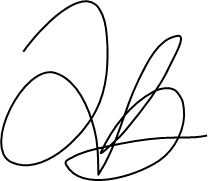
**Bulgarian Diploma Thesis**

**Simple English Translator  
  
 Damjan Kulukovski, 200041131**



**Student: \_\_\_\_\_\_\_\_\_\_\_\_ Date: 12/04/2023**

**Supervisor:\_\_\_\_\_\_\_\_\_\_\_\_ Date: 12/04/2023**

**Department of Computer Science, AUBG**

**Blagoevgrad, 2023**

**Declaration template for Senior Project and Diploma Thesis**

**Title:** Simple English Translator

**Author:** Damjan Kulukovski

**Abstract:** A program that automatically detects and autocorrects complex english text and translates it into simple english. The program uses several complex algorithms to achieve clear translation of the words as well as precise spell checking. Receiving input in the form of text files and outputting in the form of text files, the program can handle large inputs. It will also have the ability to have custom word mapping if necessary.

**Declaration of authorship:**

“The Senior Project/Bulgarian Diploma Thesis presented here is the work of the author solely, without any external help, under the supervision of Dimitar Christozov. All sources, used in development, are cited in the text and in the Reference section.”

**Author:** Damjan Kulukovski

Table of contents

1. Introduction……………………………………………………………………4

2. Software requirements………………………………………………………8

4. Software Solution……………………………………………………………21

5. Implementation………………………………………………………………43

6.Testing……………………………………………………………………….53

7. Conclusion…………………………………………………………………57

8.References………………………………………………………………….60

English to simple english translator

Abstract

An application that will input complex english text that a user intends to simplify. This program will make use of various algorithms that process the words, as well as maps that provide the link between the complex terms and the simple terms. The output of the program will be simple and understandable english texts.

1. Introduction

The premise of this idea is to have a program that recieves english text as an input, preferably complex english, and outputs a simplified version of the text that is easier to read. There is a lot more complexity in this issue than simply having a one to one translation of complex words to more simple ones, because of the complexity of language itself, as well as the occasional difficulty in even simplifying of words themselves.

Using complicated English can have a high barrier of entry for a lot of people, especially those from foreign countries that speak English as a second or even third language. This project could help a lot of people get over their fear of english, a large problem I have seen with people with limited English is their fear of reading complex terms. They often resort to translating the terms back into their native language, which could distort or completely ruin the meaning of the sentence/word. A better method to counter this will be to translate from english back to english but with simplified terms that most readers will easily understand. The aim of this software is to curb that barrier to entry enough regardless of the complexity of the given text.

The program will take in information about the sentences and words used and map them out to a set of predetermined and curated simple english words. Once it has determined which words will be necessary for translation it will convert them and then reform the sentences/paragraphs back as grammatically correct as possible. Obviously some words or concepts will be more challenging to coherently simplify than others, but the aim of the project is not a perfect result, it is to get the larger meaning of the text across.

Amount of people that have speak english at a lower level.

1. Examples of different simple english methods

The first example of english simplification is from wikipedia

* An example of text from English wikipedia

A continent is any of several large geographical regions. Continents are generally identified by convention rather than any strict criteria. A continent could be a single landmass or a part of a very large landmass, as in the case of Asia or Europe. Due to this, the number of continents varies; up to seven or as few as four geographical regions are commonly regarded as continents. Most English-speaking countries recognize seven regions as continents. In order from largest to smallest in area, these seven regions are Asia, Africa, North America, South America, Antarctica, Europe, and Australia.[1] Different variations with fewer continents merge some of these regions, examples of this are merging North America and South America into America, Asia and Europe into Eurasia, and Africa, Asia, and Europe into Afro-Eurasia.

* An example of the same page but with simple English.

A continent is a large area of the land on Earth that is joined together.

There are no strict rules for what land is considered a continent, but in general it is agreed there are six or seven continents in the world, including Africa, Antarctica, Asia, Europe, North America, Oceania (or Australasia),[1] and South

The most populous continent by population is Asia, followed by Africa. The third most populous continent is Europe. The fourth most populous is North America, and then South America. In sub-Saharan Africa, the largest age group are denarians (in their teens). In north Africa, the largest age group are vicenarian (in their twenties). In Europe, most people are tricenarian (in their thirties) or quadragenarian (in their forties).

The second example is with some simple english translation software

* An example of an english text

North America was reached by its first human populations during the Last Glacial Period , via crossing the Bering land bridge approximately 20,000 to 17,000 years ago. The so- called Paleo - Indian period is taken to have lasted until about 10,000 years ago (the beginning of the Archaic or Meso - Indian period ). The classic stage spans roughly the 6th to 13th centuries. The first recorded Europeans to visit North America (other than Greenland ) were the Norse around 1000 AD. Christopher Columbus's arrival in 1492 sparked a transatlantic exchange which included migrations of European settlers during the Age of Discovery and the early modern period. Present-day cultural and ethnic patterns reflect interactions between European colonists , indigenous peoples , African slaves , immigrants from Europe , Asia , and the descendants of these groups.

* And its translation to simple english

North America was got to by its first to do with man groups during the Last of ice stretch of time, via crossing the Bering land bridge approximately 20,000 to 17,000 years ago. The so- called Paleo - indian stretch of time is taken to have lasted until about 10,000 years ago (the start of the of very old times or Meso - indian stretch of time ). The great stage goes across roughly the 6th to 13th hundreds of years. The first recorded Europeans to go to North America (other than Greenland ) were the Norse around 1000 advertisement. christopher Columbus's getting in 1492 sparked a transatlantic exchange which included movings of European persons living in new lands during the existence-stage of Discovery and the early current-day stretch of time. present-day art and learning and of divisions of man designs give signs of effects on one another between European representatives of a country controlling another, naturally growing in a given place groups of persons, African persons as property, persons coming into the country in the near past from Europe 1, Asia 2, and the ones coming after of these groups.

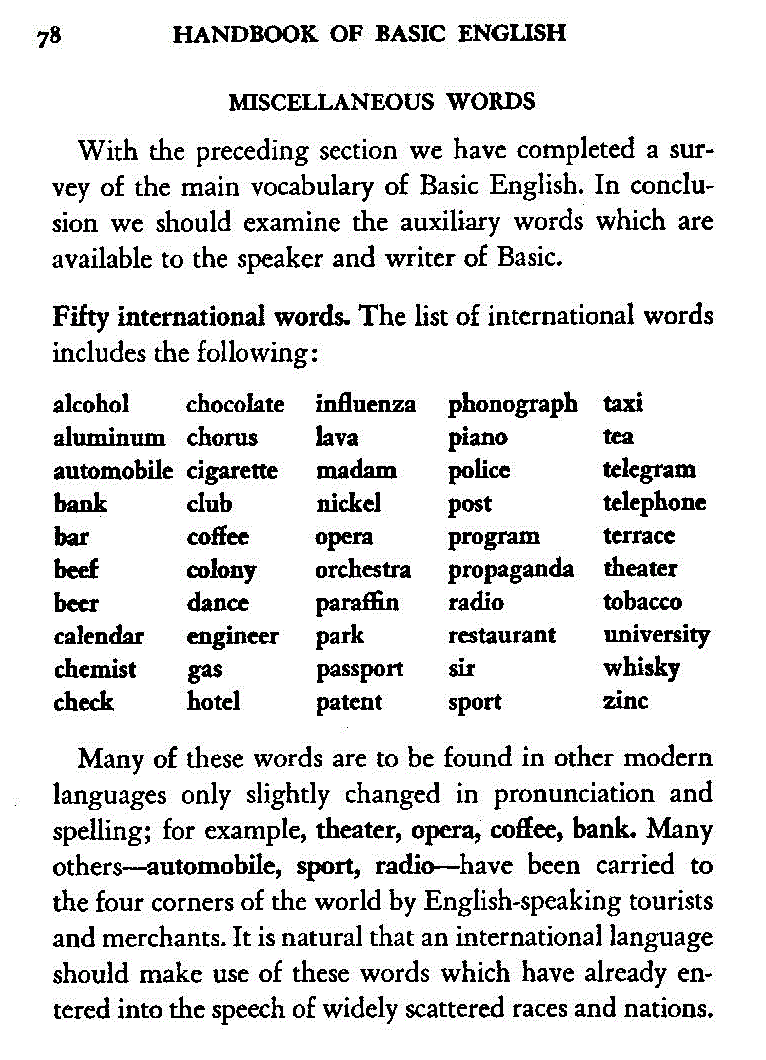
The reason why I highlighted these two is because the first one reads more smoothly yet it still has some complex words, and the second one has more basic words yet it becomes longer and harder to manage grammatically. The goal of this project will be to take in all of these existing methods and try and establish one that is not very cumbersome to read, yet it does not drag on too much. This can be done by leaving some words that prove troublesome for the sake of the structure, but adding footnotes to explain their purpose.

