Approximate Matching

Jaak Vilo
2017 fall

Problem

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

Problem statement

• Let $S=s_1s_2...s_n \subseteq \Sigma^*$ be a text and $P=p_1p_2...p_m$ the pattern. Let $k$ be a pre-given constant.
• Main problems
• $k$ mismatches
  -- Find from $S$ all substrings $X$, $|X|=|P|$, that differ from $P$ at max $k$ positions (Hamming distance)
• $k$ differences
  -- Find from $S$ all substrings $X$, where $D(X,P) \leq k$ (Edit distance)
• best match
  -- Find from $S$ such substrings $X$, that $D(X,P)$ is minimal
• Distance $D$ can be defined using one of the ways from previous chapters
Algorithm for approximate search, \( k \) edit operations

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

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
  \( h_{j,i} = \min( h_{i-1,j-1} + (\text{if } p_j = s_i \text{ then } 0 \text{ else } 1), h_{i-1,j} + 1, h_{i,j-1} + 1 ) \)
if \( h_{m,i} \leq k \) Report match at \( i \)
Trace back and report the minimizing path (from-to)

Example

\[
\begin{array}{c|ccccccc}
 & a & b & r & a & c & a & d & a & b & r & a \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
r & 1 & 1 & 0 & 0 & 1 & 3 & 2 & 3 & 4 & 5 & 6 \\
a & 2 & 1 & 0 & 1 & 2 & 4 & 5 & 6 & 7 & 8 & 9 \\
d & 3 & 2 & 1 & 2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
a & 4 & 3 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

- **Theorem** Let assume that in the matrix \( h \) the path that leads to the value \( h_{m,j} \) in the last row starts from square \( h_{0,i} \). Then the edit distance \( D(P, s_{j-1}...s_j) = h_{m,j} \) and \( h_{m,j} \) is the minimal such distance for any substring starting before \( j'th \) position,

  \[
  h_{m,j} = \min\{ D(P, s_t...s_j) \mid t \leq j \}
  \]

- **Proof by induction**
  - Every minimizing path starts from some value in the row 0
  - Since it is possible to reach to 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 complexity O(m)
• If no need to find the path m 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) atr the worst case.

Improved average case


1. // Preprocessing
2. for j=0..m do C[j]=j //last active row
3. // Searching
4. for i=0..n do
5. pC=0 //previous and new column value
6. nC=0 //new column value
7. if S[i]==P[j] then nC=pC //why?
8. else
9. if pC < nC then nC=pC
10. if C[j]<nC then nC=C[j]
11. pC=nC
12. C[j]=nC
13. while C[lact] > k do
14. lact=lact-1
15. if lact=m then report match at position i
16. else last=last+1
17. end while

Ukkonen 1985; O(kn)

Fig. 6.3 An O(kn) expected time dynamic programming algorithm. Note that it works with just one columns vector.
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 $d*j = (d0j, ..., dmj)$ 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

![](image)

**NFA/DFA**

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

Four Russians technique

- There are 13 different possibilities:

| 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 0 0 0 0 0 1 1 1 1 1 1 1 1 |
| 0 0 1 1 1 0 1 1 1 2 2 2 |
| 0 1 0 1 2 1 0 1 2 1 2 3 |

- From each state compute possible next states for all characters $a, b, c$, and $x$ (not in $P$).
- The states with $d_{mj} \leq 1$ are final states.
- This can become too large to handle.
- Cut the regions into smaller pieces, use that to reduce the complexity.

Navarro and Raffinot Flexible Pattern Matching in Strings. (Cambridge University Press, 2002), pp. 152 Fig 6.5.

12.7. The Four-Russians speedup

In this section we will discuss an approach that leads both to a theoretical and to a practical speedup of many dynamic programming algorithms. The idea, comes from a paper (28) by four authors, A. Asvani, O. Sevc, K. Krestel, and F. Raffinot, concerning boolean matrix multiplication. The general idea taken from this paper has come to be known in the West as the four-Russian technique even though only one of the authors is Russian. The applications in the string domain are quite different from matrix multiplication, but the general idea suggested in (28) applies. We illustrate the idea with the specific problem of computing (unweighted) edit distance. This application was first worked out by Maier and Pervin (313) and was further discussed by those authors in (313); many additional applications of the Four-Russian idea have been developed since then (for example 246).

Fig. 6.15. Approximate matching

Fig. 6.18. Converting the edit-distance problem into a shortest path problem. Bold arrows show the optimum path, of cost 4.
Regular expressions

- q-gram (also k-mer, oligomer)
- (sub)string of length q
- Lets 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

- For 3-mismatch match, at least one substring of length (m-3)/4 must occur exactly.

