Assignment 4

Takunda Takaindisa

April 2024

1 Appendix

1.1 mySA.h

Listing 1: mySA.h

```
//header file, declare functions and structs

#ifndef MYSA_H
#define MYSA_H

#define MAX_WORD_LENGTH 50
#define SIS_ARRAY_SIZE 10
#define BUFFER_SIZE 2048

typedef struct {
    char *word;
    float score;
    float SD;
    int SIS_array[SIS_ARRAY_SIZE];
} words;

words *rLex(const char *filename, int *size);
void freeLex(words *lexi, int size);
int vRead(int argc, char *argv[]);
float do_sent_score(const char *sentence, words *lexi, int size);
int sentiment(int argc, char *argv[]);
char *dup_str(const char *s);

#endif
```

1.2 Makefile

```
CC=gcc
CFLAGS=-Wall -g
TARGET=mySA


all: $(TARGET)

$(TARGET): mySA.c
	$(CC) $(CFLAGS) -o $(TARGET) mySA.c

clean:
```

```
	rm -f $(TARGET)

.PHONY: all clean
```

1.3 mySA.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "mySA.h"

#define MAX_WORD_LENGTH 50
#define SIS_ARRAY_SIZE 10
#define BUFFER_SIZE 2048

int vRead(int argc, char *argv[]); // Forward declaration of vRead
int sentiment(int argc, char *argv[]); // Forward declaration of sentiment

int main(int argc, char *argv[]) {
    // Check # of CLI args
    if (argc == 2) {
        // If 2, run vRead
        return vRead(argc, argv);
    } else if (argc == 3) {
        // If 3, run sentiment
        return sentiment(argc, argv);
    } else {
        // If the number of arguments is neither 2 nor 3, print usage message
        fprintf(stderr, "Usage: %s <path_to_vader_lexicon.txt> [<path_to_sentences.
            txt>]\n", argv[0]);
        return 1;
    }
}

/*
If argc = 2 then run Vread function (reads the vader_lexicon.txt file)
iF argc = 3 then run sentiment function (returns line by line sentiment analysis)

Reminder == usage = ./mySA <vader_lexicon.txt> [<validation.txt>]
*/

// function reads lexicon file and fills up words structure
words *rLex(const char *filename, int *size) {
    FILE *file = fopen(filename, "r"); // Openfile for read
    if (!file) {
        perror("Sorry, could not open the file"); // file open fails
        return NULL;
    }

    int bucket = 10; // Initial bucket for the array of words
    words *lexi = malloc(bucket * sizeof(words)); // Allocate initial memory
    *size = 0; // Initialize size of array to zero

    char line[512]; // temp location to hold each line of the file
    while (fgets(line, sizeof(line), file)) { // readlines until EOF
```

```
        if (*size >= bucket) {
            bucket *= 2; // increase capacity if needed
            lexi = realloc(lexi, bucket * sizeof(words)); // Reallocate memory
            if (!lexi) {
                perror("Check memory allocation as there is a failure somewhere.");
                    // print error Msg if reallocation fails
                fclose(file);
                return NULL;
            }
        }

        // tkenize line struct parameteres (word,polar,sd,score)
        char *token = strtok(line, "\t");
        lexi[*size].word = dup_str(token); // make copy of word
        token = strtok(NULL, "\t");
        lexi[*size].score = atof(token); // score to float
        token = strtok(NULL, "\t");
        lexi[*size].SD = atof(token); // SD to  float

        // Pars SIscores, assume theyre full
        for (int i = 0; i < SIS_ARRAY_SIZE; ++i) {
            token = strtok(NULL, ",\n"); // Tokenize every score
            if (token != NULL) {
                lexi[*size].SIS_array[i] = atoi(token); // Convert score to int
            }
        }

        (*size)++; // Increment the size of the lexi array
    }

    fclose(file); // Close vader_lexicon file
    return lexi; // Return the populated lexi
}
// We need to free dynamically allocated mem for lexicon
void freeLex(words *lexi, int size) {
    for (int counter = 0; counter < size; ++counter) {
        free(lexi[counter].word); // Free each word string
    }
    free(lexi); // Free lexi array itself
}

int vRead(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <path_to_vader_lexicon.txt>\n", argv[0]); //
            Print if CLI argument is not correct
        return 1;
    }

    int lexSize = 0;
    words *lexi = rLex(argv[1], &lexSize); // Read lexicon from file
    if (!lexi) {
        return 1; // If reading failed, exit program
    }
 /*
    for (int loop = 0; loop < lexSize; ++loop) {
        printf("Word: %s\n", lexi[loop].word); // Print word
```

```
        printf("Score: %.2f\n", lexi[loop].score); // Print  polar_score
        printf("SD: %.2f\n", lexi[loop].SD); // Print std_dev
        printf("SIS: ");
        for (int j = 0; j < SIS_ARRAY_SIZE; ++j) {
            printf("%d ", lexi[loop].SIS_array[j]); // Print SIS_array
        }
        printf("\n\n");
    }
*/
    freeLex(lexi, lexSize); // Free allocated mem
    return 0;
}

