BULK SPELL CHECKER

March 9, 2014

Ajaypal Singh(120101005)

Introduction

- Bulk Spell Checker is a program for bulk spell-checking and correcting an entire website.
- There are various components in this program .Firstly, there is a parser that scans a local copy of the entire website, ignoring HTML (Hypertext Markup Language) markup, creating a list of all the words used anywhere on the entire website, with a count of how many times each word was used along with their co-ordinates.
- A hashtable is made for the words in a dictionary and the words are decided to be good or bad based in this hashtable of the dictionary words.
- The list of the bad words is displayed to the user in a listbox sorted on the basis of frequency and he can choose from there any word at random and correct the occurrences of that word.
- For the selected words a list of suggestions is displayed to the user and he can select the replacement word from there or he can type it himself.
- If the user is not satisfied with a suggestion and is not sure of the replacement word then he can search for the suggestions online just by a button click.
- If the user is sure of the word that it is correctly used then he can ignore that occurrence of that word.He can also ignore all the occurrences of a wrong word.
- The user can replace the words one by one based on context or he can replace all the words at a time just by a button click.
- The user has an option to save changes and overwrite the original file or he can use Save As option to save it as a new file.
- The user can also change one occurrence of that word and leave the other occurrence .Even then the wrong word is there in the list of wrong words until all the occurrences of that word is dealt.
- The user can see the occurrences of all the wrong fords at a click of just one button or he could see them one by one.

Modules, functions Projectflow and Working

There are various modules and functions used in the project.A brief description of all the modules and functions with their functionality is explained in the later part of this document. The projectflow is also described in this document.

- PARSING AND TOKENISING

  For the purpose of parsing a header file named parseandtokenise.h is used. In this file there is a class called parsefile in which the primary function used is void parse(string filepath).This function takes an a input the filepath i.e. the path of the file to be spell checked.

  – Storing Co-ordinates

  Then it scans through the whole file and stores the co-ordinates of the words.Here co-ordinates mean linenumber as y co-ordinate and the start and ending index of each word using the index of characters in the whole file. For storing the co-ordinates of possible words the logic uses four di↵erent cases cases such as whether the two characters are both normal alphabetic characters, both special characters, one special and one alphabetic character.

  – Removing contents between ”<” and ”>”

  After getting a temporary file called ”temp.txt” which is obtained after storing the co- ordinates and is removed afterwards parsing of the file is done. This takes place in several steps where one by one the rules to parse a html file is used.Firstly all the contents between ”<” and ”>” are removed as they come under HTML syntax and need is beyond the scope of our Spell Checker.

  – Removing contents between { and }

  After we have removed content between < and > we now remove the contents between { and }. Note that after each step we are creating a temporary file and we are using that temporary file as an input to next step. Also the co-ordinates are also retained in the files.

  – Removing contents between $ and ;

  We now remove the contents between $ and the first ; symbol as this contents are also a part of the HTML and they need to be ignored.

  – Tokenising and putting in a file

  We now tokenise the file that we get after previous step. For this we read the file line by line and then tokenise the lines using the delimiters.
  const char*const DELIMITER= ” ’,@;()|”‘←!#$%V&*– = +[ ]{ }:\.\\”; We are using the strtok(buf,DELIMITER); Here we have assumed two things

  const int MAX CHARS PER LINE = 1000;
  const int MAX TOKENS PER LINE = 200;

  We are also making sure that words of size less that 2 are not considered in spell checking as it will just create trouble for the user.

  – Storing the words with alphabetic character

  After getting the token we now add these token to a file long with their three co-ordinates and we are making sure that the token only have alphabetic characters.If the token has any numeric or special character then we are neglecting that word.For this we are built a function bool isalphabetic (string s,int siz)which takes the string and its length as input arguments and returns whether that string is alphabetic or not.
  We have also a function void MarshalString (string ˆ s,string & os). This function takes a string as an input and a String ˆ and assigns the content of String ˆ to the string.

After this step we have a temporary file that has all the possible strings not within the html context and along with the words we have their co-ordiantes.

The Parsing and tokenising steps are done when the user selects a file path from the homepage and clicks on SpellCheckButton.

