10802EE 398000 Algorithms

Homework 11. Transforming Text Files

- **Introduction**

Given two similar text files, t1a.txt and t1b.txt, one can use three editing commands: **change line, insert line and delete line** to transform one to another. In this homework, I will write a C program that takes two files as its input and output a series of commands to perform such transformation. Besides, the number of transformation commands should be as small as possible. There are 6 sets of files: t1a.txt and t1b.txt; t2a.txt and t2b.txt; t3a.txt and t3b.txt; t4a.txt and t4b.txt; t5a.txt and t5b.txt; t6a.txt and t6b.txt. I will use these 6 sets of files to verify the time complexity of my program.

In **Approach** part, I will explain how I define and get the cost of inserting a line, deleting a line, and changing from a line to the other line. After that, I will illustrate how I get the minimum cost matrix and trace it to get a series of commands. Also, I will show pseudocodes of them and analyze their time complexities and space complexities.

In **Result** part, I will show CPU time results and CPU time graph to see whether they are consistent to my analysis. Please note that the execution time will exclude both input and output time, but it will take average over at least 500 executions.

Approach

To read text a and text b into string lists X and Y, I use Dynamic_Store algorithm. The reason I use it is that we don’t know how many lines are there in the text file. To handle data without prior knowledge of its size, dynamically allocated string list should be used.

```
// Store temp (a string) into a dynamic string list A with capacity equal to size.
// Input: A[1 : size], temp, int size and index
// Output: A[index] := item.
Algorithm Dynamic_Store(A, size, index, temp)
{
    if (size = 0) then { // Initial call.
        size := 1; A := malloc(size× sizeof(typeA)); // Allocate A.
    }
    else if (index > size) then { // string list A is full. Double A.
        size := 2 × size;
        B := malloc(size × sizeof(typeA));
        for i := 1 to index−1 do {
            B[i] = malloc(length of A[i] × sizeof(typeA[i]));
            copy a string from A[i] to B[i];
        }
        free the string list A;
        A := B; // Pointer assignment.
    }
    A[index] = malloc(length of temp × sizeof(typetemp));
    copy a string from temp to A[index]; // Store into string list A.
}
```

Algorithm 1 Dynamic_Store.

After reading text a and text b using Dynamic_Store, we can get two string lists X and Y. X[i] is the $(i + 1)$th line of text a. Y[j] is the $(j + 1)$th line of text b.

Next, I will get the cost of inserting a line, deleting a line, and changing from a line to the other line. Since there is no definition of the cost in Homework 11. Instruction, I define my own.

Definitions are shown below. Please note that we want to transform text a into text b.

Def: The cost of inserting a line in text b is the number of words in a line. For example, if we want to insert a line: “I have a pen.” The cost will be 4 because there are four words: “I”, “have”, “a”, “pen.”
Definition 1 the definition of the cost of **inserting a line**.
Def: The cost of deleting a line in text a is the number of words in a line. For example, if we want to delete a line: “I have an apple.” The cost will be 4 because there are four words: “I”, “have”, “an”, “apple.”
Definition 2 the definition of the cost of **deleting a line**.
Def: The cost of changing a line in text a into the line in text b is (the number of different words sequentially in a line) * 2. For example: 1. If we want to change a line: **“I have a pen.”** into **“I have an apple.” the cost will be 4** because the first word and the second word are the same. The third word and the forth word are different. Thus, the cost is 2 * 2 = 4. 2. If we want to change a line: **“I have pen.”** into **“I have a pen.” the cost will be 4** because the first word and the second word are the same. The third word and the forth word are different. Thus, the cost is 2 * 2 = 4. Please note that either in case 1 or 2, **the cost will be less than deleting a line then inserting a line**. This makes sense since there are lots of common between two lines.
Definition 3 the definition of the cost of **changing from a line to the other line**.

