String Matching

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Εθνικό Μετσόβιο Πολυτεχνείο

Problem Definition

- String : array of characters
  - Σ: alphabet
- Two strings are given:
  - a text T[1 … n]
  - a pattern P[1 … m]
- Problem: find the first substring that is the same as the pattern
- For every shift s: T_s =T[s … s+m-1]
- Problem definition rephrased: find the smallest s such that T_s = P.
- In most cases m<<n

Examples

- T=“AMANAPLANACATACANAPANAMA”
  - P=“CAN”
  - S=15
- T=“AMANAPLANACATACANAPANAMA”
  - P=“SPAM”
  - S=None

Almost Brute Force Algorithm

```
AlmostBruteForce(T[1..n], P[1..m]):
  for s ← 1 to n − m + 1
      equal ← True
      i ← 1
      while equal and i ≤ m
          if T[s + i − 1] ≠ P[i]
              equal ← False
          else
              i ← i + 1
      if equal
          return s
  return None
```

worst case:
Text: A..A   n A's
Pattern A..AB   m-1 A's
Complexity:
O((n-m)m)=O(nm)

Almost: break out of the inner loop at the first mismatch

Strings as Numbers

- Σ (alphabet) ={0,1,2,3,4,5,6,7,8,9}
  - p : Numerical Value of pattern P
  - T_s : Numerical Value of T_s

$$p = \sum_{i=1}^{m} 10^{m-i} \cdot P[i] \qquad t_s = \sum_{i=1}^{m} 10^{m-i} \cdot T[s+i-1]$$

- T= 31415926535897932384626433832795028841971
  - m=4 T_{17} =2384
- Rephrasing problem definition: find the smallest s such that $p=t_s$

Strings as Numbers

- Compute p using Horner's Rule
  - time O(m)

$$p = P[m] + 10\left(P[m-1] + 10\left(P[m-2] + \cdots + 10\left(P[2] + 10 \cdot P[1]\right)\cdots\right)\right)$$

  - Computing t_s by the same way is useless (we get the same brute force algorithm)
- Compute t_{s+1} from t_s in constant time
  - subtract the most significant digit T[s] * 10^{m-1}
  - shift everything up by one digit
  - add the new least significant digit T[r+m]

$$t_{s+1} = 10\left(t_s - 10^{m-1} \cdot T[s]\right) + T[s+m]$$

  - T= 31415926535897932384626433832795028841971
  - t_s =2384 t_{s+1}=3846

Strings as Numbers

NUMBERSEARCH($T[1..n], P[1..m]$):

$\sigma \leftarrow 10^{m-1}$

$p \leftarrow 0$

$t_1 \leftarrow 0$

for $i \leftarrow 1$ to m

  $p \leftarrow 10 \cdot p + P[i]$

  $t_1 \leftarrow 10 \cdot t_1 + T[i]$

for $s \leftarrow 1$ to $n - m + 1$

  if $p = t_s$

    return s

  $t_{s+1} \leftarrow 10 \cdot (t_s - \sigma \cdot T[s]) + T[s+m]$

return NONE

Complexity: $O(n)$?

Karp Rabin Fingerprinting (1981)

- Perform all arithmetic modulo some prime number q
  - q: 10*q fits into a standard integer variable (avoid long integer data type)
  - (p mod q) fingerprint of P (t_s mod q) fingerprint of T_s
- Compute (p mod q), (t_s mod q) in O(m). (Horner's rule)

$$p \bmod q = P[m] + \left(\cdots + \left(10 \cdot \left(P[2] + \left(10 \cdot P[1] \bmod q\right) \bmod q\right) \bmod q\right) \cdots \right)\right) \bmod q$$

- Given (t_s mod q) compute (t_{s+1} mod q) in constant time

$$t_{s+1} \bmod q = \left(10 \cdot \left(t_s - \left(\left(10^{m-1} \bmod q\right) \cdot T[s] \bmod q\right) \bmod q\right) \bmod q\right) + T[s+m] \bmod q$$

Karp Rabin Fingerprinting (1981)

