

Approximate String Matching

Ricardo Baeza-Yates

Center for Web Research

www.cwr.cl

Depto. de Ciencias de la Computación

Universidad de Chile

Santiago, CHILE

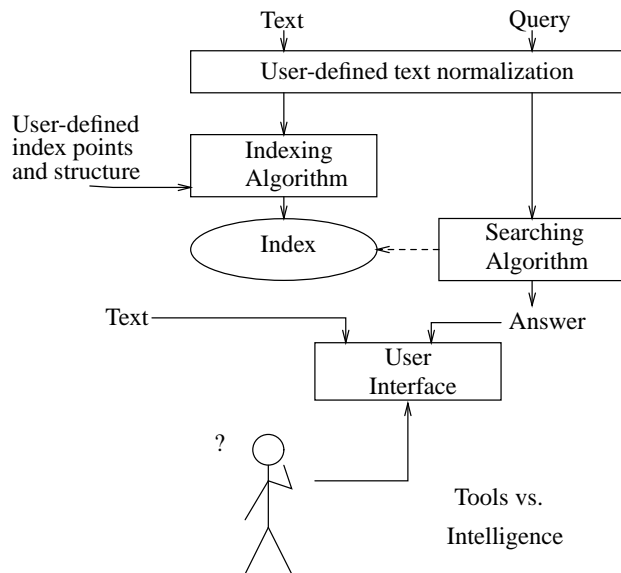
rbaeza@dcc.uchile.cl

Outline

- Text Retrieval
- Theory vs. Practice
- Problem
- String searching
- From automata to algorithms
- Filtering
- Indices
- ASM with Indices
- Concluding remarks

Based in surveys by Baeza-Yates [3], Baeza-Yates [5], Navarro [19] and Navarro *et al.* [23], and own work and views.

User's point of view



Applications to other areas:

Web retrieval, XML processing, NL processing, text mining, multimedia search, bioinformatics, signal processing,

Theory vs. Practice

How we can measure the goodness of an algorithm?

- Asymptotic worst case behavior
- Asymptotic average case behavior
- Practical behavior

D. Knuth [IFIP'89 Invited speech]

- Balance between theory and practice
- Software is hard

The best theory

is

inspired by practice

—

The best practice

is

inspired by theory

Problem

- Σ : finite alphabet of size σ
- Text $t \in \Sigma^*$ of length n
- Pattern $p \in \Sigma^*$ of length m ($m \ll n$)
 m is considered bounded
- Problem: find all occurrences of p in t
- Space complexity S : the extra space used for the search (index)
RAM model: words of size $O(\log n)$
- Time complexity C : time needed to find the pattern
or equivalent measure (for example, comparisons)
 - Worst-case
 - Average-case (uniform text and pattern)

Search Models

1. p is a word (depends on the language)
2. p is any sequence starting in an index-point

Some data structures assume the first model

Answer Models

- exact match
- approximate match (distance function needed)
- closest match or all matches at a certain distance

Computation Models

- Text-Pattern comparisons
- Arithmetical/Bitwise operations

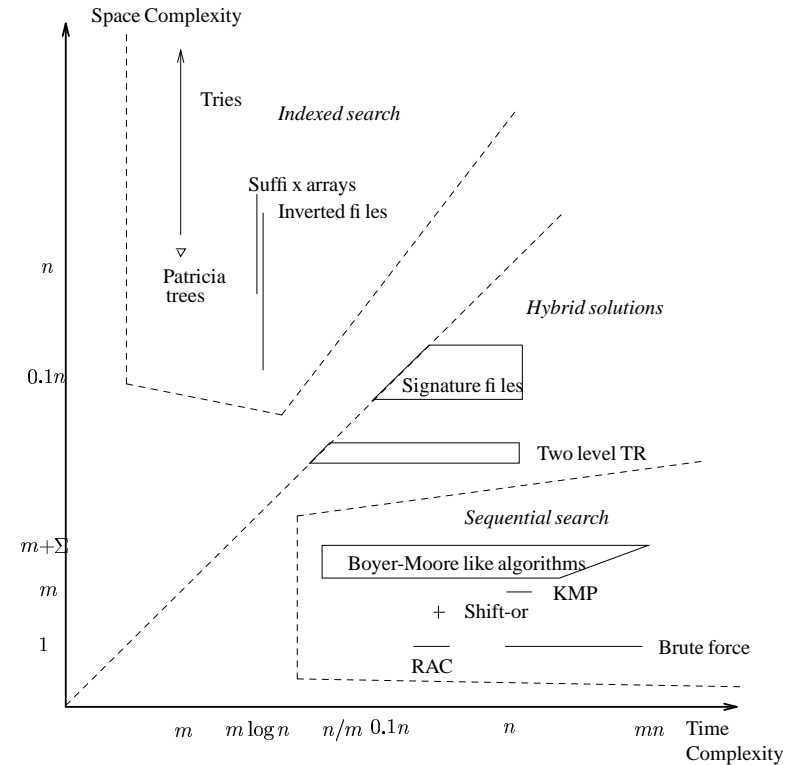
Algorithmic point of view

Input data:

- Raw pattern and text
 - sequential, on-line, real-time algorithms
- Preprocessing of the pattern
 - pattern is known in advance
- Preprocessing of the text → index
 - Inverted index
 - Suffi x trees (tries, Patricia trees,)
 - Suffi x arrays
 - Based on q -grams
 - Automata: DAWGs, suffi x based
- Hybrid solutions:
 - Filtering or Filtration
 - Two Level TR

7

String-Matching Space-Time Trade-Offs



8

String Matching: Definition

- Basic problem: find exact occurrences of a pattern in a text
- Variations
 - Allow k mismatches (Hamming distance)
 - Allow k insertions (Episode distance, not symmetric)
 - Allow k insertions and deletions (LCS distance)
 - Allow k mismatches, insertions and deletions
 - Language dependent measure: phonetic, morphemes, etc.

- Examples:

```
text
text
text: This is a text example ...
text
ex
```

- Software examples: *grep* command in Unix (sequential) or *Google* in the Web (index based).

String Matching Complexity

n : size of the text

m : size of the pattern

- Raw text
 - Worst case:
lower and upper bound of $n + O(n/m)$ comparisons
 - Average case: $O(\log m n/m)$ lower and upper bound
 - ASM: $O(kn)$ worst case, $O((k + \log m)n/m)$ average case
- Preprocessed text:
 - Index construction: $O(n)$ time and space (finite alphabet)
 - Worst case: $O(m)$ comparisons
 - Average case: $\min(m, \log n)$ comparisons
 - ASM: several results, still open

Knuth-Morris-Pratt Algorithm

Fascinating story.... from theory and practice

Preprocessing:

$next[j] = \max\{i | (pattern[k] = pattern[j - i + k] \text{ for } k = 1, \dots, i - 1)$
and $pattern[i] \neq pattern[j]\}$,

for $j = 1, \dots, m$.

