Strings

Dov Kruger

Department of Electrical and Computer Engineering Stevens Institute of Technology

November 6, 2022



Overview

The core string algorithms are conceptually simple

- Find a small string in a big one
- Compare two documents to find out how many are similar.
- Edit one document letter by letter until it turns into a different one
- Find a regular expression pattern in text

The ideas in the implementations are instructive





Brute Force String Search

The obvious way to compare strings is not the best

Consider searching "testing testing 123" for the string "testosterone"

t	е	S	t	i	n	g	t	е	s	t	i	n	g		1	2	3	
t	е	S	t	0			t	е	S	t	0	t	t	t	t	t	t	
	t	t	t	е				t	t	t	е							

Complexity of Brute Force String Compare

Every char must be compared with the first char of the target

Every time there is a match, the second char must be compared to the next char

Worst case, for n letters in the search string and k letters in the target

O(nk)





Pseudocode of Brute Force Search

```
bruteForceSearch(search, target)
  for i \leftarrow 0 to length(search)-1 //O(n)
    for j \leftarrow 0 to length (target)-1 \frac{1}{O(k)}
      if search[i] ≠ target[0]
         skip to next i loop
      end
    end
    return i; // match found
  end
  return -1; // no match found
end
```

Boyer-Moore: Faster String Search

The problem with the brute force approach

- Trying multiple times on each character
- Not using information to skip matches that cannot possibly work

Approach

- Start with the last letter and match k characters forward
- If the letter found is nowhere in the target, then the target must be further on

Example: target = "marmalade", t is not present, therefore marmalade cannot be to the left

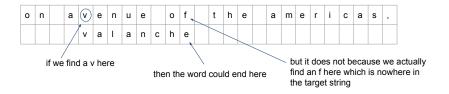




Boyer-Moore: The main concept

For each example below, the algorithm can jump forward by an amount that depends on what letter is found

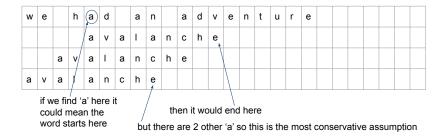
target string = "avalanche" k=9



Boyer-Moore: Another Example Jumping Forward

When a letter is found multiple times in the target (there are 3 letter 'a' in avalanche) then conservatively the algorithm jumps forward by the smallest number

target string = "avalanche" k=9





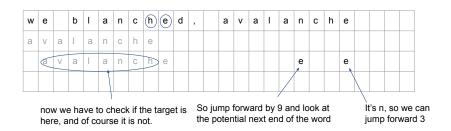


Boyer-Moore: Once the Last Letter is Found

The first step in Boyer-Moore is to rapidly scan forward until the last letter of the target is found

That does not prove that the target is there, though

Only by checking if each letter matches can we be sure







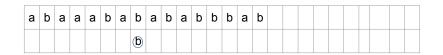
Boyer-Moore: Worst Case

Boyer-Moore is amazing if you can find letters that are not in the target

In such cases you can jump ahead by the length of the target

The worst case is when the alphabet is small and the repetitiveness gets high

Example: looking for target "abababab"



Complexity of Boyer-Moore, and Building the Table

The longer the target string, the faster Boyer-Moore executes O(n/k)

However, it requires a table of offsets to be generated

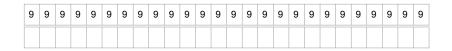
```
\begin{array}{lll} \mbox{buildBoyerMooreTable(table, target)} \\ \mbox{$k \leftarrow $ length(target)$} \\ \mbox{$for$ $i \leftarrow 0$ to $255$} \\ \mbox{$table[i] \leftarrow k$} \\ \mbox{end} \\ \mbox{$for$ $i \leftarrow 0$ to length(target)-1$} \\ \mbox{$table[target[i]] \leftarrow k-1-i$} \\ \mbox{end} \end{array}
```



Example: For the target string "avalanche" (k=9) compute the table for Boyer-Moore

We will show only the letters of the alphabet

First: set all elements of the table to the length of the target string (9)



Example: For the target string "avalanche" (k=9) compute the table for Boyer-Moore

We will show only the letters of the alphabet

For the first letter 'a' if it is found, the end of the word is 8 letters ahead

8	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
а	b	С	d	е	f	g	h	i	j	k	I	m	n	0	р	q	r	s	t	u	٧	w	х	у	z

Example: For the target string "avalanche" (k=9) compute the table for Boyer-Moore

We will show only the letters of the alphabet

For the second letter 'v' if it is found, the end of the word is 7 letters ahead

8	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	7	9	9	9	9
а	b	С	d	е	f	g	h	i	j	k	I	m	n	0	р	q	r	s	t	u	٧	w	х	у	z

Example: For the target string "avalanche" (k=9) compute the table for Boyer-Moore

We will show only the letters of the alphabet

For the third letter 'a' it is the same as the first letter. It turns out we cannot jump

8	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	7	9	9	9	9
а	b	С	d	е	f	g	h	i	j	k	I	m	n	0	р	q	r	s	t	u	v	w	х	у	z

Boyer-Moore Code

Full code is rather large, wikipedia has a good one:

 $https://en.wikipedia.org/wiki/Boyer\%E2\%80\%93Moore_string-search_algorithm$

Finite State Machines

Boyer-Moore is ultra-fast for searching one string. What if you want to search for multiple strings, or a pattern?

