# 3 Searching for Information

- What is and what does a search engine?
- Basic algorithms of information retrieval:
  - pattern matching in textual documents
  - constructing an index
  - analysing a search phrase (query)
  - measuring the relevance of retrieved documents
- Survey of current search engines
  - see e.g. http://searchenginewatch.com/
    (contains lots of free information on search engines and related topics):
    - A Webmaster's Guide To Search Engines:
      Search Engine Design, Major Search Engines, Ranking Web Pages,...
    - Subscriber-Only Area Content
      How AltaVista Works (and AOL, NetFind, Excite, HotBot, Infoseek, Lycos, WebCrawler, Yahoo,...);
      More About Meta Tags, Spamming, Dynamic Pages,...
    - Search Engines Facts And Fun
    - Search Engine Status Reports
    - Search Engine Resources

## Classification

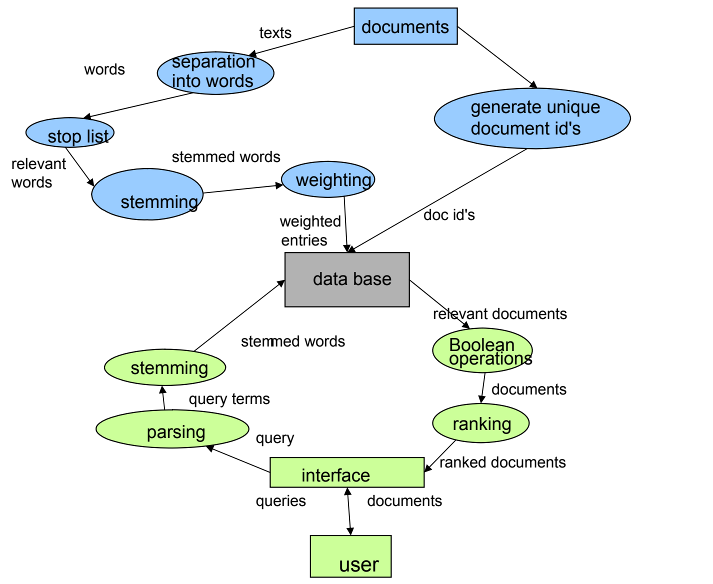
- **Search Engines**:
  search engines constantly visit web sites on the Internet in order to create catalogues of web pages.
  $\Rightarrow$ these catalogs are created automatically and updated constantly without human interference
- **Directories**:
  - manually created catalogues of web pages
  - sites must be submitted, then they are assigned to an appropriate category or categories.
  - directories can often provide better results than search engines (example: Yahoo)
- **Hybrid Search Engines**:
  - search engine + associated directory
  - listing of (search engine detected) sites that have been reviewed or rated.
  - reviewed sites do not appear as the "default" answer to queries
  - reviews are shown only on request.
- **Many current search engines are hybrid search engines (details later).**

## How do search engines work?

Search engines have three major elements:

- the **spider** (also called crawler):
  - visits a web page, reads (and analyses) it, and then follows links to other pages within the site.
  - returns to the site on a regular basis, such as every month or two, to look for changes.
- the **index** (also called catalogue):
  - like a giant book containing a copy of (or information on) every web page that the spider finds.
  - updated, whenever web page changes are detected.
  - only the pages listed in the index are available for searching.
  - uses efficient data structures to support searching
- the **search engine software**:
  - analyses a query,
  - searches the index for documents matching the query,
  - ranks the matching documents in order of relevance.

## General structure of an "Information Retrieval"-System:

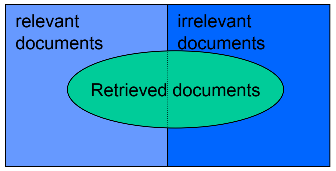


## Quality of information retrieval

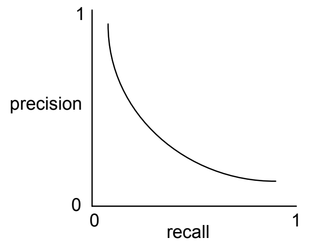
**Standard criteria for assessing the quality of information retrieval (search engines):**

- **recall**: $R(\text{query}) = \dfrac{\#\text{relevant retrieved documents}}{\#\text{relevant documents}}$

- **precision**: $P(\text{query}) = \dfrac{\#\text{relevant retrieved documents}}{\#\text{retrieved documents}}$



Relationship between recall and precision:



⇒ High recall or high precision are easily achieved, but it is hard to achieve both.

## What makes a document relevant for a query?

quick answers

- match or no match
- number of query terms with matches
- number of matches
- location of matches (title, abstract, distance from start,...)
- quality of matches (complete or partial match?)
- direct (explicit) information on relevance

⇒ knowledge of method of evaluation may influence the way documents are presented *( → later)*

⇒ statements on relevance are not unique, depend on method of evaluation *( → later)*

## What is a query?

- Specification of a string (word, term) w , or several words $w_1,\dots,w_k$, which are searched for in documents (for all (Boolean operation AND) or for at least one (Boolean operation OR))

example:

Search for all documents containing

a) “Universität Karlsruhe”

b) ”Algorithmus”

Assumption (i): document = sequence of symbols from an alphabet Σ

⇒ there is a match for query (a) in D ⇔ $D \in \Sigma^*$Universität Karlsruhe$\Sigma^*$

⇒ – query corresponds to regular expression

– presence of a match can be checked by an appropriate finite automaton.

– requires transformation of a query into an automaton prior to execution of a query

Assumption (ii): document is represented by an index, i.e. by a data structure allowing for efficient retrieval of all relevant words of the document

⇒ requires document analysis and transformation into an index (sub) structure prior to any queries.

## What is a regular expression (over Σ)?

- standard definition of set RE of all regular expressions *(cf. Grd.Info II / Informatik III)*:
  - $\emptyset \in RE$ – $a \in \Sigma \Rightarrow a \in RE$
  - $\alpha, \beta \in RE \Rightarrow (\alpha+\beta), (\alpha\beta), \alpha^* \in RE$

- rules for simplification:
  - $\varepsilon = \emptyset^*$
  - $\alpha? = \varepsilon+\alpha$
  - $[a_i..a_j] = a_i+a_{i+1}+a_{i+2}+\dots+a_j$ *(assumption: Σ is ordered)*
  - $\alpha^{\le k} = \sum_{i=0}^{k} \alpha^i$ $\alpha^{\ge k} = \sum_{i=k}^{\infty} \alpha^i$

- example: $\Sigma^*((A+a)lgorithm((s+us+en+ic+isch)?)\Sigma^*)^{\ge 5}\Sigma^*$

  i.e. look for documents, containing at least 5 times certain variants of the word “Algorithmus”.

