Information Retrieval Methods

Helena Ahonen-Myka
Spring 2007, part 8
Text-scanning methods
Translation from Finnish: Greger Lindén

1

In this part

- Text-scanning methods
  - Usage: searching for a query string in a text when the document collection is small
  - Methods
    - Brute force method
    - Fast string matching: MP, KMP, BM

2

Text-scanning systems

- When the document collection is large, the best way to implement a retrieval system is usually to use an inverted file
- If the document collection is small and can fit into main memory, we can also implement a search by comparing the query word directly to the text of the document
- Normal usage: post-processing of search results
  - E.g. implementing the proximity operator: do the search words occur close enough in the document?

3

Text-scanning systems

- The query word is searched through sequential scanning by comparing the characters in the word to the characters of the document starting from the first character in the document
- We assume, that document S is a string of characters
  - $S = s_0 s_1 \ldots s_{n-1}$, where s_i is a character of some vocabulary
- And the search pattern P is also a string
  - $P = p_0 p_1 \ldots p_{m-1}$, where p_j is a character of the vocabulary
- $m \leq n$

4

Example

- document S: abracabracadabra
- pattern P: abracadabra

5

Brute force method

```
abracabracadabra
abracadabra
 abracadabra
  abracadabra
   abracadabra
    abracadabra
     abracadabra
```

6

Brute force method

- In the worst case, the pattern will match the text in every character comparison until the last character of the pattern
  – We need n · m comparisons → O(nm)
- S = aaaaaaaaaab, P = aab
- Usually the pattern does not match the text in a certain position and this can be proven only after comparing a few characters

7

```
S = aaaaaaaaaab
P = aab
    ++-
-----------------------------------------------------
S = aaaaaaaaaab
P =  aab
      ++-
------------------------------------------------------
...
S = aaaaaaaaaab
P =         aab
            ++-
------------------------------------------------------
S = aaaaaaaaaab
P =          aab
             +++  (match!)
```

8

Fast string matching

- The brute force method moves the pattern only one character at each comparison
- The method does not benefit from any information about which characters the pattern contains
- More efficient methods analyse the pattern first and recognise repeated characters in the pattern
- Based on the analysis, the pattern can be moved several characters at each comparison
- methods: MP (Morris-Pratt), KMP (Knuth-Morris-Pratt) and BM (Boyer-Moore)

9

MP (Morris-Pratt)

- S = ... s_i s_{i+1} s_{i+2} s_{i+3} s_{i+4} s_{i+5} | s_{i+6} s_{i+7}...
- P = p_0 p_1 p_2 p_3 | p_4

- The first part of the pattern, $p_{0..3}$, is found in the text, but s_{i+6} and p_4 do not match
- An occurrence of a pattern P can start in this fragment at $s_{i+2..i+5}$ only if some prefix of P is identical to a suffix of the matching part of S

10

MP (Morris-Pratt)

- S = ...barba | papa...
- P = barba | ari
- P = ba rbaari

- S = ...sey | chelles...
- P = sey | moyr
- P = seymoyr

11

MP (Morris-Pratt)

- It is enough to analyse only the pattern, because
  – The prefix of the pattern has matched a text fragment
  – The suffix of the fragment is identical to the suffix of the prefix of the pattern, before the point where the characters differ

12

MP (Morris-Pratt)

- Preprocessing the pattern:
  - We look for the substrings in the pattern that are repeated
  - We construct a transition table mpNext
    - mpNext[i] tells which is the longest prefix of $P_{0..i-1}$ which is also a suffix of $P_{0..i-1}$
    - if the characters up to i-1 matched and ith did not → i – mpNext[i] positions can be safely skipped

13

```
void preMP (char *x, int m, int mpNext[]) {
        int i,j;
        i = 0;
        j = mpNext[0] = -1;
        while (i < m) {
                while (j > -1  &&  x[i] != x[j])
                        j = mpNext[j];
                mpNext[++i] = ++j;
        }
}
```

14

```
0 1  2 3 4 5  6 7  8
G C A G A G A G
  G                       C ≠ G,  -> mpNext[2] = 0
    G                     A ≠ G, -> mpNext[3] = 0
      G                   G = G, -> mpNext[4] = 1
      G C                 GA ≠ GC  -> j=mpNext[1]=0
        G                 A ≠ G, -> mpNext[5] = 0
          G               G = G, -> mpNext[6] = 1
          G C             GA ≠ GC  -> j=mpNext[1]=0
            G             A ≠ G, -> mpNext[7] = 0
              G           G = G, -> mpNext[8] = 1
```

15

MP (Morris-Pratt)

