Text Algorithms (6EAP)

Approximate Matching

Jaak Vilo
2019 fall

Problem

• Given P and S – find all approximate occurrences of P in S

P
S

Problem statement

• Let S=s_1,s_2,...,s_n ∈ Σ^* be a text and P=p_1,p_2,...,p_m the pattern. Let k be a pregiven constant.

• Main problems
  • k mismatches
    — Find from S all substrings X such that D(X,P) < k
  • k differences
    — Find from S all substrings X where D(X,P) ≤ k
  • best match
    — Find from S such substrings X that minimize D(X,P)

• Distance D can be defined using one of the ways from previous chapters

Exact vs approximate search

• In exact search we searched for a string or set of strings in a long text
• The we learned how to measure the similarity between sequences
• There are plenty of applications that require approximate search
• Approximate matching, i.e. find those regions in a long text that are similar to the query string
• E.g. to find substrings of S that have edit distance < k to query string m.

Reviews

• Algorithms

Multiple approximate matching

• Kimmo Fredriksson - publications [http://www.cs.uku.fi/~fredriks/publications.html]

Applications


• Tools
  • Webglimpse - glimpse, agrep
  • agrep for Win/DOS
  • Original agrep

• Links
  • Pattern Matching Pointers [Stefano Lonardi]
  • Articles

Tools

• Webglimpse - glimpse, agrep
• agrep for Win/DOS
• Original agrep

Links

• Pattern Matching Pointers [Stefano Lonardi]
Measure edit distance  
Find approximate occurrences

**Algorithm for approximate search, $k$ edit operations**

Input: P, S, k  
Output: Approximate occurrences of P in S (with edit distance ≤ k)

for $j=0$ to $m$ do  
    $h_{0,j}$ = $j$  
// Initialize first column

for $i=1$ to $n$ do  
    $h_{0,i}$ = 0

for $j=1$ to $m$ do  
    $h_{j,i}$ = min(  
        $h_{i-1,j-1}$ + (if $p_j=s_i$ then 0 else 1),  
        $h_{i-1,j}$ + 1,  
        $h_{i,j-1}$ + 1  
    )

if $h_{m,i}$ ≤ $k$ Report match at $i$
Trace back and report the minimizing path (from-to)

**Example**

```
abracadabra
0 0 0
r 1 1 0
a 2 1
d 3 2
a 4 3
```

- **Theorem**  
  Lets assume that in the matrix $h_{ij}$ the path that leads to the value $h_{mj}$ in the last row starts from square $h_{0r}$. Then the edit distance $D(P, s_{ts+1}...s_j)$= $h_{mj}$, and $h_{mj}$ is the minimal such distance for any substring starting before $j$th position, $h_{mj}$=$\min\{ D(P, s_{ts+1}...s_j) \mid t \leq j \}$

- **Proof by induction**
  - Every minimizing path starts from some value in the row 0
  - Since it is possible to reach the same result via multiple paths, then the approximate match is not always unique
• Time and space complexity $O(mn)$
• As $n$ can be large, it is sufficient to keep the last $m+k$ columns only, which can fully fit the full optimal path.
• Space complexity $O(m^2)$
• Or, one can keep just the single last column and in case of a match to recalculate the exact path.
• Space complecity $O(m)$
• If no need to find the path then $O(m)$

• Diagonal lemma will hold
• If one needs to find only the regions with at most $k$ edit operations, then one can restrict the depth of the calculations

It suffices to compute until $k$-border
• Modified algorithm (home assignment) will work in average time $O(kn)$
• There are better methods which work in $O(kn)$ at the worst case.

Improved average case

```
```

```
1. // Preprocessing
2. for j=0..m
do
3.   C[j]=j
4.   last = k+1
5. // Searching
6. for i=0..n
do
7.   pC=0;  nC=0
8.   for j=1 .. last
do
9.     if S[i]==P[j]
10.        nC=pC
11.     else
12.        if pC   < nC
13.           nC = pC
14.        if C[j] < nC
15.           nC = C[j]
16.     nC = nC+1
17.     pC = C[j]
18.     C[j] = nC
19. while C[last] > k
do
20.       last = last - 1
21.   if last < 0
22.      report match at position i
```

Ukkonen 1985; $O(kn)$
**Four Russians technique**

- This is a general technique that can be applied in different contexts
- It improves the speed of matrix multiplications
- Has been used for regular expression and approximate matching
- Let the column vector $d^j = (d_0^j, \ldots, d_m^j)$ present the current state
- Let's preprocess the automaton from each state
- $F(X, a) = Y$, s.t. column vector $X$ after reading character $a$ becomes column vector $Y$.
- **Example:** Let's find $P=abc$ approximate matches when there is at most 1 operation allowed.

