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

18 November 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-11-18 15:53`

Learning Outcomes

Unit 6: ***String Matching***

1. Know and use typical notions for *strings* (substring, prefix, suffix, etc.).
2. Understand principles and implementation of the *KMP*, *BM*, and *RK* algorithms.
3. Know the *performance characteristics* of the KMP, BM, and RK algorithms.
4. Be able to solve simple *stringology problems* using the *KMP failure function*.

Outline

6 String Matching

6.1 String Notation

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
    comprehensive standard character set including emoji and all known symbols
  - $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ...
- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: 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
    comprehensive standard character set including emoji and all known symbols
  - $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ...
- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
- $\Sigma^\star = \bigcup_{n \geq 0} \Sigma^n$: set of **all** (finite) strings over Σ
- $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$: set of **all** (finite) **nonempty** strings over Σ
- $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)
- zero-based (like arrays)!
  for $S \in \Sigma^n$, write $S[i]$ (other sources: S_i) for i**th** character $(0 \leq i < n)$
- 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** $S[i] \cdot S[i+1] \cdots S[j]$ $(0 \leq i \leq j < n)$
  - $S[0..j]$ is a **prefix** of S; $S[i..n-1]$ is a **suffix** of S
  - $S[i..j) = S[i..j-1]$ (endpoint exclusive) $\rightsquigarrow$ $S = S[0..n)$

Clicker Question

$\bigcup_{n \geq 0} \Sigma^n$ $\qquad$ $\bigcup_{n \geq 1} \Sigma^n$

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

A True

B False

→ sli.do/cs566

Clicker Question

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

A True ✓

B ~~False~~

→ sli.do/cs566

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`, in Python as `str.find`

6.2 Brute Force

Abstract idea of algorithms

String matching algorithms typically use *guesses* and *checks*:

- A **guess** is a position i such that P might start at $T[i]$.
  Possible guesses (initially) are $0 \leq i \leq n - m$.
- A **check** of a guess is a comparison of $T[i + j]$ to $P[j]$.

Abstract idea of algorithms

String matching algorithms typically use *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 comparison of $T[i + j]$ 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

$\rightsquigarrow$ #checks $\le n \cdot m$ (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`

a	b	b	b	a	b	a	b	b	a	b
a	b	b	~~a~~							

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!

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 $m \le n/2$ 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 13): Do *preprocessing* on the **text** T
  Can find matches in time *independent of text size(!)*
  - **inverted indices**
  - **Suffix trees**
  - **Suffix arrays**

6.3 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!

→ sli.do/cs566

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!

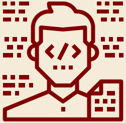
Theoretical Computer Science to the rescue!

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

$\rightsquigarrow$ $\exists$ *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!?`

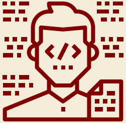
Theoretical Computer Science to the rescue!

- 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!

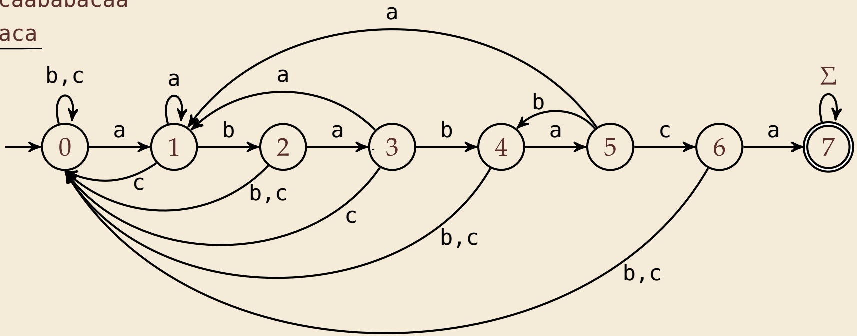
String matching with DFA

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

Example:

T = aabacaababacaa

P = ababaca



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

String matching with DFA

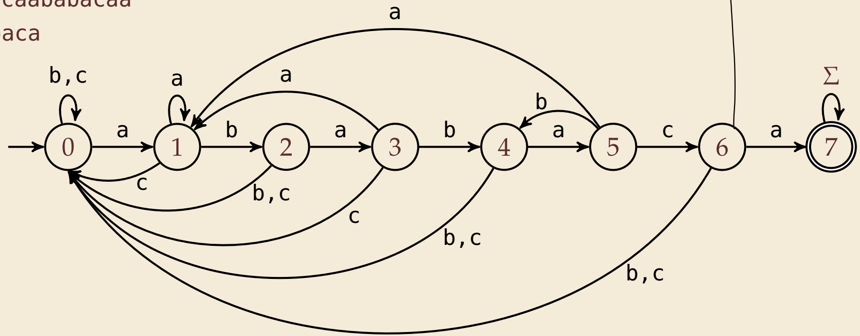
- Assume first, we already have a deterministic automaton
- How does string matching work?

q \ c	a	b	c
0	1	0	0
1			
2			

Example:

T = aabacaababacaa

P = ababaca

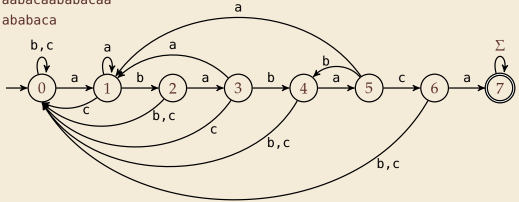


text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

String matching DFA – Intuition

Why does this work?

T = aabacaababacaa
P = ababaca

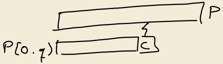


text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

- Main insight:

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

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

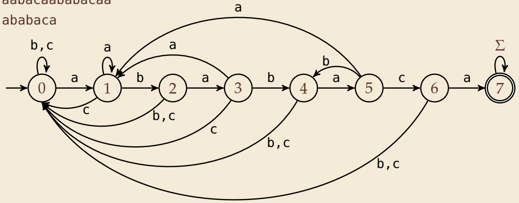


$T = \cdots$ $\boxed{P[0..q)}$ $\boxed{c}$
$\boxed{P[0..q')}$
with $q' < q$

String matching DFA – Intuition

Why does this work?

T = aabacaababacaa
P = ababaca

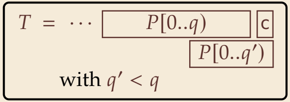


- Main insight:

> 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	1	2	3	4	5	6	7	7

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



$\rightsquigarrow$ New longest matched prefix will be (weakly) shorter than q

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