Boosting Pattern Matching Performance via k-bit Filtering

M. Oğuzhan Külekci [1] Jeffrey Scott Vitter [2] Bojian Xu [3]

[1] National Research Institute of Electronics & Cryptology, Turkey
kulekci@uekae.tubitak.gov.tr

[2] Dept. of Elect. Eng. and Computer Science, The University of Kansas, USA
jsv@ku.edu

[3] Dept. of Computer Science and Engineering, Texas A&M University, USA
bojianxu@tamu.edu

The 25th International Symposium on Computer and Information Sciences, 2010

Outline

1. Introduction
   - Motivation
   - Statement of the problem
   - Related literature
2. The Method
   - Text decomposition
   - Pattern matching on k-bit filtered text
3. Experimental Results
4. Conclusions

Motivation

How do we write a piece of text on a paper?

- Letter-by-letter in a sequential manner.
- Strictly preserving the order of appearance.

Why?

- Easy to read and follow in human convention.
- How would it be if the next letter is on a different place?

Motivation

How do we store a file on digital media?

- Byte-by-byte in a sequential manner.
- Strictly preserving the order of appearance.

The same way we do while writing on a paper!!!

Why?

- Simple (both from hardware and software perspectives).
- Preserves locality of reference (good for edit performance).

Motivation

The difference while reading...

The reader

- reads directly from paper,
- reads indirectly from digital media (word processors convert from internal representation to human readible format)

The Problem

In general

Find an alternative storage format for a file type, and device algorithms accordingly aiming to enhance the performance of specific operations on that file type.

In particular, this study focuses on ...

an alternative way of storing text files so as to speed up the pattern matching performance.

In this study

We propose to

store the file in a different format that will help us in pattern matching.

and does not propose (yet)

- creating an index or any other auxilary data structure over the file (may further improve performances on the proposed format)
- any compression or succinct data structure over the file

Related Literature

Pattern Matching

Locate and/or count the occurrences of a queried pattern P on text T.

- Off–line PM : Preprocess P, scan T. (Charras et al. , Handbook of exact string matching algorithms)
- On–line PM : Build an index for T, search P by using that index. (Hon et al., Compressed text indexing CPM'10)

There also exist semi-indexing approaches, e.g., speeding up pattern matching by text sampling

Related Literature

Filter-then-search paradigm

A powerful mechanism in off-line pattern matching

- Filter T for possible positions of occurences
- Verify those detected positions against P

A good filter should

- be easy to compute, e.g., hash based heuristics (Karp&Rabin'87, Manber&Wu'92 (agrep), Lecroq'07 (q-hash)) or benefit from processors architectural properties (Fredrikksonn'07 (FAOSO), Kulekci'09 (SSEF))
- call verification routine as less as possible

Basic Notation

- Alphabet $\leftarrow \Sigma$, Alphabet size $\leftarrow |\Sigma| = \sigma$
- File $F = s_1 s_2 s_3 \ldots s_n$.
- Byte $s_i = b_i^1 b_i^2 b_i^3 b_i^4 b_i^5 b_i^6 b_i^7 b_i^8$.
- $k \leftarrow$ the number of bits to be filtered out from each byte.
- $R = \{r_1, r_2, \ldots r_k\} \leftarrow$ the indices of the filter bits.

k-bit filtered file format

Assume $k = 2$ and $R = \{3, 5\}$.

s_1	s_2	...	s_n
$b_1^1 b_1^2 b_1^3 b_1^4 b_1^5 b_1^6 b_1^7 b_1^8$	$b_2^1 b_2^2 b_2^3 b_2^4 b_2^5 b_2^6 b_2^7 b_2^8$	...	$b_n^1 b_n^2 b_n^3 b_n^4 b_n^5 b_n^6 b_n^7 b_n^8$

Original file

$b_1^3 b_1^5 \; b_2^3 b_2^5 \ldots b_n^3 b_n^5$	$b_1^1 b_1^2 b_1^4 b_1^6 b_1^7 b_1^8 \; b_2^1 b_2^2 b_2^4 b_2^6 b_2^7 b_2^8 \ldots b_n^1 b_n^2 b_n^4 b_n^6 b_n^7 b_n^8$
Filter Part	*Payload Part*
$k \cdot n$ bits	$(8 - k) \cdot n$ bits

k-bit filtered file

Main idea

- Select most informative k bit positions.
- Move those bits of each byte to the beginning of the file.
- Remaining bits form the payload.
- Given a queried pattern P, search its most informative bits in the filter part, and then verify those detected positions by the payload.

How to detect most informative k-bit positions? (Theoretical)

- Given the value of the jth-bit, what is the probablity that the character is c_i, where $c_i \in \Sigma$?
  $P(c_i|b_j) = \frac{number\ of\ occurrences\ of\ c_i}{total\ number\ of\ characters\ having\ their\ j^{th}\ bit\ set\ to\ b_j}$
- The entropy (information content) of bit position $j, 1 \leq j \leq 8$, is
  $H_0(b_j) = -\sum_{c_i \in \Sigma} P(c_i|b_j) \cdot \log P(c_i|b_j)$
- We are looking for the top k positions.