To transforming string lists X and Y into C matrix (C[i][j] is the cost to change from line i in text a to line j in text b), D array (D[i] is the cost to delete line i in text a), and I array (I[j] is the cost to insert line j in text b), Algorithm getCost_C and getCost_I_and_D are shown below:

```
// get C matrix
// Input: n (the number of lines in text a), m (the number of lines in text b), X (the string list of text
//          a), Y (the string list of text b).
// Output: C[n + 1][m + 1] matrix.
1 Algorithm getCost_C(n, m, X, Y, C)
2 {
3       for i := 0 to n - 1 do {
4               for j := 0 to m - 1 do {
5                       C[i + 1][j + 1] := 0;
6                       wordx = get the word in line X[i];
7                       wordy = get the word in line Y[j];
8                       while wordx and wordy are both successfully gotten then {
9                               if wordx isn't same as wordy then C[i + 1][j + 1] := C[i + 1][j + 1] + 2;
10                      }
11                      if wordx isn't successfully gotten then {       // if Y[j] is longer than X[i]
12                              wordy = get the word in line Y[j];
13                              while wordy is successfully gotten then C[i + 1][j + 1] := C[i + 1][j + 1] + 2;
14                      }
15                      else if wordy isn't successfully gotten then { // if X[i] is longer than Y[j]
16                              wordx = get the word in line X[i];
17                              while wordx is successfully gotten then C[i + 1][j + 1] := C[i + 1][j + 1] + 2;
18                      }
19                      C[i + 1][j + 1] := C[i + 1][j + 1] + 2;
20              }
21      }
22 }
```

Algorithm 2 getCost_C.

In getCost_C, I check all combinations between lines in text a and lines in text b to calculate costs and store them in C matrix. In Line 6 to line 10, I get words from line X[i] in text a and line

Y[j] in text b simultaneously and compare whether they are same or not. If we can't get the word from line X[i] or line Y[j], it goes to line 11 to 18 to see which line is end. If line X[i] is end, then continuing getting words from Y[j] and calculating the cost, and vice versa.

The time complexity of getCost_C is O(nm) where n and m are the number of lines in text a and text b respectively. The overall space complexity of getCost_C is also O(nm)

```
// get I array or D array
// Input: n (the number of lines in the text), A (the string list of the text)
// Output: B[n + 1] array.
1 Algorithm getCost_I_and_D (n, A, B)
2 {
3       for i := 0 to n - 1 do {
4               B[i + 1] := 0;
5               word = get the word in line A[i];
6               while word is successfully gotten then B[i + 1] := B[i + 1] + 1;
7               B[i + 1] := B[i + 1] + 1;
8       }
9 }
```

Algorithm 3 getCost_I_and_D.

getCost_I_and_D calculates how many words are there in a line of the text. In line 6, I get words from line A[i] until the word cannot be successfully gotten (the end of line). Also, I calculate the cost B[i + 1] simultaneously.

The time complexity of getCost_I_and_D is O(n) where n is the number of lines in the text. The overall space complexity of getCost_I_and_D is also O(n)

How can we get the word from a line? Algorithm getword is shown in the next page. It's a simple tokenizer.

```
// get a word from the line each time (If we finish traveling the line, return 1)
// Input:, str[] (a line in the text), start (the index of str[] which is currently travelled).
// Output: word (a word in a line), end (end = 1 means the line is finished travelling).
1 Algorithm getword(word, str, start)
2 {
3       index := start;
4       i := 0;
5       end := 0;
6       while str[index] is not space, new line, or tab and index is less than the length of str then {
7             word[i] := str[index];
8             i := i + 1;
9             index := index + 1;
10      }
11      index := index + 1;
12      if index is greater then or equal to the length of str then end := 1;
13      start := index;
14      return end;
15 }
```

Algorithm 4 getword.

In line 6 to 10, I travel the str[] until there is a space, new line, or tab which is the end (or the beginning) of a word. I also record the character of the word simultaneously. In line 12, I judge whether the line is end of not.

