Text Analytics

Searching with Keyword-Trees

Ulf Leser
Content of this Lecture

- **Searching Multiple Pattern: Keyword-Trees**
  - Failure Links
  - Output links
- **Approximate Search with PETER**
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 \( P = \{P_1, P_2, ..., P_z\} \), \( n = |P_1| + |P_2| + ... + |P_z| \)
- 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 $P$ by a set of patterns. The *Keyword Tree* (or *trie*) for $P$ is a tree with
  
  - Every edge is labeled 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**
  
  - 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: \(O(n+m\times n_{\text{max}})\), 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

• Searching Multiple Pattern: Keyword-Trees
  – Failure Links
  – Output links

• Approximate Search with PETER
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:
  \[
  \text{label}(\text{fl}(k)) = \text{label}(k)[|\text{label}(k)|-\text{length}(k)+1 \ldots |\text{label}(k)|]
  \]
  - If $\text{length}(k)=0$, set $\text{fl}(k)=\text{root}$

• Remarks

- The link $(k, \text{fl}(k))$ is called the Failure Link for $k$
- $\text{label}(\text{fl}(k))$ 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

$P = \{ \text{banane, bohne, bohnern, wohnen, bohren} \}$

- All Failure Links point to root
  - Letters b und w are nowhere in a pattern at a position $\neq 1$
  - 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

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

- **Complexity**: \(O(m)\)
But ...

\[ P = \{ \text{knabt, nabe, na} \} \]

\[ T = \text{knabenschaft} \]

- Algorithm matches KNAB in T
- B is the last matching symbol, failure link to NAB
- Proceed to NABE, report \( P_2 \)
- Follow fl to root and match on in T with NSCHAFT
- **We missed \( P_3 \) (NA)!**
  - Why?: \( P_2 \) contains \( P_3 \)
  - But hold on a second
Content of this Lecture

- Searching Multiple Pattern: Keyword-Trees
  - Failure Links
  - Output links
- Approximate Search with PETER
Our Problematic Case

- Patterns containing other patterns
- 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 (in $P$) 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 $\text{mark}(k')$ 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

• Definition: The Output Link of node $k$, $\text{out}(k)$, points to the node $k'$ with
  – $k'$ is marked
  – $k'$ is the first marked node on the path from $k$ following failure links
Complete Search Algorithm

\[
j := 1; \quad \text{// Next comparison in } T \\
k := \text{root}(K); \quad \text{// Root node of keyword tree} \\
\text{while } (j < |T|) \\
\hspace{1em} \text{while exists edge } (k,k') \text{ with label}(k,k')=T(j) \\
\hspace{2em} \text{if mark}(k') \neq \text{NULL then} \\
\hspace{3em} \text{report mark}(k'); \\
\hspace{2em} \text{end if}; \\
\hspace{2em} z = \text{out}(k'); \\
\hspace{1em} \text{while } (z \neq \text{NULL}) \quad \text{// Check output links} \\
\hspace{2em} \text{report mark}(z); \quad \text{// Found a match} \\
\hspace{2em} z = \text{out}(z); \quad \text{// Recursion} \\
\hspace{1em} \text{end if}; \\
\hspace{1em} k := k'; \quad \text{// Down the tree} \\
\hspace{1em} j := j+1; \quad \text{// Check next character} \\
\hspace{1em} \text{end while;} \\
\hspace{1em} \text{if } k=\text{root}(K) \text{ then} \quad \text{// Mismatch: move on in } T \\
\hspace{2em} j := j+1; \\
\hspace{1em} \text{else} \\
\hspace{2em} k := fl(k); \quad \text{// Follow the failure link} \\
\hspace{1em} \text{end if;} \\
\hspace{1em} \text{end;}
\]
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
    - Algorithm not covered here
  - Search \( O(m+k) \)

- **Total: \( O(n+m+k) \)**
Content of this Lecture

- Searching Multiple Pattern: Keyword-Trees
- **Approximate Search with PETER**
  - Prefix trees for string searching
  - Pruning the search space
  - Evaluation
Approximate Search

- Often, we want to find all occurrences of a term with small deviations
  - E.g., searching with typos or morphological variations
- In the following, we assume tokenization – we are looking for similar terms, not arbitrary substrings
- **Approximate search**
  - Given a doc d converted into a list K of terms and a query q: Find all terms from K that are similar to q
- **Approximate join**
  - Given a doc d converted into a list K of terms K and a list Q of search terms: Find all terms from K that are similar to at least one term from Q
PETER

- PETER: Prefix tree based indexing algorithm
  - Keyword tree = prefix tree

- Exact and similarity (threshold), search and join
- Similarity measures: Hamming and Edit distance
- Uses many tricks for speeding-up searches
- Real software (standalone, library, plug-in for RDBMS)
Searching Prefix-Trees

- To search all approximate occurrence of k in D, we index all words of D in a prefix tree T.

