4 String Matching – What's behind Ctrl+F?

24 February 2020

Sebastian Wild

version 2020-02-25 14:45

Outline

4 String Matching

4.1 Introduction

Ubiquitous strings

string = sequence of characters

- universal data type for . . . everything!
  - natural language texts
  - programs (source code)
  - websites
  - XML documents
  - DNA sequences
  - bitstrings
  - . . . a computer's memory $\rightsquigarrow$ ultimately any data is a string

$\rightsquigarrow$ many different tasks and algorithms

Ubiquitous strings

string = sequence of characters

- universal data type for . . . everything!
  - natural language texts
  - programs (source code)
  - websites
  - XML documents
  - DNA sequences
  - bitstrings
  - . . . a computer's memory $\rightsquigarrow$ ultimately any data is a string

$\rightsquigarrow$ many different tasks and algorithms

- This unit: finding (exact) **occurrences of a pattern** text.
  - Ctrl+F
  - grep
  - computer forensics (e. g. find signature of file on disk)
  - virus scanner
- basis for many advanced applications

Notations

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *"a string over alphabet Σ"*
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...)
  - "what you can type on a keyboard", Unicode characters
    comprehensive standard character set including emoji and all known symbols
  - $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ...

Notations

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *"a string over alphabet Σ"*
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...)
  - "what you can type on a keyboard", Unicode characters
  - $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ...

comprehensive standard character set including emoji and all known symbols

- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
- $\Sigma^\star = \bigcup_{n\ge 0} \Sigma^n$: set of **all** (finite) strings over Σ
- $\Sigma^+ = \bigcup_{n\ge 1} \Sigma^n$: set of **all** (finite) **nonempty** strings over Σ
- $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)

Notations

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *“a string over alphabet Σ”*
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, …)
  - “what you can type on a keyboard”, Unicode characters
  - $\{0,1\}$; nucleotides $\{A,C,G,T\}$; …

comprehensive standard character set including emoji and all known symbols

- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
- $\Sigma^\star = \bigcup_{n\ge 0} \Sigma^n$: set of **all** (finite) strings over Σ
- $\Sigma^+ = \bigcup_{n\ge 1} \Sigma^n$: set of **all** (finite) **nonempty** strings over Σ
- $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)

$S[0..n-1]$

zero-based (like arrays)!

- for $S \in \Sigma^n$, write $S[i]$ (other sources: S_i) for i**th** character $(0 \le i < n)$
- for $S, T \in \Sigma^\star$, write $ST = S \cdot T$ for **concatenation** of S and T
- for $S \in \Sigma^n$, write $S[i..j]$ or $S_{i,j}$ for the **substring** $S[i] \cdot S[i+1] \cdots S[j]$ $(0 \le i \le j < n)$
  - $S[0..j]$ is a **prefix** of S; $S[i..n-1]$ is a **suffix** of S
  - $S[i..j) = S[i..j \overset{-}{\cancel{+}} 1]$ (endpoint exclusive) $\rightsquigarrow$ $S = S[0..n)$

Clicker Question

True or false: $\Sigma^\star = \Sigma^+ \cup \{\varepsilon\}$

A True

B False

`pingo.upb.de/622222`

Clicker Question

True or false: $\Sigma^{\star} = \Sigma^{+} \cup \{\varepsilon\}$

A True ✓

B ~~False~~

pingo.upb.de/622222

String matching – Definition

Search for a string (pattern) in a large body of text

- **Input:**
  - $T \in \Sigma^n$: The *text* (haystack) being searched within
  - $P \in \Sigma^m$: The *pattern* (needle) being searched for; typically $n \gg m$
- **Output:**
  - the *first occurrence (match)* of P in T: $\min\{i \in [0..n-m) : T[i..i+m) = P\}$
  - or `NO_MATCH` if there is no such i ("P does not occur in T")
- Variant: Find **all** occurrences of P in T.

  $\rightsquigarrow$ Can do that iteratively (update T to $T[i+1..n)$ after match at i)

- **Example:**
  - T = "`Where is he?`"
  - P_1 = "`he`" $\rightsquigarrow$ $i = 1$
  - P_2 = "`who`" $\rightsquigarrow$ `NO_MATCH`
- string matching is implemented in Java in `String.indexOf`

4.2 Brute Force

Abstract idea of algorithms

Pattern matching algorithms consist of *guesses* and *checks*:

- A **guess** is a position i such that P might start at $T[i]$.
  Possible guesses (initially) are $0 \le i \le n - m$.
- A **check** of a guess is a pair (i, j) where we compare $T[i + j]$ to $P[j]$.
- Note: need all m checks to verify a single correct guess i,
  but it may take (many) fewer checks to recognize an incorrect guess.
- Cost measure: #character comparisons = #checks

$\rightsquigarrow$ cost $\le n \cdot m$ (number of possible checks)

Brute-force method

```
procedure bruteForceSM(T[0..n), P[0..m))
    for i := 0, . . . , n − m − 1 do
        for j := 0, . . . , m − 1 do
            if T[i + j] ≠ P[j] then break inner loop
        if j == m then return i
    return NO_MATCH
```

- try all guesses i
- check each guess (left to right); stop early on mismatch
- essentially the implementation in Java!

