Matthew Cummings
November 7, 2021
CISS 247 Computer Systems
Assignment 2 Report

Introduction
Assignment 2 tasked us with writing a basic command line spellchecker in C. The goal is to find the closest words in a wordlist as defined by their "Hamming distance" to what the user has typed. If the inputted word was found in a wordlist, then it is assumed to be spelled correctly. When the word is not found in the list, the first 5 closest words found will be shown as suggestions. Specifically in this assignment we were also required to implement 2 helper functions, one to load the wordlist from a file into an array of strings, and another function to find the Hamming distance between two words passed to it.

Process
In general, my implementation of the spell checker iterates over the whole wordlist, looking for any words close to what has been typed. Any words of different lengths are immediately iterated over. Then, equal length words are passed to my findHamm() function to return the Hamming distance between both words. If the distance is found to be smaller than the previously found minimum, the index to this newly found closest word is saved as the only close word. When a word of equal distance is found, its index is added to the array of close words. Words of distance greater than the currently known minimum distance are ignored.

In the special case when findHamm() returns 0, this indicates that an exact match for the entered word was found in the supplied wordlist. Once this new minimum distance of 0 has been found, the code stops looping over the list, tells the user their word is spelled correctly, and prompts for a new word.

```c
// determines hamming distance between 2 words
// https://en.wikipedia.org/wiki/Hamming_distance
int findHamm(char word1[], char word2[])
{
    // get length of words to loop over
    // both strings are assumed to be of same length
    int wordlen = strlen(word1);

    // tallied distance between both words
    int dist = wordlen;

    //printf("word1: %s\n", word1);
    //printf("word2: %s\n", word2);

    // for every character in both strings, add 1 to dist tally
    // per matching character
    for (int i = 0; i < wordlen; i++)
    {
        if (word1[i] == word2[i])
        {
            dist--;
        }
    }

    return dist;
}
```

This findHamm() function is the most integral to the program's execution. Whenever a word of equal length to the inputted word is found, this function is called to determine the Hamming distance between the 2 words.

All useful computation is accomplished in the for loop. For every character in each word that matches, the counter "dist" decrements by 1. By the end of this for loop, the "dist" variable is equal to the Hamming distance between the 2 supplied words.

However, before the Hamming distance between 2 words can be calculated, the wordlist must be filled into an array of Strings, accomplished by loadWL.

```c
// function to load word list into given array
int loadWL(FILE *input, char wordlist[][MAX_WORD_LEN])
{
    // fill wordlist[][] array of strings
    int i = 0;
    while (fgets(wordlist[i], MAX_WORD_LEN, input))
    {
        // trim tailing /r/n from word
        strtok(wordlist[i], LINE_END);
        i++;
    }

    // return length of wordlist
    return i;
}
```

Function loadWL() fills the wordlist[][] array of strings with lines from the file specified by the user as a runtime argument. Additionally, loadWL() returns the complete size of the wordlist. This size is used to ensure the main loop does not iterate past the end of a supplied wordlist.

Here fgets() is used to get each line out of the file, and strtok() is used to trim trailing line-ending characters like '\n' and '\r'. Without strtok, these line ending characters would have to manually be omitted.

Much of the logic of my spellchecker is performed by multiple if else-if else statements within the main while(true) loop. This is required to ensure the program keeps running checking for the next word in the list, as well as ask for the next word to check. Organization of the conditional statements could be improved for easier code legibility, but, overall, the logic works.

```c
// move on to the next word if the current word isn't the same
// length as the user entered word
while (strlen(word) != strlen(wordlist[i]))
{
    i++;
    if(i == listLen - 1)
    {
        foundClosest = true;
        break;
    }
}
```

First is a while() loop that skips over all words in the wordlist that do not have the same length as the inputted word. For every word not of equal length, the wordlist index is incremented by 1.

There needs to be a specific check here to make sure the wordlist index does not get incremented past the end of the list. Without this check, the word ends up being compared to itself and the program thinks it has been spelled correctly. Also, foundClosest is a Boolean value to skip some conditionals if we have found the closest word in the list by reaching the end or finding the exact word.

```c
// certain things need to be set if we found new closest word
if (diff > currDiff && !foundClosest)
{
    // update diff to new min diff
    diff = currDiff;

    // if the diff is 0 we have found the word in the list
    if (diff == 0)
    {
        foundClosest = true;
    }

    // clear out old top 5 closest
    for (int j = 0; j < 5; j++)
    {
        closest[j] = -1;
    }

    // reset index for closest[]
    index = 1;

    // set new closest
    closest[0] = i;
}
```

