Text Analytics
Searching Multiple Terms

Ulf Leser
Types of String Searching

- **Exact or approximate**
  - Exact search: Find all occurrences of $k$ in $D$
  - Pattern matching: Given a regular expression, find all matches 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
    - Strings that can be produced from $k$ by at most $n$ operations of type “insert a letter”, “delete a letter”, “change a letter”
    - ...

- **Word or substring**
  - Searching words ($k$ is a word): After tokenization
  - Searching substrings: Across token/sentence... boundaries

- **Searching one or multiple strings at once** in $D$
Naive Substring Searching

1. Align \( P \) and \( T \) at position 1
2. Compare symbols from \( P \) with symbols from \( T \) from left to right
   - If symbols are not equal: goto 3
   - Otherwise
     - All symbols of \( P \) have been compared: Shout “here”, goto 3
     - Otherwise: compare next symbol from \( P \) and \( T \), goto 2
3. Move \( P \) one position to the right, goto 2
4. If position of \( P \) in \( T \) < \(|T| - |P| + 1\), goto 2
5. Stop
Preprocessing

• Definition
  – Let $i > 1$. Then we define $Z_i(S)$ to be the length of the longest substring $x$ of $S$ with
    • $x = S[i..i+|x|-1]$ (x starts at Position $i$ in $S$)
    • $S[i..i+|x|-1] = S[1..|x|]$ (x is a prefix of $S$)
  – Then, we call $x$ the Z-Box of $S$ at position $I$ and length $Z_i(S)$
Z-Algorithm, Case 1

- Case 1: \( k > r \)
  - Thus, no previously computed Z-box contains \( k \)
  - Thus, we have never before looked further than \( k \)
  - Then there isn’t much we can do
    - Compute \( Z_k \) naively, symbol by symbol
    - If \( Z_k > 0 \), set \( r = r_k \) and \( l = l_k \)

**Example 1**

<table>
<thead>
<tr>
<th></th>
<th>CTCGAGTTGCAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>

**Example 2**

<table>
<thead>
<tr>
<th></th>
<th>CTACTACTTTGCAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
</tr>
</tbody>
</table>
Z-Algorithm, Case 2.1

- Two sub-cases. **First sub-case:** \( Z_{k'} < |\beta| = r-k+1 \)
  - This implies that the symbol at position \( k' + Z_{k'} \) does not match with the symbol at position \( Z_{k'} \). However, this implies that \( S[k+Z_{k'}] \) will produce the same mismatch.
  - Thus: \( Z_k = Z_{k'} \); keep \( r \) and \( l \)
Z-Algorithm, Case 2.2

- **Second sub-case:** $Z_{k'} \geq |\beta|$
  - This implies that $\beta$ is a prefix of $S$ (but not necessarily the longest)
  - We do know that, if $Z_{k'} > |\beta|$, then $S[|\beta|+1] = S[k'+|\beta|]$
  - But we don’t know anything about $S[r+1]$
    - We have never positively matched this symbol
  - Procedure
    - Match naively $S[r+1..]$ with $S[|\beta|+1..]$
    - Let the first mismatch occur at position $q$
    - Let $Z_k = q-k; r = q-1; \text{if } q \neq r+1: l = k$
Complexity

• Theorem

*The Z-Box algorithm computes all Z-boxes in $O(|S|)$*

• Proof
  – We estimate two counts: $m=\text{"number of positive matches"}$ and $m'=\text{"number of mismatches"}$
  – We show that $m'<|S|$
    • At $k=2$, we can produce at most one mismatch
    • Case 1: Maximally one
    • Case 2.1: No comparisons at all, hence no mismatches
    • Case 2.2: Maximally one
  – Thus, there is at most one mismatch for every position of $S$
  – This shows that $m' \leq |S|$
Boyer-Moore Algorithm

- Basic idea
  - Align strings as in the naive algorithm
  - Compare symbols from right to left, starting with the last character of P
  - Outer loop: Try to shift P more than one position
  - Inner loop: Try to start comparisons of symbols at a position left from the end of P
  - Use two independent tricks: bad character rule and good suffix rule
- Especially the outer-loop optimization yields large average case improvements, especially if $|\Sigma|$ is large (like for English text)
- BM is sublinear in average case
  - Our presentation has quadratic worst case
  - Improvements to linear worst case exist
Bad Character Rule 2