- **Example:**
  T = abbbababbab
  P = abba

$\rightsquigarrow$ 15 char cmps
(vs $n \cdot m = 44$)
not too bad!

	i=0	i=1	i=2	i=3	i=4						
T =	a	b	b	b	a	b	a	b	b	a	b
	a	b	b	~~a~~							
		~~a~~									
			~~a~~								
				~~a~~							
					a	b	~~b~~				

Brute-force method

procedure bruteForceSM($T[0..n)$, $P[0..m)$)
    for $i := 0, \ldots, n - m - 1$ **do**
        for $j := 0, \ldots, m - 1$ **do**
            if $T[i + j] \neq P[j]$ **then break** inner loop
        if $j == m$ **then return** i
    return `NO_MATCH`

- try all guesses i
- check each guess (left to right); stop early on mismatch
- essentially the implementation in Java!

- **Example:**
  $T =$ `abbbababbab`
  $P =$ `abba`

$\rightsquigarrow$ 15 char cmps
(vs $n \cdot m = 44$)
not too bad!

a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							
	a									
		a								
			a							
				a	b	b				
					a					
						a	b	b	a	

Brute-force method – Discussion

Brute-force method can be good enough

- typically works well for natural language text
- also for random strings

but: can be as bad as it gets!

a	a	a	a	a	a	a	a	a	a	a
a	a	a	b							
	a	a	a	b						
		a	a	a	b					
			a	a	a	b				
				a	a	a	b			
					a	a	a	b		
						a	a	a	b	
							a	a	a	b

- Worst possible input: $P = a^{m-1}b$, $T = a^n$
- Worst-case performance: $(n - m + 1) \cdot m$

$\rightsquigarrow$ for $m \le n/2$ that is $\Theta(mn)$

Brute-force method – Discussion

Brute-force method can be good enough
- typically works well for natural language text
- also for random strings

but: can be as bad as it gets!

a	a	a	a	a	a	a	a	a	a	a
a	a	a	b							
	a	a	a	b						
		a	a	a	b					
			a	a	a	b				
				a	a	a	b			
					a	a	a	b		
						a	a	a	b	
							a	a	a	b

- Worst possible input: $P = a^{m-1}b$, $T = a^n$
- Worst-case performance: $(n - m + 1) \cdot m$

$\rightsquigarrow$ for $m \leq n/2$ that is $\Theta(mn)$

- Bad input: lots of self-similarity in T! $\rightsquigarrow$ can we exploit that?
- brute force does 'obviously' stupid repetitive comparisons $\rightsquigarrow$ can we avoid that?

Roadmap

- **Approach 1** (this week): Use *preprocessing* on the pattern P to eliminate guesses
  (avoid 'obvious' redundant work)
  - Deterministic finite automata (**DFA**)
  - **Knuth-Morris-Pratt** algorithm
  - **Boyer-Moore** algorithm
  - **Rabin-Karp** algorithm

- **Approach 2** ($\rightsquigarrow$ Unit 6): Do preprocessing on the text T
  Can find matches in time *independent of text size(!)*
  - **inverted indices**
  - **Suffix trees**
  - **Suffix arrays**

4.3 String Matching with Finite Automata

Clicker Question

Do you know what regular expressions, NFAs and DFAs are, and how to convert between them?

- **A** Never heard of this; are these new emoji?
- **B** Heard the terms, but don't remember conversion methods.
- **C** Had that in my undergrad course (memories fading a bit).
- **D** Sure, I could do that blindfolded!

`pingo.upb.de/622222`

Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!

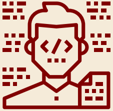
Theoretical Computer Science to the rescue!

- string matching = deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\leadsto$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\leadsto$ can check for occurrence of P in $|T| = n$ steps!

Job done!



`WTF!?`

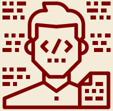
Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!



`WTF!?`

We are not quite done yet.

- (Problem 0: programmer might not know automata and formal languages . . .)
- Problem 1: existence alone does not give an algorithm!
- Problem 2: automaton could be very big!