1. Software requirements

To start off this project it is necessary to figure out exactly what constitutes “complex” english and what constitutes “simple” english. Thankfully, research into these specific things has already been done to some extent. Usually dictionaries that are for Simple English have a very limited amount of words ( some being roughly only 1000 words), this is because those words carry a lot of meaning, as well as being understood by as large of a group of people as possible. One example of a constructed dictionary of simple english words would be Odgen’s simple english dictionary[1]. This provides a base that can be used to weed out the complex words in a sentence.

Now the second step in creating some sort of model will be to create a dictionary of complex words and begin mapping it to the simple words, this step will be very demanding due to how it will require some manual labor to get it off the ground. The complexity of this step will be the problem of how simplification will affect the complex words, it’s obviously impossible to make a 1 to 1 map between the complex and the simple english words, therefore it will require a lot of constructing new phrases from the simple english words to describe the more complex terms in an understandable manner. Acquiring a complex dictionary won’t be terribly complex considering there is a plethora of resources regarding this, finding their more simple counterparts might be challenging but very doable.

After we have those two dictionaries we will need to build the relationship between them and also find the overlaps. While most words will need to be translated, a large section of common words will not be, those words or phrases will be directly transferred over with no need to simplify.



1. Creating the method of translation

With the dictionaries we have previously established we can go into the more complex part of the program which is the actual method of translation. A parser will be used to find the words that need to be translated as well as the linking words or redundant words that can be omitted. This will essentially mean that a tokenizer will have to be made in order to break down sentences into smaller parts so it's more manageable.  
 This process will require algorithms to match the words from the input to the words in the complex english dictionary. The act of pattern matching as well as tokenizing sentences will make it so I'll have to build a few custom algorithms from the ground up to accommodate the use case.   
 After the actual sentence or sentences have been broken down we can continue with the translation part, which will see the complex english dictionary being mapped to the simple english dictionary. As previously stated this cannot be a 1 to 1 comparison due to the differences in sizes between the two dictionaries. So a lot of mappings will overlap and even use two or more words from the simple english dictionary, this is not a problem and is the exact reason mapping exists. This will constitute the bulk of the translation within the program, while it is the most key to the functioning of the program itself it is not particularly the most complex aspect surprisingly. Eventually this process will spit out different simple words that have been translated and are ready to be merged back into the simple sentences.

The final part of the translator will be taking the words that we got from the previous process using the non translated connector words from the original sentence and piece them back together in a coherent way. This step will also require some effort on how it will have to figure out which words are redundant to the sentence structure as well as introduce back punctuation into the sentences and string it back together. This step will also require the program to provide deeper translations regarding some words because of how they cannot be simply translated, this could be seen in the form of a footnote or an explanation in parentheses for the convenience of the user.

After the text has been fully translated and corrected to be coherent, it will be given back to the user with the proper footnotes or descriptions necessary to understand it fully. Inevitably there will be some grammatical mistakes or awkward sentences but the act of translating is always imperfect in its nature, this project is no different.

Another Element that will be implemented during this process somewhere between the tokenizing and the translation will be the spell checking of the words themselves. This will constitute the bulk of the algorithms that will be used throughout the program due to the complex nature of figuring out the differences between worse and their proximity so we can try and translate them.  
 To achieve this I have developed a custom algorithm that incorporates two preexisted, but modified algorithms, into one. The algorithms that are used are levenshtein distance as well as Dijkstra's algorithm. Both fairly complex and key in their own way.  
 Arguably the bigger emphasis will be on the levenshtein distance due to how every misspelled word will be pumped through it to find the distance between the misspelled words as well as the list of regular, correctly spelled, english words.

Simply finding the distance between two string isn’t necessarily the issue we're going to be struggling with, levenshtein distance already does that by default. It’s finding the correct, or what we estimate to be the correct word, out of the plethora of words that will be returned which will be the challenge. To deal with that is the rest of the algorithm, once we have the levenstein distance, that was also modified by me to include some alterened distances based on certain scenarios such as the letters being flipped or words having an additional letter or lacking one. We will take the numbers that come out with a levenshtein distance of 1 and continue working with them.  
 This part is where the dijkstra's algorithm comes into play. We will take the words of distance one that the previous part of the algorithm gave us and start comparing the letters that are off within the misspelled words, now this is a fiddly problem due to how the previous solution was already altered to have some nonstandard 1 distance words such as more letters or less letter or switched letters. This is where simply personal judgment was added. Without using NLPs and knowing the context of the words surrounding the word we are checking it is somewhat difficult to precisely determine which word we are actually looking for. Our best guess is simply to look at how the word we are given was misspelled and get an educated guess based on that. So from my personal judgement i made the order of misspellings like so:

* Letters being switched will warrant a 0.5 letter distance(I will explain exactly what letter distance is soon)
* 1 letter being off in a word and having a letter distance of 1
* Having an additional letter or a lack of a letter will warrant a letter distance of 1.1
* Afterwards the rest of the 1 letter being off but having a letter distance of >1

Now what is letter distance? Letter distance is mostly the dijkstra length between the letters on the keyboard. When that is impossible, such as the case of 2 letters switching places in a word, a letter distance is assigned without using the dijkstra algorithm. Essentially once the program finds that a word is misspelled it finds out which letter is off in the word and then runs the dijkstra algorithm to find the distance between the two letters and spits out a value of how far away they are.  
 I found this to be the most consistent way of figuring out what to correct the misspelled word to. Because usually when we make a spelling error we will most likely make the mistake with a letter close by or by switching up letters or simply forgetting to press a button. This is by no means an exact science do not be fooled. At the worst of times this algorithms will just be a large guesstimation and will get some things wrong. But such are the difficulties of trying to automatically solve human error, especially when we do not know their intent behind their goal.

One challenging thing within this whole project was realizing the importance of context as well as the grander meaning of a sentence and how that affects our translation and our spell correction. I wanted to test out the limits of no machine learning based solutions for these problems and it is quite interesting the results I got.

Now that we are equipped with the tools on how to compare different strings, how does the

Finding if the word was misspelled was quite straight-forward. I have 14 files worth of words that are all the english language words of that specific length. I split them up on based on the length to avoid having to compare every word to every word in the english language(there are over 460k+ words in the english language, that is a lot of words and stress to put on a program). Once the program determines that the word is misspelled because it cannot find it in the set of same length words, it begins the comparison of words.

Now yet again this presented the biggest time restraint challenge, it is not horribly quick to cycle through every word in the english language and then put an algorithm to compare. So i thought the easiest solution would be to compare the misspelled word to only words of its length, one higher, and lower. This would provide a reasonably large sample size of words to see what word it could be while also preserving time efficacy.

The final step now is to simply go through all the words and see which one is closest to our misspelled word. The algorithm gives us all the words with a levenshtein distance of 1 and also its letter distance. The word with the lowest letter distance will then be picked to replace the misspelled word in our program, and if the letter distances are the same the program will simply pick one( again it's not a perfect science, because spelling mistakes are never perfectly wrong either). It pumps the word back into the program and then the program checks the complexity of the word and sees if it needs to be translated.

And after that whole process the program spits out the final text for the user to read, hopefully easier than the original text that it was given.

The program will also have some UI written in C# that will allow the user to more easily input the text they want if they don't want to manually make a .txt file and another box that enables the user to add words that they do not want to be spell checked, which will be added to the list of valid words and then not be autocorrected. This is done for a bit more straightforwardness for the user so they do not have to know the code by heart(unfortunately) to use the program in any capacity.

Here is some pseudo code of how some of the elements of the program work:

ifstream fi("file.txt");

Ofstream fo(“output.txt”);  
 for(sentence tokenizer(fi) : sentences){

vector<string> word = word tokenizer(sentences);

for(word : words){

if(spellcheck(word) == false)

{

Word = correctword(word);

}

if(checkifcomplex(word) == 1)

{

Word = simpleversion(word);

}

Of << word;

}

}

That is the very basics of the main function but here is some pseudo code surrounding the spell checker

Class spellchecker{

Private:  
 Variables and lists

Public:  
 Initializer ()

{

Add keyboard

Open spelling files;

while(files)

{

Put in vector of sets;

}

}

Int Dijkstra(char a, char b)

{

Return dijkstradistance;

}

strut levenshtein(string a, string b)

{

Return enhancedlevenshtein,regularlevenshtein;

}

String checker(string a){

String b = “”;

While( regularlevenshtein(a) == 1)

{

if(b.enhancedlevenshtein < a.ehancedleveshtein){

b = a;

}

return b;

}

Int spell(string a)

{

if(wordset.find(a) == 0)

{

return 0;

}else

{

return 1;

}

}

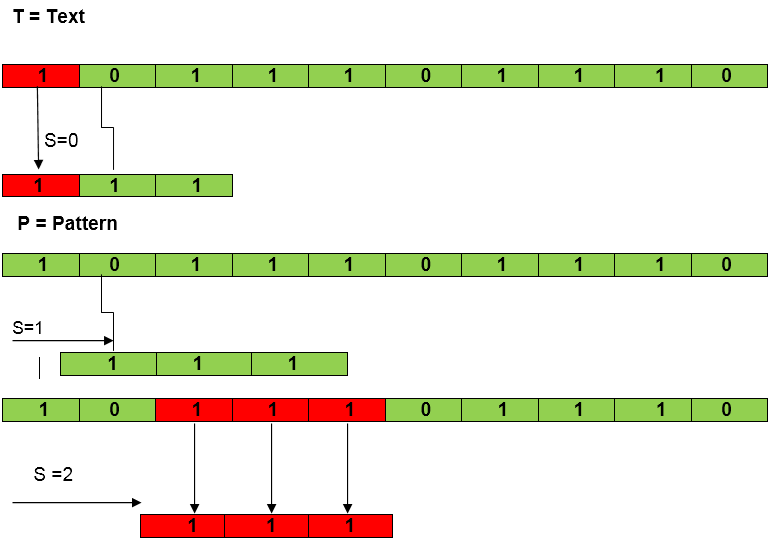
This class essentially contains all the necessary functions for the program to do a successful spellcheck and replacement when necessary.