Example:

```
      a b r a c a d a b r a
next[j] 0 1 1 0 2 0 2 0 1 1 0 5
```

Worst case complexity: $2n + O(matches)$

Extension to multiple patterns: Aho-Corasick

Algorithm

```
search( text, n, pat, m ) // Search pat[1..m] in text[1..n]
char text[], pat[];
int n, m;
{
    int next[MAX_PATTERN_SIZE];

    pat[m+1] = CHARACTER_NOT_IN_THE_TEXT;
    kmp( pat, m+1, pat, m+1, next ); // Preprocess pattern
    kmp( text, n, pat, m, next );    // Search text
    pat[m+1] = END_OF_STRING;
}

kmp( text, n, pat, m, next )
char text[], pat[];
int n, m, next[];
{
    static dosearch = 0;
    int i, j;

    i = 1;
    if( !dosearch ) // Preprocessing
        j = next[1] = 0;
    else j = 1;
    do {
        if( j == 0 || text[i] == pat[j] )
        {
            i++; j++;
            if( !dosearch ) { // Preprocessing
                if( text[i] != pat[j] ) next[i] = j;
                else next[i] = next[j];
            }
        }
        else j = next[j];

        if( dosearch && j > m ) { // Search
            Report_match_at_position( i-m );
            j = next[m+1];
        }
    }
    while( i <= n );
    dosearch = 1;
}
```


Example

Pattern = t h a n

Text = t h i s i s a n e x a m p l e t h a t

Count = 2 0 0 0 0 0 2 0 0 0 0 1 0 0 0 0 0 0 0 3 0 0 1

Each step is:

```
For all j such that pattern[j] = text[i]
    increment count[i-j+1]
```

Code

```
for (i=0; i<n; i++) {
    if ((off1=(aptr=&alpha[c=*t++])->offset) >= 0) {
        count[(i+off1)&MOD256]--;
        for (aptr=aptr->next; aptr!=NULL; aptr=aptr->next)
            count[(i+aptr->offset)&MOD256]--;
    }
    if (count[i&MOD256] <= k) printf("%d",count[i&MOD256]);
    count[i&MOD256] = m;
}
```

Running time

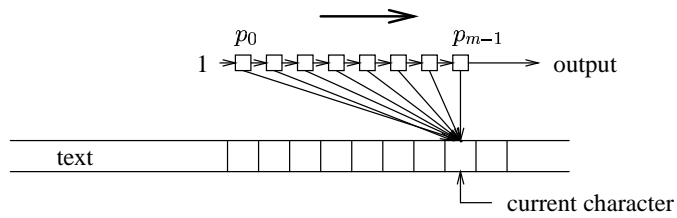
- $O(R + n + m + |\Sigma|)$ total cost
- R is the number of text-pattern symbol matches

$$0 \leq R \leq f_{max}n \leq mn$$

- On average $R = O(\frac{m}{|\Sigma|}n)$
- Cost is independent of number of mismatches
- Not suitable for $m \gg \Sigma$ (e.g. DNA)

Bit Parallelism: Baeza-Yates/Gonnet, 1989 [2]

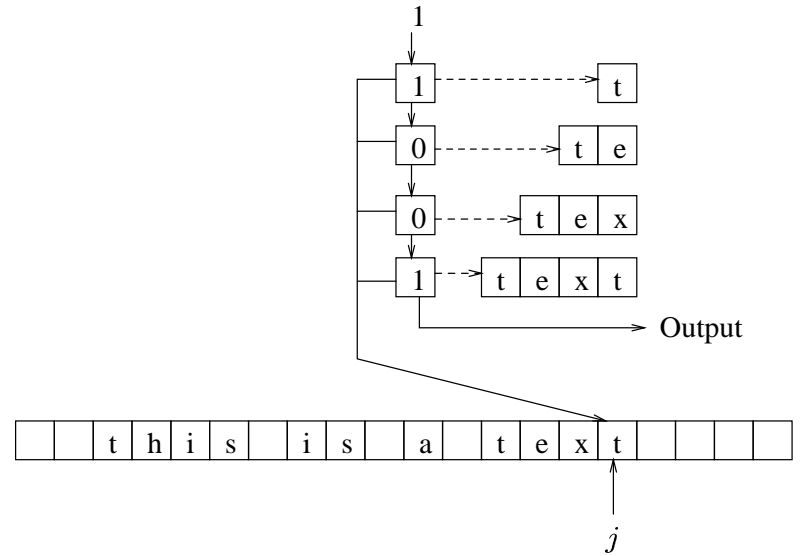
Parallel algorithm using m processors



Processor i : 1 if $p_0, \dots, p_i = t_{j-i}, \dots, t_j$
 0 otherwise

$$p_i^{j+1} \leftarrow p_{i-1}^j \ \& \ (pattern[i] =? text[j])$$

Example:



Bit sequence simulation

One bit per processor \rightarrow simulation with a bit vector!

$$\vec{p} \leftarrow (\text{shift left } \vec{p}) \& \begin{pmatrix} p \\ a \text{ } =? \text{ text}[j] \\ t \end{pmatrix}$$

For finite alphabets, all possible comparisons can be precomputed before the search

$$\vec{p} \leftarrow (\text{shift left } \vec{p}) \& T[\text{text}[j]]$$

In the example:

	T[t]	T[e]	T[x]	T[*]
t	1	0	0	0
e	0	1	0	0
x	0	0	1	0
t	1	0	0	0

$0 \longleftrightarrow 1 \rightarrow$ shift-and/or algorithm

Handbook of Algorithms and Data Structures, 2nd Ed, 1991

Complexity

For the uniform cost RAM model, we have

$$\text{word size} = O(\log_2 n)$$

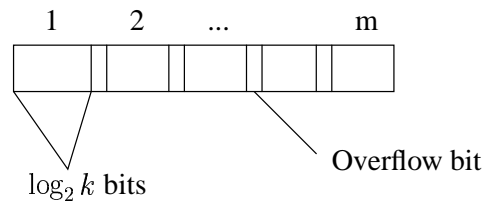
- Preprocessing time: $O(\Sigma + m)$
- Search time: $O(mn / \log n)$
- Space needed: $O(\Sigma m / \log n)$ words

Code:

```
// Preprocessing
for( i=0; i<MAXSYM; i++ ) T[i] = ~0;
for( lim=0, j=1; *pattern != EOS; lim |= j, j <<= B, pattern++ )
    T[*pattern] &= ~j;
lim = ~(lim >> B);
// Search
matches = 0; state = ~0; // Initial state
for( ; *text != EOS; text++ )
{
    state = (state << B) | T[*text]; // Next state
    if( state < lim )
        // Match at current position-len(pattern)+1
}
```

Extensions

- Every pattern element is a class of symbols
Just change T !
- Don't care symbols on the text: $T[*] = 11 \dots 1$
- Multiple patterns: just one longer sequence
- Mismatches: count the number of mismatches
+ instead of &, using $\log k + 1$ bits per position
 k is maximal number of mismatches allowed



- Insertions and deletions [Wu & Manber, 1991]

$k + 1$ bit sequences

Agrep: Was the fastest approximate search tool for Unix, now
nrgrep

Approximate String Matching: Dynamic Programming

- Minimum number of errors to match $P_{1..i}$ to a suffix of $T_{1..j}$

$$C[0, j] = 0$$

$$C[i, 0] = i$$

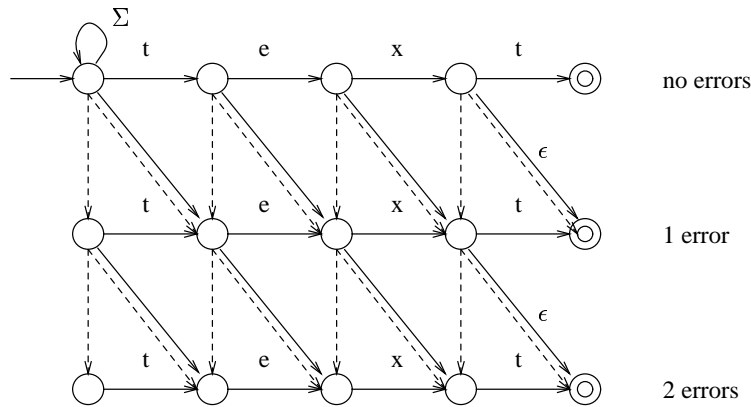
$$C[i, j] = \begin{cases} C[i-1, j-1] & \text{if } (P_i = T_j) \\ 1 + \min(C[i-1, j], C[i, j-1], C[i-1, j-1]) & \text{else} \end{cases}$$

- Example:

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

- Bit-wise approach to DP is the fastest for long strings (≥ 8)

Horizontal bit parallelism: Wu & Manber



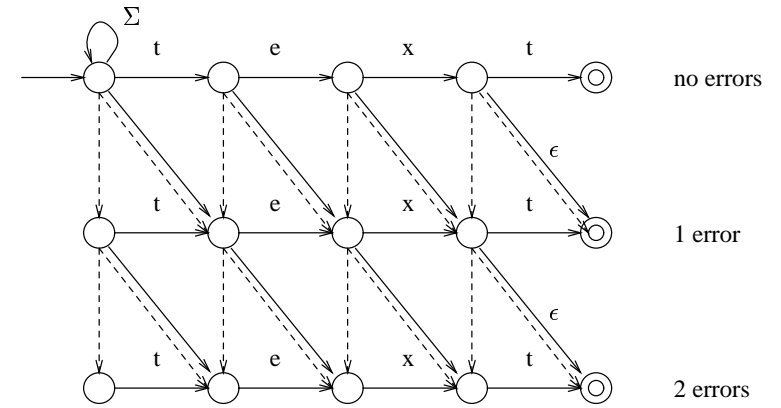
$$\vec{R}'_k = (\text{shift } \vec{R}_k \& T[\text{text}[j]]) \text{ or } \vec{R}_{k-1} \text{ or } (\text{shift}(\vec{R}_{k-1} \text{ or } \vec{R}'_{k-1}))$$

Initially $\vec{R}_k = 11..100..0$ (k ones)

Drawback: Dependency on R'_{k-1}

Complexity: $O(kmn / \log n)$ search time
 $O((k + \Sigma)m / \log n)$ space

Vertical bit parallelism:



Key information: highest (smallest error) active state per column

State of the search: m numbers on the range $0..k + 1$.

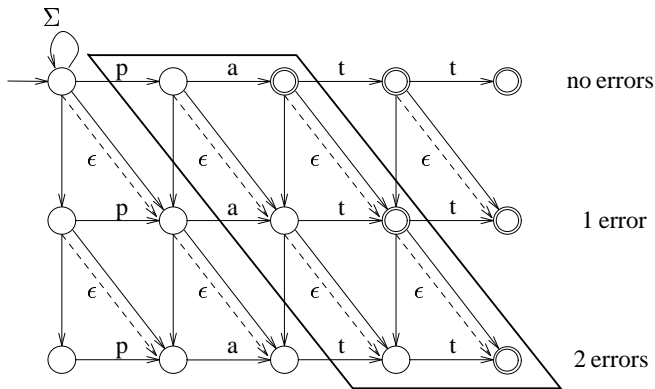
$$R'_i = \min(R_{i-1} + (\text{text}[j] == \text{patt}[i - 1]), R_i + 1, R'_{i-1} + 1)$$

Initially $R_0 = 0$ and $R'_0 = R_0$

Related to dynamic programming:

- Longest common subsequence, string editing
- Related to Ukkonen's automata approach

Diagonal bit parallelism: Baeza-Yates & Navarro [1996]



Each diagonal represents an ϵ -closure (longest match): D_i

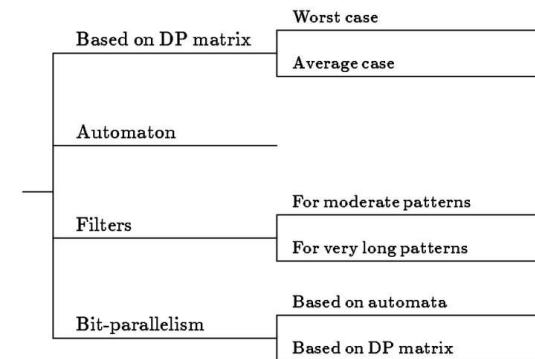
$$D'_i = \min(D_i + 1, D_{i+1} + 1, g(D_{i-1}, c))$$

where

$$g(D_i, c) = \min(\{k + 1\} \cup \{ j / j \geq D_i \wedge pat[i + j] == c \})$$

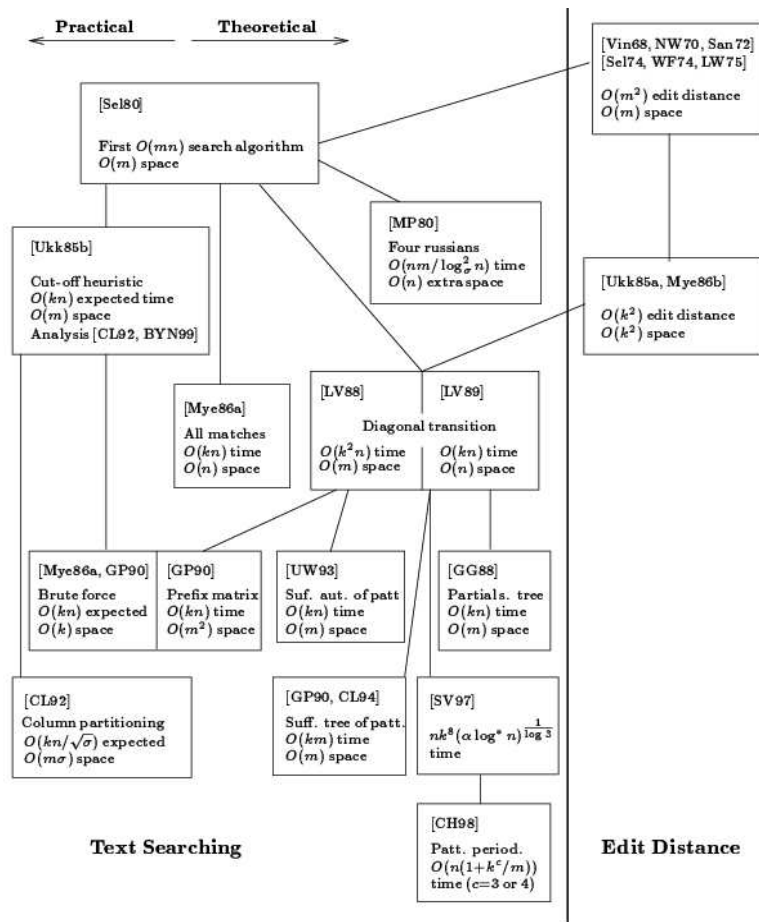
- Advantage: all R'_i 's can be computed in parallel
- Mixing this with filtration we get $O(n\sqrt{mk/\log n})$ time
- Related to simulation of DP over a suffix array (later) and bioinformatic applications

