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.

Problem

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

<table>
<thead>
<tr>
<th>Problem statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let S=s_1s_2...s_n ∈ Σ* be a text and P=p_1p_2...p_m the pattern. Let k be a pregiven constant.</td>
</tr>
<tr>
<td><strong>Main problems</strong></td>
</tr>
<tr>
<td><strong>k mismatches</strong></td>
</tr>
<tr>
<td>- Find from S all substrings X,</td>
</tr>
<tr>
<td><strong>k differences</strong></td>
</tr>
<tr>
<td>- Find from S all substrings X, where D(X,P) ≤ k (Edit distance)</td>
</tr>
<tr>
<td><strong>best match</strong></td>
</tr>
<tr>
<td>- Find from S such substrings X, that D(X,P) is minimal</td>
</tr>
<tr>
<td><strong>Distance D can be defined using one of the ways from previous chapters</strong></td>
</tr>
</tbody>
</table>
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 \( h_{m,i} \leq k \) Report match at \( i \)
Trace back and report the minimizing path (from-to)

Example

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>r</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>b</th>
<th>r</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>r</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Theorem

Let's 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 \( D(P, s_{s+1}s_{s+2}...s_j) = h_{mj} \) 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 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 at 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 $O(kn)$ worst case.

Improved average case


Ukkonen 1985; $O(kn)$

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

Fig 6.3. An $O(kn)$ expected time dynamic programming algorithm. Note that it works with just one column 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 $d^j = (d_0^j, ..., d_m^j)$ present the current state
- Lets 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**

- There are 13 different possibilities:
  
  $\begin{align*}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 1
  \end{align*}$

  - From each state compute possible next states for all characters $a, b, c$, and $x$ ($x$ 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.

**NFA/DFA**

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

- 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

- If P has k mismatches, then S must have at least one substring of length (m-k)/k
- 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
- P
- For 3-mismatch match, at least one substring of length (m-3)/4 must occur exactly.

Filtering techniques with q-grams
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 10:6 9:7 ...
  • word2 2:5 3:6 7:10 6:7 3 ...
  • word3 3:5 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_m 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]

    paganamaa
    0 0 0 0 0 0 0 0 0 0
    a 1 1 0 1 0 1
    [kpt] 2 2 1 1 2 1
    a 3 3 2 2 1 3
    ¬a 4 3 3 2 2 1

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.
  • $B_j = d_{m_1}2^{b(m-1)} + d_{m-1,j}2^{b(m-1)} + ... + d_{1,j}$. ($d_{0,j}$ 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 $P$, 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.

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

<table>
<thead>
<tr>
<th>Pattern = AC\text{GT}&lt;\text{GA}&gt;[\text{TG}]A</th>
<th>length 7, \ # = +</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV[\text{char}]</td>
<td>0101110</td>
</tr>
<tr>
<td>A 65</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>C 67</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>G 71</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>T 84</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>WILDCARD</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>ENDMASK</td>
<td>11111111111111111111111111111111</td>
</tr>
<tr>
<td>NO_ERROR</td>
<td>00000000000000000000000000000000</td>
</tr>
<tr>
<td>\text{ENDMASK}</td>
<td>00000000000000000000000000010000</td>
</tr>
<tr>
<td>\text{NO_ERROR}</td>
<td>00000000000000000000000000000000</td>
</tr>
</tbody>
</table>

- 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:
  
  $\begin{align*}
  & (P[i] \text{ shift 1 } \text{ bitor } CV[\text{ textchar }] ) \text{ & } // \text{ match} \\
  & (P[i] \text{ bitor WILDCARD } ) \text{ & } // \text{ wildcard} \\
  & (P[i-1] \text{ shift 1 bitor NO_ERROR}) // \text{ mismatch}
  \end{align*}$

- 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:
  
    \[
    \begin{align*}
    & \{ P[i] \text{ shift 1 } \text{ bitor } CV[\text{ textchar }] \} \\
    & \text{ // previously active, now match with character} \\
    & \{ P[i] \text{ bitor WILDCARD } \} \\
    & \text{ // wildcard match – the same position remains active} \\
    & \{ P[i-1] \text{ shift 1 bitor NO_ERROR} \} \\
    & \text{ // Previously 1 less errors (unless NO_ERROR allowed)}
    \end{align*}
    \]}
1. If newR < 10
2. oldR < 10.
4. Searching
5. for i ∈ 0...k Do Ri ← Om-lii
6. for pos ∈ 1...n Do
7. oldR ∈ 0
8. newR < 0
9. oldR ∈ newR
10. for i ∈ 1...k Do
11. newR < ((newR ∈ 1) & B[pos]) | oldR | ((oldR | newR) ∈ 1) end of for
12. if newR & 10 Do newR ← 0 Then report an occurrence at pos
13. End of for

Writing of an overview, implementing the algorithm and creating a useful tool could be a big topic for a BSc or MSc thesis. However, the main focus should be on developing a preprocessing algorithm using the BPR (p = p1p2...pm, T = t1t2...tn, k) technique, which is a time-saving approach to approximate string matching based on dynamic programming. This algorithm is particularly useful when k/m is sufficiently small, as it allows for rapid computation of blocks of the dynamic programming matrix for the problem at hand.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The paper introduces an algorithm of comparable simplicity that requires only O(nm/w) time by virtue of computing a bit vector algorithm for approximate string matching based on dynamic programming. This algorithm requires only O(nm/w) time, which is significantly faster than the previous algorithms. Moreover, because the algorithm is not dependent on k, it can be used to rapidly compute blocks of the dynamic programming matrix for the problem. Here we present an algorithm of comparable simplicity that requires only O(nm/w) time by virtue of computing a bit vector algorithm for approximate string matching based on dynamic programming.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k errors. There has been a significant amount of work on this problem, and many algorithms have been designed to solve it. One such algorithm is the BPR algorithm, which is based on a vector approach and is known for its efficiency in computing the similarity automaton for the query and the text. The algorithm is designed to handle large texts and queries, and it relies on a novel technique called approximate string matching based on dynamic programming (BPR). This approach allows for rapid computation of blocks of the dynamic programming matrix, making it particularly useful for large-scale applications.
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