Text Analytics
Searching Terms

Ulf Leser
A Probabilistic Interpretation of Relevance

• We want to compute the **probability** that a doc $d$ is relevant to query $q$

• The probabilistic model determines this probability iteratively **using user (or automatic) feedback**
  – Best compared to VSM with relevance feedback

• Assume there is a subset $R \subseteq D$ which contains all (and only) relevant documents for $q$

• For each document, we want to compute the probability $p(R|d)$ that $d$ belongs to $R$ (for $q$)

• Then, we use odds-scores

$$rel(d,q) \sim sim(d,q) = \frac{p(R|d)}{p(D \setminus R|d)}$$
Binary Independence Model

- Bayes (with N=D\R)

\[
sim(d, q) = \frac{p(R | d)}{p(N | d)} = \frac{p(d | R) * p(R)}{p(d | N) * p(N)} \sim \frac{p(d | R)}{p(d | N)}
\]

- \(p(R)\) is the relative frequency of relevant docs in D
- \(p(d|R)\) is the random experiment of drawing d when drawing from R
- \(p(R)\) and \(p(N)\) are independent from d – thus, both are constant for q and irrelevant for ranking documents

- Representing docs by their terms and assuming term independence

\[
sim(d, q) = \frac{\prod_{k \in d} p(k | R) * \prod_{k \notin d} p(-k | R)}{\prod_{k \in d} p(k | N) * \prod_{k \notin d} p(-k | N)}
\]
Continuation 2

- Obviously, the last term is identical for all docs. Thus

\[
sim(d,q) \approx \prod_{k \in d \cap q} \frac{p(k \mid R) \ast (1 - p(k \mid N))}{p(k \mid N) \ast (1 - p(k \mid R))}
\]

- \( \text{sim}(d,q) = \) probability of a document comprising the terms of \( d \) being relative to query \( q \)
- If we \textbf{knew} \( R \) and \( N \)
  - Life would be easy
  - Using max likelihood, we approximate all probabilities by counting term occurrences
Back to Reality

• But we don’t know R and N
• Idea: Approximation using an iterative process
  – Start with some “educated guess” for R (and set N=D\R)
  – Compute probabilistic ranking of all docs wrt q based on first guess
  – Chose relevant docs (by user feedback) or hopefully relevant docs (by selecting the top-r docs)
  – This gives new sets R and N
    • If top-r docs are chosen, we may chose to only change probabilities of terms in R (and disregard the questionable negative information)
  – Compute new term scores and new ranking
  – Iterate until satisfied

• Variant of the Expectation Maximization Algorithm (EM)
Pros and Cons

• Advantages
  – **Sound probabilistic framework**
    • Note that VSM is strictly heuristic – what is the justification for those distance measures?
  – Results converge to most probable docs
    • Under the assumption that relevant docs are similar by sharing term distributions that are different from distributions in irrelevant docs

• Disadvantages
  – First guesses are pretty bad – slow convergence
  – Terms are not weighted \((w_{ij} \in \{0,1\})\), as in the Boolean model
  – Assumes statistical independence of terms (as most methods)
  – Efficient implementation?
  – “Has never worked convincingly better in practice” [MS07]
Probabilistic Model versus VSM with Rel. Feedback

- Published 1990 by Salton & Buckley
- Comparison based on various corpora
- Improvement after 1 feedback iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>CACM 1033 Dok. 30 Anfr.</th>
<th>CISI 12684 Dok. 84 Anfr.</th>
<th>CRAN 1397 Dok. 225 Anfr.</th>
<th>INSPEC 1460 Dok. 112 Anfr.</th>
<th>MED 3204 Dok. 64 Anfr.</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Request</td>
<td>Precision 0.1459</td>
<td>0.1184</td>
<td>0.1156</td>
<td>0.1368</td>
<td>0.3346</td>
<td></td>
</tr>
<tr>
<td>IDE (dec hi)</td>
<td>Precision 0.2704</td>
<td>0.1742</td>
<td>0.3011</td>
<td>0.2140</td>
<td>0.6305</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Improvement 86%</td>
<td>47%</td>
<td>100%</td>
<td>56%</td>
<td>88%</td>
<td></td>
</tr>
<tr>
<td>Selected Terms</td>
<td>Precision 0.2479</td>
<td>0.1924</td>
<td>0.2498</td>
<td>0.1976</td>
<td>0.6218</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Improvement 70%</td>
<td>63%</td>
<td>116%</td>
<td>44%</td>
<td>86%</td>
<td></td>
</tr>
<tr>
<td>BIR-Model</td>
<td>Precision 0.2289</td>
<td>0.1436</td>
<td>0.3108</td>
<td>0.1621</td>
<td>0.5972</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Improvement 75%</td>
<td>21%</td>
<td>100%</td>
<td>19%</td>
<td>78%</td>
<td></td>
</tr>
</tbody>
</table>