- **Observation**
  - Assume we aligned $P[n]$ with $T[j]$ for some $j \geq n$
  - Let $i$ be the position in $P$ of the first (right-most) mismatch
  - Let $x$ by the symbol at position $j-n+i$ in $T$
  - Let $l$ be the right-most occurrence of $x$ in $P$
  - Where may $x$ match in $P$?
    - Case 1: $x$ doesn’t occur at all in $P$. Shift by $i$ positions.
    - Case 2: $l<i$. Shift by $i-l$ positions

---

**Diagram:**

- $T$: xabxkabzzabwzzbzzb
- $P$: abzwyabzz

- How far can we shift now?
Good-Suffix Rule

• Idea
  – When we have a mismatch, we usually had some matches before
  – Can we know where this match occurs in P (apart from the suffix)?
  – To know this, we need to preprocess P

- We may always shift to the right-most occurrence of the substring t in P (which is not a suffix)
- If this doesn’t exist, we may shift P by |P|-|t| positions
Case 1

- Let $i$ be the position of the first mismatch in $P$, let $t=P[n-i+1,..]$
- Let $k$ be the right end of the right-most occurrence of $t$ in $P$ with $k<n$ and $P(k-|t|) \neq P(n-|t|)$ ('y')
  - If no such occurrence exists, let $k=0$
- If $k \neq 0$: Shift $P$ by $n-k$ positions

- Why don’t we demand that $P(k-|t|)=x'$?
Empirical Comparison

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

- Keyword-Trees
- Failure Links
- Constructing failure links
- Repairing the algorithm: Output links
Searching Multiple Strings

- Often, we need to search for more than one string
  - Search all gene names from a dictionary in a given text
  - We want to search for the entire dictionary at once
- Let $P=\{P_1, P_2, \ldots, P_z\}$, $n=|P_1|+|P_2|+\ldots+|P_z|$
- First attempt
  - Z-box requires $O(m+|P_i|)$ for searching for pattern $P_i$
  - Naïve extension to $z$ patterns requires $O(z*m+n)$
    - There is nothing we can save
- We shall improve this to $O(m+n+k)$
  - For same special definition of $k$ - later
Idea

• Usually, patterns share substrings
• We are especially interested in **shared prefixes** (because we shall compare from left to right)
• We need to find a data structure to represent common prefixes
• Using the structure, we want to search for all patterns concurrently
Keyword Trees

- **Definition**
  
  Let $P$ be a set of patterns. The **Keyword Tree** for $P$ is a tree with
  - Every edge is labels with exactly one symbol
  - If a node has more than one child, the edges labels are all different
  - A node $k$ represents a pattern $P_i$ iff $\text{label}(k)=P_i$. It must hold that
    - Every leaf represents exactly one $P_i$
    - Every $P_i$ is represented by exactly one node (inner node or leaf)

- **Remark**
  
  - The label of a node is the concatenation of the edges labels on the path from root to this node
  - Recall that in a tree the path from root to every node is unique
  - If a node $k$ represents a pattern $P_i$, we say that $k$ is marked with $i$, i.e., $\text{mark}(k)=i$
Example

- $P = \{\text{banane, bohne, bohnern, wohnen, bohren}\}$
Constructing a Keyword Tree

• Complexity?

