Computing Patterns in Strings I: Specific, Generic, Intrinsic

Bill Smyth[1,2,3]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada
email: smyth@mcmaster.ca

[2] Digital Ecosystems & Business Intelligence Institute
Curtin University, Perth, Western Australia
email: b.smyth@curtin.edu.au

[3] Department of Computer Science
King's College London, UK

DEBII 2008

Outline

Outline

Outline

Outline

Outline

Outline

Computing patterns in strings constitutes the combinatorial nuts and bolts of many more general technologies: pattern “recognition”, data mining, data compression, bioinformatics, cryptography, information retrieval, security systems.

In this series of three lectures, I give a nontechnical overview, guaranteed intelligible to the non-mathematician, of these methods, organized into three categories:

* specific patterns (pattern-matching);
* generic patterns (“regularities” in strings);
* intrinsic patterns (always there, they make things happen!).

Outline

What is a string?

A string is just a sequence of "letters" (symbols) drawn from some (finite or infinite) "alphabet" (set):

- a word in the English language, whose letters are the upper and lower case English letters;
- a text file, whose letters are the ASCII characters;
- a book written in Chinese, whose letters are Chinese ideograms;
- a computer program, whose elements are certain "separators" (space, semicolon, colon, and so on) together with the "words" between separators; also a compiled .exe program;
- a DNA sequence, perhaps three *billion* letters long, containing only the letters C, G, A and T, standing for the nucleotides cytosine, guanine, adenine and thymine, respectively;
- a stream of *trillions* of bits beamed from a space vehicle;
- a list of the lengths of the sides of a convex polygon, whose values are drawn from the real numbers.

Why are strings important?

Because everything is a string!

Examples

* Fibonacci

$$
\begin{array}{cccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\boldsymbol{f} = & a & b & a & a & b & a & b & a & a & b & a & a & b & \cdots
\end{array}
$$

* WWW (courtesy Lewis Carroll)

1	2	3	4	5	6	7	8	9	10	
'Twas	*brillig*	*and*	*the*	*slithy*	*toves*	*did*	*gyre*	*and*	*gimble*	···

* highly periodic

$$001010010110100101001011010010100 \cdots$$

Conferences on String Processing

* AFL: International Conference on Automata & Formal Languages
* CIAA: International Conference on Implementation & Application of Automata;
* CPM: Symposium on Combinatorial Pattern Matching;
* DLT: Developments in Language Theory;
* ECCB: European Conference on Computational Biology;
* FSMNLP: Finite-State Methods & Natural Language Processing;
* LATA: International Conference on Language & Automata Theory & Applications;
* LSD: London Stringology Days;
* PSC: Prague Stringology Conference;
* SPIRE: Symposium on String Processing and Information Retrieval;
* StringMasters (@ McMaster): “How long is a piece of string?”
* WABI: Workshop on Algorithms in Bioinformatics;
* WORDS: International Conference on Words.

In 1980 AFL started, the next one was CPM in 1990 — all the others have started since then.

Important Ideas

* combinatorial
* specific
* generic
* intrinsic

Outline

The Pattern-Matching Task

The problem: to find all occurrences of a given pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in a given text $\boldsymbol{x} = \boldsymbol{x}[1..n]$ – hundreds, if not thousands, of algorithms have been proposed. More than 30 are given, with descriptions and C code, at

`http://www-igm.univ-mlv.fr/ lecroq/string/index.html`

Two approaches:

(1) skip sections of $\boldsymbol{x}$ where $\boldsymbol{p}$ cannot occur;
(2) use the bit-parallel property of computer words to go through $\boldsymbol{x}$ fast!
(3) hybrid – do first one, then the other.

Two modes:

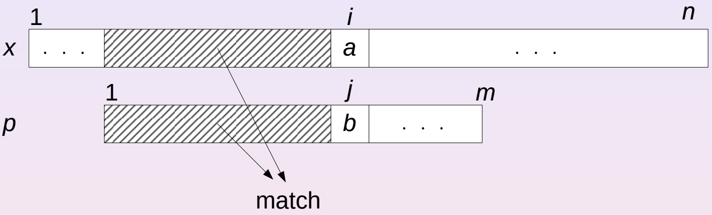
(1) exact – *issi* occurs twice in *Mississippi*;
(2) approximate – *ipsi* occurs three times with one error in *Mississippi*!

Outline

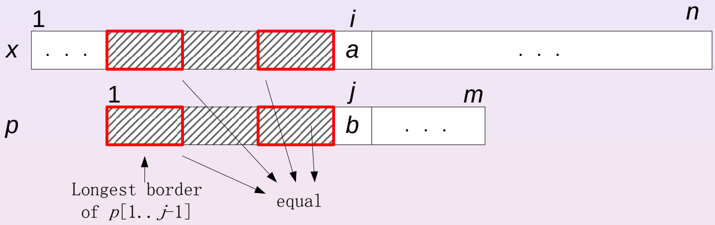
The Knuth-Morris-Pratt Algorithm

- The most famous pattern-matching algorithm.
- Preprocessing: compute the border of every prefix of $\boldsymbol{p}$.
- Requires at most $2n$ letter comparisons (linear!).
- However, not very fast in practice.