⇒ Allows for very specific searches but presumably not suitable as a format for specifying queries for search engines on the Internet (at most suitable as a purely internal format).

Search engines usually allow for simple specification of queries (list of words plus choice of conjunctive or disjunctive search ) and extensions for more complex queries.

## Examples of search queries

In google (and in other search engines) the following queries are possible:

- Angewandte OR Informatik OR Universität OR Karlsruhe
  **97.300.000** **pages at google**
  **2007: 88.200.000**
- Angewandte AND Informatik AND Universität AND Karlsruhe
  **493.000** **pages at google**
  **2007: 268.000**
- Angewandte AND Informatik AND Karlsruhe AND NOT Mathematik
  **403.000** **pages at google**
  **2007: 41.300**
- "Angewandte Informatik Karlsruhe“
  **467 pages at google**
  **2007:739**

## Similarity search

**Look for documents, containing a word w *or for a word which is very similar to w.***

- example:
  - Hausaufgabe, Hausarbeit, Haus-auf-gabe, Heimarbeit, ...
  - Stadt, Städte, Statt,...
  - Karlsruhe, Kalrsruhe, Karlrsuhe, Kalsruhe,...

**How do we measure the similarity of words u, v?**

a) **Hamming distance**: (usually restricted to words of same length)

$d_H(u,v)$ = number of positions where u and v differ

example: $d_H$(Hausarbeit, Heimarbeit)=3,
$d_H$(Karlsruhe, Kalsruher)=7

b) **Editing distance**:

$d_E(u,v)$ = number of editing operations (delete, insert, replace, switch,..) needed to transform u into v

example: $d_E$(Hausarbeit, Heimarbeit)=3,
$d_E$(Karlsruhe, Kalsruher)=2

- obviously, editing distance is closer to intuitive notion of similarity.
- **but**: very different complexity of determining $d_H$ or $d_E$

## Complexity of determining similarity

- Hamming distance: obviously linear time
- Editing distance:
  - obviously more difficult to determine
  - depends on allowed operations:

    e.g. – arbitrary replacement
    – replacement with characters which are neighbours on the keyboard
    – insertion of hyphen, blank, …

**Lemma**:
If the only allowed operations are insertions and deletions, we have

(a) $d_E(u,v) \geq | |u| - |v| |$

(b) $d_E(u,v) = |u| + |v| - 2|w|$

where w is the longest common subsequence of u and v.

Proof: *exercise*

**Definition**: w is a **common subsequence** of u and v :
$\Leftrightarrow$ there are $u_0,\dots,u_k$ , $v_0,\dots,v_k$ , $w_1,\dots,w_k$ ,
such that $u=u_0w_1u_1\dots w_ku_k$ und $v=v_0w_1v_1\dots w_kv_k$

possible query:

- *Search for all words v such that $d_E(u,v) \leq 2$.*

## Levenshtein distance

- Operations: Insert, delete, replace
- Dynamic programming approach
- Principle of optimality: The best path from A to B has the property that, whatever the initial decision at A, the remaining path to B, starting from the next point after A, must be the best path from that point to B.

Main idea:

- Two words, s[1...m] and t[1...n]
- Gradual transformation: d[i,j] is number of operations needed to transform s[1...i] into t[1...j]
- If d[i-1,j], d[i-1,j-1], d[i, j-1] are known, d[i,j] is minimum of:

```
d[i-1,j]+1      //deletion
d[i, j-1] +1    // insertion
d[i-1,j-1] +1 // replacement if s[i]≠t[j]
d[i-1,j-1]      // if s[i]=t[j]
```

## Example

| | | S | a | t | u | r | d | a | y |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| S | 1 | | | | | | | | |
| u | 2 | | | | | | | | |
| n | 3 | | | | | | | | |
| d | 4 | | | | | | | | |
| a | 5 | | | | | | | | |
| y | 6 | | | | | | | | |

## Example

| | | S | a | t | u | r | d | a | y |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| S | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| u | 2 | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 |
| n | 3 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 6 |
| d | 4 | 3 | 3 | 3 | 3 | 4 | 3 | 4 | 5 |
| a | 5 | 4 | 3 | 4 | 4 | 4 | 4 | 3 | 4 |
| y | 6 | 5 | 4 | 4 | 5 | 5 | 5 | 4 | 3 |

## Pseudocode

```
int LevenshteinDistance(char s[1..m], char t[1..n])
int d[0..m, 0..n]
for i := 0 to m
  d[i, 0] := i
for j := 1 to n
  d[0, j] := j
for i := 1 to m
  for j := 1 to n
    if s[i] = t[j] then cost := 0
               else cost := 1
    d[i, j] := minimum(
               d[i-1, j] + 1,      // deletion
               d[i, j-1] + 1,      // insertion
               d[i-1, j-1] + cost ) // substitution
return d[m, n]
```

Running time is $O(n^2)$

## Checking for w in document D ("pattern matching")

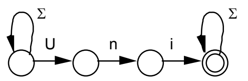
- Input: D, $w \in \Sigma^*$ where $|D| = n$ $|w| = m$ and $m \le n$
- Output: position $p \in [n-m+1]$ , where w starts in D (or all such positions or #positions)

first approach:

1. Determine the finite automaton A corresponding to $*w*$
2. Check, whether $D \in \mathbf{L}(A)$.

example:

- w= "Uni" $\Rightarrow$



A is nondeterministic, i.e. we need a deterministic version:

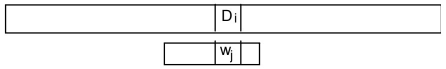
- naive approach:

```
FOR i:= 0 TO n-m DO
      j:=1;
      WHILE j ≤ m AND D[i+j] = w[j] DO
      j := j+1
      END; (* while*)
      IF j > m THEN ReportMatch(i+1)
END (*for i *)
```

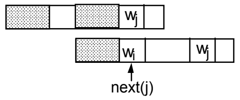
- Analysis: obviously time $O(n \cdot m)$

## Algorithm by Knuth, Morris, Pratt 1977

- **Idea**: exploit the knowledge of the prefix of w that matched before a mismatch occurred, i.e. after a mismatch move w as far as possible to the right



