C++ Text Searching Algorithms

Developing powerful and efficient text searching techniques in C++

Trieu Nguyen ECE 480 Design Team 1 4/2/2010

Table of Contents

ABSTRACT	3
KEYWORDS	3
OBJECTIVE	4
INTRODUCTION	4
TEXT PARSING	6
TEXT SEARCHING	8
A. SIMPLE SEARCHING B. PARTIAL SEARCHING	8 9
C. COMPLEX SEARCHING	11
CONCLUSION	14
REFERENCES	14

Abstract

Text parsing and searching are problems which programmers are always passively solving. Most user level applications will at some point, will require at least some basic form of text manipulation. Even if this is only a small part of the overall solution, it can often be very time consuming and frustrating. It is often common practice to develop an ad hoc solution for every application of use in order to save time, but creating a generic solution will always help out more in the future. The C++ standard library has several classes which can often help with simple functionality; however, not all of these functionalities are intuitive and can cause ambiguity for simple problems.

Keywords

C++, String, Search, Text, Algorithm, Tokenize, Programming, Parse

Objective

This application note will cover several simple algorithms which deal with text searching and parsing. This document will take several incremental steps to achieving successful text searching algorithms. Not only are these algorithms simple and powerful, but most importantly, they were created to solve a more general problem and can be reused with minor modifications.

Introduction

First we will consider a simple C++ character array:

"This is an array."

Now let's see what this looks like from a coding perspective:

	1												
T	h	i	s	i	S	а	n	а	r	r	а	У	

As you can see, the example is rather simple and intuitive. It's easy to see that we can just use array indexing if we want to modify any of the letters in our array. What if you wanted to remove all the whitespaces? This may require a little more work, but is still pretty simple for this particular example.

													13	
Т	h	i	S	i	S	а	n	а	r	r	а	у		

We can accomplish this by constructing a new array in which every time we run into a whitespace, we shift all the contents to the left. Programming a procedure to do this wouldn't be too difficult. How about matching two arrays or searching for a word within an array? Now the problem becomes much more complex. You can match two arrays by iterating through both arrays and using a boolean operator to compare every index. However, this is not a very versatile technique. The reason for this is because this technique may produce undesirable results if the second array is slightly different.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Т	h	i	S		i	S		а	n		а	r	r	а	У	•
`																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

These two arrays are similar except for one minor difference, the first element in the second array is a lower case 't' and the corresponding index in the first array is an upper case 'T'. Using a boolean operator on this index would return a false because the decimal representation of 't' is 116 and for 'T' is 84. The problem becomes even more complex if we want to search for a particular word within an array. If you wanted to search for the word "is", you could find all the cases where if an index matches 'i', then the word is found if the succeeding index matches 's'. For example:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Т	h	i	S		i	S		а	n		а	r	r	а	у	•
•																•
		i	S		i	S										

We can already see a problem with this technique. The first result is not a word by itself. It is a part of another word. The algorithm would have to be modified to also verify that the proceeding and succeeding indexes are whitespaces. This won't work if the word is at the end or the beginning of the array though. All these edge cases are what turn simple problems into something much more complex.

Text Parsing

The first step in making a versatile text searching algorithm is reorganizing your data into something that is easier to work with. From the examples in the introduction, we can see two major obstacles involved in text searching, whitespaces and upper/lower case letters. A good way to start would be to convert all the characters to either upper or lower case. C++ contains some simple functions that can convert characters to upper or lower case. We can write a simple function which iterates through a string and converts all the characters to upper or lower case.

```
//Function which converts all the characters in a string to upper
string ConvertToUpper(string str)
      //Loop through the size of the string
      for(int i=0;i< str.length();i++)</pre>
            //If the character is not a space
            if(str[i] != ' ')
                  //Reset the value of the array position to the new
                  upper case letter
                  str[i] = toupper(str[i]);
            }
      return str;
}
//Function which converts all the characters in a string to lower
string ConvertToLower(string str)
      //Loop through the size of the string
      for(int i=0;i< str.length();i++)</pre>
            //If the character is not a space
            if(str[i] != ' ')
                  //Reset the value of the array position to the new
                  upper case letter
                  str[i] = tolower(str[i]);
            }
      return str;
}
```

