ group.formPattern) do
                  pattern := form.pattern;
                  case pattern.partOfSpeech of
                       NOUN, ADJECTIVE, VERB:
                            s := form.prefix +
                                 form.base +
                                 form.inflection;
                            if (n >= suffixLength) then
                                 s := s[n − suffixLength.. n];
                            if (<s, pattern> ∉ appended) then
                                 ant := <group.formPattern,
form>;
                                 s := reverse(s);
                                 predictionDFA.addWord(s, ant);
                                 appended := appended ∪
                                      {<s, pattern>};
end;
```

```
    if (forms ≠ ∅) then
     break;
  return forms;
end;
```

Функция secondPassPredict реализует алгоритм вторичного предсказания, который используется в том случае, если первичное предсказание не дало результатов. Данный алгоритм основывается на поиске не самого слова, а его суффикса, причем в перевернутом виде (в силу того, левая граница суффикса не известна, а следовательно, сканирование строки в этом случае необходимо выполнять в противоположном направлении — от конца к началу). Для этой цели конструируется специальный автомат (predictionDFA). Для поиска исходная строка переворачивается ($s = <\overset{n}{\underset{i=1}{s_i}}> \rightarrow s' = <\overset{n}{\underset{i=1}{s_{n-i+1}}}>$) и затем выполняется поиск по критерию максимально соответствующего префикса (для исходной же строки этот префикс на самом деле будет являться перевернутым суффиксом).

```
function secondPassPredict(s)
  list := predictionDFA.searchMaxPre-
          fix(reverse(s));
  postprocess(list);
  return list;
end;
```

Собственно функция поиска объединяет обход основного автомата с алгоритмами первичного и вторичного предсказания:

```
function search(s)
  list := mainDFA.search(word);
    if (list = ∅) then
     list := firstPassPredict(word);
    if (list ≠ ∅) then
     list := secondPassPredict(word);
    return list;
end;
```

ЛИТЕРАТУРА

1. *Jackson P., Moulinier I* Natural Language Processing for Online Applications. — John Benjamins Publishing, 2002. — 237 pages.

2. Ахо А. В., Хопкрофт Дж., Ульман Д. Д. Структуры данных и алгоритмы.: Пер. с англ.: М.: Вильямс, 2003. — 384 с.: ил. — Пар. тит. англ.

3. *Daciuk J., Watson B.W., Mihov S., Watson R.E.* Incremental construction of minimal acyclic finite-state automata, Computational Linguistics. V. 26, № 1, P. 3—16, March 2000

4. *Carrasco R.C.* Incremental construction and maintenance of minimal finite-state automata. Computational Linguistics. V. 28, № 2, P. 207—216, June, 2002.

Статья принята к опубликованию
25 декабря 2006 г.