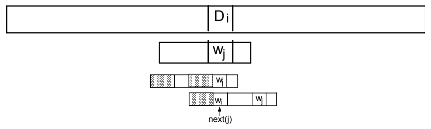
$\Rightarrow$ before executing the search compute for every position within w the maximally reasonable shift distance:



- therefore: $\text{next}(j) := \max\{i \in \mathbb{N} \mid i < j \wedge w[i] \neq w[j] \wedge \forall k \in [i-1]\ w[k] = w[j-i+k]\}$

(assuming that max(Ø)=0 $\Rightarrow$ next(1) = 0 )

## Algorithm KMP search

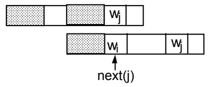


```
i:=1; j:=1;
REPEAT
   IF j=0  OR  D[i] = w[j]
       THEN   i:=i+1;  j:=j+1;
              IF j > m THEN ReportMatch(i-m); j:=next(j)
              END
       ELSE   j:= next(j)
   END
UNTIL   i > n;
```

- Analysis
  - i is incremented exactly n times.
  - j is always incremented together with i.
  - j cannot be decremented more than i is incremented,
  - $\Rightarrow$ maximally O(n) operations

$\Rightarrow$ significant reduction of worst case run time

## How to compute next(·)?



```
j:=1;  i:=0;  next[j]:=0;  w[m+1]:=x;  (* x≠w[j] for all j *)
REPEAT
  IF  i = 0  OR  w[i] = w[j]
     THEN   (*)  i := i+1;  j := j+1;
            IF w[i] ≠ w[j]
                  THEN   next[j] := i
                  ELSE   next[j] := next[i]
     ELSE   i := next[i]
  END
UNTIL  j > m
```

**Analysis**

analogously to KMP search we have a time complexity of O(m)

**Correctness**:

- to prove: before any execution of **(*)** we have:
  *For all k ∈[i] next[k] is defined correctly*
  (prefix condition) *and for all k ∈[i] we have w[k]=w[j-i+k].*

Proof: (by induction )

## Proof:

- Obviously, before the first execution of **(*)** we have j=1 and i=0, next[1]=0 is correct, and the prefix condition is satisfied.
- Assume the statement holds for all previous executions and **(*)** is reached again. Then we have to show:

  *a) next[i] is defined correctly :*

  Before the previous execution of **(*)** the statement was true, after that w[i] and w[j] have been compared:
  - If w[i] ≠ w[j] , then next[j]:=i is correct, since due to the inductive assumption we have for all k<i : w[k]=w[j-i+k] .
  - Otherwise, we set next[j]=next[i], i.e. according to the inductive assumption to the next smaller position of w, satisfying the requirements on next.

  *b) For alle k ∈[i] we have w[k]=w[j-i+k].* (prefix condition)

  Before the previous execution of **(*)** the prefix condition was true.

  If after that we have w[i] ≠ w[j], then we repeatedly decrease i by i:=next[i], respecting the prefix condition, until i=0 or an i is reached such that w[i]=w[j], i.e. the prefix condition is satisfied again.

  After the last execution of **(*)** we always have_w[j]=w[m+1] ≠ w[i], i.e. next[m+1] gets correctly defined, too.

## Example for computing next:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w = | a | b | r | a | k | a | d | a | b | r | a | **x** | |
| | | | | | | | | a | b | r | a | k | a |
| next: | 0 | 1 | 1 | 0 | 2 | 0 | 2 | 0 | 1 | 1 | 0 | 5 | |

Computing next(·):

```
j:=1;  i:=0;  next[j]:=0;    w[m+1]:=x;  (* x≠w[j] for all j *)
REPEAT
  IF  i = 0  OR  w[i] = w[j]
      THEN (*)  i := i+1;  j := j+1;
           IF w[i] ≠ w[j]
               THEN next[j] := i
               ELSE next[j] := next[i]
      ELSE   i := next[i]
  END
UNTIL  j > m
```

## Further improvement: Boyer, Moore 1977

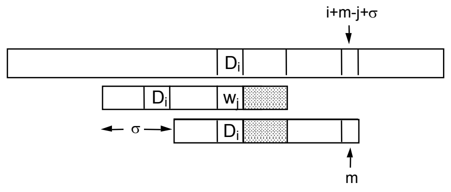
- **Idea**:
  - **Compare w from right to left instead of from left to right.**
  - **Whenever there is a mismatch, move w as far as possible to the right.**

⇒
- *worst case:*
  Mismatch occurs at leftmost symbol, but w may be moved by one position only, i.e. in the worst case we get O(n·m) comparisons.
- *best case:*
  Mismatch occurs at rightmost symbol, w may be moved by m positions,
  i.e. in the best case we have O(n/m) comparisons only.
- *average case:*
  hard to analyse, but simulations and practical applications show sublinear behaviour O(n/m)!

## How to determine the shift distance σ?

Assumptions:

- i: current position in D
- j: current position in w
- $w_j \neq D_i$



Heuristics for computing σ:

- <u>**occurrence heuristic**</u>:
  *shift w to the next occurrence of symbol $D_i$ in w.*

⇒ Computation of a table δ, containing for every character c of the alphabet the distance to the rightmost occurrence of c in the pattern:

$$\delta(c) := \begin{cases} m, & \text{if } c \text{ not in } w \\ m-r, & \text{if } c = w_r \text{ and } c \neq w_k \text{ for } r < k \leq m \end{cases}$$

- This leads to the following shift distance for $D_i$=c :

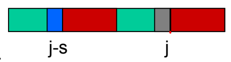
$$\sigma_V(j,c) := \begin{cases} \delta(c)+1, & \text{if } m-j \geq \delta(c) \quad \text{i.e. the rightmost ‘c‘ is to the right of position j} \\ \delta(c)-(m-j), & \text{if } m-j < \delta(c) \quad \text{i.e. the rightmost ‘c‘ is to the left of position j} \end{cases}$$

⇒ For most symbols of the alphabet we get δ(c)=m, i.e. we get large shift distances and, consequently, sublinear behaviour.