Examples

- Find any occurrence of "cat" or "dog" or "elephant"
 - This would require 3 separate invocations of Boyer-Moore, 3 generations of table (slow)
- Find any occurrence of "cat" followed by any letters ending in "og"
 - Would find "catalog" or "cat sat on the log"
 - This simply is not supported by Boyer-Moore

For these cases, the optimal algorithm is a finite state machine The language used to generate these is called regular expressions



Regular Expressions

Regular expressions are complicated, and take storage to create but once in existence they can search far faster for multiple words than a regular search algorithm

To test regular expressions we will use the website regexr.com

Regex Summary

/abc/	abc	
/ab*c/	the letter a fol- lowed by zero or more bs, followed by c	ac, abc, abbc, abbbc, abbbbbbbc
/ab?c/	b is optional	ac, abc
/ab+c/	b must be at least 1 time	abc, abbc, abbbc
/cat dog/	either cat or dog	cat, dog, cat alog, dog ged
/[aeiou]/	any of the letters aeiou	
/[a-z]/	any letters a through z	a, b, c, d, e, f, g, , z
/x[a-z]*y[a-z]+z/		xybz, xabcyabcdefz



Interactive Regex

To explore how regular expressions work we will use them interactively

There are a number of good sites

- https://regexr.com
- https://regex101.com/
- https://www.regextester.com/

Interactive Regex Problem set

Let's do some problems live in class. Write regex to parse:

Phone Numbers

Markdown

C++ variable declarations



State Machines: How Regular Expressions work

Regular expressions can be implemented as state machines Start in a known state 0 For each state, each input can result in

- An action
- A transition to another state

There is no other "memory" in a pure state machine Only the state number conveys information

State Machines: Recognizing the String "cat"



State Machines: Recognizing the String "cat"

Longest Common Subsequence (LCS)

LCS compares two strings to determine what they have in common For strings that start the same, this is easy "hellox" compared to "helloa" obviously the first 5 characters match

The problem is when the prefixes are different Compare "hello" to "ohell"

On the surface, they are completely different 'h' \neq 'o'
'e' \neq 'h'
'l' \neq 'e'
'l' one match, almost an accident 'o' \neq 'l'



LCS: Brute Force Recursion (Intractable)

LCS can be easily written recursively, but the complexity is staggering

```
 \begin{split} \mathsf{LCS}(\mathsf{a},\mathsf{b}) & /\!/O(2^n) \\ & \mathsf{if} \ \mathsf{a}.\mathsf{length} = 0 \ \mathsf{or} \ \mathsf{b}.\mathsf{length} = 0 \ /\!/ \ \mathit{if} \ \mathit{either} \ \mathit{string} \ \mathit{is} \ \mathit{empty}, \\ & \mathsf{no} \ \mathsf{match} \ (\mathsf{base} \ \mathsf{case}) \\ & \mathsf{return} \ 0 \\ & \mathsf{end} \\ & \mathsf{if} \ \mathsf{a}[0] = \mathsf{b}[0] \\ & \mathsf{return} \ 1 + \mathsf{LCS}(\mathsf{a}.\mathsf{substr}(1), \ \mathsf{b}.\mathsf{substr}(1)) \ /\!/ \ \mathit{found} \ 1 \\ & \mathsf{letter}, \ \mathsf{do} \ \mathsf{the} \\ & \mathsf{rest} \ \mathsf{recursively} \\ & \mathsf{end} \\ & \mathsf{return} \ \mathsf{max}(\mathsf{LCS}(\mathsf{a}, \ \mathsf{b}.\mathsf{substr}(1)), \ \mathsf{LCS}(\mathsf{a}.\mathsf{substr}(1), \ \mathsf{b})) \\ \mathsf{end} \end{aligned}
```

Dynamic Programming for LCS

With any exponential recursion, dynamic program can be used to reduce the cost

The price is memory

By storing a table it is possible to reduce the complexity of LCS but the cost is O(mn) space, where m and n are the sizes of the strings.

LCS Table

Compare "hello" and "hi Jello"

		h	i		J	е	I	I	О
		0	0	0	0	0	0	0	0
h	0 -	1 -	-1 -	- 1	1	1	1	1	1
е	0	1 -	-1 -	-1 -	-1 -	-2 -	-2 -	-2 -	-2
I	0						3		
I	0						3	4	
0	0								5