Practical and Optimal String Matching

Kimmo Fredriksson

Department of Computer Science, University of Joensuu, Finland

Szymon Grabowski

Technical University of Łódź, Computer Engineering Department

Problem Setting

- The classic string matching problem:

  Given *text* $T[1..n]$ and *pattern* $P[1..m]$ over some finite alphabet Σ of size σ, find the occurrences of P in T.
- We focus on the case where m is relatively small

  $\Rightarrow$

  Bit-parallelism.

Previous work

Vast number of algorithms exist. Some of the most well-known are (classics):

Knuth-Morris-Prat: The first $O(n)$ worst case time algorithm.

Boyer-Moore(-Horspool)-family: Numerous variants, sublinear on average.

(bit-parallel:)

Shift-or: $O(n)$ for $m \leq w$ (Baeza-Yates & Gonnet, 1992).

BNDM family: $O(n \log_\sigma(m)/m)$ on average for $m \leq w$. SBNDM (Navarro, 2001; Peltola & Tarhio, 2003), LNDM (He & Fang, 2004), FNDM (Holub & Durian, 2005).

Previous work

- In practice, the best current algorithms for short patterns are the BNDM-family of algorithms (Navarro & Raffinot, 2000).

This work

- We develop a novel pattern partitioning technique that allows us to use shift-or while skipping text characters.
- The algorithm has optimal $O(n \log_\sigma(m)/m)$ average case running time if $m \leq w$.
- Very simple to implement, simple inner loop (comparable to plain shift-or)

  $\Rightarrow$

  very efficient in practice.

- $O(mn)$ worst case, but can be improved to $O(n)$ without destroying the simplicity of the search algorithm.

Our algorithm: the idea

The algorithm is based on the preprocessing / filtering / verification paradigm.

- The preprocessing phase generates q different alignements of the pattern, each containing only every qth pattern character.

  I.e. we partition the pattern into q pieces.
- The filtering phase searches all the q pieces in parallel using shift-or algorithm, reading only every qth text character.
- If any of the q pieces match, then we invoke a verification algorithm.

Preprocessing

- Given a pattern P, generate a set $\mathcal{P} = \{P^0, \ldots, P^{q-1}\}$ of q patterns as follows:

$$P^j[i] = P[j + iq], \quad j = 0 \ldots q-1, \quad i = 0 \ldots \lfloor m/q \rfloor - 1.$$

- I.e. we generate q different alignments of the original pattern P, each alignment containing only every qth character.
- Each new pattern has length $m' = \lfloor m/q \rfloor$.
- The total length of the patterns is $q\lfloor m/q \rfloor \leq m$.
- For example, if $P = \texttt{abcdef}$ and $q = 3$, then

$P^0 = \texttt{ad}$, $P^1 = \texttt{be}$ and $P^2 = \texttt{cf}$.

Preprocessing: the rationale

Assume that P occurs at $T[i..i+m-1]$.

$\Rightarrow$

$$P^j[h] = T[i+j+hq], \quad j = i \bmod q, \quad h = 0 \ldots m'-1.$$

$\Rightarrow$

(1) We can use the set $\mathcal{P}$ as a filter for the pattern P

(2) The filter needs to scan only every qth character of T.

Preprocessing: the rationale

P: a b c d e f

i p

T: x x a b c d e f x x x

P^0: a d

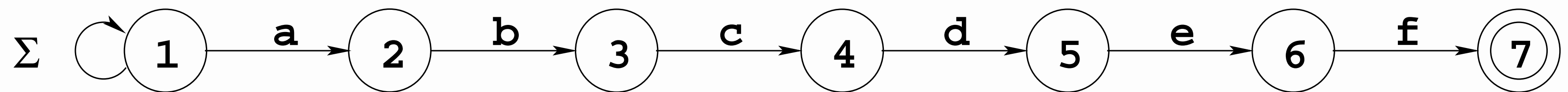
P^1: b e

P^2: c f

P': a d b e c f

Prelude to filtering: Shift-or algorithm

- The algorithm is based on a non-deterministic automaton. The automaton for $P =$ '`abcdef` is:



- The transitions are encoded in a table B of bit-masks: For $1 \leq i \leq m$, the mask $B[c]$ has the ith bit set to 0, iff $P[i] = c$.
- The bit-vector D has one bit per state in the automaton, the ith bit of the vector is set to 0, iff the state i is active (initially all bits are 1).
- It can be shown that the automaton can be simulated as:

$$D \leftarrow (D << 1) \mid B[T[i]]$$

Prelude to filtering: Shift-or algorithm

- If after the simulation step, the mth bit of D is zero, then P occurs at $T[i-m+1...i]$.

  Can be detected as

  $$(D \mathbin{\&} mm) \neq mm$$

  where mm has only the mth bit set.

- Clearly each step of the automaton is simulated in time $O(\lceil m/w \rceil)$, which leads to $O(n\lceil m/w \rceil)$ total time.

Filtering

- The whole set $\mathcal{P}$ of patterns can be searched simultaneously using the Shift-or algorithm (Baeza-Yates & Gonnet, 1992).
- All the patterns are preprocessed together, as if they were concatenated:

  For $P = \texttt{abcdef}$, we effectively preprocess a pattern $P' = P^0\ P^1\ P^2 = \texttt{adbecf}$.
- If the pattern P^j matches, then the $(j+1)m'$-th bit in D is zero. This can be detected as

$$(D \ \&\ mm) \neq mm$$

  where mm has every $(j+1)m'$-th bit set to 1.