1. Pattern matching

A key part of the functionality of this program will be pattern matchers.  
  
 Pattern matchers are algorithms that go through text and find patterns when comparing to another library of words. This will obviously be very useful when the program goes through the text and matches them with the complex english dictionary. Efficiency in this section of the program will be key due to how this will be the most computationally demanding part of the program. It will run this algorithm hundreds, thousands of times to go through a single text so it will have to be finely tuned to the program at hand.

While there are already preexisting pattern matching algorithms, one will have to be custom created for this program to match its needs. Functionality such as understanding when a word doesn't line up with any words given in the complex english dictionary. In this case there are two potential reasons, misspelling or the word is a place/ not technically a word in the dictionary. The algorithm will have to determine how to deal with this word and progress it to dictionary mapping or directly to the text as a connector word with information about its role.

A problematic algorithm here could break the entire program immediately, either by using up too many resources or providing simply wrong translations, which will obviously both largely hamper the effectiveness of the program.



Example of Naive pattern matching algorithm

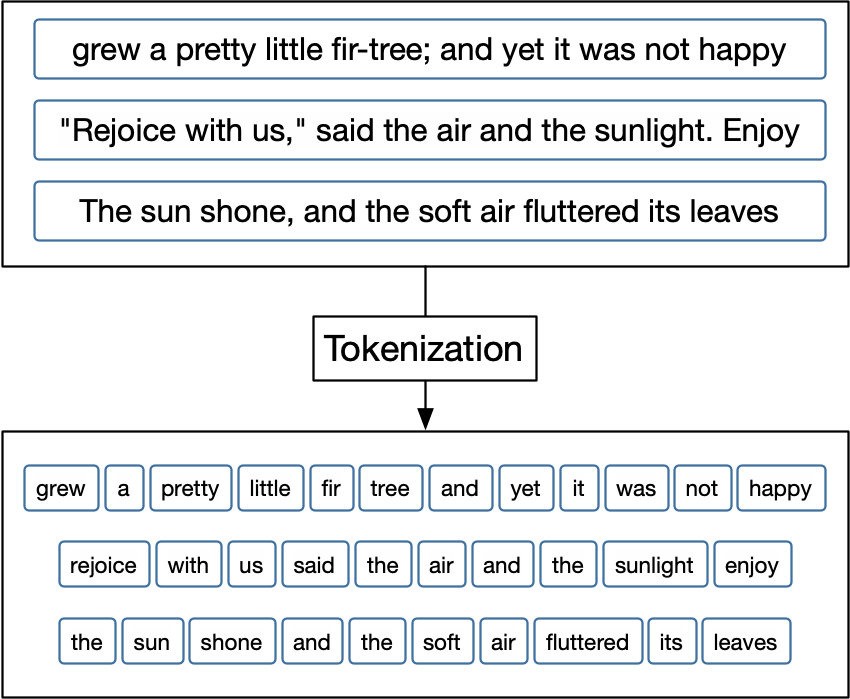
1. Tokenizer

The other key algorithm in the first section of the program is a Tokenizer.

This algorithm will be used in order to break down the text into smaller pieces which will let the pattern matching algorithm be effective. Otherwise the pattern matching algorithm will not be able to simple process the whole text.

Again much like the pattern matching algorithm there are ready algorithms that have been produced that act as tokenizers but the one that will be used in the program will have to be custom tuned to the needs of the program. Issues such as words being connected without a space as well as letters belonging to the wrong word will have to be weeded out through these algorithms.

This algorithm will also have to work sectionally, as in the entire text cannot be run through immediately as to not straight the memory of the device. This means that larger texts will have to be broken down in the separate sections. This will also be useful due to how language works in terms of sentences, missing punctuation can result in the program misunderstanding the link between different different words which will obviously result in insufficient translations. The parser will deal with these issues in order to aid the pattern matching algorithm as well as the mapping further on.



Visual demonstration of a tokenizer

1. Software solution

The program will essentially consist of the main algorithms and the main body of a program which will function as two large loops.  
 On the first loop the program will keep looping through the sentences, applying the algorithms to the sentences and once finished pushing them to a separate file with the primitive translations.

On the second loop the program will go through the text again and adjust punctuation, redundant words as well as adding the various footnotes that will be necessary to the understanding of the text as a whole.

From what I have gathered this will be the most efficient method of solving the problem that is presented. Other methods will be more problematic to implement and result in needless complications.

1. The Tokenizer

The tokenizer plays a highly important role in the program because of how it determines the flow of words that are to be accessed by the rest of the program. It’s not incredibly complex in terms of execution but the main thing that's necessary from this is reliability even with weird word structures.  
  
 It’s broken up into two separate tokenizers where one specifically splits up sentences and the other tokenizer takes those sentences and builds vectors with the specific words so the translator and the spellchecker can function properly.  
  
 It mainly works based on punctuation, special characters such as ‘.’,’,’,’:’ and so on are used to split up the sentences firstly and then the word tokenizer deals with taking those special characters and removing them from the words themselves and re-adds them back into the final text.

1. Spellchecker  
    The spellchecker was the element that took the most time to figure out because there was the most complexity involved in this element. It was built as a class so its very easy to implement at any point in the main program and makes it so the entire program is very well structured.

It initializes by preparing the keyboard graph for the dijkstra algorithm, so it doesn't have to initialize it every single time it runs. Besides that it also takes the library of correctly spelled words and puts them into a vector of sets. Isn’t that an unnecessary amount of compartmentalizing? Well no not at all actually. The entire dictionary of words in the English language being put into a single set would just cause unnecessary waste of processing power having to go through it every single time. Therefore it would be easier to split them into various groups depending on the size of the words. This makes it easier to search through them for the two main functions of the class which are the spelling corrector as well as the spell checker. And at the end of the initialization it adds the words that the user wants to not spellcheck into the library of correctly spelled words.  
  
 The two main functions that are provided by the spellchecker class are the check() and the spelling() functions.  
  
 Spelling() is the function that gets called first and is used a lot more than the check() function. This function takes the word that is given to it and compares it to the set of words that have the same length as it. This is a very efficient program due to it using the .find function that is integrated within the set data structure. And with good reason. Every single word that is processed by this program has to go through this one function, if it was to be very inefficient like looping through the entire set of same sized words, the program would take an unacceptably long time to process anything

The check() function is where most of the algorithmic heavy lifting is done. When the program detects that a word is misspelled using the spelling() function this function is called to try and determine which word the user tried to write. This function takes the word and immediately runs a modified levenshtein distance algorithm on it. The way the levenshtein is modified is by firstly adding some additional checks for the actual spelling of the word, such as if it's a number for example or a name. Secondly there is special additional tests done by the algorithm to determine if the word is misspelled in other more human ways instead of just pure transformation distance( for example maek instead of make). It firstly though will implement the mostly standard levenshtein distance algorithm on the word to determine the transformation number, after which it will take the list of words that were generated on top of some additional ones and start assigning a letter distance number on them. This is where I simply had to put in some manual bias in terms of which spelling mistakes take priority over others. It is an imperfect system due to how our spelling mistakes are very rarely perfectly consistent or sometimes even reasonably able to be deciphered. But regardless the function does its job as well as reasonably expected. After doing all of these checks it finds the word that is most likely the word the user was aiming for and pushes it back to the main function.

1. Levenshtein distance  
    It’s very common for users to simply misspell a word or two when writing applications. So when making this program I realized the need to implement a spell checking algorithm. Nothing insanely complex if a person wrote “trpst” instead of “trust” the program will pick it up and fix it so it can continue with the rest of the program.

Usually these jobs are done by AI or machine learning, but well that was not the point of my program so I couldn't rely on just some read made library to fix the text for me. So I set out to find an algorithm that would do that job for me. Levenstein distance algorithm proved to be a very good fit because of how it compares the distance of two different strings, brilliant.

The algorithm essentially gives us a levenshtein distance number which represents how similar the two words that are given to it are. The lower the distance that is returned the closer the words are in terms of transformations to each other. So if we pump “Trpst” into it and it has access to a dictionary of words it’ll show us that the closest matching word is “trust” at a levenshtein distance of 1.

So how does it work?

