Introdunction to Algorithms_SWE2016_42

(Professor Hyungsik Kim)

Plagiarized Source Code Detection

Assignment 02

2019313611 JiHun Kim

2023-5-14

Table of Contents

1. Introduction

This programming assignment focuses on conducting Plagiarized source code detection by comparing two source code files. Plagiarism detection is performed using a token-based approach, utilizing the Longest Common Subsequence (LCS) algorithm to calculate the plagiarism score.

The token-based approach follows the following steps:

- Remove all comments.
- Remove all parentheses, braces (or curly brackets), and brackets.
- Remove all commas and semicolons.
- Remove all whitespace characters.
- Replace all string literals with "STR LITERAL."
- Replace all numeric literals with "NUM LITERAL."
- Replace all variable names with "VAR."
- Replace all user-defined functions with "FUNC."
- Please note that the original text you provided is already in English.

2. Implementation

① Overall design and structure

This code is a program that compares two source code files and calculates the plagiarism score. Here is a brief explanation of the structure of the code:

- **Preprocessing and Initial Setup**

  It includes necessary header files and defines constants and global variables.

- **main Function**

  This is the entry point of the program where input arguments are checked and the source code files are read.

- **Source Code Preprocessing**

  It calls the function preprocessSourceCode to preprocess the given source code and

tokenize it. The tokens are stored in an array that represents the token list of the file.

- **Saving Token Lists**

  It saves the preprocessed tokens into an output file (hw2_output.txt).

- **Longest Common Subsequence (LCS) Calculation**

  It compares the token lists of the two source code files and calls the function LCS to find the longest common subsequence.

- **Plagiarism Score Calculation**

  Using the length of the longest common subsequence, it calculates the plagiarism score.

- **Saving the Plagiarism Score to the Output File**

  It saves the plagiarism score to the output file.

This program performs tokenization and the longest common subsequence algorithm to accomplish this task.

② Assignment2.c code

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0
#define POTENTIAL -1

#define MAX_SOURCE_CODE_LENGTH 102401 // Max size of file is 100KB
#define MAX_TOKEN_COUNT 102401
#define MAX_TOKEN_LENGTH 100

#define MAX_USER_DEFINED_COUNT 1000 // Assume 1000
#define MAX_USER_DEFINED_NAME_LEN 100 // Assume 100

#define DATATYPES_COUNT 20 // Initial count of data types

char
user_defined_variables[MAX_USER_DEFINED_COUNT][MAX_USER_DEFINED_NAME_LEN];
int user_defined_variables_count = 0;
```

```
char
user_defined_funcs[MAX_USER_DEFINED_COUNT][MAX_USER_DEFINED_NAME_LEN];
int user_defined_funcs_count = 0;

char datatypes[100][20] = {
        "short", "int", "unsigned", "signed", "long",
        "int8_t", "int16_t", "int32_t", "int64_t",
        "float", "double", "void", "size_t", "char", "bool",
        "struct", "typedef", "union", "enum", "FILE"
};
int datatypes_count = DATATYPES_COUNT;

int isBlank(char* str); // Is 'str' empty?
int isOperator(char c); // Is 'c' an operator?
int isSplit(char c); // Should 'c' be used as split?
int isDataType(char* word); // Is 'word' a data type?
// Return the character at index 'index' in 'word', or the next non-blank
character if it's a blank.
char next_char(char* word, int index);

int isAlphabet(char c); // Is 'c' an alphabet character?
int isNum(char* word); // Is 'word' a number (integer or float)?
int isVariable(char *word); // Is 'word' a user-defined variable?
int isFunc(char *word); // Is 'word' a user-defined function?