- Probabilistic model in general worse than VSM+rel feedback
  - Probabilistic model does not weight terms in documents
  - Probabilistic model does not allow to weight terms in queries
Latent Semantic Indexing (Sketch with little Math)

- Until now, we were comparing terms by using equality
- Ignores **semantic relationships** between terms
  - Homonyms: bank (money, river)
  - Synonyms: House, building, hut, villa, ...
  - Hyperonyms: officer – lieutenant
  - Co-occurrence of terms in a given theme
- Idea of **Latent Semantic Indexing (LSI)**
  - Map terms into (less) **semantic concepts**
    - Which are hidden (or latent) in the docs
  - Represent and query in **concept space** instead of term space
- Finds docs that don’t even contain the query terms
Singular Value Decomposition (SVD)

- We want to find the **most important components of** $M$
  - Let $r$ be the rank of $M$
- We compute a decomposition of $M$ into the following form:
  $$M = X \cdot S \cdot Y^t$$
  - $S$ is the diagonal $r \times r$ matrix of the **singular values** of $M$, sorted in dec. order
  - $X$ is the matrix of Eigenvectors derived from $M \cdot M^t$
  - $Y$ is the matrix of Eigenvectors derived from $M^t \cdot M$
  - This decomposition is unique and can be computed in $O(r^3)$
Approximating M

- The $S_{ii}$ may be used to approximate $M$
- Compute $M_s = X_s \cdot S_s \cdot Y_s^\top$
  - First $s$ columns in $X$ -> $X_s$
  - First $s$ columns and last $s$ rows in $S$ -> $S_s$
  - First $s$ rows in $Y$ -> $Y_s$
- $M_s$ has the same size as $M$, but other (approximated) values
The similarity of any two docs can be computed as the cosine distance between their columns
- $M^t_s \cdot M_s$ is the document correlation matrix in concept space

Approximated docs are represented by their rows in $Y^t_s$

How can we compute the distance between a query and a doc in concept space?
- Easy
- Assume $q$ a new row in $M$
- We have to first apply the same transformations to $q$ as we did to all the docs
  - $q' = q^t \cdot X_s \cdot S_s^{-1}$
  - This vector may now be compared to the doc vectors as usual
Pros and Cons

- Strong argument: Made it into practice, used by many search engines
- Pros
  - Speed-up through less computation in query evaluation
  - Generally leads to an increase in precision (rather than recall)
- Cons
  - Computing SVD is expensive
    - Fast approximations of SVD exist
    - Do not update with every new document
    - Use stemming, stop-word removal etc. to already shrink the original term-document matrix
  - Comparing the ranks is expensive
    - VSM etc. use inverted files (later) on the terms of the document
    - But we cannot simply index the “concepts” of $M_s$
    - Thus, LSI needs other techniques than indexing (read: lots of memory)
Content of this Lecture

- Searching strings
- Naïve exact string matching
- Searching in linear time
- Boyer-Moore / Hoorspool / Sunday
Searching Strings in Text

- All IR models require to find occurrences of terms in documents
- Fundamental operation: find(k,D) -> P^D
  - Where k is a term, not an entire query
- **Online searching**: Consider docs and query as new, no preprocessing
  - Now
- **Indexing**: Preprocess docs and use index for searching strings
  - Later
Types of String Searching

- Exact or approximate
  - Exact search: Find all occurrences of k in D
  - Pattern matching: Given a regular expression, find all matches in D
  - Approximate search: Find all substrings in D that are similar to k
    - Strings that are phonetically similar (Soundex)
    - Strings that are only one typo away
    - Strings that can be produced from k by at most n operations of type “insert a letter”, “delete a letter”, “change a letter”
    - ...