ASM: Sequential Algorithms

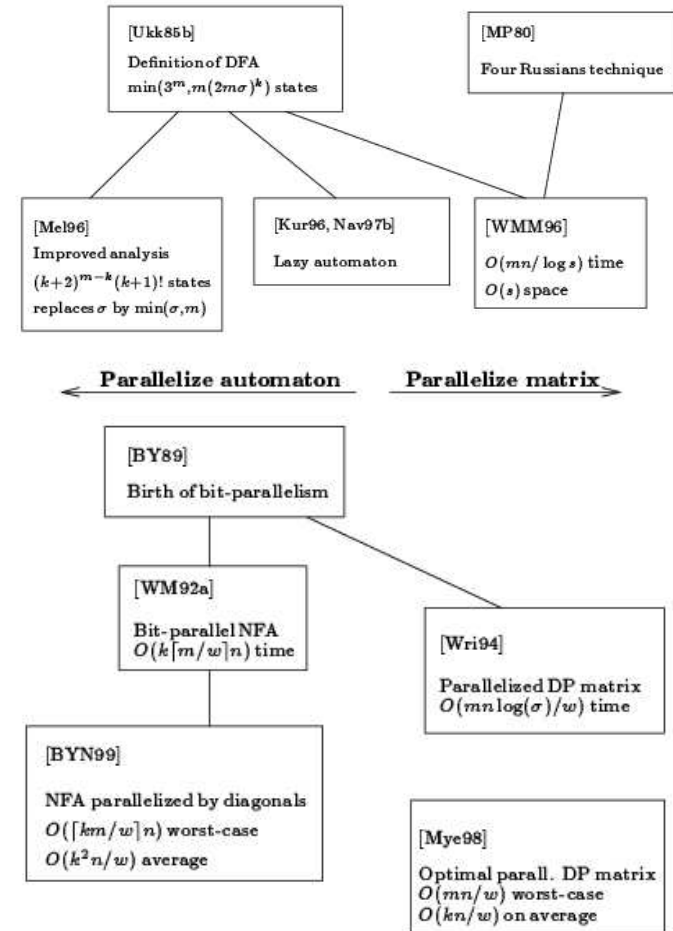


first algorithm	80	[Sel80] [MP80]		
best worst cases	85	[LV88] [Ukk85b]	[Ukk85b]	
	86	[LV89] [Mye86a]		
	87			
	88	[GG88]		
	89	[GP90]		
first filter	90	[CL94] [UW93]		[TU93] [CL94]
	91			[JTU96]
first bit-parallel	92	[CL92]	[WMM96]	[WMM92a] [WMM92a] [BYP96] [Ukk92]
	93			
avg. lower bound	94		[Wri94]	[CM94] [Tak94]
	95		[Mel96]	[ST95]
	96		[Kur96]	[BYN99] [BYN99] [Shi96] [GKHO97]
	97	[SV97]		[Nav97a]
fastest practical	98	[CH98]	[Mye98]	[NBY98a] [NBY98c] [NR98b]
			Dyn.Prog.	Automata Bit Par. Filters

Dynamic Programming



Automata and Bit-Parallelism



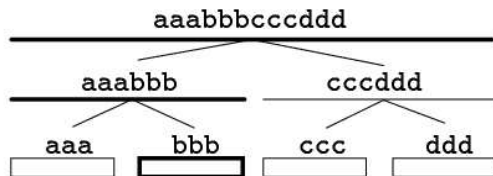
Filtering or Filtration

Find potential matches and then apply a sequential algorithm to check each candidate

- Search time is

$$Filtration(n) + Candidates(n) O(mk)$$

- Filtration can be done by a sequential scan or by an index
- There is trade-off between $Filtration(n)$ and $Candidates(n)$
- There is always a maximum error ratio $\alpha = k/m$ up to where Filtration is useful, as for larger error levels the text areas to verify cover almost all the text
- Verification can be done in a hierarchical fashion



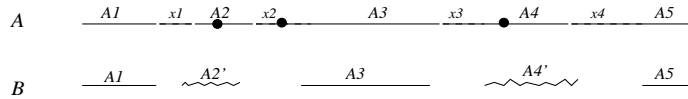
A First Lemma for Filtering

Lemma 1: Let A and B be two strings such that $d(A, B) \leq k$. Let $A = A_1x_1A_2x_2\dots x_{k+s-1}A_{k+s}$, for strings A_i and x_i and for any $s \geq 1$. Then, at least s strings $A_{i_1} \dots A_{i_s}$ appear in B . Moreover, their relative distances inside B cannot differ from those in A by more than k .

Consider the sequence of at most k edit operations that convert A into B .

- Each edit operation can affect at most one of the A_i 's, at least s of them must remain unaltered.
- Relative distances: the k edit operations cannot produce misalignments larger than k .

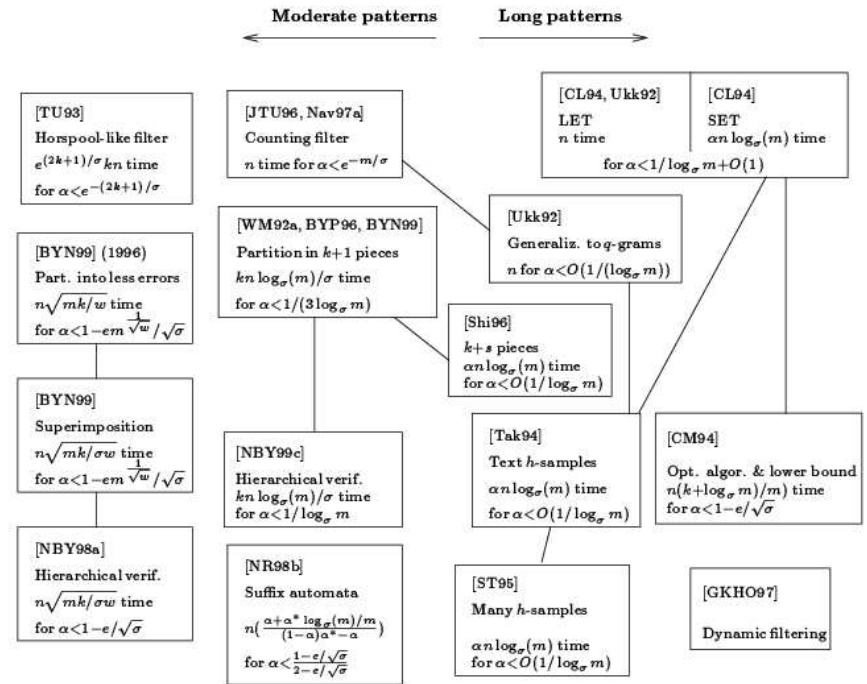
Example



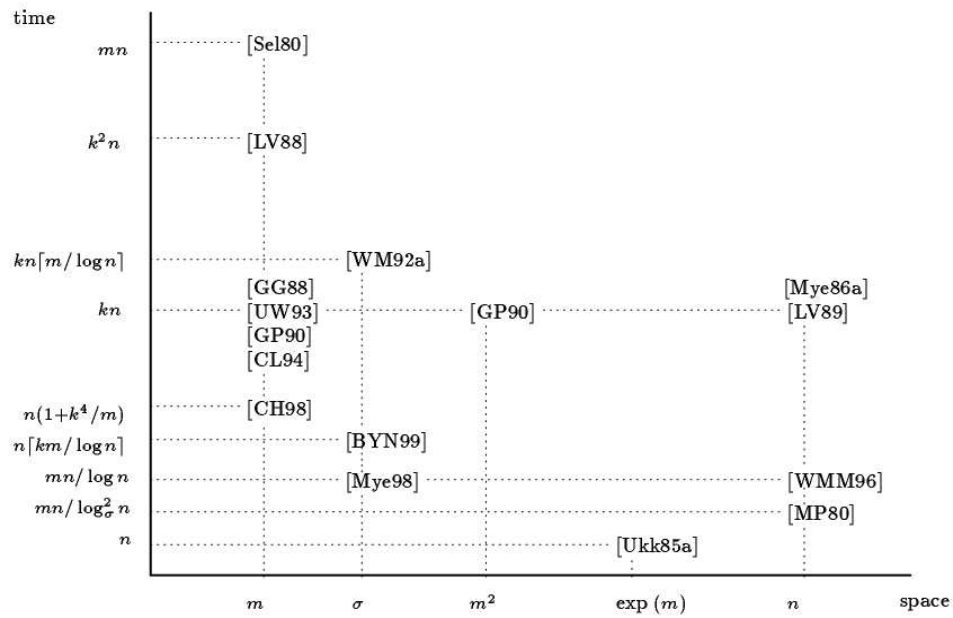
An example of Lemma 1 with $k = 3$ and $s = 2$:

- At least 2 of the A_i 's survive unaltered.
- They are actually 3 such segments because one of the errors appeared in x_2 .
- Another possible reason could have been more than one error occurring in a single A_i .

