DESIGN OF A HIGH SPEED STRING MATCHING CO-PROCESSOR FOR NLP

Presented by:
Guru Prasad(Y5187)
Hemraj Bairwa(Y5195)
Hitesh Khandelwal(Y5202)
Abstract: *In this project we try to introduce an efficient way for string matching. We use parallel processing for fast matching and have organized the lexicon in least space using manner (Here we assume that all length words are equally predominant in the input text). This idea is not limited to English, any language text can use it, as matching finally boils down to character matching.*

1. Introduction: In many real life application in which parsed input string needs further to be processed then parsing speed plays important role. Faster parsing is required especially when the output of a speech recognizer must further be processed to filter out syntactically incorrect sentences. Some applications like syntactic pattern recognition, syntactic analysis of programming languages, natural language processing etc., the parser speed is an important factor. Also advanced information retrieval, Text Mining and related areas require parsing to enhance performance by integrating syntactic knowledge about the language. An efficient low complexity parser will surely increase the efficiency since large amount of data need to be processed.

Since Speed of Co-Processor decides how fast input test can be parsed, so any improvement in the string-matching algorithm will manifest itself as a significant improvement in the overall performance of any NLP task where parsing is indispensable. So here we are proposing a fast string matching Co-Processor to deal with such problem which require a lot of data to be parsed and processed later.

We have used parallel processing and memory interleaving techniques to give a improved algorithm for string matching with time complexity $O(\log_2 n)$. Also we are saving a considerable amount of space in our method, with the proper organization of lexicon that avoids redundancy. So overall system which is suggested by author is intended to serve as an efficient NLP Co-Processor.

2. Matching Process: Matching can be classified into two types—one is *Perfect Matching (PM)* and another is *Approximate Matching (AM)*. PM is that in which input word completely matches with lexicon stored (*e.g* input word ‘play’ matches with ‘play’ stored in lexicon). Approximate Matching is required when PM fails to accept the input word. We have a concept of inflections here which can be derived from a root word (*for e.g. inflections of play are plays, played, playing etc.*). If input word is inflection of root word then AM succeeds (*e.g. ‘plays’ matches with inflection of ‘play’*).
Now let's describe how exactly we are going to match input word. For this we need some formalization.

Let $X = x_1 \ x_2 \ldots \ x_n$ be the input word, and $Y = y_1 \ y_2 \ldots \ y_n$ be the lexical entry. We define the following parameters required for matching $X$ and $Y$.

$$
VALUE[i] = \begin{cases} 
1 & \text{if } x_i = y_i \\
0 & \text{otherwise}
\end{cases}
$$

for $i = 1, 2, \ldots, n$

We take $X$ and $Y$ in two registers, input register and lexical entry register. $Y$ is loaded from the lexicon. $X$ and $Y$ are compared from left to right character wise. When size of words are less than size of register then remaining character position in register are filled with null character, say $\varphi$.

When two characters match, then we fill the bit corresponding to that character in the VALUE register as 1. If one of the character in register is $\varphi$ and another is character or vice-versa or both characters in both registers are $\varphi$ which are supposed to match then we fill the bit corresponding to that character in the VALUE register as 0.

The size of the VALUE register depends on the size of the lexical entry. If the lexical entry is 4 to 8 characters long, the VALUE register will be 8 bits long. Hence, if the matching is perfect, the value in the VALUE register will be 1111 1111 (FFh). This parameter is introduced in order to get the correct matching entry from the lexicon for the input word. An integer denoting the last $n$ characters of the input word, which fail to match with those of the lexical entry is called LEVEL. To explain the meaning of LEVEL, let’s consider length of input word to be $n$ and length of lexical entry to be $m$. Let there is no null character in any of the register and let $k+1$ be the first non-matching character such that $x_{k+1} \neq y_{k+1}$ then value of LEVEL will be equal to $m-k$ e.g LEVEL$(X,Y)=m-k$, where $X$ and $Y$ are same as defined above, while the substrings $x_{k+1}, x_{k+2}, x_{k+3}, \ldots x_m$ defined to be Last Character Information (LCI).

Here we define the Condition for Perfect Match (PM) and Approximate Match (AM)-

$$
\text{PM} \leftarrow (\text{VALUE}_C == \text{FF}_h)
\text{AM}
\leftarrow \sim\text{PM} \land (\text{VALUE}_L = \text{LEVEL}_L)^\land (\text{LCI}_L = \text{LCI}_C)
$$

where $\text{VALUE}_C$, $\text{LEVEL}_C$ and $\text{LCI}_C$ are the computed parameters using the input word and lexical entry, and $\text{VALUE}_L$, $\text{LEVEL}_L$ and $\text{LCI}_L$ are the parameters of the root words with respect to inflections stored in the lexicon.

