Module 9: String Matching

Rabin-Karp

Idea: • don't compare strings
• compare hash values of strings instead
the fingerprint

• Don't explicitly create a Hash Table
but still need a random prime number
for the hash function
Module 7 Slide 9 Uniform Hashing Assumption

If we have a good hash function,
we expect that if $h(k_1) = h(k_2)$
then k_1 is likely the same as k_2.
Collisions may happen but we should
have good distribution; i.e. 2 keys that
are similar will likely have different hash
values

Randomly choose M => Randomized alg
=> expected run times

Find hash value for Pattern P, h(P)
- if string, flatten, avoid overflow
- Module 7, Slide 24

Find hash value of Text (same length as P)

T: 3 1 4 1 5 9 2 6 5 3

"rolling hash"

- keeps on shifting right by 1 char
- compute hash for substring in text & compare with h(P)
  - if match "Probably" the same string
    - use naive approach (char by char) to verify match $\Theta(m)$

Rabin & Karp found a way to update "rolling hash" in constant time, $O(1)$, for updates
- still $\Theta(m)$ to compute first hash value

Update: use previous hash to compute next

Suppose Initial hash value is 41592 mod 97 = 76

$4\times10^4 + 1\times10^3 + 5\times10^2 + 9\times10^1 + 2\times10^0 \bmod 97$

and next is 15926 mod 97

- MSD is removed, add new LSD

How can we compute this in constant time?

$4\times10^4 + 1\times10^3 + 5\times10^2 + 9\times10^1 + 2\times10^0$

-4×10^4 O(1)

$*$ (1592) by 10 O(1)?*

$+6$ O(1)

mod 97 O(1)

* to keep numbers small work with mod values

$10^4 \bmod 97 = 9$

$= 7(76 - (4*9))*10 + 6 = 406$

↑ old hash value

$\bmod 97 \Rightarrow 18$

If keys are bit strings, $s \in \{0,1\}^*$

$H(T[i+1 \ldots i+m]) = (H(T[i \ldots i+m-1]) * 2$

$- T[i]\cdot 2^m$ ← bitwise operations

$+ T[i+m]$

$\bmod M$ O(1)

What size M should we use?

- random prime from $\{2, \ldots, mn^2\}$
- a number mn^2 requires $\log mn^2$ bits

$\Rightarrow \log m + 2\log n < 3\log n \in O(\log n)$ if $m \leq n$

$\Rightarrow$ operations using this number require about the same time as n

Probability {failure}: $H(A) = H(y)$ but $A \neq y$

- complicated ... density of primes (Number Theory)

$$\frac{\#\text{primes } p < M \text{ and } p \text{ divides } |A-y| < 2^h}{\#\text{primes} < N}$$

... very small chance.

Expected runtime is $O(m+n)$

Worst-case (bad luck) $\Theta(m \cdot n)$

- all substrings of text hash to same value that is also hash of Pattern
- but no matched strings $\Rightarrow$ failure
- all false positives

Aside: Monte Carlo vs Las Vegas Algs (CS466

$\Theta(m+n)$ $\Theta(mn)$

DFA

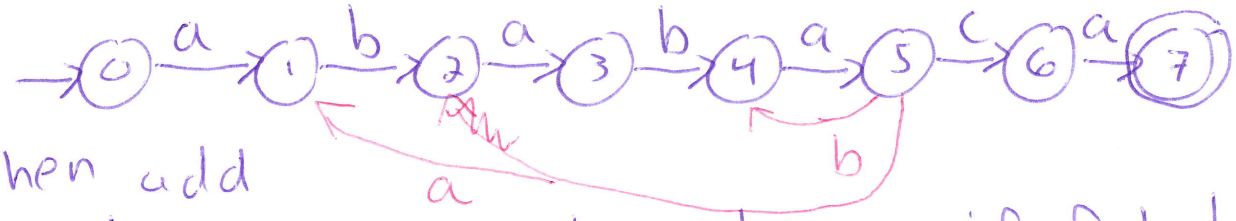
- given a pattern, the DFA is a pattern matching checker

Recall CS 241, use DFA to check tokens

Start by building backbone to find pattern

eg P = a b a b a c a

CS 360/365



Then add transitions on where to go if fail char found

- for a string s, $\ell(s) \in \{0, .., m\}$ is the length of the longest prefix of P that is also a suffix of S

eg P = a b a b a c a

S = a b a b a a

=> $\ell(S) = 1$

=> $\delta(5, a) = 1$

- a is the longest prefix of P that is a suffix of S

From state 5 on 'a' goto state 1

eg $P = abababaca$

$S = abababab$

$\Rightarrow \ell(S) = 4$

$\Rightarrow \delta(5, b) = 4$

OR create an NFA

end of 341

NFA to DFA - Polynomial time to remove non-determinism

- potentially exponential blowup

KMP

Goal: O(n) worst-case

- compare pattern left to right
- when mismatch occurs, shift pattern "intelligently"
  - similar to mismatch transitions in DFA
- never move the "guess" index backwards

Precompute: failure array F

F[index of last char of Pattern that matched text]

=> index of pattern to check next
(at location of text where mismatch occurred)

Ex

	0	1	2	3	4	5
T:	a	b	a	b	a	d
	✓	✓	✓	✓	✓	✗
P:	a	b	a	b	a	c
shift →	a	b	a	b	...	

