Text Analytics

Searching Terms

Ulf Leser
Content of this Lecture

- Searching strings
- Naïve exact string matching
- Boyer-Moore
- BM-Variants and comparisons
Searching Strings in Text

- All IR models require finding occurrences of terms in documents
- Fundamental operation: find(k,D) -> P^D
- **Online searching**: Consider docs and query as new
  - No preprocessing
  - Can or cannot use tokenization
    - No advantage in terms of speed
    - But important for “semantics” of a query
  - Classical algorithmic problem: Substring search
- **Indexing**: Preprocess docs and use index for searching
  - Does use tokenization; can only find “words”
  - Classical IR problem: Inverted files
Types of Substring Searching

- **Exact search**: Find all exact occurrences of \( k \) in \( D \)
- **Pattern matching**: Given a regular expression \( k \), find all matches of \( k \) 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 (keyboard errors)
  - Strings that can be produced from \( k \) by at most \( n \) operations of type “insert a letter”, “delete a letter”, “change a letter”
  - ...
- **Searching one or multiple strings** at once in \( D \)
Strings

• Definition

A **String S** is a sequential list of symbols from an 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 \(i \neq |S|\)
Substring Search

- Sometimes, we want to **search for substrings**
  - Online case: docs are changing constantly
  - 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), ...

- 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 substrings is increasing much longer
    - Across word boundaries
Exact Substring Matching

• Given: Pattern P to search for, text T to search in
  – We require |P| ≤ |T|
  – We assume |P| << |T|
• Task: Find all occurrences of P in T
  – Where is “GATATC”
How to do it?

- The straight-forward way (**naïve algorithm**)
  - We use two counter: \(t, p\)
  - One (outer, \(t\)) runs through \(T\)
  - One (inner, \(p\)) runs through \(P\)
  - Compare characters at position \(T[t+p]\) and \(P[p]\)

```plaintext
for t = 1 to |T| - |P| + 1
    match := true;
    p := 1;
    while ((match) and (p <= |P|))
        if (T(t + p - 1) <> P(p)) then
            match := false;
        else
            p := p + 1;
        end if;
    end while;
    if (match) then
        -> OUTPUT t
    end if;
end for;
```
Examples

Typical case

T
P
ctgagatcgcgta
gagatc
gagatc
gagatc
gagatc
gagatc
gatatc
gatatc
gatatc

Worst case

T
P
aaaaaaaaaaaaaaa
aaaaat
aaaaat
aaaaat
aaaaat
aaaaat
...

• How many comparisons do we need in the worst case?
  • t runs through T
  • p runs through the entire P for every value of t
  • Thus: $|P|*|T|$ comparisons
  • Indeed: The algorithm has worst-case complexity $O(|P|*|T|)$
Other Algorithms

• Exact substring search has been researched for decades
  – Boyer-Moore, Z-Box, Knuth-Morris-Pratt, Karp-Rabin, Shift-AND, ...
  – All have WC complexity $O(|P| + |T|)$
  – Real performance depends much on size of alphabet and composition of strings (algs have their strength in certain settings)

• In practice, our naïve algorithm is quite competitive for non-trivial alphabets and mostly unbiased frequencies
  – E.g., English text

• But we can do better: Boyer-Moore
  – We present a simplified form
  – BM is among the fastest algorithms in practice

• Note: Much better performance possible if $T$ maybe preprocessed (up to $O(|P|)$)
Content of this Lecture

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

  - ~1600 citations in Google Scholar

- Main idea
  - Again, we use two counters (inner loop, outer loop)
  - Inner loop runs from right-to-left
  - If we reach a mismatch, we know
    - The character in T we just haven’t seen
      - This is captured by the bad character rule
    - The suffix in P we just have seen
      - This is captured by the good suffix rule

- Use this knowledge to make longer shifts in T
Bad Character Rule

• Setting 1
  – We are at position \( t \) in \( T \) and compare right-to-left
  – Let \( i \) be the position of the first mismatch in \( P \)
    • We saw \( n-i+1 \) matches before
  – Let \( x \) be the character at the corresponding pos \( (t-n+i) \) in \( T \)
  – Candidates for matching \( x \) in \( P \)
    • Case 1: \( x \) does not appear in \( P \) at all – we can move \( t \) such that \( t-n+i \) is not covered by \( P \) anymore

\[
\begin{align*}
T & \quad xabxfabzzabxzzbzzb & \quad T & \quad xabxfabzzabwzzbzzb \\
P & \quad abwxyabzz & \quad P & \quad abwxyabzz
\end{align*}
\]