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
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 7:9 167:987 ...
  - word2 2:5 3:67 8:10 673 ...
  - word3 3:5 6:7 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_1p_2...p_n consists of generalized characters p_i such that each p_i represents a non-empty subset of alphabet Σ^*;
- p_i = a, a ∈ Σ
- p_i = #, "wildcard" (any nr any symbols)
- p_i = [group]; e.g.: [abc], [^abc], [a-h], ...
- p_i = ¬C; Characters from a set Σ-C.
- Example: [T][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¬a // agrep a[b-h]a[^a]

paganamaa

a 110101
[b-h] 221011
a 332101
¬a 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¬a // agrep a[kpt]a[^a]

<table>
<thead>
<tr>
<th></th>
<th>paganamaa</th>
<th>0000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>110101</td>
<td></td>
</tr>
<tr>
<td>[kpt]</td>
<td>221121</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>332213</td>
<td></td>
</tr>
<tr>
<td>¬a</td>
<td>433221</td>
<td></td>
</tr>
</tbody>
</table>

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.
• Bj = d_{m-1,j} + d_{m-1,j} 2^{b(m-1)} + ... d_{1,j}. (d_{0,j} 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_i 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_i and 1 otherwise.

<table>
<thead>
<tr>
<th></th>
<th>010 001 000 001 011</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>001 001 000 000 001</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>011 010 000 001 100</td>
</tr>
</tbody>
</table>

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

010 001 000 001 011 * 8
= 001 000 001 011 000
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)

```
Pattern = ACIT<GA>-[TG]A length 7, # = .*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>WILDCARD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ENDMASK</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NO_ERROR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NO_ERROR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>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:
  ```c
  ( 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
  ```
The Wu-Manber Algorithm of Text Searching Allowing Errors (1992)

\[
\begin{align*}
&\text{Complexity: } O(ec) \\
&\text{Constraint: } m \leq c \leq n \\
&\text{Proposed by Wu and Manber, 1992} \\
&\text{Implemented in the agrep software} \\
&\text{Allows 3 types of errors: substitution, insertion, deletion} \\
&\text{Called Bit-Parallelism Row-wise (BPR), in Navarro and Raffinot, 2002} \\
\end{align*}
\]

Agrep examples (from man agrep)

\* \text{agrep} \text{2} \text{-} \text{aDkFgy} \text{b} \\
\* \text{agrep} \text{2} \text{-} \text{aDkFgy} \text{b} \text{-} \text{w} \\
\* \text{agrep} \text{-} \text{b} \text{-} \text{w} \\
\* \text{agrep} \text{-} \text{b} \text{-} \text{w} \text{-} \text{r} \\
\* \text{agrep} \text{-} \text{b} \text{c} \text{d} \text{e} \text{f} \text{g} \text{h} \text{i} \\
\* \text{agrep} \text{-} \text{b} \text{c} \text{d} \text{e} \text{f} \text{g} \text{h} \text{i} \text{-} \text{w} \\
\* \text{agrep} \text{-} \text{b} \text{c} \text{d} \text{e} \text{f} \text{g} \text{h} \text{i} \text{-} \text{w} \text{-} \text{r} \\
\* \text{agrep} \text{-} \text{b} \text{-} \text{w} \text{-} \text{r} \\
\* \text{agrep} \text{-} \text{w} \\
\* \text{agrep} \text{w} \\

\begin{align*}
\text{match} & : \{ (0, \text{match}) \} \\
\text{insertion} & : \{ (1, \text{insertion}) \} \\
\text{deletion} & : \{ (2, \text{deletion}) \} \\
\text{error} & : \{ (3, \text{error}) \} \\
\end{align*}

\[\text{Complexity: } O(ec)\]

\[\text{Constraint: } m \leq c \leq n\]


\text{psuedo code}\]

```c
for (i = 0; i < S.length(); i++)
    if (S[i] == T[j])
        for (j = 1; j < T.length(); j++)
            if (S[i] == T[j])
                Continue;
            else
                continue;
    if (T[j] == S[i])
        for (j = 1; j < T.length(); j++)
            if (T[j] == S[i])
                Continue;
            else
                continue;
    if (T[j] == S[i])
        for (j = 1; j < T.length(); j++)
            if (T[j] == S[i])
                Continue;
            else
                continue;

for (i = 0; i < S.length(); i++)
    for (j = 0; j < T.length(); j++)
        if (S[i] == T[j])
            for (j = 1; j < T.length(); j++)
                if (S[i] == T[j])
                    Continue;
                else
                    continue;
        if (T[j] == S[i])
            for (j = 1; j < T.length(); j++)
                if (T[j] == S[i])
                    Continue;
                else
                    continue;
        if (T[j] == S[i])
            for (j = 1; j < T.length(); j++)
                if (T[j] == S[i])
                    Continue;
                else
                    continue;
```

% End for
% End of for

\[\text{agrep}\]

\begin{itemize}
\item \text{Insertions, deletions}
\item \text{Wildcards}
\item \text{Non-uniform costs for substitution, insertion, deletion}
\item \text{Find best match}
\item \text{Mask regions for no errors}
\item \text{Record orientated, not line orientated}
\end{itemize}
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