Information Retrieval

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
- Indexing: Preprocess docs and use index for searching
  - Apply tokenization; can only find entire words
  - Classical IR technique (inverted files)
- Online searching: Consider docs and query as new
  - No preprocessing - slower
  - Usually without tokenization – more "searchable" substrings
  - Classical algorithmic problem: Substring search

Properties

- Advantages of substring search
  - Does not require (erroneous, ad-hoc) tokenization
    - "U.S.", "35,00=.000", "alpha-type1 AML-3' protein", …
  - Search across tokens / sentences / paragraphs
    - ", that ", "happen. ", …
  - Searching prefixes, infixes, suffixes, stems
    - "compar", "ver" (German), …
- Searching substrings is "harder" than searching terms
  - Number of unique terms doesn't increase much with corpus size (from a certain point on)
    - English: ~ 1 Million terms, but 200 Million potential substrings of size 6
  - Need to index all possible substrings

Types of Substring Searching

- Exact search: Find all exact occurrences of a pattern (substring) p in D
- RegExp matching: Find all matches of a regular exp. p in D
- Approximate search: Find all substrings in D that are "similar" to a pattern p
  - Phonetically similar (Soundex)
  - Only one typo away (keyboard errors)
  - Strings that can be produced from p by at most n operations of type "insert a letter", "delete a letter", "change a letter"
  - …
- Multiple strings: Searching >1 strings at once in D

Strings

- Definition
  A *String S is a sequential list of symbols from a finite alphabet* Σ
  - *|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≠0 and i≠|S|*

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
  - Where is “GATATC”

```
tcagcttactaattaaaaattctttctagtaagtgctaagatcaagaaaataaattaaaaataatggaacatggcacattttcctaaactcttcacagattgctaatga
ttattaattaaagaataaatgttataatttttatggtaacggaatttcctaaaatattaattcaagcaccatggaatgcaaataagaaggactctgttaattggtact
attcaactcaatgcaagtggaactaagttggtattaatactctttttacatatatatgtagttattttaggaagcgaaggacaatttcatctgctaataaagggattac
atatttatttttgtgaatataaaaaatagaaagtatgttatcagattaaacttttgagaaaggtaagtatgaagtaaagctgtatactccagcaataagttcaaatagg c
gaaaaacttttaataacaaagttaaataatcattttgggaattgaaatgtcaaagataattacttcacgataagtagttgaagatagtttaaatttttctttttgtatt
acttcaatgaaggtaacgcaacaagattagagtatatatggccaataaggtttgctgtaggaaaattattctaaggagatacgcgagagggcttctcaaatttattcaga
gatggatgtttttagatggtggtttaagaaaagcagtattaaatccagcaaaactagaccttaggtttattaaagcgaggcaataagttaattggaattgtaaaagatat
ctaattcttcttcatttgttggaggaaaactagttaacttcttaccccatgcagggccatagggtcgaatacgatctgtcactaagcaaaggaaaatgtgagtgtagact
ttaaaccattttattaatgactttagagaatcatgcatttgatgttactttcttaacaatgtgaacatatttatgcgattaagatgagttatgaaaaaggcgaatatat
tattcagttacatagagattatagctggtctattcttagttataggacttttgacaagatagcttagaaaataagattatagagcttaataaaagagaacttcttggaat
tagctgcctttggtgcagctgtaatggctattggtatggctccagcttactggttaggttttaatagaaaaattccccatgattgctaattatatctatcctattgagaa
caacgtgcgaagatgagtggcaaattggttcattattaactgctggtgctatagtagttatccttagaaagatatataaatctgataaagcaaaatcctggggaaatat
tgctaactggtgctggtagggtttggggattggattatttcctctacaagaaatttggtgtttactgatatccttataaataatagagaaaaaattaataaagatgatat
```

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]

```
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 while;
        if (match) then
                -> OUTPUT t
end for;
```

Examples

Typical case

```
T   ctgagatcgcgta
P   gagatc
     gagatc
      gagatc
       gagatc
        gagatc
         gatatc
          gatatc
           gatatc
```

Worst case

```
T   aaaaaaaaaaaaa
P   aaaaat
     aaaaat
      aaaaat
       aaaaat
            ...
```

- How many comparisons do we need in worst case?
  - Always: t runs through T
  - Worst-case: 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|)
  - For some, WC=AC, but empirical performance differs much
  - Real performance depends much on size of alphabet and composition of strings (algs have their strength in certain settings)
  - Better performance possible if T is indexed (up to O(|P|))
- In practice, our naïve algorithm is quite competitive for non-trivial alphabets and biased letter 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