- Word or substring
  - Searching words (k is a word): After tokenization
  - Searching substrings: Across token/sentence... boundaries

- Searching one or multiple strings at once in D
Preliminaries: Strings

- **Definition**
  
  A **String** $S$ is a sequential list of symbols from a finite alphabet $\Sigma$
  
  - $|S|$ is the number of symbols in $S$
  - Positions in $S$ are counted from 1,...,$|S|$.
  - $S[i]$ denotes the symbol at position $i$ in $S$.
  - $S[i..j]$ denotes the substring of $S$ starting at position $i$ and ending at position $j$ (including both).
  - $S[..i]$ is the prefix of $S$ until position $i$.
  - $S[i..]$ is the suffix of $S$ starting from position $i$.
  - $S[..i]$ ($S[i..]$) is called a true prefix (suffix) of $S$ if $i \neq 0$ and $i \neq |S|$.

- When studying the complexity of string algorithms, we usually only count the number of comparisons of strings.
Substring Search

• Sometimes, we want to search for substrings
  – Does not require (erroneous) tokenization
    • “U.S.”, “35,00=.000”, “alpha-type1 AML-3’ protein”, ...
  – Search across tokens / sentences
    • “, that ”, “happen. “, ...
  – Searching prefixes, infixes, suffixes, stems
    • “compar”, “ver” (German), ...

• Substring search is a step towards RegExp search
  – Not this lecture

• Searching substrings is harder than searching terms
  – Number of terms doesn’t increase much from a certain point on
    • English: ~ 1 Million terms, but 200 Million potential substrings of size 6
  – Number of terms in a document versus number of substrings
    • 1 Million words, average length 10: 10 Million substrings
Exact Substring Matching

- **Given**: Pattern $P$ to search for, text $T$ to search in
  - We require $|P| \leq |T|$
  - We assume $|P| << |T|$
- **Task**: Find all occurrences of $P$ in $T$

**Eco RV - GATATC**

tcagcctactaatataaatctttctgtagtaagtgctaatgaagatcagaaataataaatatcagctactacagagtttcctaaactcttcacagattgtcataatgtagttaataaatgaagatcagaaataataaatatcagctactacagagtttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctachtacattatgtagttaataaatgaagatcagaaataataaatatcagctactacagagttttcttctach
Content of this Lecture

- Searching strings
- Naïve exact string matching
- Searching in linear time
- Boyer-Moore
- BM-Variants and comparisons
Naive Substring Searching

1. Align P and T at position 1
2. Compare symbols from P with symbols from T from left to right
   - If symbols are not equal: goto 3
   - Otherwise
     - All symbols of P have been compared: Shout “here”, goto 3
     - Otherwise: compare next symbol from P and T, goto 2
3. Move P one position to the right, goto 2
4. If position of P in T < |T|-|P|+1, goto 2
5. Stop

T  cttgagatcgctga
P  gagatc
    gagatc
    gagatc
    gagatc
    gagatc
    gatatc
    gatatc
    gatatc
Algorithm

for i = 1 to |T| - |P| + 1
    match := true;
    j := 1;
    while ((match) and (j <= |P|))
        if (T[i+j-1] <> P[j]) then
            match := false;
        else
            j++;
        end while;
    if (match) then
        OUTPUT i
    end for;

Worst-case

<table>
<thead>
<tr>
<th>T</th>
<th>aaaaaaaaaaaaaaaaa</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>aaaaaat</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

N\# of comparisons: : n * (m-n+1) => O(m*n)
Content of this Lecture

- Searching strings
- Naïve exact string matching
- Searching in linear time
- Boyer-Moore algorithm
- BM-Variants and comparisons
Z-Box Algorithm