The levenshtein distance compares the two words. It essentially tells us how many transformations are needed within the string so the two words are the same. To do this it uses three things:

Inserting

Deleting   
Replacing

And every time an operation is used it increases the distance by 1(0 meaning that the words are exactly the same).

So if we want to explain this with examples it would go like this  
  
 Let’s take the words were already using trpst and trust. The program will go through the words and start replacing letters within trpst and find that if it replaces ‘p’ with ‘u’ it will arrive at the word trust. Meaning that the distance between the two words is 1.

Now if we add yet more mistakes to the word such as tpst for trust it will start doing all of its operations and find out that we need to replace ‘p’ with ‘u’ again, adding 1 distance. But it will also have to insert another letter in the form of ‘r’ so it can match up with trust. Meaning that we are using two operations once, so that is now a distance of 2 between the two words.

And we can obviously continue adding or removing letters so if its pst then we would have to add two characters with ‘r’ and ‘t’ before we can even start with substitution. The distance will be 3 so we can use the same operation multiple time until we get to the word.

Now that's all well and good but we need to get on to how the program actually solves levenshtein distance which is with the use of distance matrices. I will try and explain here exactly how to calculate a distance using dynamic programming. Meaning that given two different words the distance matrix will hold all the distances between all of the prefixes of the first word and all of the prefixes of the second word.

So lets go back to our example of tpst and compare it with trust. The first couse of action is finding the prefixes of tpst, which are: , t, tp, tps, and tpst. That’s four for the people counting. While trust has five which are: t, tr, tru, trus, trust. The distance matrix can calculated the distance of all the prefixes that we have laid out. This means we will have a 2x2 matrix of size 4x5, or 5x4 depending on how the words are inputted obviously

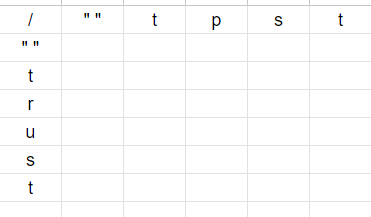
Now lets compare the prefixes. We start off with t from tpst, and we have to compare it with t which is the first prefix of trust. They are equal so as of now there is no need to do any substitutions meaning that the levenshtein distance here will be 0. Now we compare it with the other 4 prefixes.  
  
 Next up t and tr. The levenshtein distance here is determined by one addition of the letter ‘r’ to the first prefix meaning that the distance between these two will be 1. The next prefix we have to deal with is tru. Now comparing these two it’s much the same story as the previous operation where we need to add another character, ‘u’, in order to make them equal. That means that the distance between the two prefixes will be 2.

The more we continue the process the more we will be the same, with each prefix the distance will increase by 1. So the distances of all of the prefixes compared to t will be 0, 1, 2, 3, 4.

But if the first letters were different between these words this would have been different lets say if it was epst and trust the first prefix would be e which when compared with the other prefixes would pump out the values of 1, 2, 3, 4, 5. Which makes sense because we had to do an additional transformation in the first prefix due to ‘e’ not being equal to ‘t’.

Once we calculate that the program continues with calculating the rest of the values and distances between the remaining prefixes of the first string and the prefixes of the second string. It essentially builds a matrix of both of the words and then cycles through it to see the changes to output a final number of the distance that there is between the two words/strings.

Now how this works in a dynamic programming sense.  
  
 For best visual representation let's take the two words and put them in a matrix so we can explain easily how all the operations are done and how we can get the smallest amount of transformations to get from one string to another



Now we have added both words and we have to remember that we can only use the three operations as stated before, insertion, deletion and replacement. And these can be represented with this key when we are using this map

Replace, Delete

Insert, Current Operation

What this means is that we can take the value from the top left, top , or left and if the characters are the same we take the smallest one of them and add 0 due to us not using any operation. Or if the characters are different we will have to use +1 because we are changing something between the characters. Confusing? Hopefully its less confusing with this example.

3 , 4

4, 0

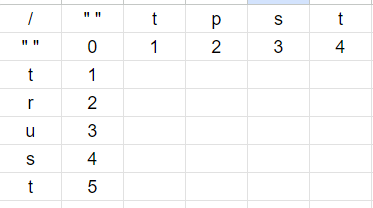
We have these values in the boxes adjacent to us, and the characters we are comparing are ‘b’ and ‘h’. Now firstly we find the lowest value from the 3 cells, which happens to be the 3 at the top left. We take that and add +1 because we are doing an operation what we get is

3, 4

4, 4

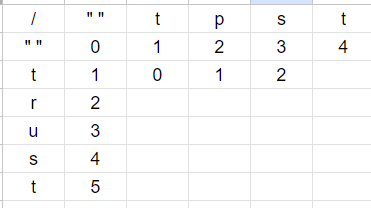
Meaning so far the shortest distance between those two characters is 4.   
  
Now we can go back to our original problem. To start off we have to fill the first column and row of the matrix to get us started. This is done by the method we described earlier.

We go through the prefixes and we get the order 0, 1, 2, 3 etc. due to how it always will require +1 on every transformation from the empty string to any other string of distance n.



Now we can start filling out the first row and the first two characters we have to compare are ‘t’ and ‘t’. This will require no transformations so we will be doing a +0 for now and also because of how the top right of the cell is a 0 we can take that and add +0.

That is a good start but now we have to fill out the rest of the row which leads us to having two characters in a row that dont match from tpst which are ‘p’ and ‘s’. Now the lowest number for the first cell will be 0, so we add plus 1 to that to get 1. After that we will do the same operation in the next row, we see that our lowest number is the 1 next to it so we get 1+1 is 2. We jot that down.



For the last character they are the same so yet again we can just simply take the lowest being two and add zero. Making the end of the first row 2.

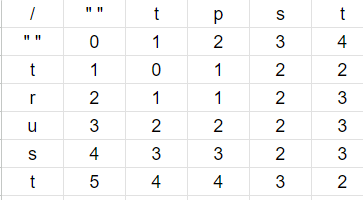
Now for the next row we can keep doing the same process. We start with comparing ‘t’ and ‘r’, they are not the same so that means we add a + 1 and see the smallest neighbour which is 0, that becomes one. For the rest of the row its all + 1s because of how ‘r’ does not show up in tpst. Therefore the table reads as follows.

We continue with the next row and again ‘u’ does not have any characters that are the same so it will be +1s across the board. The table will continue as follows.



Now when we get to the second to last row we can see that there are matching characters between the ‘s’ and the ‘s’ meaning that there will be +0 carried over in this row. The smallest neighbour it will have at that point will be a two on the top and top left therefore 2 + 0. The table will continue like this.

And, finally, on the last row there will be another + 0 with the two ‘t’ characters at the very end. Meaning that when checking the last cell we will see the lowest value in the top left being a 2. 2 + 0 will be 2 and that will be the result in the final cell.



And that will also be the final levenshtein distance between these two strings,2. Essentially the way this algorithm works is by mapping out all of the possible paths we can take when transforming the two words between each other and through that finding the path of least resistance as it were.

This is absolutely perfect for my use case due to how it can optimally see the closest words when I plug in an entire dictionary to compare against at a very efficient rate due to how simplistic the operations it does are.

I have also added to the algorithm by doing additional checks where the program will automatically swap two letters in a word to see if that can possibly make them the same( an example of this will be misspelling “trust” as “trsut”, the ‘s’ and the ‘u’ swapping places in the wrong word which is a very common mistake when writing quickly.  
  
 This algorithm as mentioned previously works along with the dijkstra algorithm which I will explain now.

5. Dijkstra’s algorithm

In 1959 a Dutch man was shopping in Amsterdam with his fiancee and when they grew tired they sat down at a cafe to drink a nice cup of coffee. And immediately he wondered “How could I get from Rotterdam to Groningen?”, a very normal thought, sure, but he was determined to solve this problem mathematically. He grabbed some pen and paper and within 20 minutes he had made some algorithm, Dijkstra looked content with his solution, little did he know this would grow to be one of the most influential algorithms in computer science as well as making him famous.

Dijkstra’s algorithm is an algorithm that calculates the shortest distance between two points on a graph. This was necessary to implement in my program due to how I need to know the distance of the keys on the keyboard for my spell checking algorithm, and this provides the easiest and cleanest solution.

To understand this algorithm we firstly need to understand graphs. Graphs are basic data structures that can be used to show connections between elements. These elements are referred to as nodes in the graph. Connections between these nodes are referred to as edges, meaning that two nodes are only connected if there is an edge between them

Graphs are very useful due to how they represent connections between different elements. Connections such as the distance between the keys on a keyboard. All of these are just nodes and distances.

Graphs can be divided into two categories:  
 Directed graphs: These are graphs where the movement between the nodes are in a single direction. That means that arrows are implemented between the various nodes to signify that it is only possible to travel in that specific direction and not bidirectionally

Undirected graph: This type of graph is simply the other version, where you can freely move between nodes in both directions without limitations.

While these are the two types of graphs there is also another type of graphs that we will be making use of in our program, those are weighted graphs. These are graphs that have a “weight” to the node connections. These weights can mean almost anything between the nodes, distance, size, time etc. And we will be using these because we need the distance between the keys to be used as weighted values.

