Алгоритм поиска шаблонов в тексте для большого количества правил представленных в виде конечных автоматов

Олоничев Сергей Александрович
компания Рамблер
olonichev@rambler-co.ru

Конечные автоматы успешно используются в различных областях вычислительной лингвистики. Основной причиной их успешного использования является возможность построения минимальных детерминированных автоматов. Однако препядствием к их использованию является размер детерминированных автоматов, который может быть экспоненциально больше чем размер соответствующих лингвистических правил и не помещаться в память современных компьютеров.

В данной работе мы предлагаем новый алгоритм для сопоставления текста с большим набором правил представленных в виде конечного автомата. Алгоритм основан на вычисление пересечения для автомата подстрок входного текста и экспоненциально меньшего, по сравнению с традиционным подходом, автомата лингвистических правил. Основным преимуществом нашего подхода является использование экспоненциально меньших автоматов для представления лингвистических правил. Результаты сравнения двух подходов, основанные на правилах снятия лексико-грамматической неоднозначности, показали что скорость поиска образцов в тексте выше при использование нашего метода чем при использовании традиционного метода, когда правила делятся на группы.

Introduction

Multiple pattern matching problem consists of finding the occurrences of a set of patterns $P=\{p_1,...,p_m\}$ in text. The patterns themselves are usually defined as strings or regular expressions. The first efficient multiple string pattern matching algorithm was suggested by A. Aho and M. Corasick [3]. This algorithms looks for all patterns simultaneously and has $O(N)$ complexity. Here and below N denotes the length of input text. Another even earlier approach for multiple string pattern matching is based on suffix trees. The main advantage of suffix trees is the ability to find the occurrences of the string in text in linear time with respect to the length of string. A suffix tree can be constructed in linear time $O(N)$ with respect to the length of text [4,5,6]. Hence, each string pattern $p \in P$ can be found with $O(|p|)$ time, where $|p|$ - is the length of the pattern.

Further generalizations in pattern matching algorithms led to the multiple pattern matching with patterns represented as regular expressions $P=\{re_1,...,re_m\}$.

Before making the definition of the regular expression pattern matching problem, lets make common denotations. T denotes an input text, a sequence of symbols over finite alphabet. $T[1, i]$ denotes the prefix of the text T of length i . $T[i, j]$,$i \le j$ denotes a sub-string of the length j - i. Σ denotes the finite alphabet of the input text, a set of all possible input symbols. L_Σ is the language of all arbitrary strings from Σ. $L_{Fact(T)}$ is the language of all possible sub-strings of the input text. L_P is the union of languages of all regular expression patterns $L_P \equiv L_{re1}|\cdots|L_{rem}$. The language of concatenation of L_Σ and L_P is denoted by $L_\Sigma \cdot L_P$. The finite state automaton [2] corresponding to the language L is denoted by $M(L)$.

The definition 1 defines the problem of regular expression pattern matching.

Definition 1 *The problem of regular expression pattern matching is to find a set of occurrences I, such that* $\forall i \in I$, *T[1, i]* $\in L_\Sigma \cdot L_P$, $i = 1 \ldots N$.

The standard solution of regular expression patterns matching problem is to construct Rabin-Scott automaton [2] for each pattern and add extra initial state with the alphabet Σ loop and draw epsilon transitions from the added state to all the initial states of all patterns. And then finally construct minimal deterministic automaton $M(L_\Sigma \cdot L_P)$. Using this automaton it is possible to solve the problem of matching regular expression patterns as it was stated above with $O(N)$ complexity. However, the resulting automaton may take $O(2^M)$ space, where M is the sum length of all regular expressions.

In practice it is not always necessary to find all occurrences of patterns but to find out whether there is at least one of them or to find the left-most longest match in the input text. Since these tasks seem to be less complex the automaton can still have $O(2^M)$ states.