Now here is time for an example –

Consider the input word playing and lexical entry play. $\text{VALUE}_C$ and $\text{LEVEL}_C$ are computed as illustrated bellow.

**INPUT WORD:** playing$\varphi$

**LEXICAL ENTRY:** play$\varphi\varphi\varphi$

$\text{VALUE}_C = (11110000)$ or $(\text{F0}_h)$

and here we have $m = 7$ and $k = 4$ so $\text{LEVEL}_C = m-k = 3$ and value of
$LCI_{C} = \text{ING}$. The LCI parameter is needed to resolve the ambiguity that may persist even after an approximate match. For example, the lexical entry turn will be:

Root Word: turn \ (wrt turns)

$VALUE_L$, $LEVEL_L$, $LCI_L$

F7  1  's'

Now suppose that input word is mistakenly given as turnt then the $VALUE$ and $LEVEL$ parameter will be same but what differ only is $LCI$ value. As $LCI_L$ value is ‘t’ which forces the Approximate Match (AM) to fail. So correct input word turns can match with the lexical entry.

### 3. Language Independent:

Easily it can be shown that this string matching algorithm is independent of language of our choice. Because the calculation of stored parameter $LEVEL_L$, $VALUE_L$, $ICL_L$ as well as the calculated parameter $LEVEL_C$, $VALUE_C$, $ICL_C$ all are calculated from at character level. So what we want to tell here is we are dealing here with characters not the words, so our logic of string matching and storing lexicon can be applied to any language. Here we would like to put forward a example from the some south Indian languages and English.

<table>
<thead>
<tr>
<th>Language</th>
<th>Root word</th>
<th>Inflection</th>
<th>$VALUE$ (hex)</th>
<th>$LEVEL$</th>
<th>$LCI$</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>sing</td>
<td>sang</td>
<td>BFF</td>
<td>3</td>
<td>ang</td>
</tr>
<tr>
<td>Telugu</td>
<td>paadu</td>
<td>paadenu</td>
<td>F1F</td>
<td>3</td>
<td>enu</td>
</tr>
<tr>
<td></td>
<td></td>
<td>paaduthu</td>
<td>F8F</td>
<td>3</td>
<td>thu</td>
</tr>
<tr>
<td>Malayalam</td>
<td>paatu</td>
<td>paati</td>
<td>F7F</td>
<td>1</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td></td>
<td>paadunnu</td>
<td>F8F</td>
<td>3</td>
<td>nnu</td>
</tr>
<tr>
<td>Tamil</td>
<td>paadu</td>
<td>paadinaar</td>
<td>F07</td>
<td>5</td>
<td>inaar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>paadukiraan</td>
<td>F81</td>
<td>6</td>
<td>kiraan</td>
</tr>
</tbody>
</table>

### 4. Matching Algorithm:

Here is the algorithm for matching the input string with lexical entry. Algorithm computes the following function:

$\text{MATCH}(w) = \begin{cases} 1 & \text{if any one of PM or AM is true} \\ 0 & \text{otherwise} \end{cases}$

**Match algorithm**
Input: A word $w$

Output: Match (True/False)

Steps:

- Compute the length of $w$ and save first character in `firstchar`.
- Select the section from memory based on the length computed.
- Pick $m$ words from the section and use $m$ matching units to match the input word. If PM then return true
- Else use $k$ AM units to match with inflections of root word and return true if it succeeds ($k$ is # of inflections)
- Else return false.

5. **Lexical Organization:** Here we can store lexical entry in three different manners

- **Firstly** Storing all inflections in the lexicon
- **Secondly** Storing only root words, and additional information on inflections, but without dividing lexicons into sections.
- **Thirdly** Same as case 2, but the lexicon is divided into sections according to the length of the root word (for e.g. Section1: 1-4, Section2: 5-8, etc).

The most optimal way of storing lexicon is **third** one. Because here each Lexical section $L_i$ will be first divided into subsections $L_{i1} L_{i2} L_{i3} .... L_{im}$. When an input word $w$ is to be inserted into the lexicon, the matching algorithm will first find the corresponding lexical section, depending on the number of characters of $w$ which can be done in $log_2 n$ time.

The lexicon is organized statically that is there is no change in the lexicon at the time of string matching. We organize lexicon as a 3rd case of lexicon arrangement. For this we first locate proper lexical section depending on length of entry to be made, which can only be root word. Now we need to insert this entry into proper subsection. For this purpose we maintain a tag which denotes the last subsection that was used for insertion.

