EE3980 Algorithms

HW9 Comparing Files

104061212 馮立俞

2018/5/11

Introduction

Given two files, there are three available operations in this assignment to perform on one file, such that after a series of editing, the content of the first file is the same as the second file. The three available options are inserting a new line, delete an existing line, and replace a line's content with a new line. The implementation is the same as `diff` command in Unix System, which is used to locate the differences in the two input files.

Approach

Wagner-Fischer Algorithm

Wagner–Fischer algorithm is a dynamic programming algorithm that computes the edit distance between two strings of characters. Suppose the input files, `s` and `t` have `m` and `n` lines, respectively.

```
1. int EditDistance(char s[1..m], char t[1..n])
// For all i and j, d[i,j] will hold the Levenshtein distance between  the
first i characters of s and the first j characters of t.
//Note that d has (m+1) x (n+1) values.
2.     let d be a 2 - d array of int with dimensions[0..m, 0..n]
3. for i in [0..m] d[i, 0]← i
4.
   // the distance of any first string to an empty second string
5. for j in [0..n] d[0, j]← j
   // the distance of any second string to an empty first string
6. for j in [1..n]
7.   for i in [1..m]
8.     if s[i] = t[j] then //element matched
9.       d[i, j]← d[i - 1, j - 1] // no operation required
10.    else
11.      d[i, j]← minimum of(d[i - 1, j] + 1, // a deletion
12.                          d[i, j - 1] + 1, // an insertion
13.                         d[i - 1, j - 1] + 1) // a substitution
14.return d[m, n]
```

The Wagner-Fischer algorithm keeps a matrix (`d[m+1][n+1]` in the above pseudo-code) that each `d[i]][j]` computes how much how much operations needed to transform the first `i` lines of file1(`s` in above) into the first `j` lines of file2(`t` in above).

As we have three operations available, and when the lines are identical, no edit is required. When we're filling `d[i][j]`, we'll decide whether edit is necessary, If the answer is positive, then compare which one of the three operations takes fewer overall steps (line 6-13 in above pseudo-code).

Suppose comparing two lines has $O(1)$ time complexity, the two nested loops would result in $O(mn)$ time complexity. Keeping `d[][]` matrix requires $O(mn)$ in space complexity as well.

Trace

After filling the matrix `d`, we can know the total editing operations required at `d[m][n]`. And to show what each edit step do, we trace back from `d[m][n]`. Check if editing occurred (`d[i][j] != d[i-1][j-1]`), find the minimal value among its upper, left and upper-left neighbors once an edit is found. The process is basically the same as filling `d[i][j]`. The editing process is in reversed order nonetheless; therefore, I stored them in a stack(FILO) to print the correct order. The implementation of tracing the edit process is shown in below pseudo-code.

```
Algorithm Trace( d[][], len1, len2)
     i = len1; j = len2; k = 0; clear stack;
1.  while (i > 0 || j > 0) {
2.      if (i > 0 && j > 0) {
3.          if ((d[i][j]) == (d[i - 1][j - 1])) { // no edit
4.              i--;
5.              j--;
6.          } else if (d[i][j] == d[i - 1][j - 1] + 1) { //change f1[i] with f2[j]
7.              stack[k].c = 'C';
8.              stack[k].index1 = i;
9.              stack[k].index2 = j;
10.             i--;  j--;   k++;
11.         }
12.     } else if (i == 0 or d[i][j - 1] is minimum) {
13.         stack[k].c = 'I'; //insert
14.         stack[k].index1 = i + 1;
15.         stack[k].index2 = j;
16.         j--;   k++;
17.     } else {
18.         stack[k].c = 'D'; //delete
19.         stack[k].index1 = i;
20.         i--;   k++;
21.     }
22. }
23. return Stack
```

In the worst case we can only move up and left, the complexity of Trace is $O(m+n)$. And in best case we go along the diagonal, which has $O(\max(m,n))$ time complexity. The two cases are on the same order.

Thus the over complexity is dominated by filling `d[][]`, i.e. $O(mn)$.

Results and analysis

In the given test cases, the file3~file5 can give us quite a clear view of the time complexity of Wagner-Fischer algorithm. From file3 to file5, the size of task (number of lines) doubles, and the execution time is four times longer. Actual execution time is shown below.

Table. Execution Time of Wagner-Fischer Algorithm

Task size (lines)	2561	5121	10241
Execution time (sec)	0.254	1.020	4.214

Observations and Conclusion

1. Running diff in Unix system is way faster than my homemade version. Less than 1 second is used to process 10241 lines.

2. The `Trace()` function should perform similar comparison on three possible editing operation to find out the previous step.