## Improving the heuristic:

- Compute δ(c) relative to the current position in w (i.e. compute 2-dimensional table with entries δ'(j,c); in this case we would have $\sigma_V(j,c)$ = δ'(j,c) ).
- If j < m, we shift w with respect to $w_m$ and not wrt the symbol at position j.
- <u>**Match heuristic**</u>:
  - *Shift w to the next occurrence of the matching suffix of w, having a different symbol to its left.*

$$\sigma_H(j) := \min\{s \mid s \geq 1 \wedge (s \geq j \vee w_{j-s} \neq w_j) \wedge (\forall k \text{ with } j < k \leq m \quad (s \geq k \vee w_{k-s} = w_k))\}$$



<u>example</u>:

w= "banana"        Suffix "ana" occurs again with next symbol 'b' instead of 'n'.

⇒ the match heuristic shifts w the more, the further to the left the suffix occurs again.

⇒ use of $\sigma_H$ is most effective for aperiodic w.

<u>**combined heuristic**</u>:

- compute        $\sigma(j) := \max(\sigma_V(j,c), \sigma_H(j))$
- this essentially corresponds to the original algorithm by Boyer,Moore
- statement from the literature (Frakes&Baeza-Yates) :
  $\sigma_H$ not effective, *since periodic words are rare.*
  *(this argument is not intelligible, since for aperiodic words $\sigma_H$ has a maximal value!!)*

## Preprocessing time

- Computation of Tables $\delta$, $\delta'$, $\sigma_H$ in time O(m+a) rsp. O(m·a) with a=size of alphabet.

There are several improved versions of the Boyer-Moore algorithm.

**related problem**:

*Search concurrently for several patterns $w_1,\dots w_k$ within document D.*

naive approach:

Apply k times the algorithm of Boyer, Moore.

$\Rightarrow$ k times the time of Boyer, Moore

improved approach:

*Search concurrently for all k patterns, i.e. compute shift distances wrt all k patterns.*

$\Rightarrow$ larger tables, but (almost) the same time as for looking for a single pattern!.

- Realised by
  - Aho,Corasick (CACM 1975, similar to Knuth, Morris, Pratt)
  - Commentz-Walter (ICALP 1979, in LNCS 71, combines ideas of Aho,Corasick and Boyer, Moore)

## Presented approach to pattern matching:

- *Analyse the pattern, to search arbitrary documents as fast as possible.*

**different approach**:

- *Analyse the document to allow for an efficient search for arbitrary patterns.*

**Step 1**: *Divide document D into words.*

*(corresponds to lexical analysis in compiling, or to string tokenising in Java)*

**What is a word?**

- string without special characters (colon, blank, hyphen, ...)
  - Problems: Jean-Claude, state-of-the-art, MS-DOS, Arbeits-recht, Chapter 2.3,
- string without digits
  - Problems: BS2000, F-16, $H_2O$ (variant: no digit as first character)
- **capital or small letters?**
  - Usually unimportant, but sometimes very important (e.g. keywords in programming languages!)

Lexical Analysis requires construction of a suitable automaton (using standard tools).

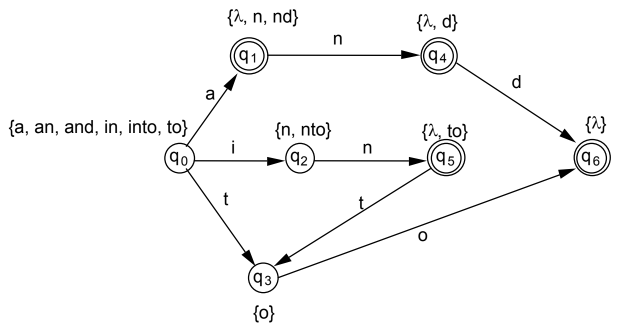
## Step 2: Filtering relevant words

**Which words are relevant?**

Answer depends on definition of a stop list ("negative list"), containing all irrelevant words.

- Stop list may be integrated into lexical analysis (by building an appropriate finite automaton):

example: automaton for stop list {a, an, and, in, into, to}:



- Such an automaton is easily constructed.

## Stemming

Examples :

| Suffix | replacement | example |
|---|---|---|
| ies | i | ponies -> poni |
| s | | cats -> cat |
| ed | | plastered -> plaster |
| ing | | motoring -> motor |
| at | ate | conflat(ed) -> conflate |
| bl | ble | troubl(ing) -> trouble |
| iz | ize | siz(ed) -> size |
| ational | ate | relational -> relate |
| y | i | happy -> happi |
| abli | able | conformabli -> conformable |
| fulness | full | hopefulness -> hopefull |
| biliti | ble | sensibiliti -> sensible |
| icate | ic | triplicate -> triplic |
| ative | | formative -> form |
| ement | | replacement -> replac |
| ive | | effective -> effect |

**advantages:**

- fewer words (reduction of search structures by up to 50%)
- automatic extension of search to related words

## Step 3: Building an efficient search structure (an index)

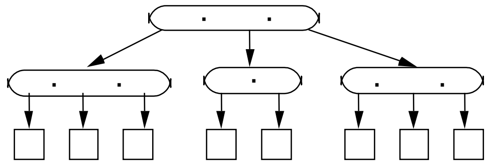
- Construct an **inverted file**, i.e.
  For every word of the document specify
  - where it occurs (document/location(s) within the document) and/or
  - its relevance (weight) and/or
  - how often it occurs

possible data structures:

- **sorted list with random access (array)**
  - pros:
    - simple structure
    - logarithmic access time (using binary search)
  - cons:
    - expensive administration
      (linear time for insertions, deletions)
- **tree structures:**
  - B-trees: balanced search trees with about one page per node, especially useful for large search structures on hard disk.
  - Tries (especially PATRICIA- tries)
    (reTrieval), separation of search words into symbols

## B-trees: *(“B” due to R.Bayer)*

- Data structure (balanced tree) supporting an efficient search for key( word)s from an ordered set
- B-tree of degree m:
  - (B1) All leaves have the same depth.
  - (B2) Every inner node (except for the root) has at least $\lceil m/2 \rceil$ successors.
  - (B3) The root has at least 2 successors.
  - (B4) Every node has at most m successors.
  - (B5) Every node with k successors contains k-1 keys.
- example: (m=3)



- Search, inserte and delete in time $O(\log_m n)$ (n = # keys)
- In practice, m is about half the memory page size.

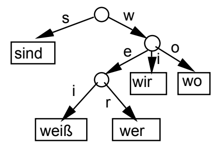
$\Rightarrow$ all the keys of a node fit into one page of memory.

