String Matching

Module 3

AoA-Even 2021-22

Introduction

- Naïve String Matching Algorithm
- String Matching with Finite Automata
- Knuth Morris Pratt Algorithm

Naïve String Matching Algorithm

- String matching or pattern recognition is a problem for searching a pattern to be searched within a text under certain conditions and find out all occurrences of it.
- Pattern and text will be in form of an array of characters drawn from finite alphabet Σ.
- Pattern is denoted as $P[1\ldots m]$ and Text as $T[1\ldots n]$ where m and n are their respective length such that $n \geq m \geq 1$.
- If pattern P occurs in Text after $\boldsymbol{s}$ shifts then
  $\boldsymbol{P[1\ldots m] = T[s+1\ldots s+m]}$ where $\boldsymbol{n - m \geq s \geq 0}$.
- If P occurs after finite shift $\boldsymbol{s}$ in T, then we can say $\boldsymbol{s}$ is a valid shift, otherwise invalid shift

Naïve String Matching Algorithm

- Example

Position
s+1=3

Position
s + m=7

$\Sigma=\{a, b, c\}$

Text T

a	b	c	a	a	b	c	c	b	a	a	b

n = 12

Pattern P

s=2

c	a	a	b	c

m = 5

Naïve String Matching Algorithm

NAIVE-STRING-MATCHER(T, P)

$n = T.length$
$m = P.length$
for $s = 0$ **to** $n - m$
    if $P[1\,..\,m] == T[s+1\,..\,s+m]$
        print "Pattern occurs with shift" s

String Matching Algorithm

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\,|\Sigma|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

String-matching algorithms, their preprocessing and matching times

String Matching with Finite Automata

Finite Automata

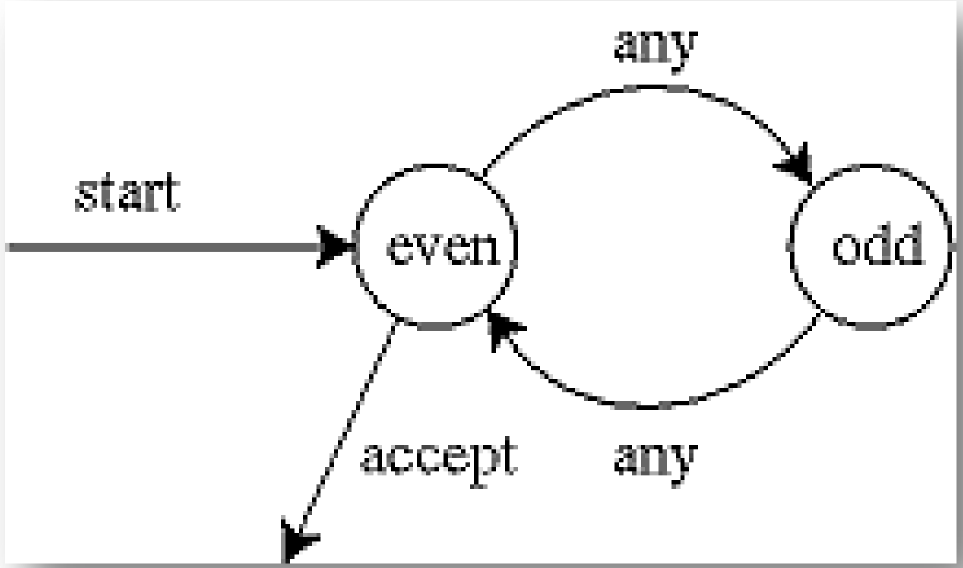
A finite automaton **M** is a 5-tuple $(Q, \Sigma, \delta, s, F)$:

- **Q**: the finite set of states
- **Σ** : the finite input alphabet
- **δ** : the "transition function of **M**" from $Q \times \Sigma$ to Q
- $\mathbf{s} \in \mathbf{Q}$: the start state
- $\mathbf{F} \subset \mathbf{Q}$: the set of final (accepting) states

String Matching with Finite Automata

How it works

A finite automaton accepts strings in a specific language. It begins in state $\mathbf{q_0}$ and reads characters one at a time from the input string. It makes transitions from state **q to** δ**(q,a)** based on these characters, and if when it reaches the end of the tape it is in one of the accept states, that string is accepted by the language.



String Matching with Finite Automata

How it works

- A finite automaton M induces a function ϕ, called **the final-state function** , from Σ^* to Q such that $\phi(w)$ is the state M ends up in after scanning the string w.
- Thus, M accepts a string w if and only if $\phi(w) \epsilon A$.
- We define the function ϕ recursively, using the transition function:

$$\boldsymbol{\phi(\varepsilon) = q_0,}$$
$$\boldsymbol{\phi(wa) = \delta(\phi(w), a) \; for \; w \epsilon \, \Sigma^*, \; a \, \epsilon \, \Sigma}$$

String Matching with Finite Automata

String-Matching Automata

- For a given pattern P, we construct a string-matching automaton in a preprocessing step before using it to search the text string.
- In order to specify the string-matching automaton corresponding to a given pattern P [1…m], we first define an auxiliary function σ, called **the suffix function** corresponding to P.
- A suffix function w.r.t. pattern P[1..m], s , is a mapping from **Σ*** to {0,1,...,m} such that σ(x) is the length of the longest prefix of P that is a suffix of x: $\sigma(x)=\max\{k: P_k \sqsupset x\}$.

String Matching with Finite Automata

The Suffix Function

In order to properly search for the string, the program must define a **suffix function (σ)** which checks to see how much of what it is reading matches the search string at any given moment.