- Two cases:
  - (p mod q) != (t_s mod q)   P != T_s
  - (p mod q) = (t_s mod q)   P = T_s ?
    - (if P != T_s) false match at shift s
    - test false match by *brute force* string comparison
    - F: number of false matches
    - Complexity *O(n+F*m)*
  - false match possibility 1/q
  - F=n/q
  - Complexity *O(n+n*m/q)*
  - if q>>m *O(n)*

Karp Rabin algorithm

$\underline{\text{KARPRABIN}(T[1..n], P[1..m])\text{:}}$

$q \leftarrow$ ***a random prime number between 2 and*** $\lceil m^2 \lg m \rceil$

$\sigma \leftarrow 10^{m-1} \bmod q$

$\tilde{p} \leftarrow 0$

$\tilde{t}_1 \leftarrow 0$

for $i \leftarrow 1$ to m

$\quad \tilde{p} \leftarrow (10 \cdot \tilde{p} \bmod q) + P[i] \bmod q$

$\quad \tilde{t}_1 \leftarrow (10 \cdot \tilde{t}_1 \bmod q) + T[i] \bmod q$

for $s \leftarrow 1$ to $n - m + 1$

$\quad$ if $\tilde{p} = \tilde{t}_s$

$\quad\quad$ if $P = T_s$ ⟨⟨*brute-force O(m)-time comparison*⟩⟩

$\quad\quad\quad$ return s

$\quad \tilde{t}_{s+1} \leftarrow \big(10 \cdot \big(\tilde{t}_s - (\sigma \cdot T[s] \bmod q) \bmod q\big) \bmod q\big) + T[s+m] \bmod q$

return NONE

Karp Rabin algorithm

- $\pi(u)$ the number of prime numbers less than u
- $\pi(m^2 \log m)$ possible values of q

- Lemma 1 $\pi(u) = \Theta(u/\log u)$
- Lemma 2 any integer x has at most $\lfloor \lg x \rfloor$ distinct prime divisors (if x has k prime divisors x>=2^k, since every prime number is bigger than 1)

- if there is a true match the algorithm ends early otherwise p!= t_s for every s
- if there is false match at s then q divides |p- t_s |

Karp Rabin algorithm

- $|p - t_s| < 10^m$ since both $p, t_s < 10^m$
- $|p - t_s|$ has at most $O(m)$ prime divisors (lemma 2)
- q *is randomly chosen from a set of* $\pi(m^2 \log m)$ prime numbers
- probability of false match at shift s *O(1/m)*
- probability of false match at any shift *O(n/m)*
- Karp Rabin runs in *O(n)* **expected time**

Knuth Morris Pratt algorithm(1977)

- Redundant Comparisons (brute force algorithm)
  - text = “HOPUSCOPUSABRA**B**RACADABRA”
  - pattern = “ABRA**C**ADABRA”
  - for s<11 algorithm fails from the very beginning
  - for s=11 algorithm fails at fifth position
  - for s=12 , s=13 algorithm fails
  - for s=14, T[14]=P[4] match
  - *Once we’ve found a match for a text character, we never need to do another comparison with that text character again.(T[12], T[13])*
  - *The next reasonable shift is the smallest value of s such that T[s .. i-1] which is a suffix of the previously-read text is also a proper prefix of the pattern (ABRA CAD ABRA)*