- Searching phase:
  - The algorithm moves a window over the text and a pointer inside the window
  - Each time a character matches, the pointer is advanced
    - If the pointer reaches the end of the window, a match is reported
  - Each time a character does not match, the window is shifted forward in the text, to the position given by mpNext
    - The position in the text does not change

16

```
Void MP (char *x, int m, char *y, int n) {
        int i, j, mpNext[XSIZE];

        preMP(x, m, mpNext);   /* Preprocessing */
        i = j = 0;             /* Searching */
        while (j < n) {
                while (i > -1  &&  x[i] != y[j])
                        i = mpNext[i];
                i++; j++;
                if (i >= m) {
                        OUTPUT(j – i);
                        i = mpNext[i];
                }
        }
}
```

17

MP (Morris-Pratt)

- P =   a b r a c a d a b r a
- next =   -1 0 0 0 1 0 1 0 1 2 3 4

- S = abracadab | babracadabra
- P = abracadab | r
- P =   ab | r   (the same comparison!)

18

KMP (Knuth-Morris-Pratt)

- The MP method can be optimized → KMP (Knuth-Morris-Pratt) method
- In preprocessing the pattern, we also require that the characters that follow the prefix and suffix parts are not identical

19

```
Void preKMP (char *x, int m, int kmpNext[]) {
        int i,j;
        i = 0;
        j = kmpNext[0] = -1;
        while (i < m) {
                while (j > -1  &&  x[i] != x[j])
                        j = kmpNext[j];
                i++; j++;
                if (x[i] == x[j])
                        kmpNext[i] = kmpNext[j];
                else
                        kmpNext[i] = j;
        }
}
```

20

KMP (Knuth-Morris-Pratt)

- P = a b r a c a d a b r a
- next = -1 0 0 -1 1 -1 1 -1 0 0 -1 4

- S = abracadab | babracadabra
- P = abracadab | r
- P = a
- P = abracadabra

21

```
0 1 2 3 4 5 6 7 8
G C A G A G A G
  G                 C ≠ G  -> kmpNext[1] = 0
    G               A ≠ G -> kmpNext[2] = 0
      G             G = G -> kmpNext[3] = -1
      G C           GA ≠ GC  -> kmpNext[4]=1
        G           A ≠ G
          G         G = G -> kmpNext[5] = -1
          G C       GA ≠ GC  -> kmpNext[6]=1
            G       A ≠ G
              G     G = G -> kmpNext[7] = -1
```

22

KMP (Knuth-Morris-Pratt)

- Searching phase
  - Like in MP
  - Only the transition table is different (kmpNext)

23

```
Void KMP (char *x, int m, char *y, int n) {
        int i, j, kmpNext[XSIZE];

        preKMP(x, m, mpNext);   /* Preprocessing */
        i = j = 0;              /* Searching */
        while (j < n) {
                while (i > -1  &&  x[i] != y[j])
                        i = kmpNext[i];
                i++; j++;
                if (i >= m) {
                        OUTPUT(j – i);
                        i = kmpNext[i];
                }
        }
}
```

24

KMP (Knuth-Morris-Pratt)

- Preprocessing of the pattern can be done in O(m) time
- The search algorithm analyses each character in the document and for each document character at most one character in the pattern → at most 2n comparisons
- → O(m + n)
- In practice KMP may not work better than the brute force method
- The method can easily be extended to a situation with several patterns
  - Occurrences of all patterns are searched at the same time

25

BM (Boyer-Moore)

- We can also compare the pattern and the text starting from the end of the pattern and continue toward its beginning → BM (Boyer-Moore) method
  - The KMP algorithm analyses the prefix of the pattern each time; the BM algorithm analyses the suffix of the pattern each time
- There are two principles on how to shift the pattern in relation to the text
  - Matching shift (aka good-suffix shift)
  - Occurrence shift (aka bad-character shift)
- Each principle tells how many positions can be shifted → the larger shift wins

26

BM (Boyer-Moore)

- Matching shift
  - Corresponds to the transition table of the KMP algorithm
  - We store for each suffix of the pattern information if it is repeated in the pattern
  - When we move through the pattern from the end to the start and we encounter a mismatch between the pattern and the text, we can safely shift the previous similar suffix of the pattern to this point

27

BM (Boyer-Moore)

- S = abraca<u>b</u> | abra…
- P = abracad | abra

- b ≠ d
- matching shift
  - ”abra” found → the pattern can be shifted safely 7 steps (the first ”abra” in the pattern can be moved to the location after the mismatch)

- S = abraca<u>b</u>abra…
- P =   abracadabra

28

BM (Boyer-Moore)