What next?
Bad Character Rule 2

• Setting 2
  – We are at position t in T and compare right-to-left
  – Let i by the position of the first mismatch in P
  – Let x be the character at the corresponding pos (t-n+i) in T
  – Candidates for matching x in P
    • Case 1: x does not appear in P at all
    • Case 2: Let j be the right-most appearance of x in P and let j<i – we can move t such that j and i align
Bad Character Rule 3

• Setting 3
  – We are at position t in T and compare right-to-left
  – Let i by the position of the first mismatch in P
  – Let x be the character at the corresponding pos (t-n+i) in T
  – Candidates for matching x in P
    • Case 1: x does not appear in P at all
    • Case 2: Let j be the right-most appearance of x in P and let j<i
    • Case 3: As case 2, but j>i – we need some more knowledge

T xabxkabzzabwzzbzzb
P abzwyabzz
Preprocessing 1

• In case 3, there are some “x” right from position i
  – For small alphabets (DNA), this will almost always be the case
  – In human languages, this is mostly the case for vowels
  – Thus, case 3 is a usual one

• These are irrelevant – we need the right-most x left of i

• This can (and should!) be pre-computed
  – Build a two-dimensional array $A[|\Sigma|,|P|]$  
  – Run through P from left-to-right (pointer i)
  – If character c appears at position i, set all $A[c,j] := i$ for all $j >= i$
  – Needs time (complexity?), but negligible because
    • P is small; complexity independent from T

• Array: Constant lookup, needs some space (lists ...)
(Extended) Bad Character Rule

- EBCR: Shift t by $i-A[x,i]$ positions
- Simple and effective for larger alphabets
- For random strings over $\Sigma$, average shift-length is $|\Sigma|/2$
  - Thus, n# of comparisons down to $|P|*|T|*2/|\Sigma|$
- Worst-Case complexity does not change
  - Why?
(Extended) Bad Character Rule

- EBCR: Shift t by i-A[x,i] positions
- Simple and effective for larger alphabets
- For random strings over \( \Sigma \), average shift-length is \( |\Sigma|/2 \)
  - Thus, n# of comparisons down to \( |P|*|T|*2/|\Sigma| \)
- Worst-Case complexity does not change
  - Why?
Good-Suffix Rule

- **Recall**: If we reach a mismatch, we know
  - The character in T we just haven’t seen
  - The suffix in P we just have seen

- **Good suffix rule**
  - We have just seen some matches in P (S)
  - Where else does S appear in P?
  - If we know the right-most appearance $S'$ of S in P, we can immediately align $S'$ with the current match in T
  - If S does not appear once more in P, we can shift t by $|P|$
Good-Suffix Rule – One Improvement

- Actually, we can do a little better
- Not all $S'$ are of interest to us
Good-Suffix Rule – One Improvement

- Actually, we can do a little better
- Not all $S'$ are of interest to us

We only need $S'$ whose next character to the left is not $y$
- Why don't we directly require that this character is $x$?
  - Of course, this could be used for further optimization
Concluding Remarks

- **Preprocessing 2**
  - For the GSR, we need to find all occurrences of all suffixes of P in P
  - This can be solved using our naïve algorithm for each suffix
  - Or, more complicated, in linear time (not this lecture)

- **WC complexity of Boyer-Moore is still O(|P|*|T|)**
  - But average case is sub-linear
  - WC complexity can be reduced to linear (not this lecture)
Example

```
bbcggbcbaaaggbbaacabaaabgbbaacgcabaaabcab

cabaabgbba
```

```
bbcggbcbaaaggbbaacabaaabgbbaacgcabaaabcab

EBCR wins
cabaabgbba
```

```
bbcggbcbaaaggbbaacabaaabgbbaacgcabaaabcab

GSR wins
cabaabgbba
```

```
bbcggbcbaaaggbbaacabaaabgbbaacgcabaaabcab

GSR wins
cabaabgbba
```

- **Match**
- **Good suffix**
- **Mismatch**
- **Ext. Bad character**

GSR wins
Content of this Lecture

- Searching strings
- Naïve exact string matching
- 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 EBCR
    • 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)