Content of this Lecture

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

Boyer-Moore Algorithm

R.S. Boyer /J.S. Moore. „A Fast String Searching Algorithm“, CACM, 1977

- Main idea
  - As for the naïve alg, we use two counters (inner loop, outer loop)
  - Outer loop runs from left-to-right
  - Inner loop runs from right-to-left

```
T    xabxfabzzabxzzbzzb
P    abwxyabzz
```

  - If we reach a mismatch, we know
    - The mismatch: Character in T we just haven't seen –
      - This is captured by the bad character rule
    - Match so-far: 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

```
        x   t
T   xabxfabzzabxzzbzzb        T   xabxfabzzabwzzbzzb
P   abwxyabzz                 P        abwxyabzz
        <----                             <----
        i
```

What next?

Bad Character Rule 2

- Setting 2
  - We are at position t in T and compare right-to-left
  - Let i be 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

```
T   xabxkabzzabwzzbzzb
P        abzwyabzz
            ↑  ↑
            j  i
```

```
T   xabxkabzzabwzzbzzb
P           abzwyabzz
                 ←
```

What next?

Bad Character Rule 3

- Setting 3
  - We are at position t in T and compare right-to-left
  - Let i be 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 often the case (e.g. for vowels)
  - Thus, case 3 is a usual one
- These “X” are irrelevant – we need the right-most x left of i
- This can (and should!) be pre-computed
  - Build a two-dimensional array A[|Σ|,|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
  - Requested time (complexity?) negligible
    - Because |P|<<|T| and complexity independent from T
- Array: Constant lookup, needs some space (lists …)

(Extended) Bad Character Rule

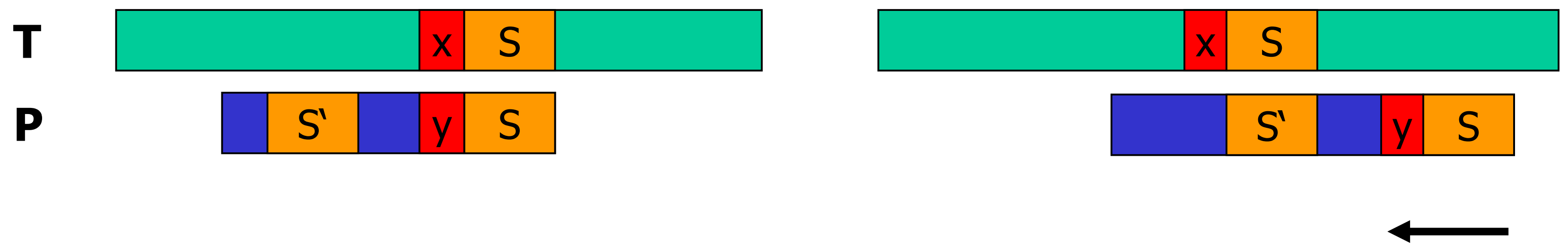
- EBCR: Shift t by i-A[x,i] positions
- Simple and effective for larger alphabets
- For random strings over Σ, average shift-length is |Σ|/2
  - Thus, n# of comparisons down to |T|*2/|Σ|
- Worst-Case complexity does not change
  - Why?

(Extended) Bad Character Rule

- EBCR: Shift t by i-A[x,i] positions
- Simple and effective for larger alphabets
- For random strings over Σ, average shift-length is |Σ|/2
  - Thus, n# of comparisons down to |P|*|T|*2/|Σ|
- 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 a suffix S from P in T
  - Where else does S appear in P?
  - If we know the right-most appearance S' of S in P with S'≠S, we can immediately align S' with the current match in T
  - If S' does not exist, we can shift t by |P|

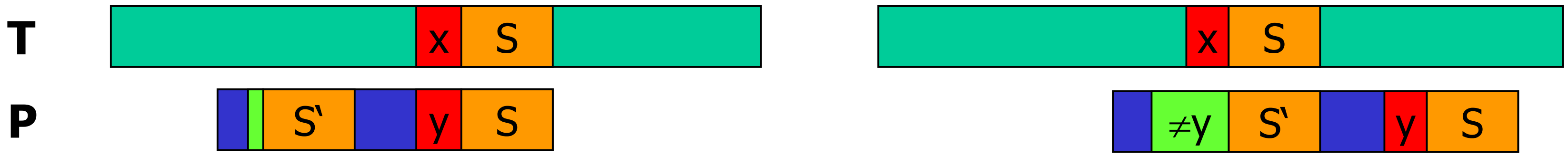


Good-Suffix Rule – One Improvement

- 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), but this usually doesn't pay-off on real data