- Search
  - Recursively match k with a path starting from root.
  - If no further match: k $\not\in T$.
  - If matched completely: k $\in T$.

- Search complexity
  - Only depends on depth of T.
  - Independent from |D|.
Compressed Prefix Trees (Patricia Trees)

- Slightly more complex implementation
  - Different kinds of edges/nodes
Large Prefix Trees

- Tree of common prefixes is kept in **main memory**
  - Many (failed) searches never access disc
  - At most one disc access per search

- Unique suffixes are stored (sorted) on disk
  - Minus a short preview-prefix (as fitting into a node structure)
Similarity Search on Prefix-Trees

- In similarity search, a mismatch doesn’t mean that $k \notin T$
- **Several mismatches** are allowed
  - Depending on query threshold

- **Idea**
  - Depth-first search on the tree as usual
  - Keep a **counter for the n# of mismatches** spent in the prefix so far
  - If counter exceeds threshold – stop search in this branch
  - Apply tricks to stop although n# of mismatches not yet too big
Example: Search

Hamming distance search for $k = \text{CTGAAATTGGT}$, $e=1$
Example: Search

Hamming distance search for $t = \text{CTGAAATTGGT}$, $k=1$
Example: Search

Hamming distance search for \( t = \text{CTGAAATTGGT}, k=1 \)
Example: Search

Hamming distance search for $t = \text{CTGAAATTGGT}$, $k=1$

$d(\text{CTGAAATT}, \text{CTGAAATT}) > 1$

Ulf Leser: Text Analytics, Winter Semester 2010/2011
Example: Search

Hamming distance search for $t = \text{CTGAAATTGGT}, k=1$
Example: Search

Hamming distance search for $t = \text{CTGAAATTTGGT}$, $k=1$
Searching with Edit Distance

- Slightly more complicated
- Requires computation of an edit-distance matrix
- Since a node represents a common prefix, all its successors share this part of the matrix
- During DFS, keep growing matrix
- Next letter: Add one col and/or row
- Use k-banded alignment based on n# of allowed mismatches e
(Similarity) Joins on Prefix Trees

• We compare growing prefixes with growing prefixes
• Essentially: Compute intersection of two trees
• Traverse both trees in parallel
  – Upon (sufficiently many) mismatches, entire subtrees are pruned
• Same idea for exact and similarity join
Content of this Lecture

- Searching Multiple Pattern: Keyword-Trees
- Approximate Search with PETER
  - Prefix trees for string searching
  - Pruning the search space
  - Evaluation
Pruning Search Space

• Can we do better?

• What information can we store inside a node that would allow us to stop a search early
  – Although n# of allowed mismatches not yet exceeded
  – We must be sure that finally there will be too many mismatches

• Trade-Off
  – “Information” must not be too big (memory issue)
  – Test must be fast
Filter by String Length

- Consider strings $s,t$ with $d(\text{pref}(s,k)),\text{pref}(t,l))=x$, search threshold $y$
  - Hamming: No match if: $|(|s|-k) - (|t|-l)| > y-x$
  - Edit: No match if: $|(|s|-k) - (|t|-l)| > y-x$
  - Example: Let $y=4$, Hamming

\[
\begin{align*}
\text{ACCTGTAGAATCTG} & \quad (17-6)-(14-6) > 4-2 \\
\text{ACTTGAXXXXXXXXXXX} & \quad (17-6)-(14-6) > 4-2
\end{align*}
\]

- Test inside prefix trees
  - Store MAX and MIN length of all strings below each node
  - Stop if all strings in this range must incur too many mismatches
  - Needs two bytes per node, very fast test
Filter by Character Frequency [Aghili et al., 2003]

- Example: y=3, Hamming

\[
\begin{align*}
\text{ACCTGTAGAATCTG} \\
\text{ACTTGATTTTGGGG}
\end{align*}
\]

- Count **frequencies of all letters** in the suffixes
  - At given position: \([3,1,2,2]\) and \([0,0,4,4]\)
  - Thus, we must perform \(-3*A, -1*C, +2*G, +2*T\)
  - This requires **at least 4 moves** - prune