So how do we figure out the distance between these graphs?

Glad you asked, this is the point of dijkstra's algorithm. It takes essentially as an input the starting point on the graph, and will find the shortest distance to any other point in the graph. The algorithm makes a tree of all the connections it has made from the original node to the other nodes in the graph.

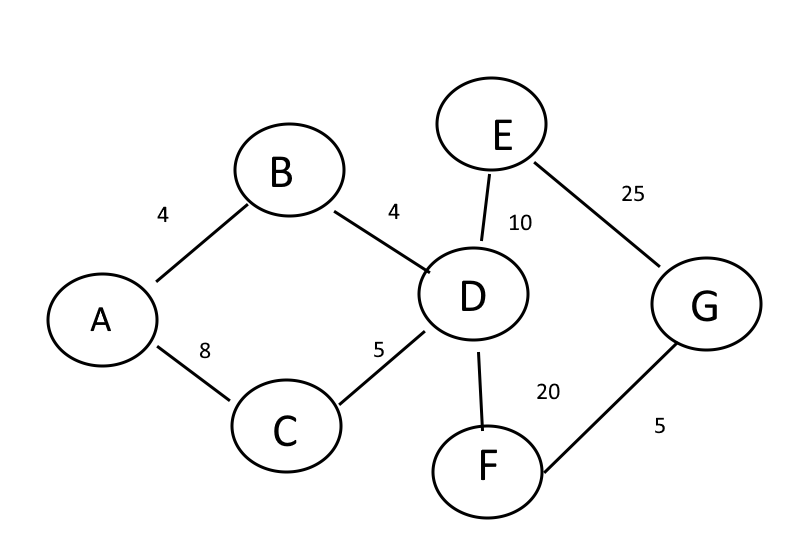
This algorithm makes use of the different distances(weights technically but i'll just say distance for the sake of simplicity from now on) on the edges to find the path that has the smallest possible distance. This is also sometimes called the shortest path algorithm.

This algorithm due to using weighted maps is not the same as an algorithm that calculates the smallest amount of nodes it is necessary to travel to to reach a certain other point, that’s a minimum spanning tree, this algorithm takes the distance of the nodes first and foremost.

Only caveat with dijkstra's algorithm is that all the distances need to be positive in order for the algorithm to work because it simply adds together all of the distances to get the final number. This won't be an issue here, however, because keys usually don't have negative distance between them. Other algorithms such as the Bellman-Ford algorithm can do this thought.

The algorithm will also keep track of what is the shortest distance that it knows of from its current location back to the source node and it will keep updating it regularly. Once it finds a shortest path between the original node and another one it essentially deems the node “marked” and adds it to the overall path.  
  
 This process keeps going until everything has been added to the path, so we know for sure which path will be the shortest from the original node to the given node we are trying to find.

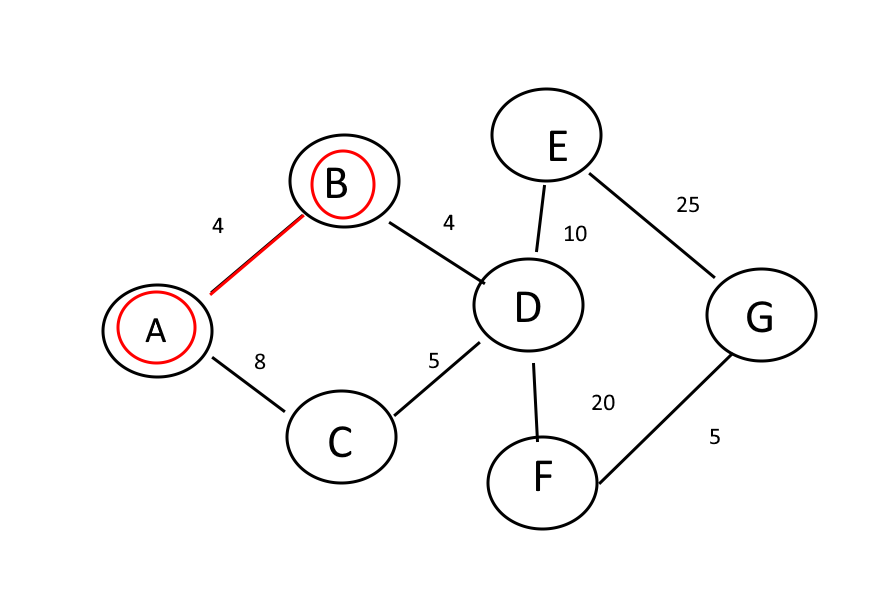
Example let's take the node A as our starting point and we want to get to G in the shortest distance possible



Firstly we have to check the distance from node A to its neighbours, which are B and C. The distance between A and the to other nodes now get added to the distance array we have stored so that means

A:0 B:4 C:8 D:? E:? F:? G:?

Now the next step is visiting the closest node which happens to be B and marking it as visited and as well adding it to the path.



We will also go to our list and put an asterisk on the B node to signify that it has been visited now.

A:0 B:4\* C:8 D:? E:? F:? G:?

So now we can take our list of nodes and mark two of them as visited

Nodes: A, B, C, D, E, F, G

Now we move on and see that we have new paths open and we have to visit a new connection that has not yet been visited. We have extended our reach to all of the nodes that are marked in red therefore we have two paths we can attempt. B -> D and a A -> C. We can now add the distance to D on our list now since we have established a path to it. C does not need updated due to it already being visited by us.

A:0 B:4\* C:8 D:8 E:? F:? G:?

The reason we have put 8 as the distance to D is because of the nodes we have already visited on our path so far, that being A -> B and B -> D. When we combine the distance that was taken between those nodes we get 4 + 4 being 8. Or so far the shortest distance we know of to D.

Now our next step will be defining which node we will travel to next which is determined by the lowest distance to it. The node with the lowest distance here happens to be C with 8.

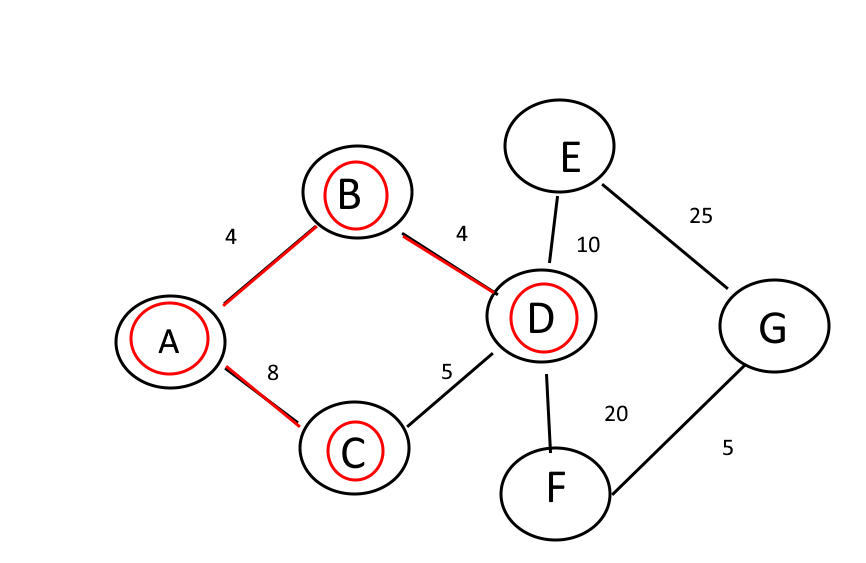
Now we can see that we have found a new path to D, which is A -> C then C -> D. But this distance is 8 + 5 which is 13. When we compare the different paths to D we can see that the A -> B -> D path will be shorter at 8 so we will not change that path. Now we mark C as a visited node.

A:0 B:4\* C:8\* D:8 E:? F:? G:?

And also

Nodes: A, B, C, D, E, F, G

Now we can continue with accessing the next node which happens to be D. We will traverse to it with the shortest path we have found to be A -> B -> D.



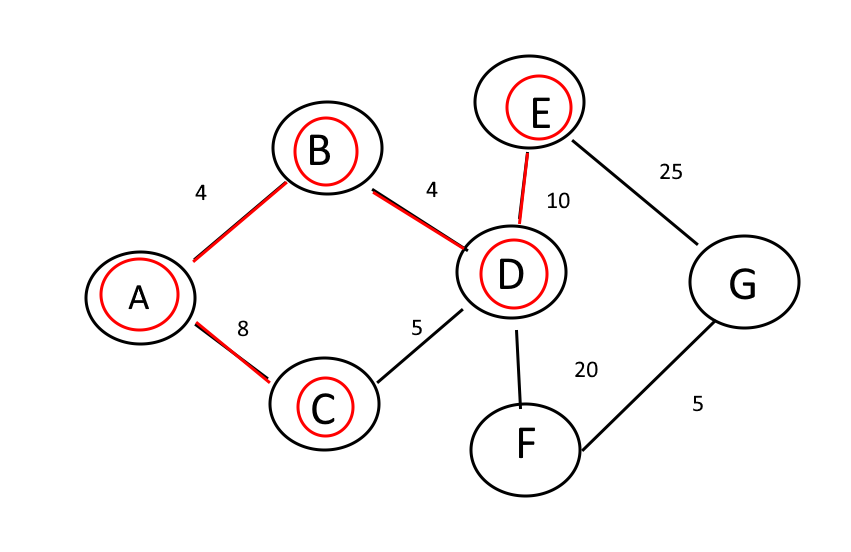
From here we can spot two new paths to two new nodes we have not encountered before that being E and F. so we can begin by updated our visited nodes as well as the distances

A:0 B:4\* C:8\* D:8\* E:18 F:28 G:?