// Function to preprocess the source code
int preprocessSourceCode(char** token, char* source_code, int
source_code_count);

int min(int a, int b); // Returns a smaller value
int max(int a, int b); // Returns a larger value

// Function to find the Longest Common Subsequence (LCS)
int LCS(char** tokens1, int token1_count, char** tokens2, int
token2_count);
```

Main function

The main() function performs the following operations:

- It reads the contents of the file specified by argv[1] (typically original.c) character by character and stores them in the sourceCode1 string, including newline characters ('\n'). The total length of the source code is stored in the variable sourceCode1_count.
- It reads the contents of the file specified by argv[2] (typically copied.c) character by character and stores them in the sourceCode2 string, including newline characters ('\n'). The total length of the source code is stored in the variable sourceCode2_count.

- It tokenizes the input source codes into string arrays token1 and token2 by calling the preprocessSourceCode function. The number of tokens in each source code is returned and stored in the variables token_count. The obtained tokens are then written to the file hw2_output.txt.
- It uses the LCS (Longest Common Subsequence) algorithm to determine the longest common subsequence of tokens between the two source codes. The common tokens are written to hw2_output.txt, and the count of common tokens is returned and stored in the variable common_token_count.
- It calculates the plagiarism score by dividing the length of the shorter token sequence by the length of the common token sequence and writes it to hw2_output.txt.

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage:\n %s <source_code_file1>
<source_code_file2>\n", argv[0]);
        return 1;
    }

    // Read first source code file
    FILE* fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "File open error (%s)\n", argv[1]);
        return 1;
    }

    char* sourceCode1 = (char *)malloc(sizeof(char) *
MAX_SOURCE_CODE_LENGTH);
    int sourceCode1_count = 0;

    char ch;
    while((ch = fgetc(fp)) != EOF) {
        sourceCode1[sourceCode1_count++] = ch;
    }
    sourceCode1[sourceCode1_count] = '\0';
    fclose(fp);


    // Read second source code file
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "File open error (%s)\n", argv[2]);
        return 1;
    }
```

```
    char* sourceCode2 = (char *)malloc(sizeof(char) *
MAX_SOURCE_CODE_LENGTH);
    int sourceCode2_count = 0;

    while((ch = fgetc(fp)) != EOF) {
        sourceCode2[sourceCode2_count++] = ch;
    }
    sourceCode2[sourceCode2_count] = '\0';
    fclose(fp);

    /* (1) the token lists for each source code file */
    // Preprocess source code files
    char** token1 = (char **)malloc(sizeof(char *) * MAX_TOKEN_COUNT);
    for(size_t i = 0; i < MAX_TOKEN_COUNT; i++)
        token1[i] = (char *)malloc(sizeof(char) * MAX_TOKEN_LENGTH);

    char** token2 = (char **)malloc(sizeof(char *) * MAX_TOKEN_COUNT);
    for(size_t i = 0; i < MAX_SOURCE_CODE_LENGTH; i++)
        token2[i] = (char *)malloc(sizeof(char) * MAX_TOKEN_LENGTH);

    int token1_count = preprocessSourceCode(token1, sourceCode1,
sourceCode1_count);
    int token2_count = preprocessSourceCode(token2, sourceCode2,
sourceCode2_count);

    // Write token to the output file
    FILE* outputFile = fopen("hw2_output.txt", "w");

    if (outputFile == NULL) {
        fprintf(stderr, "File open error (%s)\n", "hw2_output.txt");
        return 1;
    }

    for (size_t i = 0; i < token1_count; i++) {
        fprintf(outputFile, "%s\n", token1[i]);
    }

    fprintf(outputFile, "*****\n");
    for (size_t i = 0; i < token2_count; i++) {
        fprintf(outputFile, "%s\n", token2[i]);
    }
    fprintf(outputFile, "*****\n");
    fclose(outputFile);


    /* (2) the common tokens found in the longest common subsequence
        between the token lists of the two source code files */
```

```
    int common_token_count = LCS(token1, token1_count, token2,
token2_count);


    /*  (3) the plagiarism score, represented as a percentage (rounded to
two decimal places)
         into the output file named ‘hw2 output.txt.’ */
    float plagiarism_score = (float)common_token_count / min(token1_count,
token2_count) * 100;

    // Write plagiarism score to the output file
    outputFile = fopen("hw2_output.txt", "a");

    if (outputFile == NULL) {
        fprintf(stderr, "File open error (%s)\n", "hw2_output.txt");
        return 1;
    }
    fprintf(outputFile, "%.2f", plagiarism_score);
    fclose(outputFile);


    // Memory allocate
    free(sourceCode1);
    free(sourceCode2);
    for(size_t i = 0; i < token1_count; i++)
        free(token1[i]);
    free(token1);
    for(size_t i = 0; i < token2_count; i++)
        free(token2[i]);
    free(token2);

    return 0;
}
```