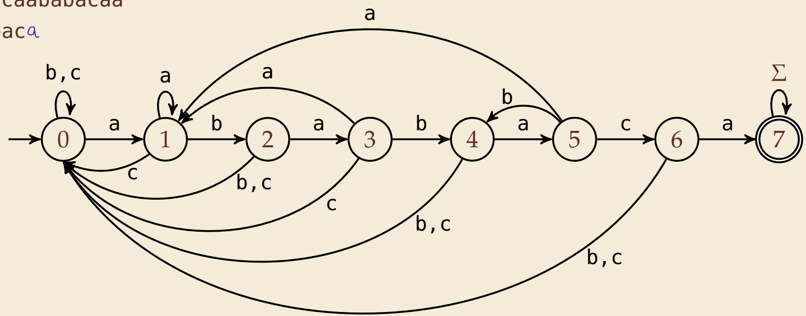
String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

Example:

T = `aabacaababacaa`

P = `ababac`a



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3										

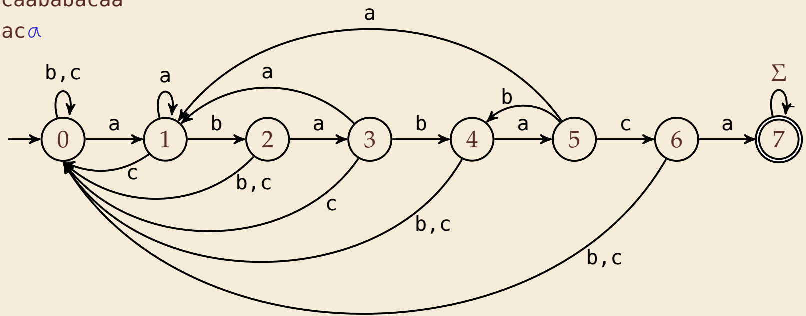
String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

Example:

T = aabacaababacaa

P = ababaca

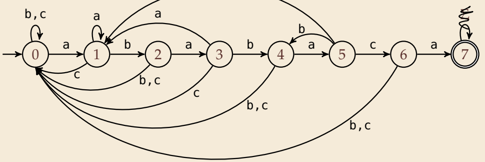


text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	~~0~~ 1	2	3	4	5	6	7	7

String matching DFA – Intuition

Why does this work?

T = aabacaababacaa
P = ababaca



- Main insight: Invariant

> **State q means:**
> *"we have seen $P[0..q)$ until here*
> *(but not any longer prefix of P)"*

text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	~~0~~ 1	2	3	4	5	6	7	7

$P[q]$ $\neq$

- If the next text character c does not match, we know:
  - (i) text seen so far ends with $P[0...q) \cdot c$
  - (ii) $P[0...q) \cdot c$ is not a prefix of P
  - (iii) without reading c, $P[0..q)$ was the *longest* prefix of P that ends here.

$T = \quad \ldots \boxed{P[0..q)}\, c$
$\boxed{P[0..q')}$

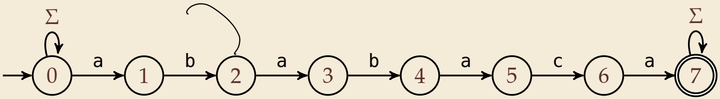
$q' \leq q$

⇝ New longest matched prefix will be (weakly) shorter than q

⇝ All information about the text needed to determine it is contained in $P[0...q) \cdot c$!

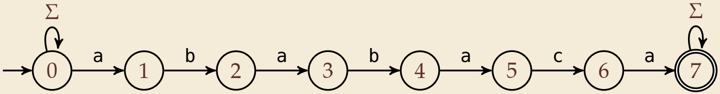
NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part: 

NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part: 

- that actually is a ***non**deterministic finite automaton* (NFA) for $\Sigma^{\star} P\, \Sigma^{\star}$

⇝ We *could* use the NFA directly for string matching:

- at any point in time, we are in a ***set* of states**
- accept when one of them is final state

Example:

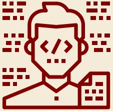
text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0, 1	0, 1	0, 2	0, 3	0	0, 1	0, 1	0, 2	0, 1, 3	0, 2, 4	0, 3, 5	0, 6	0, 1, 7	0, 1, 7

But maintaining a whole set makes this slow . . .

Computing DFA directly

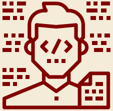
You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

```
The powerset method has exponential state blow up!
      I guess I might as well use brute force ...
```



Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

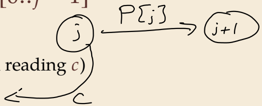


```
The powerset method has exponential state blow up!
      I guess I might as well use brute force ...
```

Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_{j-1} for $P[0..j-1]$

- add new state and matching transition $\leadsto$ easy
- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from j when reading c)



Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

```
The powerset method has exponential state blow up!
      I guess I might as well use brute force ...
```

Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_{j-1} for $P[0..j-1]$

- add new state and matching transition $\rightsquigarrow$ easy
- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from j when reading c)
- $\delta(j, c)$ = length of the longest prefix of $P[0..j]$ that is a suffix of $P[1..j]$
  = state of automaton after reading $P[1..j]$
  $\leq j$ $\rightsquigarrow$ can use known automaton A_{j-1} for that!

State q means:
"we have seen $P[0..q)$ until here
(but not any longer prefix of P)"

$\rightsquigarrow$ can directly compute A_j from A_{j-1}!

seems to require simulating automata $m \cdot \sigma$ times