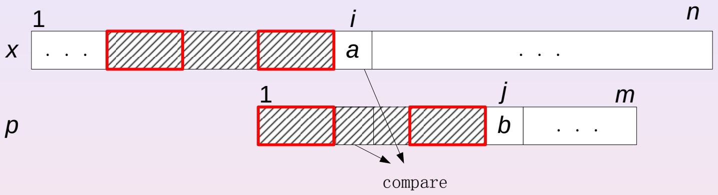
The KMP Algorithm - 1



The KMP Algorithm - 2



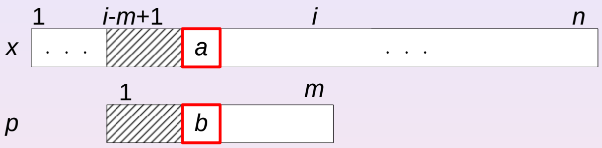
The KMP Algorithm - 3



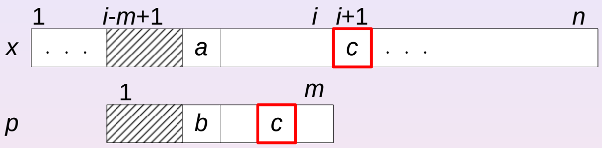
Sunday Variant of Boyer-Moore

- A simplified version of the Boyer-Moore algorithm.
- Preprocessing: find the rightmost occurrence of each letter in $\boldsymbol{p}$.
- Time complexity $O(mn)$.
- However, very fast in practice.

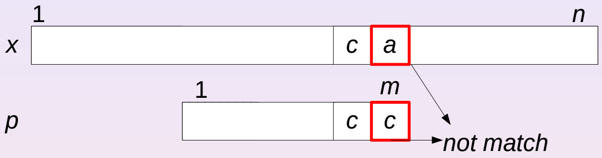
The BMS Algorithm - 1



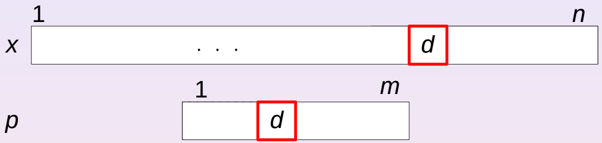
The BMS Algorithm - 2



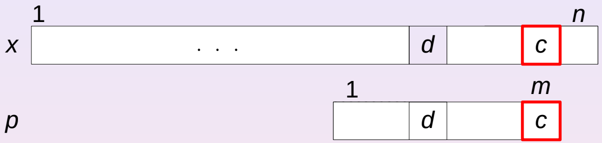
The BMS Algorithm - 3



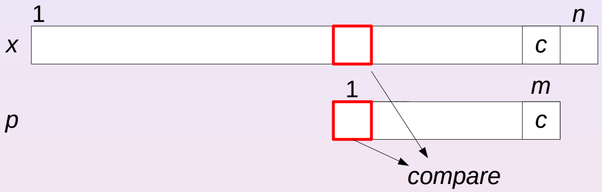
The BMS Algorithm - 4



The BMS Algorithm - 5



The BMS Algorithm - 6



The DBG Shift-Or Algorithm

- Makes use of the bit-parallel nature of computer words.
- Preprocessing: computes a bit array $\boldsymbol{B}$ identifying each letter in $\boldsymbol{p}$.
- Time complexity $O(mn/w)$, where w is the computer word length.
- Fast for shorter patterns.
- Very flexible – easily modified for approximate matching.

DBG Shift-Or

		$\boldsymbol{B}$		
$\boldsymbol{p} \backslash \Sigma$	A	C	G	T
A	0	1	1	1
A	0	1	1	1
T	1	1	1	0
C	1	0	1	1
G	1	1	0	1

Preprocess($\boldsymbol{B}$)
$\boldsymbol{s}[0..m] \leftarrow 0(1^m)$ — *initialize state vector*
for $i \leftarrow 1$ **to** n **do**
    $\boldsymbol{s} \leftarrow rightshift(\boldsymbol{s}, 1) \lor \boldsymbol{B}\big[1..m, \boldsymbol{x}[i]\big]$
    if $\boldsymbol{s}[m] = 0$ **then output** $i - m + 1$

At each step the state vector $\boldsymbol{s}$ is shifted right one bit (0 enters from the left) and a logical OR is done with the column of $\boldsymbol{B}$ corresponding to the current position i in $\boldsymbol{x}$.

Franek-Jennings-Smyth

- Combines KMP & BMS.
- So inherits the merits of both algorithms: very fast both asymptotically ($O(n)$) and in practice.

Here it is:

1. Perform **Sunday** shift along text.
2. When a match of letters is found at the end of the pattern, switch to **KMP** matching.
3. Continue **KMP** matching until no border can be used, then switch back to **Sunday** shift.

Smyth-Wang-Yu

- Combines BMS & Shift-Or.
- Adapts to the nature of the text, thus on average faster than either component.