**Four Russians version**

![Diagram](image)

Fig. 6.5. On the left is the full DFA, where each column is a state. On the right is the Four Russians version, where each region of a column is a state. The arrows show dependencies between consecutive regions.

**NFA/DFA**

- Create an automaton for matching a word approximately
- Allow $0, 1, \ldots, n$ errors
11.12.19

Regular expressions

Filtering techniques

- q-gram (also k-mer, oligomer)
- (sub)string of length q
- Let's have a pattern P of length m
- Assume pattern P is rather long and k is small, find occurrences with at most k mismatches
- How long substrings of P must have an exact match?
- If mismatches are most evenly, then we get ~ m/k pieces

Filtering techniques with q-grams

- If P has k mismatches, then S must have at least one substring of P whose length is at least \( \lceil \frac{m-k}{k} \rceil \)
- Filter for all possible q-mers where q is carefully selected.
  - Be careful with overlapping and non-overlapping q-grams.
  - If non-overlapping, then how long exact matches can we find?
- Use multiple exact matching O(n) (or sublinear) algorithms
- When an exact match of such substring is found, there is a possibility for an approximate overall match.
- Check for the actual match

K mismatches

- K = 3
- For 3-mismatch match, at least one substring of length \( \frac{m-3}{4} \) must occur exactly.
Filter and verify!

- P

Filtering techniques cont.

- Lots of research on approximate matching using q-gram techniques
- Lots of times reinvented the wheel in different fields

Indexing using q-grams

- Filtering can also be used for indexing. E.g. index all q-grams and their matches in S.
- If one searches for P, first search for q-grams in index. If a sufficient nr of matches is found, then make the comparison to see if the match is real.
- Filtering should be efficient for cases where a high similarity match for a long pattern is looked for.
- This is like reverse index for texts:
  - word doc_id:word_id doc_id:pos_id
  - word1 1:5 9:7 17:987 ...
  - word2 2:5 3:67 8:10 67:3 ...
  - word3 3:5 5:67 7:10 16:3 ...
  - ...
- Q: where do the word1 and word3 occur together?

Bit parallel search

- Can we use bit-parallelism for approximate search?

Generalized patterns

- A generalized pattern P=p_1,p_2,...,p_m consists of generalized characters p_i such that each p_i represents a non-empty subset of alphabet E^*;
  - p_i = a, a ∈ E^*
  - p_i = #, "wildcard" (any nr any symbols)
  - p_i = [group]; e.g.: [abc], [*abc], [a-h], ...
  - p_i = ¬C; Characters from a set I-C.
- Example: [Tt][aeiou][kpt][*aeiou][mnr] matches Tekstialgoritm but not word tekstuur.
- Problem: Search for generalized patterns from text
- Compare to SHIFT-OR algorithm!
\[ P = a \{b-h\} a \neg a \quad // \quad agrep \quad a \{b-h\} a[\neg a] \]

\[ \text{paganamaa} \]

\[ a \quad 110101 \]
\[ \{b-h\} \quad 221011 \]
\[ a \quad 332101 \]
\[ \neg a \quad 433210 \]

zero at last row – exact match!

- What about mismatches?
  - Mismatch if character does not belong to class defined by pattern. Unit cost 1.
  - SHIFT-ADD - similar to SHIFT-OR, but instead of OR an ADD is used. (no insertions deletions on this example)

- (no insertions deletions on this example)
  \[ P = a \{kpt\} a \neg a \quad // \quad agrep \quad a \{kpt\} a[\neg a] \]

\[ \text{paganamaa} \]
\[ \begin{align*}
0 & 0 0 0 0 0 0 0 0 0 \\
a & 1 1 0 1 0 1 \\
\{kpt\} & 2 2 1 1 1 1 \\
a & 3 3 2 2 1 3 \\
\neg a & 4 3 3 2 2 1 
\end{align*} \]

1 at last pos - match with 1 mismatch!

- Each value of matrix \( d_{ij} \) can be presented with \( b \) bits (4 bits allows values up to 16). Columns can be simple integers.
  - \( B_j = d_{mj} 2^{b(m-1)} + d_{m-1,j} 2^{b(m-1)} + ... + d_{1,j} \). (\( d_{0j} \) is always 0, can be omitted)
  - When adding another integer, where 0 is on position \( i \) if the next char at \( j \)’th position belongs to a set represented by \( P \), and 1 otherwise.