• Construction is $O(n)$
  – Start with $P_1$
  – Constructing the “tree” needs $O(|P_1|)$
  – Take $P_2$. Traverse the prefix of $P_2$ in the tree until
    • ... there is a mismatch at position $i$ in $P_2$. Insert a fork and create the branch for the rest of $P_2$ in $O(|P_2| - i)$
    • ... $P_2$ was matched completely
    • This needs $O(|P_2|)$ in either case
  – Repeat for $P_3$ – $P_z$

• Since paths are unique, there is no backtracking etc.
Naive Usage – A First Attempt

- Given set $P$ of Patterns and Template $T$
- Build the Keyword Tree $K$ for $P$ in $O(n)$
- Run $i$ through the positions in $T$
  - Traverse the prefix of $T[i..]$ in $K$
  - When passing by a marked node, report the mark
  - If we cannot match further, restart with $i := i+1$ at the root of $K$
- **Complexity:** $O(n+m^*n_{\text{max}})$, with $n_{\text{max}} = \max(|P_i|)$
  - Maybe faster than our naïve approach (if $n_{\text{max}} < z$)
  - Maybe not
  - Problem: We are matching symbols in $T$ more than once
Content of this Lecture

- Keyword-Trees
- Failure Links
- Constructing failure links
- Repairing the algorithm: Output links
Failure Links

- **Definition**
  
  *Let $K$ be the Keyword Tree for a set $P$ of pattern. Let $k$ be a node of $K$*
  
  - *Let $\text{length}(k)$ be the length of the longest true suffix of $\text{label}(k)$ which is also a prefix of any pattern in $P$*
    - *If suffix with length $>0$ exists, set $\text{length}(k)=0$*
  
  - *Let $\text{fl}(k)$ denote the node with:*
    - $\text{label}(\text{fl}(k)) = \text{label}(k)[|\text{label}(k)|-\text{length}(k)+1 .. |\text{label}(k)|]$*
    - *If $\text{length}(k)=0$, set $\text{fl}(k)=\text{root}$*

- **Remarks**
  
  - The link $(k, \text{fl}(k))$ is called the *Failure Link* for $k$
  
  - $\text{label}(\text{fl}(k))$ exactly is the „longest true suffix“ of $\text{label}(k)$
  
  - $\text{fl}(k)$ must be unique
Example

$P=\{\text{banane, nabe, abnahme, na, abgabe}\}$
Example

$P = \{\text{banane, nabe, abnahme, na, abgabe}\}$

FLs to root are not shown
Example

P={banane, bohne, bohnern, wohnen, bohren}

• All Failure Links point to root
  – Letters b und w are nowhere in a pattern at a position ≠1
  – Thus, no true suffix can be also be a prefix
Searching with Failure Links

• Assume we search at position \( j \) in \( T \)
• We match substring \( T[j..] \) in \( K \)
  – If there is a match
    • Traverse down that match and set \( j++ \)
    • If the reached node is marked, report the mark as match
  – If there is a mismatch at position \( x \) in \( T \)
    • Let \( k \) be the last match node
      – All children of \( k \) are mismatches for \( T[j+x] \)
    • Follow the failure link of \( k \) to node \( fl(k) \)
      – We have just seen \( label(fl(k)) \) in \( T \)
    • Continue matching at position \( j+x \) in \( T \) and node \( fl(k) \) in \( K \)
  – If we reach a leaf \( k \) at position \( j+x-1 \) in \( T \)
    • Report the mark of the leaf
    • Follow the failure link to node \( fl(k) \)
    • Continue matching at position \( j+x \) in \( T \) and node \( fl(k) \) in \( K \)
Example

\[ P = \{ \text{banane, nabe, abnahme, na, abgabe} \} \]

\[ T = \text{radnaben} \]
Example

\[ P = \{ \text{banane, nabe, abnahme, na, abgabe} \} \]

\[ T = \text{abnabeln} \]
Algorithm

j := 1; // Next comparison in T
l := 1; // Start of pattern in T
k := root(K); // Current node in keyword tree
while (j<|T|)
    while exists edge (k,k') with label T(j)
        if mark(k')≠NULL then
            report mark(k') with start l;
        end if;
        k := k'; // Down the tree
        j := j+1; // Check next character
    end while;
    if k=root(K) then // Immediate mismatch: move on in T
        j := j+1;
        l := l+1;
    else
        k := f1(k); // Follow the failure link
        l := j-len(k);
    end if;
end;

• Complexity: O(m)
But ...