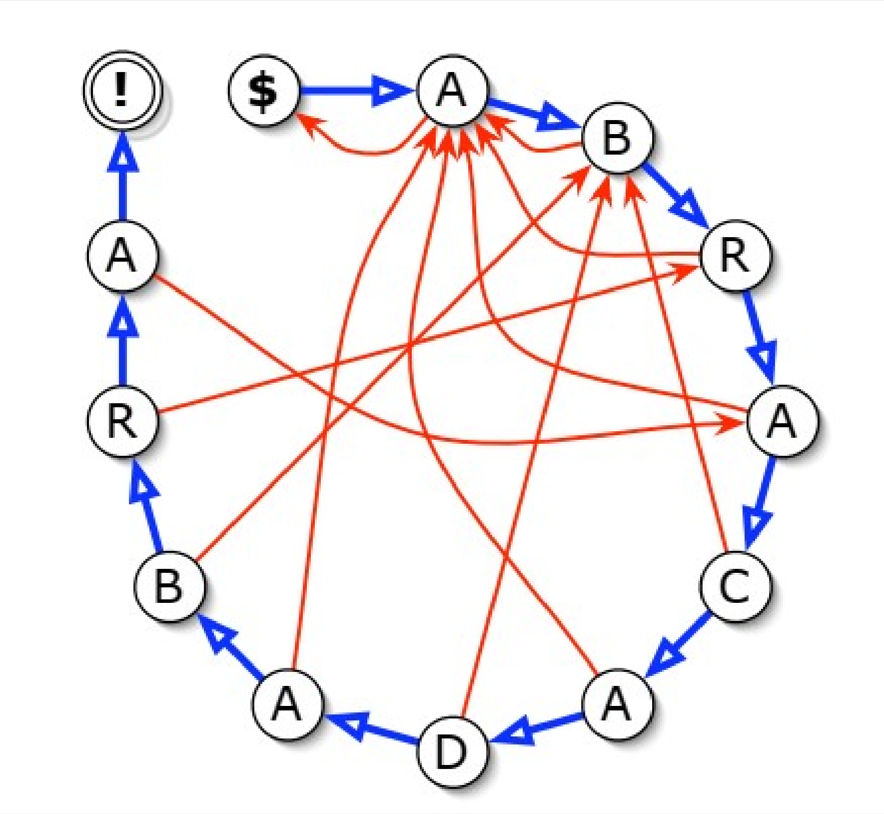
example

- text: “qwe**r**qwedqwrqwedqwedqwegwqedg”
- pattern: “qwe**d**qweg”   r!=d
- q,w,e differ (there was a match from q do not expect match from w)
- text: “**r**qwedqwrqwedqwedqwegwqedg”
- pattern: “**q**wedqweg”   r!=q
- text: “qwedqw**r**qwedqwedqwegwqedg”
- pattern: “qwedqw**e**g”
- search before “e” for prefix=suffix **qw** start pattern from e
- text: “**r**qwedqwedqwegwqedg”
- pattern: “edqw**e**g” r!=e
- pattern before e qw (q,w differ start pattern from scratch)
- text: “rqwedqwedqwegwqedg”
- pattern: “qwedqweg” r!=q advance text by 1

example

- text: “qwedqwe**d**qwegwqedg”
- pattern: “qwedqwe**g**” d!=g
- search pattern for prefix-suffix before g: “qwe”
- text: “dqwegwqedg”
- pattern: “dqweg”

Finite State Machine



Finite State Machine

Labels: characters from the pattern

Edges: 2 outgoing **success**, failure

Iterate by 2 rules:

if T[i]=P[j] or current label $ follow the success edge. Increment i.

if T[i]!=P[j] follow the failure edge. Do not change i.

! pattern found

Is it always possible to construct the whole graph? If the pattern is long?

The answer is:

failure function: fail[j] how far to shift after character mismatch (T[i]!=P[j])

Knuth Morris Pratt algorithm

```
KnuthMorrisPratt(T[1..n], P[1..m]):
  j ← 1
  for i ← 1 to n
      while j > 0 and T[i] ≠ P[j]
          j ← fail[j]
      if j = m     ⟨⟨Found it!⟩⟩
          return i − m + 1
      j ← j + 1
  return None
```

Assume failure function known
worst case complexity: *O(n)*
At most n-1 failed comparisons (the number of time we decrease j can not exceed the number of time we increment j)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
e	k	e	k	e	d	e	k	e	k	e	d	e	k	e	k	e	k
0	0	1	2	3	0	1	2	3	4	5	6	7	8	9	10	11	4

failure array example

- pattern: “akekedekekedekekek”
- fail[i]: which is the longest suffix that is also prefix
- fail[0]=0
- i=1, j=0 p[0]!=p[1] fail[1](=j)=0
- i=2, j=0 p[j]=p[i], fail[2]=j+1=0+1=1, suffix length 1 same as prefix
- i=i+1=3, j=j+1=1 p[j]=p[i], fail[3]=j+1=1+1=2, suffix length 2 same as prefix
- i=i+1=4, j=j+1=2 p[j]=p[i], fail[3]=j+1=2+1=3, suffix length 3 same as prefix
- i=i+1=5, j=j+1=3 p[j]!=p[i], j=2 (2 is the point that the highest suffix = prefix)
- p[5]!=p[2], j=p[2]=1,
- p[5]!=p[1], j=p[1]=0
- i=i+1=6, p[j]=p[i], fail[6]=j+1=1
- i=i+1=7, j=j+1=1, p[j]=p[i], fail[7]=j+1=2
- i=i+1=8, j=j+1=2, p[j]=p[i], fail[8]=j+1=3
- i=i+1=9, j=j+1=3, p[j]=p[i], fail[9]=j+1=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
e	k	e	k	e	d	e	k	e	k	e	d	e	k	e	k	e	k
0	0	1	2	3	0	1	2	3	4	5	6	7	8	9	10	11	4

