Jumble Solver

Michael Lehmann & John Fullard

CSE 30331 - Data Structures

Final Project Fall 2008
Abstract

You often see a word jumble in today's major newspapers. These jumbled words get more difficult as the week goes on and sometimes by Friday, the words are very tricky to solve. Our Jumble Solver allows you to type in the jumbled word and it will output the solution based on a 27,000 word dictionary. There are multiple different data structures you could use to store the dictionary but we ultimately decided on the way we found to be the most efficient. This data structure was used to store along with an algorithm to search through the tree.

Key Terms

Trie, Tree, Dictionary, Search, Word Jumble, Solving, Newspaper games.

Introduction

Word Jumbles can be very frustrating if you stare at a set of letters for a long time and cannot figure out what the word is. The objective of this program is to assist you in figuring out the puzzle when you have almost given up. You open the program, type in the jumbled word and click “Solve.” It will then display the solution. The idea of the program came from us constantly being stumped by jumbles in The Observer day in and day out.

Goals

The most important goal of the program was to solve a jumbled word efficiently without the user needing to wait for any amount of time for the word to be displayed. Less important goals were a decent looking GUI and some type of animation to be displayed while the program was running. Originally, we wanted to display what the algorithm was running through while it was searching for the word.
**Functionality and Platform**

The ideal functionality quickly loads the dictionary into the tree at program startup and then quickly retrieves the correct “un-jumbled” word. This is achieved with consistent, successful results and an easy-to-use GUI. Whether the user is new to computers or a veteran, they should be able to open the program and figure out how to use it. We reached this by writing a very straight-forward GUI. The target platform is Linux as it was compiled in Linux and the GUI was written in QT4. QT4 will have to be installed on the computer as the default QT.

**UML of main class**

The following table is the UML of the main class, Dict. This does not contain all of the functions in the program as it leaves out functions in the GUI class. All data members are not included as there is a node class which contains both private and public functions. All classes can be viewed in the HTML Doxygen files.

```
<table>
<thead>
<tr>
<th>Class Dict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public:</td>
</tr>
<tr>
<td>Dict() ;</td>
</tr>
<tr>
<td>~Dict() ;</td>
</tr>
<tr>
<td>void fillDictionary() ;</td>
</tr>
<tr>
<td>vector&lt;char*&gt;* findwords(const char*) ;</td>
</tr>
<tr>
<td>Private:</td>
</tr>
<tr>
<td>bool moveOnTree(letter*, const bool, const bool) ;</td>
</tr>
<tr>
<td>letter* root ;</td>
</tr>
<tr>
<td>int cnt[26] ;</td>
</tr>
</tbody>
</table>
```
Key Data Structure

As far as the data structure goes that was going to hold the words from the dictionary, it needed to be fast and provide easy access to the words in full.

There are many data structures we could have used for our needs. A vector of words would have worked. The vector would have not been as fast because of the searching which would be needed when trying to find the word we want. It could have been stored alphabetically but there are 27,000 words in the dictionary we are using. That would not have been efficient.

It is pretty standard for a spell check type program to use a Trie to store the words. In that case, each node is a new letter. The following is a great example of a Trie which stores a word letter by letter, going down the tree.

Notice that at the root of the tree, there is a “K” which branches into possible letters to create a word by the leaf nodes, like “kind.”

Our data structure is similar except it uses numbers instead of characters. Each letter has a number specified to it. It stores the instances of a letter that occur in a word. When loading the dictionary in, it counts the occurrences of each letter in the specified word and stores it in the trie that
The next step in our program is to search for the word which the user entered. Let's say they entered the word “fulfill.” Fulfill has three l's, two f's, one I and one u. This occurrence counting is much more efficient than taking the jumbled word and trying different sequences. What I mean by this is that if you were given a sequence of letters, you would need to try every possible combination of those letters against the entire 27,000 word dictionary. What if the user enters the sequence “Pneumonoultramicroscopicsilicovolcanoconiosis”? That sequence has 45 letters and, yes, it is the longest word in a major English dictionary. You would need to figure out every possible combination of those 45 letters and make sure you don't use a combination twice. Without using any combination twice, it already has $1.19622221 \times 10^{56}$ different letter sequence possibilities. Good luck going through each combination and comparing it to a 27,000 word dictionary in a timely matter. When you multiply those together, you are looking at $3.22979996 \times 10^{60}$ instructions for the computer. With the trie we are using, you only need to run through the 45 letters once, only to count the number of times each letter appears. Our search algorithm is looking at a much easier computation when trying to search for any word.

The solutions were stored in a vector of char*s which was returned by the findwords() function. This was because there are times when there could be two different solutions. A good word jumble in a newspaper will not have this happen but it is a possibility.

**Conclusions**

We wanted to create something which can solve the frustration of not being able to figure out what a jumbled word is. This was accomplished and it also looks nice. While most people may not have a need for this program, it will be useful for those who look for it and have a use for it. We know this was accomplished because everyday Mike's roommates come to him to use the program to figure
out words they had yet to get.

We learned a lot about programming a trie and learned even more about trying to program with QT.

References


http://doc.trolltech.com/4.0/ - QT4 API and tutorials

Biographies

John Fullard is a Junior Computer Science major at Notre Dame. He is from Birmingham, Alabama and enjoys mountain biking, soccer, and boxing. He is currently working with Jesus Izaguirre and Faruck Morcos on undergraduate research in bioinformatics and will continue in the Spring. His parents were born in Melbourne, Australia and lived there until 1984. He has two older sisters. Michelle graduated from Vanderbilt University in 2006, and Bonnie graduated from Notre Dame in 2008.

Michael Lehmann is also a Junior Computer Science major at Notre Dame. He spends most of his time being a coxswain on the Rowing Team here at Notre Dame. In his spare time, he enjoys biking, fishing and being on the water. Mike was born in raised in southeast Michigan in a town called Monroe. Mike wants to be on the U.S. National Rowing team and have a good job after college.