Filtering: the simplicity illustrated

- Plain shift-or search:

1 $D \leftarrow {\sim}0;\ i \leftarrow 0$
2 **while** $i < n$ **do**
3 $\quad D \leftarrow (D << 1) \mid B[T[i]]$
4 $\quad$ **if** $(D \ \&\ mm) \neq mm$ **then** report match
5 $\quad i \leftarrow i + 1$

- Our shift-or search:

1 $D \leftarrow {\sim}0;\ i \leftarrow 0$
2 **while** $i < n$ **do**
3 $\quad D \leftarrow ((D \ \&\ {\sim}mm) << 1) \mid B[T[i]]$
4 $\quad$ **if** $(D \ \&\ mm) \neq mm$ **then** Verify
5 $\quad i \leftarrow i + q$

Verification

- If any of the pattern pieces in $\mathcal{P}$ match, we verify if the original pattern P matches (with the corresponding alignement).
- Can be done by brute force algorithm, with $O(m)$ worst case cost.

Complexity

- The filtering time is $O(n/q)$.
- Assuming that each character occurs with probability $1/\sigma$, the probability that P^j occurs in a given text position is $(1/\sigma)^{\lfloor m/q \rfloor}$.

  $\Rightarrow$

  The verification cost is on average at most $O(mn/\sigma^{m/q})$
- We select q so that $mn/\sigma^{m/q} \leq n/q$, i.e. $q = O(m/\log_\sigma m)$.

  $\Rightarrow$

  Total average time is $O(n \log_\sigma m/m)$, which is optimal.

Long patterns

- If $qm' > w$, we must use several computer words

  $\Rightarrow$ Asymptotic running time becomes $O(n \log_\sigma(m)/w)$.

- The trick in (Peltola & Tarhio, 2003) to make BNDM work with $m > w$ can be applied to our algorithm too.

  $\Rightarrow$ Omitting the details, we obtain $O(n \log_{\sigma/h}(m)/m)$ average time where $h = \lfloor (m-1)/w \rfloor + 1$.

- Not optimal anymore.

Linear worst case time

- The worst case running time is $O(mn)$.
- Use any $O(n)$ worst case time algorithm for the verifications, and do the verifications incrementally, saving the search state of the worst case algorithm after each verification.

  $\Rightarrow$

  'Standard trick', worst case becomes $O(n)$.

- Not a real problem: if verification time is a problem, then the filter does not work well, and can use the linear time algorithm instead.

Implementation

- In modern pipelined CPUs branching is costly.
  $\Rightarrow$
  Unroll U times (i.e. repeat inline U times) the code
  $D \leftarrow (D << 1) \mid B[T[i]]$.
  $\Rightarrow$
  The bit positions indicating the occurrences will overflow
  $\Rightarrow$
  Reserve $U - 1$ extra bits per pattern to avoid interference.
  $\Rightarrow$
  $q(U - 1 + \lfloor m/q \rfloor)$ bits in total.
- Verification is done only every Uth step, for those (at most U) alignements that could match.
- Much faster in practice.

Experimental results

- Implementation in C, compiled using `icc 8.1` with full optimizations, run in a 2.4GHz Pentium 4 ($w = 32$), with 512MB RAM, running Linux 2.4.20-8.
- 100 patterns were randomly extracted from the text.
- Each pattern was then searched for separately.
- We report the average speed in megabytes per second.
- Our data: real DNA and protein data, English natural language and random ASCII text ($\sigma = 96$).

Experimental results

We compared against:

BNDM: (Navarro & Raffinot, 2000), competitive only for random ASCII.

SBNDM: Simplified version of BNDM (Peltola & Tarhio, 2003), competitive only for random ASCII.

BMH, BMHS: Boyer-Moore-Horspool, and the Sunday variant of BMH.
Not competitive on any data (results omitted).

Our algorihtms:

AOSO: Our basic algorithm...

FAOSO: ...with loop-unrolling.

Experiments: DNA

m, q	AOSO	FAOSO	BNDM	SBNDM
$4, 2$	321	**503**	181	210
$8, 2$	539	**763**	312	357
$12, 3$	702	**941**	438	492
$16, 3$	1029	**1229**	567	598
$20, 4$	1079	**1341**	750	804
$24, 4$	1229	**1525**	1106	1164
$28, 5$	1427	**1638**	1106	1164

Experiments: proteins

m, q	AOSO	FAOSO	BNDM	SBNDM
$4, 2$	580	**909**	415	512
$8, 4$	944	**1267**	642	678
$12, 4$	1120	**1376**	816	926
$16, 4$	1120	**1459**	963	1025
$20, 4$	1235	**1376**	1175	1204
$24, 5$	1267	**1338**	1235	1302
$28, 6$	**1302**	**1302**	**1302**	**1302**

Experiments: natural language

m, q	AOSO	FAOSO	BNDM	SBNDM
$4, 2$	579	**884**	368	476
$8, 4$	1034	**1262**	685	778
$12, 4$	1144	**1279**	797	845
$16, 5$	1200	**1389**	831	944
$20, 6$	1279	**1389**	1013	1092

Experiments: random ASCII

m, q	AOSO	FAOSO	BNDM	SBNDM
$4, 2$	599	952	633	**1053**
$8, 4$	1124	**1333**	1064	1220
$12, 4$	1250	**1389**	1299	1282
$16, 4$	1351	**1389**	1351	**1389**
$20, 6$	1449	**1471**	1370	1429

Conclusions

- Very simple to implement.
- Very efficient in practice.
- Optimal for short patterns ($m \leq w$).
- The techniques can be adapted for several other algorithms as well, e.g.
  - Shift-add (for Hamming distance): $O(n(k + \log_\sigma(m))/m)$ average time.
- Any algorithm for multiple string matching can be used in place of Shift-or.