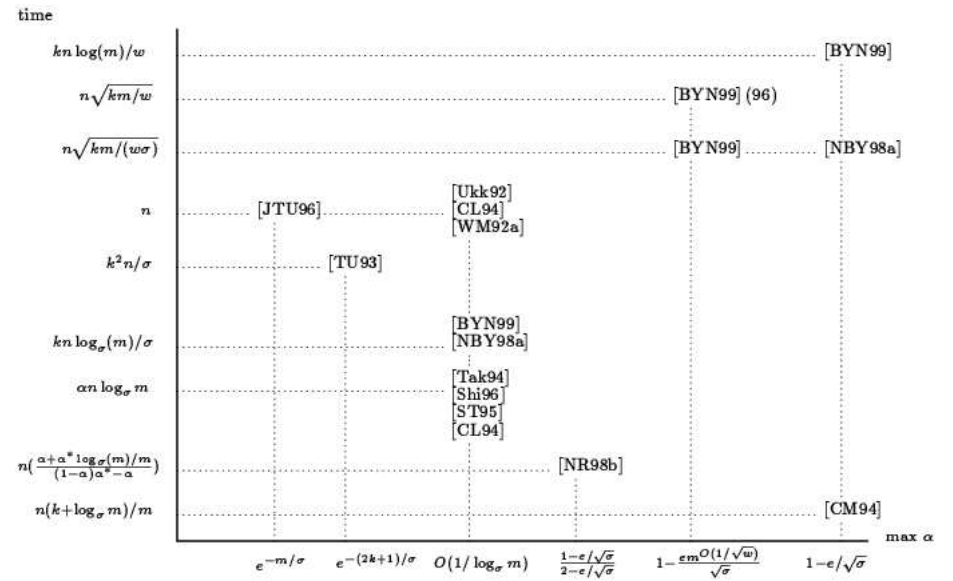
Filtering Algorithms



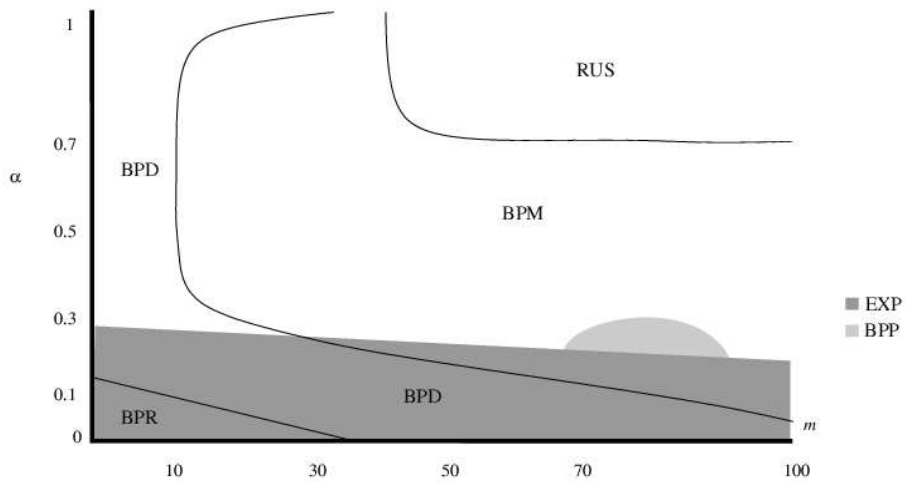
Worst Case Complexity and Space



Average Case Complexity and Error Ratio



Best Algorithms



Data Structures

- *Inverted indices* permit searching for any word in the text.
- *Suffix trees* allow searching for any substring of the text.
- *Suffix arrays* permit the same operations but are slightly slower.
- *Q-grams* allow searching for any text substring not longer than q .
- *Q-samples* permit the same but only for some text substrings.

Inverted Indices

Idea: all words and their positions

1 6 9 11 17 19 24 28 33 40 46 50 55 60
 This is a text. A text has many words. Words are made from letters.

Vocabulary	Occurrences	Text
letters	60...	Inverted Index
made	50...	
many	28...	
text	11, 19...	
words	33, 40...	

Vocabulary search: Hashing, sorted, etc.

Granularity of the occurrences depends in what we want to answer: file, word, byte

Inverted Files: Space

- Vocabulary: Heaps' law

$$V = C \times n^\beta$$

- Posting file: linear space (one occurrence = one pointer)

- Word distribution: Zipf's law

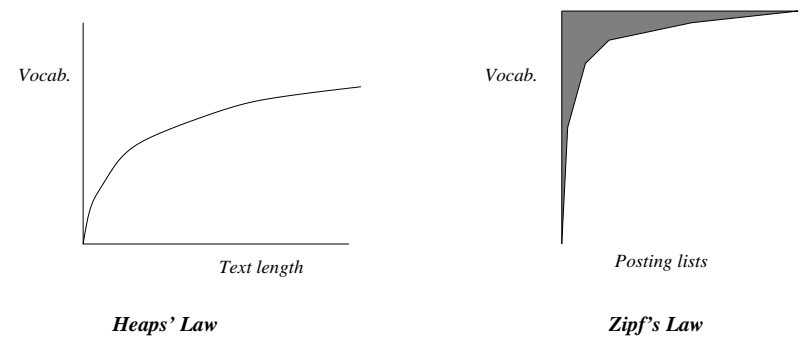
$$w_r = \frac{N}{r^\theta H_N(\theta)}$$

where

$$H_N(\theta) = \sum_{r=1}^k \frac{1}{r^\theta}$$

- Stopwords: half the posting file

- Linear space

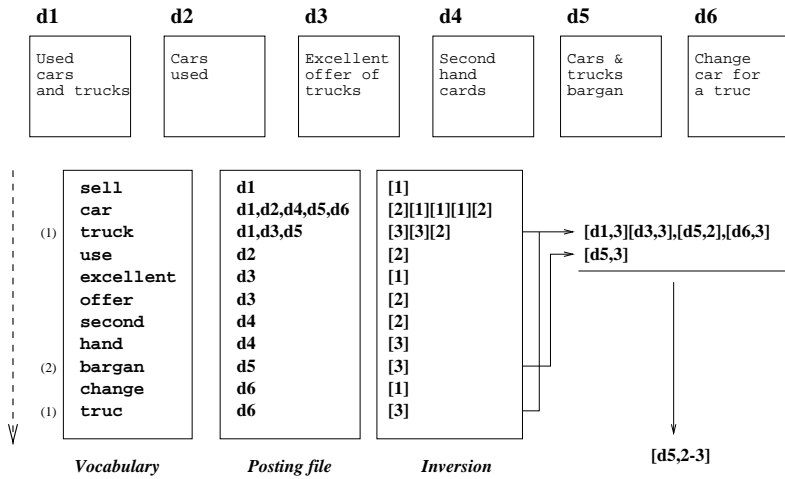


Heaps' Law

Zipf's Law

Complex Patterns

Search the vocabulary sequentially (e.g. ASM) and do set operations

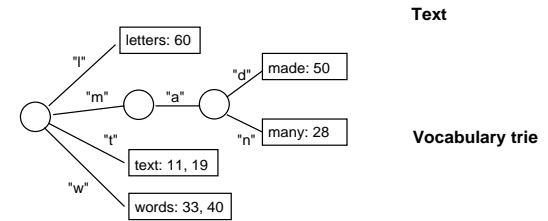


"bargain for trucks" [approximate]

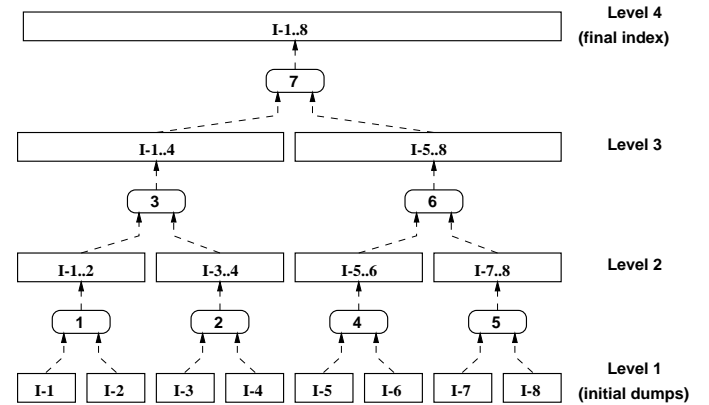
Building Inverted Indices

- Process text pieces as large as possible

1 6 9 11 17 19 24 28 33 40 46 50 55 60
 This is a text. A text has many words. Words are made from letters.