- Occurrence shift
  - assume that “c” is the character in the text at which the prefix of the pattern does not match
  - if ”c” occurs in the pattern, we can shift the pattern so that the “c” in the pattern matches the “c” in the text
  - if ”c” does not occur in the pattern, we can shift the pattern to the right of the “c” in the text

29

BM (Boyer-Moore)

- S = abraca<u>b</u> | abra…
- P = abracad | abra

- occurrence shift
  - if ”b” is part of the pattern, the closest b to the left in the pattern can be shifted to this point → the pattern can be shifted 5 steps
- S = abraca<u>b</u>abra…
- P =   abracadabra

30

BM (Boyer-Moore)

- Matching shift: 7 positions
- Occurrence shift: 5 positions
- We choose the larger shift, i.e. 7 positions

31

BM (Boyer-Moore)

- P: G C A G A G A G
- The vocabulary: A = {A, C, G, T}
- m = 8 (length of P)
- Occurrence shifts (bad character shifts) are stored in table bmBC

32

```
void preBmBc (char *x, int m, int BmBc[]) {
        int i;
        for (i=0; i < ASIZE; ++i)
                bmBc[i] = m;
        for (i=0; i < m-1; ++i)
                bmBc[x[i]) = m – i - 1;
}
```

33

P: G C A G A G A G

bmBC[A] = 8; bmBC[C] = 8;

bmBC[G] = 8; bmBC[T] = 8

bmBC[C] = 8 – 1 – 1 = 6;

bmBC[A] = 8 - 1 – 2 = 5; bmBC[G] = 8 – 1 – 3 = 4;

bmBC[A] = 8 – 1 – 4 = 3; bmBC[G] = 8 – 1 – 5 = 2;

bmBC[A] = 8 – 1 – 6 = 1;

A C G T

1 6 2 8

34

BM (Boyer-Moore)

- Matching shifts (good suffix shifts) are stored in table bmGs
- The computation uses a table suff
  - for 0 < i < m, suff[i] = max {k: x[i-k+1..i] = x[m-k..m-1]}
  - P: G C A G A G A G
  - suff[7] = 8; suff[6] = 0; suff[5] = 4; suff[4] = 0; suff[3] = 2; suff[2] = 0; suff[1] = 0, suff[0] = 1

35

```
void suffixes(char *x, int m, int *suff) {
        int f, g, i;
        suff[m - 1] = m;
        g = m - 1;
        for (i = m - 2; i >= 0; --i) {
                if (i > g && suff[i + m - 1 - f] < i - g)
                        suff[i] = suff[i + m - 1 - f];
                else {
                        if (i < g)
                                g = i;
                        f = i;
                        while (g >= 0 && x[g] == x[g + m - 1 - f])
                                --g;
                        suff[i] = f - g; } } }
```

36

```
void preBmGs(char *x, int m, int bmGs[]) {
        int i, j, suff[XSIZE];
        suffixes(x, m, suff);
        for (i = 0; i < m; ++i)
                bmGs[i] = m;
        j = 0;
        for (i = m - 1; i >= -1; --i)
                if (i == -1 || suff[i] == i + 1)
                        for (; j < m - 1 - i; ++j)
                                if (bmGs[j] == m)
                                        bmGs[j] = m - 1 - i;
        for (i = 0; i <= m - 2; ++i)
                bmGs[m - 1 - suff[i]] = m - 1 - i; }
```


P: G C A G A G A G

suff: 1 0 0 2 0 4 0 8

bmGS: 7 7 7 2 7 4 7 1


```
void BM(char *x, int m, char *y, int n) {
        int i, j, bmGs[XSIZE], bmBc[ASIZE];

        preBmGs(x, m, bmGs); preBmBc(x, m, bmBc); /* Preprocessing */

        j = 0;                                          /* Searching */
        while (j <= n - m) {
                for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
                if (i < 0) {
                        OUTPUT(j);
                        j += bmGs[0];
                }
                else
                        j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i); } }
```


BM (Boyer-Moore)

- BM does not necessarily analyse each character in the text
- Average number of comparisons
  O(n log(m) / m), worst case O(mn)
- Several alternations
  - We use occurrence shift principle only
  - We use occurrence shift only, but apply it to the character which is compared to the last character in the pattern (and not to the mismatched character)
  - As before, but we apply it to the character that follows the position of the last character in the pattern

The proximity operator

- We are searching for several words that occur closely together
- If we search for a phrase like “computer science”, we can do as when searching for single words; the space is just another character
- If the distance between and the order of the words vary, it is more productive to first search for the word that occurs more rarely and/or is longer
  - The other words are then checked if they are in the proximity of this word

In this part

- Text-scanning methods
  - A brute force method
  - The MP and KMP algorithms
  - The BM algorithm