- A not so simple **linear time** substring search algorithm
- Proceeds in two phases
  - Build $S := P*T$
  - Preprocessing: Learn in linear time about the structure of $S$
  - Search: Scan in linear through $S$ to find all occurrences of $P$ in $T$
- Disadvantage: As we will see, the linear worst case is also the average and best case
- Other algorithms have sublinear average case (and linear worst case)
  - Later: Boyer-Moore algorithm and variants
**Preprocessing**

- **Definition**
  - Let $i > 1$. Then we define $Z_i(S)$ to be the length of the longest substring $x$ of $S$ with
    - $x = S[i..i+|x|-1]$ (x starts at Position $i$ in $S$)
    - $S[i..i+|x|-1] = S[1..|x|]$ (x is a prefix of $S$)
  - Then, we call $x$ the **Z-Box** of $S$ at position $I$ and length $Z_i(S)$
Beispiele

\[ S = aabcaabxaaz \]

\[ S = aaaaaa \]

\[ S = baaaaa \]
Second Phase

- Assume we could compute in linear time all Z-boxes of S
- Then, the second phase is easy

\[
S := P || \'\$\' || T; \quad // (\$ \notin \Sigma)
\]
compute Z-Boxes for S;
for \(i = |P|+2\) to \(|S|\)
  if \((Z_i(S) = |P|)\) then
    print \(i-|P|-1\); \quad // P in T at position i
  end if;
end if;

- Complexity
  - Loop is passed \(|S|\) times => \(O(m)\)
First Phase: Computing Z-boxes

- Naive algorithm

```plaintext
for i = 2 to |S|
    Zi := 0;
    j := 1;
    while (((S[j] = S[i + j - 1])) and (j <= |S|))
        Zi := Zi + 1;
        j := j + 1;
    end while;
end for;
```

- This doesn't help much
  - Let |P|=n, |T|=m
  - $O((m+n)^2) + O(m) \sim O(m^2) = O(|S|^2)$
Preliminaries

- Definition
  - Let $i > 1$. We define
    - $r_i$ is the right-most endpoint of any Z-box starting at or before $i$
    - $l_i$ is the starting position of the longest Z-box ending at $r_i$
  - $l_i$ is unique, because there is only one Z-box at each position
  - Thus: $S[l_i..r_i]$ is the longest Z-box which contains position $i$ and reaches the furthest to the right
Computing the $Z_i$ Values

• Idea
  – Reuse already computed $Z_i$ for the computation of $Z_k$ ($k > i$)
  – Run through the string (variable $k$)
  – Always keep the last $l=l_k$ und $r=r_k$
  – Apply some clever tricks to compute $Z_k$ from previous $Z$-boxes

• Induction
  – Start at position $k=2$
    • Compute $Z_2$ (in a naïve way – nothing to save yet)
    • If $Z_2 > 0$, set $r=r_2$ (=2+$Z_2$-1) and $l=l_2$ (=2), otherwise $r=l=0$
  – Induction
    • Let $k>2$ and we know $\forall j<k$: $Z_j$
    • Furthermore, we know $r$, $l$ (from $k-1$)
Z-Algorithm, Case 1

- Case 1: $k > r$
  - Thus, no previously computed Z-box contains $k$
  - Thus, we have never before looked further than $k$
  - Then there isn’t much we can do
    - Compute $Z_k$ naively, symbol by symbol
    - If $Z_k > 0$, set $r = r_k$ and $l = l_k$

Example 1

```
CTCGAGTTGCAG
0  1  0
0  ?
```

Example 2

```
CTACTACTTTTGAGAG
0  0  5
0  ?
```
Z-Algorithm, Case 2

- Case 2: \( k \leq r \)
  - Situation:

    - Z-Box \( Z_l \) is a prefix of \( S \)
    - Substring \( \beta = S[k..r] \) also appears at position \( k' = k-l+1 \) in \( S \)
    - We have analyzed this position before; thus, we know \( Z_{k'} \)
    - Of course, \( Z_{k'} \) and \( Z_k \) may be longer/shorter than \( |\beta| = r-k+1 \)
    - Though we don’t know yet \( S[r+1..] \), we have already seen \( S[k'+1..] \)
Z-Algorithm, Case 2.1

