Automata for matching patterns

2007

Tomáš Novotný

Automata for matching patterns

- Opening
- Algorithms for matching patterns
- Matching finite sets of words
- Matching words
- Suffix automata
- Conclusion

Opening

- Word pattern matching
- Problem – locating occurrences of a pattern in text file.
- Solution to the problem is basic part of many tools, editors; used in the analysis of biological sequences.
- Several method based on automata.

Algorithms for matching patterns

Let ***t*** by the searched word. An occurrence in ***t*** of pattern represented by the language *P* is a triple (u, p, v) where $\boldsymbol{u,v} \in A^*$, $p \in P$, and such that $\boldsymbol{t} = upv$.

- Pattern described by:
  - a word
  - finite set of words (P)
  - a regular expression (L(R))

Notation

- Pattern ***p*,** we denote length(**p**) **= *m***
- Text ***t*,** we denote length**(t)** = ***n***
- $P = \{y_1, y_2, \ldots, y_p\}$, where $y_p = y_{p,1} \ldots y_{p,\text{length}(p)} \in A^*$, where $p=1,\ldots,k$

Algorithms for matching patterns

- Naive brute force method
  - in time $O(m \times n)$
  - backing up in the text
- Optimizing the naive method

Basic idea

- preprocess the text or pattern to create DFA M that accepts the pattern or text
- do the search
  - Searching prefixes of $\boldsymbol{t}$ that belong to the language $\mathbf{A}^*\mathbf{P}$

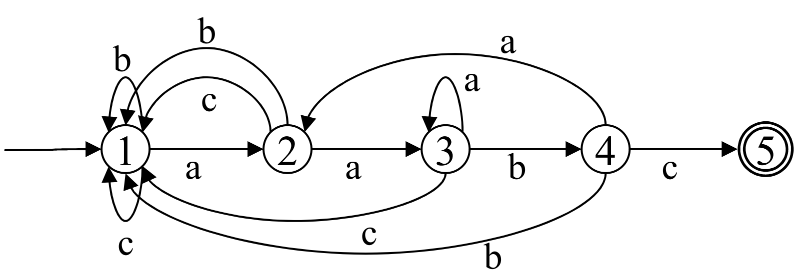
Automata for matching patterns

a b a a c a a b a a a
| : 1.a → a
a b a a c a a b a a a
| : 2.b → ø
a b a a c a a b a a a
| : 1.a → a
a b a a c a a b a a a
| : 2.a → aa
a b a a c a a b a a a
| : 3.c → ø
a b a a c a a b a a a
| : 1.a → a
a b a a c a a b a a a
| : 2.a → aa
a b a a c a a b a a a
| : 3.b → aab
a b a a c a a b c a a
| : 4.c → aabc

T = a b a a c a a b c a a
P = a a b c

- No backing up in the text

A = {a,b,c}



Matching finite set of words

Problem

Given a finite set of words **P,** the dictionary, preprocess it in order to locate words of **P** that occur in any word **t.**

- Solution by Aho an Corasick, 1975
- Implementation of complete DFA recognizing the language **A* P**.
- A preprocessing phase in $O(|P| \times \log card(A))$ time, where $|P| = |P_1|+\ldots+|P_m|$ and in $O(|P|)$ *space*
- A search phase in $O(|t| \times \log card(A))$ time, both with extra space $O(|P| \times card(A))$

Preprocessing phase

Definition

Let P be a finite language, than the automaton $M=(Q,A,q_0,\delta,F)$ recognizes the language **A*P**.

1. $Q = \{q_x \mid x \in Pref(P)\}$, $q_0 = q_\varepsilon$
2. $\delta(q_x,a) = q_{h_P(xa)}$, $x \in Pref(P)$, $a \in A$
3. $F = \{q_x \mid x \in Pref(P) \cap A^*P\}$

$h_P(v)$ = the longest suffix of **v** that belongs to Pref(P) for each $\mathbf{v} \in A^*$

- Searching automata SA = $(Q, A, q_0, \delta_{SA}, \varphi_{SA}, F)$ where δ_{SA}, φ_{SA} represents δ from M such that:
  - δ_{SA}, : $Q \times A \rightarrow Q \cup \{fail\}$ is goto function
  - φ_{SA}: $Q-\{q_0\} \rightarrow Q$ is failure function