- Merge partial indexes



Two-level Text Retrieval: Block addressed inverted files

Idea used in PIRS (Personal Information Retrieval System)

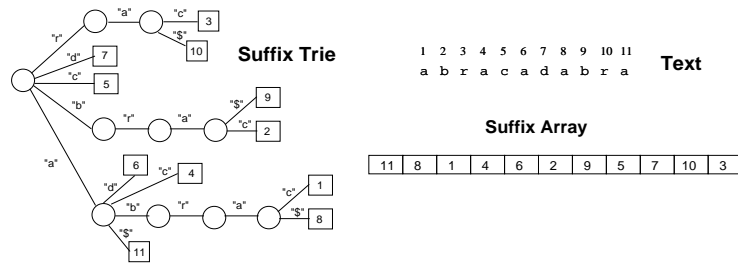
[Wu and Manber 1993]

- The text is divided in 256 blocks of the same size
- An inverted file of all the different words of the text is built
- Each entry indicates only the blocks where the word appears
→ 1 byte per block
- First we search in the inverted file
next in the corresponding blocks using a fast sequential algorithm
Complexity depends on the number of occurrences
→ locality of reference is important
- For large texts, empirical results show that the index requires
less than 5% of the text size
- This idea works reasonable well up to 200Mbs

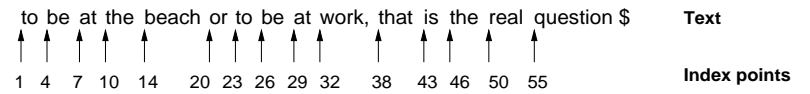
Inverted File Space in Practice

Index	Small base (1 MB)		Medium base (200 MB)		Large base (2 GB)	
Full inverted	45%	73%	36%	64%	35%	63%
Document addressing	19%	26%	18%	32%	26%	47%
Block (64K) addressing	27%	41%	18%	32%	5%	9%
Block (256) addressing	18%	25%	1.7%	2.4%	0.5%	0.7%

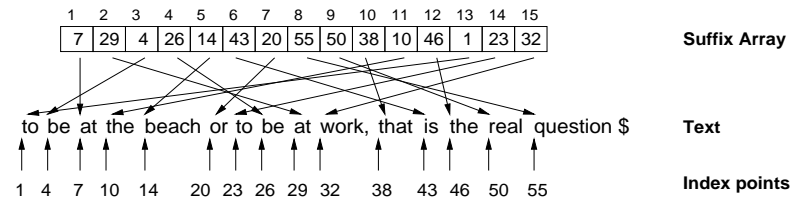
Tries and Suffix Trees



Suffix Arrays



- 1: to be at the beach or to be at work, that is the real question
- 4: be at the beach or to be at work, that is the real question
- 7: at the beach or to be at work, that is the real question
- 10: the beach or to be at work, that is the real question
-
- 55: question



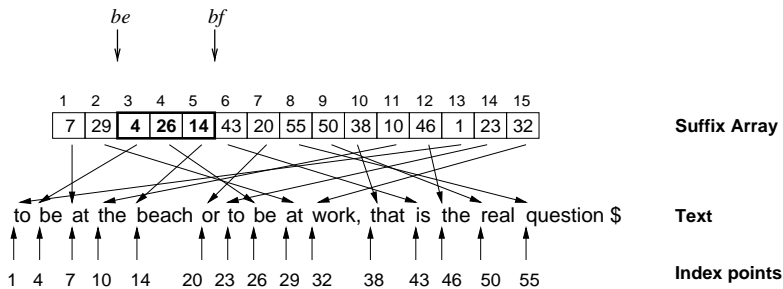
- Search time is optimal: $O(m)$
- Problem: space can be quadratic in a trie
- Compact suffix trees (Patricia trees): cut unary paths
- To remember the depth, a count is added at every node or the string associated to the path is stored
- Space is now linear ($\approx 9n$)

Useful for complex queries: regular expressions in sublinear average time [4]

Suffix Array Search

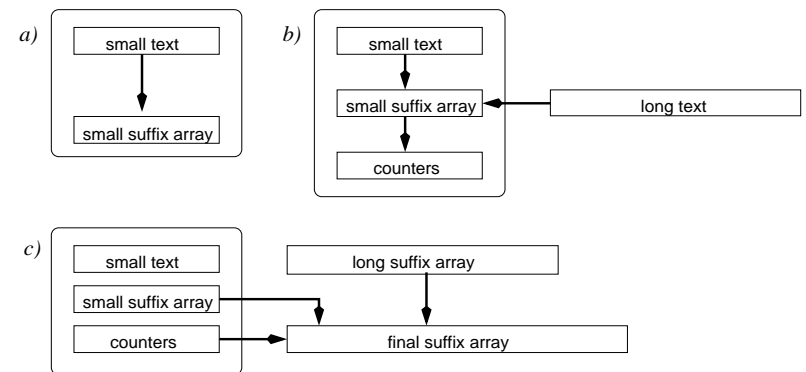
- Every substring is a prefix of a suffix
- The prefix relation can be used for lexicographic order

$$x : a \text{ prefix of } y \iff (x : a \leq y) \wedge (y < x : (a + 1))$$
- Hence, two binary searches are enough to obtain the suffix array range where all occurrences of x appear
- Number of occurrences: range size
- Time is logarithmic in the size of the array



Suffix Array: Construction

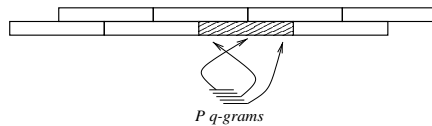
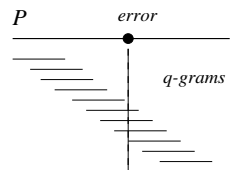
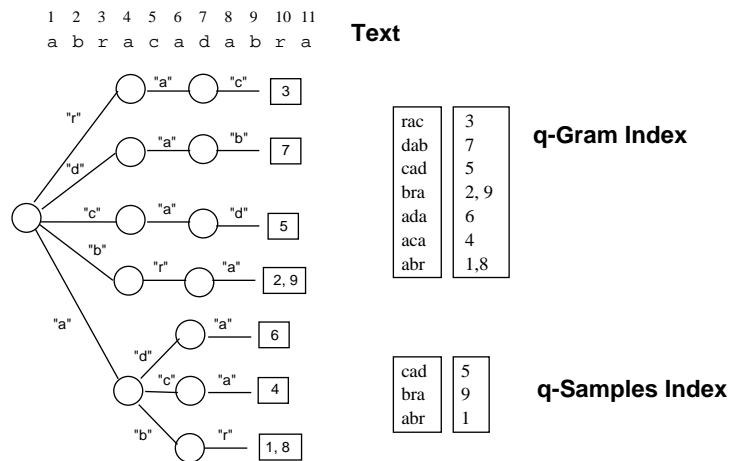
- In principle is a lexicographical order
- But suffixes are suffixes of suffixes
- However, random access to the text is the bottleneck
- Best solution: sequential scan with counting



Building time is now linear (2003!)