isBlank function

The isBlank function determines whether a given string is empty or consists solely of whitespace characters. It returns TRUE if the string is blank (empty or contains only spaces, tabs, and newlines), and FALSE otherwise.

```
// Is 'str' empty?
int isBlank(char *str) {
    size_t length = strlen(str);

    for(size_t i = 0; i < length; i++)
        if(str[i] != ' ' && str[i] != '\t' && str[i] != '\n')
            return FALSE; // not blank string
```

```
    return TRUE; // only blank string
}
```

isOperator function

The isOperator function determines whether a given character ‘c’ is an operator. It returns TRUE if ‘c’ is an operator and FALSE otherwise. The function iterates through a pre-defined array of operator characters, checking if ‘c’ matches any of the operators in the array.

```
// Is 'c' an operator?
int isOperator(char c) {
    char operators[] = "+-*/%%=><!&|^~?:";
    for(size_t i = 0; operators[i] != '\0'; i++)
        if(operators[i] == c)
            return TRUE;
    return FALSE;
}
```

isSplit function

The isSplit function determines whether a given character ‘c’ should be used as a delimiter. It returns TRUE if ‘c’ is a delimiter character and FALSE otherwise. This function iterates through a pre-defined array of delimiter characters, cheking if ‘c’ matches any of the characters in the array.

```
// Should 'c' be used as split?
int isSplit(char c) {
    char splits[] = " ,;(){}[]\t\n";
    for(size_t i = 0; splits[i] != '\0'; i++)
        if(splits[i] == c)
            return TRUE;
    return FALSE;
}
```

isDataType function

The isDataType function checks if the given ‘word’ is a data type. This function compares 'word' with the data types defined in the datatype array and returns the index + 1 of the matching data type if found. It returns index + 1 instead of TRUE to differentiate between the default data types and user-defined types when tokenizing. There are 20 default data types defined in the array with the constant DATATYPES_COUNT. If there is no match with any data type, it returns FALSE.

```
// Is 'word' a data type?
int isDataType(char* word) {
    for(size_t i = 0; i < datatypes_count; i++)
        if(!strcmp(word, datatypes[i]))
            return i + 1;
    return FALSE;
}
```

next_char function

The next_char function returns the character at the specified 'index' in the 'word' string. If the character at the given index is a blank character (space, tab, or newline), it recursively calls itself with the next index until a non-blank character is found, and then returns that character.

It is used to distinguish between variables and user-defined functions in the future. If `next_char` returns '(' character, it indicates a function, and if it returns a different character (e.g., ' , ', ' ; ', etc.), it is treated as a variable.

```
// Return the character at index 'index' in 'word', or the next non-blank
character if it's a blank.
char next_char(char* word, int index) {
    if(word[index] == ' ' || word[index] == '\t' || word[index] == '\n')
        return next_char(word, index + 1);
    else
        return word[index];
}
```

isAlphabet function

This function checks if the given character 'c' is an alphabet character. It returns TRUE if the character is an alphabet character and FALSE otherwise.