// needto calculate the sentiment score of a sentence
float do_sent_score(const char *sentence, words *lexi, int size) {
    float overall = 0.0;
    int wordC = 0;
    char *sentCopy = dup_str(sentence);
    char *token = strtok(sentCopy, " ',.-\n");
    // Iterate through each token
    while (token != NULL) {
        // Convert token to lowercase
        for (int n = 0; token[n]; n++) {
            token[n] = tolower(token[n]);
        }

        // Compare token with words in lexicon
        int present = 0;
        for (int counter = 0; counter < size; counter++) {
            if (strcmp(lexi[counter].word, token) == 0) {
                overall += lexi[counter].score;        // Add sent score to sum
                present = 1;
                break;
            }
        }

        // If word not found asssign 0 score
        if (!present) {
            overall += 0;
        }

        wordC++;        // Increment word count
        token = strtok(NULL, " ',.-\n");      // next token
    }

    free(sentCopy);
    return wordC > 0 ? overall / wordC : 0;       // Calculate then return average
        sentiment score
}
// duplicating a string
char *dup_str(const char *s) {
    size_t size = strlen(s) + 1;        // Calculate size of string
    char *p = malloc(size);      // Allocate memory for duplicated string
    if (p != NULL) {
        memcpy(p, s, size);      // Copy string
    }
```

```
    return p;        // Return pointer to dup string
}



// function does SA  on validation.txt
int sentiment(int argc, char *argv[]) {
    // Check CLI args
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <path_to_vader_lexicon.txt> <path_to_validation.
            txt>\n", argv[0]);
        return 1;
    }

    int lexi_size = 0;
    // Read lexicon from file specified in the CLI arg
    words *lexi = rLex(argv[1], &lexi_size);
    if (!lexi) {
        return 1; //can't read lexi means error
    }

    //open validation.txt in cli arg
    FILE *sentence_file = fopen(argv[2], "r");
    if (!sentence_file) {
        perror("Sorry, can't open file");//file opening file
        freeLex(lexi, lexi_size); // Free alloc mem
        return 1;
    }

    //format output
    int lineWidth = 80; // typical width is 80chars, just guess
    int scoreLength = 10; //how much spave we have for "score" and numbers
    int statements = lineWidth - scoreLength; // The spae that we'd use for the
       sentence part

    //headers
    printf("%-*s%*s\n", statements, "string sample", scoreLength, "score");

   //seperation line
    for (int i = 0; i < lineWidth; i++) {
        putchar('-'); // Print '-' for entire line
    }
    putchar('\n'); //next line

    char buffer[BUFFER_SIZE]; // stores line
    while (fgets(buffer, BUFFER_SIZE, sentence_file)) {
        buffer[strcspn(buffer, "\n")] = 0;

        // sent score for current sentence
        float score = do_sent_score(buffer, lexi, lexi_size);

        // right allign for score
        int sentLen = strlen(buffer);
        int spaces = statements - sentLen; // spacing depends on sent length

        // spacing never negative
```

```
        if (spaces < 0) spaces = 0;
        printf("%s%*s%.2f\n", buffer, spaces, "", score);
    }

    // closefile
    fclose(sentence_file);
    // Free allocated mem for lexi
    freeLex(lexi, lexi_size);

    return 0;
}
```

2 The implementation

The main function was meant to handle the input arguments in the terminal and determine what would be done. If there were only two input arguments in the command line terminal, being the file itself and the text file for the lexicon, "vader_lexicon.txt", then the program would read the vader lexicon using dynamic memory allocation, and store each word with it's associated data in a struct and then print each struct. If the number of command line arguments was 3, then the program would take in 2 files, the lexicon file and the text file, and then calculate the average score for each line in the file and print it. I used a function 'rLex', to open a file for reading. It then allocates an array called 'lexi' to store words with associated data. Each line from the lexicon file is then read and the data is parsed to be stored in the lexi array. The size of the array uses dynamic memory allocation to then ensure that we don't get a stack overflow in the event that the data that needs to be processed exceeds the data we memory we have available. The 'freeLex' function is what frees the dynamically allocated memory. The 'vRead' function prints each word's data after reading it from the lexicon. I implemented a 'do_sent_score function as well that makes a copy of the input sentence so as to prevent modifying the original and then tokenizes the sentence and processes each word. The words are converted to lower case and searches for the word in the lexicon, if the word is found, the c score is added to a variable called 'overall'. The counter is then appended after the conditional statement and if a word is not found, it is given a score of 0. The function then calculates the average sentiment score by diving overall by the counter. The formatting and output of the sentences (which I needed the help of chatGPT for) with their sentiment score is handled by the 'sentiment' function which takes in three input arguments, being the file itself, the lexicon text file, and the file containing the sentences. I also included a header file which handled the declaration of the functions I used as well as global variables and the custom data type for struct. The Makefile automated the build process and ensured the files were compiled using gcc and produced an objected file called mySA. The program can be compiled in the terminal, assuming you have all 3 source files and text files for the lexicon and example sentences using ./mySA [lexicon_text_file] [string_text_file]