failure array example

- pattern: “akekedekekedekekek”
- fail[i]: which is the longest suffix that is also prefix
- i=i+1=9, j=j+1=3, p[j]=p[i], fail[9]=j+1=4
- i=i+1=10, j=j+1=4, p[j]=p[i], fail[10]=j+1=5
- i=i+1=11, j=j+1=5, p[j]=p[i], fail[11]=j+1=6
- i=i+1=12, j=j+1=6, p[j]=p[i], fail[12]=j+1=7
- i=i+1=13, j=j+1=7, p[j]=p[i], fail[13]=j+1=8
- i=i+1=14, j=j+1=8, p[j]=p[i], fail[14]=j+1=9
- i=i+1=15, j=j+1=9, p[j]=p[i], fail[15]=j+1=10
- i=i+1=16, j=j+1=10, p[j]=p[i], fail[16]=j+1=11 suffix length 11 same as prefix
- i=i+1=17, j=j+1=11, p[j]!=p[i], one position back p[10]=5 the longest suffix that is also prefix has length 5 so j=5
- i=17, j=5, p[j]!=p[i], one position back p[4]=3 the longest suffix that is also prefix has length 3 so j=3
- i=17, j=3, p[j]=p[i], fail[17]=j+1=4

Compute Failure Function

```
COMPUTEFAILURE(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

example

$j \leftarrow 0, i \leftarrow 1$	$\$^j$	A^i	B	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		**0**											...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A^j	B^i	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	**1**										...
$j \leftarrow fail[j]$	$\$^j$	A	B^i	R	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A^j	B	R^i	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	**1**									...
$j \leftarrow fail[j]$	$\$^j$	A	B	R^i	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A^j	B	R	A^i	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	**1**								...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B^j	R	A	C^i	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	**2**							...
$j \leftarrow fail[j]$	\$	A^j	B	R	A	C^i	A	D	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C^i	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A^j	B	R	A	C	A^i	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	**1**						...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B^j	R	A	C	A	D^i	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	**2**					...
$j \leftarrow fail[j]$	\$	A^j	B	R	A	C	A	D^i	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D^i	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A^j	B	R	A	C	A	D	A^i	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	**1**				...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B^j	R	A	C	A	D	A	B^i	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	**2**			...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B	R^j	A	C	A	D	A	B	R^i	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	**3**		...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B	R	A^j	C	A	D	A	B	R	X^i	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	**4**	...
$j \leftarrow fail[j]$	\$	A^j	B	R	A	C	A	D	A	B	R	X^i	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D	A	B	R	X^i	...

```
COMPUTEFAILURE(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

Compute Failure Function

- Is failure function computed correctly? Proof by Induction:
- Base case: fail[1]=0.
- Hypothesis: In line (*) fail[1] through fail[i-1] are correct.
- Induction step: is fail[i] correct?
- After *i*-th iteration of line (*) j=fail[i], so P[1..j-1] is the longest proper prefix of P[1..i-1] that is also a suffix.
- Definition of the iterated failure function $fail^c[j]$
- $fail^0[j]=j$, $fail^1[j]=fail(fail^0[j])=fail[j]$, $fail^c[j]=fail[fail^{c-1}[j]]$

$$fail^c[j] = fail[fail^{c-1}[j]] = \overbrace{fail[fail[\cdots[fail}^{c}[j]]\cdots]]$$

- Compute failure is a dynamic programming implementation of the following recursive implementation:

$$fail[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max_{c \geq 1} \{ fail^c[i-1]+1 \mid P[i-1] = P[fail^c[i-1]] \} & \text{otherwise.} \end{cases}$$