This function is used later on to determine whether the '.' character represents the decimal point in a float number or is used as an operator.

```
// Is 'c' an alphabet character?
int isAlphabet(char c) {
    return ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));
}
```

isNum function

This function checks whether the given 'word' is a number, either an integer or a float. It iterates through each character in the word and checks if it is a digit (0-9) or a decimal point ('.'). If all the characters in the word are digits or a single decimal point, it returns TRUE indicating that the word is a number. Otherwise, it returns FALSE.

```
// Is 'word' a number (integer or float)?
int isNum(char* word) {
    for(size_t i = 0; word[i] != '\0'; i++) {
        if((word[i] >= '0' && word[i] <= '9') || word[i] == '.')
            continue;
        else
            return FALSE;
    }
    return TRUE;
}
```

isVariable function

This function checks whether the given 'word' is a user-defined variable. It compares the 'word' with the list of user-defined variables stored in the array 'user_defined_variables'. If a match is found, indicating that the 'word' is a user-defined variable, it returns TRUE. Otherwise, it returns FALSE.

```
// Is 'word' a user-defined variable?
int isVariable(char *word) {
    for(size_t i = 0; i < user_defined_variables_count; i++)
        if(!strcmp(word, user_defined_variables[i]))
            return TRUE;
    return FALSE;
}
```

Write file function

This function checks if the given 'word' is a user-defined function. It compares 'word' with the user-defined functions stored in the 'user_defined_funcs' array. If a match is found, it returns TRUE to indicate that 'word' is a user-defined function. If no match is found, it returns FALSE.

```
// Is 'word' a user-defined function?
int isFunc(char *word) {
    for(size_t i = 0; i < user_defined_funcs_count; i++)
        if(!strcmp(word, user_defined_funcs[i]))
            return TRUE;
```

```
    return FALSE;
}
```

preprocessSourceCode function

This function is a preprocessor for the given source code. It tokenizes the source code and stores the tokens in the `token` array, returning the count of tokens. The function utilizes several variables and states to process the source code.

The main functionalities are as follows:

- It removes comments, including single-line comments starting with ' // ' and multiline comments enclosed between ' /* ' and ' */ '.
- It processes the source code word by word, using delimiters, operators, and specific characters as separators.
- It identifies data types, distinguishing between predefined data types and user-defined types.
- It identifies variables and functions, using the `next_char` function to differentiate between them.
- It identifies string literals, character literals, and numeric literals.
- It processes header files and macro definitions.

After the preprocessing steps, the tokens are stored in the `token` array, and the count of processed tokens is returned.