And

Nodes: A, B, C, D, E, F, G

We can write down our known distances of E and F based on the previously explained addition between the known paths so far and the program will now decide which new node to go to. It will be the E node as it is the shortest distance that we know.



From here we have found our last node which is G, so we can add it to our list.

A:0 B:4\* C:8\* D:8\* E:18\* F:28 G:43

And

Nodes: A, B, C, D, E, F, G

But now the program even though it sees the new node G it cannot go to it yet because the distance to it is smaller 28<43 therefore it has to backtrack and go to the not yet visited node of F.

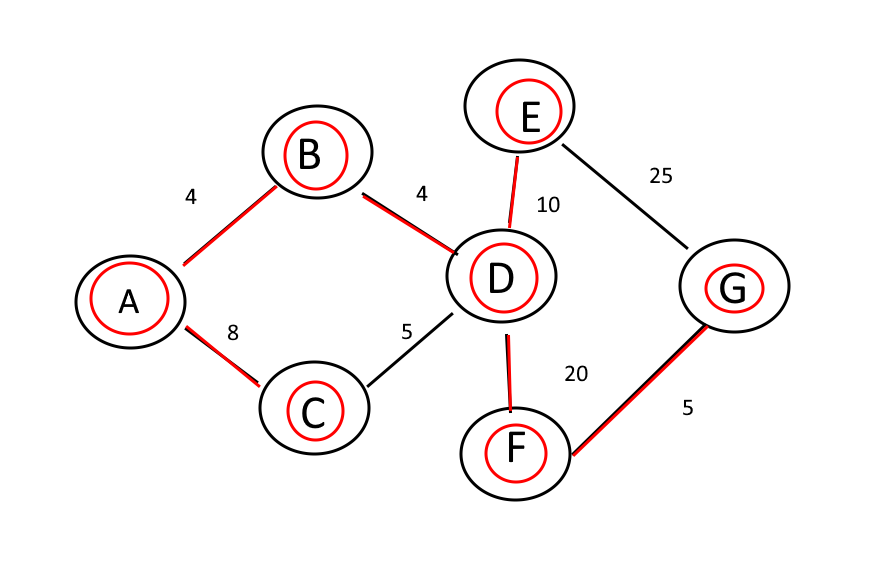
So it visits F and we have to update our lists again

A:0 B:4\* C:8\* D:8\* E:18\* F:28\* G:33

And

Nodes: A, B, C, D, E, F, G

And finally with that we haven found the shortest distance to G by virtual of having explored all of the nodes in the graph so far. And we can visit G and update our list for the very last time



A:0 B:4\* C:8\* D:8\* E:18\* F:28\* G:33\*

And

Nodes: A, B, C, D, E, F, G

And now that we have everything filled out we can also determine the shortest path from node A to any single node on the graph but key to us is the fact that we have the shortest distance to the node G which is what we were looking for.

Now how does this apply to the program I have built? Well as you can see the graph is already kind of looking familiar to a keyboard with the distances and everything, so that's simply what i did, I made the entire QWERTY keyboard into a graph so I can at any moment find the shortest distance between any two letters when I am doing spell checking.

It’s very easy to see the similarities between a dijkstra specific graph and just a regular keyboard. And this also proves to be a very efficient way of roughly estimating how a person could have messed up their keypressed and in what region of the keyboard which is exactly what I need for this specific algorithm.

Implementation

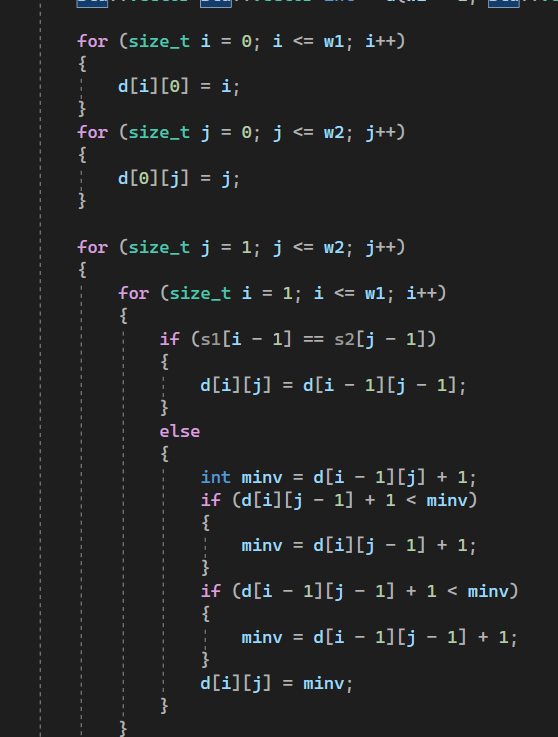
The majority of the program was developed in C++. I chose this programming language for a few reasons.

C++ being as old as it is, developed in 1979, its main appeal is the fact that it was made to be very efficient even for the systems back then. So plugging it into a machine today runs significantly faster than most other scripts. And considering how long it has lasted so far I am sure it is not going away anytime soon.   
 Firstly, it is a very efficient and fast script. For programs such as this one, reliable speed was my top priority, so a not very bulky script like c++ was perfect for this occasion. A lot of the elements such as maps and sets were very simplistic therefore reliable and quick.

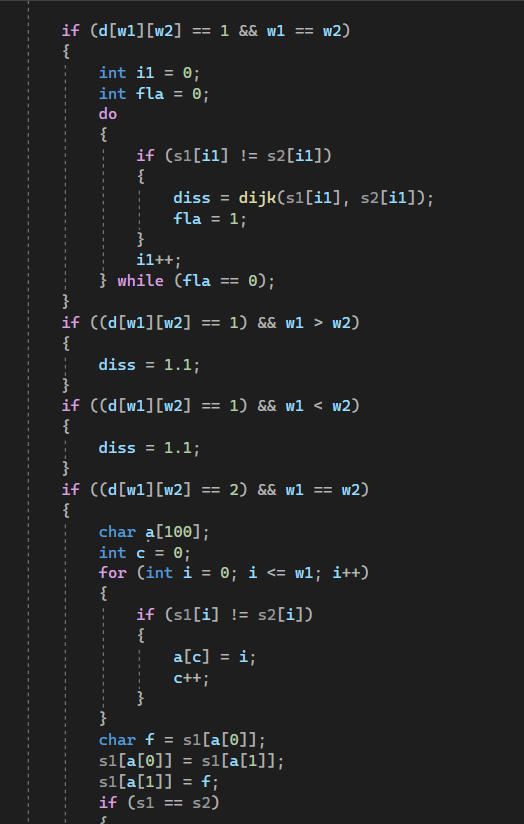
Secondly, I am simply comfortable with this language and I knew if I was to make a senior project it would have to be in an environment I can trust. While it’s quite boring to not experiment for a larger project, it is still necessary to know the tools you are using to sculpt something of quality.

The front end of the program was developed with C# for much the same reasons. Comfort and reliability. While I didn’t expect to put in any front end I realized it was necessary for the user experience to add that feature.

1. Implementing algorithms

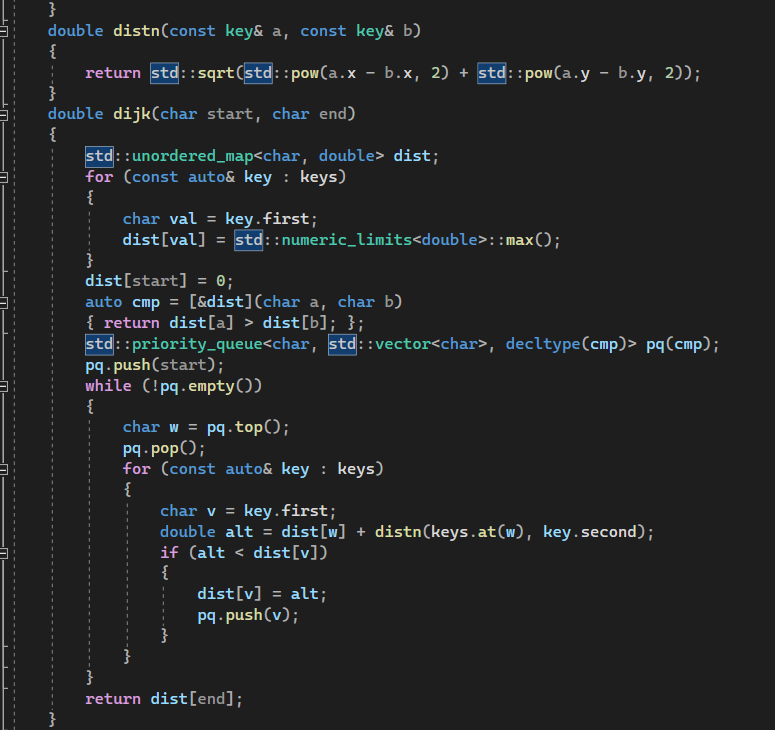


This was the most important implementation in the program the way levenshtein was implemented. I mostly started off standard with it. It’s not insanely different from other levenshtein distance algorithms for this specific part, I did use strings though because of problems I was running into when I would use char arrays so I scrapped that. The following piece of code is what starts setting it apart from normal levenshtein distance algorithms