Q-gram Indexes

In a q -gram index, every different text q -gram is stored, and all its positions are stored in increasing text order.



Q-Sample Indexes

- In a q -sample index, only *some* text q -grams are stored. In this case the samples do not overlap.
- Useful to search long strings and using much less space (from $.5$ to $3n$)
- Search for a string can be constant on average
- Building them takes linear time

Algorithms for ASM using an Index

Search approaches:

- *Neighborhood generation* generates and searches for, using an index, all the strings that are at distance k or less from the pattern
- *Partitioning into exact searching* selects pattern substrings that must appear unaltered in any approximate occurrence, uses the index to search for those substrings, and checks the text areas surrounding them.
Assuming that the errors occur in the pattern or in the text leads to radically different approaches.
- *Intermediate partitioning* extracts substrings from the pattern that are searched for allowing fewer errors using neighborhood generation.

Current Results on this Taxonomy

Data Structure	Search Approach				
	Neighborhood Generation	Partitioning into Exact Searching		Intermediate Partitioning	
		Errors in Text	Errors in Pattern	Errors in Text	Errors in Pattern
Suffi x Tree	[13] Jokinen & Ukkonen 91 [29] Ukkonen 93 [8] Cobbs 95		[24] Shi 96		
Suffi x Array	[10] Gonnet 88				[21] Navarro & Baeza-Yates 99
Q-grams	n/a	[13] Jokinen & Ukkonen 91 [12] Holsti & Sutinen 94	[20] Navarro & Baeza-Yates 97		Myers 90 [17]
Q-samples	n/a	[26] Sutinen & Tarhio 96	n/a	[22] Navarro et al. 2000	n/a

Neighborhood Generation

The Neighborhood of the Pattern:

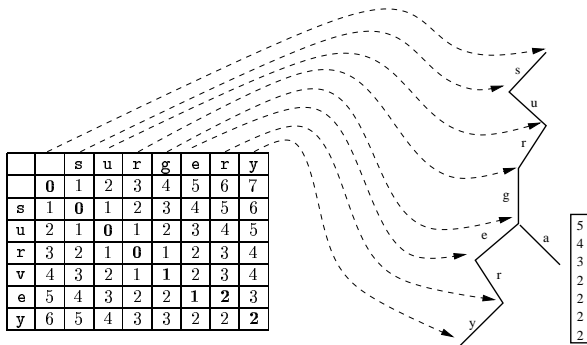
- Let $U_k(P) = \{x \in \Sigma^*, d(x, P) \leq k\}$ be the “ k -neighborhood” of P (is finite set).
- Generate $U_k(P)$ and use an index to search for their text occurrences [17]
- Problem: $U_k(P)$ is quite large.
- Good bounds [28, 17] show $|U_k(P)| = O(m^k \sigma^k)$ [28]
- This approach works well for small m and k .

Backtracking

- Use a suffix tree or array to find $U_k(P)$ in the text [4, 10, 29]
- Just some branches will be followed, but they factorize many matches
- While searching we have three cases at node N :
 - a) $C_{m,N} \leq k$, which means that $y \in U_k(P)$, and we report all the leaves of the current subtree as answers.
 - b) $C_{j,N} > k$ for every j , which means that y is not a prefix of any string in $U_k(P)$ and we can abandon this branch
 - c) Otherwise, we continue descending by every branch of that node. If we arrive at a leaf node, we have to use a sequential algorithm in the rest
- Some improvements [13, 30, 8] avoid processing some redundant nodes at the cost of a more complex node processing
- The same idea can be used to compare a whole text against another one or against itself [6]

Example

The matrix can be seen now as a stack that grows to the right



In the example:

- With $k = 2$ the backtracking ends indeed after reading "surge"
- With $k = 1$ the search would have been pruned after considering "surger", and "surga", since in both cases no entry of the matrix is ≤ 1

Partitioning into Exact Search: Errors in the Pattern

- We use Lemma 1 under the setting $P = A$, $x_i = \varepsilon$. That is, the pattern is split in $k + s$ pieces, and hence s of the pieces must appear inside any occurrence.
- Then, the $k + s$ pieces are searched and the text areas where s of those pieces appear under the stated distance requirements are verified for a complete match.
- Search time in the index is $O(m)$ or $O(m \log n)$, but the checking time dominates.
- The case $s = 1$, proposed in [20], shows an average time to check the candidates of $O(m^2 kn / \sigma^{m/(k+1)})$.
- The case $s > 1$ is proposed in [24] without any analysis.
- If s grows, the pieces get shorter and hence there are more matches to check, but on the other hand, forcing s pieces to match makes the filter stricter [24]. Recent results show that this is slower.
- Note that, since we cannot know where the pattern pieces can be found in the text, all the text positions must be searchable.

Partitioning into Exact Search: Errors in the Text

- Assume now that the errors occur in the text, i.e., A is an occurrence of P in T .
- We extract substrings of length q at fixed text intervals of length $h \geq q$.
- Those q -samples correspond to the A_i 's of Lemma 1, and the space between q -samples to the x_i 's.
- What the lemma ensures is that, inside any occurrence of P containing $k + s$ text q -samples, at least s of them appear in P at about the same positions ($\pm k$).
- Now we need to ensure that any occurrence of P in T contains at least $k + s$ text q -samples, i.e., $h \leq \lfloor (m - k - q + 1) / (k + s) \rfloor$.

Search Algorithm

- At search time, all the $m - q + 1$ (overlapping) pattern q -grams are extracted and searched for in the index of text q -samples.
- When s pattern q -grams match in the text at the proper distances, the text area is verified.
- This idea is presented in [26], and earlier versions in [13, 12, 27].
- The best value of q :
 - Should be small to avoid a very large set of different q -samples.
 - Should be large to minimize the amount of verification.
- Some analysis [25] show that $q = \Theta(\log_{\sigma} n)$ is the optimal value.
- The best s value? A larger s may trigger fewer verifications.

Intermediate Partitioning

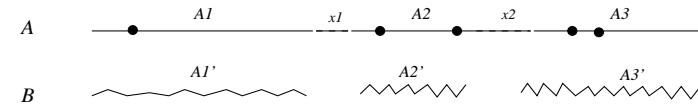
We filter the search by looking for pattern pieces, but those pieces are large and still may appear with errors in the occurrences.

However, they appear with *fewer* errors, and then we use neighborhood generation to search for them.

Lemma 2: Let A and B be two strings such that $d(A, B) \leq k$. Let $A = A_1x_1A_2x_2\dots x_{j-1}A_j$, for strings A_i and x_i and for any $j \geq 1$. Let k_i be any set of nonnegative numbers such that $\sum_{i=1}^j k_i \geq k - j + 1$. Then, at least one string A_i appears with at most k_i errors in B .

Proof and Example

The proof is easy: if every A_i needs more than k_i errors to match in B , then the total distance cannot be less than $(k - j + 1) + j = k + 1$. Note that in particular we can choose $k_i = \lfloor k/j \rfloor$ for every i .



Let $k = 5$ and $j = 3$. At least one of the A_i 's has at most one error (in this case A_1)

Intermediate Partitioning: Errors in the Pattern

Search approaches based on this method have been proposed in [17, 21]. The algorithm is:

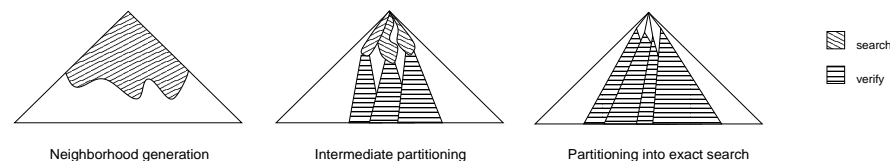
- Split the pattern in j pieces, for some j .
- Use neighborhood generation to find the text positions where those pieces appear, allowing $\lfloor k/j \rfloor$ errors.
- For each such text position, check with an on-line algorithm the surrounding text.

