COMP261 Lecture 18

Lindsay Groves

String Searching 2 of 2

String search

- Simple search
  - Slide the window by 1
    - k = k +1;
- KMP
  - Slide the window faster
    - k = k + i – M[i]
  - Never recheck the matched characters
    - Is there a “suffix == prefix”?
      - No, skip these characters
        - » M[i] = 0
      - Yes, reuse, no need to recheck these characters
        - » M[i] is the length of the “reusable” suffix

abcdmndsjhhhsjgrjgslagfiigirnvkfir

abcdefg

ananfdfjoijtoiinkjjkjgfjgkjkkhgklhg

ananaba

Knuth Morris Pratt

input: string S[0 .. m-1], text T[0 .. n-1], *match table* M[0 .. m-1]
output: the position in T at which S is found, or -1 if not present
variables: k ← 0 *start of current match in T*
i ← 0 *position of current character in S*

```
  while   k + i  <  n
     if  S[ i ] = T[ k + i ]  then             //  match
          i ← i + 1
          if  i = m  then return  k           // found S
     else if M[ i ]  =  -1  then              // mismatch, no self overlap
          k ← k + i + 1, i ← 0
     else                                     // mismatch, with self overlap
          k ← k + i - M[ i ]                  // match position jumps forward
          i ← M[ i ]
return  -1     // failed to find S
```

How do we build the table?

- Need to know when there is a suffix of a failed match which is a prefix of the search string.
- abcmndsjhhhsjgrjgslagfiigirnvkfir
  abcefg
  - No. Resume checking at m.
- ananfdfjoijtoiinkjjkjgfjgkjkkhgklhg
  ananaba
  - Yes. Resume checking at second a.
- But a suffix of a partial match is also part of the search string!!
- So we can find possible partial matches by analysing the search string!

How do we build the table?

- Consider the search string `abcdabd`.
- Look for a proper suffix of failed match, which is a prefix of S, starting at each position in S
  – so suffix ends at previous position.
- `0: abcdabd`
  We can’t have a failed match at position 0.
  Special case, set M[0] to -1.
- `1: abcdabd`
  a not a proper suffix.
  Special case, set M[1] to 0.
- `2: abcdabd`
  b not a prefix, set M[2] to 0.

How do we build the table?

- `3: abcdabd`

  bc has no suffix which is a prefix, set M[3] to 0.
- `4: abcdabd`

  bcd has no suffix which is a prefix, set M[4] to 0.
- `5: abcdabd`

  a is longest suffice which is a prefix, set M[5] to 1.
- `6: abcdabd`

  ab is longest suffice which is a prefix, set M[6] to 2.
- Knowing what we matched before allows us to determine length of next match.

KMP – Partial Match Table

Index	0	1	2	3	4	5	6
S	a	n	a	n	a	b	a
M	-1	0	0	1	2	3	0

```
M[i] = pm(S[0…i-1], S);
pm(A, B){
    C = largest proper suffix of A
        which is also a prefix of B;
    return C.length;
}
```

KMP – partial matching table

Index	0	1	2	3	4	5	6
S	A	B	C	D	A	F	G
M	-1	0					

KMP – example

Index	0	1	2	3	4	5	6
S	A	B	C	D	A	B	D
M	-1	0	0	0	0	1	2

ABCDABD

ABCABCDAABABCDABCDABDE

KMP – example

Index	0	1	2	3	4	5	6
S	A	A	A	A	A	A	A
M	-1	0					

KMP: Build the partial match table.

M: 0 1 2 3 4 5 6
| | | | | | | |

input: S[0 .. m-1] *// the string*

output: M[0 .. m-1] *// match table*

initialise: M[0] ← -1 // -1 is just a flag for KMP
M[1] ← 0
j ← 0 *// position in prefix*
pos ← 2 *// position in table*

ananaba
abcdefg

abcdefg
ananaba

while pos < m
  if S[pos - 1] = S[j] *//substrings ...pos-1 and 0..j match*
    M[pos] ← j+1,
    pos ← pos+1, j ← j+1
  else if j > 0 *// mismatch, restart the prefix*
    j ← M[j]
  else *// j = 0* *// we have run out of candidate prefixes*
    M[pos] ← 0,
    pos ← pos+1

KMP: Building the table.

```
M:  0    1    2    3    4    5    6    7
    |    |    |    |    |    |    |    |   .
```

input: S[0 .. m-1] *// the string*

output: M[0 .. m-1] *// match table*

initialise: M[0] ← -1
M[1] ← 0
j ← 0 *// position in prefix*
pos ← 2 *// position in table*

```
andandba
andandba
```

while pos < m
- **if** S[pos - 1] = S[j] *//substrings ...pos-1 and 0..j match*
  - M[pos] ← j+1,
  - pos++, j++
- **else if** j > 0 *// mismatch, restart the prefix*
  - j ← M[j]
- **else** *// j = 0* *// we have run out of candidate prefixes*
  - M[pos] ← 0,
  - pos++

KMP – example (hard)

0	1	2	3	4	5	6	7	8	9	10
A	A	B	A	A	A	A	B	A	C	A

String search: Knuth Morris Pratt

- Searches forward,

- Never matches a text character twice (and never skips a text character)

- Jumps string forward based on self match within the string:
  - prefix of string matching a later substring.
  - doesn't use the character in the text to determine the jump

- Cost:

- What happens for the worst case for brute force search?

String search: Boyer Moore

- Searches backward
- Actually jump and skip many characters
- Use the characters in the text to determine the jump

abanana
alongpieceoftextwithnofruit

Boyer Moore: string search

abanana

- string: s[0] .. s[m-1]
- text: t[0] .. t[n-1]

bananfanlbananabananafan

Why look at every character in the text?

Start searching from the end of the string, backwards

When there is a mismatch,

move the string forward by an appropriate jump and restart:

table 1: what was the text character that mismatched?

⇒ what is the shortest jump that could make a match?

table 2: what has already been matched

⇒ what is the shortest jump that would match again

(take the longer of the two jumps suggested)