After getting the cost of inserting a line (I array), deleting a line (D array), and changing from a line to the other line (C matrix). We can start to find the minimum cost of transforming text a into text b. The Wagner Fisher Algorithm shown in Algorithm 6.3.1, Unit 6.3, class notes of EE3980 can be applied to this problem. There are some differences: the inputs X and Y in Algorithm 6.3.1 are two strings. However, the inputs X and Y in this case are string lists. Besides, the cost D, I, and C are in the unit of character in Algorithm 6.3.1. However, the cost D, I, and C in this case are in the unit of line. The corresponding Algorithm WagnerFischer is shown below:

```
// Transform X[n] into Y[m] with minimum cost using matrix M[n, m].
// Input: int n, m, string lists X[n], Y[m], cost D[n], I[m], C[n, m]
// Output: min cost matrix M[n, m].
Algorithm WagnerFischer(n, m, X, Y, D, I, C, M)
{
    M[0,0] := 0;
    for i := 1 to n do M[i, 0] := M[i−1, 0] + D(X[i]);
    for j := 1 to m do M[0, j] := M[0, j−1] + I(Y[j]);
    for i := 1 to n do {
        for j := 1 to m do {
            if (X[i] = Y[j]) then m1 := M[i−1, j−1];
            else m1 := M[i−1, j−1] +C(X[i], Y[j]);
            m2 := M[i−1, j] +D(X[i]);
            m3 := M[i, j−1] +I(Y[j]);
            M[i, j] := min(m1, m2, m3);
        }
    } // When done, M[n, m] contains the minimum cost of the transformation
}
```

Algorithm 5 WagnerFischer.

After using Algorithm WagnerFischer, we can get the matrix M where M[n, m] contains the minimum cost of the transformation.

The time complexity of WagnerFischer is O(nm) where n and m are the number of lines in text a and text b respectively. The overall space complexity of WagnerFischer is also O(nm).

After WagnerFischer algorithm, the following algorithm traces the M matrix to generate the transformation sequence. Please note that array T has the transformation sequence but is i**n reverse order.**

```
// Trace the matrix M[n, m] to find the transformation operations.
// Input: int n, m, traceSize (the size (not capacity) of T), cost D[n], I[m], C[n, m] and M[n, m]
// Output: T[n + m][3] transformation, change (the number of changes).
Algorithm Trace(n, m, traceSize, M, D, I, C, T)
{
    i := n; j := m; k := 0; change := 0;
    while (i > 0 and j > 0) do {
        if (M[i, j] = M[i−1, j−1] + C(X[i], Y[j])) then {
            Record transformation information in T;
            i := i−1; j := j−1; k := k + 1; change :=change + 1; // Change X[i] to Y[j].
        }
        else if (M[i,j] = M[i,j−1] + I(Y[j])) then { // Add Y[j].
            Record transformation information in T;
            j := j−1; k := k + 1; change := change + 1;
        }
        else if (M[i,j] = M[i−1,j] + D(X[i])) then { // Delete X[i].
            Record transformation information in T;
            i := i−1; k := k + 1; change := change + 1;
        }
        else { // No changes.
            Record transformation information in T;
            i := i−1; j := j−1; k := k + 1;
        }
    } // Array T has the transformation sequence but is in reverse order.
    while (i > 0) do {
        Record transformation information in T;
        i := i−1; k := k + 1; change := change + 1;
    }
    while (j > 0) do {
        Record transformation information in T;
        j := j−1; k := k + 1; change := change + 1;
    }
    traceSize = k;
    return change;
}
```

Algorithm 6 Trace.

After using Trace, we can get the T array which contains transformation information in it. Thus, we can travel it reversely from traceSize – 1 to 0 to get the corresponding sequence of commands.

The time complexity of Trace is O(n + m) where n and m are the number of lines in text a and text b respectively. The overall space complexity of Trace is O(nm) for C matrix.

Algorithm	Overall Space Complexity	Time Complexity
getCost_C	O(nm)	O(nm)
getCost_I_and_D	O(n)	O(n)
WagnerFischer	O(nm)	O(nm)
Trace	O(nm)	O(n + m)

Table 1 space and time complexities. Please note that n is the number of lines in text a, m is the number of lines in text b, except for n in getCost_I_and_D, which indicates the number of lines for the input text.

- **Results**

	t1a.txt t1b.txt	t2a.txt t2b.txt	t3a.txt t3b.txt	t4a.txt t4b.txt	t5a.txt t5b.txt	t6a.txt t6b.txt
Number of lines	9	28	46	68	125	205
Number of changes	3	6	9	12	15	18
CPU Time	3.58963e-06 seconds	2.86298e-05 seconds	7.25436e-05 seconds	1.32788e-04 seconds	3.91212e-04 seconds	9.50426e-04 seconds

Table 2 results table.

