Fast exact string matching

We will discuss

- Horspool algorithm
- Wu-Manber algorithm

Fast exact string matching (2)

This exposition has been developed by C. Gröpl, G. Klau, and K. Reinert based on the following sources, which are all recommended reading:

- Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, pages 15ff.
- A nice overview of the plethora of exact string matching algorithms with animations implemented in java can be found under http://www-igm.univ-mlv.fr/ ~lecroq/string.

Thoughts about string matching

Let's start with the classical exact string matching problem:

Find all occurrences of a given pattern $P = p_1, ..., p_m$ in a text $T = t_1, ..., t_n$, usually with $n \gg m$.

The algorithmic ideas of *exact* string matching are useful to know although in computational biology algorithms for *approximate* string matching or *indexing* methods are often in use. However, in online scenarios it is often not possible to precompute an index for finding exact matches.

String matching is known for being amenable to approaches that range from the extremely theoretical to the extremely practical. In this lecture we will get to know two very practical exact string matching algorithms.

Thoughts about string matching (2)

Some easy terminology: Given strings *x*, *y*, and *z*, we say that *x* is a *prefix* of *xy*, a *suffix* of *yx*, and a *factor* (:=substring) of *yxz*.



In general, string matching algorithms follow three basic approaches. In each a *search window* of the size of the pattern is slid from left to right along the text and the pattern is searched within the window. The algorithms differ in the way the window is shifted.



Thoughts about string matching (3)

1. *Prefix searching*. For each position of the window we search the longest prefix of the window that is also a prefix of the pattern.



 Suffix searching. The search is conducted backwards along the search window. On average this can avoid to read some characters of the text and leads to sublinear average case algorithms.



3. *Factor searching*. The search is done backwards in the search window, looking for the longest suffix of the window that is also a factor of the pattern.



Suffix based approaches

Idea of the Horspool algorithm

As noted above, in the suffix based approaches we match the characters from the back of the search window. Whenever we find a mismatch we can shift the window in a safe way, that means without missing an occurrence of the pattern.

We present the idea of the *Horspool* algorithm, which is a simplification of the *Boyer-Moore* algorithm.



Idea of the Horspool algorithm (2)

For each position of the search window we compare the last character β with the last character of the pattern. If they match we verify until we find the pattern or fail on the text character σ . Then we simply shift the window according to the next occurrence of β in the pattern.



Horspool pseudocode

Input: text T of length n and pattern p of length m**Output:** all occurrences of p in T

```
Preprocessing:

for c \in \Sigma do d[c] = m;

for j \in 1 \dots m - 1 do d[p_j] = m - j

Searching:

pos = 0;

while pos \leq n - m do

j = m;

while j > 0 \land t_{pos+j} = p_j do j - -;

if j = 0 then output "p occurs at position pos + 1"

pos = pos + d[t_{pos+m}];
```

The Horspool algorithm (2)

We notice two things:

- 1. The verification could also be done forward. Many implementations use builtin memory comparison instructions of the machines (i.e. memcmp). (The java applet also compares forward.)
- 2. The main loop can be "unrolled", which means that we can first shift the search window until its last character matches the last character of the pattern and then perform the verification.

Horspool example

pattern: announce text: cpmxannualxconferencexannounce bmBc: aceflmnoprux 7 1 8 8 8 8 2 4 8 8 3 8 attempt 1: cpmxannualxconferencexannouncee Shift by 3 (bmBc[u]) attempt 2: cpmxannualxconferencexannouncee Shift by 8 (bmBc[x]) attempt 3: cpmxannualxconferencexannouncee Shift by 2 (bmBc[n]) attempt 4: cpmxannualxconferencexannounce a....E Shift by 8 (bmBc[e])

```
attempt 5:
cpmxannualxconferencexannounce
                     ....e
Shift by 1 (bmBc[c])
```

```
attempt 6:
cpmxannualxconferencexannounce
                      ANNOUNCE
Shift by 8 (bmBc[e])
```

```
cpmxannualxconferencexANNOUNCE
```

```
String length: 30
Pattern length: 8
Attempts: 6
Character comparisons: 14
```

Experimental Map

(Gonzalo Navarro & Mathieu Raffinot, 2002)



* Backward Nondeterministic DAWG Matching algorithm, factor-based, not covered in this lecture ** Backward Oracle Matching, factor-based, not covered in this lecture

Thoughts about multiple exact string matching

The multiple exact string matching problem is:

Find all occurrences of a given set of *r* patterns $P = \{p^1, ..., p^r\}$ in a text $T = t_1, ..., t_n$, usually with $n \gg m_i$. Each p^i is a string $p^i = p_1^i, ..., p_{m_i}^i$. We denote with |P| the total length of all patterns, i. e. $|P| = \sum_{i=1}^r |p^i| = \sum_{i=1}^r m_i$.

As with single string matching we have three basic approaches:

- 1. *Prefix searching*. We read each character of the string with an automaton built on the set *P*. For each position in the text we compute the longest prefix of the text that is also a prefix of one of the patterns.
- 2. *Suffix searching*. A positions *pos* is slid along the text, from which we search backward for a suffix of any of the strings.
- **3**. *Factor searching*. A position *pos* is slid along the text from which we read backwards a factor of some prefix of size *lmin* of the strings in *P*.