P = \{knabt, nabe, na\}

T = knabenschaft

- Algorithm matches KNAB in T
- B is the last matching symbol, failure link to NAB
- Proceed to NABE, report P₂
- Follow fl to root and match on in T with NSCHAFT
- **We missed P₃ (NA)!**
  - Why?: P₂ contains P₃
  - But hold on a second
Content of this Lecture

- Keyword-Trees
- Failure Links
- Constructing failure links
- Repairing the algorithm: Output links
Failure Link Construction

• Definition
  – \textit{Let \textit{depth}(k) be the length of label (k)}

• We proceed as follows
  – Build the keyword tree in linear time
  – Then construct all failure links in linear time \(O(n)\)
  – Note that failure links always point to true suffixes
  – This implies that for all \(k\): \textit{depth}(k) > \textit{depth}(\textit{fl}(k))

• We construct failure links using a \textit{breadth-first traversal} of the keyword tree
Observation

- Let’s fix a node $k$
- We may reach all prefixes of any pattern which are identical to a suffix of $\text{label}(k)$ by following failure links
  - The longest such prefix is $\text{fl}(k)$
  - The others are $\text{fl}(\text{fl}(k)), \text{fl}(\text{fl}(\ldots))$

$$P = \{\text{knabe, nabe, abe, bele}\}$$
Algorithm

- **Start:** For every node $k$ with $\text{depth}(k)=1$ let $\text{fl}(k)=\text{root}(K)$
- **Induction from $i-1$ zu $i$**
  - We assume that all $\text{fl}$ from nodes $l$ with $\text{depth}(l)<i$ are known
  - $\forall k\in K$ with $\text{depth}(k)=i$
    - Let $k'$ be the father node of $k$ and let $x$ be the label on the edge from $k'$ to $k$
    - Every suffix of $\text{label}(k')$, extended by $x$, is a suffix of $\text{label}(k)$
    - Every prefix which are identical to a suffix of $\text{label}(k')$ are reached by **traversing failure links** from $k'$
      - Including the longest such prefix
    - We are only interested in those prefixes which can be extended by $x$
      - This also holds if we reach root
More formally

- **Induction from i-1 to i**
  - We assume that all fl from nodes l with depth(l)<i are known
  - \( \forall k \in K \) with depth(k)=i
    - Let \( k' \) be the father node of k and let x be the label on the edge from \( k' \) to k
    - Follow the failure link from \( k' \) to fl(\( k' \))=v
    - If there exists an edge \((v,v')\) with label x, then set fl(k):=v'
    - Otherwise, if v=root(K), then set fl(k):=root
    - Otherwise, follow failure link from v to v'' and check if there is an edge from v'' labeled x ... (recursion)
Example

\[ P = \{ \text{knabe, nabr, abt, beil} \} \]
Example

\[ P = \{\text{knabe, nabr, abt, beil}\} \]
Algorithm

// We search failure link for k, depth(k)>1
// Let k’ be the father of k, label(k’,k)=x
v := fl(k’);
while (v≠root(K)) and (not exists edge (v,v’) with label(v,v’)=x)
    v = fl(v);          // Follow failure link
end while;
if (v=root(K)) then
    if (exists edge (v,v’) with label(v,v’)=x)
        fl(k) = v’;
    else
        fl(k) = root(K);
else
    fl(k) = v’;        // Continuation of prefix with x

• Complexity is O(n), proof omitted
Content of this Lecture

- Keyword-Trees
- Failure Links
- Constructing failure links
- Repairing the algorithm: Output links
Our Problematic Case

- **Patterns containing other patterns**
- **Solution**: We construct another set of pointers called **Output Links**
- **Observation**
  - Let $P_1$ be contained in $P_2$
  - Then $P_1$ must be the suffix of a prefix $P_2[1..i]$ for some $i \geq |P_1|$
  - If $P_1$ is the longest prefix (n P) of $P_2[1..i]$, then $fl(P_2[i]) = P_1$
    - Which doesn’t help – usually, we will not follow this link during search
  - If this is not the case, there must exist a $P'$ with
    - $P'$ is the longest suffix of $P_2[1..i]$
    - Thus, $fl(P_2[i]) = P'$
    - Again: $P_1$ is suffix of $P'$ – but is it the longest?
      - Search recursive using failure links
      - Eventually, we must reach $P_1$
Example

\[ P = \{\text{knabe, na}\} \]

\[ P = \{\text{eknabe, na, kna}\} \]
Induction

• Starting from a node k ...
  – Following failure links
  – Reaching a marked node k’
  – Then the pattern mark(k’) is contained in T