Example

b b c g g b c b a a g g b b a a c a b a a b g b a a c g c a b a a b c a b

c a b a a b g b a a

b b c g g b c b a a g g b b a a c a b a a b g b a a c g c a b a a b c a b

EBCR wins

c a b a a b g b a a

b b c g g b c b a a g g b b a a c a b a a b g b a a c g c a b a a b c a b

GSR wins

c a b a a b g b a a

b b c g g b c b a a g g b b a a c a b a a b g b a a c g c a b a a b c a b

GSR wins

c a b a a b g b a a

b b c g g b c b a a g g b b a a c a b a a b g b a a c g c a b a a b c a b

c a b a a b g b a a

Match

Mismatch

Good suffix

Ext. Bad character

Content of this Lecture

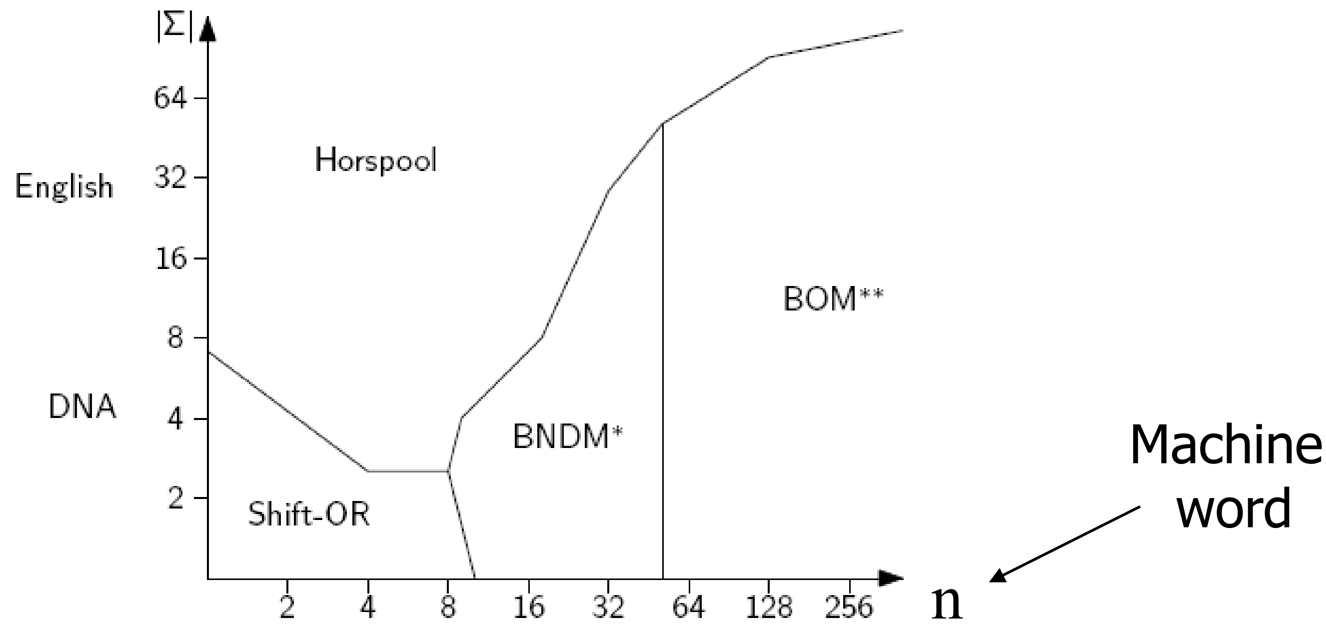
- Searching strings
- Naïve exact string matching
- Boyer-Moore
- BM-Variants and comparisons

Two Faster Variants

- BM-Horspool
  - Drop good suffix rule – often makes algorithm slower in practice
    - Rarely generates shifts longer than EBCR
    - Always 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
- Alternative: Always look at the least frequent (in the language of T) symbol of P first

Empirical Comparison

(Gonzalo Navarro & Mathieu Raffinot, 2002)



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

Self Assessment

- Explain the Boyer-Moore algorithm
- Which rule is better – GSR or EBCR?
- How can we efficiently implement EBCR?
- How does the Sunday algorithm deviate from BM?
- How can we use character frequencies to speed up BM? If we do so - which part of the algorithm is sped up?