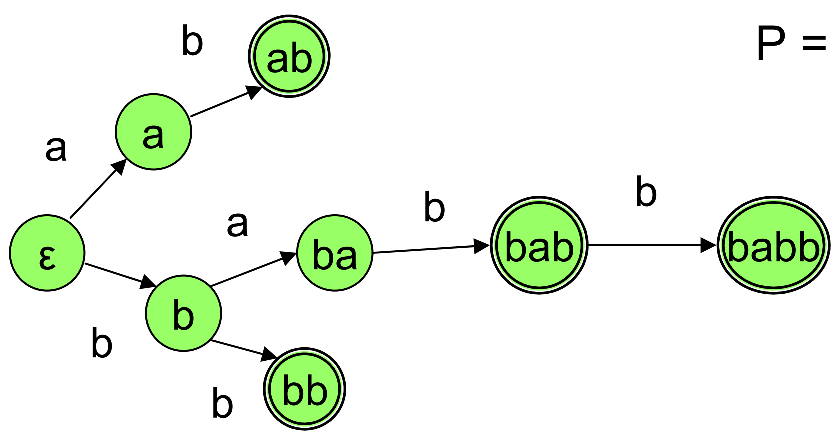
- Implementation of SA:
  1. Construct tree-like FA accepting language P
  2. Computing φ_{SA}

SA implementation

Construction of trie of a finite set of words P

- Input: finite set of words P, **a** $\in$ P, **q** $\in$ Q
- Output: DFA accepting set P, we denote by Trie(P)
- Method
  1. Q := $\{q_0\}$
  2. Create all possible states. Each new state corresponds to some prefix of one or more pattern. Define $\delta(q,a_{j+1}) = q'$, where q' corresponds to prefix $a_1a_2\ldots a_{j+1}$ of one or more patterns
  3. Define $\delta(q_0,a) = q_0$ for all a such that $\delta(q_0,a)$ was not defined in step 2
  4. $\delta(q,a)$ = fail for all a and q which $\delta(q,a)$ was not defined in step 2 or 3
  5. Each state corresponding to the complete pattern is the final state

Example



P = {ab,babb,bb}

SA implementation

Construction SA of a finite set of words P

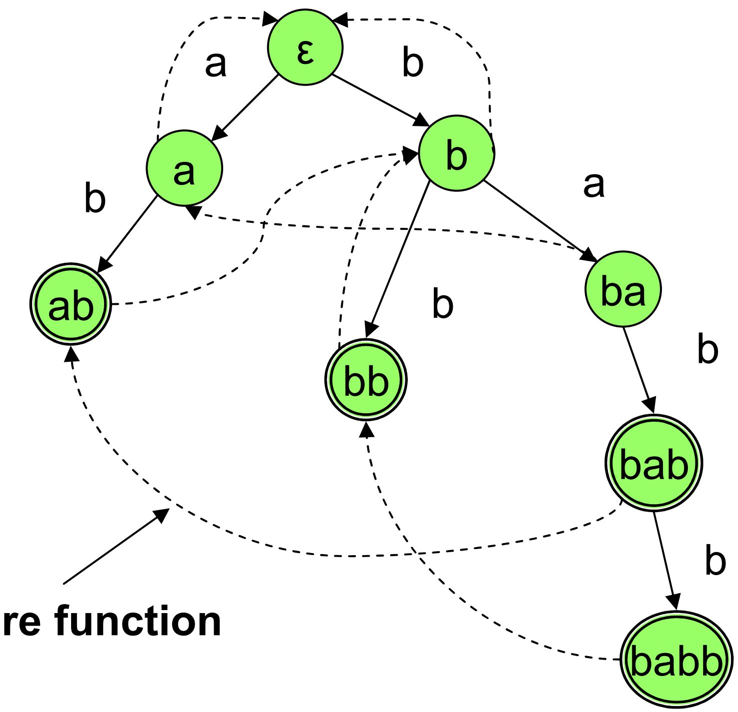
Input: Trie(P)

Output: DFA accepting set P with failure function, we denote by D(P)

$P = \{ab, babb, bb\}$

Method:

$\mu \leftarrow$ EMPTYQUEUE
ENQUEUE(μ, q_0)
while not QUEUEISEMPTY (μ)
  loop p $\leftarrow$ DEQUEUE (μ)
    for each letter *a* such that $\delta(p,a) \neq$ fail
      loop $q \leftarrow \delta(p,a)$
        $\varphi(q) \leftarrow \gamma(\varphi(p),a)$
        ENQUEUE(μ, q_0)