The CPU time graph is shown in the next page. The CPU time only consists of WagnerFischer and Trace because we may exclude both input and output time. getCost_C and getCost_I_and_D are also included in the input time, so I ignore them.

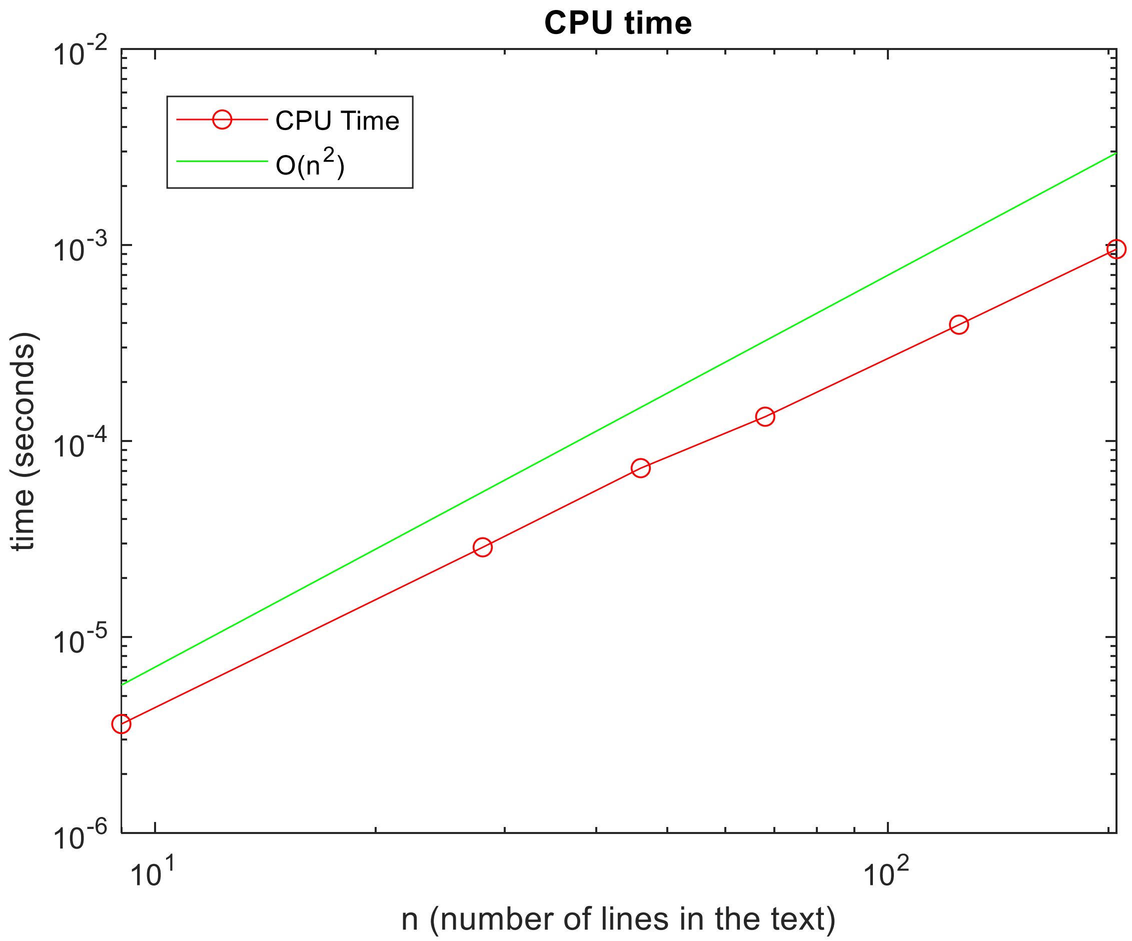

Figure 1 CPU time.

Each pair of input texts has the same number of lines. Thus, the time complexity in **Table 1** will become $O(n^2)$ for O(nm) and O(2n) for O(n + m). Since $O(n^2)$ is larger than O(2n), the time complexity will be dominated by WagnerFischer Algorithm. That is, the time complexity is $O(n^2)$ which is consistent with **Figure 1**.