Passing in our original string to the ConvertToLower function results in the following:

			3										
t	h	i	s	i	S	а	n	а	r	r	а	У	•

The second obstacle is to eliminate issues involved with whitespaces.

```
//Function which parses out a string based on the delimeter of choice
The results are stored back into a vector which is passed in by memory
void GetTokens(string str,vector<string>& tokenVector, char token)
    //Skips the delimeters are the beginning of the string
   int lastPosition = str.find_first_not_of(token, 0);
    //Find the first non delimeter
    int position = str.find_first_of(token, lastPosition);
    //While loop which iterates through a string to subtract tokens
   while (npos != position || npos != lastPosition)
        //Adds found token to the vector
       tokenVector.push_back(str.substr(lastPosition, position -
       lastPosition));
       //Finds the next delimeter
       lastPosition = str.find_first_not_of(token, position);
       //Finds the next non delimeter
       position = str.find_first_of(token, lastPosition);
}
```

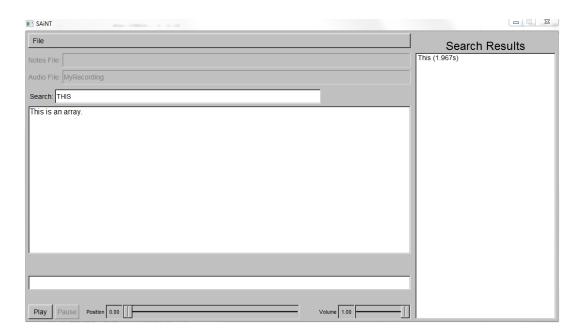
In the above function, we can pass in a string and a delimiter of choice. The function will separate out words based on the delimiter and store them into a vector. The function will look for the delimiter and keep track of the start and ending position. It will then subtract that section from the original string, thus giving us a single word. Once the loop has finished, the resulting vector will contain all of the individual words. In C++, a vector is a dynamic container class which can hold user specified data types. Basically, what we end up with is an array of words. Using vectors is good in this application because a size is not needed to initialize a vector and we won't know ahead of time how many words will be separated out of our original string. By passing our text through both of the above algorithms, we end up with an array of individual words which contain only lower case letters:

0	1	2	3	4	
t	h	i	S		
i	S			_'	
a	n				
а	n				
а	r	r	а	У	•

Text Searching

A. Simple Searching

Now that the data has been cleaned up, we can focus on searching the text and finding matches. If we were to run both the string to search and the original text into the two parsing algorithms, we could simply loop through the resulting word vector to see if any of the arrays match.



This is an example of the simple searching algorithm being used in a GUI application. Notice in the picture that searching for "THIS" still resulted in a success. The parsing algorithms allowed for a much more desirable result.

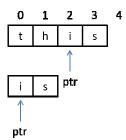
B. Partial Searching

The simple searching technique works well, but it will not return partial matches. In order to accomplish this, we can actually just make a minor adjustment.

This function above will loop through our vector of words. For every word in the vector, the strstr function is used which will return a pointer to the first occurrence of one string in another. If no result is found, the function will return a null pointer. This is a very simple, but effective way to find partial matches.

0	1	2	3	4	
t	h	i	S		
i	S				
а	n				
а	n				
а	r	r	а	У	٠

Searching for the word "is" would achieve two results in the above vector.



While the pointer returns the actual index in which the match is found, for partial matches, we actually don't care about the index position. We just simply need to know if the pointer is NULL or not. This is enough to tell us if the string is a part of a word or not.