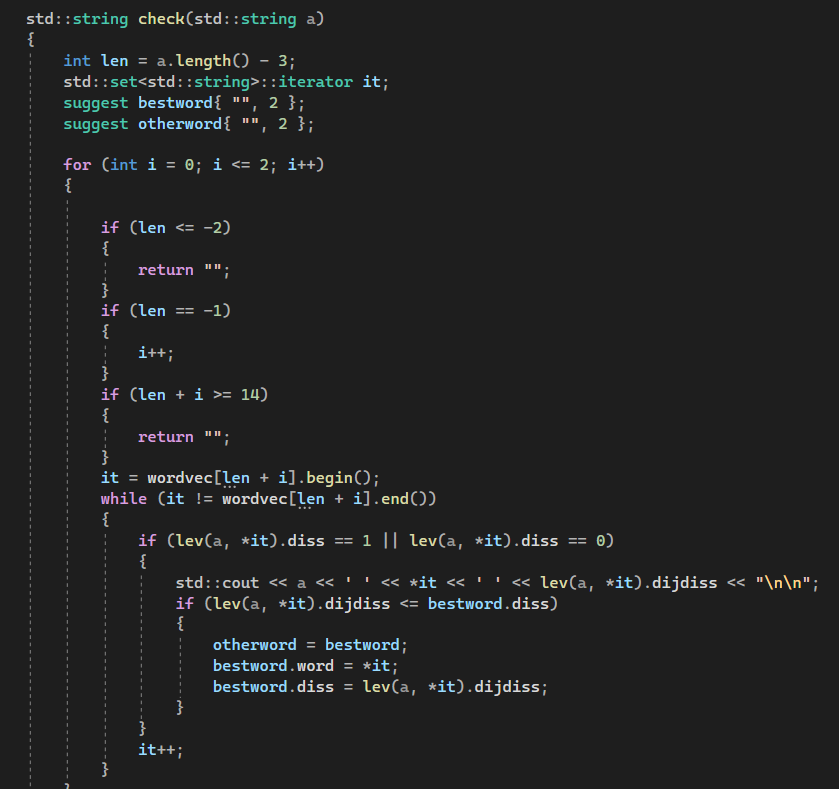


What this specific piece of code does is that it takes the values that were put out by the initial levenshtein distance and it does additional checks about the words to firstly find out the dijkstra letter distance on some of them but on others it does a check for other potential misspellings of the word and adapts accordingly to give it a fair estimate on how close it actually is relative to the misspelling.

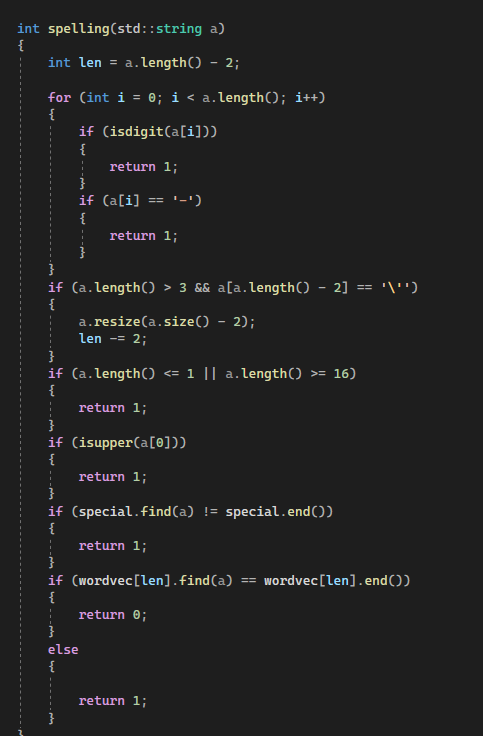
The next part of the complex algorithm implementation will be the implementation of the dijkstra within the leveshtein which looks like this



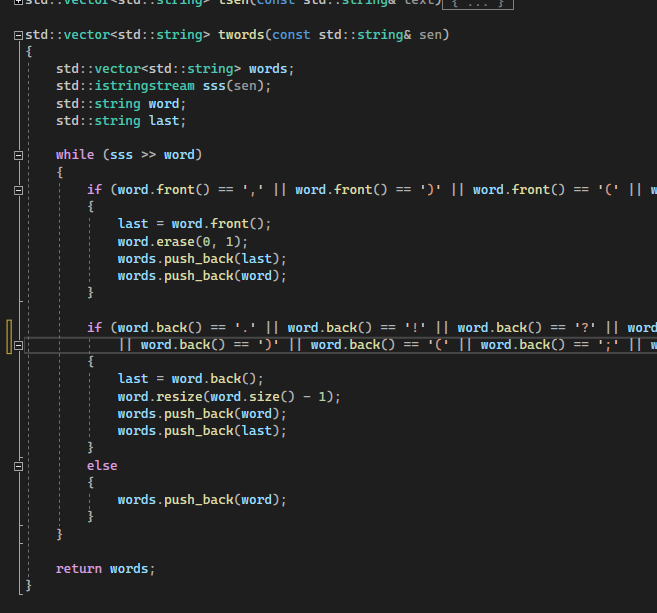
This code now takes the words that were given to it by the levenshtein estimate and begins to track the nodes from those words(the misspelled letter vs the letters around it). It keeps pushing out the proximity of the misspelled word to the actual words until it has gone through all of them and then the program determines which of them has the highest chance of being the misspelled word via the letter distance.



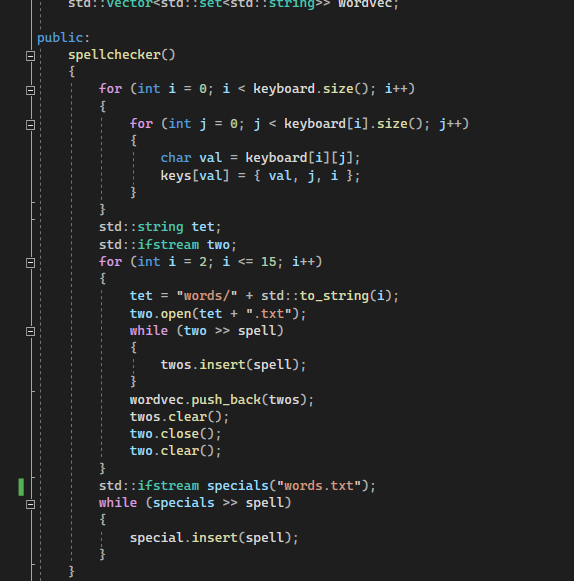
This is the function for the general checking of the words and then correcting them. It is the communication between the main function of the program as well as the back end algorithms. It essentially takes as input the misspelled word and then uses the various algorithms to determine which word is most likely based on the process i described above. It also prints out the words and their values just in the console mostly for control purposes.



This is the function for checking the actual spelling of the words themselves. At its core it simply performs a search in the set of words with the same length that the given word is, but on top of that it does a fair amount of additional checks to make sure the word is not a number or name and so on. It's crucial for this to be a bit more relaxed with what it deems to be a misspelled word because if it lets too many words be flagged as misspelled when they are simply either numbers or another word that does not need autocorrect it will decrease the time efficiency of the program as well as simply the accuracy of the spellchecker.



This is the tokenizer for the words, which simply goes through every word in the document and parses them so the rest of the program can work effectively. It is not too complex this implementation but because of how standard punctuation is in the english language this is more than effective enough to parse the words correctly for the algorithms to work on.



This is the initializer of the spellchecker class, it gathers all of the files with words to be spell checked and initializes them into a vector of sets for later use by the various algorithms. It also initializes the keyboard graph for use in the dijkstra algorithm later on so it doesn't have to initialize it every single time the algorithm is used. And it also puts the words not to be checked by the spellchecker in a set to be used later on.

1. Testing

Testing on this project was fairly straightforward, once the first versions of the program are established, simple sentences will be pumped into the system. These will allow me to test how it handles different types of sentences and different types of complex english(essay, legal documents, etc.) After I receive the output from the program I will see if I can read the simplified version clearly and without any issues. Naturally the first few versions of the program will be quite flawed and result in rapid improvement to the algorithms as well as the dictionaries and their mappings.