- Two sub-cases. First sub-case: $Z_k' < |\beta| = r-k+1$
  - This implies that the symbol at position $k'+Z_k'$ does not match with the symbol at position $Z_k'$. However, this implies that $S[k+Z_k']$ will produce the same mismatch.
  - Thus: $Z_k = Z_k'$; keep $r$ und $l$
Example

\[
\begin{array}{cccc}
  \gamma & \alpha & [\gamma] \beta & \alpha \\
  Z_k' & k' & l & k + r \\
  k' + Z_k, -1 & l + Z_k, -1
\end{array}
\]

\[
\beta = |ABD|; \ k' = 6; \ Z_6 = 2 < |\beta|
\]
Z-Algorithm, Case 2.2

• Second sub-case: \( Z_{k'} \geq |\beta| \)
  - This implies that \( \beta \) is a prefix of \( S \) (but not necessarily the longest)
  - We do know that, if \( Z_{k'} > |\beta| \), then \( S[|\beta|+1]=S[k'+|\beta|] \)
  - But we don’t know anything about \( S[r+1] \)
    • We have never positively matched this symbol
  - Procedure
    • Match naively \( S[r+1..] \) with \( S[|\beta|+1..] \)
    • Let the first mismatch occur at position \( q \)
    • Let \( Z_k=q-k; \ r=q-1; \) if \( q\neq r+1: \ l=k \)

\[
\begin{array}{c}
\beta \rightarrow \alpha \rightarrow \beta \\
\k' \rightarrow \k'+Z_{k'-1} \\
q
\end{array}
\quad
\begin{array}{c}
\alpha \rightarrow \beta \\
l \rightarrow k \rightarrow r \\
q
\end{array}
\]
Example

\[ \beta = |AB|; \ k' = 6; \ Z_6 = 4 > |\beta| \]
Algorithm

match $Z_2$;
set $l, r$;
for $k = 3$ to $|S|$ 
    if $k > r$ then
        match $Z_k$;
        set $r, l$;
    else
        $k' := k - l + 1$;
        $b := r - k + 1$; // This is $|\beta|$
        if $Z_{k'} < b$ then
            $Z_k := Z_{k'}$;
        else
            match $S[r+1..]$ with $S[b+1..]$ until $q$;
            $Z_k := q - k$; $r := q - 1$; $l := k$;
        end if;
    end if;
end for;
Complexity

• Theorem
   *The Z-Box algorithm computes all Z-boxes in $O(|S|)$*

• Proof
  – We estimate two counts: $m =$ "number of positive matches" and $m' =$ "number of mismatches"
  – We show that $m' < |S|$
    • At $k=2$, we can produce at most one mismatch
    • Case 1: Maximally one
    • Case 2.1: No comparisons at all, hence no mismatches
    • Case 2.2: Maximally one
  – Thus, there is **at most one mismatch** for every position of $S$
  – This shows that $m' \leq |S|$
Complexity 2

**Proof (continuation)**

- We estimate two counts: \( m = \text{"number of positive matches"} \) and \( m' = \text{"number of mismatches"} \)
- We show that \( m < |S| \)
  - Fundamental observation: Every positive match moves \( r \), and we never compare a symbol left from \( r \) again
  - \( k = 2 \): At most \( |S| \) matches, \( r \) is moves for every match
  - Case 1: At most \( |S| \) matches, \( r \) is moves for every match
  - Case 2.1: No comparisons, hence no matches
  - Case 2.2: At most \( |S| \) matches, \( r \) is moves for every match
- Thus, every position in \( S \) may produce at most one positive match
- This shows that \( m \leq |S| \)
- qed.
Together