- mismatch at index 5 of pattern and text
  - do not consider mismatch char of text when shifting => shift is only based on the pattern (not text)

lookup F[4]

Mismatch at j, define

F[j-1] = maximal prefix of P[0..j-1]
that matches suffix of P[1..j-1]

eg F[4]: P[0..4] = ababa
P[1..4] = baba

- longest prefix that matches suffix is aba
  - by shifting pattern we know aba (0 1 2) already matches, check starting at next char => index 3

F[4] = 3

eg F[5]: P = abacaba
P[0..5] = abacab
P[1..5] = bacab
=> F[5] = 2
length 2

Confusing part: a mismatch at index 6 of pattern
=> lookup F[5]

Boyer-Moore

Idea: Match pattern in reverse (R->L)

- bad character heuristic

What if the text character doesn't appear in the pattern? => shift pattern beyond this text char
- shift O(m) chars

- can potentially make large shifts with few checks

If text char appears in pattern (text ≠ pattern mismatch)
=> shift so last occurrence of char in pattern lines up

- good suffix (often do not cover this anymore)

Idea: Shift pattern forward to align with previous occurrence of suffix.
- Similar to failure array but may not be a prefix

If using both heuristics: compute both, take better - one that shifts pattern most

22] Check: T[3] ≠ P[3]

- 'r' does not occur in "aldo"
  => jump past bad character
- align P[0] with T[4]

In forward-searching, a mismatch with char not in string of P at index 0 only moves P over by 1.

In Reverse-searching, we start further in the Text so jumping passed may move us farther along with fewer checks.

23] If mismatch, align with last occurrence of text char in pattern

- then start checking from end of pattern

Check: T[i=4] ≠ P[j=4] but 'a' occurs in Pattern P
=> shift so last occurrence of 'a' in P lines up
=> shift right by 3

- mismatch at P[4], last occurrence is at P[1]
  update $i = i + (m-1-1)$

- calc update i is tricky: slides 25-27

Last-Occurrence Array

- mapping for all characters c in alphabet Σ (usually group "all others")

L(c) is the index where c occurs last in pattern

- -1 if c does not occur in pattern

eg $P = \underset{0}{a}\underset{1}{b}\underset{2}{a}\underset{3}{c}\underset{4}{a}\underset{5}{b}$, $\Sigma = \{a, b, c, d, \ldots, z\}$

c	a	b	c	d	e	...	z
L(c)	4	5	3	-1	-1	...	-1

"all others"

Note: if char not in pattern, we shift

$i = i + m - 1 - -1 \Rightarrow i = i + m$

$L[T(i)] = -1$

Boyer-Moore typically performs well on English text.

Preprocessing $O(m + |\Sigma|)$ - set "all others" to -1
- they are individual entries

Worst case (with only bad char): $O(mn)$

- with good suffix, $O(n + m + |\Sigma|)$
  - difficult to prove ... Galil Rule ...
  - skip sections known to match, etc

Suffix Trees

- preprocess Text instead of Pattern
- compressed trie of all suffixes of T

The first char of all suffixes are all potential starting points to match the pattern

- the trie then groups
  - follow 1 branch to all suffixes that start w. th 'a', etc
- search is then based on length of pattern not n

Building Suffix tree (compressed trie)

T has length n

Length of Suffixes: $n + (n-1) + (n-2) + \ldots + 0$

$\in \Theta(n^2)$

- each node has $\Theta(|\Sigma|)$ children

  => $\Theta(n^2|\Sigma|)$

  - could do $\Theta(n|\Sigma|)$ but its complicated

Search - essentially search for P in compressed trie but only want a prefix match

=> $O(|\Sigma| m)$

Suffix Array

- simple to build
- log factor slower than suffix tree
  - saves on space : we usually go the other way
    more space => faster runtime
    * simplicity is the winning factor here

- array
  - suffixes T[i..n-1] for i=0..n-1
  - sort lexicographically, order of corresponding i values fills array

Sorting trick: use MSD-Radix-Sort

after 1 round buckets contain all suffixes with the same first char

- ordered by rest of the chars
  - sorted elsewhere

=> double length of sorted part every round

=> $O(\log n)$ rounds

=> $O(n \log n)$ sorting total

Search: Binary Search for P

- $O(\log n)$ comparisons, each comparison $O(m)$

=> $O(m \log n)$