- MAKING THE HASHTABLE OF DICTIONARY WORDS

For this purpose we have used a header file hashtable.h.There is a class called class hashtable which has the following functions and variables in it:

– Variables

```
struct node
    {
        ItemType CurrentItem;
        node*next;
    };
    node** head; // array of linked list declaration - front of each hash table
    cell int hashtable; // the size of the hash table (how many cells it has)
    int totElems; // holds the total number of elements in the entire table
    int* bucketSize; // holds the total number of elems in each specific hash table cell
```

– Functions

⇤HashTable(int hashsze=HASH SIZE));
Function:Constructor initializes hash table
Precondition:None
Postcondition:Defines private variables

⇤bool IsEmpty(int key);
Function:Determines whether hash table is empty at the given key Precondition:Hash table has been created
Postcondition:The function = true if the hash table is empty and the function = false if hash table is not empty

⇤bool IsFull();
Function:Determines whether hash table is full Precondition:Hash table has been created
Postcondition:The function = true if the hash table is full and the function = false if hash table is not full

⇤int Hash(ItemType newItem);
Function:Computes and returns a unique hash key for a given item The returned key is the given cell where the item resides
Precondition:Hash table has been created and is not full
Postcondition:The hash key is returned

⇤void Insert(ItemType newItem);
Function:Adds newItem to the back of the list at a given key in the hash table. A unique

hash key is automatically generated for each newItem.
Precondition:Hash table has been created and is not full
Postcondition:Item is in the hash table

⇤ void Append(int key, ItemType newItem);
Function:Adds new item to the end of the list at a given key in the hash table
Precondition:Hash has been created and is not full
Postcondition:Item is in the hash table

⇤ bool Remove(ItemType deleteItem, int key = −1);
Function:Removes the first instance from the table whose value is ”deleteItem”. Optional second parameter indicates the key where deleteItem is located
Precondition:Hash table has been created and is not empty
Postcondition:The function = true if deleteItem is found and the function = false if deleteItem is not found

⇤ void Sort(int key);
Function:Sort the items in the table at the given key
Precondition:Hash table has been initialized
Postcondition:The hash table is sorted

⇤ void TableSize();
Function:Return the size of the hash table
Precondition:Hash table has been initialized
Postcondition:The size of the hash table is returned

⇤ void TotalElems();
Function:Return the total number of elements contained in the hash table Precondition:Hash table has been initialized
Postcondition:The size of the hash table is returned

⇤ int BucketSize(int key);
Function:Return the number of items contained in the hash table cell at the given key Precondition:Hash table has been initialized
Postcondition:The size of the given key cell is returned

⇤ int Count(ItemType searchItem);
Function:Return the number of times searchItem appears in the table.Only works on items located in their correctly hashed cells
Precondition:Hash table has been initialized
Postcondition:The number of times searchItem appears in the table is returned

⇤ void MakeEmpty();
Function:Initializes hash table to an empty state
Precondition:Hash table has been created
Postcondition:Hash table no longer exists

⇤ HashTable();
Function:Removes the hash table

Precondition:Hash table has been declared
Postcondition:Hash table no longer exists

⇤ In the hashtable.h we also have a class called class Iterator; Function:Class declaration to the iterator
Precondition:Hash table has been declared
Postcondition:Hash Iterator has been declared

⇤ Iterator begin(int key){return(!IsEmpty(key)) ? head[key]:NULL;}
Function:Returns the beginning of the current hash cell list
Precondition:Hash table has been declared Postcondition:Hash cell has been returned to the Iterator

⇤ Iterator end(int key=0){return NULL;} Function:Returns the end of the current hash cell list Precondition:Hash table has been declared Postcondition:Hash cell has been returned to the Iterator

We do the pre-processing of making the hashtable before loading the application. We read the words from a dictionary and add them to the hashtable. This takes a while but it is feasible as there may be various files to be spellchecked.