## Tries

- data structure for the efficient retrieval of words by character based comparisons:
- tree with branching degree $\leq$ size of alphabet

example:

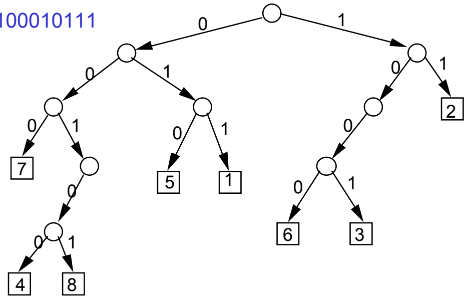
- {wer, weiß, wo, wir, sind}



- all "semi-infinite strings (*sistrings*)“ / suffixes of a character sequence:

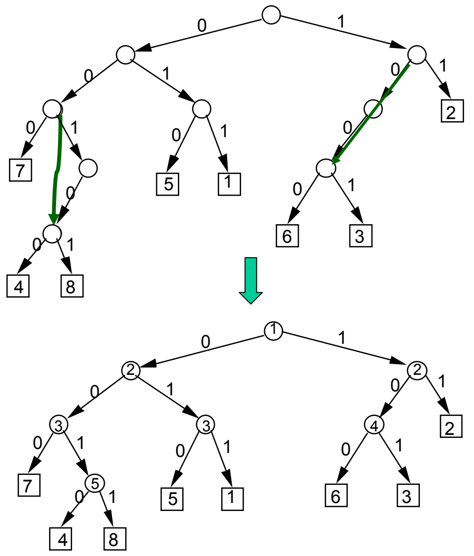
  e.g. the first eight suffixes of 01100100010111

(the different sistrings are identified by their starting position)



## Improvement:

- delete redundant comparisons, i.e. delete nonbranching paths



- these nonredundant trees are also called PATRICIA-trees:

  *Practical Algorithm To Retrieve Information Coded In Alphanumerical*

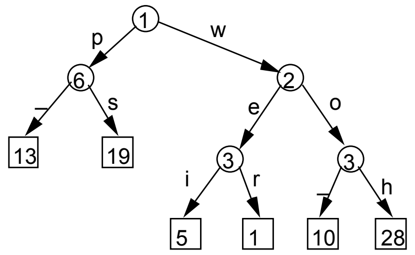
## PAT-Tree™ (Gonnet):

PATRICIA tree, containing all the suffixes of a text.

*(rsp. all the suffixes starting at reasonable text positions , e.g. at the beginning of a word)*

example:

- "wer weiß wo Peter Petersen wohnt"



**Searching in a PAT tree:**

a)*Determine the starting point for pattern matching*
(i.e. the suffix of the text coinciding with the query at the predetermined positions).

b)Compare search term and suffix.

**Insertion** into a PAT-Tree: *(similar to insertion into search trees)*
(example: build PAT-Tree for abbabbbabbc, *see next slide*)

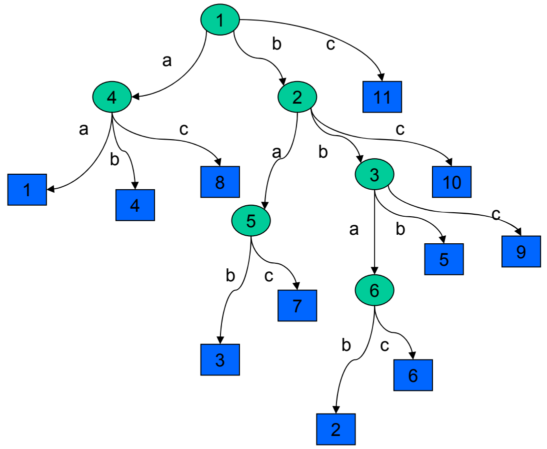
**frequency of w** in the text =#leaves in the corresponding subtree of the PAT tree
(every node stores the number of leaves in the subtree)

**most frequent word** in the text =
search all paths from the root to the next blank, determine largest subtree found

further efficient operations:

- **prefix search**: determine all the words (suffixes) matching a given prefix
- **range search** (e.g. all words lexicographically in between "abc" and "acc")
- **longest repetition search**: search for the
  "longest text in between two character repetitions"
  (longest path from the root to an inner node)
- **regular expression search**..
- ...

## Pat tree for the string abbabbbabbc:

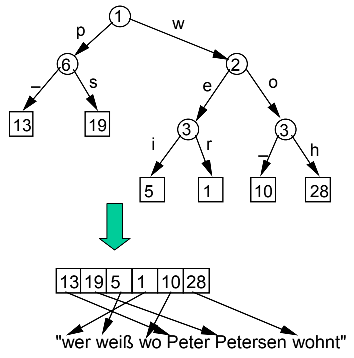


## Implementation variants:

- "**bucketing**": transform subtrees of a certain size into supernodes (as in a B-tree)
- **PAT-Array**: extreme case of bucketing, i.e. all leaves are ordered lexicographically in an array

$\Rightarrow$ simpler data structure, but at most a factor of log n more time per operation than in a PAT tree

example



## Remarks

- PAT trees have been designed because of an external project at the University of Waterloo:
  - *Construction of an index for the Oxford English Dictionary:*
- size: 600 Mb text.
- Quick (and naïve) estimate of necessary time:
  On the average, the insertion of a new entry into the index needs about log n random accesses to disk.
  - at most about 30 disk accesses per sec (*standard access time 5 to 50 ms*).
  - 120,000,000 index positions

  $\Rightarrow$ about 30,000 hours or 3.5 years to build the index!

  Even if there is only one disk access per entry, we need about 46 days!
- At the end of the project:
  consequent use of "practically efficient" algorithms reduced the time to a weekend!

## What does "practically efficient" mean?

- Avoid frequent disk accesses
  - construct small indices in main memory (O(n/m) indices of size O(m))
  - efficient merging of small indices into a large index using
    - sequential access to secondary memory
    - random access only in main memory

$\Rightarrow$ total effort
  - $O(n^2/m)$ sequential accesses to disk
  - $O(n^2 \log n/m)$ operations with random access to main memory

  - for details look into Frakes & Baeza-Yates: *Information Retrieval*

## Memory requirements of an index:

- Typical size of a complete index:
  - between 50% to 200% of the size of the document
- Size of a Patricia tree
  - well below 100%

- typical restrictions in applications:
  - consider only a prefix of the document
    (e.g. maximally the first 200 words)
  - typical size of an index in search engines: 2% to 10% of the document size

## Measuring the relevance of query terms

- Range of different methods:
  - *statistical*: based on term frequencies in documents
  - *information theoretical*: based on information content of the terms (predictability of the occurrence of terms )
  - *probabilistic*: based on probabilities for the occurrence of words in relevant documents, requires training sets of documents, and relevance judgements by the user.