Here, diff is the minimum Hamming distance currently known between the entered word and the contents of the wordlist, and currDiff is the distance for the current word being examined from the list. This conditional tests if the currDiff is smaller than the known minimum diff. When a new minimum diff is found, diff is updated, as well as the array of closest indexes. In closest[], -1 signifies an empty cell that does not contain an index pointing to a word.

```
// certain different things if we've found yet another word
// with the same matching minimum hamming distance
else if (diff == currDiff && !foundClosest)
{
    // make sure we haven't already found the 5 closest words at this diff
    if (index < 5)
    {
        closest[index++] = i;
    }
}
```

If the program has found a new word matching the previous diff, the index to this new word is added to the array of word indexes closest[], but only if we have not already filled the array. This ensures only the first 5 close words indexes are stored.

```
// if diff=0 this is special case where user word was found
// in the given wordlist so we need to break out
else if (currDiff == 0)
{
    foundClosest = true;
    diff = 0;
}
```

This code block handles the case when the user entered word has been found in the list (i.e., when the Hamming distance is 0). In this case, foundClosest needs to be set to true and the minimum diff must be updated to 0.

```
// else is only triggered when currDiff > diff
if (!foundClosest) i++;

// make sure we haven't already iterated to the end of the wordlist
if (i == listLen)
{
    // need to stop iterating loop if it gets to end of list
    // otherwise infinite loop if the user word is not found
    // in the wordlist
    foundClosest = true;
}
```

Here and only here is the wordlist index incremented. After incrementing, the index is checked to make sure it has not incremented past the length of the list. I am not sure an additional bounds check is necessary here, since the while() loop above also checks the bounds, but this double checks that the wordlist index does not get incremented past the wordlist length.

```
// if we've found the closest word(s) or the actual word in the list then tell the user
// and prepare for the next iteration of the loop
if (wordlen > 0 && foundClosest)
{
    // if diff is 0 then word is correctly spelled
    if (diff == 0)
    {
        printf("The spelling of \"%s\" is correct.\n", word);
    }

    // else the word is incorrectly spelled and we need to print suggestions
    else
    {
        printf("The spelling of \"%s\" is incorrect. ", word);
        printf("Here are some spelling suggestions...\n");

        for (int i = 0; i < 5; i++)
        {
            if (closest[i] != -1)
            {
                printf("%s ", wordlist[closest[i]]);
            }
        }
        printf("\n");
    }
```

Now this code only executes once the exact word has been found in the list, or end of the list has been reached. When the exact word as been found (diff==0), then the program says as much. Otherwise, the program will print the closest words found from within the list.

To print all close words, the closest[] array of indexes is iterated over, and for every index that is not -1 the word pointed to by the index is printed. Instead of using -1, NULL could have been substituted, but -1 was considered easier to implement.

```
// update word
printf("\nPlease enter a new word... ");
fgets(word, MAX_WORD_LEN, stdin);

// update wordlen
wordlen = strlen(word) - 1;

// exit if no word was entered
if (wordlen == 0) return 0;

// set word to lowercase
for(int k = 0; word[k]; k++){
    word[k] = tolower(word[k]);
}

// trim newline from new word
word[wordlen] = '\0';

// reset diff
diff = MAX_WORD_LEN;

// reset closest[]
for(int j = 0; j < 5; j++) {
    closest[j] = -1;
}

// do some formatting to make it pretty
printf("\n");

// reset wordlist index to top of list
i = 0;

// reset foundClosest boolean
foundClosest = false;
```

At this point in execution the first word entered has already been either corrected or confirmed to already be correct, so now the program must ask for a new word to check. Library function fgets() is used, same as before, to get this new word from the user. If no word is entered (wordlen==0), then the program exits.

Many variables must be updated for the new word being checked. For example, diff must return to the starting value of 40, or MAX_WORD_LEN, and the closest[] array needs to be reset to -1 for all indexes.

Re-assigning all these variables like this seems redundant, as they are set like this in the exact same way before the main while(true) loop runs, however, the loop could not be updated to process a new word without these variables being reset. Conceivably a function may be called to accomplish the same task, but this seems more complicated than worthwhile.

Testing

Test cases were taken from the Canvas Assignment page, as well as words discovered as edge cases while writing the program. Words to test are apple, apfel, asdf, terminatex, zyzzyvas, and a. Words “zyzzyvas” and “a” were selected to test the top and bottom of the list, “apple” was chosen as a known correct word, and the rest were chosen to test spelling suggestions of the first 5 closest words.

Results
./spellcheck wordsEn.txt runs as expected.

Conclusions
Finally, I was able to correctly pass arrays between functions and modify them within a function, as it would be impossible to implement the loadWL() function without being able to do so. Still, my coding style could be improved by breaking up the code into different functions more, but I am pleased that this program works as intended. The logic works, but the excessive conditional statements could be simplified and improved, as currently the conditional trees are a mess to follow, but that seems to be something that is picked up through experience.