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

The obvious way to compare strings is not the best

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

t	e	s	t	i	n	g		t	e	s	t	i	n	g		1	2	3	.
t	e	s	t	o				t	e	s	t	o	t	t	t	t	t	t	
	t	t	t	e					t	t	t	e							

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 ← 0 to length(search)−1 //O(n)
    for j ← 0 to length(target)−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

t	e	s	t	i	n	g		t	e	s	t	i	n	g		1	2	3	.

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

target string = "avalanche" k=9

o	n		a	v	e	n	u	e		o	f		t	h	e		a	m	e	r	i	c	a	s	,
				v	a	l	a	n	c	h	e														

if we find a v here

then the word could end here

but it does not because we actually find an f here which is nowhere in the target string

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

w	e		h	a	d		a	n		a	d	v	e	n	t	u	r	e							
				a	v	a	l	a	n	c	h	e													
		a	v	a	l	a	n	c	h	e															
a	v	a	l	a	n	c	h	e																	

if we find 'a' here it could mean the word starts here

then it would end here

but there are 2 other 'a' so this is the most conservative assumption

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

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

w	e		b	l	a	n	c	h	e	d	,		a	v	a	l	a	n	c	h	e				
a	v	a	l	a	n	c	h	e																	
	a	v	a	l	a	n	c	h	e									e			e				

now we have to check if the target is here, and of course it is not.

So jump forward by 9 and look at the potential next end of the word

It's n, so we can jump forward 3

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"

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

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

However, it requires a table of offsets to be generated

```
buildBoyerMooreTable(table, target)
  k ← length(target)
  for i ← 0 to 255
    table[i] ← k
  end
  for i ← 0 to length(target)−1
    table[target[i]] ← k−1−i
  end
```

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)

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	9

Boyer-Moore Table

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
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY

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
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	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
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	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

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 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 followed 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"

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' $=$ 'l' one match, almost an accident
'o' $\neq$ 'l'

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

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

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	e	l	l	o
		0	0	0	0	0	0	0	0
h	0	↑ ←1	←1	←1	1	1	1	1	1
e	0	↑ 1	←1	←1	←1	←2	←2	←2	←2
l	0						3		
l	0						3	4	
o	0								5