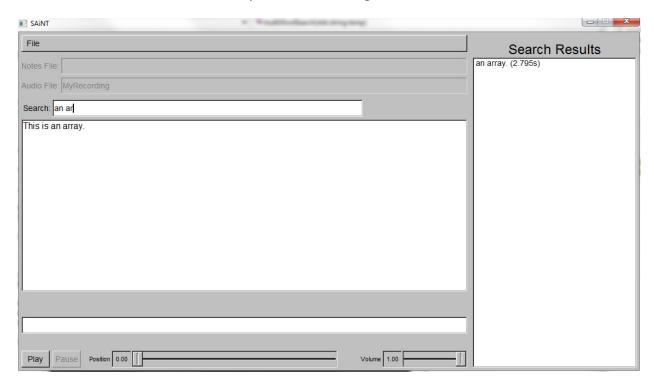
Partial text searching in action:

C. Complex Searching

The next step in searching would be to allow for results that span across multiple words. There are several approaches that can be taken to accomplish this. The general idea for this is that if a search spans across two words, we must keep track whitespaces because that will allow us to differentiate between a single and a multi word search. As before, we can use the strstr function to retrieve a pointer to the first occurrence of our search string. Once that instance is found, we need to subtract all the complete words that matched the search string. We can do this by starting at the pointer index plus the size of the search string. We then iterate through the array until a space is found. Once this is done, we can simply iterate through the original word list to see if we have an ordered match with the search string.

```
//Complex searching algorithm to handle multi word searches
void complexSearch(string temp)
         //This uses the strstr function to find if an occurance of our search string has occurred in the
        original string
         const char *ptr = strstr(originalNotes.c_str(), temp.c_str());
         //A match is found if the pointer returned by the strstr function is not NULL
         if( ptr != NULL)
                 //Create a new string by using strdup to duplicate the string starting at the position an
                 occurance was found
                 string str = strdup(ptr);
                 string searchString = temp;
                 int start = 0;
                 string final;
                 bool finished = false;
                 //Round up the last word by finding the next whitespace
                 while(start < searchString.size()-1 || finished == false && start<str.size())</pre>
                          //Make sure we don't go out of bounds while itterating the string
                          if(start > searchString.size()-1)
                                   //If we find anything, but a space, add the character to our last
                                   if(str[start] != ' ')
                                            //Add the character to the final word and increase the position
                                           final = final + str[start];
                                           start++;
                                   //If a space was found, it means that the word has been rounded up
                                   else
                                            //Set a boolean so our loop will break
                                           finished = true;
                          //This is an edge case for if the word to be rounded is the last word in the original
                          string
                          else
                          {
                                   //Add the character to the final word and increase the position
                                   final = final + str[start];
                                   start++;
                 //Inilize a vector to store tokenized strings into
                 vector<string> tokens;
                  //Call the tokenizing function and separate the words by space
                 GetTokens(final, tokens, ' ');
                 int foundPosition;
                 //Look for the occurance of the first search word in vector of original words
                 for (int j = 0; j < formattedText.size(); j++)</pre>
                          const char *ptr = strstr(formattedText[j].c_str(), tokens[0].c_str());
                          //Found an occurance
                          if(ptr != NULL)
                                   //If we find a match, we need to start at that position and see if the next
                                   words in the search vector match the corresponding order of words in the
                                   original word vector
                                   foundPosition = j;
                                   int count = 0;
                                   //Loop through the vector of tokens
                                   for (int i = 0; i < tokens.size(); i++)
                                           //If the arrays don't match, then break from the loop
                                           if(tokens[i] != formattedText[foundPosition])
                                           {
                                           foundPosition++:
                                           count++;
                                   ^{\prime} //If all occurance of the search string occurred in correct order, then we are
                                   done
                                   if ( count == tokens.size() )
                                           searchResults.push back(final);
                      }
                                                                                                        Page | 12
}
```

Complex text searching in action:



In the above example, we searched for "an ar"; the complex searching algorithm was able to find the first instance of that string. The function than kept iterating until it reached a whitespace and subtracted "an array" from the original text. The new string was then tokenized to form a two word vector which was then compared to the original vector to obtain a result.

Conclusion

Text searching can turn into a very complex problem and if not programmed properly, can be very time consuming to implement and test. By implementing the solution in small iterative steps, such as those discussed earlier, a versatile searching algorithm can be quickly made for any specific application.

References

STRSTR Reference

http://www.cplusplus.com/reference/clibrary/cstring/strstr/

Nyhoff, Lary. ADT's, Data Structures, and Problem Solving with C++.

Prentice Hall, 2005.

C++ Standard Library: The string Class

http://pages.cs.wisc.edu/~hasti/cs368/CppTutorial/NOTES/STRING.html