More detailed: **statistical method:** *(used most often)*

Index contains m terms $T_1,\ldots,T_m$ from n documents $D_1,\ldots,D_n$

- **term frequency** $tf_{ij}$ := frequency of $T_i$ in $D_j$
- **document frequency** $df_i$ := #documents containing $T_i$

  (sometimes also $df_i$ := frequency of $T_i$ in $D_1,\ldots,D_n$)
- **inverse document frequency** $idf_i$ := $\log(n/df_i)$
- **weight** of a term $T_i$ in $D_j$ $w_{ij}$ := $tf_{ij} \cdot idf_i$

  (i.e. the weight is large if the term occurs often within this document and rarely outside)

## Different statistical approach

use "discriminating strength " of a term (***term discrimination***):

- every document is a point (vector) in a "document space"
- every document is represented by a set of terms
- the distance of documents is proportional to the relative number of differences in their term sets.
- **discrimination value** of a term $T_i$
  $dv_i$ := change of the average distance of documents, if $T_i$ is added to (removed from) the term set.
- **weight** of $T_i$ in $D_j$ $w_{ij}$ := $tf_{ij} \cdot dv_i$
- when term T is added:
  - distances among documents containing T decrease
  - distances among documents not containing T decrease as well
  - distances between documents from the above two classes increase

$\Rightarrow$ the best term discrimination (average separation of documents) is achieved with medium frequency terms

*(see "A Vector Space Model for Automatic Indexing" Salton and Yang 1975)*

## Hyperlink analysis

- The internet contains additional information: the hyperlink structure
- Links as recommendation
  - Assumption: A hyperlink from page A to page B is a recommendation of pag B by the author of page A
  - Graph structure: Edge from A to B if A has a link to B (-> link graph)
  - Used mostly for ranking
- Common parent node
  - Assumption: If page A and page B are connected by a hyperlink, then they might be on a similar topic
  - Edge from A to B if there exists a document C that has a link to A and B (->co-citation graph)
  - Used mostly for categorization
- Simple ranking algorithm:
  - The more links pointing to a page, the more important is the page.
  - But: does not take into account the importance of referring pages

## PageRank algorithm

- Proposed by Brin&Page in 1998
- Basis for Google search
- Ideas:
  - Link is weighted by PageRank of referring page
  - Weight of referring page is divided among all outgoing links
  - PageRank corresponds to probability that a random surfer ends up at this page

$$R(A) = \frac{d}{N} + (1-d) \sum_{(B,A)\in G} \frac{R(B)}{outdegree(B)}$$

where

  - d is a parameter, usually d∈[0.1,0.2]
  - N is the total number of documents
  - G is the edge set of the link graph
- Calculation:
  - start with some arbitrary initial values
  - repeat calculation several times
- Problems:
  - not immune to manipulation (but better than pure text-based approaches)
  - favors older pages

## Further measure of relevance: popularity

- Page popularity:
  - number of accesses
  - number of appearences in query result listings

## How search engines work:

(based on SEWatch & "Information Retrieval on the WWW" IEEE Internet Computing, Vol.1, No.5, 1997)

- AltaVista
- http://altavista.digital.com/
- Ask Jeeves
- http://www.askjeeves.com/
- AOL NetFind
- http://www.aol.com/netfind/
- Excite
- http://www.excite.com/
- GoTo
- http://www.goto.com/
- HotBot
- http://www.hotbot.com/
- Inktomi
- http://www.inktomi.com/
- Infoseek
- http://www.infoseek.com/
- LookSmart
- http://www.looksmart.com/
- Lycos
- http://www.lycos.com/
- MSN (Microsoft)
- http://www.msn.com/
- Netscape
- http://www.netscape.com/
- Northern Light
- http://www.northernlight.com/
- Search.com
- http://www.search.com/
- Snap
- http://www.snap.com/
- WebCrawler
- http://www.webcrawler.com/
- Yahoo
- http://www.yahoo.com/
- Google
- www.google.com

## AltaVista: (http://altavista.digital.com/ )

- Opened in Dec. 1995, run by Compaq, partnered with Yahoo from 1996 to 1998
- largest search engine on the web, in terms of pages indexed
  *(in1999 the new search engine FAST became larger)*
- spider "Scooter" searches WWW and Usenet newsgroups
- index based on full text (for documents >100k only links are considered)
- first lines are used for abstract
- uses META tags in HTML documents for index terms and description
- daily updates
- visiting frequency depending on update frequency of document
- search types: Boolean, phrase, case-sensitive
- stemming optional
- ranking depending on
  - position in the document
  - distance between terms
- results: title, short abstract, size, date of last change
- includes information from Ask Jeeves, RealNames, LookSmart

## Search engines (cont'd):

**Excite:** (http://www.excite.com/)

- since late 1995, Excite grew quickly and took over Magellan and WebCrawler.
- Excite Search: taps into the traditional search engine listings, created by crawling the web
- Channels By Excite: lists sites by topics, plus subject information, discussions areas,...
- Excite NewsTracker: search only listings generated by crawling specialty news sites
- spider searches WWW and Usenet newsgroups
  (2 Mill. every week, rest every three weeks)
- index based on full text
- generates abstract (searches for "most important phrases")
- search types: Boolean, names, similarity, concept
- Relevance: prefers title and frequency
- no stemming
- results: rank, title, abstract, optionally sorted by web site

## Search engines (cont'd):

**Google** (www.google.com)

- makes extremely heavy use of link popularity to rank the pages it has indexed. Other services also use link popularity, but none do to the extent that Google does.
- Crawling: refreshes its index on a monthly basis
- Relevancy:
  - number and quality of links pointing at the page.
  - Link Content:
    Google looks at the text in and around links and relates them to the page they point at. That means for a page to rank well for "travel," it would need to have many links that use the word travel in them or near them on the page. Of course, it also helps if the page itself is textually relevant for travel.
  - text styles: bold text, or in header text, or a large font size
- Content: based on full text, not meta tags