In case when the number of patterns is large, for example these patterns represent linguistic rules for part of speech tagging or shallow parsing, it may not be possible to construct the minimal deterministic automaton by the scheme described above due to the huge number of states and transitions in the resulting automaton.

Several methods have been suggested to overcome this problem. The simplest one is splitting rules into parts and constructing minimal automaton for each part separately. The method suggested in [7] computes failure function similar to the one presented in Aho-Corasick algorithm [3]. But this algorithm seems work well for automatons with acyclic transition graphs only, and it produces too many additional states in case when automaton graph for P is cyclic. Another method suggested in [9] solves the problem stated above in sub-linear time, but requires $O(N \times \log N)$ preprocessing time and space, which makes it less attractive for static patterns and huge amount of text, as in the case of computational linguistics.

In this article we propose a regular expression matching algorithm which combines usage of $M(L_P)$ automaton instead of $M(L_\Sigma \cdot L_P)$ with factor automaton [8] constructed from the input text. We suggest two variants of the algorithm, the first modification uses non-deterministic factor automaton, the second uses minimal deterministic factor automaton. Both deterministic and non-deterministic factor automata can be efficiently constructed in $O(N)$ time and space. Thus, using our algorithms it is possible to solve the regular expression pattern-matching problem in almost linear time with significant benefits in memory usage.

The paper is organized as follows: section 1 describes algorithm and gives proof of its correctness, section 2 makes comparison of the described algorithm with the straight-forward implementation when rules are split into groups.

1. Algorithm description

Let us first remind the definition of factor automaton and outline some of its properties.

Definition 2 *The factor automaton $M(L_{Fact(T)})$ of the string T is an automaton accepting the language $L_{Fact(T)}$, such that* $\forall b \in L_{Fact(T)}$, $\exists (i,j)$, $i \le j$, $b = T[i, j]$.

The following properties of factor automata are important for our algorithm:

1. Each state of a non-deterministic factor automaton corresponds to the exactly one position in the input string T, see Figure 1.
2. Minimal deterministic factor automaton can be constructed in $O(N)$ time and space [8]. Non-deterministic factor automaton can be constructed in $O(1)$ time.

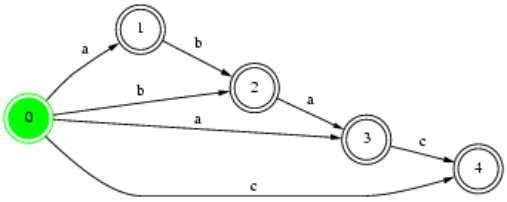

Figure 1: Non-deterministic factor automaton for the input text *T*="*abac*". Note: 0-state is initial; all automaton states are final.

The main idea of the algorithm is to calculate the intersection of the patterns automaton $M(L_P)$ and the factor automaton $M(L_{Fact(T)})$ finding a set of occurrences from the set of final states of the resulting intersection automaton. This can be described by the following equation:

$$\begin{aligned}
&(Q_1, i_1, F_1, \Sigma, \delta_1) \equiv M(L_{Fact(T)}), (Q_2, i_2, F_2, \Sigma, \delta_2) \equiv M(L_P) \\
&(Q, i, F, \Sigma, \delta) = M(L_{Fact(T)}) \cap M(L_P), \\
&i = i_1 \times i_2, Q \subseteq Q_1 \times Q_2, F \subseteq F_1 \times F_2, \delta = \delta_1 \cap \delta_2, \\
&I_O = Left[Q]
\end{aligned} \tag{1}$$

Where (Q, i, F, Σ, δ) is the resulting intersection automaton; Q is a set of states, i is an initial state, F is a set of final states; $Left[Q]$ returns a set of states from $M(L_{Fact(T)})$, the left part of Cartesian product. I_O is a set of occurrences, the ending positions of the matched patterns.

Theorem 1 *The set of occurrences I_O obtained by Equation 1 is a solution of the regular expression pattern matching problem of the Definition .*

Proof.