This process continued until the results begin to be more satisfactory for the end user. This step was very hands-on, translation is not the most exact of sciences sometimes, especially when NLP libraries are not used. For the most part it was simply me figuring out the best way to translate words as well as calibrating the program to output in a cohesive way.  
 This step was also important in understanding the spellchecker. The first few versions had a very basic spell checker that ended up being very unsatisfactory. Because while it could mostly tell which words were close to each other it simply did not know how to differentiate them on a more nuanced level. This proved frustrating because of how applying weights and biases to essentially wrong input to improve it is a very fiddly task. But eventually I settled on using the dijkstra algorithm to try and provide a nuanced and better guesstimation for the spell checker. Even after it still gets some misspellings wrong, but that is to be expected when there is no context given to the algorithm besides a wrong word and an entire dictionary worth of words to figure it out from.

I also inputted text from as many various places as I could, blogs, white papers, messages from friends. This was important due to how it has to be fairly adaptive. While it will still produce some unsatisfactory results if the input is too messy it behaved very well when the text was at least somewhat coherently presented. A lot of clauses had to be added to the word checking functions due to inconsistent punctuation as well as special symbols.

Possibly the biggest improvement that was taken from testing was just the expansion of the translation dictionary, while it's not all encompassing, it shows off that the code that it has behind it works. If I was to continue working on this project the first course of action would be to expand the word list even more due to how It’s still somewhat limited.

In terms of glitches in the actual algorithms themselves I did not run into a massive amount of issues in testing because of how I programmed very compartmentally when making all of the parts.   
 To elaborate I would open up new projects and develop and scrutinize the smaller components extensively before adding them to the main program. This resulted in a very smooth operation because there were almost never issues after I would plug in the functions I had been working on separately. While this might have been slightly more time consuming in the short term because I would ignore the main program a lot and work on what felt like side projects a lot, in the end it saved a lot of headaches because of how I could easily refine any element by itself before adding it to the main program proper. This in the long term would save time because I could almost always immediately tell what was going wrong in the program and fix it quickly before it had time to corrupt any other part of the program.

The final way I would compare once I had all the code mostly finalized was to put it side by side with online translators that cover the same area. These translators are already highly developed and advanced so being able to see how they would do the same job as my program gave me a new perspective so I could tinker more with the issue and produce a more coherent result.   
 Overall with all of these extensive testing methods at work the program turned as functional as I expected it to be, whilst adding some features that I didn’t even consider at the beginning that were interesting to implement and work with (such as the additional algorithms in the spellchecker).

1. Challenges

The main challenges that will have to be dealt with in this project is the issue with the large dataset of words that will have to be processed through in order to ensure the program can run smoothly. This required some manual labor in order to be satisfied with the end product.

Most of the other challenges were ensuring the efficiency of the algorithms that will be used within the program. Considering a lot of the algorithms will run a considerable amount, if they seem to be needlessly inefficiently this could make the program very cumbersome.

Another challenge will be picking out exactly what sort of simple english dictionary/ mapping will be used in the final product. There is potential for the program to have access to multiple forms of translation(Ogden's, simple english wikipedia etc.), secondary translation libraries could be an option yet refining one and getting it to a good level will be my main goal throughout this project.\

The biggest challenge mostly was the lack of access to NLP libraries.

NLPs are Language models that are used to accurately guess next words and general word structure based on analyzing data that is fed to it. They are essentially giant language specific machine learning algorithms.

The complex algorithms inside these NLPs are the things that create the rules for the natural language they are analyzing. Their model develops and prepares itself by learning the characteristics of a language that is fed to it. With this ability is able to understand the meaning behind phrases and even correct mistakes, predict upcoming words.

When training these types of models usually the more data the better. The larger sample size they have access to the more rules they can construct and the better they can understand the exceptions that are often unknowingly made in natural human language.

I purposefully avoided them, even besides the no libraries requirement, because I wanted to understand the finer and more mathematical side of language processing and the difficulties that come with it, and there are a lot

The main disadvantage I kept running into is simply that the translator i was building was very simplistic in its nature. It cannot analyze the text given to it like a human or a very advanced AI would, it can simply take the words and put them through mathematical models to try and give a satisfactory answer. While it certainly does the job it was tasked with, when programming I couldn't help but wonder if it would be easier if I could just plug in a machine learning algorithm to simply automatically figure out the issues within the text and just solve them. But I did understand the whole process on a more intimate level with this restriction which was the main goal of this project.

While this project can be somewhat primitive with the methods used in the program, It will still be functional. Although, if the project were to continue outside of this scope it would include a lot of interesting technologies that are used today in more complex language translation applications.

The bleeding edge of translation technology today uses a lot of machine learning to understand the patterns between different languages and translate very naturally between them. Usually this is done by providing a massive sample size in both languages of the same text and letting the various ML algorithms understand by themselves. This could be an interesting direction to take this project by building or using existing ML models to see the patterns between simple english as well as complex english. Although that at this current moment is out of the scope of the project. The potential within this field is fairly limitless and leads directly into very advanced AI.

1. Conclusion

In conclusion this was a very good learning experience in terms of development and implementation. The goal of the project at least for me was avoiding using the machine learning algorithms to see if I could break them down to some degree and develop my own solutions for this unique challenge. Don’t get me wrong, machine learning is the correct path for this specific problem. But i wanted to know why, i wanted to understand how the complexity of language and its errors can be worked on with more traditional let's say programming methods.

As a disclaimer, this program is a student project developed to explore technologies, it is not the most efficient, or the best program at doing what it does, but that's not the point. It is made to further prove the efficacy of NLP models as well as in general the use of machine learning in this field. The issue of language is immensely complex and almost incomprehensibly difficult to try and tackle with simply using algorithms and static programming methods. There is a lot of nuance when it comes to spell checking, translating and fixing human speech that, to some degree, is impossible to solve algorithmically because of how it is essentially brute forcing rules onto something that does not adhere to rules all time. Language is dynamic and alive, and the only way to deal with that is to implement a system that is dynamic and alive as well. This is where machine learning comes in, making models that can learn from this chaotic sphere and learn to replicate it accurately will be the only way we will tackle this problem effectively.

NLPs are already massively complex and, most importantly, functional. Translators and spell checkers such as DeepL, grammarly, or even things such as ChatGPT are tackling these issues successfully using these technologies. Issues which algorithms written on paper in 20 minutes could never keep up with. And they keep improving, the next years ahead of us are very exciting in this field, AI models are improving exponentially, and so is language processing along with it. AI models have passed the turing test, there is no going back now. Problems like the one I was attempting to solve will become simply a trivial exercise for these all encompassing solutions. And these are the exact issues they will excel in, problems where the solution is very interconnecting with the context of the issue itself.

But that isn’t to say the program I developed was useless or a waste of time, quite the opposite. By attempting to solve these complex problems with these methods we learn what the actual path to success is, it's not inefficient, it's simply laying the path for something greater. The algorithms for example I have used are not throw away algorithms, they still have large real world implications and uses, and on top of that they were used as some of the building blocks to get to the AI revolution we are being faced with today. Knowing the limitations of a certain method is incredibly valuable knowledge. We can see where something excels and where it staggers behind.

And how this solution can be improved in the future, well it has to be rebuilt, NLPs are necessary for a solution like this. Training a model on complex english and teaching it to dumb it down as well as spell check on the site will be the ideal solution for this problem. As much as I wanted to prove myself wrong that it was possible to take a different approach to this issue, the answer was very much in front of me the whole time.   
 The program works in a satisfactory manner for me. It has all the features I wanted to implement from the beginning and it behaves in a coherent way whilst retaining efficiency. Which might be the only advantage of a very simple solution such as this one. It will run faster than a machine learning algorithm, it uses its resources sparingly, so potentially for absolutely massive pieces of text it could, while of lesser quality, do the job quicker. And that’s a lesson for me at least for the future. While AI will take over this sector in terms of quality of execution there will always be a need for programs that have a quick and reliable system behind them. Doing a basic spell check for a single word doesn’t necessarily need a NLP model that has been trained for 3 years to execute, simplicity counts for something.

Overall I’m happy with the execution and the program that was developed, I learned a lot of things that extend beyond the function of the program. Things that will help me further my career in this field. There is always a need to experiment and do things differently, without it we would never know what the right path is.

Resources

[1]http://ogden.basic-english.org/words.html

[2]<https://www.systransoft.com/download/white-papers/systran-white-paper-PNMT-12-2016_2.pdf>

[3]<https://cpsc.yale.edu/sites/default/files/files/technical-reports/TR17%20Linear%20Pattern%20Matching%20ALgorithms.pdf>

[4]<https://www.ibm.com/topics/natural-language-processing#:~:text=Natural%20language%20processing%20(NLP)%20refers,same%20way%20human%20beings%20can>.

[5] <https://strathprints.strath.ac.uk/2611/1/strathprints002611.pdf>  
[6] <https://www.scaler.com/topics/data-structures/dijkstra-algorithm/>

[7]<https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm>

[8]<https://www.kdnuggets.com/2020/10/optimizing-levenshtein-distance-measuring-text-similarity.html>

[9]<https://smltar.com/tokenization.html>

[10]https://copylists.com/words/list-of-5-letter-words/