• The reverse is also true: All patterns contained in T are found through paths of failure links

• Of course, we don’t want to follow all such paths during online search

• We need some preprocessing
Output Links

- **Definition**
  The *Output Link of node* $k$, $out(k)$, *points to the node* $k'$ *with*
  - $k'$ is marked
  - $k'$ is the first marked node on the path from $k$ following failure links

- **Remark**
  - Not all nodes have output links
  - Output links always point to shorter pattern

- **We construct output links in constant time during the breadth-first traversal of the keyword tree for computing the failure links**
Failure Links and Output Links

```plaintext
// We search failure link for k, depth(k)>1
// Let k' be the father of k, label(k',k)=x
v := fl(k');
while (v ≠ root(K)) and (not exists edge (v,v') with label(v,v')=x)
  v = fl(v);            // Follow failure link
end while;
if (v=root(K)) then
  if (exists edge (v,v') with label(v,v')=x)
    fl(k) = v';
    if mark(v') ≠ NULL then out(k) = v'; else out(k) = NULL;
  else
    fl(k) = root(K);
    out(k) = NULL;
else
  fl(k) = v';          // Continuation of prefix with x
  if mark(v') ≠ NULL then
    out(k) := v';      // Obviously the closest marked node
  else
    out(k) = out(v');
end if;
end if;
```
Construction of Output Links

$P = \{\text{knarzt, arzth, zt, zta}\}$
Searching with Output Links

- Whenever we pass a node \( k \), we must follow its output link (if it exists at all) in a little detour
- From the target node (which must be marked), again we must follow its output link and so forth
Example

1. Algorithm matches KNA ...
   • Following the output link leads to reporting $P_3$
2. ... matches further KNAB
   • Following the output link leads to reporting $P_4$
3. „b“ is the last match - failure link zu NAB
4. Further matching to NABE – Report $P_2$
Complete Search Algorithm

\[ j := 1; \quad /\text{ Next comparison in } T \]
\[ k := \text{root}(K); \quad /\text{ Root node of keyword tree} \]
\[ \text{while } (j < |T|) \]
\[ \quad \text{while exists edge } (k,k') \text{ with label}(k,k')=T(j) \]
\[ \quad \quad \text{if mark}(k') \neq \text{NULL} \text{ then} \]
\[ \quad \quad \quad \text{report mark}(k'); \]
\[ \quad \quad \end{if}; \]
\[ \quad z = \text{out}(k'); \]
\[ \quad \text{while } (z \neq \text{NULL}) \quad /\text{ Check output links} \]
\[ \quad \quad \text{report mark}(z); \quad /\text{ Found a match} \]
\[ \quad \quad z = \text{out}(z); \quad /\text{ Recursion} \]
\[ \quad \end{if}; \]
\[ \quad k := k'; \quad /\text{ Down the tree} \]
\[ \quad j := j+1; \quad /\text{ Check next character} \]
\[ \end{while}; \]
\[ \text{if } k=\text{root}(K) \text{ then} \quad /\text{ Mismatch: move on in } T \]
\[ \quad j := j+1; \]
\[ \text{else} \]
\[ \quad k := \text{fl}(k); \quad /\text{ Follow the failure link} \]
\[ \end{if}; \]
\[ \end{end}; \]
Complexity

• During search
  – Let \( k \) by the number of matches of all patterns
  – The inner WHILE-loop is passed at most \( k \) times
  – Thus: \( O(m+k) \)

• Overall complexity
  – Build the keyword tree for \( P \) \( O(n) \) (trivial)
  – Compute failure links \( O(n) \) (BF)
    • This includes the output links
  – Search \( O(m+k) \)

• Total: \( O(n+m+k) \)