- When adding another integer, where 0 is on position \( i \) if the next char at \( j \)’th position belongs to a set represented by \( P \), and 1 otherwise.

\[ \begin{align*}
010 & 001 000 001 011 \\
+ & 001 001 000 000 001 \\
= & 011 010 000 001 100 
\end{align*} \]

- One needs to be very careful not to have overflow (111 + 001 = 1000).
  - Shift by 3 positions == multiply by 8

\[ \begin{align*}
010 & 001 000 001 011 \\
\times 8 & = 001 000 001 111 000 
\end{align*} \]
Use multiple vectors, one for each k value

- One can also use several individual 1-bit vectors, each corresponds to different k
- Can be extended to mask out regions where mismatches are NOT allowed
- Can introduce wildcards of arbitrary length

Bit-parallelism

- Maintain a list of possible “states”
- Update lists using bit-level operations

Example

(note: least significant bit is left in this output)

<table>
<thead>
<tr>
<th>Pattern = ACR[T&lt;GA&gt;T][G]A</th>
<th>length 7, # = .*</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV[0]</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>A 65</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>C 67</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>G 71</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>T 84</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>WILDCARD</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>ENDMASK</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>NO_ERROR</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

0 – position is “active”
- R[0] – vector for (so far) 0 mismatches
- R[1] – vector for (so far) 1 mismatch
- R[2] – vector for (so far) 2 mismatches

“Minimum” by bitwise AND
- If (even) one of the vectors has 0, then bitwise AND produces 0 (which is smaller of 0 and 1, 1 and 0, 0 and 0)
- If both (or all) of the vectors have 1, then bitwise AND produces 1 (which is smaller of 1 and 1)

The algorithm

- R[i] in general is the minimum of 3 possibilities:

```
(P[i] shift 1) bitor CV[ textchar ] &     // match
(P[i] bitor WILDCARD ) &                  // wildcard
(P[i-1] shift 1 bitor NO_ERROR)           // mismatch
```

Last – Add one mismatch unless errors not allowed

- How to get new values from old ones
- P[0] P[1] ... => R[0] R[1] ... R[0]
  – is min of three possibilities:

<table>
<thead>
<tr>
<th>{ P[i] shift 1 } bitor CV[ textchar ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>// previously active, now match with character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>{ P[i] bitor WILDCARD }</th>
</tr>
</thead>
<tbody>
<tr>
<td>// wildcard match – the same position remains active</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>{ P[i-1] shift 1 bitor NO_ERROR}</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Previously 1 less errors (unless NO_ERROR allowed)</td>
</tr>
</tbody>
</table>
public static void BPR(string pattern, string text, int errors)
{
    // Initialize NFA states
    for (int i = 0; i < pattern.Length; i++)
        states[i] = new int
    for (int j = 0; j < text.Length; j++)
        B[j] = new int
    // Initialize all characters positions
    for (int i = 0; i < pattern.Length; i++)
        for (int j = 0; j < text.Length; j++)
            states[i] |= 1 << (j - pattern.Length + i)
    // Compute NFA
    for (int i = 0; i < pattern.Length; i++)
        for (int j = 0; j < text.Length; j++)
            B[j] |= (1 << (j - pattern.Length + i)) | states[i] & B[j]
    // Compute final answer
    int result = 0;
    for (int i = 0; i < pattern.Length; i++)
        result |= (1 << (pattern.Length - i - 1)) & states[i]
    return result;
}

The Wu-Manber Algorithm of Text Searching Allowing Errors (1992)

Complexity: O(mn)
Constraint: m ≤ w
Proposed by Wu and Manber, 1992 [Wu and Manber, 1992], implemented in the agrep software

Agrep examples (from man agrep)

- agrep -f FILEINFILE OUT
  Gives the number of errors for that contain BASELINE within two errors.
- agrep -f FILEINFILE OUT
  Outlines the list of words containing BASELINE, within arbitrary distance, by T0, with up to one additional insertion (I) or deletion (D) and substitutions (as "expensive").
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
- agrep -f FILEINFILE OUT
  Outputs the list of all words containing at least the first 2 letters of the alphabet in order.
Multiple approximate string matching

- How to find simultaneously the approximate matches for a set of words, e.g. a dictionary.
- Or a set of regular expressions, generalized patterns, etc.
- One can build automatons for sets of words, and then match the automatons approximately.
- Filtering approaches – if close enough, test
- Not many (good) methods have been proposed

- Overimpose NFA automata
- Filter on all (necessary) factors