EE3980 Algorithms

Hw11 Transforming Text Files

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing, implementing, and observing 1 algorithm. The goal of the algorithm is to get transform one text file to another using the 3 ways, change, insert and delete, and the output of them will be the steps for transformation.

During the analysis process, I will first introduce why and how dynamic programming can be applied to this task. Then, I will be using counting method to calculate the time complexities of the algorithm. Finally, I will calculate their space complexity for the total spaces used by the algorithm.

The implementation of the algorithm on C code will find the transformation steps for 5 text files. Furthermore, I will evaluate their CPU times.

Analysis:

1. **Dynamic programming:**

   Dynamic programming is to simplifying a complicated problem by **breaking it down into simpler sub-problems** in a recursive manner. Since there are several ways to transform a text file into another, we can break any text file

transformation in a recursive manner. For example, for the string “alogirten”, there are several ways to transform it into “Algorithm” by deleting, inserting or changing characters. Therefore, dynamic programming is a perfect way to implement this task.

2. Variables set:

N, M	Number of lines in target text file X, Y.
X[N], Y[N]	Target files, transform X into Y.
S[N x N]	Solution set
T	Transformation steps list

3. Dynamic programming, WagnerFisher(N, M, X, Y, M):

The text transforming problem is to transform texts in file X into those in Y using the follow editing operations with corresponding cost to find the sequence of operations that minimize the cost.

i. Delete line X[i] from X with cost D(X[i]).

ii. Insert line X[i] from X with cost I(X[i]).

iii. Change line X[i] from X into Y[j] from Y with cost C(X[i], Y[j]).

In this case, I set those costs to 0.

A corresponding dynamic programming algorithm is as the following:

```
1.  // Transform X[n] into Y[m] with minimum cost using matrix S[n, m].
2.  // Input: int n, m, files X[n], Y[m]
```

```
3.  // Output: min cost matrix S[n, m].
4.  Algorithm WagnerFischer(n, m, X, Y, S)
5.  {
6.      S[0, 0] := 0 ;
7.      for i := 1 to n do M[i, 0] := S[i – 1, 0] + 1 ;
8.      for j := 1 to m do M[0, j] := S[0, j – 1] + 1 ;
9.      for i := 1 to n do {
10.         for j := 1 to m do {
11.             if (X[i ] = Y[j ]) then m1 := S[i – 1, j – 1] ;
12.             else {
13.                 m1 := S[i – 1, j – 1] + 1 ;
14.                 m2 := S[i – 1, j ] + 1 ;
15.                 m3 := S[i, j – 1] + 1 ;
16.                 S[i, j ] := min(m1, m2, m3) ;
17.             }
18.         }
19.     }
20. }
```

WagnerFisher(N, M, X, Y, S) generates a table of transforming steps from the first line to the last. It goes through all possible changing operations on line 13, all possible inserting operations on line 14 and all possible deleting operations on line 15.

For WagnerFisher(N, M, X, Y, S), loops on 7 and 8 execute n times each. Loop on 9 execute n times. The inner loop on 10 ~ 18 executes N * M times to generate a table of transforming steps. Thus, the total **time complexity is O(N * M)**. **The space complexity is O(N * M)** for S[N * M] array.

4. **Trace(N, M, X, Y, S, T) printing transformation steps:**

The following algorithm can be used to print the steps for transformation:

```
// Trace the matrix S[n, m ] to find the transformation operations.
// Input: int n, m, cost D[n ], I[m ], C[n, m ] and S[n, m ]
// Output: T[n + m ] transformation.
Algorithm Trace(N, M, X, Y, S, T)
{
    i := n ; j := m ; k := 0 ;
    while (i > 0 and j > 0) do {
        if (i > 0 and j > 0 and (S[i, j] = S[i – 1, j – 1] + 1)) then {
            T[k] := 'C' ; i := i – 1 ; j := j – 1 ; k := k + 1 ;
        }
        else if (j = 0 or (S[i, j ] = S[i, j – 1] + 1 )) {
            T[k] := 'I' ; j := j – 1 ; k := k + 1 ;
        }
        else if (i = 0 or (M[i, j ] = M[i – 1, j ] + 1 )) then {
            T[k] := 'D' ; i := i – 1 ; k := k + 1 ;
        }
        else { // No changes.
            T[k] := ' – ' ; i := i – 1 ; j := j – 1 ; k := k + 1 ;
        }
    }
    Print T;
}
```