Suppose, $\exists\, k \notin I_O$, s.t. $T[1, k] \in L_\Sigma \cdot L_P \Rightarrow \$\ \exists\, a \in L_\Sigma$ and $\exists\, b \in L_P$, s.t. $T[1, k] \equiv a \cdot b$, and $b \notin L_{Fact(T)}$, this contradicts the definition of $M(L_{Fact(T)})$.

Suppose, $\exists\, \$f_1 \in I_O$, s.t. $T[1, f_1] \notin L_\Sigma \cdot L_P \Rightarrow \$ \exists\, (f_1, f_2) \in F$, where $f_1 \in F_1$, $f_2 \in F_2$, $\Rightarrow\ \exists\, b \in L_{Fact(T)}$ which is not a sub-string of T, this contradicts the definition of $M(L_{Fact(T)})$.

□

The complexity of the algorithm can be calculated as a sum of the complexities of:

1. $M(P)$ construction from a set of regular expressions P.
2. $M(L_{Fact(T)})$ construction from text T.
3. Intersection calculation.

For big static rule bases, which are usually the case of computational linguistic, the corresponding automaton construction requires to be done only once, so the first stage does not require any operations during the text processing stage.

The complexity of the factor automaton construction is $O(N)$ for a minimal deterministic automaton [8] and $O(1)$ for non-deterministic automaton. Indeed, in case of non-deterministic factor automaton, the text T does not require any preprocessing at all. The text itself can serve as implicit representation of the δ of $M(L_{Fact(T)})$.

The complexity of intersection of factor automaton with automaton constructed from patterns is $O(N^2)$ in general (note that factor automaton of the input string has number of states proportional to the input string length). However taking into account that $M(L_{Fact(T)})$ is acyclic and the fact that we do not require to calculate the entire intersection automaton but only a set of its final states the time complexity of this stage can be drastically reduced.

During the intersection calculation, we associate with each state $q \in M(L_{Fact(T)})$ a set of states from $M(L_P)$ reachable by all possible paths from the $M(L_{Fact(T)})$ coming into the q. These sets then can be iteratively calculated traversing the factor automaton in the topological order. The initial association is a pair of $(i_1, \{i_2\})$ with only one state in associated set.

These considerations lead us to the following pattern matching algorithm based on non-deterministic factor automaton, see Algorithm 1. The analogous algorithm based on minimal deterministic factor automaton is not listed here.

Algorithm 1 Calculates the set I_O according to the Equation 1

Require: T - input sequence of symbols of alphabet Σ, $\mathcal{M}(\mathcal{L}_P) \equiv (Q, I, F, \Sigma, \delta)$
Ensure: I_O, s.t. $\forall T[1,i] \in \mathcal{M}(\mathcal{L}_{\Sigma^*} \cdot \mathcal{L}_P), i \in I_O$

```
1:  I_O ← ∅
2:  for i = 1 to N do
3:     Set[i] ← δ(I, T[i])
4:  end for
5:  for i = 1 to N − 1 do
6:     for all q ∈ Set[i] do
7:        p ← δ(q, T[i + 1])
8:        if p ∈ F then
9:           I_O ∪= p
10:       end if
11:       Set[i + 1] ∪= p
12:    end for
13: end for
```

The first loop of the algorithm processes transitions of the initial state of the implicitly specified factor automaton $M(L_{Fact(T)})$. The second loop processes all the other states except the last one as it does not have any transitions, see Figure 1. These loops are separated here for the reason of clarity. It is possible to merge these loops making the algorithm online over the input text. Also there is no need to remember all sets during the processing, in online version only current and the following set should be kept. So the time complexity of Algorithm 1 is $O(|S|_{\max} N)$ and space complexity is $O(N)$, where $|S|_{\max}$ is the maximum set size, which is limited by N.