I have added detailed explanations for each line in the code as comments.

```
// Function to preprocess the source code
int preprocessSourceCode(char** token, char* source_code, int
source_code_count) {
    int token_count = 0;

    // Buffer to store a word
    char* word = (char *)malloc(sizeof(char) * MAX_SOURCE_CODE_LENGTH);
    word[0] = '\0';
    int word_idx = 0;

    // Flags to track various states
    int in_header_file = FALSE; // Inside header file
```

```
    int in_define = FALSE;      // Inside macro

    int in_line_comment = FALSE;    // Inside single-line comment
    int in_comment = FALSE;         // Inside multiline comment

    int is_char = FALSE;    // Inside character literal (' ')
    int is_string = FALSE;  // Inside string literal (" ")

    int dot_is_operator = FALSE;    // Dot is an operator

    int after_data_type = FALSE;    // After a data type declaration

    int typeDef_data_type = FALSE;  // After a typedef data type
declaration
    int in_typeDef = FALSE;         // Inside typedef block

    int enum_data_type = FALSE;     // Inside an enum data type declaration
    int in_enum = FALSE;            // Inside enum block

    for(size_t i = 0; i < source_code_count; i++) {
        // Remove comments (//)
        if(!in_line_comment && source_code[i] == '/'
                            && source_code[i + 1] == '/') {
            in_line_comment = TRUE;
            continue;
        }
        else if(in_line_comment && source_code[i] == '\n') {
            in_line_comment = FALSE;
            continue;
        }
        else if(in_line_comment) {
            continue;
        }

        // Remove coments (/* */)
        if(!in_comment && source_code[i] == '/'
                       && source_code[i + 1] == '*') {
            in_comment = TRUE;
            continue;
        }
        else if(in_comment && source_code[i - 1] == '*'
                           && source_code[i] == '/') {
            in_comment = FALSE;
            continue;
        }
        else if(in_comment) {
            continue;
        }
```

```
/* If the current character is a delimiter that needs to be split or an
   operator (including the '.' operator) and it is not within a character
   literal, string literal, or header file, tokenize the word read so far.
   */
       if((isSplit(source_code[i]) || isOperator(source_code[i])
            || (source_code[i] == '.' && dot_is_operator))
            && !is_char && !is_string && !in_header_file) {

           if(typeDef_data_type && source_code[i] == '{')
               in_typeDef++;
           else if(typeDef_data_type && source_code[i] == '}')
               in_typeDef--;

           if(enum_data_type && source_code[i] == '{')
               in_enum++;
           else if(enum_data_type && source_code[i] == '}') {
               in_enum--;
               if(!in_enum)
                   // End of enum block
                   enum_data_type = FALSE;
           }

           if(isBlank(word)) {
               // Just whitespace character case
               ;
           }
           else if(in_enum) {
               // Inside enum, treat word as variable
               strcpy(token[token_count++], "VAR");
               strcpy(
                  user_defined_variables[user_defined_variables_count++]
                  , word);
           }
           else if(isNum(word)) {
               // Numeric literal
               strcpy(token[token_count++], "NUM_LITERAL");
           }
           else if(isDataType(word)) {
               // Word is a data type

               // default data type case
               if(isDataType(word) <= DATATYPES_COUNT)
                   strcpy(token[token_count++], word);
               // user-defined type case
               else
                   strcpy(token[token_count++], "VAR");
               after_data_type = TRUE;
```

```
            // word == typedef case
            if(!strcmp(word, "typedef")) {
                typeDef_data_type = TRUE;
            }
            // word == enum case
            else if(!strcmp(word, "enum")) {
                enum_data_type = TRUE;
            }
        }
        else if(isVariable(word)) {
            // Word is a variable
            strcpy(token[token_count++], "VAR");
        }
        else if(isFunc(word)) {
            // Word is a function
            strcpy(token[token_count++], "FUNC");
        }
        else if((after_data_type == TRUE || in_define)
                && next_char(source_code, i) != '(') {
// Word is a variable (new VAR -> add to user_defined_variables)
            strcpy(token[token_count++], "VAR");
            strcpy(
               user_defined_variables[user_defined_variables_count++]
               , word);
        }
        else if(after_data_type == TRUE || in_define) {
// Word is a function (new FUNC -> add to user_defined_funcs)
            strcpy(token[token_count++], "FUNC");
            strcpy(user_defined_funcs[user_defined_funcs_count++]
                  , word);