## Search engines (cont’d):

**HotBot:** (http://www.hotbot.com/ )

- since May 1996, run by Wired Digital
- based on the Inktomi search engine (http://www.inktomi.com/)
- has a partnership with LookSmart for directory listings.
- uses spider "SmartCrawl" to search the web for HTML and text-documents
- uses parallel computers (workstation cluster):
  - Slurp extracts URLs, distributes them to workstations (least recently accessed CPU first)
  - index distributed over several (at least 100) machines ($\rightarrow$ parallel search!)
- index based on full text, weighted terms
- abstract from first lines
- uses META tags from HTML documents for index terms and description
- search types: Boolean, phrase, case-sensitive, media type,
- no stemming
- relevance: position and frequency
- results: title, short abstract, last date of change, term frequencies

## Search engines (cont’d):

**Lycos:** (http://www.lycos.com/ )

- since May 1994, one of the oldest of the major search engines, it began as a project at Carnegie Mellon University (Lycos comes from the Latin word for "wolf spider”)
- sites are listed in search engine listings, and in an associated directory called "Community Guides”. Sites are automatically listed in these using technology from WiseWire, a company Lycos acquired in early 1998.
- runs a rating service called Top 5% (reviews of what's best on the web).
- searches WWW (HTML), Gopher, FTP
- index distributed over several machines
- index based on full text
- uses first 135 characters for description (+60 characters from the title)
- search types: Boolean, names, symbols, similarity
- automatic stemming (may be switched off)
- relevance: sum of term weights, link popularity, user voting
- results: relevance, title, description, URL, size

## Search engines (cont’d):

**MetaCrawler**: (http://www.metacrawler.com/)

- does not have its own index
- **distributes query to several search engines**
- **analyses and verifies search results of individual search engines**
- **does additional relevance checks**
- **produces reduced list in different formats (your own choice)**
- **checks the user behaviour (clicking on a listed document leads to log script at MetaCrawler and from there to the document!)**
- can run as ”client-software" locally (i.e. as local search agent)

## Search engines (cont’d):

**Yahoo:** (http://www.yahoo.com/ )

- since late 1994, oldest major web site **directory**.
- largest directory (as opposed to search engine), listing 750,000 web sites, as of Dec.1997.
- site submissions
- associated search engines listed at the bottom of each results page (strategic advantage for those listed first!)

## Comparison of search results (done in 2001)

- Search for "latex software"
  *(goal: information on how to get Latex Software)*

| Search engine | disjunctive search | conjunctive search | Phrase search |
|---|---|---|---|
| AltaVista | 1.150.000.000 | 4.940.000 | 10.600 |
| Netscape | 85.900.000 | 381.000 | 1980 |
| WebCrawler | 87 | 88 | 67 |
| Google | 1.710.000.000 | 3.780.000 | 23.700 |
| Yahoo | 1,160,000,000 | 4,870,000 | 10,700 |
| MetaCrawler | 85 | 86 | 63 |

Quality of results:

•Precision better for conjunctive and phrase search than for disjunctive search

•In both tests

–relevant and irrelevant documents were mixed in delivered listing

⇒ it is not always sufficient to look only at the first listed documents!

## How search engines work: condensed survey

**Crawling: Factors that affect if and when a page is indexed**

| Search Engine | AltaVista | Excite | HotBot/ Inktomi | Infoseek | Lycos | Northern Light | Web Crawler |
|---|---|---|---|---|---|---|---|
| Size (millions) | Big (140) | Medium (55) | Big (110) | Small (30) | Small (30) | Medium (80) | Tiny (2) |
| Pages crawled per day | 10 million | 3 million | Up to 10 million | - | 6 to 10 million | 3 million+ | - |
| Freshness | 1 day to 1 month | 1 to 3 weeks | 1 day to 2 weeks | 1 day to 2 months | 2 to 3 weeks | 2 to 4 weeks | Updated weekly |
| Submitted Pages | 1 day | 1 to 3 weeks | Within 2 weeks | Within 2 days | 2 to 3 weeks | 2 to 4 weeks | 1 to 6 weeks |
| Non-submitted pages | 1 day to 1 month | 3 weeks | 2 weeks | 1 to 2 months | 2 to 3 weeks | 2 to 4 weeks | 1 to 6 weeks, if at all |
| Depth | No limit | No limit | No limit | Sample | Sample | No limit | Sample |
| Frames Support | Yes | No | No | No | No | Yes | No |
| Image Maps | Yes | No | No | Yes | No | Yes | Yes |
| Password Protected Sites | No | Yes | No | Yes | Yes | Yes | No |
| Link Popularity | No | Yes | Yes | No | Yes | No | Yes |
| Learns Frequency | Yes | No | Yes | Yes | No | No | No |
| Robots.txt | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Meta Robots | Yes | No | Yes | Yes | Yes | Yes | Yes |
| URL Status Check | Yes | No | Yes | Yes | Form | No | No |

*Source: sewatch*

## Condensed survey (cont'd)

**Ranking: Factors that affect how a page is ranked ---**

| Search Engine | AltaVista | Excite | HotBot | Infoseek | Lycos | Northern Light | Web Crawler |
|---|---|---|---|---|---|---|---|
| Reviewed | n/a | Yes | n/a | Yes | No | n/a | Yes |
| Link Popularity | No | Yes | No | Yes | Yes | No | Yes |
| Meta Tags | No | No | Yes | Yes | No | No | No |
| Meta Refresh | Spam | OK | OK | Spam | OK | OK | OK |
| Invisible Text | Spam | OK | Spam | Spam | Spam | Spam | OK |
| Tiny Text | Spam | OK | Spam | OK | Spam | OK | Spam |
| Indexes ALT text | Yes | No | No | Yes | Yes | No | No |
| Indexes comments | No | No | Yes | No | No | No | No |
| Stop Words | Yes | Yes | Yes | No | Yes | No | No |
| Stemming | No | No | No | Yes | Yes | Yes | No |
| Case Sensitive? | Yes | No | Mixed | Yes | No | Mixed/Title | No |