- Similar argument for edit distance

- For prefix trees
  - Store **letter frequency ranges** of subtree in each node
  - Prune if threshold will be hurt for certain
  - Needs \(|\Sigma|\) bytes per node, very fast test
Filter by q-Gram [Xiao et al., VLDB, 2008]

- A similar lower bound can be derived from q-grams
- Idea (assume |s|=|t|)
  - Pad s and t with q-1 special characters (#) at start and end
  - Let qg() be the function computing the bag of q-grams for s
  - If s has one I/D/R compared to t, this change can destroy at most q q-grams
  - Thus, if |qg(s) \cap qg(t)| > |qg(s)|-q, then ed(s,t)=0

- General case:
  \[ ed(s,t) \leq k \iff |qg(s) \cap qg(t)| \geq \max(|s|, |t|)-1-(k-1)q \]
Implementation

- Makes only sense for edit distance
  - Computing hamming distance is faster than computing q-grams
- Not for free, but mostly cheaper than edit distance
  - K-banded edit distance: $k \times \max(|s|,|t|)$ operations
  - Computing the q-gram bags: $|s| + |t|$ operations
  - Comparing the q-gram bags: sort + merge (linear)

- For prefix trees
  - Storing all q-grams in every node requires too much space
  - Compute q-grams on-the-fly when entering a unique suffix
Content of this Lecture

• Searching Multiple Pattern: Keyword-Trees
• Approximate Search with PETER
  – Prefix trees for string searching
  – Pruning the search space
  – Evaluation
Configurations

- We tested each filter separately using various data sets, various thresholds, hamming and edit distance
- Best configuration: Length + q-gram
  - Q-gram filter very important on our data set
    - >90% of EST have long, unique suffixes
    - Most mismatches occur only in these suffixes
      - Using filter earlier makes little sense
    - Large benefits although q-grams not pre-computed
  - Frequency filter yields almost no improvement
    - DNA: Very small alphabet, high entropy
    - Decreases the sharpness of the lower bound
    - Filter rarely hits
Setup

- Data: Several EST data sets from dbEST
  - Search: All strings of one data set in another data set
  - Join: One data set with another one
  - Varying similarity thresholds
- (Linear) Index creation not included
Effect of Filter on Search / Join

- Length + q-gram works best
- Frequency filter has only negligible influence
  - On this alphabet
- Same for joins
Search: Comparing to GREP + Friends

- PETER an **order of magnitude faster** in all cases
- Comparison is a bit unfair
  - Agrep and nrgrep **do not build an index**
  - But: Indexing amortizes fast (e.g. searches 15 for agrep)
Search: Comparing to Flamingo

- **Flamingo**: Library for approximate string matching
  - Based on an inverted index on q-grams
  - Uses length and charsum filter
Joins

- We are not aware of any available software to perform similarity joins
  - UNIX join: Only exact (always faster if with sorting excluded)
  - Flamingo: Does not include approx. join of same authors
  - BLASTALL: based on local alignment

- Similarity joins in PETER
  - Again, length + q-gram works best
  - Join execution time grows exponentially with similarity threshold
  - Self-join on 10K strings in <1 minute
  - $T_1 \bowtie T_X (~1.500.000.000.000 pairs, hamming, k=3)$ in 13 hours
    - But: $T_1 \bowtie T_X (~1.500.000.000.000 pairs, edit, k=3)$ in 3 days
    - But: $T_1 \bowtie T_X (~1.500.000.000.000 pairs, edit, k=1)$ in 80 minutes
Similarity Search RDBMS

- Peter (behind extensible indexing interface) versus UDF implementing hamming / edit distance calculations
- Difference: 2-3 orders of magnitude, independent of data set, threshold, or search pattern length
(Similarity) Join inside RDBMS

- PETER (behind extensible indexing interface) versus build-in join (exact join, hash and merge) or UDF
- Similarity join
  - Join T3 with T2e, k=2, inside RDBMS: Stopped after 24 h
  - Same join with PETER: 1 minute
- Exact join
  - For long strings, PETER is significantly faster even when compared to commercial join implementations
Conclusions

• Fastest available library for approximate search and join
• Especially good for large databases and long strings
• Currently only available for DNA (4-letter alphabet)
• Available as plugin for RDBMS – much faster than build-in operations
• Index creation time is not negligible, but also not dramatic

• Current work: Move everything into main memory