- ADDING THE WORDS FROM TEMPORARY FILE TO THE LIST OF BAD WORDS

When we have parsed the file and we have the file with all the words and their co- ordinates then we add them to the list of bad words. For maintaining the list of bad words we have created a header file called node.h. It has a class called class word.
There are various components in it which are described below:

1. vector<Node> node;
2. vector<Node>::iterator it;

Here we have Node as a structure which has the following
structure struct Node{

```
int key;
string word;
vector<position> pos;
vector<position>::iterator posit;
Node(string wor, int y, int x1, int x2)
{posit=pos.begin();
word=wor;
key=1;
position *newposition=new position(y,x1,x2);
pos.push_back(*newposition);
}
};
```

Here Node(string wor, int y,int x1,int x2) s the constructor for the node and posit is an iterator for the vector pos.Each node in the vector will symbolise a word and the position vector will store all the occurrences of that word in nodes called position.Also, here key is the frequency of a word and it is incremented each time the word is added and decresed each time the word is deleted.
Again here position is a structure and it has the following

```
structure struct position {
        int y,x1,x2;
        position(int yin, int x1in, int x2in)
        {
        y=yin;
        x1=x1in;
        x2=x2in;
        }
};
```

Here y,x1,x2 are the linenumber, starting co-ordinate and the ending co-ordinates of a word respec-tively.
There are various functions in the class words and they are as described below:

- void addsame(string wor, int y, int x1, int x2)
  This function assumes that the word is already present in the vector and the add an occurrence of the word to the position vector.
- void addnew(string wor, int y, int x1, int x2)
  This function adds the first occurrence of a word in the vector of words.
- void updatenode(int xindex, int di↵erence)
  This function updates the co-ordinates of the words that come after a particular index using the index and the di↵erence.It is used when a word is deleted and co- ordinates of the words after that word in file needs to be updated as the replacement word need not have the same number of characters as the word to be replaced.
- void getfirstcoordinates(string word, int& ycor, int& xstart, int& xend)
  This function assigns the co-ordinates of the first occurrence of a word to the variable ycor,xstart,and xend.
- int getkey(string word)
  This function is used to get the frequency of a give word.
- void deletealloccurrenceofword(string word)
  This function deletes a node from the vector of nodes that store the words and their occur-rences.This is used to delete a words occurrence when its spellcheck is complete or when it is ignored.
- void deletefirstoccurrenceofstring(string word)
  This function deletes the first co-ordinates from a node in the vector of nodes that store the words and their occurrences. This is used to delete a words occurrence when it is replaced once or when it is ignored once.
- bool ispresent(string word)
  This function is used to see if a word is present in the vector already or not.if it is present then it return true otherwise false.

– void sortkey()

This function is used to sort the nodes on the basis of their frequency so that the user can be shown the words that are misspelled sorted on the basis of their frequency.

– void print()

This function is used to print the vector of nodes with words and all the co- ordinates and it is just used for debugging purpose.

- GRAPHICAL USER INTERFACE

For the GUI part one form is used with three tabs in it.

– Parsing the file and adding the bad words in listbox

Initially, tabpage 1 opens which has the textbox1 and a button called browse (button1) and a button called SpellCheck (button2).

On the click event of button 1 a openDialogBox opens where the user can browse the file to be spellchecked.the filepath is then added to textbox1 and when the user clicks on spellcheck button then the functions parse of the class parsefile is called which makes a list of the words in a vector of bad words if these words are not present in the hastable of the dictionary words and then finally adds them to the listbox1 present in tabpage2 of the form. Then the selected tab is changed to tabpage2 and the user can now see tabpage2.

– richtextBox1 to show the bad words

Now the user can see the occurrences of bad words by clicking on the bad words.he can do this for one word at a time or for all the words at a time.

On the selectedindex changed event of the listbox1 the selected word is highlighted in the richtextbox1.On the click event of Show All words (button18) all the words that are misspelled are highlighted.