2. Practical results

We made the comparison of the proposed algorithm with the traditional approach when rules had to be split into groups and for each group a separate automaton was constructed in order to fit the memory.

For our experiments, we used rules for part-of-speech tagging in WRE notation [1]. Constructing $M(L_\Sigma \cdot L_P)$ automaton, it was necessary to split the rules into 5 groups. While we were able to construct $M(L_P)$ without any splitting.

The execution process consisted of the following stages: first we read the input sentence, then assigned a digital value to each word of the sentence and then matched the sequence of digits with the automaton/automata constructed from the rules. We made our tests on 7.8 Mb of input plain-text corpus. The cumulative times of execution at each stage are summarized in Table 1. Note: the first two algorithms require no preprocessing time; the third one requires constructing minimal deterministic factor automaton and re-enumerating its states in topological order.

Table 1: Comparison of algorithms : the first algorithm is based on rules-splitting, the second one uses non-deterministic factor automaton, the third one uses minimal deterministic factor automaton. The numbers are cumulative time given in seconds for processing 7.8 Mb of the input text.

Alg	Input/Output, sec	Digitizing, sec	Preprocessing, sec	Matching, sec
1.	25.78	30.00	30.00	31.41
2.	25.78	30.00	30.00	31.02
3.	25.78	30.00	31.67	32.57

The speed of text matching with $M(L_\Sigma \cdot L_P)$ split into 5 parts is 5.53 Mb/sec. While the speed of matching by intersection of automaton of rules $M(P)$ with the factor non-deterministic automaton is 7.64 Mb/sec.

As it was shown in the previous section, the maximum possible size of the set associated with $q \in M(L_{Fact(T)})$ is N. However, for the practical rule bases the average set size is much smaller than N and can be considered as a constant. For example for our part-of-speech tagging rules we used the average set size was about 3 which is much smaller in comparison to the size of the input text.

3. Conclusions

In this paper we presented a new approach for matching text over large rule bases represented as finite state machines. The main advantage of the algorithm is the possibility of usage of exponentially smaller automata for representation of rules. The results of comparison showed that matching speed can be higher using our approach instead of traditional, when rules are split into groups.

While the matching algorithm based on intersection of minimal deterministic factor automaton of the input text with automaton of rules showed poorer running time than algorithm based on intersection with non-deterministic factor automaton, we believe it can be improved by more accurate implementation of the factor automaton construction. Moreover, this algorithm can be significantly faster when there are many similar sub-strings in the large input text. This can be the case of DNA sequences where the input alphabet size is small [10].

References

1. 1) Cheusov A. The Word-based Regular Expressions in Computational Linguistics // Proceedings of the seventh International conference “Pattern recognition and information processing”, Minsk, 2003.
2. 2) Brauer W. Eine Enfuhrung in die Theorie endlicher Automaten // Stuttgart, 1984.
3. 3) Aho A., Corasick M. Efficient string matching: an aid to bibliographic search // Comm. ACM. 1975. Vol 18. pp. 333-340.
4. 4) Weiner P. Linear pattern matching algorithms // Proc. of the 14-th IEEE Symp. on Switching and Automata Theory. 1973. pp. 1-11.
5. 5) McCreight E.M. A space-economical suffix tree construction algorithm // J.ACM. 1976. Vol. 23. pp. 262-272.
6. 6) Ukkonen E. On-line construction of suffix-trees // Algorithmica. 1995. Vol. 14. pp. 249-260.
7. 7) Mohri M. String-matching with automata // Nordic Journal of Computing, 1995.
8. 8) Crochemore M. and Hancart C. Automata for matching patterns, 1996.
9. 9) Baeza-Yates R.A., Gonnet G.H. Fast text searching for regular expressions or automaton searching on tries // Journal of the ACM, Vol. 43, No. 6, November 1996.
10. 10) Gusfield D. Algorithms on Strings, Trees and Sequences. Computer science and computational biology, // Cambridge University Press, 1997.