*Source: sewatch*

## Search engine survey (cont'd)

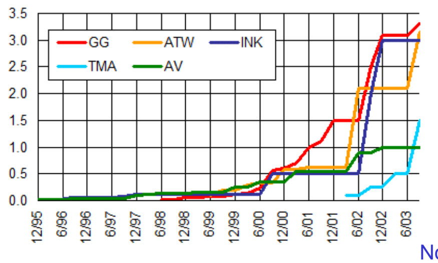
**Display: Factors that affect how a page is listed ---**

| Search Engine | AltaVista | Excite | HotBot | Infoseek | Lycos | Northern Light | Web Crawler |
|---|---|---|---|---|---|---|---|
| Title Length | 78 | 70 | 115 | 70 | 60 | 80 | 60 |
| If No Title | Says "No title" | Says "Untitled" | Lists URL | Uses first line on page | Uses first line on page | - | Lists URL |
| Description Length | 150 | 395 | 249 | 170 - 240 | 135 - 200 | 150 - 200 | 395 |
| Meta Tag Support | Yes | No | Yes | Yes | No | No | No |
| Date | Yes | No | Yes | Yes | No | Yes | No |
| Results at a time | **10** | **10**, 20, 30, 40, 50 | **10**, 25, 50, 100 | **10**, 20, 25, 50 | **10**, 20, 30, 40 | **25** | 10, **25**, 100 |
| Display Options | Standard, Compact, Text-Only | Summaries, Titles only, Sort by site | Full Brief Titles only | Summaries, Titles Only | None | None | Titles only, Summaries |

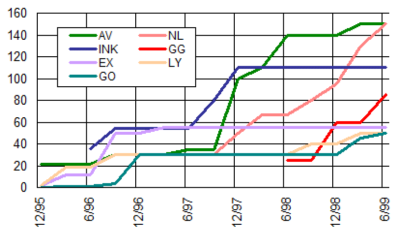
*Source: sewatch*

## Search engine sizes (source: searchenginewatch.com)

Number of indexed text documents (billions)
December 1995-September 2003

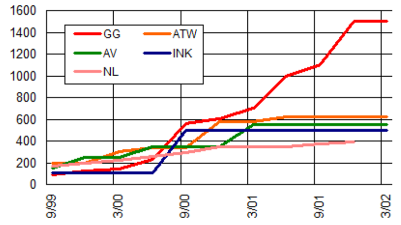


Number of indexed text documents (millions)
December 1995-June 1999

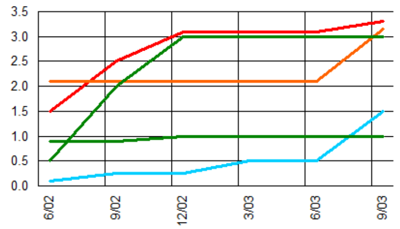


November 2004: 8 billion docs. at Google

Number of indexed text documents (millions)
September 1999-March 2002



Number of indexed text documents (billions)
June 2002-September 2003



## Naive thoughts on complexity

- Typical size of the index of an average search engine:
  Information on about 1 Bill.. Web pages
- Typical number of pages visited every day:
  10 - 100 million pages (to keep the index up to date and to expand it)
- Typical data structure for an index is a tree:
  $\Rightarrow$ average cost of one update or one query about O(log n) disk accesses

- Simplified assumptions:
  - 100 disk accesses per second (i.e. about. 10 ms per access)
  - maximally 100 port accesses per second (i.e. $\geq$ 10 ms per access)
  - #disk accesses per query < log n = log ($10^9$)< log ($2^{30}$)=30
    *(but, base 2 is not quite adequate, rather $2^8$ or $2^{10}$)*

$\Rightarrow$ time of one query $\geq$ (#disk accesses + 2)•10 ms

$\Rightarrow$Even with only one disk access per query maximally 2.8 Mill. queries per day, with 10 accesses maximally 280.000 queries.

Note: we did not consider all the additional computational costs of query processing!
For comparison: Google gets about 90 Mio queries per day (in the US)

## Some more statistics

| Searches | Per Day (Millions) | Per Month (Millions) |
|---|---|---|
| Google | 91 | 2,733 |
| Yahoo | 60 | 1,792 |
| MSN | 28 | 845 |
| AOL | 16 | 486 |
| Ask | 13 | 378 |
| Others | 6 | 166 |
| **Total** | **213** | **6,400** |

- Source: searchenginewatch.com

| Search Engine | Domain | 3/06 | 4/06 | 5/06 | 6/06 | 7/06 |
|---|---|---|---|---|---|---|
| Google | www.google.com | 58.3% | 58.6% | 59.3% | 59.8% | 60.2% |
| Yahoo | search.yahoo.com | 22.3% | 22.2% | 22.0% | 22.3% | 22.5% |
| MSN | search.msn.com | 13.1% | 12.6% | 12.1% | 12.1% | 11.8% |
| Ask | www.ask.com | 4.0% | 4.2% | 4.4% | 3.6% | 3.3% |
| AOL | search.aol.com | 1.0% | 1.0% | 0.9% | 1.1% | 1.0% |
| Others | n/a | 1.3% | 1.2% | 1.2% | 1.1% | 1.0% |
| For The 4 Week Period Ending: | | 4/1/06 | 4/29/06 | 5/27/06 | 7/1/06 | 7/29/06 |

## Consequences for system architecture of search engines

- Search engines have to update their index and answer queries simultaneously.
- Simple calculations indicate performance limits.
- Search engines need computers with large main memory to minimize disk accesses.
- To support large numbers of queries, many parallel systems are needed (large search engines usually have several hundred “query servers” arranged in so-called “web server farms”).

- But:
  - This requires to distribute the index over many computers and
  - the application of appropriate distributed algorithms for query processing.

Therefore:
To provide high performing information services in the Internet you have to know about distributed systems and distributed algorithms!

## What is spamming?

Spamming is an attempt to achieve higher relevance in searches by e.g.

**Keyword Stuffing**

- repeated use of a word in an attempt to increase a page's relevancy, e.g.

  money money money money money money money money money money money money money money money money money
  money money money money money money money money money money money money money money money money money
  money money money money money money money money money money money money money money money money money

**Invisible Text**

- setting the stuffed keywords to be the same color as the page background.
  (problem: white on coloured background):

  oney money money money money money money money money
  oney money money money money money money money money
  oney money money money money money money money money

**Tiny Text**

- repeating a word over and over in a small font size:

  money money money money money money money money money money money money money money money money money
  money money money money money money money money money money money money money money money money money
  money money money money money money money money money money money money money money money money money

and so on

Search engines are trying to detect spamming and avoid such pages.

## Conclusion

- Quality of the search engines has improved over the last years, changes have been drastic sometimes.
- Searching should be extended to semantic contents, not just syntactic.
  (one such attempt: ontobroker at this institute, course on “intelligent systems in the world wide web”, development of **semantic web, web 2.0, 3.0,…, blogs, mash-ups,…**)
- Web page designers should know about the technics of search engines in order to design pages for better, faster appearance in search engine listings.