        }
        else if(typeDef_data_type && !in_typeDef) {
// Word is a variable (typedef new datatype -> add to datatypes)
            strcpy(token[token_count++], "VAR");
            strcpy(datatypes[datatypes_count++], word);
        }
        else {
            // Word is a generic token
            strcpy(token[token_count++], word);
        }

        if(isOperator(source_code[i])) {
            if(source_code[i] == '-' && source_code[i + 1] == '>') {
                // Special case: '->' operator
                token[token_count][0] = source_code[i];
                token[token_count][1] = source_code[i + 1];
                token[token_count++][2] = '\0';
```

```
                i++;
            }
            else {
                // Oter operators
                token[token_count][0] = source_code[i];
                token[token_count++][1] = '\0';
            }
        }
        else if(source_code[i] == '.' && dot_is_operator) {
            // Dot operator
            token[token_count][0] = source_code[i];
            token[token_count++][1] = '\0';
        }

        if(in_define && source_code[i] == '\n')
            in_define = FALSE;

/* Case 1: If a data type has been declared earlier and it is followed by
           a ';' or '{' or ')', it means that no new VAR or FUNC is
           declared after that point. */
        if((source_code[i] == ';' || source_code[i] == '{'
                || source_code[i] == ')') && after_data_type == TRUE)
            after_data_type = FALSE;
/* Case 2: If a data type has been declared before and '=' is
            encountered, the subsequent tokens are not new VAR or FUNC.
            However, it is possible for new VAR or FUNC to be declared
            later, so it is marked as POTENTIAL. */
        else if(source_code[i] == '=' && after_data_type == TRUE)
            after_data_type = POTENTIAL;
/* Case 3: If a data type has been declared earlier and it is in the
           POTENTIAL state, when a ',' is encountered, it means that new
           VAR or FUNC can occur again after that point.*/
        else if(source_code[i] == ',' && after_data_type == POTENTIAL)
            after_data_type = TRUE;

        if(source_code[i] == ';' && !in_typeDef)
            // End of a typedef block
            typeDef_data_type = FALSE;

        dot_is_operator = FALSE;

        // Initializing the word
        word_idx = 0;
        strcpy(word, "\0");
      } // end if
```

```
        else {
            if(source_code[i] == ' ' || source_code[i] == '\t'
                                     || source_code[i] == '\n')
                continue;

            if(!is_string && !strncmp(source_code + i, "#include", 8)) {
                // Preprocessor directive: #include
                strcpy(token[token_count++], "#include");

                i += 7;
                in_header_file = TRUE;
                continue;
            }
            else if(!is_string && !strncmp(source_code + i, "#define", 7))
{
                // Preprocessor directive: #define
                strcpy(token[token_count++], "#define");

                i += 6;
                in_define = TRUE;
                continue;
            }

            if(isAlphabet(source_code[i]))
         // If there is at least one alphabet character preceding it,
         // the subsequent occurrence of '.' is considered as an operator.
                dot_is_operator = TRUE;

            // The charactor is stored in the string 'word'
            word[word_idx++] = source_code[i];
            word[word_idx] = '\0';

            if(source_code[i] == '\'' && source_code[i - 1] != '\\'
                                      && !is_string) {
                if(is_char) {
                    // End of character literal
                    strcpy(token[token_count++], "STR_LITERAL");
                    word_idx = 0;
                    strcpy(word, "\0");
                    is_char = FALSE;
                }
                else
                    // Start of character literal
                    is_char = TRUE;
            }
            if(source_code[i] == '\"' && source_code[i - 1] != '\\'
                                      && !is_char) {
                if(is_string) {
```

```
                    // End of string literal
                    strcpy(token[token_count++], "STR_LITERAL");
                    word_idx = 0;
                    strcpy(word, "\0");
                    is_string = FALSE;
                }
                else
                    // Start of string literal
                    is_string = TRUE;
            }

            if(in_header_file && source_code[i] == '>') {
                // End of header file name
                strcpy(token[token_count++], word);
                word_idx = 0;
                strcpy(word, "\0");
                in_header_file = FALSE;
            }

        }
    }

    free(word);
    user_defined_variables_count = 0;
    user_defined_funcs_count = 0;
    return token_count;
}
```

