Fast Approximate String Matching with Finite Automata

Mans Hulden
mhulden@email.arizona.edu
The University of Arizona
Helsinki Finite State Technology Research Group
University of Helsinki

The problem

The problem addressed here is a classic search problem: given a word \( w_{in} \), not found in a list \( W \), which word in \( W \) most closely resembles \( w_{in} \)?

\[
\begin{array}{c}
? \\
W_1 \\
W_2 \\
\vdots \\
W_n \\
W_{in}
\end{array}
\]

This can be costly using standard edit distance calculations for several reasons:

- If the size of the list \( W \) is large, we cannot practically calculate a ‘distance’ between every word on the list and our candidate word to find the solution
- The list \( W \) may be infinite: we may have a grammar that models unlimited compounding and affixing and thus allows infinitely long words
- We may want to use complex distance metrics to define the similarity between two words

An alternate formulation

Instead of considering a list \( W \) to find approximate matches for a word \( w_{in} \), we consider finding the words in a finite-state automaton \( A \) that most closely resemble \( w_{in} \).

Generalizing the problem has a number of potential advantages:

- Any finite list \( W \) can be converted into a deterministic finite state automaton
- A finite state automaton can encode an infinite number of words
- Morphological analyzers are often designed to be finite-state transducers (FSTs). It is trivial, given a morphological FST, to extract an automaton that encodes all the legal words in the language

Applying A*-search to the problem

We apply the classic A*-search algorithm to the problem. In effect, we match letters in \( w_{in} \) against arcs in the automaton \( A \) taking into account the possibility of insertion, deletion and substitution. For each step and node expansion in the search space we recalculate the score \( f = g + h \), where \( g \) is the accumulated cost so far, and \( h \) our heuristic guess of the future score. We maintain nodes in a priority queue and iteratively expand the one with the cheapest \( f \) and keep going until we find a solution.

Heuristics

The most important question when doing first-best/A*-type search strategies is the heuristic \( h \) used to decide the node expansion strategy. The requirements on \( h \) are basically:

- \( h \) must be consistent (never overestimates the remaining cost)
- \( h \) must be fast to calculate
- additional data needed to calculate \( h \) must take up little space

For this algorithm, several experiments with different heuristics \( h \) were made, and we settled for a strategy where:

- We precalculate for each state in the automaton \( A \), what symbols can possibly be encountered on future paths starting from that state
- The path length is variable from \( 1 \ldots \infty \)
- Whenever we need to calculate \( h \) in the search we compare the number of symbols different in the word remainder vs. the symbols stored in the state

Choosing a heuristic

Since several different \( h \) (varying with lookahead length) were available, we conducted experiments to find an overall reliable strategy. Most results were similar to that of table 1.

<table>
<thead>
<tr>
<th>( h )</th>
<th>( h_0 )</th>
<th>( h_1 )</th>
<th>( h_2 )</th>
<th>( h_3 )</th>
<th>( h_4 )</th>
<th>( h_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_0 )</td>
<td>( n=\infty ) (we only use the ( \infty ) lookahead)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_1 )</td>
<td>( n=2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_2 )</td>
<td>( n=3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_3 )</td>
<td>( n=4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_4 )</td>
<td>( n=\max(h_1, h_2) ) (also, ties in priority queue broken depending on value of ( pos ))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example & Results

Input word: dat

\[
\text{Average times for MED search for two algorithms}
\]

Comparison against Schulz & Mihov’s algorithm for 1,000 random words, 100 of each edit distance between 0 and 10; words taken from the Free Ling Spanish dictionary and randomly perturbed.

Conclusions

- A*-search works well for approximate string matching with the relatively simple heuristic presented here
- The algorithm has been implemented and is included in the freely available finite-state toolkit foma, found at http://foma.sf.net.
- Additional features that have been implemented (also in foma) include the possibility of specifying context-dependent confusion matrices to specify different costs for different types of substitutions, deletions and insertions, depending on the environment where they occur