6. **Implementation details:**

To begin with we design circuitry for computing the VALUE, LEVEL and LCI. Following diagram explains the implementation according to definition of VALUE.

```
  I1   L1       I2   L2                       I16  L16

  C1   C2   C16

  16-bit VALUE Register
```

In above diagram characters of input words upto 16 characters long are matched against lexical entry from lexicon using 16 8-bit comparators and final result is stored in a

<table>
<thead>
<tr>
<th>Root Word</th>
<th>Inflections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ask</td>
<td>Asks</td>
</tr>
<tr>
<td></td>
<td>Asked</td>
</tr>
<tr>
<td></td>
<td>Asking</td>
</tr>
</tbody>
</table>
16-bit VALUE register.

While matching above two words, We maintain a byte value corpus which keeps information about how much characters are matching in both words. To compute LEVEL we just subtract this value from numchar, which is total number of characters in the input word.

![Diagram of matching process](attachment:matching_diagram.png)

LCI is the remaining characters from currpos.

Now we describe the lexicon organization in memory. Available space is divided into 4 sections for respective words (only root words) of sizes ranging from 1-4, 5-8, 9-12, 13-16 characters. Now in each section all word are sorted lexicographically and stored in subsections in interleaved fashion. Information about starting and finishing of words with firstchar as ‘a’, ‘b’, …… ‘z’ is kept in astart, afinish and bstart, bfinish and so on respectively.

The table below indicates one section divided into 4 subsections (m = 4).

<table>
<thead>
<tr>
<th>Subsection</th>
<th>S00</th>
<th>S01</th>
<th>S02</th>
<th>S03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a…</td>
<td>a…</td>
<td>a…</td>
<td>a…</td>
</tr>
<tr>
<td>1</td>
<td>a…</td>
<td>a…</td>
<td>a…</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b…</td>
<td>b…</td>
<td>b…</td>
<td>b…</td>
</tr>
<tr>
<td>3</td>
<td>b…</td>
<td></td>
<td>b…</td>
<td></td>
</tr>
<tr>
<td>……</td>
<td>……</td>
<td>….</td>
<td>….</td>
<td>….</td>
</tr>
</tbody>
</table>

All inflections and other values like POS etc. and also the pre-computed VALUE, LEVEL, LCI for each inflection is stored in the different part of memory.

Diagram below shows complete implementation of a matching unit.

Here we are using 2X4 decoder to select appropriate comparator according to length of the input word. After selecting the comparator we select m words from the corresponding section and start matching input word with stored one if there is PM then we directly tell there is match else we go to AM units and select k parallel units and do the same as in case of PM. If there is match then we tell there is match else we tell there is no match found for input string.
Based on length of input word the decoder selects which comparator to use. Result of each comparator is wire-ORed, which provides information about success or failure of perfect match. In approximate match unit, we implement the logic described in section 2 for approximate marching. This black box for AM includes the logic circuitry for matching VALUE, LEVEL, LCI of input word and lexical entry. Final OR gate outputs 1 for successful match and 0 for failure.

Now with this implementation in hand we can match an input word in a section. If the section matching fails then we go to the lower section and perform the same matching again and if still it fails to match then output “input word fails to match”.

**Performance and Benefits:**

**In space:**

We store root word and its inflections separately (for e.g. ask is stored and ‘s’, ‘ed’, ‘ing’ are stored separately) instead of storing all the inflections in brute force manner.

We further divide the lexicon in sections purposefully. We have benefit of allocating
only 4 bytes to words of length range 1-4 and similarly. By doing so we save lot of space again.

For example in the above table we are using 4 sections. In S1 we are saving $\frac{3}{4}$ space as compared to Method 2 of storing lexicon, Similarly in S2 we are saving $\frac{1}{2}$ space , in S3 we are saving $\frac{1}{4}$ space thus overall we saved ($\frac{3}{4} + \frac{1}{2} + \frac{1}{4}$)/4 = 37.5% space as compared to the second case where there were no sections (Assuming almost equal distribution of lexical entries among all the sections). So it is clear that we are saving space while storing lexical entry.

**In performance:**

As now we can see from the implementation that we need not match whole lexicon entries for an input word, just match in one or two sections with parallel processing through using m matching units and we are done!

Further increase in number of matching units and increase in number of sections will improve the performance at least theoretically but practical implementation puts restriction on this increase.

Below table shows %space saving by increasing number of sections.

<table>
<thead>
<tr>
<th>Sections</th>
<th>1-4</th>
<th>5-8</th>
<th>9-12</th>
<th>13-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Different No. of Lexical Sections (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = 1</td>
</tr>
<tr>
<td>% Space saving</td>
</tr>
</tbody>
</table>