- We can compute Z-boxes in $O(|S|) = O(m+n)$
- Using the Z-boxes, we can find all occurrences of $P$ in $O(|S|)$
- Taken together, the Z-box algorithm solves the exact substring search problem in $O(m+n)$
\begin{array}{|l|l|c|c|c|}
\hline
k & \text{What happens} & Z_k & l & r \\
\hline
2 & \text{Start of induction} & 0 & 0 & 0 \\
3 & k>r; \text{Naïve matching, 1 mismatch} & 0 & 0 & 0 \\
4 & k>r; \text{Naïve matching, 1 mismatch} & 0 & 0 & 0 \\
5 & k>r; \text{Naïve matching, 3 matches, 1 mismatch} & 3 & 5 & 7 \\
6 & 6 \leq 7; k'=2;b=2; Z_2=0; \text{Thus, } Z_k < b \text{ and } Z_k = Z_{k'} & 0 & 5 & 7 \\
7 & 7 \leq 7; k'=3;b=1; Z_3=0; \text{Thus, } Z_k < b \text{ and } Z_k = Z_{k'} & 0 & 5 & 7 \\
8 & 8>7; \text{Naïve matching, 1 mismatch} & 0 & 5 & 7 \\
9 & 9>7; \text{Naïve matching, 1 mismatch} & 0 & 5 & 7 \\
10 & 10>7; \text{Naïve matching, 1 mismatch} & 0 & 5 & 7 \\
11 & 11>7; \text{Naïve matching, 7 matches, 1 mismatch} & 7 & 11 & 17 \\
12 & 12 \leq 17; k'=2;b=6; Z_2=0; Z_k < b \text{ and } Z_k = Z_{k'} & 0 & 11 & 17 \\
13 & 13 \leq 17; k'=3;b=5; Z_3=0; Z_k < b \text{ and } Z_k = Z_{k'} & 0 & 11 & 17 \\
14 & 14 \leq 17; k'=4;b=4; Z_4=0; Z_k < b \text{ and } Z_k = Z_{k'} & 0 & 11 & 17 \\
15 & 15 \leq 17; k'=5;b=3; Z_5=3; \text{Thus } Z_k \geq b; \text{ match } S[18..] \text{ with } S[4..]; 5 \text{ matches} & \text{und success} & & \\
\hline
\end{array}

\text{k'} := k-l+1; b := r-k+1; \\
Z_k := q-k; l := k; r := q-1;
What happens

<table>
<thead>
<tr>
<th>( k )</th>
<th>What happens</th>
<th>( Z_k )</th>
<th>( l )</th>
<th>( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Start</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>( k &lt; r; k' = 2; b = 2; Z_2 = 3; Z_k \geq b ); match ( S[5..] ) with ( S[3..] ); 1 mismatch; ( q = 5 )</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>( k \leq r; k' = 3; b = 1; Z_3 = 2; Z_k \geq b ); match ( S[5..] ) with ( S[3..] ); 1 mismatch; ( q = 5 )</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>( k &gt; r; ) Naïve matching, 1 mismatch</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>( k &gt; r; ) Naïve matching, 1 mismatch</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>( k &gt; r; ) Naïve matching, 4 matches, 1 mismatch</td>
<td>4</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>( 8 \leq 10; k' = 2; b = 3; Z_2 = 3; Z_k \geq b ); match ( S[11..] ) with ( S[4..] ); 1 / 1; ( q = 12 )</td>
<td>4</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>( 9 \leq 11; k' = 2; b = 3; Z_2 = 3; Z_k \geq b ); match ( S[12..] ) with ( S[4..] ); 1 / 1; ( q = 12 )</td>
<td>4</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>( 10 \leq 12; ) ...</td>
<td>4</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ k' := k-l+1; b := r-k+1; \]
\[ Z_k := q-k; l := k; r := q-1; \]
Content of this Lecture

- Searching strings
- Naïve exact string matching
- Searching in linear time
- Boyer-Moore
- BM-Variants and comparisons
Boyer-Moore Algorithm

- Basic idea
  - Align strings as in the naive algorithm
  - Compare symbols from right to left, starting with the last character of P
  - Outer loop: Try to shift P more than one position
  - Inner loop: Try to start comparisons of symbols at a position left from the end of P
  - Use two independent tricks: bad character rule and good suffix rule
- Especially the outer-loop optimization yields large average case improvements, especially if $|\Sigma|$ is large (like for English text)
- BM is sublinear in average case
  - Our presentation has quadratic worst case
  - Improvements to linear worst case exist
Bad Character Rule