5. Time & Space:

	WagnerFisher()
Time complexity	**O(N * M)**
Space complexity	**O(N * M)**

Implementation:

1. Workflow:

In order to get measure the CPU time more accurately, the workflow is as followed:

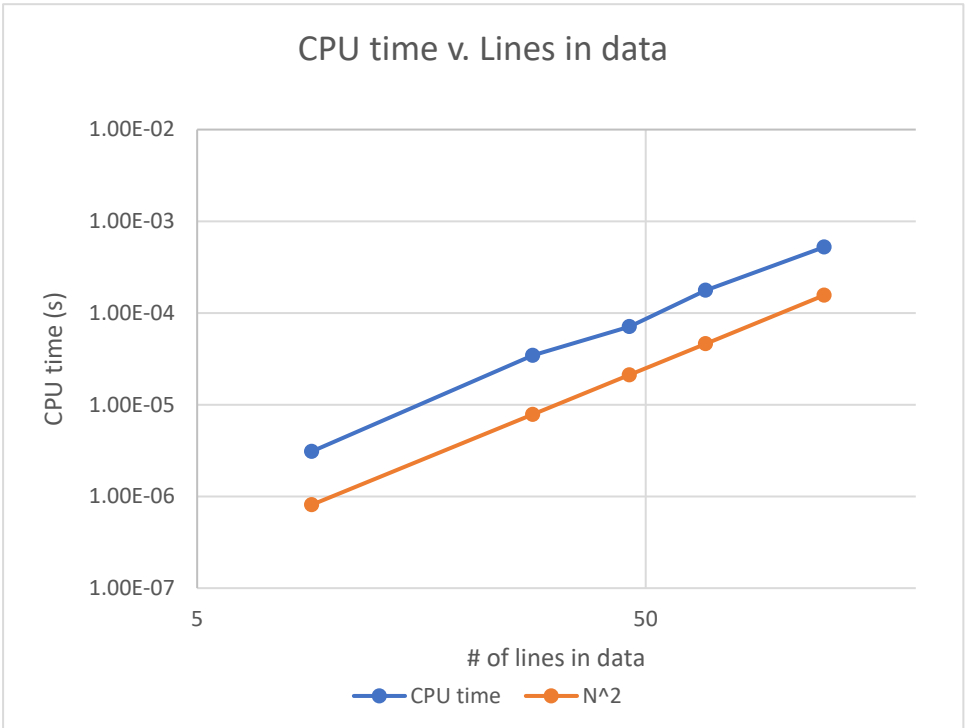
```
Algorithm workflow(X, Y)
{
```

```
3.      T = GetTime();
4.      for i := 1 to 500 do {
5.          WagnerFisher(N, M, X, Y, S);
6.      }
7.      T = (GetTime() - T) / 500;
8.  }
```

2. Result:

	Lines	Changes	CPU time
File 1	**9**	**3**	**3.09E-06**
File 2	**28**	**6**	**3.45E-05**
File 3	**46**	**9**	**7.11E-05**
File 4	**68**	**12**	**1.77E-04**
File 5	**125**	**15**	**5.23E-04**

Observation:



In this case, the text files X and Y has the same amount of text lines, therefore M is equal to N. We can find that the trend of CPU times has the same slope with that of N^2. And this is same as I expected, which is O(N * M).

Conclusions:

1. Text transforming problem can be implemented by dynamic programming.
2. Time and space complexities of *getCoin()*:

	WagnerFisher()
Time complexity	**O(N * M)**
Space complexity	**O(N * M)**

3. The implemented time complexity is O(N * M).