min function

This function returns the smaller value between two integers ‘a’ and ‘b’.

```
// Returns a smaller value
int min(int a, int b) {
    return (a < b) ? a : b;
}
```

max function

This function returns the larger value between two integers ‘a’ and ‘b’.

```
// Returns a larger value
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

Write file function

This function calculates the Longest Common Subsequence (LCS) between two sets of tokens. It uses dynamic programming to find the LCS length and then backtracks through the dynamic programming table to extract the common tokens. The function writes the common tokens to an output file and returns the count of common tokens.

```
// Function to find the Longest Common Subsequence (LCS)
int LCS(char** tokens1, int token1_count, char** tokens2, int
token2_count) {
    /* 1. Use dynamic programming to calculate the LCS length */
    // Create a dynamic programming table
    int** dp = (int **)malloc(sizeof(int *) * (token1_count + 1));
    for(size_t i = 0; i <= token1_count; i++)
        dp[i] = (int *)malloc(sizeof(int) * (token2_count + 1));

    for(size_t i = 0; i <= token1_count; i++) {
        for(size_t j = 0; j <= token2_count; j++) {
            // Initialize the dynamic programming table
            if(i == 0 || j == 0)
                dp[i][j] = 0;

            // Using a recurrence relation to solve this problem
            else if(!strcmp(tokens1[i - 1], tokens2[j - 1]))
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);
        }
    }

    /* 2. Use the LCS algorithm to find common tokens */
    // Create an array to store common tokens
    char** common_tokens = (char **)malloc(sizeof(char *) *
MAX_TOKEN_COUNT);
    for(size_t i = 0; i < MAX_TOKEN_COUNT; i++)
        common_tokens[i] = (char *)malloc(sizeof(char) * MAX_TOKEN_LENGTH);
    int common_token_count = 0;

    // Backtrack through the dynamic programming table to find common
tokens
    for(size_t i = token1_count, j = token2_count; i > 0 && j > 0;) {
        if(!strcmp(tokens1[i - 1], tokens2[j - 1])) {
            strcpy(common_tokens[common_token_count++], tokens1[i - 1]);
            i--;
            j--;
        }
        else if(dp[i - 1][j] >= dp[i][j - 1])
```

```
            i--;
        else
            j--;
    }

    /* 3. Write common token to output file */
    // Open the output file in append mode
    FILE* outputFile = fopen("hw2_output.txt", "a");
    if (outputFile == NULL) {
        fprintf(stderr, "File open error (%s)\n", "hw2_output.txt");
        return 1;
    }

    // Write common tokens to the output file
    for(int i = common_token_count - 1; i >= 0; i--)
        fprintf(outputFile, "%s\n", common_tokens[i]);
    fprintf(outputFile, "*****\n");

    fclose(outputFile);

    // Return the count of common tokens
    return common_token_count;
}
```

3. Performance Analysis

① Time Complexity

Let's consider the following lines for time complexity analysis in terms of the length of the original.c input as 'n' and the length of the copied.c input as 'm':

- **Line for reading the file into the sourceCode string**

  Since each character is read individually, it takes time proportional to the length of the input.

  Time Complexity: O(n + m)

- **Line for tokenizing the source code in the preprocessSourceCode function**

  The function processes each character in the source code. Although there are function calls like isSplit, isOperator, isDataType for each character, they eventually involve constant time operations. The only potential non-constant factor can be the number of data types, but assuming a maximum of 100 data types, which is a reasonable

assumption within a 100KB code, the time complexity becomes.

Time Complexity: O(n + m)

- **Line for obtaining the common tokens in the LCS function**

  The size of token1 is bounded by 'n', and the size of token2 is bounded by 'm'. The process of calculating the LCS length in the function involves nested for loops, resulting in a time complexity of O(nm). Additionally, during the backtracking process to find the common tokens, the number of iterations in the for loop cannot exceed 'n' and 'm'. Therefore, the overall time complexity is as follows:

  Time Complexity: O(n * m)

- **Line for calculating the plagiarism score**

  Time Complexity: O(1)

Therefore, the time complexity of Assignment2.c can be concluded as **O(n * m)**.

③ Space complexity

Let's analyze the space complexity considering the length of the original.c as 'n' and the length of copied.c as 'm'. The following factors can be considered:

- **sourceCode**

  Dynamically allocated with a size of MAX_SOURCE_CODE_LENGTH

  Space complexity: O(1), but it can be seen as O(n + m) since it depends on the lengths of original.c and copied.c.

- **token**

  Dynamically allocated with a size of MAX_TOKEN_COUN * MAX_TOKEN_LENGTH

  Space complexity: O(1), but it can be seen as O(n + m) since it depends on the lengths of original.c and copied.c. (The maximum length of token is limited to 100.)

- **word**

  Dynamically allocated with a size of MAX_SOURCE_CODE_LENGTH

  Space complexity: O(1), but it can be seen as O(n + m) since it depends on the lengths of original.c and copied.c.

- **dp**

  Dynamically allocated with a size of (token1_count + 1) * (token2_count + 1)

  Space complexity: O(n * m)

Therefore, the overall space complexity is **O(n * m)**.