• Observation
  – Assume we aligned $P[n]$ with $T[j]$ for some $j \geq n$
  – Let $i$ be the position in $P$ of the first (right-most) mismatch
  – Let $x$ be the symbol at position $j-n+i$ in $T$
    • Which as lead to the mismatch
  – Where may $x$ match in $P$?
    • Case 1: $x$ doesn’t occur at all in $P$
    • We can safely shift $P$ by $i$ positions to the right
      – Thus, we jump behind $x$ in $T$

\[\begin{array}{c}
T & \text{xabxfabzzabxzzbzzb} & T & \text{xabxfabzzabwzzbzzb} \\
\text{P} & \text{abwxyabzz} & \text{P} & \text{abwxyabzz} \\
\text{\textup{\textcircled{P}}} & & & \\
\end{array}\]

How far can we shift now?
Bad Character Rule 2

- Observation
  - Assume we aligned $P[n]$ with $T[j]$ for some $j \geq n$
  - Let $i$ be the position in $P$ of the first (right-most) mismatch
  - Let $x$ by the symbol at position $j-n+i$ in $T$
  - Let $l$ be the right-most occurrence of $x$ in $P$
  - Where may $x$ match in $P$?
    - Case 1: $x$ doesn’t occur at all in $P$. Shift by $i$ positions.
    - Case 2: $l < i$. Shift by $i-l$ positions

How far can we shift now?
Bad Character Rule 3

• Observation
  – Assume we aligned $P[n]$ with $T[j]$ for some $j \geq n$
  – Let $i$ be the position in $P$ of the first (right-most) mismatch
  – Let $x$ be the symbol at position $j-n+i$ in $T$
  – Let $l$ be the right-most occurrence of $x$ in $P$
  – Where may $x$ match in $P$?
    • Case 1: $x$ doesn’t occur at all in $P$. Shift by $i$ positions.
    • Case 2: $l<i$. Shift by $i-l$ positions
    • Case 3: $l>i$. Shift by 1 position (wait ...)

There is a „z“ to the right from $i$. What can we do?
• Definition

\textit{Given }P, \textit{ let }R(x)\textit{ by defined as (for all }x \in \Sigma)\textit{ }

\begin{itemize}
  \item If }x \notin P, \textit{ then }R(x) := 0,
  \item Otherwise, let }R(x)\textit{ be the position of the right-most occurrence of }x \textit{ in }P\textit{ }
\end{itemize}

• We can easily compute }R\textit{ in }O(n)\textit{ }

\begin{itemize}
  \item How?
\end{itemize}

• Bad character rule in compact form

\begin{itemize}
  \item Assume we aligned }P[n]\textit{ with }T[j]\textit{ for some }j \geq n\textit{ }
  \item Let }i\textit{ be the position in }P\textit{ of the first (right-most) mismatch}
  \item Let }x\textit{ by the symbol at position }j-n+i\textit{ in }T\textit{ }
  \item Shift }P\textit{ by max(1, }i - R(x)\textit{)}
\end{itemize}
Extended Bad Character Rule

- Clearly, all occurrences of x to the right of i can be ignored
- Thus, it is better to shift P such that x aligns with the right-most occurrence of x in P that is to the left of i

\[
\begin{array}{c}
T \quad \text{xabxkabzzabwzzbzzb} \\
\text{P} \quad \text{abzwyabzz}
\end{array}
\quad \quad \quad
\begin{array}{c}
T \quad \text{Xabxkabzzabwzzbzzb...} \\
\text{P} \quad \text{abzwyabzz}
\end{array}
\]

- This requires relative positions of x for each position i in P
Intermediate Result

- BCR: Simple, very powerful for larger alphabets
- No reduction in worst-case complexity compared to naïve string matching
Good-Suffix Rule

- Idea
  - When we have a mismatch, we usually had some matches before
  - Can we know where this match occurs in \( P \) (apart from the suffix)?
  - To know this, we need to preprocess \( P \)