Failure function

We define $\gamma(p,a)$:

- $\delta(p,a)$ if $\delta(p,a)$ is defined
- $\gamma(\varphi(p),a)$ if $\delta(p,a)$ is undefined and $\varphi(p)$ is defined
- q_0 otherwise

Searching

We can locate words of **P** that occur in any word **t**

Input: automaton SA recognizing A* P
Output: Occurrences of words from P in t

Method
1. $p \leftarrow q_0$
2. **for** i:=1 to m do
3.   **while** $\delta(p,a_i) = \emptyset$
4. $p \leftarrow \varphi(p)$ // follow fail
5. $p \leftarrow \delta(p,a_i)$ // follow a goto
6.   **if** $p \in F$ **then** print I, print p // print position and patterns

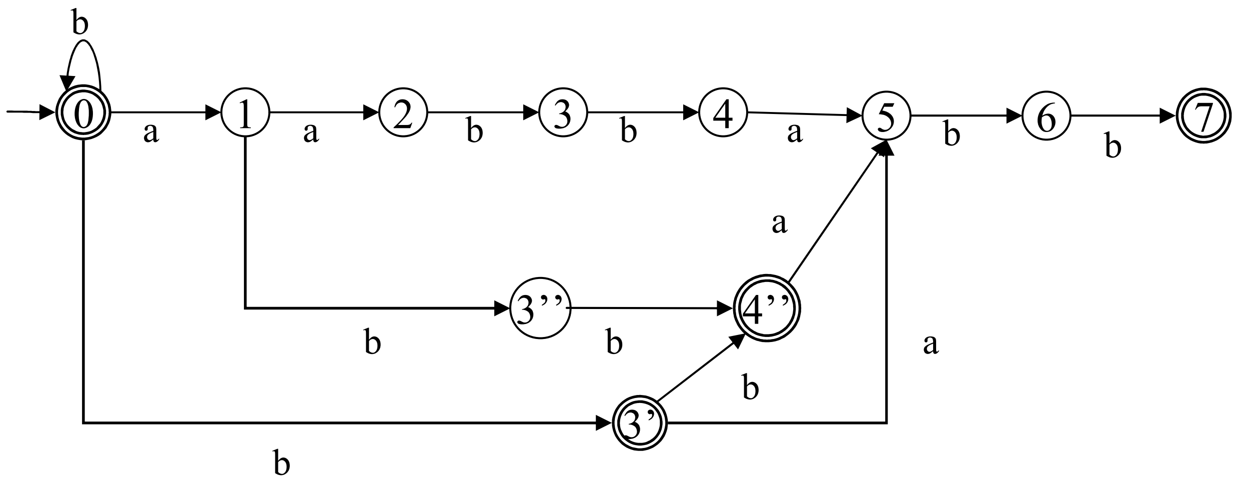
Matching word

Problem

Given a word **p** preprocess it in order to locate all its occurrences in any given word **t.**

- Particular case of previous problem – dictionary has one element
- Solution by Knuth, Morris and Pratt, 1977
- Automaton $M(A^* p)$ is minimal
- A preprocessing phase in $O(|p|)$ time
- A search phase in $O(|t|)$ time

Suffix automaton



The minimal DFA recognizing suffixes of aabbabb

Suffix automaton

- An alternative solution for string-matching problem
- Also used to search a word **p** for factors of **t**

Alternative data structures for storing the suffixes of a text

- Suffix tries – quadratic size in length of the word
- Suffix trees – compact representation of suffix tries
- Suffix automaton – minimization of suffix tries

Definition

Suffix automaton of a word **t** is defined as the minimal deterministic (not necessarily complete) automaton that recognize the finite set of suffixes of **t**.

We denote M(Suff(**t**))

Problem

Given a word **t** and preprocess it in order to locate all occurrences of any word **p** in **t.**

Suffix automaton - properties

- It can be constructed in O(n x log card(A)) time and O(n) space
- It allows to check whether a pattern occurs in a text in O(m) time
- It has linear size limited by the number of states, which is less than 2n-2; the number of transitions is less than 3n-4, where n>1
- Represents complete index of input text **t**
  - occurrences of different patterns can be found fast

Conclusion

- Matching patterns with automata
  - No backing up the searched text
  - We pay for preprocessing, but we have fast search
  - Improve the performance