Here it is:

1. Perform **Sunday shift** along text.
2. When a match of letters is found at the end of the pattern, switch to **Shift-Or matching**.
3. Continue **Shift-Or** until no match can be found at the current position (the state vector $\boldsymbol{s}$ is all ones), then skip to next possible position and switch back to **Sunday** shift.

Outline

Overview

There are four main paradigms of approximate pattern-matching:

- pattern $\boldsymbol{p}$ and text $\boldsymbol{x}$ are well-defined; handled by dynamic programming in $O(mn)$, maybe $O(n \log m)$, time;
- letters in either $\boldsymbol{p}$ or $\boldsymbol{x}$ may be indeterminate; handled by modifications to exact pattern-matching algorithms (especially DBG & Sunday);
- letters are exact or indeterminate, with bounds given on both maximum and total distance (for example, (δ, γ)-matching of musical texts in musical databases);
- match is exact but positions may be scrambled (Abelian matching); handled by convolutions.

Applications of Approximate Matching

- Recognition/correction of misspellings or word inversion in database/internet search.
- Tolerance of transcription errors in DNA sequences copied elsewhere in the genome.
- Appropriate handling of legitimate ambiguity, such as in protein/DNA entries, or in spelling variants (among dialects, or between past and present — for example, "itemise" and "itemize", or Smyth and Smith).
- Matching of inherently approximate texts, such as musical passages or rhythms (to detect plagiarism, for example).

$\boldsymbol{p}$ & $\boldsymbol{x}$ Well-Defined

- Hamming distance (substitution only)
- Edit distance (plus insertion & deletion)
- Scoring distance (distinct scores for each pair of letters)

Usually a threshhold k is given; if the pattern $\boldsymbol{p}$ is no more than distance k from a substring $\boldsymbol{u} = \boldsymbol{x}[i..i+m-1]$, then $\boldsymbol{u}$ is a k-match for $\boldsymbol{p}$.

Hamming distance

$$
\begin{array}{lccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\boldsymbol{x} = & m & i & \underline{s} & s & i & \underline{s} & s & i & p & \underline{p} & i \\
\boldsymbol{p} = & & i & \underline{p} & s & i & & & & & & \\
 & & & & & i & \underline{p} & s & i & & & \\
 & & & & & & & & i & p & \underline{s} & i
\end{array}
$$

Hamming distance $d(\boldsymbol{p}, \boldsymbol{x}[2..5]) = 1$ implies that one substitution yields a match.

Edit/Scoring distance I

Using deletions and insertions can sometimes reduce the distance:

$$\boldsymbol{x} = abcd; \quad \boldsymbol{p} = adbc.$$

Here the Hamming distance $d(\boldsymbol{p}, \boldsymbol{x}) = 3$, but deleting d from $\boldsymbol{x}$, then reinserting it after position 1 (two operations), yields edit distance $d'(\boldsymbol{p}, \boldsymbol{x}) = 2$.

More generally, scoring distance (often used in DNA analysis) gives different weights (or scores) to each operation — for exxample, reflecting the probability that letter A is deleted, or that $C \rightarrow G$.

Pattern-matching using all of these forms of distance is implemented using dynamic programming.

Edit/Scoring distance II

	1	2	3	4	5	6	7	8	9	10	11
$\boldsymbol{x} =$	m	i	$\underline{s}$	s	i	$\underline{s}$	s	i	p	$\underline{p}$	i
$\boldsymbol{p} =$		i		s	i						
					i		s	i			

Edit distance $d(\boldsymbol{p}, \boldsymbol{x}[2..5]) = 1$ since deleting $\boldsymbol{x}[3]$ yields a match. (Also $d(\boldsymbol{p}, \boldsymbol{x}[2..4] = 1$ by substituting $\boldsymbol{x}[4] \leftarrow i$, $d(\boldsymbol{p}, \boldsymbol{x}[3..5]) = 1$ by substituting $\boldsymbol{x}[3] \leftarrow i$.)

Indeterminate Pattern-Matching

This is common in applications to DNA/protein sequences, where a letter may legitimately take one of several values, and so match with each of them. For example, the string

$$h\{a, i, o, u\}t$$

matches

$$hat,\ hit, h\{o, u\}t.$$

A new area of research since 2003; the subject of Shu Wang's Ph.D. dissertation. Bit-parallel and hybrid approaches are effective.

(δ, γ)-Matching

In musical texts, the notes can be represented by integers. A match occurs if

$$\begin{aligned} \max_{j=1}^{m} |\boldsymbol{p}[j] - \boldsymbol{x}[i+j-1]| &\leq \delta, \\ \Sigma_{j=1}^{m} |\boldsymbol{p}[j] - \boldsymbol{x}[i+j-1]| &\leq \gamma. \end{aligned}$$

Shift-Or is used, and modifications of other exact methods.

Abelian Pattern-Matching

This is an even newer area of research, again motivated by applications in bioinformatics. In

$$\boldsymbol{x} = mississippi,$$

under Abelian matching, we find five matches with $\boldsymbol{p} = sis$!