What value for j ?

- In [17], the pattern is partitioned because they use a q -gram index, so they use the minimum j that gives short enough pieces (they are of length m/j).
- In [21] the index can search for pieces of any length, and the partitioning is done in order to optimize the search time.
- Consider the evolution of the search time as j moves from 1 (neighborhood generation) to $k+1$ (partitioning into exact search).
 - We search for j pieces of length m/j with k/j errors, so the error level α stays about the same for the subpatterns.
 - As j moves to 1, the cost to search for the neighborhood of the pieces grows exponentially with their length.
 - As j moves to $k+1$ this cost decreases, reaching even $O(m)$ when $j = k+1$. So, to find the pieces, a larger j is better.

- Cost to verify the occurrences: consider a pattern that is split in j pieces, for increasing j . Start with $j = 2$.
 - Lemma 2 states that every occurrence of the pattern involves an occurrence of at least one of its two halves with $k/2$ errors, although there may be occurrences of the halves that yield no occurrences of the pattern.
 - Consider now halving the halves ($j = 4$), so we have four pieces now (call them “quarters”). Each occurrence of one of the halves involves an occurrence of at least one quarter with $k/4$ errors, but there may be many quarter occurrences that yield no occurrences of a pattern half.
 - Hence, the verification cost grows from zero at $j = 1$ to its maximum at $j = k + 1$.

Trade-off



- In [21] we show that the optimal j is $\Theta(m/\log_\sigma n)$, yielding a time complexity of $O(n^\lambda)$, for $0 \leq \lambda \leq 1$.
- This is sublinear ($\lambda < 1$) for $\alpha < 1 - e/\sqrt{\sigma}$, a pessimistic and is replaced by 1 in practice).
- The same results are obtained in [17] by setting $q = \Theta(\log_\sigma n)$.
- The experiments in [21] show that this intermediate approach is by far superior to both extremes.

Intermediate Partitioning: Errors in the Text

- Consider an occurrence containing a sequence of j q -samples, which must be chosen at steps of $h \leq \lfloor (m - k - q + 1)/j \rfloor$.
- By Lemma 2, one of the q -samples must appear in the pattern with $\lfloor k/j \rfloor$ errors at most.
- Moreover, if every q -sample i appears in the pattern block $Q_i = P_{hi-k, hi+q-1+k}$ with k_i errors, then it must hold that $\sum k_i \leq k$.
- This method [26, 22] searches every block Q_i in the index of q -samples using backtracking, so as to find the least number of errors to match each text q -sample *inside* Q_i .
- If a zone of consecutive samples is found whose errors add up to at most k , the area is verified.
- To allow efficient neighborhood searching, we need to limit the maximum error level allowed.
- Permitting q errors may be too expensive, as every text q -sample will be considered.

- We chose $q > e \geq \lfloor k/j \rfloor$ and assume that every text q -sample indeed matches with $e + 1$ errors.
- We search the pattern blocks permitting only e errors. Every q -sample found with $k_i \leq e$ errors changes its estimation from $e + 1$ to k_i , otherwise it stays at the optimistic bound $e + 1$.
- There is a trade-off here:
 - For a small e value, the search of the e -neighborhoods is cheaper, but as we must assume that the text q -samples not found have $e + 1$ errors, some useless verifications are done.
 - Using larger e values gives more exact estimates of the actual number of errors of each text q -sample, reducing useless verifications in exchange for a higher cost to search the e -environments.
- Optimal e ? In [22] it is mentioned that, as the cost of the search grows exponentially with e , the minimal $e = \lfloor k/j \rfloor$ can be a good choice. Experimentally this scheme tolerates higher error levels than the corresponding partitioning into exact search.

Future

- Further study on the power of non-comparison based algorithms:
Many new bit-based algorithms
- Problem reduction works for text searching
Example: Multiple string searching plus checking
 - Two dimensional case [Baeza-Yates and Regnier, 1990]
 - Approximate pattern matching [Wu and Manber, 1991]
- The final optimal algorithm depends on the input
Further study of input adaptive algorithms?
- New uses for old concepts. Example: q -grams
- Indexing for ASM on NL text can be done better
- Approximation algorithms with worst-case performance guarantees [16].
- Use a metric space to search [7].
- New text indexes tailored to special cases: ASM

References

- [1] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, 1985.
- [2] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35:74–82, Oct 1992.
- [3] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, 1992.
- [4] R. Baeza-Yates and G.H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, Nov 1996.
- [5] R. Baeza-Yates. A unified view of string matching algorithms. In Keith Jeffery, Jaroslav Král, and Miroslav Bartosek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, pages 1–15, Milovy, Czech Republic, November 1996. Springer Verlag.
- [6] R. Baeza-Yates and G. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proc. 6th Symp. on String Processing and Information Retrieval (SPIRE'99)*. IEEE CS Press, 1999. Previous version unpublished, Dept. of Computer Science, Univ. of Chile, 1990.
- [7] E. Chávez and G. Navarro. A metric index for approximate string matching. In *Proc. 5th Symp. on Latin American Theoretical Informatics (LATIN)*, 2002. Cancun, Mexico.
- [8] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. 6th Ann. Symp. on Combinatorial Pattern Matching (CPM'95)*, LNCS 807, pages 41–54, 1995.
- [9] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. 3rd Workshop on Algorithm Engineering (WAE'99)*, LNCS 1668, pages 30–42, 1999.
- [10] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zurich, Switzerland, 1992.
- [11] G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.

- [12] N. Holsti and E. Sutinen. Approximate string matching using q -gram places. In *Proc. 7th Finnish Symp. on Computer Science*, pages 23–32. Univ. of Joensuu, 1994.
- [13] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science (MFCS'91)*, pages 240–248, 1991.
- [14] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, 22(5):935–948, 1993.
- [15] E. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23(2):262–272, 1976.
- [16] S. Muthukrishnan and C. Sahinalp. Approximate nearest neighbors and sequence comparisons with block operations. In *Proc. ACM Symp. on the Theory of Computing*, pages 416–424, 2000.
- [17] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994. Earlier version in Tech. report TR-90-25, Dept. of CS, Univ. of Arizona, 1990.
- [18] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46 (3), 395–415, 1999.
- [19] G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surv.*, 33(1):31–88, 2001.
- [20] G. Navarro and R. Baeza-Yates. A practical q -gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. <http://www.clei.cl>. Earlier version in *Proc. CLEI'97*.
- [21] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms*, 1(1):205–239, 2000. Hermes Science Publishing. Earlier version in *CPM'99*.
- [22] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q -grams. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, pages 350–363, 2000.
- [23] Gonzalo Navarro, Ricardo Baeza-Yates, Erkki Sutinen, Jorma Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 2000.
- [24] F. Shi. Fast approximate string matching with q -blocks sequences. In *Proc. 3rd South American Workshop on String Processing (WSP'96)*, pages 257–271. Carleton University Press, 1996.
- [25] E. Sutinen and J. Tarhio. On using q -gram locations in approximate string matching. In *Proc. 3rd European Symp. on Algorithms (ESA'95)*, LNCS 979, pages 327–340, 1995.
- [26] E. Sutinen and J. Tarhio. Filtration with q -samples in approximate string matching. In *Proc. 7th Ann. Symp. on Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, pages 50–61, 1996.
- [27] T. Takaoka. Approximate pattern matching with samples. In *Proc. 5th Int'l. Symp. on Algorithms and Computation (ISAAC'94)*, LNCS 834, pages 234–242, 1994.
- [28] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
- [29] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Ann. Symp. on Combinatorial Pattern Matching (CPM'93)*, LNCS 684, pages 228–242, 1993.
- [30] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, 1995.