- We may always shift to the right-most occurrence of the substring \( t \) in \( P \) (which is not a suffix)
- If this doesn’t exist, we may shift \( P \) by \(|P| - |t|\) positions
Case 1

- Let $i$ be the position of the first mismatch in $P$, let $t=P[n-i+1,..]$
- Let $k$ be the right end of the right-most occurrence of $t$ in $P$ with $k<n$ and $P(k-|t|) \neq P(n-|t|)$ ('$y$')
  - If no such occurrence exists, let $k=0$
- If $k \neq 0$: Shift $P$ by $n-k$ positions

Why don’t we demand that $P(k-|t|)=,x'$ ?
Case 2

- Let i \( \ldots t \ldots , k \ldots \)

- If \( k \neq 0 \): Shift \( P \) by \( n-k \) positions
- If \( k=0 \) and \( P \neq t \): Shift \( P \) by \( n-|t|+1 \) positions

- There is an advanced trick to move even further - how?
Case 3

- Let \( i \ldots t \ldots, k \ldots \)

- If \( k \neq 0 \): Shift \( P \) by \( n-k \) positions
- If \( k=0 \) and \( P \neq t \): Shift \( P \) by \( n-|t|+1 \) positions
- If \( k=0 \) and \( P=t \): Shift \( P \) by 1

\[
\begin{array}{c}
\text{T} \\
\text{P}
\end{array}
\begin{array}{c}
\text{X} \\
t
\end{array}
\begin{array}{c}
\text{X} \\
t
\end{array}
\begin{array}{c}
t
\end{array}
\begin{array}{c}
t
\end{array}
\]
Example

- Our previous worst-case example now runs in linear time
  - But only with the advanced trick
Preprocessing

- We need to know where t appears in P for t being any suffix
  - And we want the endpoint of the right-most such t

- Definition
  
  _Let L'(i) be the largest value with the following properties:_
  
  1: \( P[L'(i)-|t|+1 .. L'(i)] = P[i..n] \)
  
  2: \( L'(i) < n \)
  
  3: \( P[L'(i)-|t|] \neq P[i-1] \) (Strong good suffix)
  
  \( L'(i) = 0 \) if no such substring exists
Preprocessing

- For every suffix of $P$, we need to know the end position of its right-most re-occurrence ($L_i$) with a different symbol at the next (to the right) position
- Recall Z-box: For every prefix, the Z-boxes with the same length are occurrences of the prefix with a different symbol at the next (to the left) position
- BM-preprocessing can be accomplished by Z-box algorithm on the inverted string
- We skip the details
compute \( L'(i) \);
compute \( R(x) \) for each \( x \in \Sigma \);  // Simple BCR
\( k := n \);  // Runs thru T
while (\( k \leq m \)) do
    align \( P \) with \( T \) with right end \( k \);
    match \( P \) and \( T \) from right to left until
        mismatch: Compute shift \( s_1 \) using BCR and \( R(x) \);
        Compute shift \( s_2 \) using GSR and \( L'(i) \);
        \( k := k + \max(s_1, s_2) \);
    P matched: print \( k \);
        \( k := k + 1 \);  // Could be impr.
end while;
Example (using the EBCR)

Using BCR, we would shift by only 1

With /without trick
Content of this Lecture

- Searching strings
- Naïve exact string matching
- Searching in linear time
- Boyer-Moore
- BM-Variants and comparisons
Two Faster Variants

- **BM-Horspool**
  - Drop the good suffix rule – *GSR makes algorithm slower* in practice
    - Rarely shifts longer than GSR
    - Needs time to compute the shift
  - Instead of looking at the mismatch character \( x \), always look at the symbol in \( T \) aligned to the last position of \( P \)
    - Generates longer shifts on average (\( i \) is maximal)

- **BM-Sunday**
  - Instead of looking at the mismatch character \( x \), always look at the symbol in \( T \) after the symbol aligned to the last position of \( P \)
    - Generates even longer shifts on average
Empirical Comparison

- Shift-OR: Using parallelization in CPU (only small alphabets)
- BNDM: Backward nondeterministic Dawg Matching (automata-based)
- BOM: Backward Oracle Matching (automata-based)