– Correct button

The user can select a word see if the word is really misspelled in the context and then click on Correct button if he want to make changes to the file. This occurs on the event button click of button8.Then the selected tabpage is tabpage3.On the click of this button all the suggestions are added to a listbox2 in tabpage3. In tabpage3 the words that was selected in tabpage 2 is kept highlighted in richtextbox2 so that the user can see the occurrences of a misspelled word.For this purpose a function called void SpellCheck(HashTable)<string>&hashTable, string word) is used. In this function we take a word and make various changes in the word and see if the changed word is there in the hastable of dictionary words and if the changed word is there in the hashtable then they are given as suggestion to the user. The various changes made to a word are alteration of a character, insertion of a character, transposition & deletion of a character.

– Replace Button

In tabpage3 there is a button named Replace button (button10). This button when clicked replaces the first occurrence of the selected word. This is done using the function.

void replaceoneoccurence(string old, string neww) This function takes two arguments first the old word that is the word to be replaced and the new word that is the replacement word. In this function there are calls to various other functions like

```
wo.ispresent(old),
wo.getfirstcoordinates(old,y,xstart,xend),
wo.deletefirstoccurrenceofstring(old)
wo.updatenode(xstart+1, di↵)
```

These functions have already been discussed in previous sections of this manual.

– Replace All Button

This button in tabpage3 (button11) when clicked replaces all the occurrences of a word from the richtextbox2 which where in vector of bad words.It also removes the word from the listbox1 of tabpage2 as all the cases relevant to that word will be dealt.This button call the function void replaceoneoccurence(string old, string neww) multiple times to replace all occurrences of a word.

Then a MessageBox is displayed displaying the number of occurrences replaced.

– Ignore Once button

This button present in tabpage3 (button13) when clicked calls the functions

```
bo.MarshelString(textBox2− >Text, word)
wo.getkey(word)
wo.getfirstcoordinates(word,y,xstart,xend)
wo.deletefirstoccurrenceofstring(word)
```

These functions when called in order helps in getting the first co-ordinates and then ignoring that and deleting that occurrence of the word from the vector of bad words.

– Ignore All button

This button in tabpage3 (button14) when clicked removes that word from the listbox1 in tab-page2 and this is sufficient to ignore all the occurrences of the word selected.

– Add to dictionary

This button on tabpage3 (button12) when clicked adds the word in Textbox2 i.e, the word selected to be spellchecked to the dictionary so that it is not displayed to the user when he spellchecks in future.Before adding the word to the dictionary it is first converted to small letters . For this we use the function string ToLowerCase(string word).

This function takes a word as input argument and return it with all characters in small letters.

– Search Online Button

This button is useful when a word is not there in our dictionary and the user is also not sure about the word.This button(button16) when clicked opens a url in default browser. The user is redirected to a site where he could directly see the word and suggestions with definitions. For this the function used is system(”start” URL)

– Save Button

This button(button15) when clicked saves the changes made to the original file.And the original file is overwritten by the contents of richTextBox2.

– Save As Button

This button (button19) when clicked opens a Savefiledialogbox open the user specifies the file type and the filepath along with the file name then the a new file is created with the contents of that file being same as the content of richtextbox2.

– Back Button

This button (button9) when clicked checks if all the occurrences if the selected word were dealt or not. If yes then the word is removed from the listbox1 otherwise it is retained in the listbox1.

– Back to homepage Button

These buttons (button4 and button5) in tabpage 2 and tabpage 3 respectively are the buttons that take the user to the file selection page.where he can select another file to be spell checked.

– Exit Button

These button (button6 and button7) present in tabpage2 and tabpage 3 respectively are the buttons that are used to exit the application. The user is given a prompt if he really wants to

exit and if yes then the application is closed.
The function used for this is Application::Exit().

– Suggestions when selected get added to textbox3

When a suggestion is selected by the user then it automatically get added to the listbox3 and the user can replace the misspelled word with suggested word.

– Menu bar

There are various options provided in the menu bar like File menu, Edit menu and help. File menu has options like New, Save, Save As, Exit and they have the same code as the buttons for these funtions in tabpage3.New indicates homepage button.

In edit menu There are options like replace, replace all and ignore one and ignore all and they too have the same code as their buttons in tabpage3.The help menu open area only User manual in Textbox in new tab tabpage4.

LIMITATIONS

– Size of the input file must be less than 20kb.

– Maximum number of characters per each line is 1000.

– Maximum number of tokens per each line is 200.