Suffix based approaches

Suffix based approaches

For single string matching, the suffix based approaches are usually faster than the prefix based approaches, hence it is natural to extend them to sets of patterns. The first algorithm with sublinear expected running time was that of *Commentz-Walter* in 1979. It is a direct extension of the *Boyer-Moore* algorithm. The search window is verifed from right to left using a trie for the set of reversed patterns, $P^{\text{rev}} = \{(p^1)^{\text{rev}}, \dots, (p^r)^{\text{rev}}\}$. Again there are three rules to determine a safe shift.

The *Horspool* algorithm also has a straightforward extension to multiple patterns. But it is much less powerful matching than for a single pattern, because the probability to find any given character in one of the strings gets higher and higher with the number of strings. A stronger extension is the *Wu-Manber* algorithm which is in fact superior to other algorithms for most settings.

Do you see why the performace of the Horspool algorithm deteriorates quickly as the number of patterns increases?

Wu-Manber algorithm

Next we introduce a suffix-search based multi-pattern search algorithm, the *Wu-Manber* algorithm. Recall that we want to search simultaneously for a set of *r* strings $P = \{p^1, p^2, ..., p^r\}$ where each P^i is a string $p^i = p_1^i, p_2^i ... p_{m_i}^i$. Let *Imin* be the minimum length of a pattern in *P* and *Imax* be the maximum length. As usual we search in a text $T = t_1 ... t_n$.

The key idea of Wu and Manber is to use *blocks* of characters of length *B* to avoid the weakness of the Horspool algorithm.

For each string of length *B* appearing at the end of the window, the algorithm "knows" a safe shift (after some preprocessing).

Wu-Manber algorithm (2)

Instead of using a table of size $|\Sigma|^B$ each possible block is assigned a *hash value* which is used to store the blocks in hash tables. Note that $|\Sigma|^B$ can be quite a large number, and the distibution of blocks is usually very unevenly.

You can ignore the hash function for a moment to catch up the main idea (i.e., take the identity as a hash function). (Don't forget to think about it later, though.)

Wu-Manber algorithm (3)

The algorithm uses two tables *SHIFT* and *HASH*.

SHIFT(j) contains a *safe* shift, that means it contains the minimum of the shifts of the blocks *BI* such that $j = h_1(BI)$. More precisely:

- If a block BI does not appear in any string in P, we can safely shift Imin B + 1 characters to the right. This is the default value of the table.
- If *BI* appears in one of the strings of *P*, we find its rightmost occurrence in a string p^i , let *j* be the position where it ends, and set *SHIFT*($h_1(BI)$) to $m_i j$.



Wu-Manber algorithm (4)

To compute the *SHIFT* table consider separately each $p^i = p_1^i \dots p_{m_i}^j$. For each block $BI = p_{j-B+1}^i \dots p_j^i$, we find its corresponding hash value $h_1(BI)$ and store in *SHIFT*($h_1(BI)$) the minimum of the previous value and $m_i - j$.

During the search phase we can shift the search positions along the text as long as the *SHIFT* value is positive. When the shift is zero, the text to the left of the search position might be one of the pattern strings.

Wu-Manber algorithm (5)

The entry *j* in table HASH contains the indices of all patterns that end with a block *BI* with $h_2(BI) = j$. This table is used leter for the verification phase.

To access the possible strings, $HASH(h_2(BI))$ contains a list of all pattern strings whose last block is hashed to $h_2(BI)$. In the original paper Wu and Manber chose $h_1 = h_2$ to save a second hash value computation.

Wu-Manber pseudocode

- 1 WuManber($P = \{p^1, p^2, ..., p^r\}, T = t_1 t_2 ... t_n$)
- 2 // Preprocessing
- 3 Compute a suitable value of B (e.g. $B = \log_{|\Sigma|}(2 \cdot Imin \cdot r));$
- 4 Construct Hash tables SHIFT and HASH;
- 5 // Searching
- 6 pos = Imin;
- 7 while $pos \le n$ do
- 8 $i = h_1(t_{pos-B+1} \dots t_{pos});$
- 9 $\underline{if} SHIFT[i] = 0$
- 10 <u>then</u>
- 11 $list = HASH[h_2(t_{pos-B+1}...t_{pos})];$
- *Verify all patterns in list against the text;*
- *pos++*;
- *14* **<u>else</u>**

fi

- 15 pos = pos + SHIFT[i];
- 16
- 17 <u>od</u>

Wu-Manber example

Assume we search for $P = \{announce, annual, annually\}\$ in the text $T = CPM_annual_conference_announce$. Assume we choose B = 2 and a suitable hashfunction (exercise) and assume we are given the following tables:

Then the algorithm proceeds as follows:

Wu-Manber example (2)

- CPM_<u>an</u>nual_conference_announce
 SHIFT[an] = 4.
- CPM_annu<u>al</u>_conference_announce

SHIFT[al] = 0. List = HASH[al] = $\{2\}$. Compare p^2 against the string and mark its occurrence. Shift search positions by 1.

- CPM_annual_conference_announce
 SHIFT[I_] = 5.
- CPM_annual_con<u>fe</u>rence_announce
 SHIFT[fe] = 5.

Wu-Manber example (3)

- CPM_annual_conference_announce
 SHIFT[ce] = 0. List = HASH[ce] = {3, 1}.
 Compare p¹ and p³ against the text. No string matches. Shift by one.
- CPM_annual_conference_announce
 SHIFT[e_] = 5.
- CPM_annual_conference_announce SHIFT[ou] = 3.

CPM_annual_conference_announce
 SHIFT[ce] = 0. List = HASH[ce] = {3, 1}.
 Compare p¹ and p³ against the text. Test succeeds for p¹. Mark its occurrence.