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

$$\mathrm{P} = \mathrm{abaabc}$$

$$\mathrm{P}_1 = \mathrm{a}$$

$$P_2 = \mathrm{ab}$$

$$P_3 = \mathrm{aba}$$

$$P_4 = \mathrm{abaa}$$

$$\sigma(\mathrm{abbaba}) = \mathrm{aba}$$

String Matching with Finite Automata

String-Matching Automata

- A suffix function is well defined since the empty string $P_0 = \boldsymbol{\varepsilon}$ is a suffix of every string.
- For example, **$P=ab$, $P_0 = \varepsilon$, $\sigma(\varepsilon)=0$, $\sigma(ccaca)=1$, $\sigma(ccab)=2$.**
- For P[1..m], $\sigma(x)=m$ if and only if $P \sqsupset x$ (* a valid shift *). The whole pattern is the suffix of x.

String Matching with Finite Automata

String-Matching Automata

- For any pattern P of length m, we can define its string-matching automata:

$Q = \{0,\ldots,m\}$ (states)

$q_0 = 0$ (start state)

$F = \{m\}$ (accepting state)

- The transition function δ is defined by the following equation, for any state q and character a:

$$\delta(q,a) = \sigma(P_q a)$$

- We define $\delta(q,a) = \sigma(P_q a)$ because we want to keep track of longest prefix of the pattern P that has matched the string T so far.

String Matching with Finite Automata

String-Matching Automata

- We consider the most recently read characters of T .
- In order, for a substring of T (let's say the substring ending at T[i]) to match some prefix P_j of P, this prefix P_j must be a suffix of T_i .
- Suppose that q= $\phi(T_i)$, so that after reading T_i , the automaton is in state q.
- We design the transition function δ so that this state number, q, tells us the length of the longest prefix of P that matches a suffix of T_i . That is, in state q, $P_q \sqsupset T_i$ and q=$\sigma(T_i)$.
- Whenever q=m, all m characters of P match a suffix of Ti , and so we have found a match.
- Thus, since ϕ (Ti) and σ(Ti) both equal q, we shall see that the automaton maintains the following invariant:

  ϕ (Ti) and σ(Ti)

String Matching with Finite Automata

Finite-Automaton-Matcher

The simple loop structure implies a running time for a string of length *n* is O(n).

However: this is only the running time for the actual string matching. It does not include the time it takes to compute the transition function.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n
do q ← δ(q, T[i])
    if q = m
        then s ← i − m
            print "Pattern occurs at shift" s
```

String Matching with Finite Automata

Computing the Transition Function

```
1  Compute-Transition-Function (P,Σ)
2  m ← length[P]
3  For q ← 0 to m
4      do for each character a ∈ Σ
5          do k ← min(m+1, q+2)
6              repeat k ← k-1
7                  until P_k ⊃ P_q a
8              δ(q,a) ← k
9  return δ
```

This procedure computes $\delta(q,a)$ according to its definition. The loop on line 2 cycles through all the states, while the nested loop on line 3 cycles through the alphabet. Thus, all state-character combinations are accounted for. Lines 4-7 set $\delta(q,a)$ to be the largest k such that $P_k \sqsupset P_q a$.

String Matching with Finite Automata

Example :

- **P= a b a b a c a**
- q= 3 (implies text is :a b a) (step 2)
- a ← Σ (step 3)

  k = min(7+1, 3+2)=5, k-1=4, (steps 4,5)

  $p_4 \sqsupset p_3.a$? No. k ← k-1=3 (step 5)

  $p_3 \sqsupset p_2.a$? Yes; **δ(2,a) ← 3** (steps 6,7)
- b← Σ (step 3)

  k = min(7+1, 3+2)=5, k-1=4, (steps 4,5)

  $p_4 \sqsupset p_3.b$? Yes; **δ(3,b) = 4** (steps 6,7)
- Similarly for c← Σ ; **δ(3,c) = 0**

```
1  Compute-Transition-Function (P,Σ)
2  m ← length[P]
3  For q ← 0 to m
4      do for each character a ∈ Σ
5          do k ← min(m+1, q+2)
6              repeat k ← k-1
7                  until P_k ⊃ P_q a
8              δ(q,a) ← k
9  return δ
```

String Matching with Finite Automata

- This procedure builds $\delta(q,a)$ is a straight-forward way by definition. It considers all states q and all characters in Σ.
- For each combination, to find the the largest k such that $P_k \sqsupset P_q a$. The worst case time complexity is $O(m^3|\Sigma|)$

```
Compute-Transition-Function (P,Σ)
m ← length[P]
For q ← 0 to m
    do for each character a ∈ Σ
        do k ← min(m+1, q+2)
            repeat k ← k-1
                until P_k ⊃ P_q a
            δ(q,a) ← k
return δ
```

KMP Algorithm

- KMP is the first linear time algorithm for string matching.
- Prevents re examination of previously matched characters.
- This algorithm avoids computing the transition function δ altogether, and its matching time is $\theta(n)$ using just an auxiliary function π.
- $\pi[q]$ (Prefix Table ot LPS Table) stores information that is needed to compute transition function $\delta(q,a)$ but that does not depend on a.
- array $\pi[q]$ has only m entries, whereas δ has $\theta(m|\sum|)$.

KMP Algorithm

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                         // number of characters matched
for i = 1 to n                                // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                              // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                             // next character matches
    if q == m                                 // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                              // look for the next match
```

KMP Algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

The running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$