How to detect the most informative k-bit positions? (Practical)

- We can use compression to detect most informative bits.
- Compression ratio is inversely proportional with the information content.
- Consider the sequences $B^i = b^i_1 b^i_2 \ldots b^i_n$, for $1 \leq i \leq 8$.
- The most informative bit is the one with the smallest compression ratio.
- Select the first k positions sorted according to increasing compression ratio.

Decompose pattern

- Apply same decomposition on queried pattern $P = p_1 p_2 \ldots p_m$ of m bytes.

p_1	p_2	$\ldots$	p_m
$\rho_1^1\rho_1^2\rho_1^3\rho_1^4\rho_1^5\rho_1^6\rho_1^7\rho_1^8$	$\rho_2^1\rho_2^2\rho_2^3\rho_2^4\rho_2^5\rho_2^6\rho_2^7\rho_2^8$	$\ldots$	$\rho_n^1\rho_m^2\rho_m^3\rho_m^4\rho_m^5\rho_n^6\rho_n^7\rho_n^8$

Input pattern

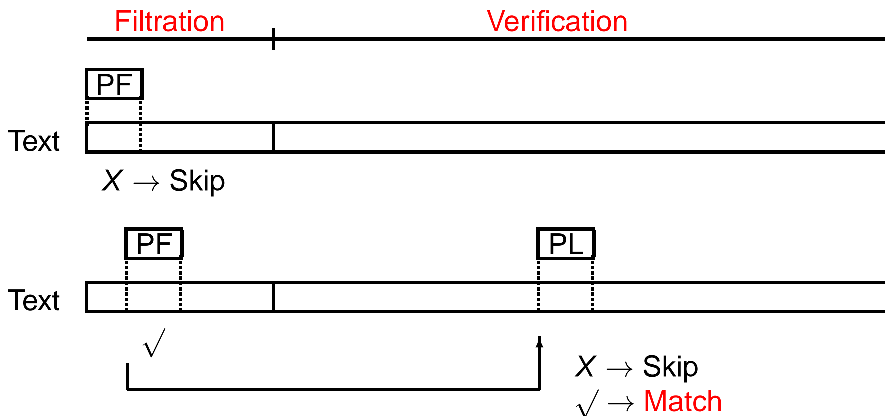
$\rho_1^3\rho_1^5\ \rho_2^3\rho_2^5\ \ldots\ \rho_n^3\rho_n^5$	$\rho_1^1\rho_1^2\rho_1^4\rho_1^6\rho_1^7\rho_1^8\ \rho_2^1\rho_2^2\rho_2^4\rho_2^6\rho_2^7\rho_2^8\ \ldots\ \rho_n^1\rho_n^2\rho_n^4\rho_n^6\rho_n^7\rho_n^8$
*Pattern Filter **PF***	*Pattern Payload **PL***
$k \cdot m$ bits	$(8-k) \cdot m$ bits

Decomposed pattern

The search algorithm

- Decompose P into PF and PL
- Slide PF over the filter part of the text
- On matching positions, verify PL against the respective position on the payload of the text. If verified, report a match.

The search algorithm



Keep on sliding *PF* until the end of the filtration block.

Bit-pattern search

- The filtration phase requires searching PF bitvector in the filter part of the file. We need a bitwise pattern matching algorithm.
- Some of the recent studies on bit vector matching are Klein&Ben-Nissan'07, Kim *et al.*'07, Faro&Lecroq'09
- Note that *PF* can only begin at some specific bit positions according to k, e.g. if $k = 2$, *PF* can begin at bits $1, 3, 5 \dots$
- Thus, we have implemented and used the bitwise variant of the BLIM (Kulekci'08) algorithm.

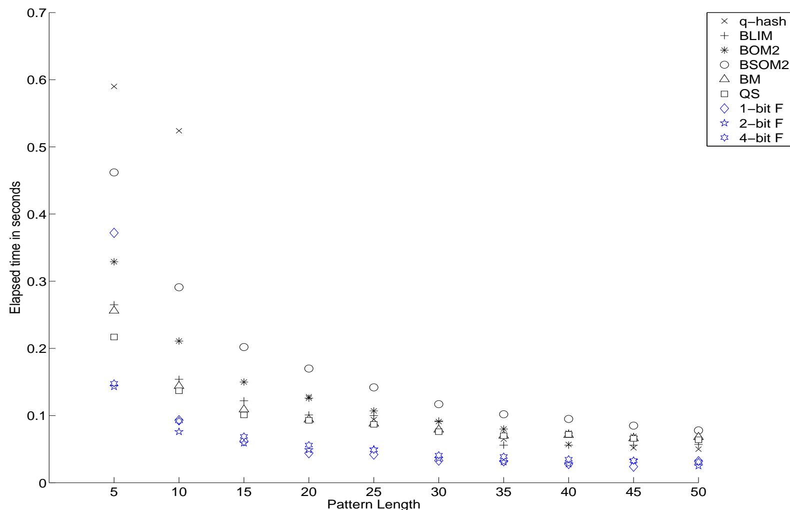
Bit-pattern search

- The original BLIM algorithm: Kulekci'10 (*BLIM: A new bit-parallel pattern matching algorithm overcoming the computer word size limitation, Mathematics in Computer Science, 2010*).
- Please refer to the journal version of this study that is to appear in *Computer Journal* (*Fast pattern matching via k-bit filtering based text decomposition) for the details of the bitwise variant of BLIM algorithm.*

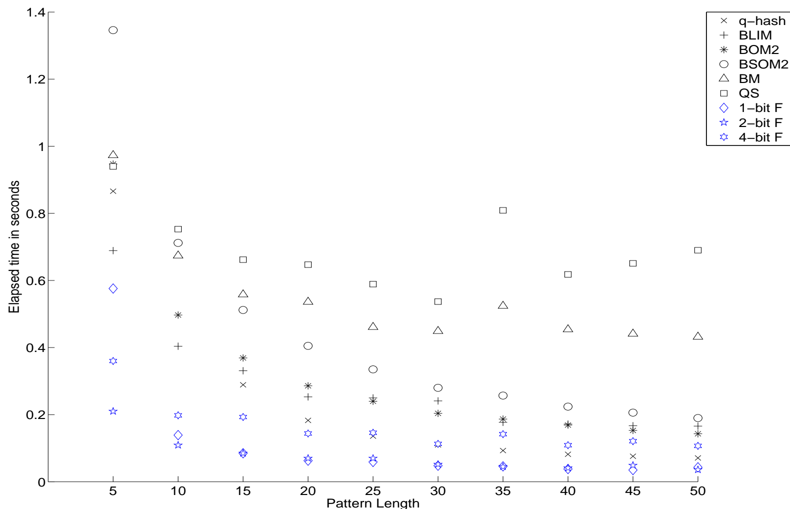
Experimental Setup

- Pattern matching performance analyzed on natural language text, DNA sequences (ASCII coded), and random byte sequences.
- All files are 30 MB in size
- Patterns of length 5 to 50 are searched over the files, each for 10 times, average times are reported
- Benchmarked with algoritms that are known to be fast

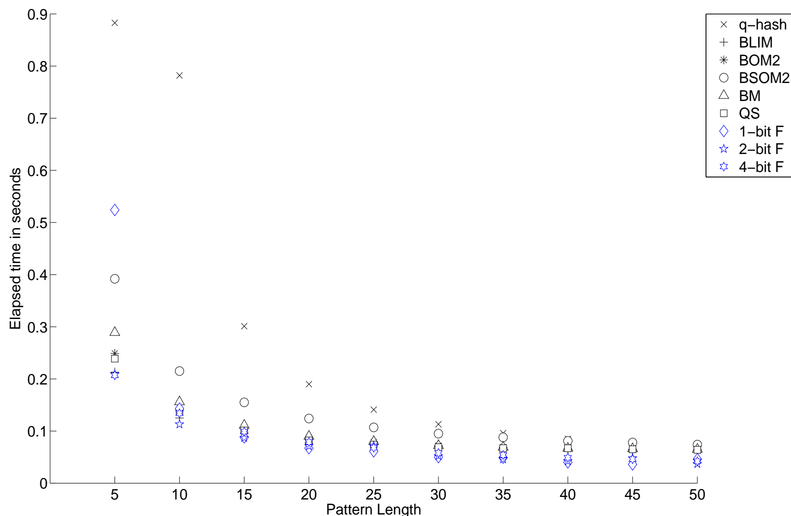
Results on English Language Text



Results on DNA sequences



Results on random byte sequences



Speed-up observed

Pattern Length	English			ASCII DNA			Random Byte		
	1-bit	2-bit	3-bit	1-bit	2-bit	3-bit	1-bit	2-bit	3-bit
5	0.58	1.51	1.46	1.19	3.28	1.91	0.40	1.01	1.02
10	1.47	1.80	1.48	2.90	3.70	2.04	0.87	1.10	0.93
15	1.62	1.71	1.46	3.40	3.40	1.49	1.09	1.10	0.96
20	2.11	1.89	1.66	2.90	2.61	1.27	1.14	1.08	0.97
25	2.07	1.74	1.77	2.30	1.94	0.93	1.21	1.02	1.08
30	2.30	2.11	1.85	2.29	2.11	0.97	1.38	1.32	1.18
35	1.75	1.80	1.43	2.02	2.06	0.65	1.30	1.39	1.18
40	2.00	2.00	1.60	2.15	2.00	0.75	1.60	1.52	1.30
45	2.16	1.57	1.57	2.17	1.55	0.62	1.77	1.33	1.39
50	1.56	2.00	1.61	1.65	1.97	0.66	1.36	1.70	1.50
Average	1.76	1.81	1.59	2.30	2.46	1.13	1.21	1.26	1.15

Conclusions and Future Work

Conclusions

- Proposed an alternative storage format to speed up searching.
- Approximately doubling the speed on exact pattern matching.

Future Work

- Some other transformations to speed up some other operations (e.g. compression, approximate matching, ...)?
- Dynamic versus static filtering (the most informative bit positions of individual characters may differ)?

Thank you!